FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Jogos Digitais

André Luiz Donizete Sanches
Leandro Correia de Oliveira
Patrícia Tieme Miyazima

LUMENIDE

Americana, SP
2017

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Jogos Digitais

André Luiz Donizete Sanches
Leandro Correia de Oliveira
Patrícia Tieme Miyazima

LUMENIDE

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Jogos Digitais, sob a orientação do Professor Me. Bruno Daniel.

Área de concentração: Desenvolvimento de Jogos.

Americana, SP
2017

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

S19L SANCHES, André Luiz Donizete

Lumenide. / André Luiz Donizete Sanches, Leandro Corrêa de Oliveira, Patrícia Tieme Miyazima. – Americana: 2017.

136f.

Monografia (Curso de Tecnologia em Jogos Digitais) - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Ms. Bruno Daniel

1. Jogos eletrônicos I. OLIVEIRA, Leandro Corrêa de II. MIYAZIMA, Patrícia Tieme III. DANIEL, Bruno IV. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU: 681.6

André Luiz Donizete Sanches
Leandro Correia de Oliveira
Patrícia Tieme Miyazima

LUMENIDE

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Jogos Digitais, pelo CEETEPS/Faculdade de Tecnologia – Fatec/ Americana.
Área de concentração: Desenvolvimento de Jogos.

Americana, 29 de junho de 2017.

Banca Examinadora:

Bruno Daniel (Presidente)
Mestre
Fatec Americana

Gustavo Carvalho Gomes de Abreu (Membro)
Especialista
Fatec Americana

Benedito Aparecido Cruz (Membro)
Graduado
Fatec Americana

AGRADECIMENTOS

Agradeçemos à Fatec Americana e todo seu corpo docente, além da direção e a administração que trabalha incansavelmente para que nós, alunos, possamos contar com um ensino de extrema qualidade.

Também agradecemos ao nosso orientador, Bruno Daniel, pela paciência, dedicação e ensinamentos que possibilitaram que nós realizassemos este trabalho. E ao professor Gustavo Gomes, pelo suporte e apoio ao projeto.

André: Agradeço à minha esposa Mariane pela paciência e por estar sempre do meu lado em todos os momentos da vida. Agradeço também à minha família e amigos pelo apoio, em especial ao Leandro e Patrícia pelo excelente trabalho. Além disso, agradeço também a minha sogra Lucy pela revisão de todo o texto deste documento.

Leandro: Agradeço à minha esposa pela paciência e apoio. Agradeço também à minha família e amigos que me motivaram e em especial ao André e Patrícia pelo excelente trabalho.

Patricia: Agradeço aos meus pais, Atsumi e Pedro Miyazima, e ao meu irmão Thiago Miyazima pelo apoio durante toda a minha vida. Agradeço ao meu companheiro Levi Tourinho pelo apoio, compreensão e auxílio durante os momentos bons ou difíceis. Agradeço aos meus amigos pela dedicação e motivação e em especial ao André Sanches e Leandro Correia pelo excelente trabalho.

DEDICATÓRIA

A todos os professores do curso de Jogos Digitais da Fatec Americana, que foram tão importantes na nossa vida acadêmica e no desenvolvimento deste trabalho.

RESUMO

O estudo sobre Jogos Digitais é algo relativamente novo no meio acadêmico, havendo então uma falta de estudos sobre a compreensão no todo sobre o processo de criação de jogos, tanto a nível teórico quanto a nível prático. Como o desenvolvimento de um jogo envolve várias áreas e necessita de conhecimentos teóricos diversos para a real aplicação prática, este texto propõe documentar todo o processo, desde o desenvolvimento do conceito até a publicação, explicitando todas as ferramentas e metodologias utilizadas, assim como os resultados atingidos, auxiliando aqueles que tenham interesse no desenvolvimento de jogos digitais.

O trabalho desenvolvido visa a criação de um jogo de forma ágil e produtiva. Houve o desenvolvimento de um conceito, que juntamente com uma análise de mercado e de público-alvo, deu forma ao que seria o jogo e como sua produção deveria ser executada, considerando o tempo como principal fator limitante. Para isso, foram utilizados vários conhecimentos e ferramentas no desenvolvimento das metodologias e na implementação.

O produto final desenvolvido é um jogo *puzzle* casual, para dispositivos mobile com o sistema operacional Android. O jogo conta com uma arte em estilo cartoon, utilizando uma câmera de visão ¾ similar a jogos como “Zelda: A Link to the Past” (Nintendo, 1991) e uma história baseada em mitologia egípcia.

A partir da realização do trabalho foi possível expandir conhecimentos e desenvolver a capacidade de gestão de projeto adequada para que todas as tarefas fossem cumpridas e o jogo final fosse completo e divertido. A experiência de desenvolvimento de um jogo completo foi bastante enriquecedora do ponto de vista de formação profissional na área de Jogos Digitais.

Palavras Chave: Jogos eletrônicos; *Unity; Mobile.*

ABSTRACT

Digital games as a study subject is something relatively new in the academic, having as such a lack of published studies about the comprehension of the creative means as a whole, both in the theoretical and practical levels. Because the development of a game encompasses many different areas and requires a diverse knowledge set for practical application, this text proposes itself to document the entire process, from concept development to the game's publish, making explicit all of the tools and methodologies utilized, as well as the results obtained, helping those who are interested in game development.

The developed project aim at the creation of a game in an agile and productive way. There was a concept development, which aligned with a market and target audience analysis, gave shape to what would the game be and how its production should be done, considering time as a limiting factor. For that purpose, many knowledges and tools were used at the methodology development and implementation.

The final product is a casual puzzle game, made for mobile Android devices. The game has a cartoon art style, utilizing a ¾ view similar to games like "Zelda: A Link to the Past" (Nintendo, 1991) and a story based on egyptian mythology.

Throughout the project development, it was possible to expand knowledge and develop adequate management capabilities in order to achieve completion in all required tasks and to make the final game complete and fun. The experience of development of a full game was very enriching through the point of view of a professional qualification in the area of digital games.

Keywords: *Electronic games; Unity; Mobile*.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

GLOSSÁRIO

APK: Do inglês *Android Package*, é uma forma de compactação de aplicativo para *Android*, ou seja, é o arquivo já compilado e pronto para ser instalado.

Game Object: São objetos que pode representar um personagem, adereços e cenários.

GB: *Gigabyte*, unidade de medida computacional.

GDD: *Game Design Document*, documento de referencia para o *design* de jogos.

HUD: *Heads-up display*, elementos na tela.

Inspector: Componente do Unity onde é apresentado todas as propriedades de um objeto selecionado.

JSON: *JavaScript Object Notation*.

Line Renderer: Componente do Unity que desenha uma linha reta.

Mobile: Dispositivo móvel ou *SmartPhone*. Telefone celular com tecnologias avançadas, semelhante à um computador.

NPC: *Non-playable character*, personagem de jogo não jogável.

PDCA: Do inglês, *Plan*, *Do*, *Check* e *Action*. Planejar, executar, analisar e agir.

Pixels: O pixel é a menor unidade de uma imagem digital.

Plugin: Programa de computador usado para adicionar funções a outros programas maiores, provendo alguma funcionalidade especial ou muito específica.

Prefab: É um tipo de Game Object com propriedades e componentes já configurados em cena, que podem ser armazenados e reutilizados.

Script: Texto com uma série de instruções escritas para serem seguidas.

Sprite: É um objeto gráfico bi ou tridimensional que se move numa tela sem deixar traços de sua passagem.

Tiles: Um pedaço de um Sprite, geralmente quadrado.

UI: *User Interface*.

UX: *User Experience*.

1. INTRODUÇÃO

O nosso fascínio pelos jogos começou na infância. A sensação da primeira descoberta de um item mágico na caverna mais profunda, a adrenalina de uma corrida de carros, seja em casa, com os amigos, ou em outro cenário, foi um certo jogo, diferente para cada um, que nos motivou até hoje. O que realmente importa é que jogos marcaram nossas vidas, nos modelaram através do tempo, e nos fizeram buscar por algo além de jogar: nos motivaram a querer fazer outras pessoas experimentarem essa sensação de jogar.

Durante o curso de jogos digitais, várias ideias surgiram, algumas melhores, outras piores, mas uma sempre ficou guardada: Um jogo de *puzzle* baseado em espelhos, com *dungeons* estilizadas como em jogos da série *The Legend of Zelda (Nintendo)*. A ideia ficou guardada durante meses e quando estudamos a matéria de Física Aplicada aos Jogos Digitais, aprendemos vários conceitos na teoria e na prática e percebemos que seria possível criar um jogo do gênero. Mas algo ser desejado e possível não a transforma em realidade.

Criar um jogo somente tornou-se realidade através dos estudos e práticas oferecidas pela **Faculdade de Tecnologia de Americana - FATEC - AM**, que consolidou uma base de formação, e através da nova possibilidade de Trabalho de Conclusão de Curso - TCC, permitiria experimentar todo esse processo de criação de um jogo, desde seu planejamento até sua publicação, passando por diversos processos durante o seu desenvolvimento.

O jogo deveria ser desenvolvido em quatro meses, portanto a escolha foi um *puzzle* para *mobile* com mecânicas simples e bem implementadas, uma arte *cartoon* bem apresentada e adequada, uma história que apoiasse o jogo e um *level design* bem estruturado, o que criaria um jogo divertido e casual para o público alvo escolhido.

Todo o processo de motivação e escolhas para o desenvolvimento do jogo foi complexo e será detalhado nos próximos capítulos.

1.1. Da motivação à criação do conceito

Tudo começou com um *brainstorm* no início do curso. O grupo estava empolgado discutindo ideias, várias foram anotadas e, após esse primeiro momento de euforia, veio a frustração: ao se perceber que, como recém chegados à faculdade, tínhamos muitos sonhos e pouco conhecimento para torná-los realidade.

Uma das grandes barreiras no desenvolvimento de jogos foi aprender a lidar com as frustrações e não desanimar, pois muitas ideias surgem e, por mais que pareçam geniais, uma ideia boa por si só não se sustenta; é necessário analisar outros fatores como: mercado, público-alvo, plataforma, formas de desenvolvimento, recursos disponíveis e outros. O contrário também se aplica, não basta ter um vasto conhecimento dos processos de criação de um jogo, se a essência do jogo for ruim. Em ambos os casos, o jogo final tem grandes chances de falhar ou sequer chegar a ser produzido.

Apesar das dificuldades, o grupo não desanimou com o passar dos dias e depois de todo estudo e esforço, foi possível chegar a esse grande momento, que foi criar o jogo Lumenide.

Lumenide é um jogo *puzzle* casual de espelhos para *mobile* com câmera isométrica. Seu *gameplay* consiste em envolver o jogador para resolver os *puzzle*s de espelhos propostos, que tem como objetivo redirecionar a luz do Sol até o receptor, que, quando ativado pela luz, abre a porta para o próximo nível. Cada conjunto de fases tem um chefe de nível final, que o jogador deverá derrotar usando o mesmo conceito de redirecionar a luz.

A diversão do jogo envolve descobrir a resolução dos *puzzle*s de espelhos e procurar pelos elementos escondidos, que são portas secretas que possuem coletáveis de história.

Lumenide é um jogo casual que o usuário pode jogar em qualquer lugar, desde que tenha um aparelho com sistema *Android*, não exigindo *hardware* de alta performance.

1.2. Mercado para Lumenide

Uma vez que o conceito do jogo estava criado, os próximos passos foram: discutir o mercado, a plataforma, a forma de publicação, o público alvo e os recursos disponíveis. Todos esses pontos são diretamente ligados e se influenciam.

O recurso disponível crucial e limitante foi o tempo; foi ele que influenciou praticamente todas as escolhas, desde a plataforma até quais mecânicas deveriam realmente estar presentes no jogo. O segundo ponto considerado foi a forma de publicação. Deveria ser escolhida uma plataforma que permitisse fácil publicação do jogo, que é requisito mínimo para o TCC ser aprovado. O terceiro recurso considerado foi o conhecimento, utilizamos ferramentas com muita documentação *online*, mas mesmo assim, a cada nova mecânica surgiam novas dúvidas, o que requer mais tempo para implementar. Esses três pontos principais guiaram o planejamento do projeto.

1.2.1. Tempo

O tempo de duração de um projeto de jogo varia muito, a depender do tamanho da equipe, do orçamento, da plataforma, da complexidade e outros.

De acordo com o livro **Introdução ao desenvolvimento de games** (RABIN, 2011), o tempo de desenvolvimento varia de acordo com o tipo de plataforma:

- Telefone móvel (*Smartphone*): 4-6 meses;
- *PlayStation* 3 ou *Xbox* 360: 18-24 meses;
- Jogos licenciados de filme ou franquia anual de esportes: 10-14 meses;
- Nintendo Wii ou PC: 12-18 meses.

Tomando os dados como base e considerando que o tempo disponível para se desenvolver o jogo Lumenide seria de três a quatro meses e possuindo uma equipe de três pessoas, a melhor plataforma seria para telefone móvel.

1.2.2. Plataforma

Segundo Jeannie Novak (2010), há diversos tipos de plataformas: fliperama, console, computador, portáteis e jogos de mesa.

Cada plataforma possui suas características e deve ter os jogos desenvolvidos de acordo. O tipo de plataforma impacta em todo projeto, sendo um fator crucial que foi discutido e decidido na fase de planejamento.

A plataforma 'portáteis' abrange desde consoles portáteis, como *Nintendo 3DS,* até *smartphones* como *iPhone* ou *Samsung Galaxy*; deve-se lidar com o tamanho reduzido e os diferentes formatos de tela e quais seriam os *inputs* e *outputs* do jogador.

Dentre as opções de portáteis, a melhor opção seria desenvolver para dispositivos móveis como *smartphones* e *tablets*, sendo dispositivos mais acessíveis, atingindo um público maior.

1.2.3. Forma de publicação

Após concluir que a plataforma escolhida seria para dispositivos móveis, era necessário escolher para qual sistema operacional o jogo estaria disponível.

Analisando rapidamente como deveria ser feito a publicação para *iOS*, constatou-se que seria necessário comprar o *iOS Developer Program* ao preço de 99 dólares e válido por um ano e possuir um computador com sistema *Mac OS*. Conclui-se que seria muito custoso produzir o jogo para o sistema operacional *iOS*.

A outra opção viável de sistema operacional seria o *Android*, que tem um menor custo para se publicar, sendo de 25 dólares e pago somente uma vez.

Além dessa, existem também o serviço da *Amazon*, onde é possível publicar jogos para *Android* e *FireOS*, sistema da *Amazon* que comporta seus tablets e TVs. Esse serviço é gratuito, mas pouco conhecido no mercado brasileiro e por isso descartado.

Outros sistemas operacionais menores foram também desconsiderados e a forma de publicação escolhida foi para o sistema operacional *Android*.

1.2.4. Público alvo

De acordo com a pesquisa **Game Brasil 2017** (Sioux, 2017), a idade de quem joga está entre 25 e 34 anos. Os gêneros apontados como mais apreciados são os de estratégia, de aventura e de ação.

Com relação ao que os atrai em um jogo, mais da metade respondeu que gosta que o jogo “tenha diversas fases, para ir passando de nível”. Outros aspectos frequentemente considerados importantes incluem “Que tenha de pensar, definir uma estratégia”, “Que seja desafiador” e “Que seja ‘leve’, para distrair/passar o tempo”.

A plataforma mais utilizada é o celular, escolhido por 77,9% dos entrevistados, seguido de 66,4% de jogadores que jogam em um computador. A plataforma preferida também foi apontada como o celular, sendo escolhida por 37,6% dos entrevistados. Dentre os que jogam no celular, o local onde mais jogam é em casa, seguido de no trânsito e no trabalho.

O perfil dos jogadores é composto por cada vez mais mulheres, que já compõem mais de metade do público.

Finalmente, mais da metade dos jogadores entrevistados se consideraram jogadores casuais, onde apenas 6,1% dos jogadores se identificaram como jogadores *hardcore*.

Com essas informações em mente, Lumenide foi concebido como um jogo casual de quebra-cabeça organizado em fases curtas e leves, que forçam o jogador a pensar, mas também permitem que o mesmo encerre ou retome o jogo de onde parou sem maiores penalizações.

1.2.5. Mercado

O mercado de jogos para dispositivos móveis vem crescendo. De acordo com a mesma pesquisa citada em “1.2.4. Público alvo”, o mercado *mobile* é principalmente ocupado por usuários do sistema operacional *Android*. Também foi constatado que 93,6% utilizam o celular para jogos. Os tipos de aplicativo mais baixados são aplicativos

de jogos ou entretenimento, com 29,3% dos entrevistados tendo admitido que baixavam jogos para o celular toda semana.

Quanto ao sistema de monetização, a maioria dos jogadores baixam apenas jogos gratuitos, grande parcela por se sentirem inseguros se posteriormente não gostarem do jogo, ou então por acharem os jogos caros. A maioria dos entrevistados, em um total de 76,6% destes, aceitam que os jogos que baixem tenham anúncios, porém 83,4% acreditam que os anúncios interferem negativamente com a experiência do jogo.

Há muitas opções de jogos disponíveis para *mobile* e com o crescente mercado, Lumenide precisava se destacar. Logo no início do projeto, foi realizada uma análise competitiva para verificar se a ideia do jogo seria viável ao mercado estudado. Os resultados para os jogos da Figura 1 são mostrados a seguir.

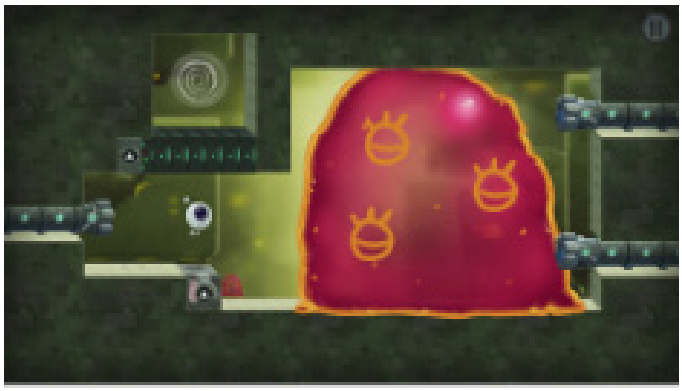
- **Espelhos e reflexos** é um jogo *puzzle* de espelhos que tem como objetivo direcionar um feixe de luz ou mais, de um ponto A a um ponto B respectivo ao feixe. É um jogo com HUD simples, porém não é visualmente agradável com sua paleta de cores.
    - Tela em *portrait*.
    - Possui tutorial, mapa de fases e *feedback* sonoro e visual simples.
    - Não possui som de fundo, história ou personagem.
    - Interação através de toque e arraste de dedo no *touchscreen*.
    - Disponível para *iOS* e *Android*.
    - Monetização: primeiras fases gratuitas e últimas fases pagas.
    - Diferenciais: grande quantidade de fases total e botão para *link* direto para a *Google Play Store* e para redes sociais.
- **Laserbreak Lite** é um jogo *puzzle* de espelhos que tem como objetivo direcionar um feixe de luz do ponto inicial ao ponto final. Possui HUD simples e poucas animações.
    - Tela em *landscape*.
    - Possui tutorial simples e *feedback* sonoro, visual simples e som de fundo somente na tela inicial.
    - Não possui som de fundo, história ou personagem.

- Interação através de toque e arraste de dedo no *touchscreen*.
- Disponível para *iOS* e *Android*.
- Monetização: propaganda em vídeo entre as fases.
- Diferenciais: mudar ângulo de saída do feixe de luz e possuir outros objetos de interação no cenário, como caixas que bloqueiam o feixe.

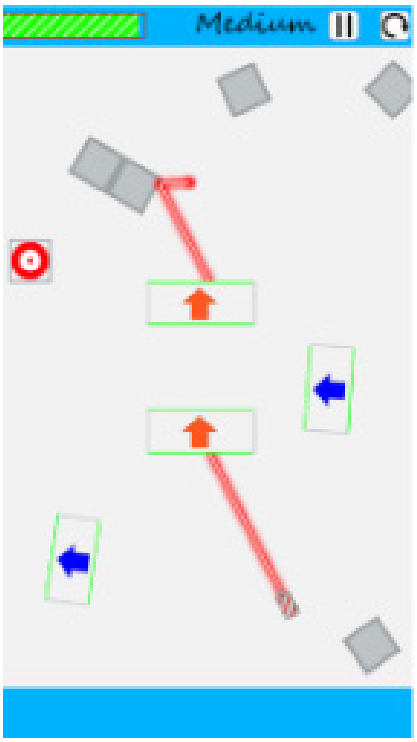
- **Laser Pop** é um jogo *puzzle* de espelhos que tem como objetivo direcionar o ângulo de saída de um raio para que ele atinja certo ponto. Possui HUD muito simples e não personalizada.
  - Tela em *portrait*.
  - Possui mapa de fases e *feedback* sonoro e visual simples.
  - Não possui tutorial, som de fundo, história ou personagem.
  - Interação através de toque e arraste de dedo no *touchscreen*.
  - Disponível para *Android*.
  - Monetização: propaganda constante na parte inferior da tela e propaganda na tela inteira em certo intervalo de tempo.
  - Diferenciais: O raio tem uma barra de energia que determina quanto pode percorrer pela tela, então deve-se buscar a melhor trajetória.
- **Tetrobot and Co** é jogo *puzzle* que possui, em certas fases, o redirecionamento de feixes de laser. Possui HUD simples e muito agradável, com uma paleta de cores harmônica.
  - Tela em *landscape*.
  - Possui um personagem principal com controle simples através do toque na tela e arraste de dedo no *touchscreen*.
  - Possui tutorial e história em menu.
  - Disponível para *iOS* e *Android*.
  - Monetização: jogo pago no valor de R$7,99.
  - Diferenciais: O foco do jogo não é a reflexão por espelhos, mas essa mecânica também aparece em uma parte do jogo, onde espelhos são utilizados para refletir raios de laser, para abrir novos caminhos ou acionar dispositivos.

Figura 1 – Telas dos jogos analisados

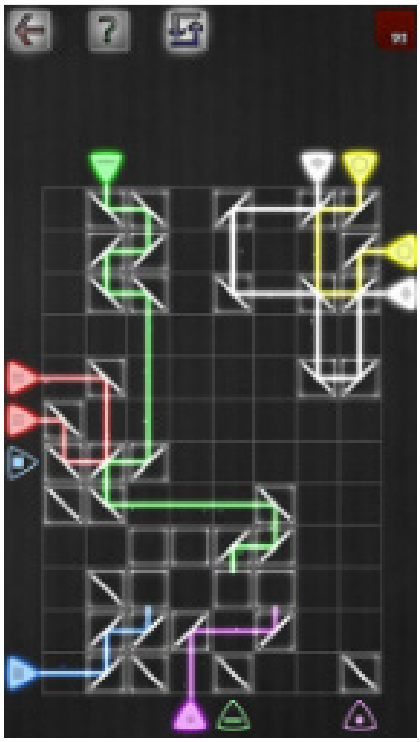
Tetrabot and Co



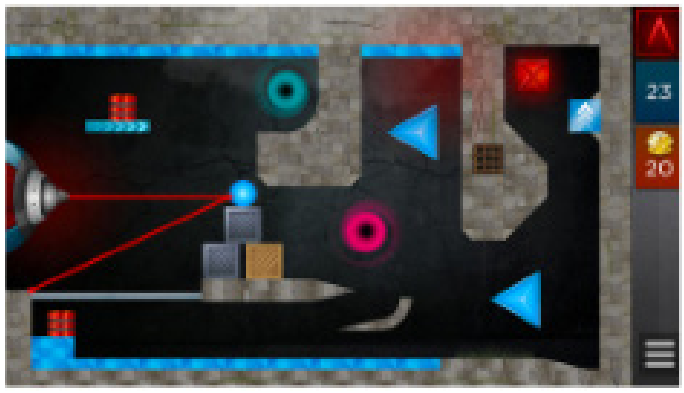
Laser Pop



Espelhos e Reflexos



Laserbreak Lite



A partir dos jogos analisados foi possível constatar que:

- O objetivo principal de todos os jogos analisados foi direcionar o feixe que saía do ponto A e acertar o ponto B.
- O direcionamento da tela varia entre o *landscape* e *portrait*.
- A HUD de todos eram simples.
- A maioria possuía um tutorial simples.
- A maioria não possuía personagem principal, história ou som ambiente que apoiasse o jogo.

- Majoritariamente a disponibilidade do jogo é para os dois sistemas mais utilizados: *iOS* e *Android*.
- A forma de monetização varia bastante.

Lumenide também tem como objetivo redirecionar um feixe de luz. Porém, para se destacar da concorrência, teve uma história para basear o seu conceito. A história se reflete diretamente na arte, no *gameplay* e no *level design*. Toda a HUD seria personalizada para caracterizar o jogo e Lumenide também possui um tutorial, que auxilia o jogador a aprender o jogo de forma mais dinâmica.

De acordo com a pesquisa de mercado, a melhor estratégia para Lumenide é a de monetizar através de propaganda, porém de uma forma que não atrapalhe na experiência durante a fase, como por exemplo, na forma de propaganda que apareça entre as fases e andares do jogo.

A partir da análise concluiu-se que há poucos jogos *mobile* envolventes e desafiadores com a mecânica de espelhos e luz e que Lumenide teria potencial para destaque no mercado.

1.3. Lumenide - Uma breve análise

Lumenide é caracterizado como um jogo *mobile puzzle* casual. Podemos definir um jogo *mobile* como jogo desenhado para dispositivos móveis, um jogo casual como um jogo acessível a maioria do público, sendo simples e rápido de aprender, e um jogo *puzzle* como um jogo com desafio que exige habilidade mental para ser resolvido.

Para fazer uma breve análise do jogo, o grupo utilizou os conceitos do livro **Design de Games** (SCHUYTEMA, 2013), que explica que todo jogo possui átomos, que são elementos básicos aplicáveis a qualquer jogo.

Tabela 1 – Elementos básicos de Lumenide

Átomo	Explicação
Objetivo claro	Permite ao jogador entender com clareza o objetivo. Resolver os *puzzle*s de espelhos propostos com o objetivo de redirecionar a luz até atingir o receptor, que quando ativado abre a porta para a próxima fase.

Átomo	Explicação
Vitórias aninhadas	Possibilita ao jogador conquistar subvitórias e subobjetivos. Há coletáveis escondidos nas fases, que servem ao *gameplay* como um subobjetivo.
Jogador como agente de mudança	As ações do jogador têm consequência. O jogador pode interagir com o ambiente, empurrando caixas, caixas com espelhos, interagindo com botões e alavancas e descobrindo portas secretas.
Contexto compreensível	Situa o jogador dentro do contexto e facilita a imersão. A utilização de metáforas dentro do jogo facilita o entendimento das funções. Os objetos de interação foram baseados em objetos reais. A utilização de paleta de cores foram adequadas para dar o devido enfoque.
Regras compreensíveis	Devem ser lógicas, para que o jogador compreenda como suas ações afetam o jogo. A regra básica é que a luz é refletida pelos espelhos e o ângulo influencia para onde a luz é redirecionada. O jogador deve utilizar os objetos disponíveis para redirecionar a luz.
Habilidade é necessária	O progresso do jogo deve ser baseado na habilidade do jogador, criando desafios e evitando a vitória vazia. Habilidade mental é a mais requisitada para resolver os *puzzle*s e a habilidade de destreza é testada em momentos onde há projéteis.
Feedback do sucesso	Deve estar presente para dar informações essenciais ao jogador sobre seu progresso. Os *feedback*s são feitos através de sons e animações condizentes.
Interface coerente	Permite um aprendizado suave e divertido, em que o jogador entende o jogo e como controlar a interface. A interface é limpa e simples, com o mínimo de botões e bem localizados, uma vez que a tela *mobile* é restrita.
Inteligência artificial para oferecer desafios	Criada para controlar os seres do jogo e dar desafio. A inteligência artificial, implementada nos chefes, é usada para os movimento dos inimigos sejam independentes da ação do jogador, oferecendo um maior desafio.
Dê descanso ao jogador	A ação do jogo é importante, mas deve ser balanceada com momentos de descanso. O jogo é salvo automaticamente ao se completar a fase e o jogador pode decidir sair, continuar ou refazer a fase.
Casualidade	Permite deixar o jogo menos previsível, dando um toque interessante e deve-se evitar os extremos de muita sorte ou sorte nenhuma. O chefe de nível possui movimentos influenciados por aleatoriedade.

Átomo	Explicação
Não deixe o jogador se perder	O jogador não conseguir se localizar pode levar à frustração. Há um mapa geral de fases que o jogador pode acessar a qualquer momento.
Padrões não devem ser muito simples	Os padrões devem ser bem estruturados para oferecer desafio ao jogador. Os padrões são bem claros e fáceis, porém o desafio está em combinar os diferentes espelhos e ativações de botões e alavancas.
Falhas devem ter um custo	A falha ter um custo auxilia a criar um jogo mais desafiador. Em caso de dano, o personagem morre. Em caso de caixas, elas não devem ser encostadas nas paredes. A sala pode ser reiniciada a qualquer momento através do botão *retry* (tentar novamente).
Ajude o jogador a se preparar para objetivos maiores	Os desafios aumentam de acordo com o progresso do jogo e deve-se auxiliar o jogador a se preparar. Toda mecânica nova que o chefe de nível terá será apresentada ao jogador em alguma sala anterior à última sala.
A história serve ao jogo	Um jogo não é um livro; toda história contada deve ser para enriquecer a experiência de jogo Os coletáveis de história servem para contar a história por trás do personagem principal e estão escondidos em certas fases, oferecendo ao jogador a oportunidade de explorar a fase e conhecer a história.

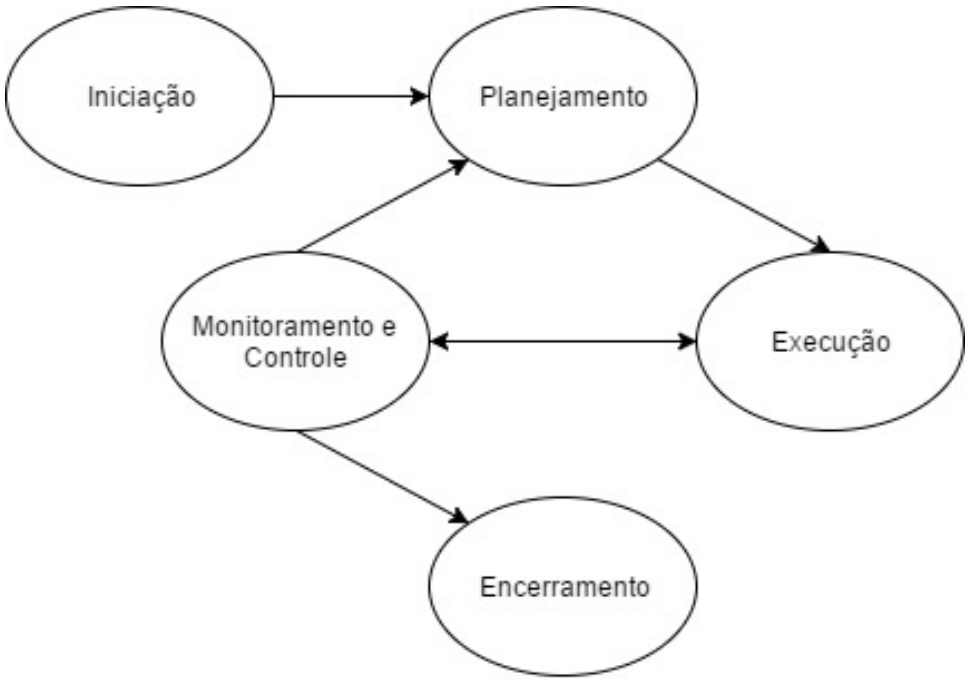
2. METODOLOGIA

Neste capítulo serão descritos as metodologias utilizadas para o desenvolvimento do jogo Lumenide, que envolveu gestão de projeto, história, *game design*, *game art design*, *level design*, som e programação. Todas as áreas se complementam e foram importante para que o jogo fosse desenvolvido.

2.1. Gestão de projeto

Possuir uma ótima ideia não significa desenvolver um ótimo projeto; há muitos fatores que podem levá-lo a falhar. Por isso, um bom gerenciamento de projeto é vital.

Figura 2 – Ciclo de vida de um projeto



A Figura 2 esquematiza o ciclo de vida de um projeto e em Lumenide, o ciclo se resume em:

- Iniciação (pré-produção): o período inicial pode ser dividido em:
  - *Brainstorm*: a ideia guardada de meses atrás foi discutida em grupo, novas ideias e mecânicas foram anotadas. Após uma análise, criou-se um primeiro esboço de como seria o jogo;

- Análise competitiva: foi realizada uma análise competitiva para ver se esse primeiro esboço seria original e bem aceita pelo público alvo;
- Orientação: o esboço inicial foi exposto ao professor orientador Bruno Daniel para discutir se as mecânicas que envolvem física seriam possíveis de ser implementadas.

- Planejamento (pré-produção e produção): após concluir a viabilidade do projeto, deu-se início ao planejamento, vários pontos foram discutidos, anotados e decididos. Outros pontos menos importantes foram anotados e somente planejados de acordo com a necessidade e andamento do projeto.
  - Ferramentas para gerenciamento do projeto:
    - Auxiliar na organização e desenvolvimento do projeto como um todo.
    - Ferramentas: Trello, Google Drive, Dropbox, Unity Collaborate, Facebook e Whatsapp.
  - Ferramentas para desenvolvimento:
    - Utilizadas no desenvolvimento para a produção em si.
    - Ferramentas: Unity, Clip Studio Paint, Spriter Pro, Visual Studio Community, Adobe Indesign e Audacity.
  - Ferramentas de divulgação:
    - Divulgar o jogo de forma eficiente, buscando atingir maior público.
    - Ferramentas: Facebook e Whatsapp.
  - Cronograma do projeto:
    - Auxiliar a organizar as tarefas de acordo com o tempo disponível.
    - Ferramentas: Google Drive.
  - *Backup*:
    - Armazenar um *backup* de todos os arquivos gerados.
    - Ferramentas: Google Drive e Dropbox.
  - Alinhamento do projeto:
    - Buscar um método ágil para realizar as tarefas, sempre alinhar as tarefas com o grupo e adaptá-las de acordo.

- Documentação:
  - Tudo referente ao projeto foi documentado para servir de base para a criação do jogo.
  - Ferramentas: Google Drive.

- Execução (produção): a execução do projeto utilizou várias ferramentas, buscando assim a melhor eficiência possível.
  - Fase de nivelamento: Antes de começar a execução do projeto de jogo em si, foi feito um nivelamento para que todos conseguissem usar as ferramentas do projeto, por exemplo, criar uma conta no Trello e aprender a usá-lo.
  - Fase de produção: Durante a execução, as tarefas listadas foram sendo realizadas de acordo com uma ordem de prioridade, que poderia mudar de acordo com a necessidade.
- Monitoramento e controle (produção): o monitoramento e controle ocorreram simultaneamente com a execução do projeto.
  - Durante a execução do processo, buscou-se sempre realizar reuniões e alinhamentos semanais para fazer as devidas adaptações do projeto e verificar o andamento das tarefas.
  - O ciclo definido foi de uma semana; toda semana era verificado se o objetivo foi cumprido, se ele precisava de revisão ou se poderia já ser considerado finalizado.
  - Até o protótipo, o próprio time testou o jogo; na versão *Alpha*, amigos e colegas da FATEC testaram e, a partir da versão *Beta*, o jogo foi publicado na Google PlayStore e houve *feedback* do público.
  - A versão *Alpha* é quando os recursos básicos do jogo estão implementadas, as mecânicas funcionando, mas podem haver problemas de *bugs*. É um teste de jogabilidade.
  - A versão *Beta* consiste em um jogo que os recursos do *Alpha* estão realmente funcionais, e o visual está mais próximo do jogo finalizado.

- Encerramento (pós-produção): após o projeto entrar em estágio final, foi dado início ao encerramento.
    - Os últimos detalhes foram revisados e implementados.

2.1.1. Ferramentas para gerenciamento de projeto

As ferramentas de gerenciamento de projeto auxiliam no desenvolvimento do projeto e foram escolhidas de acordo com a versatilidade e livre uso do *software*.

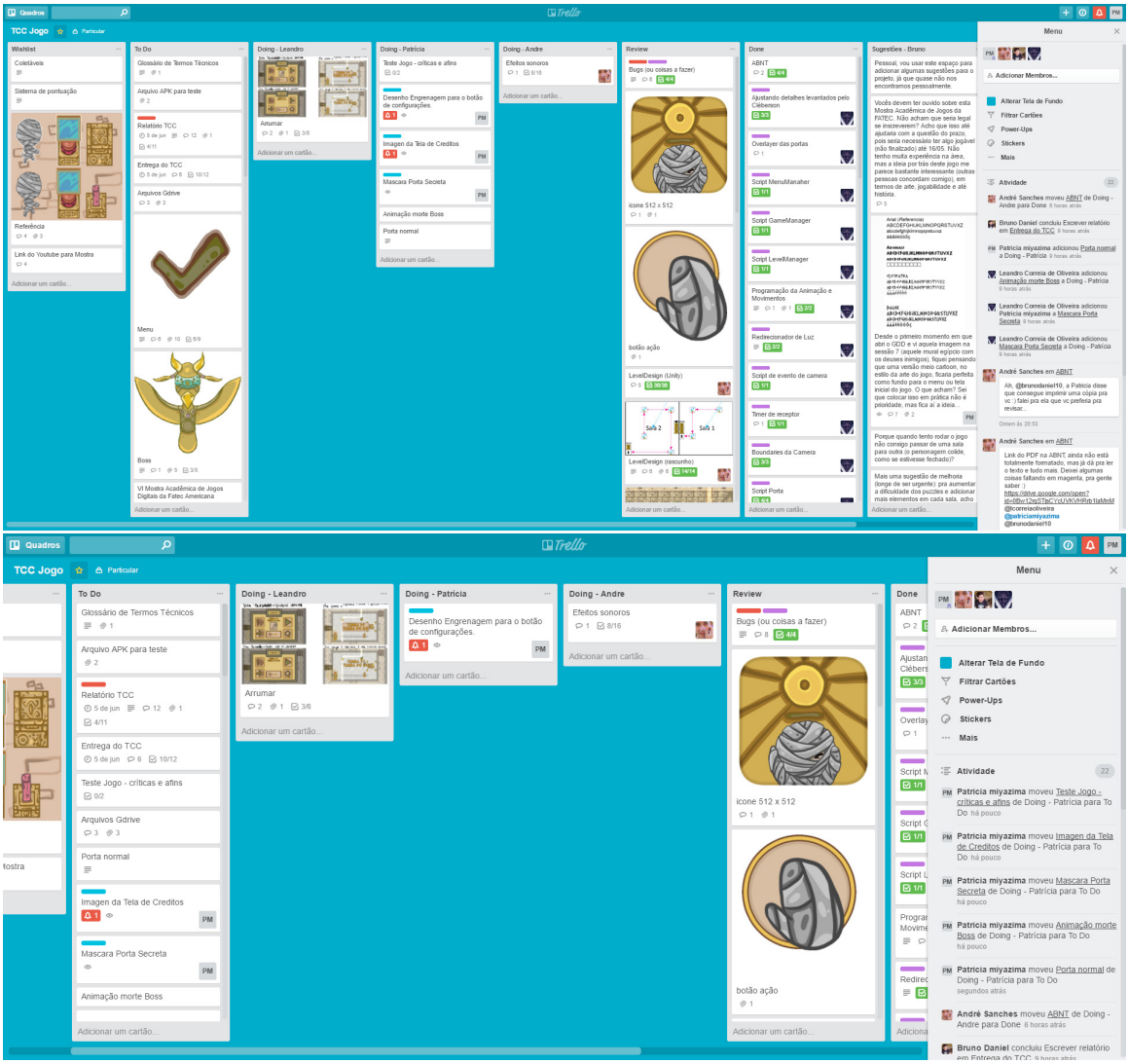
Todo projeto foi documentado em um *Game Design Document* (GDD); a partir desse documento, tarefas menores foram criadas de acordo com cada área do time:

- Arte: engloba todas as tarefas relacionadas ao *design* visual e artístico do jogo e ao *User Interface* (UI) e *User Experience* (UX) do jogo. A arte do jogo ficou sob encargo principalmente de Patrícia Tieme Miyazima.
- *Level Design* e audio: engloba todas as tarefas relacionadas a sons e músicas, ao *design* de níveis e jogabilidade em geral. O *level design* do jogo foi, na maior parte, desenvolvido por André Luiz Donizete Sanches.
- Programação: engloba todas as tarefas relacionadas à programação do jogo, implementação de mecânicas, movimentação de personagens e objetos, entre outros. Leandro Correa de Oliveira foi o principal programador no desenvolvimento do jogo.

2.1.1.1. Trello

Trello é uma ferramenta de colaboração que organiza seus projetos em quadros. Seu objetivo é que, em um olhar, sua interface informe o que está sendo trabalhado, quem está trabalhando em quê, e onde algo está em um processo. O Trello foi a ferramenta de gerenciamento de projeto escolhida pela sua facilidade, versatilidade, interface amigável e podendo ser utilizado de graça. A Figura 3 exemplifica como é a interface do trello e como as tarefas foram organizadas.

Figura 3 – Exemplo de utilização do Trello

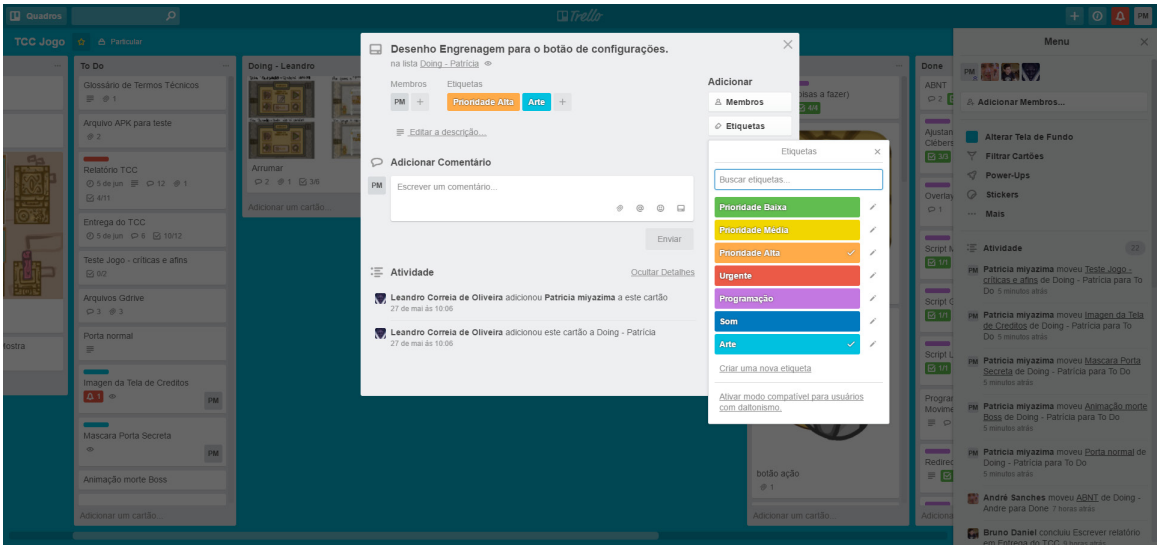


As tarefas foram também organizadas em ordem de importância, tarefas essenciais seriam realizadas primeiro e para uma melhor organização foram utilizadas as etiquetas como na Figura 4.

Para organizar o status das tarefas, foram criado os parâmetros de controle:

- WISHLIST: tarefas da lista de desejos, não precisam ser feitas;
- TO DO: tarefas a serem feitas;
- DOING: tarefas sendo feitas, uma por vez;
- REVIEW: tarefas já feitas e esperando aprovação;
- DONE: tarefas já aprovadas e finalizadas.

Figura 4 – Exemplo de utilização das etiquetas



As etiquetas de cores foram utilizadas para:

- Criar um grau de prioridades:
  - Verde: baixa prioridade;
  - Amarelo: média prioridade;
  - Laranja: alta prioridade;
  - Vermelho: urgente.
- Dividir as áreas:
  - Roxo: programação;
  - Azul escuro: *level design* ou som;
  - Azul claro: arte.

Através do grau de urgência é possível organizar melhor o projeto, por exemplo, caso apareça uma tarefa imprevista muito urgente, a tarefa é sinalizada com a cor vermelha.

2.1.1.2. Google Drive

Ferramenta de armazenamento e compartilhamento de arquivos da Google que disponibiliza 15 GB por usuário. Como tem maior espaço de armazenamento gratuito que a concorrência e sendo versátil, todo material gerado foi armazenado e organizado no Google Drive. Dentro do Trello na descrição de cada item, foi colocado um *link* para a pasta do Google Drive, onde se encontraria o *backup* do material gerado.

2.1.1.3. Dropbox

Ferramenta de armazenamento e compartilhamento de arquivos que disponibiliza 2 GB de graça por usuário. O Dropbox tem menos espaço de armazenamento gratuito, servindo de *backup* durante o desenvolvimento artístico, uma vez que permite que o usuário instale o Dropbox em sua máquina, acesse sua conta e já possa desenvolver e editar o arquivo da pasta Dropbox local. O próprio programa realiza o *sync* com a nuvem, atualizando os arquivos.

2.1.1.4. Unity Collaborate

Uma ferramenta da Unity que permite às equipes salvar, compartilhar e sincronizar o seu projeto Unity remotamente. Serviço desenhado com a facilidade de uso e rapidez de aprendizado. O Unity Collaborate foi escolhido, pois é de fácil uso, dispensando saber utilizar o git e permitiu que todos do projeto tivessem acesso rápido à última versão. Tudo que era produzido já poderia ser inserido, compartilhado e testado.

2.1.1.5. Facebook/Whatsapp

Redes sociais e aplicativos de comunicação. O Facebook serviu como principal forma de comunicação *online* do grupo, devido a sua versatilidade, pois permite criação de grupo de conversas, enviar arquivos e fotos.

O Whatsapp também foi utilizado para comunicação, porém com menos frequência.

2.1.2. Ferramentas para desenvolvimento do projeto

2.1.2.1. Unity

Unity, ou Unity3D, é uma *game engine* desenhada para facilitar o desenvolvimento de jogos para diversas plataformas. O Unity foi escolhido, pois permite a criação de jogos 2D e exportação do jogo sem custo para a plataforma ‘dispositivos móveis’ com sistema operacional *Android*. Outro fator foi a capacidade do time em interagir

com o *software*. Todos do time já tiveram contato com a *game engine* e o programador Leandro Correia de Oliveira já tinha experiência programando nela.

2.1.2.2. Visual Studio Community

Visual Studio Community é um ambiente de desenvolvimento integrado e foi escolhido, pois é uma versão gratuita especial para o uso no Unity, e contemplando melhor as funções do Unity, facilita na programação e depuração do código.

2.1.2.3. Clip Studio Paint

Um *software* para desenho, pintura e animação digital. O Clip Studio Paint foi escolhido por ser um *software* versátil com todos os requisito mínimos para se produzir arte e animação 2D, sendo sua aquisição mais barata que o Photoshop, *software* normalmente escolhido para a produção artística. Outro fator foi que a artista do grupo já possuía o programa e tinha experiência com o mesmo e poderia produzir com mais eficiência.

2.1.2.4. Spriter Pro

Software desenvolvido para auxiliar na animação bidimensional para jogos. O Spriter Pro foi escolhido por ser um *software* com custo reduzido e já adquirido pelo grupo, que permite a criação de animação 2D de forma rápida. Possui problemas de *bug* inesperados e sem motivo, mas o *autosave* auxilia nesse quesito.

2.1.2.5. Audacity

É um *software* para criação e edição de áudio. Foi escolhido por ser gratuito e de fácil utilização, além de ser um dos *software*s usados na FATEC. Existem muitos outros mais interessantes para criação de áudio no mercado, entretanto como os áudios não foram criados e sim editados, o Audacity supriu as necessidades do jogo.

2.1.2.6. Adobe InDesign

Software pago do pacote Adobe, mas um dos membros do grupo já o utiliza no trabalho. Foi utilizado para planejar o nível completo, as salas, desenhar os desafios e

soluções. Além disso, foi uma ferramenta essencial para o *Level Design* pois facilitou a aplicação no Unity.

2.1.3. Ferramentas para divulgação do projeto

2.1.3.1. Facebook

O Facebook é uma rede social ampla que permite rápida e fácil divulgação do conteúdo para um amplo público alvo, sendo ideal para a divulgação do jogo versão *Beta*.

2.1.3.2. Whatsapp

O Whatsapp não é uma rede social como o Facebook, mas permitiu divulgação do jogo através do compartilhamento dos *links* em grupos, sendo que atingiu um público alvo mais direcionado, como familiares e amigos.

2.1.4. Cronograma

O cronograma foi criado para auxiliar no desenvolvimento do TCC e levou em consideração o tempo disponível e a boa divisão das tarefas, como pode ser verificado na Tabela 2.

Tabela 2 – Cronograma

Tarefas do desenvolvimento	Março				Abril				Maio				Junho			
Brainstorm - Esboço inicial			■	■												
Criação do GDD			■	■												
Planejamento do projeto				■	■	■	■	■								
Ferramentas do projeto				■												
Ferramentas - Nivelamento				■												
Personagem - Arte/programação					■	■										
Cenário - Arte/programação					■	■										
Level design - Esboços					■	■	■									
Objeto em cena - Arte/programação					■	■	■	■	■	■						
Sistema de controle do personagem						■	■									
Implementação - Detecção de colisão						■	■									
Implementação - *Level design*							■	■	■	■	■	■				
Lançamento do *Alpha* e testes								■								
Correção do *Alpha* - *Bug*s e melhorias								■	■	■	■					
Sistema de mapa e de fases									■	■	■					
HUD - Arte/programação									■	■	■	■				
Implementação inimigos										■	■	■				
Lançamento do *Beta* e testes											■					
Correção do *Beta* - *Bug*s e melhorias											■	■	■			
Criação do relatório final TCC									■	■	■	■	■			
Entrega do relatório													■			
Polimento final do jogo										■	■	■	■			
Lançamento oficial do jogo													■			
Apresentação do TCC																■

2.2. *Game Design*

O *game design* ou *design* de um jogo engloba todo o planejamento do processo de desenvolvimento do jogo. Segundo Paul Schuytema (2013), *game design* é a planta baixa de um jogo, que servirá de guia e base para construí-lo.

O *design* de um jogo dita como será o jogo, definindo seu conteúdo, regras, *gameplay*, história, personagens e cenário. Para estruturar o *game design* do jogo foi produzido um *game design* document no google documents.

O GDD é um documento utilizado para estruturar o jogo como um todo e servir de guia, uma vez que define o jogo, como as regras e os conceitos que servem de base para a arte, o *level design* e a programação. O GDD de Lumenide começou a ser criado após os esboços iniciais do jogo, sendo um documento dinâmico que foi atualizado com o decorrer do projeto e de acordo com a necessidade.

2.2.1. *Gameplay*

Gameplay ou jogabilidade define a experiência do jogador e a maneira específica com que ele pode interagir com o jogo e quais seriam as consequências de suas ações, englobando todas as regras e padrões que o jogador pode realizar. A jogabilidade é a essência do jogo, é o que torna o jogo interessante e permite que a interação seja lúdica e significativa.

O jogo é um sistema e possui regras bem definidas que devem ser seguidas e o jogador deve estar disposto a entrar nesse círculo mágico. Define-se no livro **Regras do jogo** (SALEN, 2012), o círculo mágico como sendo o local onde o jogo acontece, regido por suas próprias regras e habitado pelos jogadores.

Um conjunto de regras isolado não define a experiência do jogo, mas é a partir delas que surgem as mecânicas, que, juntamente com a arte, história e *level design* cria-se a experiência.

Lumenide buscou estabelecer regras bem definidas que estivessem de acordo com o jogo proposto e que permitisse uma interação divertida do jogador com o jogo.

2.2.1.1. Objetivos

Segundo Jeannie Novak (2010), condições de vitória estipulam como o jogo pode ser vencido, e pode existir nenhuma, uma ou mais condições de vitória. As condições de derrota determinam como o jogador perde o jogo. Há a derrota implícita, em que o jogador perde ao não ganhar em primeiro, em relação aos outro jogadores e a derrota explícita, em que o jogador perde ao morrer ou acabar seus recursos.

Em Lumenide, as condições de vitória e derrota são as seguintes:

- A condição de vitória da sala é resolver o *puzzle* apresentado a cada sala, e a condição de vitória do jogo é o jogador passar por todas as fases, e chefes de níveis, e por todos os níveis até derrotar o último chefe de nível e finalizar o jogo.

- A condição de derrota da sala é morrer ao entrar em contato com projéteis, ou encostar caixas nas paredes, impossibilitando alguma resolução de *puzzle*, caindo em poços ou sendo imobilizado de alguma forma.

Os desafios e metas do jogo envolvem solução de enigmas, que exigem habilidade mental, tentativas e erros e paciência. Em cada nível existe um ou mais feixes de luz que descem pelo teto e o personagem deve mover objetos e interagir com eles para que, por meio de reflexão de luz, leve o feixe até uma fechadura especial, que vai abrir a porta para a próxima câmara.

2.2.1.2. Objetos

Muitos objetos foram criados no conceito inicial do jogo, e apenas alguns deles foram realmente implementados no seu estado atual. Alguns outros objetos serão apresentados em novos níveis, mais avançados.

Os objetos presentes nas salas, que são manipuláveis, estão descritos na Tabela 3.

Tabela 3 – Objetos manipuláveis

Imagem	Nome	Descrição
	Espelho fixo	Utiliza a reflexão, os raios de Sol são refletidos. No conceito inicial a luz também teria diferentes cores e alguns espelhos só iriam refletir algumas dessas cores.
	Espelho móvel	
	Espelho giratório fixo	
	Espelho giratório móvel	
Conceito	Cristais	Utiliza a refração, muda a cor, ângulo, ou aumenta a quantidade de feixes de luz.

Imagem	Nome	Descrição
	Lupa	Intensifica o feixe de luz, pode ser usado para queimar objetos.
	Caixa	Pode ser movida em quatro direções. É usada para acionar botões e impedir raios e projéteis.
	Botões (Azul e vermelho)	Estarão presentes no chão e paredes, podendo ser acionado pelo ação do jogador ou por pressão de um objeto. Alguns podem ser desativados por tempo.
	Alavancas	Acionam espelhos giratórios.
Conceito	Cristais Energizados	Alimentados com a luz do Sol, esvaziam-se com o tempo e podem ser combinados para acionar algo.

Alguns obstáculos servem para impedir o jogador de acessar alguns locais ou causar efeitos especiais, como visto na Tabela 4.

Tabela 4 – Obstáculos

Imagem	Nome	Descrição
	Veneno	Projétil que causa morte.
Conceito	Feixe intensificado	Causa morte.
	Porta	Impede o movimento. Acionada quando o raio de Sol atinge o receptor de luz.
	Pilar ou pilastra	Impede o movimento. Pode ser usado para se proteger de projéteis.

2.2.2. Interação

O personagem principal do jogo, Ramitz II, será controlado pelo jogador utilizando os controles no próprio HUD do jogo, mas os botões não ficam visíveis o tempo todo. O jogador pode fazer basicamente estas funções:

- Movimentação: O personagem pode ser movimentado com a utilização de um manche ou alavanca que está no HUD da tela, do lado esquerdo inferior. Pela movimentação o personagem pode mover caixas, espelhos móveis, e outros elementos móveis.
- Ação: O botão de ação fica invisível a maior parte do tempo e só é habilitado quando o personagem chega próximo a algum elemento acionável, como uma alavanca ou botão de acionamento. Esse botão de ação fica do lado direito inferior da tela e, quando necessário, ele fica visível. O jogador só precisa pressionar a tela para acionar.
- Acesso ao menu: Ao acessar o menu, pelo botão de *Pause* localizado no canto superior direito, o jogador tem acesso ao menu, onde pode ligar ou desligar sons e música, acessar o mapa de salas, reiniciar a sala atual ou sair do jogo.

2.2.3. Percepção e emoção

Com base nos conceitos de Percepção e Emoção descritos por Schuytema (2013), Lumenide possui:

- Efeitos sonoros: Cada objeto possui seu próprio som, indicando o tipo de interação;
- Música: O jogo possui uma música para as fases comuns, porém uma diferenciada para salas do chefe de nível e interfaces;
- Movimento: Animações elaboradas devem identificar quando o objeto está sofrendo alguma interação;

- Luz e cor: A paleta de cores selecionada foi pensada de acordo com o propósito. As HUDs possuem cores mais suaves, utilizando maior saturação para fazer destaque de elementos importantes. Dentro de fases, o fundo utiliza cores pouco saturadas para destacar os objetos de paleta mais saturada, porém cuidando para manter a harmonização;
- Padrões visuais e auditivos: Cores específicas são utilizadas nos objetos que possuem interação, criando um padrão visual para o jogador identificar no que pode atuar. Foram escolhidos sons específicos de acordo com o propósito, para que o jogador possa identificar o que está acontecendo na fase;
- Percepção da imersão: O jogo possui inspirações no mundo real, cada objeto lembrando o seu respectivo no mundo não virtual, possuindo caracterizações do mesmo;
- Fluxo de emoções: O fluxo é de um nível por vez, não segurando o jogador por muito tempo, com o propósito de ser um jogo *mobile* casual que o jogador possa jogar a qualquer momento e progredir em seu próprio ritmo;
- Realização: Permitir que o jogador se sinta realizado depois de finalizar um *puzzle*;
- Realização de problemas: O jogador deve resolver os *puzzle*s propostos;
- Reação aos personagens e à história: O jogo possui itens coletáveis que serão encontrados para contar a história do personagem;
- Comportamento viciante: Incentiva o jogador a finalizar todos os *puzzle*s e encontrar todos os coletáveis;
- Juntando tudo: O público alvo são pessoas que buscam jogos casuais *puzzle*s para plataforma *mobile*.

2.3. Contando a história

Apesar de muitos jogos de quebra-cabeça não possuírem história ou personagens dentro do jogo, Lumenide é um jogo que conseguiu combinar ambas as características dentro do seu *gameplay*. Não há uma história profunda como em *The Last Of Us* (Naughty Dog, 2013), mas há uma história que fundamentou o jogo e ela é refletida no

cenário e na progressão das fases, podendo ser descoberta a partir dos colecionáveis de história que estão escondidos em certas fases.

2.3.1. História do jogo

O jogo conta a história do faraó Ramitiz II, um grande adorador do Sol, que se vê fascinado com a ideia de poder passar à eternidade rodeado pela luz solar.

Ramitiz II então ordena a construção de sua pirâmide de forma peculiar, criando formas incríveis de redirecionar a luz para dentro das câmaras. Sua simples ideia se transforma em algo diferente após um sonho profético que teve, no qual seres malignos surgiriam dentro de sua pirâmide e destruiriam toda a luz existente. Então Ramitiz II decide criar labirintos internos e enigmas com sua adorada luz; com isso, os seres malignos nunca alcançariam seu objetivo.

Após anos de construção, a pirâmide estava quase pronta e o faraó poderia viver o resto da sua vida tranquilo, porém os seres malignos descobriram o seu plano e tomaram sua vida antes do esperado, faltando apenas a parte final a ser feita. Sem essa parte final, os seres malignos poderiam escapar.

Ramitiz II é colocado para descansar em sua câmara do rei.

Rá, o Deus do Sol, que o abençoou com o poder de sonhos proféticos durante a vida, resolve ajudá-lo também nessa pós-vida e lhe concede uma segunda chance para que enfrentasse os seres malignos com sua pirâmide.

Algum tempo depois, Ramitiz II acorda em sua tumba, desnorteado e sem saber ao certo o que está acontecendo, esquecendo inclusive quem é e o que fez durante a vida.

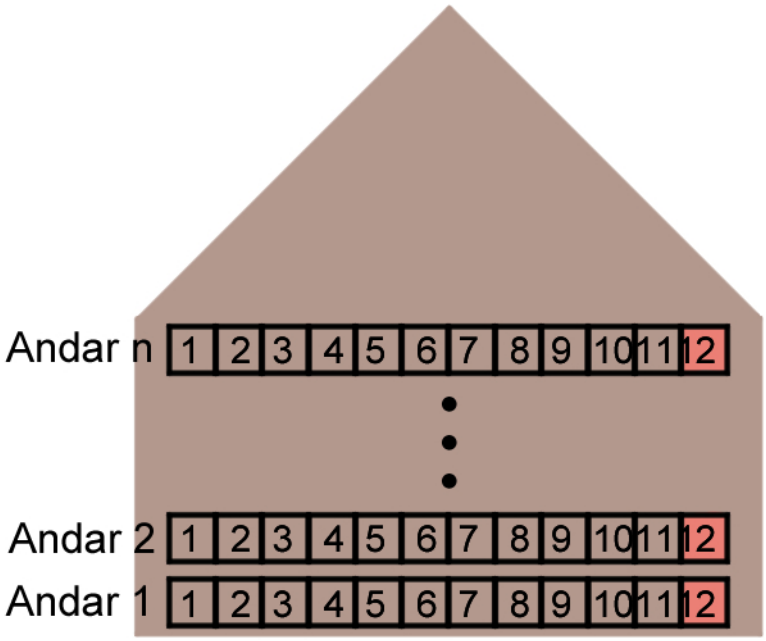
Ao dar o primeiro passo para fora do sarcófago, aparece um gato falante, que diz que irá orientá-lo nessa pós-vida.

Ramitiz II deve superar todos os obstáculos, que ele mesmo criou e derrotar os seres malignos, descobrindo sua história no processo e qual o propósito de ter voltado à vida.

2.3.2. Roteiro de três atos

O roteiro do jogo foi baseado na estrutura cinematográfica de três atos de Syd Field (2001), com a ideia de que o jogo teria um começo, meio e fim.

Figura 5 – Esquema da pirâmide com as fases



De acordo com a Figura 5, o jogo possui um escopo maior que seria toda a pirâmide com vários andares subterrâneos e um escopo menor com cada andar e suas fases.

Na visão macro do jogo, Ramitiz II deve passar pelos andares da pirâmide que são compostos por 12 fases cada e a dificuldade do andar aumenta com o progresso do jogo. O problema maior do jogo é derrotar todos os chefes de nível e no final seria revelado qual era a finalidade de sua pirâmide. Só assim, Ramitiz II entenderia o propósito da sua segunda vida e poderia descansar em paz.

Em uma visão micro, considera-se o andar e as fases, o objetivo do andar é sempre o mesmo, que é passar pelas 11 fases e derrotar o chefe de nível na última fase para poder passar para o próximo andar. Cada andar terá seu próprio contexto com chefe de nível característico e novas mecânicas, para que as fases e *puzzle*s de cada andar sejam únicos.

Cada fase é curta e possui o ciclo completo dos três atos. O problema principal de cada fase é sempre o mesmo: o personagem Ramitiz II está preso em uma sala e,

para sair, deve resolver o *puzzle* proposto. O contexto muda de acordo com a fase, pois cada *puzzle* é diferente, às vezes, deve-se utilizar alavancas, outras, empurrar caixas, e esses diferentes contextos que permitem o jogo se tornar desafiador e interessante.

Os três atos dentro da fase são:

- Ato I (início): o ato I visa introduzir a história e o problema do personagem principal, tentando ao máximo envolver a audiência.
    - Dentro do jogo Lumenide, o ato I é quando o jogador entra na fase, sendo introduzido ao problema do personagem Ramitiz II. Se é a primeira vez que o jogador se depara com esse contexto do problema, é apresentado um tutorial para que assim o jogador se situe, por exemplo, é a primeira vez que o jogador se depara com uma alavanca, então é apresentado um tutorial de como utilizar a alavanca.
- Ato II (meio): o ato II é parte intermediária da história, que engloba a tensão dramática, em que o personagem deve enfrentar os obstáculos para resolver o problema introduzido no ato I.
    - O jogo Lumenide propõe *puzzle*s que o jogador deve superar para resolver o problema de estar trancado na sala.
- Ato III (fim): o ato III é quando o problema do ato I é resolvido e tem-se o desfecho.
    - Após o jogador resolver o *puzzle*, a porta é aberta, permitindo que ele passe para a próxima fase.

2.3.3. Personagens

Os personagens de um jogo podem ser divididos em:

- Personagem de jogador: é o personagem que o jogador controla. O jogador pode controlar múltiplos personagens, como ocorre em vários jogos da franquia *Final Fantasy* (Square Enix), ou controlar somente um personagem, como em Lumenide, em que só é possível controlar o avatar faraó Ramitiz II.

- Personagem não jogável (NPC): é o personagem que não é controlado pelo jogador e possui inteligência artificial. Os NPCs comumente estão presentes para interagir com o jogador, seja para ajudar ou atrapalhar. Em Lumenide, o gato RanRan é seu guia espiritual e aparece para ajudar Ramitiz II em certas partes.

Há dois tipos de personagens dentro de Lumenide, os personagens:

- Fictícios que foram criados do zero e não existem na vida real.
- Míticos que se baseiam em seres já existentes da mitologia, no caso, egípcia.

Os arquétipos presentes são:

- Herói: é o avatar do jogador, sendo o protagonista e quem está realizando a maior parte da ação dentro do jogo. É ele quem tem o problema e precisa resolvê-lo. Em Lumenide, Ramitiz II é o herói.
- Sombra: é o opositor ao herói, sendo o antagonista. Em Lumenide, sombra é representado pelos chefões finais.
- Mentor: é o guia do herói em sua jornada, é quem o auxilia a partir. Em Lumenide, o gato RanRan representa esse papel.
- Guardião: é quem testa o personagem principal para que ele demonstre que é capaz. Em Lumenide, a estátua é o guardião que deixa a porta trancada até que o *puzzle* seja resolvido.

2.3.3.1. Faraó Ramitiz II

Figura 6 – Faraó Ramitz II

Biografia: nasceu sob a luz do Sol do solstício de verão. Ao abrir os olhos o primeiro ser que viu não foi sua mãe, mas um belo ser iluminado. A partir daquele momento, Ramitiz II viveu sua vida para o Sol, e dele recebeu a benção dos sonhos, que lhe permitia ter breves visões do futuro.

Em jogo: é o herói, o avatar que o jogador controla.

2.3.3.2. Gato guia RanRan

Figura 7 – Gato guia RanRan

Biografia: Antigo companheiro do Faraó e sempre o acompanhava. Quando o faraó ressuscita, RanRan volta como um guia espiritual, ajudando-o em sua jornada pós morte.

Em jogo: é um NPC que aparece para explicar tutoriais do jogo e a história do faraó.

2.3.3.3. Rá Deus do Sol

Figura 8 – Rá Deus do Sol 	Biografia: Rá é o Deus do Sol que sempre existiu, sendo benevolente, costuma interferir com as vidas dos mortais. Em jogo: quem concede a benção dos sonhos e a segunda chance ao faraó Ramitiz II.

2.3.4. Inimigos

Muitos jogos *puzzle*s não possuem inimigos para se enfrentar, Lumenide implementou o conceito de que os inimigos existentes seriam:

- Inimigos estáticos com poder similar ao chefe de nível, com o propósito de introduzir nova mecânica e treinar o jogador antes da grande batalha. Esses inimigos ficam na cena do jogo e não podem ser destruídos.
- Inimigos do tipo *Boss* que são os chefões finais que podem ser derrotados através da resolução dos *puzzle*s. Os chefes são baseados em deuses egípcios e possuem um corpo com mistura entre o corpo humano e outros animais. Possuem o conceito de que negavam totalmente a luz e por isso sempre teriam seus olhos cobertos.

Os chefes conceituados para o jogo são descritos a seguir, porém somente Sokar foi implementado e a intenção seria que houvesse mais chefes finais.

2.3.4.1. Sokar

Figura 9 – O Grande Chefão Sokar

Chefe 1, o grande ser maligno do céu é retratado na Figura 9. Baseado em Sokar, o Deus das Trevas associado ao falcão com coroa Atef (coroa branca do Alto Egito com duas plumas).

Conceito: é um ser que quer destruir toda luz, inclusive o Sol. Irritado com o faraó por não ser adorado como a grande ave, ele decide impedir que a luz solar chegue até o faraó.

Puzzle da sala: envolve pilares altos, feixe de luz, caixas, espelhos e portas.

Poderá ser derrotado através de feixe de luz.

2.3.4.2. Antk

Chefe 2, a grande destruição, ainda não implementado. Baseado em Antk, a Deusa da guerra associada a abelha/besouro com coroa de penas.

Conceito: é um ser que traz a destruição por diversão Antk vê essa grande pirâmide diferente e resolve se divertir destruindo-a.

Puzzle da sala: envolverá pilares, luz intensificada, feixe de luz, caixas, espelhos e portas.

Poderá ser derrotado ao cair em um buraco.

2.3.4.3. Apep

Chefe final, ainda não implementado, também conhecido como Apófis, o caos do submundo. Baseado em Apep, criatura em forma de serpente, inimigo dos deuses do Egito, principalmente do deus Rá. Ressuscita sempre ao cair da noite, e morre ao nascer do Sol.

Conceito: Personificação da destruição e do mal, é uma serpente gigantesca que cobre a luz do Sol com o seu corpo. Encontra refúgio em uma das câmaras da pirâmide, onde cresce muito em tamanho. Esta câmara é muito escura, e Apep bloqueia qualquer entrada de luz direta para esta sala.

Puzzle da sala: envolverá pilares, espelhos, caixas e a lupa (intensificador de luz do Sol). O *timing* aqui é essencial, pois Apep bloqueará qualquer luz que entrar na câmara.

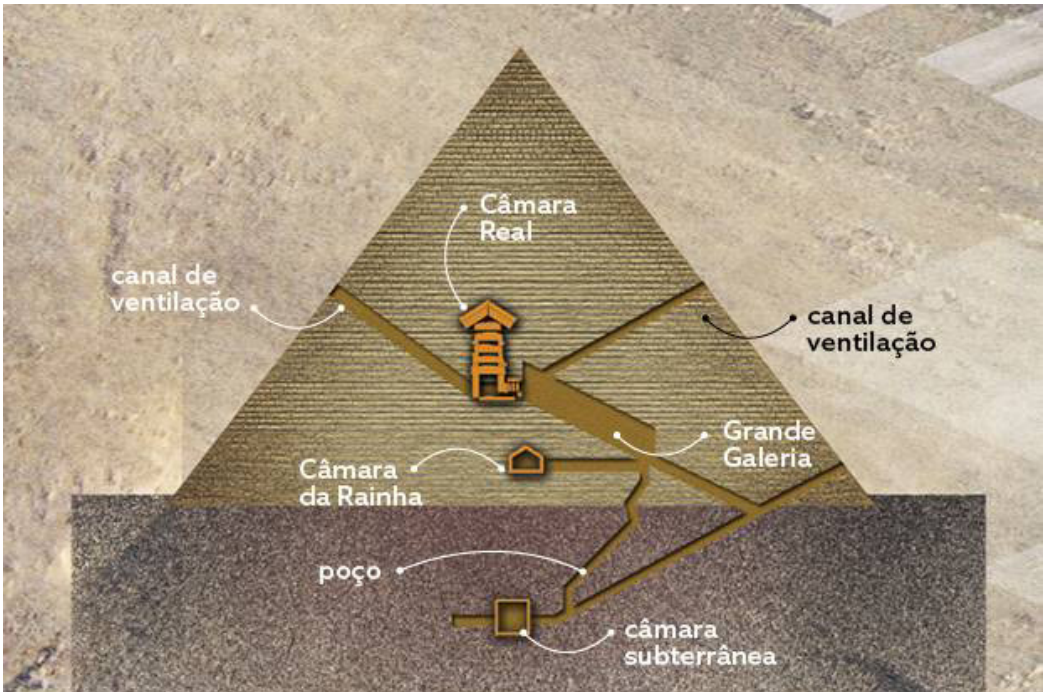
Poderá ser derrotado pela luz do Sol intensificada.

2.3.5. O mundo

O jogo inteiro explora uma pirâmide, a qual possui vários andares e cada andar possui 12 fases, 11 fases de *puzzle*s e 1 fase de *puzzle* chefão. Após completar as 12 fases, o jogador pode avançar para o próximo andar da pirâmide.

Cada câmara da pirâmide será uma fase e para avançar no jogo, a fase deve ser completada. O jogo teve como referência a estrutura de uma pirâmide real, como na Figura 10.

Figura 10 – Estrutura interna de uma pirâmide



Fonte: http://mundoestranho.abril.com.br/historia/como-era-o-interior-da-piramide-de-queops-no-egito/)

A partir da análise de uma pirâmide e considerando a Figura 10 como referência, concluiu-se que uma pirâmide tem cinco partes básicas:

- Entrada: Virada para o norte, aberta durante a vida do faraó e bloqueada com pedras após seu sepultamento.
- Câmara subterrânea: A teoria mais aceita diz que a câmara subterrânea era construída para enganar saqueadores e guiá-los para armadilhas e para longe dos tesouros reais.
- Câmara da rainha: Um mistério, pois o corpo da rainha era enterrado em outra pirâmide ao lado da pirâmide do faraó.
- Grande galeria: Uma passagem para a câmara real.
- Câmara do rei: Parte principal da pirâmide para guardar o corpo do faraó e seus objetos pessoais.

Essas partes básica da pirâmide foram referenciadas dentro do jogo como:

- Entrada: está virada para o norte e está trancada. Será o objetivo final do faraó, que só chega e abre essa porta após passar por todos os andares.
- Câmara subterrânea: Representa a maioria das fases, onde estão os *puzzle*s que o jogador deve resolver.

- Câmara da rainha: é uma câmara para enfrentar um chefe.
- Grande galeria: possui a história do faraó, porém partes foram roubadas e escondidas pelos seres malignos em galerias secretas, e com a ajuda do gato RanRan, o faraó deve coletar as partes faltantes e relembrar sua história.
- Câmara do rei: Ponto inicial do jogo, o faraó acorda em sua tumba e deseja sair da pirâmide.

Além dos pontos de ventilação, também haverá pontos de entrada de luz, que será a fonte utilizada para resolver os *puzzle*s.

2.4. *Game Art Design*

Game art design abrange toda a criação artística do jogo, desde a pré-produção com a criação de *concepts art*, passando pela produção e finalização dos itens da lista de arte até o desenvolvimento de arte promocional para a divulgação do jogo.

A arte escolhida foi o *cartoon*, devido ao curto tempo e à facilidade de se desenhar, sendo um estilo que agradaria o público alvo.

A partir do GDD, foram listados no Trello os itens que seriam necessários produzir artisticamente. Essa lista foi ordenada de acordo com uma prioridade.

Para criação do *design* visual foram considerados vários pontos como: princípios do *design*, apresentação das informações visuais, composição, formas de comunicação, cor, tipografia e criação de símbolos.

Um bom *design* gráfico apresentará as informações, de forma que o jogador entenda o jogo com o mínimo de esforço possível.

2.4.1. *Concepts Art*

Concept art ou arte conceitual é uma ilustração que representa uma ideia de um produto antes dele ser finalizado. Na produção de um jogo, as artes conceituais são

muito importantes, pois elas ajudam a definir como será o jogo, não só visualmente como também na experiência de jogo.

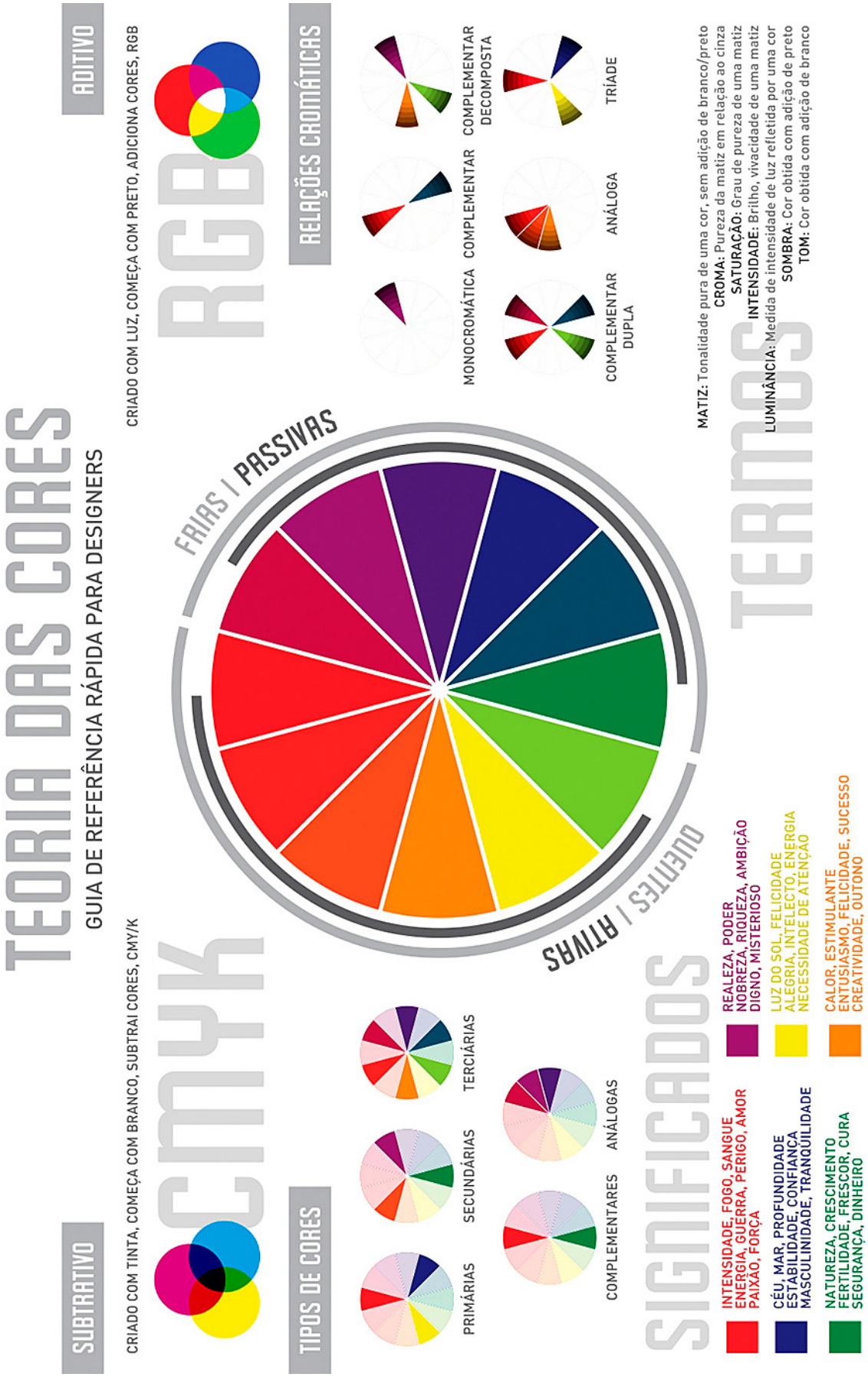
A partir dos concepts é possível transmitir várias informações, como a aparência visual, a paleta de cores, qual o propósito do objeto, qual sensação deve transmitir, entre outros.

Figura 11 – *Concepts Art* do jogo

Em Lumenide, os itens tiveram um conceito criado em forma de esboço e todos eles tentaram utilizar objetos da vida real como referência, tentando assim criar uma conexão visual entre o objeto do jogo e o da vida real, facilitando para o jogador entender o que é cada objeto como na Figura 11.

A criação de conceitos envolve muita criatividade e busca por inspiração e referências.

Figura 12 – Teoria da cor



Fonte: <https://allysoncorreia.files.wordpress.com/2012/02/> Acessado em 4/6/2017

2.4.2. Teoria da cor

A teoria da cor envolve vários conceitos que foram aplicados na criação visual do jogo e esses conceitos estão resumidos na Figura 12.

Considerando que o trabalho final seria para a mídia digital, o sistema de cores aditivas RGB com harmonia do tipo divisão complementar foi escolhido para a criação da paleta de Lumenide, que está demonstrada na Figura 13.

Figura 13 – Paleta de cores de Lumenide

O primeiro passo para definir a paleta foi entender como as cores poderiam ser utilizadas e quais seriam as cores mais adequadas para dar o clima correto ao jogo.

O jogo deveria transmitir a ideia de um clima quente, pois todo o jogo se passa dentro de uma pirâmide que está localizada em um local que é quente de dia e frio de noite. A partir do entendimento de cores frias e quentes e que cada cor tem o poder de transmitir ideias e sentimentos, foi decidido que seria necessário uma paleta que englobasse ao menos uma cor quente e uma cor fria.

Foram utilizados os conceitos de temperatura e saturação da cor para dividir o plano de jogo (com objetos de interação) do plano de fundo (cenário).

Como cores quentes transmitem a sensação de calor, vitalidade e alegria e quando utilizadas adequadamente, trazem o objeto para o primeiro plano, os objetos que o jogador poderia interagir teriam tonalidades mais quentes e saturadas.

Como as cores frias transmitem a ideia de frio, tranquilidade e distanciamento, o cenário teria cores mais frias e menos saturadas.

Para dar um destaque adequado sem perder a harmonia, os objetos importantes como portas, estátua, espelhos, botões e alavancas utilizam as cores da paleta com mais saturação como demonstrado na Figura 14 e os objetos de perigo utilizam cores que transmitem a ideia de perigo e mistério como tons de roxo.

Figura 14 – Objetos de destaque

2.4.3. *Design* de interface

Enquanto o *gameplay* define as regras e como deveria ser a interação do jogador com o jogo, é somente através da interface que essa conexão realmente se estabelece.

A interface engloba todo o visual do jogo, telas, imagens em geral e HUD.

A interface é a forma que o jogador tem para controlar o personagem, interagir com o cenário e com as funções do jogo. O *design* da interface deve ser centrado no usuário e assim otimizar a experiência e imersão do jogador.

Um ponto importante ao se desenvolver a UI e UX do jogo é considerar que há três condições nas necessidades do usuário:

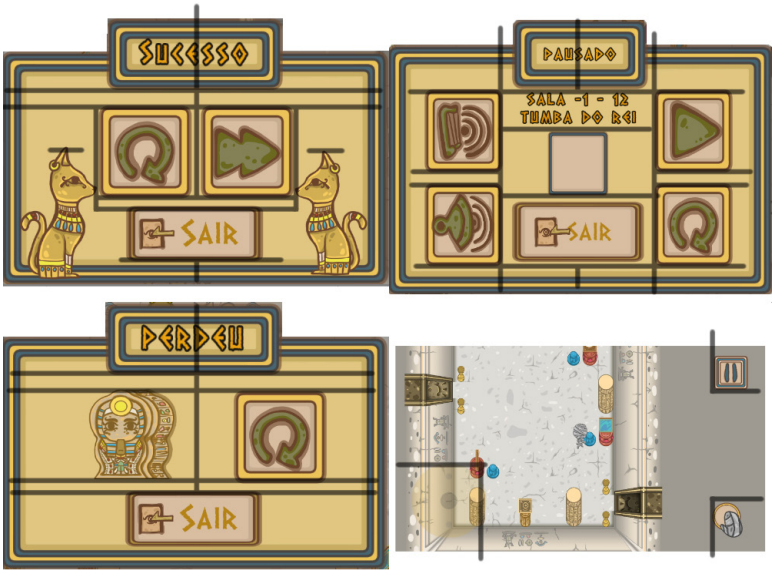
- Funções que o usuário diz precisar: não é porque o usuário acha que precisa de algo que ele deve ser implementado, deve-se fazer uma análise para ver se a função realmente é necessária ao jogo. Muitas funções podem tornar o ambiente de jogo poluído.
- Funções que o usuário realmente precisa: analisar qual o problema real que deve ser resolvido, quais são as funções realmente necessárias ao jogo e ao jogador para promover uma experiência plena.

- Funções que o usuário não sabe que precisa: focar nos desejos do usuário e nos problemas. O que o jogador realmente quer nem sempre está claro; deve-se satisfazer esses desejos ocultos.

Considerando esses fatores, Lumenide foi criado para tentar guiar o jogador pelo jogo de forma intuitiva e disponibilizando as interações necessárias para se ter a melhor experiência.

A distribuição dos elementos na HUD, nas janelas e nas telas está exemplificado na Figura 15 e levou em consideração o peso que cada elemento teria, sua cor, sua função e o balanço simétrico.

Figura 15 – Distribuição dos elementos



Considerando a tela do celular em *landscape*, avaliou-se como o jogador seguraria o celular e onde os botões ficariam melhor posicionados, deixando assim mais confortável de se jogar, buscando uma melhor ergonomia. Alguns exemplos de objetos podem ser vistos na Figura 16.

- Os objetos como Espelhos, alavanca e botões estão ilustrados na Figura 16 e cada tipo de espelho tem sua função e foi desenhado de acordo, utilizando também cores específicas para o jogador já ir criando uma conexão:
  - Espelhos empurráveis recebem cores amarronzadas e têm formato retangular.

- Espelhos Giratórios recebem a cor avermelhada e têm formato arredondado.
- Alavanca e botão que podem ser acionados através do botão de ação da tela recebem a cor azul.
- Botão que pode ser pisado recebe a cor avermelhada.

Figura 16 – Objetos com cor e formato específicos

Os 6 princípios de *design* do livro **O Design do dia a dia** (NORMAN, 2006) serviram de base para a criação da interface:

- Visibilidade: funções importantes devem estar visíveis ao jogador e transmitir a ideia correta.
- *Feedback*: é o retorno da informação em relação às ações executadas. Em jogos, o *feedback* deve ser constante, pois é uma mídia interativa.
- Restrições: é uma forma de direcionar o uso correto do item, uma vez que restringe a quantidade de escolhas que o jogador pode ter. Restrições bem utilizadas guiam o jogador com fluidez.
- Mapeamento: dita a relação entre o objeto e o que suas funções influenciam no mundo.
- Consistência: cria uma relação entre elemento e função.
- *Affordance*: perceptívelmente óbvio para a pessoa já saber como usá-lo.

2.4.3.1. Visibilidade

A visibilidade encontra-se em todo o jogo e está demonstrado na Figura 17:

- HUD: é composta por botão de ação, setas de movimentação, botão de *pause* e barra de vida do chefe de nível. As setas de movimentação e botão de *pause* sempre estão visíveis ao jogador durante a fase inteira e em todas as fases. O botão de ação somente aparece em fases em que é utilizado e quando se está perto do objeto que pode sofrer a ação. A barra de vida do chefe de nível só está presente na sala do chefe de nível e serve de guia para o jogador saber quanto falta para matá-lo.
- As janelas foram estruturadas para mostrar os itens de forma clara e com cores suaves.
- Objetos importantes utilizam cores específicas para ter o devido destaque e direcionar o olhar do jogador. Além da cor diferente, também utilizou-se o tamanho dos objetos; a escala escolhida foi de 128px por 128px por *tile*, o personagem possui 128px de largura por 256px de altura e o botão possui 128px por 128px, sendo um objeto relativamente grande, se comparado com o personagem.

Figura 17 – Visibilidade do jogo

2.4.3.2. *Feedback*

O feedback visual está presente através de animações dos objetos e efeitos visuais como o brilho, demonstrado na Figura 18.

- HUD: possui animações simples, que indicam ao jogador se algo está sendo apertado.
- Personagem principal: possui animações de andar, morrer, empurrar caixa e *idle*.
- Portas: possuem animações e efeitos visuais.
- Estátua: possui animação e efeito visual de brilho ao ser acesa.
- Espelhos: possuem animações condizentes ao que deveria estar ocorrendo com o feixe de luz; uma vez que ele direciona o feixe, o visual deve estar condizente.
- Botões e alavancas: possuem desenhos para os dois estados possíveis, ativado e desativado.
- Janelas: há janela de vitória e derrota. Quando o jogador obtém sucesso na fase, há uma janela informando-o do sucesso e quais seriam suas próximas ações. O mesmo ocorre quando o jogador morre.
- Projéteis: são perigosos ao jogador. Para dar o *feedback* correto, utilizamos cor roxa, que transmite a sensação de veneno e perigo.
- Sons: Todas as ações do jogador tem um som característicos de feedback.

Figura 18 – *Feedback* do jogo

2.4.3.3. Restrições

As restrições foram utilizadas para limitar o jogador de forma adequada:

- Sala: as paredes da sala foram criadas para que visualmente o jogador já saiba que a parede é o limite de onde ele pode andar.
- Trilhos das caixas espelho: dentro do jogo há espelhos que podem ser empurrados, e só se movem em cima de trilhos. Foi criada a imagem de trilho adequada para informar o jogador que o movimento é restrito.

2.4.3.4. Mapeamento

O mapeamento, a consistência e o *affordance* consideraram conhecimentos prévios do jogador e tentou-se criar imagens que se relacionassem com as reais, possibilitando fácil reconhecimento do que é o objeto e para que ele deveria ser utilizado.

- HUD: os elementos da HUD foram desenhados e posicionados de acordo com o convencional de um controle de console, possuindo aparência de cruz com setas. O botão de ação tem aparência redonda com uma mão do faraó em cima, para transmitir ao jogador a ideia de que é um botão para ser apertado e ao ser clicado, uma animação é disparada.
- Janelas: utilizou-se de metáforas e mapeamentos já estabelecidos no público alvo. Metáforas são representações de um objeto que realiza determinada função, a metáfora dentro de Lumenide é a representação gráfica da função como demonstrado na Figura 19.

Figura 19 – Metáforas do jogo

2.4.4. *Sprites*

Sprite em computação gráfica é um item gráfico 2D ou 3D que pode ser movido pela tela, utilizado para a criação das animações através de sequências de *sprites* (Figura 20). *Assets* refere-se a qualquer objeto que seja inserido dentro do jogo.

Para criação dos *assets* visuais utilizou-se um padrão de referência, um *tile* teria 128px por 128px e todos os outros objetos deveriam ser desenhados considerando essa referência de tamanho.

Os *sprites* estáticos e a maioria das animações foram criados no Clip Paint Studio; já animações mais complexas, que possuem efeito, foram criadas utilizando o Spriter Pro. As animações foram feitas quadro a quadro, não utilizou-se *bones* para fazer movimentação dos objetos.

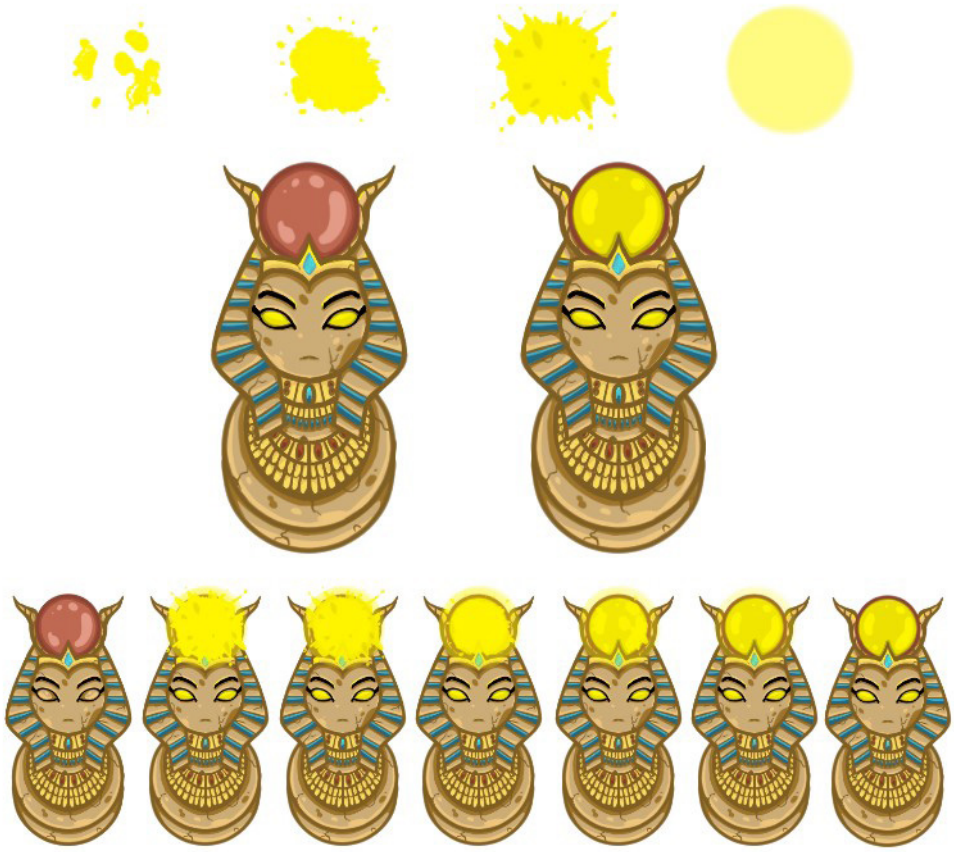
Todas as imagens foram mandadas separadamente, ao invés de montar uma *sprite sheet* com todas as imagens. Essa forma foi escolhida, pois a artista e o programador alinharam que essa seria a melhor forma de colocar as imagens dentro do jogo, já que no Unity a *sheet* precisaria ser cortada.

As animações de movimentação utilizam no mínimo cinco frames como ilustrado na Figura 20.

Figura 20 – Sprites do Personagem

Efeitos de brilho foram criados com o auxílio do Spriter Pro como demonstrado na Figura 21.

Figura 21 – Efeito do brilho na estátua

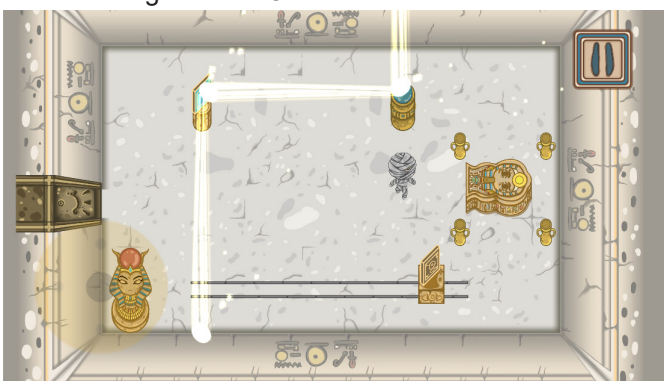


Criou-se a estátua apagada e acesa e o efeito de brilho separadamente e dentro do Spriter Pro uniu-se às imagens e para criar efeito de brilho, utilizou-se transparência (*Alpha*).

2.4.5. Cenário

O cenário é todo plano de fundo do jogo que o jogador pode ou não interagir diretamente. Em Lumenide, sendo composto pelo chão, parede e objetos decorativos. É parte essencial do jogo, auxiliando a ambientar, a dar o clima e o estilo do jogo.

Figura 22 – O cenário de Lumenide

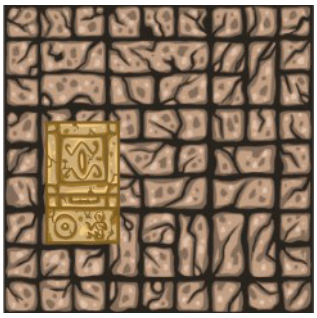


Como exemplificado na Figura 22, o cenário utilizou cores menos saturadas e com filtro azul no chão para dar a sensação de segundo plano e permitir que os objetos do primeiro plano se destacassem.

Um cenário mal planejado com detalhamento e cores incorretas deixa o jogo confuso e frustrante como evidenciado na Tabela 5.

As primeiras versões da parede e chão foram feitas sem muito conhecimento, o que resultou em cenário confuso, o que exigiu um retrabalho. Após estudos, a nova versão ficou harmônica e adequada.

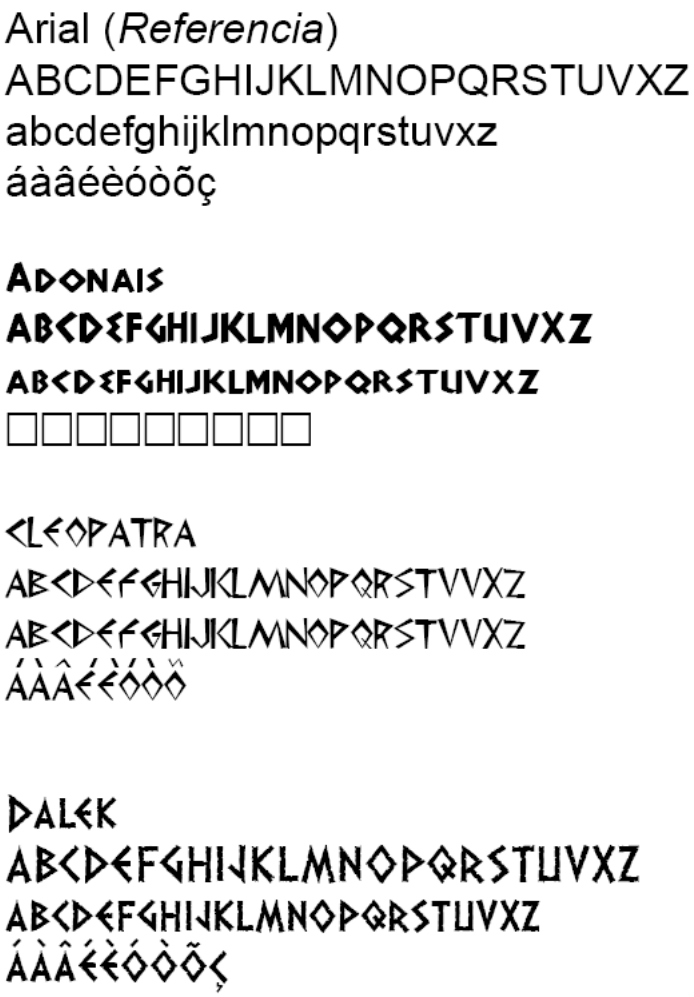
Tabela 5 – Exemplos de diferentes versões

	A primeira versão do cenário não possuia paredes e o chão foi desenhado com muitos detalhes e pintado com cores muito saturadas, o que resultou em um chão confuso, que se destacava mais que os objetos importantes ao jogo.
	A segunda versão do cenário já possua parede e chão, apesar de utilizar cores menos saturadas, o alto detalhamento e contraste de cores ainda deixava o cenário confuso e não ideal.
	A terceira versão do chão e parede utilizaram cores menos saturadas com filtro azul no chão para trazer para a tonalidade mais fria. Possui bastante detalhes, porém suaves e não contrastantes, não interferindo com o restante dos objetos, sendo um cenário ideal e harmônico.

2.5. Tipografia

A tipografia também influencia no visual do jogo. A depender do propósito, uma fonte pode ser adquirida pronta ou criada. Para Lumenide, a criação de uma fonte própria demandaria muito tempo, portanto, foi realizado uma pesquisa para procurar as fontes de livre uso mais adequadas ao jogo.

Figura 23 – Análise das fontes tipográficas



Após selecionar as fontes mais adequadas que estão ilustradas na Figura 23, o grupo decidiu que a fonte Dalek seria a mais apropriada. Para uso dentro do jogo, foi realizado um tratamento da fonte, para que ela tivesse uma borda, buscando assim maior destaque com outros elementos.

A fonte foi utilizada tanto em seu estado original como modificado com borda, a cor da fonte variou a depender do uso, mas sempre esteve dentro da paleta de cores proposta.

2.6. Programação

Para programação foi escolhida a linguagem C#, pois é a que temos conhecimento e facilidade para organização das funções e atributos. Foi utilizado o *software* ‘Visual Studio Community’, que é uma versão gratuita especial para o uso

no Unity. Ele auxilia na programação e acesso rápido as funções do *Unity,* além da depuração (*debug*) dos códigos e fácil detecção de problemas, sendo uma ferramenta mais completa que o *Monodevelop* que acompanha o *Unity*.

Foi definido que cada grupo de objetos teria um único *script*. Botões e as alavancas, por exemplo, são objetos do mesmo tipo, então compartilham um mesmo *script*, mas com os valores dos atributos diferentes. O objetivo dessa escolha é facilitar para o *Level Design*er a implementação ou criação de novos objetos.

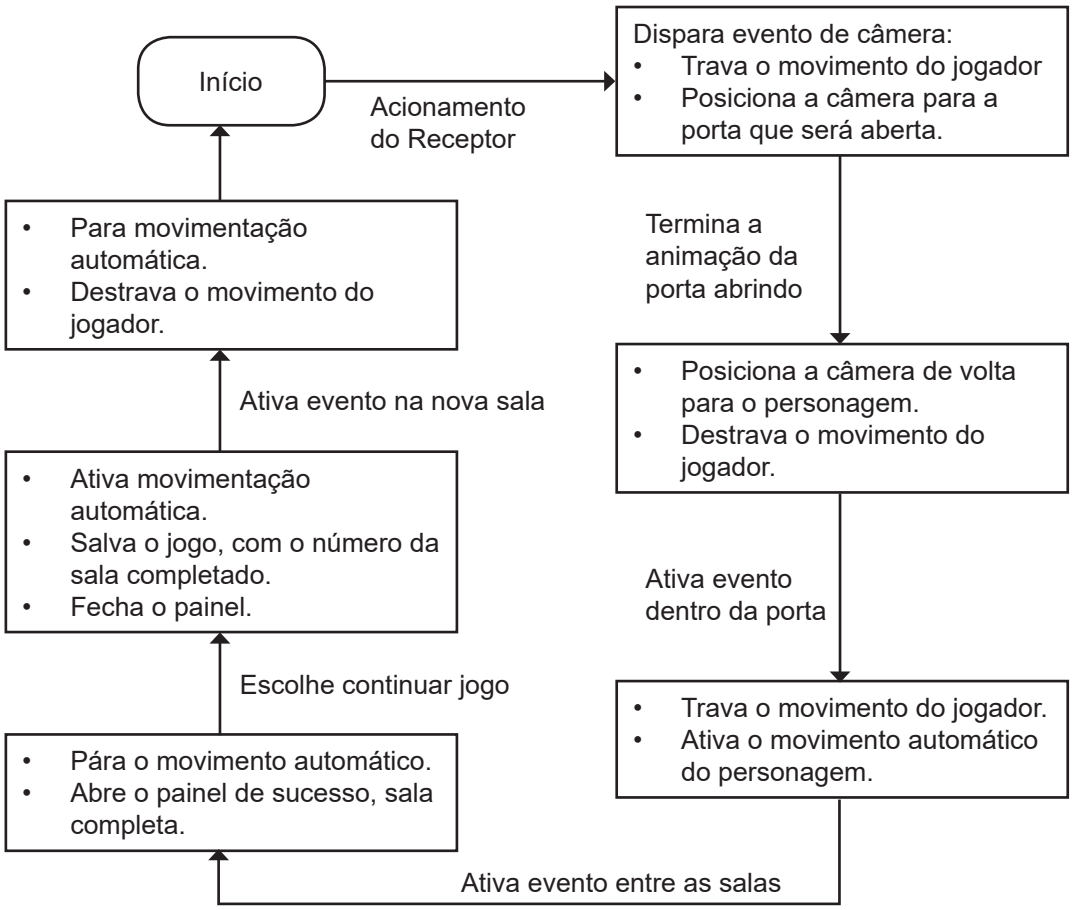
Para que algumas funções e atributos pudessem ser persistidos entre as cenas criadas, foram criados os quatro *Managers* abaixo. Estes *scripts* conterão os atributos e funções raízes do projeto e não serão apagados nas transições de cenas, como volume do som, por exemplo. Eles poderão ser acessados a partir de qualquer outro, para alteração ou registro.

- **Game Manager** (*GameManager.cs, Script 9 no Apêndice*)**:** Responsável por salvar e carregar o progresso e carregar novas cenas.
- **Sound Manager** (*SoundManager.cs, Script 20 no Apêndice*)**:** Controle de volume da música e efeitos sonoros.
- **Menu Manager** (*MenuManager.cs, Script 12 no Apêndice*)**:** Gerenciamento dos menus de janelas que o jogo possui, tentando generalizar as funções para todas as janelas.
- **Level Manager** (*LevelManager.cs, Script 10 no Apêndice*)**:** Responsável por controlar o fluxo das salas e dos níveis.

2.6.1. Fluxo Contínuo das Salas

Foi pensado um fluxo de eventos que determina o ciclo de uma sala de *puzzle*, e esse fluxo é reiniciado em toda sala. Ele define como e quando serão executadas as funções chaves para conclusão da sala. O esquema da Figura 24 ilustra este fluxo.

Figura 24 – Fluxo contínuo das salas



2.7. *Level design*

Também chamado de *design* de níveis, é a função de criação de ambientes, cenários e missões em um jogo eletrônico. Envolve a criação de mundos para praticamente qualquer gênero de jogo eletrônico e leva em conta a jogabilidade e dificuldade proposta ao jogador. Um novo nível pode introduzir um personagem, mecânica ou objeto, pode também ressaltar um ponto do roteiro do jogo ou até mesmo criar uma atmosfera mais envolvente para o jogador.

2.7.1. Conceitos e utilização

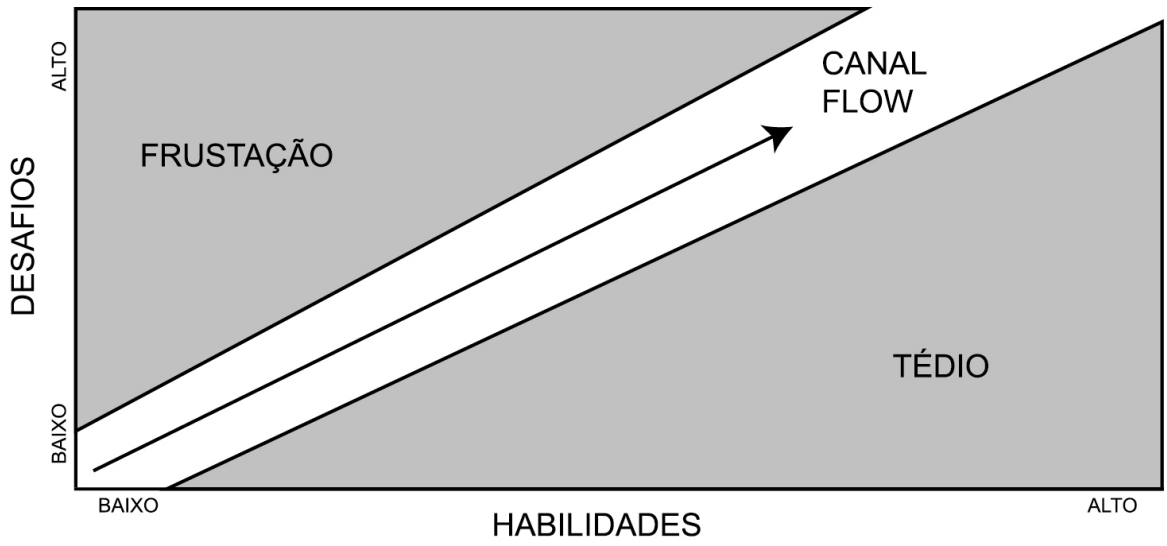
Segundo Novak (2010) os níveis podem ser utilizados para estruturar um jogo em algumas subdivisões, para organizar a progressão e aprimorar a experiência de jogo. Essas subdivisões, foram utilizadas em Lumenide como descritas a seguir.

As metas ou objetivos devem ser claros e compreensíveis para o jogador, apresentando uma sinalização de final de fase ou sala, ou mesmo apresentando algum *feedback* para o jogador saber que concluiu alguma tarefa específica. Em nosso jogo, o jogador pode acessar o mapa de nível e, a qualquer tempo, pode verificar sua sala atual e quantas salas faltam para o final do nível atual. Além disso, ao final de cada nível, um painel com a frase “Sucesso” é apresentada antes do carregamento da próxima sala, e o jogador pode identificar facilmente que aquela sala e seus objetivos foram alcançados.

Outro aspecto muito importante é manter o fluxo e aprendizado do jogo. Para isso, o jogador é contido a uma área do jogo, e não poderá avançar enquanto essa área não for completada. Em Lumenide isso é prontamente identificado, já que a porta para a próxima sala só é aberta após completar o *puzzle* ou desafio da sala, fazendo com que o raio de Sol seja direcionado corretamente para o receptor e a porta finalmente é aberta, levando à próxima sala.

É importante levar em consideração também a Teoria do fluxo, de Mihaly Csikszentmihalyi, que em seu livro **Flow: The Psychology of Optimal Experience** (CSIKSZENTMIHALYI, 1990) conceitua dessa forma: “Fluxo é um estado mental de operação em que a pessoa está totalmente imersa no que está fazendo, caracterizado por um sentimento de total envolvimento e sucesso no processo da atividade.” Essa teoria é muito aplicada em jogos, mesmo que não tenha sido desenvolvida com esse intuito, e pode ser exemplificada na Figura 25.

Figura 25 – Exemplo de um esquema da Teoria do Fluxo.



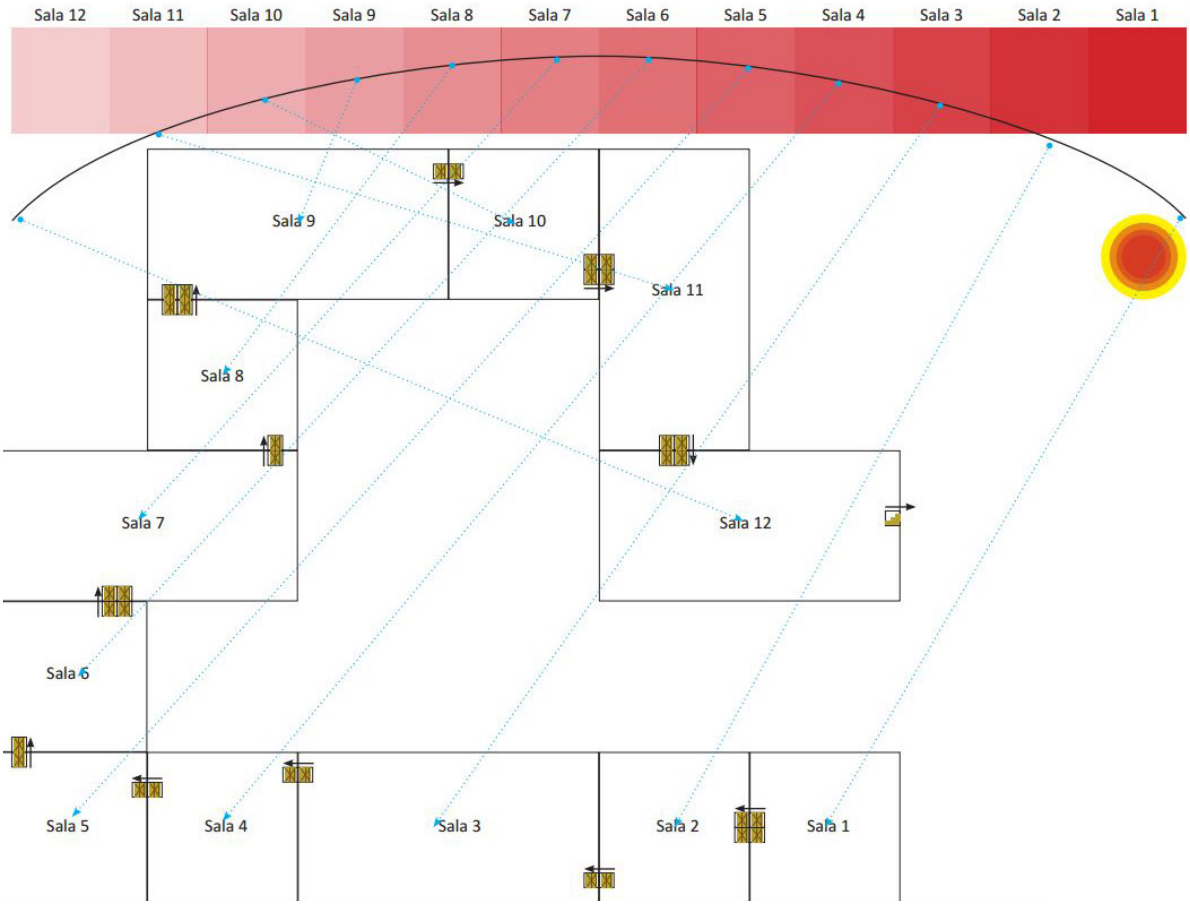
Se o jogador tiver um nível de habilidade baixo, e o desafio é muito alto, ele se frusta e deixa de jogar. Ao mesmo tempo, se suas habilidades forem altas demais para o desafio proposto o jogo se torna tedioso e repetitivo. O *Level Design* deve ser trabalhado dentro desses parâmetros. Em Lumenide, o *Level Design* foi idealizado utilizando a mecânica do jogo para funcionar como um tutorial, com enigmas bem simples no início e com um crescimento gradual, levando em conta a curva de aprendizado do jogador.

Quanto à duração do jogo, é importante notar que por ser casual, Lumenide não é limitado por tempo. Um jogador leva em média 5 minutos por sala, podendo sair do jogo a qualquer tempo e salvar a sua localização a cada sala já completada. Quando o jogador volta ao jogo, pode continuar de onde parou ou refazer alguma sala já completa. Além disso, o jogador deverá completar todas as salas, em uma ordem sequencial, não sendo possível começar o jogo sem antes passar pelas salas iniciais e completar as salas tutoriais do game. Isso condiz com o conceito de duração de jogo utilizado por Novak (2010), que diz que “Uma regra universal parece ser a de que o jogador deve concluir pelo menos um nível de qualquer game em uma única sessão”.

No conceito inicial, o tempo seria uma condição de premiação dada ao jogador que conseguisse terminar a sala em um tempo menor que a média, ou até mesmo mais rápido do que a última vez que jogou. Mas isso não foi implementado e é uma ideia para o jogo final.

Cada sala do Lumenide é independente das outras, mas contribue para o conhecimento das mecânicas que serão utilizadas em salas posteriores. Portanto, devem ser solucionadas de modo sequencial, da primeira até a décima segunda. Não existem relações diretas entre elas, a não ser as mecânicas e objetos que serão utilizadas durante todo o percurso da fase. Uma das ideias do conceito inicial, que foi parcialmente aproveitada, é a de que cada sala seria uma hora do dia e, em cada hora, uma incidência do Sol em um ângulo específico. A sala 1, por exemplo, seria 6h ou 7h da manhã e a última sala, a décima segunda, seria 18h ou 19h completando assim um dia de 12 horas de Sol. O conceito é demonstrado na Figura 26.

Figura 26 – Conceito inicial de incidência do Sol em cada sala, seguindo um relógio com 12 horas de Sol.



Esse conceito foi utilizado para o mapa de salas, como uma relação direta entre as salas, e também para indicar onde o raio de Sol entra na sala e é direcionado pelo redirecionador de cada sala. Entretanto, o conceito não foi além disso, mas serviu como uma boa referência para localização e relação entre as salas.

Lumenide é um jogo linear, e cada novo objeto ou mecânica deve ser utilizado para completar a fase atual, funcionando assim como um tutorial. Se o jogador não conseguir utilizar corretamente o novo objeto/mecânica apresentada, não é possível passar para a nova sala, tornando a utilização desse novo objeto/mecânica obrigatória. Isso leva em conta também a curva de aprendizado e dificuldade, que é crescente e vai aumentando conforme o jogador avança pelo nível.

O jogo utiliza uma visão em perspectiva isométrica em duas dimensões, onde o jogador pode observar toda a sala em um ângulo ligeiramente inclinado (por volta de 45º). Dessa forma é possível visualizar o alcance dos raios de Sol que serão refletidos nos espelhos bem como a melhor solução para o *puzzle* da sala. A câmera segue

o personagem horizontalmente no eixo X e verticalmente no eixo Y, dependendo de onde a sala está localizada. Entretanto, a câmera fica travada na sala atual, não sendo possível visualizar nada além dessa sala, mesmo quando o personagem está em um dos cantos.

Sobre o terreno apresentado, utilizamos materiais relacionados ao mundo real, caixas, espelhos, botões, alavancas, mas com uma proporção muito diferente dos objetos físicos, para facilitar a manipulação desses elementos pelo personagem, já que o jogo é em plataforma *mobile*. Até mesmo o personagem tem a sua proporção distorcida, com a cabeça quase do mesmo tamanho do corpo inteiro.

2.7.2. Metodologia de *Level design* e aplicação

Foi utilizado um *software* de *design*, não específico para jogos chamado Adobe InDesign, especificando espaços quadriculados e tamanho de salas definidos. Nesse programa, foram criados os enigmas e *puzzle*s, seu funcionamento e solução. Só depois, com essa referência em mãos, os objetos foram incluídos e ajustados no Unity.

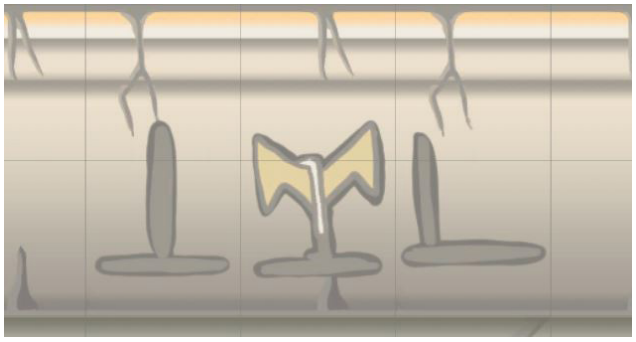
O método utilizado, foi de solução inversa, da solução para o problema. O *puzzle* é montado, já solucionado, do modo mais intricado possível e depois são colocados obstáculos para dificultar que o jogador chegue a essa solução. A sala é preparada de forma a dificultar a solução do problema, levando o jogador a pensar na melhor solução possível, dentro de suas habilidades e mecânicas aprendidas até então.

É possível verificar um aumento gradativo na dificuldade de cada sala, levando em conta a curva de aprendizagem e a Teoria do Fluxo (CSIKSZENTMIHALYI, 1990), até chegar à última sala, onde o inimigo final da fase é encontrado. Nesta sala, existem 3 soluções para atingir o inimigo, cada uma pode ser feita a qualquer tempo e é anulada logo após ser usada. O inimigo, atingido, bloqueia o raio de Sol com um pilar (coluna de pedra) e essa solução é invalidada. Desta forma, o jogador deve solucionar os 3 quebra-cabeças em qualquer tempo, não podendo utilizar a mesma solução para atingir o inimigo várias vezes. Após os 3 acertos, o inimigo é derrotado e a fase é completada. Esta sala será melhor detalhada em “3.12.2. Exemplo de salas”.

2.7.3. Outros elementos de *Level Design*

Foram utilizados elementos gráficos de decoração nas salas, sem ligação ou influência nenhuma nas mecânicas do jogo. Esses elementos são puramente decorativos, como por exemplo o sarcófago da primeira sala, alguns tipos diferentes de vasos e escrituras nas paredes. Estas escrituras nas paredes, são também utilizadas na identificação das salas secretas, quando estão em uma sequência específica. Entretanto, isso não será apresentado ao jogador, e apenas jogadores experientes conseguirão perceber a ligação entre estes elementos e a localização da sala secreta (Figura 27). Essas escrituras representam que a sala contém um sala secreta, e onde ela está localizada. As duas primeiras gravuras demonstram a existência de uma sala secreta, e a última gravura representa onde está localizada (abaixo, acima, à esquerda ou direita).

Figura 27 – Gravura de sala secreta



Nestas salas secretas o jogador encontra pequenos fragmentos da história do personagem, exemplo de sala secreta na Figura 28, e a cada fragmento encontrado, uma parte da história é liberada e pode ser acessada do menu do jogo. O jogador pode terminar o jogo sem encontrar nenhuma sala secreta, já que elas só podem ser acessadas se os espelhos forem posicionados em um ângulo correto, a Figura 29 mostra uma sala secreta acessível.

Figura 28 – A sala secreta é menor que uma sala normal, e contém apenas um pedestal com o fragmento de história

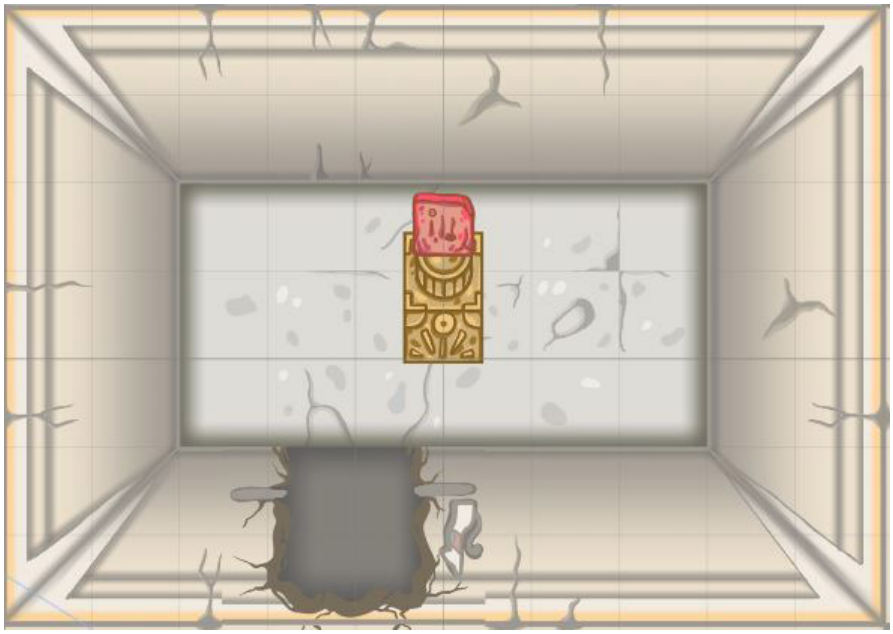


Figura 29 – Abertura da porta secreta

Para auxiliar o jogador, há tutoriais para as novas situações, toda vez que o jogador entra em uma fase e é o primeiro contato com a mecânica, uma janela é aberta automaticamente, na qual há o gato RanRan que dá uma explicação sobre o que deve ser feito. A lista de tutoriais pode ser acessada a qualquer momento através de um botão de ajuda, simbolizado por um ponto de interrogação, que faz parte da HUD.

2.8. Áudios

O áudio em jogos digitais ajuda a ambientar o jogador, e criar uma atmosfera interessante no jogo. Os elementos gráficos atraem o jogador para a cena, mas é o áudio que exerce o maior efeito imersivo. Um jogo sem sons, são só imagens sem referência ao mundo real e portanto, sem imersão.

Apesar da grande importância do áudio para os jogos digitais, em nosso grupo não há ninguém especializado em sons, por isso utilizamos bibliotecas públicas, sem restrições de uso, como a biblioteca de áudios e músicas do YouTube, por exemplo.

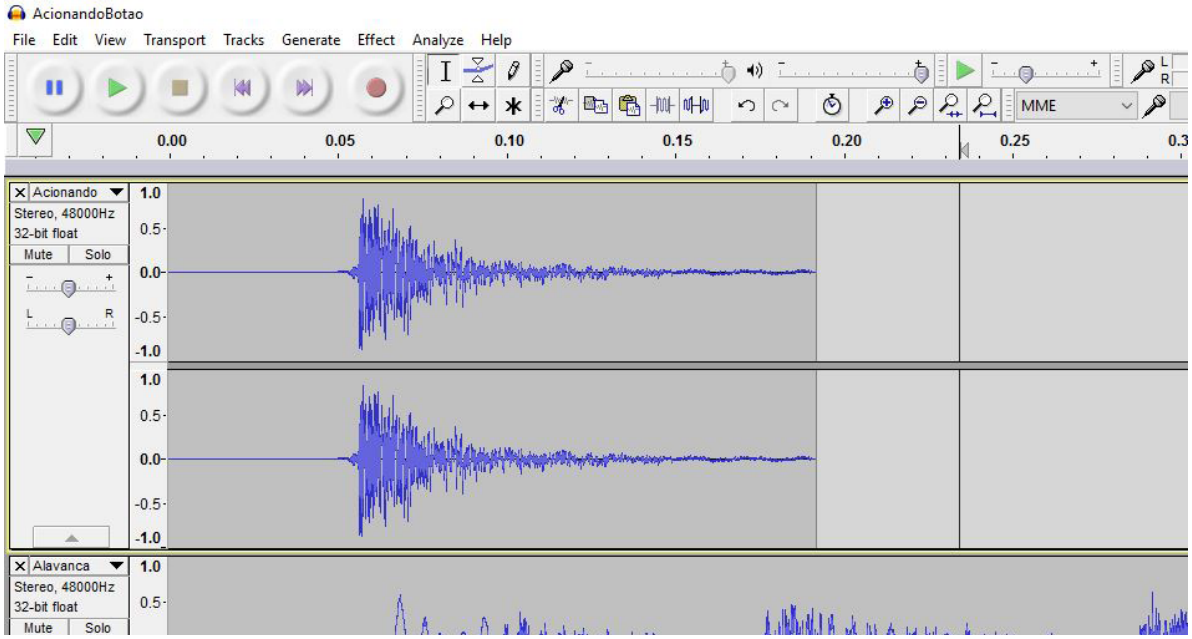
2.8.1. Músicas

Utilizamos apenas duas músicas em Lumenide, ambas da biblioteca de áudios do YouTube, uma delas mais agradável de se ouvir, e em loop infinito chamada ‘Belief’, que é a música tema do jogo. Ajuda na imersão do jogador na cena, e traz sons que remetem à solidão de estar preso dentro da pirâmide.

A outra música utilizada foi na sala 11 e na última sala, a sala do chefe final da fase, também de uso livre e gratuito, tem como título *Curse of the Scarab e foi* disponibilizada por Kevin MacLeod. Esta música de ação foi usada para criar uma atmosfera de perigo, já que nessas duas últimas salas o jogador pode realmente morrer, se atingido pelos projéteis inimigos. Essa trilha tenta passar essa ideia de emoção e suspense, e melhorar a sensação de imersão do jogador.

Foi utilizado o programa gratuito Audacity (Figura 30) para fazer as mixagens das trilhas e exportar para o formato MP3 e Ogg, utilizável no Unity e de tamanho reduzido para facilitar o download.

Figura 30 – Programa utilizado para editar as músicas e outros sons.



2.8.2. Efeitos sonoros

Existem muitos efeitos sonoros em Lumenide, e eles são realmente importantes, tanto para a imersão do jogador quanto para o *feedback,* para indicar que algo está realmente acontecendo.

Alguns exemplos são:

- **Arraste de caixas e objetos:** Só são reproduzidos quando o jogador exerce uma força nestes objetos, movendo-os para alguma direção.
- **Feixes de luz:** Fazem um som muito baixo, quase imperceptível; como a luz não faz barulho, o efeito é como uma máquina ligada, quase um zumbido, mas também ajuda na imersão.
- **Portas:** Os sons das portas se abrindo e fechando são muito importantes, já que é um dos objetivos principais de cada sala, e é importante que o jogador saiba que a porta se abriu e ele pode avançar para a próxima sala. O som da porta se fechando também tem a sua importância, já que é um caminho linear e sem volta.
- **Alavancas e botões:** O som para estes objetos é essencial, porque fornece um *feedback* para o jogador. Se não houvesse esse som, o jogador poderia não saber ao certo se a alavanca/botão foi acionado.

Os sons de efeitos sonoros (SFX) foram quase todos editados usando o mesmo programa, Audacity. Alguns efeitos foram também adicionados para modificar som e adicionar efeitos para o som finalizado:

- **Fade In e Fade Out:** Suaviza a entrada e saída de sons.
- **Compressor:** Utilizado neste projeto para modificar alguns sons, colocar mais peso e complexidade.
- **Bass and Treble:** Também foi utilizado para modificar sons, e deixá-los mais interessantes
- **Volume:** Os áudios foram normalizados para a mesma altura. O volume é alterado no Unity.
- **Change Speed:** Mudar a velocidade do som, alguns sons não são específicos para a ação.

Alguns exemplos de sons utilizados, que foram editados são os de arraste para objetos móveis, que originalmente é um som de passos, o da porta se abrindo que é um som de pedras caindo invertido, e o giro de espelhos, que são espelhos se quebrando, com edições para parecer que estão apenas sendo movidos.

3. IMPLEMENTAÇÃO

Seguindo o cronograma da Tabela 2, pôde-se observar que conforme a arte foi adicionada ao projeto, foi preciso ajustar os *sprites* corretamente, antes de ser utilizada em cena, criar o funcionamento e mecânicas dos *prefabs*, e organizar o projeto com um todo. A programação foi usada desde o início, para configurar o funcionamento de objetos e mecânicas que serão usados em cena, estes passos são explicados e exemplificados como segue.

3.1. Implementação de *sprites*

Para implementar os *sprites* na Unity foi preciso configurá-los. Como foi definido no início do projeto que o tamanho dos *sprites* seria com base de 128x128 pixels, foi preciso marcar em cada imagem que o tamanho por unidade no programa seria 128 pixels. Também foi necessário configurar os pivôs dos *sprites* com a opção *Bottom*, para que possa ser implementado um controle de *layer*, que define se o personagem está atrás ou na frente do objeto dentro da cena.

3.2. Movimentação do Personagem

Foi utilizado um *Asset* para controle em dispositivos de toque ou com mouse chamado *CNControls* (NADEZHDIN, 2016). Ele possui *joystick* virtual e botões para serem usados com o toque do dedo para controlar e executar ações do personagem. Assim, no *script* de movimentações podemos acessar a classe *CNManager* e verificar as coordenadas do *joystick* e acionamentos dos botões em tela.

No *script ActiveButton.cs, Script 1 no Apêndice,* existe uma função chamada ‘Movement’ que utiliza essas coordenadas, a horizontal (x) e vertical (y), para incrementar a posição do personagem. Com as coordenadas do *joystick*, criamos um vetor de direção que será multiplicado pela velocidade e pelo tempo. Assim incrementando-se a posição do jogador e fazendo ele movimentar-se, no final da função é armazenada a última coordenada horizontal e vertical, para quando o personagem parar de se

movimentar, sabermos para que lado o personagem fica virado e aplicarmos a animação correta.

Nesse mesmo *script* temos um detector de colisão, com o qual é verificado se o personagem está colidindo com uma caixa ou espelho móvel. Caso esteja, é utilizada a animação do personagem empurrando objetos e adicionado um *collider* novo, com o tamanho do comprimento dos braços, assim evitando que estes atravessem os objetos ao empurrá los pela lateral.

Para programação das animações do personagem foi utilizado o componente *Animator* (Figura 31), com a passagem de parâmetros para controle dos estados da animação de movimento e empurrando caixa. Para facilitar o uso do componente, usamos o estado chamado *Blend Tree* (Figura 32), componente geralmente utilizado para controle de animações de movimento de personagem utilizando as coordenadas passadas pelo *joystick*.

Figura 31 – Componente *Animator* do Unity.

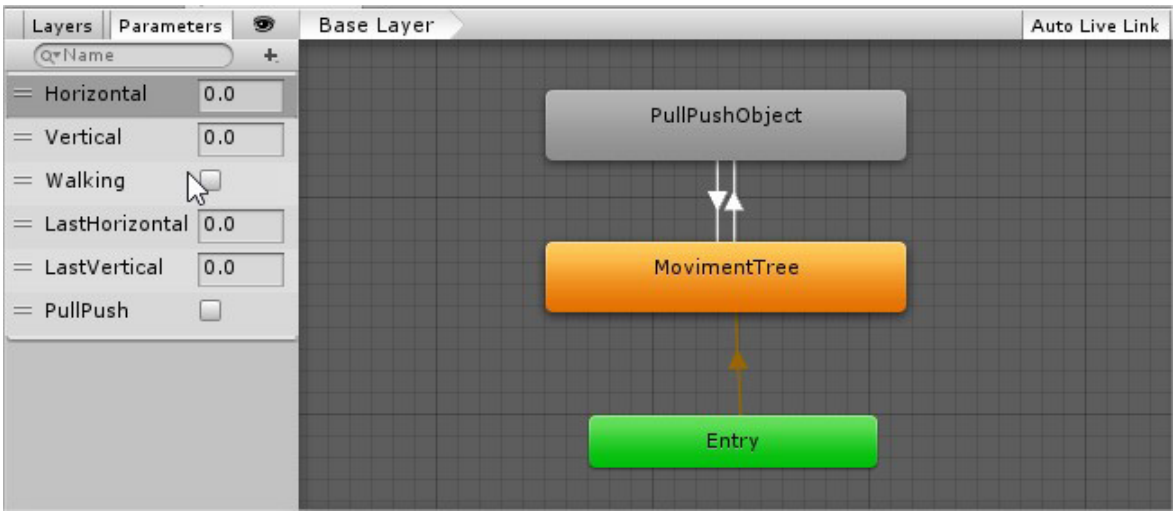
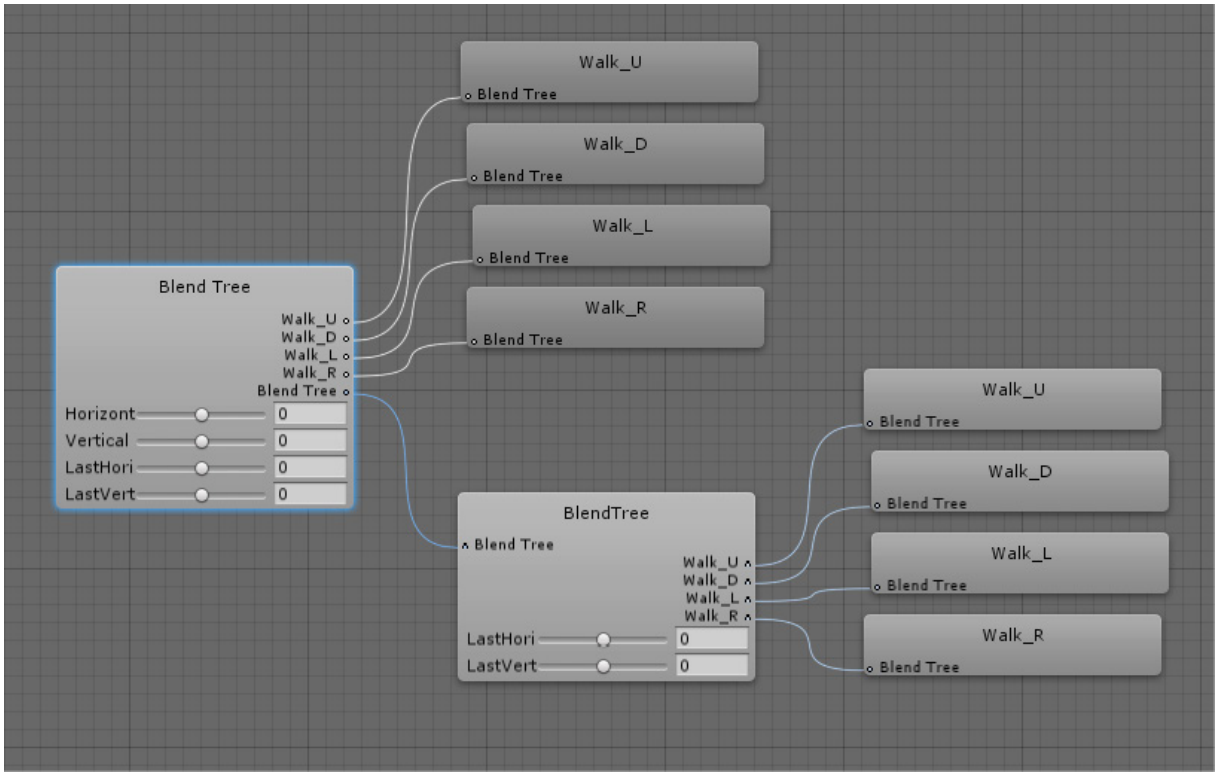


Figura 32 – Componente *Blend Tree* do Unity.



Todas os parâmetros são controlados pelo *script MovimentPlayer.cs, Script 13 no Apêndice* na função *Animation*, mudando os estados da *Figura 31*

3.3. Espelhos e Raios de Luz

Todos os espelhos do projeto usam o mesmo *script*, *ReflectionSumRay.cs, Script 17 no Apêndice*. Ele controla tanto a reflexão da luz quanto a posição em que o espelho está, ficando possível girar o espelho sem que precise de um outro *Prefab*.

No objeto espelho temos um *collider* que irá detectar colisão com raios de luz.

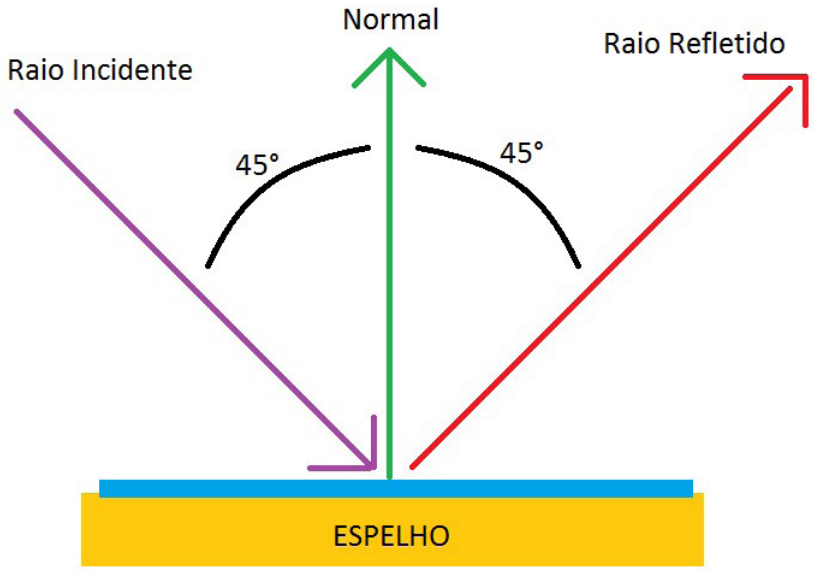
Os raios de luz são *Prefabs* que possuem o componente *Line Renderer* que são instanciados na cena, traçando-se uma linha para uma determinada direção definida pelo cálculo de reflexão total.

3.4. Fórmula Reflexão Total

Tendo como base que a normal é uma reta projetada pela superfície do espelho e o raio incidente é aquele que colide com o espelho, temos que encontrar o raio

refletido, que deve ter o mesmo ângulo que o raio incidente tem para a reta normal, conforme a Figura 33.

Figura 33 – Ilustração de Reflexão total.



Sejam $\overrightarrow{v_i}$ o vetor na direção do feixe de luz incidente e $\overrightarrow{v_n}$ o vetor unitário na direção normal à superfície refletora. Para uma situação como a da Figura 33, com a normal perpendicular à direção horizontal, é fácil obter as componentes do vetor na direção do feixe de luz refletido, $\overrightarrow{v_r}$:

$$\overrightarrow{v_i} = \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \overrightarrow{v_n} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \Rightarrow \overrightarrow{v_r} = \begin{bmatrix} x_i \\ -y_i \end{bmatrix}$$

Porém, os espelhos podem estar orientados em outras direções no jogo. É necessário obter uma fórmula de reflexão válida para todos os casos. A solução é escrevê-la num referencial que satisfaça as condições acima. Para isto, realiza-se uma rotação por um ângulo θ_e (inclinação do espelho em relação à horizontal) através da aplicação de uma matriz de rotação em duas dimensões:

$$\overrightarrow{v_i}' = \begin{bmatrix} cos(\theta_e) & -sen(\theta_e) \\ sen(\theta_e) & cos(\theta_e) \end{bmatrix} \times \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Invertendo-se o sinal da componente vertical temos o vetor do feixe refletido:

$$\overrightarrow{v_r}' = \begin{bmatrix} x_i cos(\theta_e) - y_i sen(\theta_e) \\ -x_i sen(\theta_e) - y_i cos(\theta_e) \end{bmatrix}$$

Retornando para o referencial original com uma rotação de $-\theta_e$, temos:

$$\overrightarrow{v_r} = \begin{bmatrix} x_i[cos^2(\theta_e) - sen^2(\theta_e)] - 2y_i sen(\theta_e)cos(\theta_e) \\ -2x_i sen(\theta_e)cos(\theta_e) + y_i[sen^2(\theta_e) - cos^2(\theta_e)] \end{bmatrix}$$

Os valores de $sen(\theta_e)$ e $cos(\theta_e)$ podem ser obtidos através de $\overrightarrow{v_n}$:

$$\overrightarrow{v_n} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} \Rightarrow sen(\theta_e) = x_n, cos(\theta_e) = y_n$$

Portanto, o vetor do feixe refletido em função das componentes dos vetores do feixe incidente e normal ao espelho é dado por:

$$\overrightarrow{v_r} = \begin{bmatrix} x_i(y_n^2 - x_n^2) - 2y_i x_n y_n \\ -2x_i x_n y_n + y_i(x_n^2 - y_n^2) \end{bmatrix}$$

Quando o espelho detecta um raio de luz, é usada a posição de origem do raio para determinar a direção incidente e vamos utilizá-la no cálculo para determinar a direção de reflexão total que o novo raio de luz será instanciado.

Para definir onde o raio de luz colide, foi preciso usar *RayCast*, uma função própria do Unity, que emite um raio invisível em uma determinada direção e que possui a propriedade de detectar com quem ele colide, assim pode-se saber no código com qual objeto o raio está colidindo. Dessa maneira conseguimos alterar o ponto de destino da luz emitida para quando um objeto passa pela frente. Assim, foi preciso fazer verificações se o trajeto da luz que já havia colidido com vários espelhos não foi encerrado por algum objeto, ou até mesmo o personagem. Para isso foi criada uma classe chamada *SunRay.cs, Script 22 no Apêndice,* que aplica a direção no *Prefab* de luz e controla com quem ele colidiu e se é preciso emitir outro raio por ter colidido com outro espelho. Foi criada uma função no *script ReflectionSunRay*, que verifica recursivamente todos os objetos de luz criados a partir de onde a emissão foi interrompida, assim destruindo todos os *Prefab* instanciados da cena que contemplava aquele trajeto, conforme ilustrado na Figura 34.

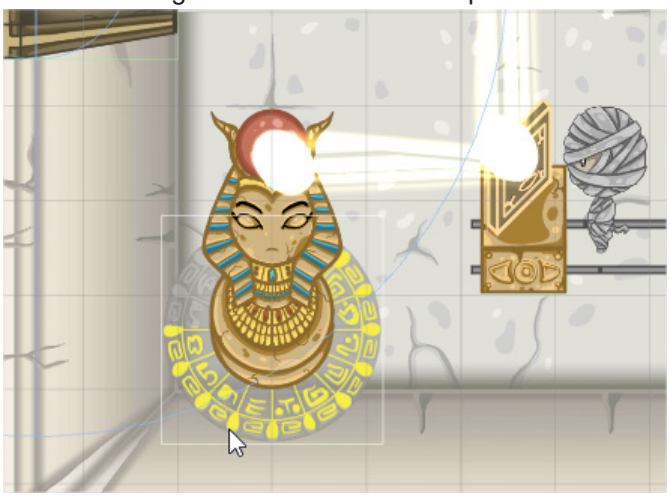
Figura 34 – Interrupção da Luz.

3.5. Receptor e Emissor de Luz

O *ReceptorLuz.cs, Script 16 no Apêndice* possui um *collider* para detectar a colisão da luz do Sol. Ao detectar a luz é ativado um *Prefab* que possui a classe *TimerUI.cs, Script 24 no Apêndice*, e que, ao ser ativado executa um *timer* enquanto a luz está no Receptor; caso a luz seja interrompida, o *timer* é desativado.

O *timer* é uma imagem que é preenchida conforme o tempo passa. A imagem só aparece enquanto estiver ativado o *Prefab*, conforme Figura 35.

Figura 35 – *Timer* do Receptor.



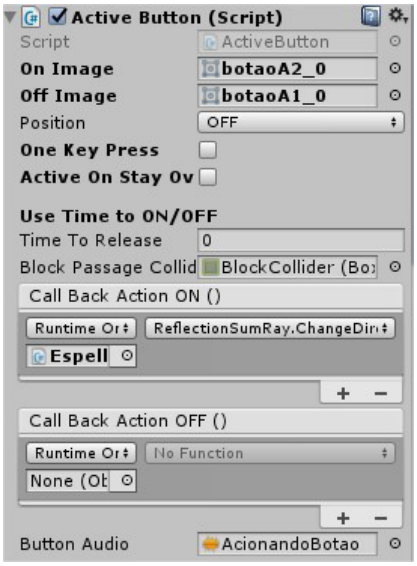
O *script ReceptorLuz* fica usando uma função pública do *TimerUI* chamada *isDone*, que verifica se o tempo foi concluído. Ao ter o retorno *true*, é executada a função atribuída no *Inspector*, para quando for ativado.

Para a fonte de luz foi criado um *Prefab* utilizando *SunRayEmitter.cs, Script 23 no Apêndice,* assim foi possível mudar as características de luz diretamente no *Inspector*, como por exemplo: a distância máxima que a luz percorre, o ângulo de saída e selecionar em qual camada, ou *layer*, a luz se colidirá em cena. Para receber o primeiro raio emitido, foi criado um *Prefab* que utiliza *ReflectionSumRay.cs, Script 17 no Apêndice,* com o tipo de reflexão 'Redirecionador', a luz é então orientada para a direção em que os espelhos atuam, agilizando o desenvolvimento de *puzzles* no *level design*.

3.6. Botões e Alavancas

Foi criado um único *script* chamado *ActiveButton.cs, Script 1 no Apêndice,* para contemplar os botões e alavancas. Nesse *script* são controlados pelo *Inspector* (Figura 36) os *sprites* de ON/OFF, qual a posição inicial do objeto, se é ativado/desativado ao pressionar um botão na tela, ou se ao pressionar o botão ativa e ao soltar o botão desativa, se vai ser ativado ao passar por cima. Além desses parâmetro que podem ajudar na diversificação do objeto, ele pode também escolher funções que serão executadas ao ativar ou desativar um botão.

Figura 36 – Script ActiveButton.



3.7. Portas e Eventos

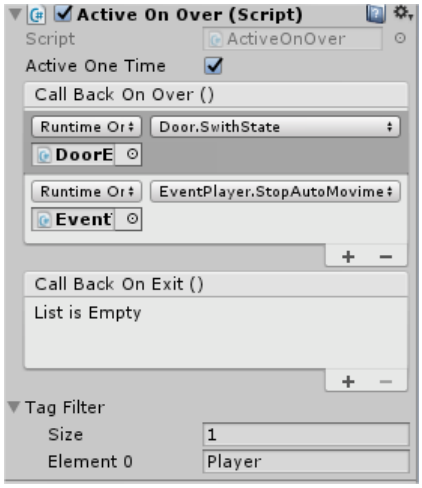
No *script Door.cs, Script 6 no Apêndice*, é controlada uma máscara, que fica por cima do sprite da porta, que é para quando o jogador passar pela porta dar a impressão de que ele está passando por baixo. Isso é controlado de acordo com a animação em um *script* de máquina de estado chamado *DoorAnimationBehavior.cs, Script 7 no Apêndice*, onde no *script Door* são atribuídas funções pelo *Inspector* que serão executadas pelo *DoorAnimationBehavior* ao detectar que a animação iniciou ou terminou.

Há uma função pública chamada *SwicthState*, que é usada no fluxo de salas para poder abrir e fechar a porta quando requisitado. Essa função tanto abre quanto fecha a porta, ela verifica o estado anterior e troca, executando animação e habilitando o *collider* para impedir a passagem do jogador, se necessário.

Os Eventos são basicamente *prefab*s vazios, que possuem um *collider* e os *scripts EventPlayer.cs, Script 8 no Apêndice* e o *ActiveOnOver.cs, Script 2 no Apêndice*.

ActiveOnOver possui um verificador que detecta objetos na cena que entram e saem do *collider*, podendo ser filtrados pela *tag* do objeto preenchida em uma lista no *Inspector*, como mostrado na Figura 37.

Figura 37 – *Inspector* ActiveOnOver.



Com as funções de *CallBack OnOver* e *OnExit*, pode ser usada qualquer função pública de outro objeto.

O *EventPlayer* possui uma função de parar/movimentar o personagem automaticamente para uma direção definida no *Inspector*, assim bloqueando os controles, e outra função para parar a movimentação automática devolver o controle ao jogador.

3.8. Câmera

Para a câmera seguir o jogador, ela possui um *script* chamado *CameraFollow. cs, Script 4 no Apêndice.* No *inspector* pode ser aplicado um *Target,* objeto alvo que a câmera vai seguir, e os limites da sala e a velocidade em que a câmera vai se locomover.

LevelLimits é um *Prefab* criado com o *script CamLimitBounds.cs, Script 5 no Apêndice.* Esse *script* é usado para definir os limites de cada sala no *level design*, para que a câmera fique enquadrada dentro da sala. Para facilitar, é possível ver um desenho com linhas vermelhas no Editor do Unity com os limites definidos no *Inspector* de largura e altura.

3.9. IA e Projéteis

Foi desenvolvido o *script BossAnimation.cs, Script 3 no Apêndice,* para realizar a movimentação do chefe de nível pela cena, no qual foi criado um atributo chamado *MoveSet*, que é selecionado pelo *Inspector* sobre qual será a movimentação adotada. Foi desenvolvida o tipo **LEFT_RIGHT**, onde o chefe de nível anda para a esquerda e a direita trocando a direção ao colidir com a parede.

Para os projéteis do chefe de nível foi usado o mesmo *script* dos canhões utilizado na sala 11, chamado *ReceptorLuz.cs, Script 16 no Apêndice*. Nele deve ser aplicado pelo *Inspector* o componente que vai fornecer a posição para instanciar os projéteis, o *Prefab* do projétil, velocidade de disparo, tempo entre os disparos e direção dos mesmos.

3.10. UI

Para controlar os painéis no *canvas*, foram criados alguns *scripts* para situações diferentes.

Para o painel de pausa foi criado um *script* para o botão de *pause* chamado *PauseMenu.cs, Script 15 no Apêndice*. Nele é controlado o painel de pausa e o de sair. Existem funções públicas para abrir e fechar os painéis.

Os painéis da tela inicial são controlados pelo *PainelBehavior.cs, Script 14 no Apêndice*, que contém funções públicas que servem para abrir e fechar paineis.

No painel de seleção de níveis foi criado um *Prefab* com *StageController.cs, Script 21 no Apêndice*, onde se pode escolher o número da sala/nível em uma lista de objetos. Cada objeto deste tipo utiliza o *Sala.cs, Script 18 no Apêndice*, neste script é controlado se a sala está habilitada e qual o seu número correspondente.

No momento em que o painel de seleção é aberto o *StageController* é realizada uma consulta para verificar qual foi a última sala que jogador esteve, bloqueando assim as outras salas onde o jogador ainda não tem acesso.

3.11. *Managers*

Na câmera foi adicionado um *script* chamado *Loader.cs, Script 11 no Apêndice* que ficou responsável por carregar todos os *managers*. Ao inicializar a cena ele verifica se existe instância dos *managers*; se não tiver instância os *Prefabs* dos *managers* para que sejam usados por todas as cenas. Todos os *managers* possuem uma função chamada *Awake* que, antes da cena inicializar, verifica se existe instância da classe e a marca para não ser destruída ao trocar de cena usando a função do Unity chamada *DontDestroyOnLoad*.

GameManager.cs, Script 9 no Apêndice possui algumas funções que serão usadas em todas as cenas. Uma delas é a que salva e carrega o progresso do jogador. Para o progresso do jogador ser salvo, foi criada uma classe em *SaveGame.cs, Script 19 no Apêndice,* que possui todos os atributos necessários para armazenar em qual

nível e sala o jogador já passou, configuração de som, lista de tutoriais já vistos e a lista de relíquias já encontradas.

Utilizamos o **JSON** *(JavaScript Object Notation),* que é um tipo de estrutura de dados em *javascript* que pode ser transformada facilmente em texto e novamente em dados. O Unity possui uma classe específica para isso, chamada *JsonUtility*, que permite transformar qualquer classe em JSON.

Para armazenar estas informações foi utilizada uma classe própria do Unity chamada *PlayerPrefs*, com ela é possível salvar informações para conservá-las em um arquivo de texto. Usando o *JsonUtility,* a classe *SaveGame* foi convertida em JSON, e suas informações guardadas no *PlayerPrefs.* Para que os dados armazenados possam ser localizados facilmente, é necessário também definir uma chave de indexação, que é também utilizada na recuperação destes dados.

Para recuperar as informações é só fazer o caminho inverso, usando a chave de indexação para resgatar os dados e convertê-los novamente na classe *SaveGame*, podendo novamente ser consultada pela função de *Load* do *GameManager*.

Outra função do *GameManager*, é carregar as cenas e realizar um suavização entre elas, assim deixando a transição mais suave.

Já o *SoundManager* cuida da parte dos sons do jogo. Ele possui um componente do Unity chamado *Mixer* que controla todos os áudios do jogo, podendo mudar o volume de todos os objetos durante o jogo. Essa classe é requisitada somente quando a configuração de sons é alterada pelo jogador.

MenuManager possui somente uma função genérica, para abrir e fechar os painéis.

LevelManager cuida das salas. Essa classe possui funções para que assim que o nível é carregado, desativar as salas onde o personagem não está, liberando memória de processamento. Também controla a transição de uma sala para outra, habilitando a nova sala e desabilitando a anterior.

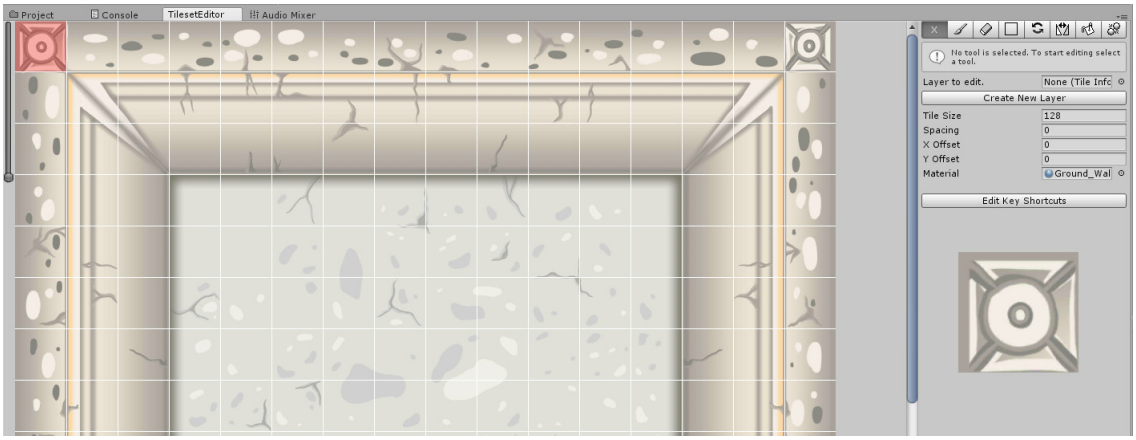
3.12. *Level Design*

Com o documento de *Level Design* definido (como explicado em “2.7.2. Metodologia de Level design e aplicação”), o processo de implementar as salas da fase no Unity foi simples.

3.12.1. *Design* das salas

As salas foram criadas com o auxílio de um *plugin* para Unity chamado *TileSetEditor (Figura 38)*, que auxilia na composição do cenário, utilizando os pixels como *tiles* agrupados. O processo é o seguinte: é criado uma camada, ou *layer*, nova do tamanho desejado, no caso das salas comuns o tamanho, em *tiles*, foi de 22 x 16, cada *tile* utilizando 128 pixels. Desses, apenas 16 x 10 foram utilizados como espaço útil, já que cada canto da sala tem 3 tiles utilizados como paredes e decoração. Ainda com o auxílio do *plugin*, os blocos foram pintados como uma tela quadriculada.

Figura 38 – *Plugin* TilesetEditor, utilizado para facilitar a implementação do layout das salas.



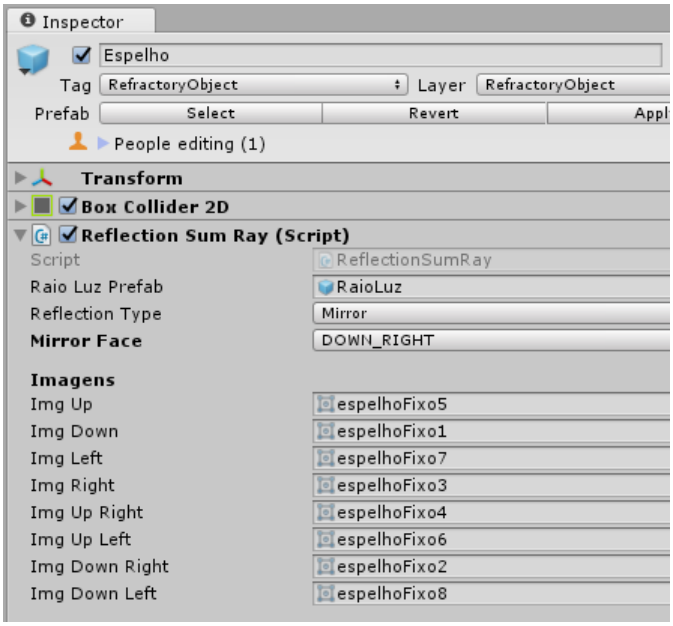
Depois da sala ajustada, cada componente foi adicionado em seu lugar específico e organizado utilizando *Empty GameObjects*, hierarquicamente como pode ser visto na Figura 39.

Figura 39 – Captura de tela do Unity, demonstrando como foram organizados os componentes utilizados.

Cada componente também segue a sequência em que deve receber e rebater o *LineRenderer* do raio de Sol. Isso ajuda muito na solução de problemas, já que o componente pode ser facilmente encontrado na cena.

Outra ótima funcionalidade programada por *script* foi a possibilidade de ajuste no próprio *Inspector* do Unity, do ângulo dos espelhos. Com isso, o ajuste dos espelhos no ângulo certo na cena pôde ser feita rapidamente, sem a necessidade de um novo *Prefab* para cada componente de ângulo diferente, isso pode ser verificado na Figura 40.

Figura 40 – *Inspector* do Unity, funcionalidade de mudança do ângulo de reflexão programada por *script*.



3.12.2. Exemplo de salas

Salas 1 e 2 (Figura 41): estas salas foram utilizadas como um tipo de tutorial. O jogador aprende a arrastar os espelhos dentro de um trilho pré estabelecido. Não há muito o que fazer nestas salas, já que, se o jogador não resolver o desafio, não poderá continuar para a sala seguinte, as salas já implementadas podem ser vistas na Figura 42.

Figura 41 – Salas 1 e 2, esquema utilizado.

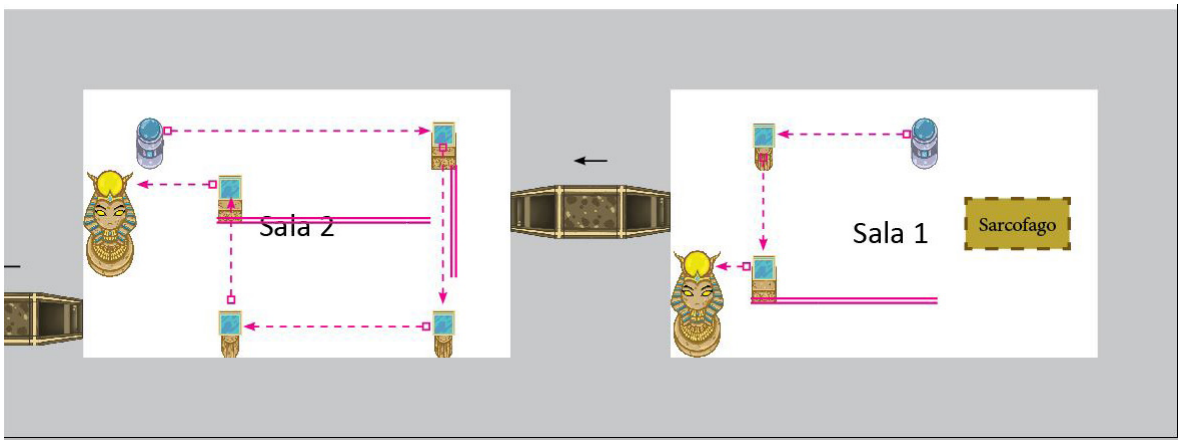
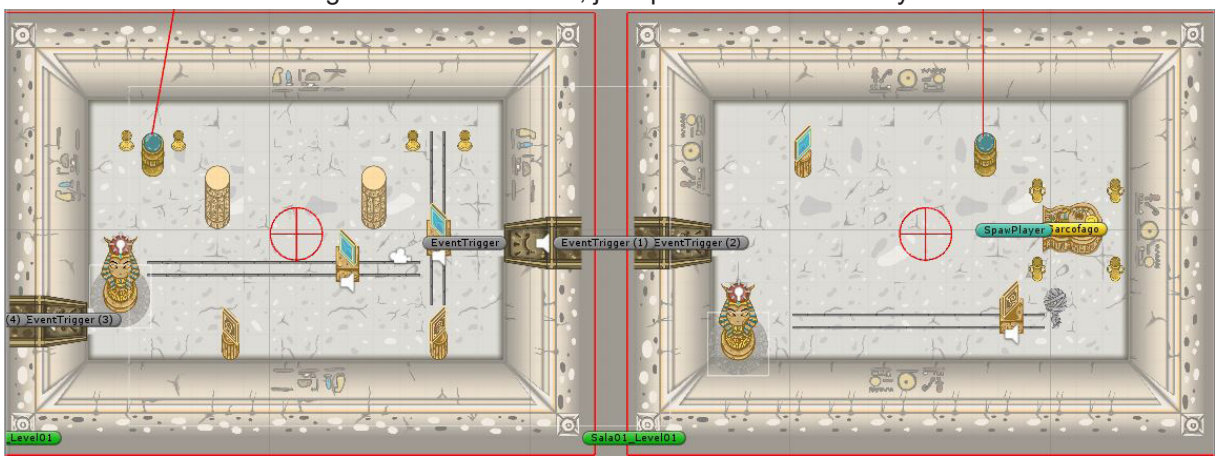


Figura 42 – Salas 1 e 2, já implementadas no Unity.



Sala 8 (Figura 43): uma sala mais complexa, perto do final da fase. Aqui o jogador já deve conhecer a maior parte dos elementos e mecânicas do jogo, e a sala finalizada no Unity na Figura 44.

Figura 43 – Sala 8, esquema utilizado.

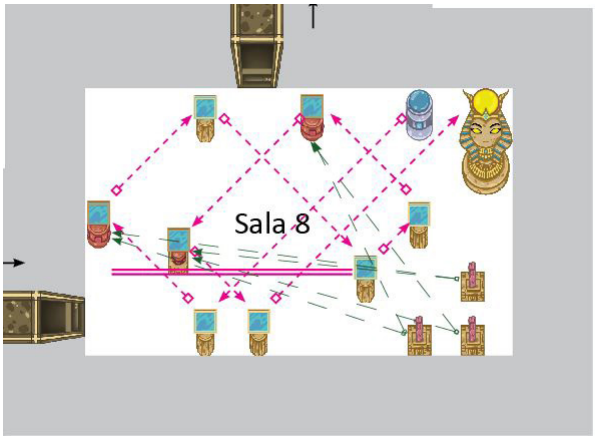


Figura 44 – Sala 8, já implementadas no Unity.

Sala 11, prévia do poder de Sokar (Figura 45): aqui é apresentado o primeiro elemento nocivo ao jogador, dois canhões que atiram projéteis contra o personagem e que, se atingí-lo, a sala é reiniciada. A implementação é mostrada na Figura 46.

Figura 45 – Sala 11, esquema utilizado, com canhões.

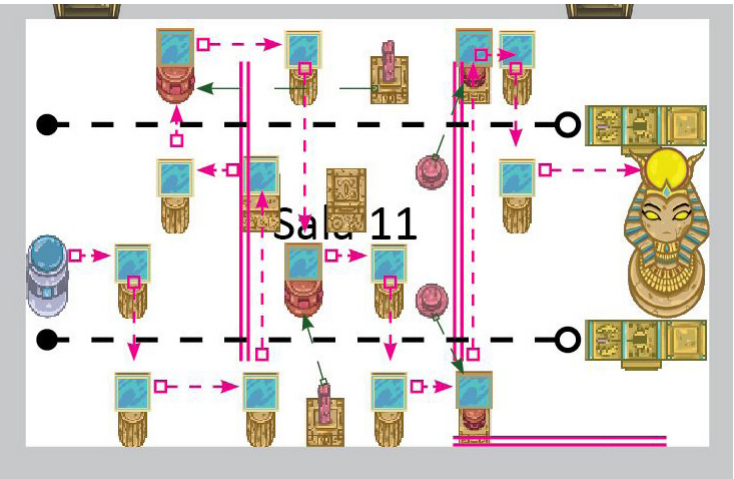
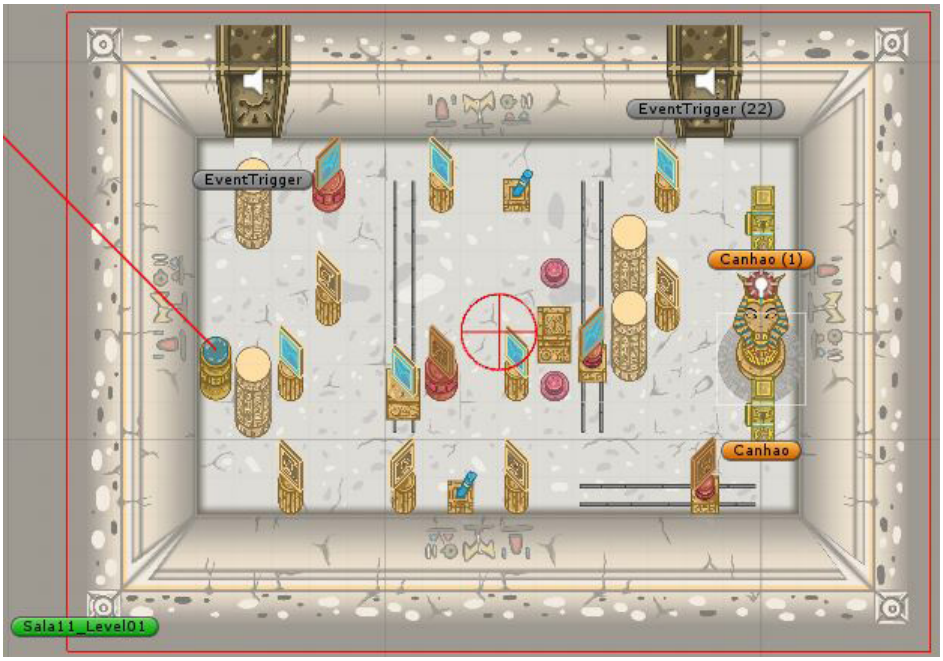
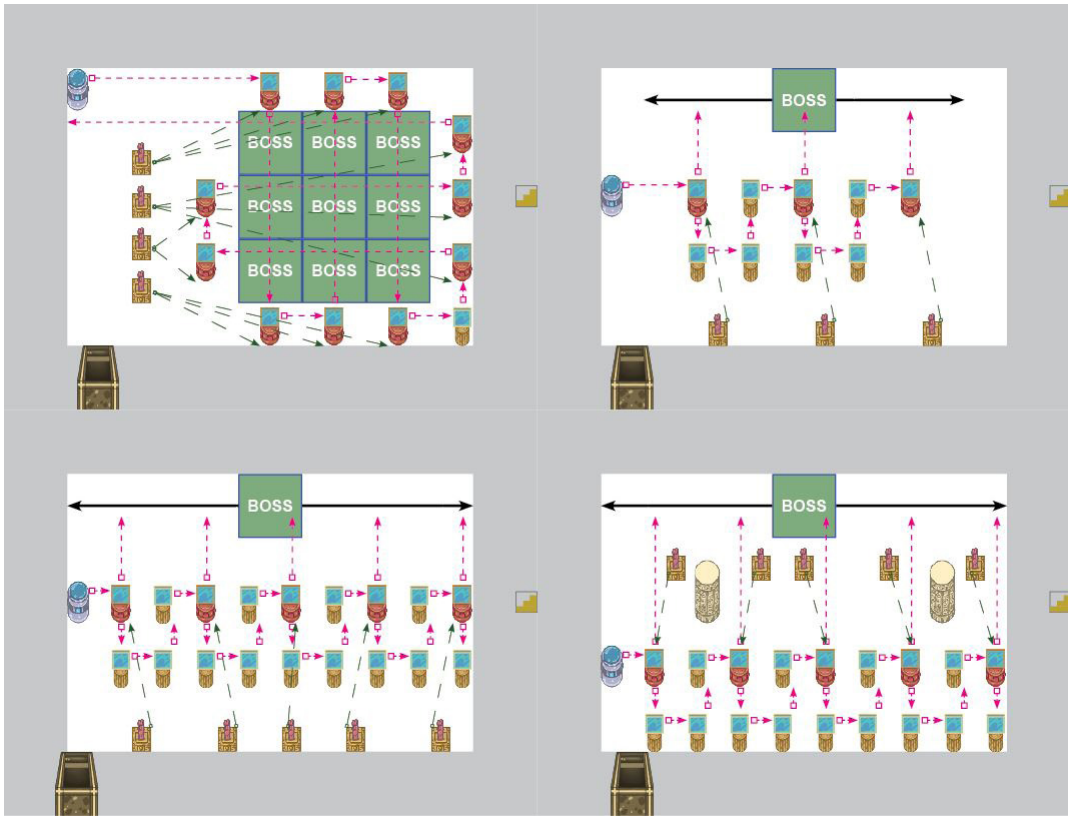


Figura 46 – Sala 11, já implementadas no Unity.



Sala 12, sala do chefe de final de fase: esta sala foi bem complicada de ser confeccionada, muitas ideias foram propostas, como pode ser visto na Figura 47.

Figura 47 – Testes preliminares de *design* para a sala de Sokar, o chefe da fase.



A ideia selecionada foi a de três desafios independentes, Figura 48, que podem ser executados a qualquer tempo dentro da sala, que atingem o inimigo com o raio de Sol. A cada acerto, a solução é bloqueada pelo inimigo, não podendo mais ser utilizada. O jogador deve resolver os 3 para conseguir vencê-lo e passar para a próxima fase. A sala já implementada ficou como demonstrado na Figura 49.

Figura 48 – Sala 12, esquema utilizado, com cada uma das três soluções demonstradas separadamente.

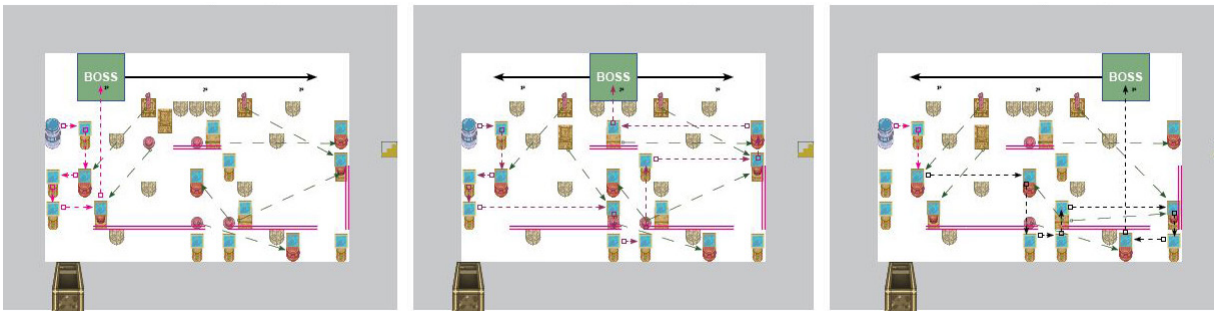


Figura 49 – Salas 12 e o chefe da sala, já implementada.

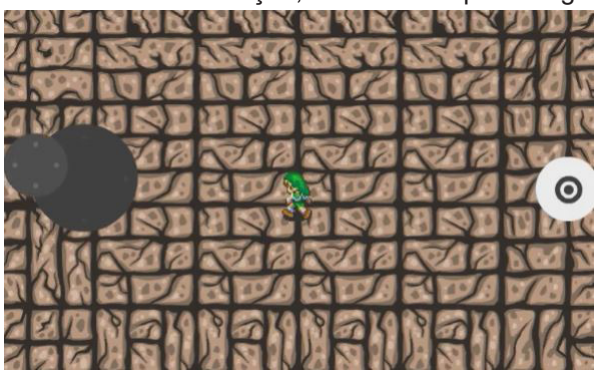
4. RESULTADOS

O jogo foi testado com sucesso, por várias pessoas. Desde a primeira versão, exportada como .APK, até a nova e mais atualizada versão, disponível na PlayStore do Google. Todas as versões foram muito importantes para identificação e correção de problemas, teste de compatibilidade e adaptação de melhorias e mecânicas.

4.1. Protótipo

A primeira versão jogável foi testada apenas pelos desenvolvedores. Nesta primeira versão foi testada a movimentação do personagem (não tinha sido desenhado ainda), artes de pisos, e algumas mecânicas iniciais, como pode ser visto na Figura 50.

Figura 50 – Teste de movimentação, ainda sem o personagem principal.



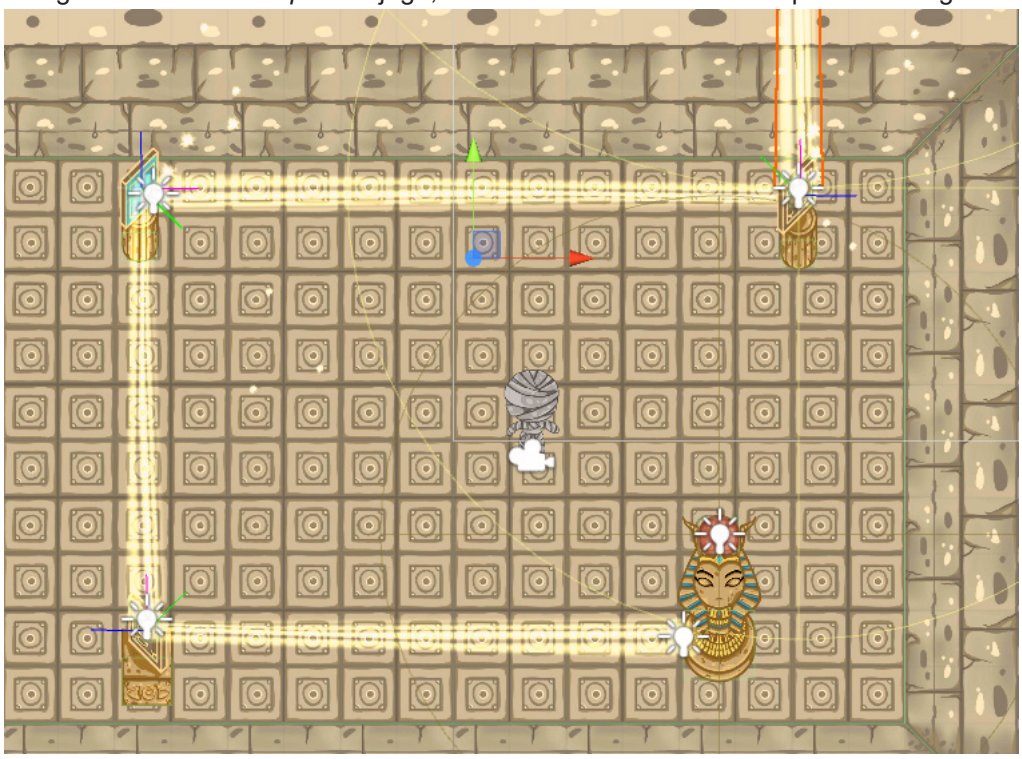
4.2. Versão *Alpha*

A versão *Alpha* do jogo, Figura 51, foi distribuída apenas por .APK entre amigos e colegas da **FATEC.** Alguns *bug*s foram detectados e a arte atualizada para melhorar o contraste. Os objetos até essa versão eram movimentados livremente, mas isso foi problemático porque o ângulo para reflexão não ficava alinhado corretamente. Foi desenvolvido então um trilho, onde alguns objetos devem permanecer e só são movimentados nesse trilho. Isso ajudou a limitar a movimentação dos espelhos.

Outra funcionalidade adicionada foi um *timer* para o receptor de raio de Sol. Antes dessa implementação, só pelo fato do raio passar pelo receptor a porta se abria e quando o personagem passava na frente do raio, a porta se fechava novamente. Agora, com o *timer*, o raio precisa estar direcionado por alguns segundos para abrir a

porta, mas, após aberta, o jogador pode transitar livremente pela fase sem o problema da porta de saída fechar novamente.

Figura 51 – Versão *Alpha* do jogo, ainda com os *tiles* de chão e paredes antigas.



4.3. Versão *Beta*

Foi a primeira versão a ser publicada na Google PlayStore, e depois atualizada em 4 de junho de 2017, com novas correções de *bug*s como uma versão finalizada do jogo apresentado. Nesta versão *Beta* foi adicionada a movimentação automática entre salas, após a porta ser aberta. Isso resolveu o problema de o jogador não ver o personagem quando ele passava de uma sala para a outra.

Nessa versão também foram adicionados os primeiros sons e músicas, sendo aprimoradas, ainda na versão *Beta*, em um novo *update*. Os menus do jogo também foram atualizados, além do salvamento automático da posição do personagem no mapa, salvando a cada sala. Assim, o jogador não precisa começar desde a primeira sala, todas as vezes que sair do jogo.

4.4. Google PlayStore

O jogo foi adicionado oficialmente na Google PlayStore em 15 de maio de 2017, em uma conta disponibilizada gratuitamente pelo professor Gustavo Gomes, da **FATEC Americana**. Até o momento da finalização deste documento, foram feitos quase 120 *downloads* e instalações e 48 pessoas desinstalaram o jogo. Neste número não estão contabilizados os .APKs distribuídos pelos desenvolvedores na **FATEC Americana** e outra formas de distribuição *online*.

Por se tratar de um jogo não finalizado, e com praticamente nenhuma divulgação externa, a não ser o Facebook, a recepção foi muito boa, o que sugere um nicho muito interessante para novos jogos deste estilo e jogabilidade.

Figura 52 – Tela do painel de estatísticas do jogo publicado. (Data da captura: 04/06/2017).

Como pode ser verificado na Figura 52 o maior pico foi na época de divulgação do jogo, e o grande número de desinstalações foi provavelmente causado por se tratar de um jogo ainda não finalizado e curto para jogar. Além de ainda conter muito *bugs* para implementação. Espera-se que com a nova atualização, e o jogo já finalizado, os *donwloads* aumentem significativamente.

4.5. IV Mostra de Jogos

O jogo foi um dos 10 selecionados, dentre 26 outros jogos, para a ***IV Mostra Acadêmica de Jogos Digitais da Fatec Americana***, no dia 16 de maio de 2017. O vídeo do *gameplay* enviado para a seleção foi postado no Youtube para avaliação.

A disputa foi difícil, já que todos os 10 jogos selecionados tinham grandes qualidades e inovações. Eram três categorias de premiação (Figura 53) e Lumenide ficou em primeiro lugar em duas dessas categorias.

Figura 53 – Categorias de premiação da IV Mostra

Em **Escolha de Público**, foi nomeado por meio de votação pelos espectadores da mostra, no dia da apresentação. Isso mostra a boa recepção do público pela ideia, arte e jogabilidade do jogo, e aponta uma aposta popular de investimento.

O segundo prêmio em que o jogo foi premiado neste dia foi o de **Escolha de Mercado**, onde uma banca especializada, composta por profissionais do mercado de jogos da região, escolheu Lumenide como a melhor aposta de mercado, o que mostra que nosso jogo realmente tem potencial para lançamento.

A terceira categoria foi a de **Escolha Acadêmica**, de responsabilidade dos professores da **FATEC Americana**. O jogo escolhido foi *Aedes Revenge*, mas ainda assim Lumenide ficou entre os melhores, também nessa categoria.

Foi uma grande e agradável surpresa para os desenvolvedores, já que o jogo não estava ainda finalizado e em teste *Beta*, mas foi possível comprovar que estamos no caminho certo e que o jogo pode realmente ser um sucesso de mercado.

Além disso, recebemos ótimos *feedback*s da banca julgadora e do público em geral, como por exemplo, o fato de nosso jogo estar jogável e quase em um estágio de lançamento, enquanto alguns outros ainda estavam em fases iniciais de desenvolvimento. Também quanto à escolha da plataforma *mobile* para nosso jogo, por se tratar de um jogo casual, possui partes completáveis curtas e salvamento automático para cada sala, o que é muito interessante para jogos *mobile* casuais.

4.6. Pesquisa de opinião

Efetuamos também uma pesquisa de opinião com os jogadores que baixaram e instalaram o jogo, e o resultado é apresentado a seguir.

4.6.1. Formulário de pesquisa

O formulário foi elaborado com o auxílio do professor e orientador Bruno Daniel, e o tempo total da pesquisa foi de 6 dias, do dia 23 de maio até o dia 29 de maio de 2017. Contendo as seguintes perguntas e alternativas:

- **Qual a sua idade?** Pergunta aberta, mas obrigatória;
- **De onde você é (região, estado ou país)?** Pergunta aberta, mas obrigatória;
- **Como ficou sabendo do nosso jogo?** Obrigatória, com as sugestões: Facebook, IV Mostra da FATEC Americana, Amigos e colegas e Outros;
- **Por quanto tempo você jogou Lumenide?** Obrigatória, com as sugestões: Não joguei, Até 10 minutos, Até 20 minutos, Mais de 30 minutos e Mais de 1 hora;
- **O que você achou da jogabilidade (experiência) do jogo?** Obrigatória, com notas de 1 a 5, sendo 5 Ótima;
- **O que você achou das artes do jogo?** Obrigatória, com notas de 1 a 5, sendo 5 Excelente;
- **O que você achou da dificuldade do jogo?** Obrigatória, com notas de 1 a 5, sendo 5 Muito difícil;
- **Até qual sala você chegou?** Obrigatória, de 1 a 12;

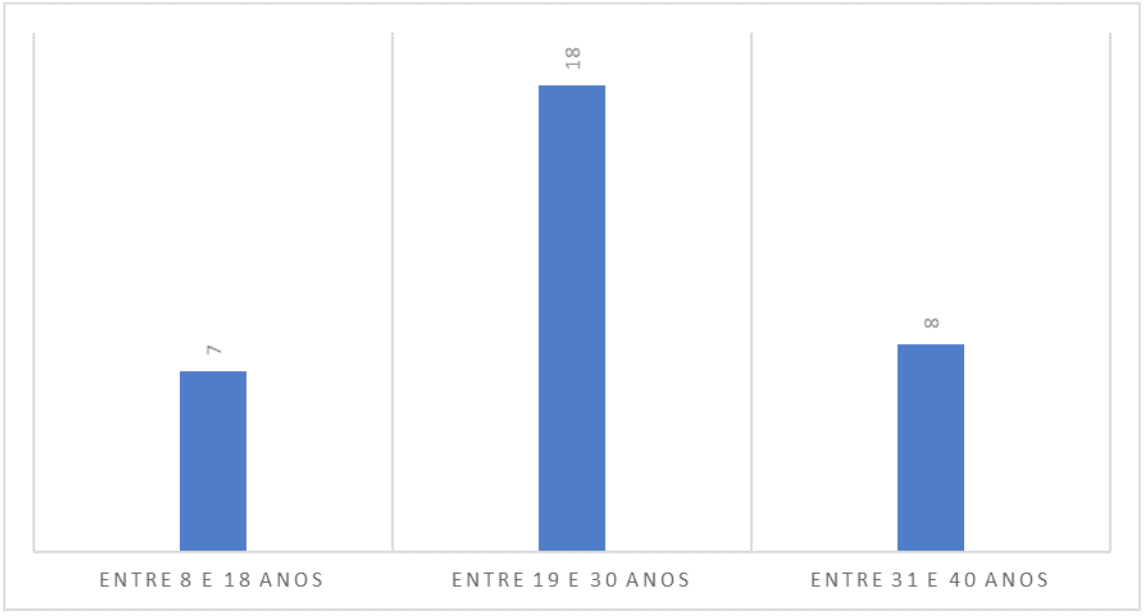
- **Você jogaria por mais tempo se houvesse uma versão final, e mais completa do jogo? Mais fases, mais salas, etc.** Obrigatória, com as sugestões: Sim, Talvez e Não;
- **Quanto você pagaria por esse jogo?** Obrigatória, com as sugestões: Não jogaria novamente, só jogaria se fosse gratuito, até R$2,50, até R$5,00, até R$10,00 e até R$20,00;
- **Você encontrou algum problema que gostaria de reportar, para que fosse corrigido em uma nova versão?** Pergunta aberta e opcional, para *feedback*;
- **Por favor, nos diga o que você achou do projeto como um todo (elogio, sugestões, críticas, etc)** Pergunta aberta e opcional.

4.6.2. Resultados

O total de respostas foi satisfatório. Levando em conta que por volta de 100 pessoas instalaram o jogo, o total de resposta foi de 30% desse número, o que foi um ótimo resultado já que a pesquisa foi divulgada apenas pelo Facebook e entre amigos dos desenvolvedores.

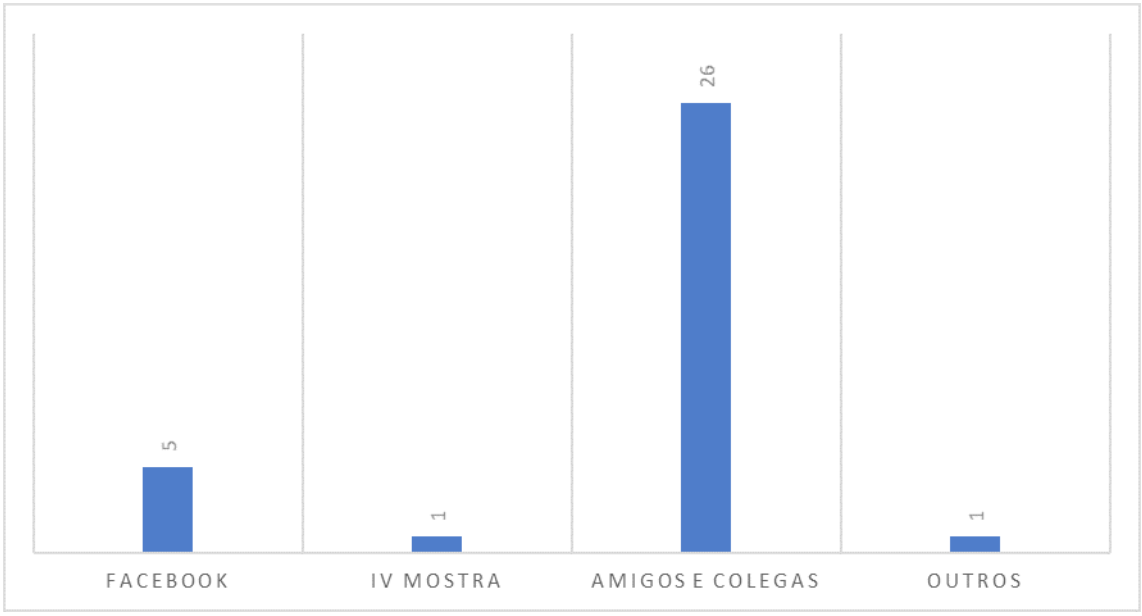
Nas figuras: Figura 54, Figura 55, Figura 56, Figura 57, Figura 58, Figura 59, Figura 60, Figura 61 e Figura 62, serão apresentados os resultados obtidos com a pesquisa *online*.

Figura 54 – Idade dos participantes



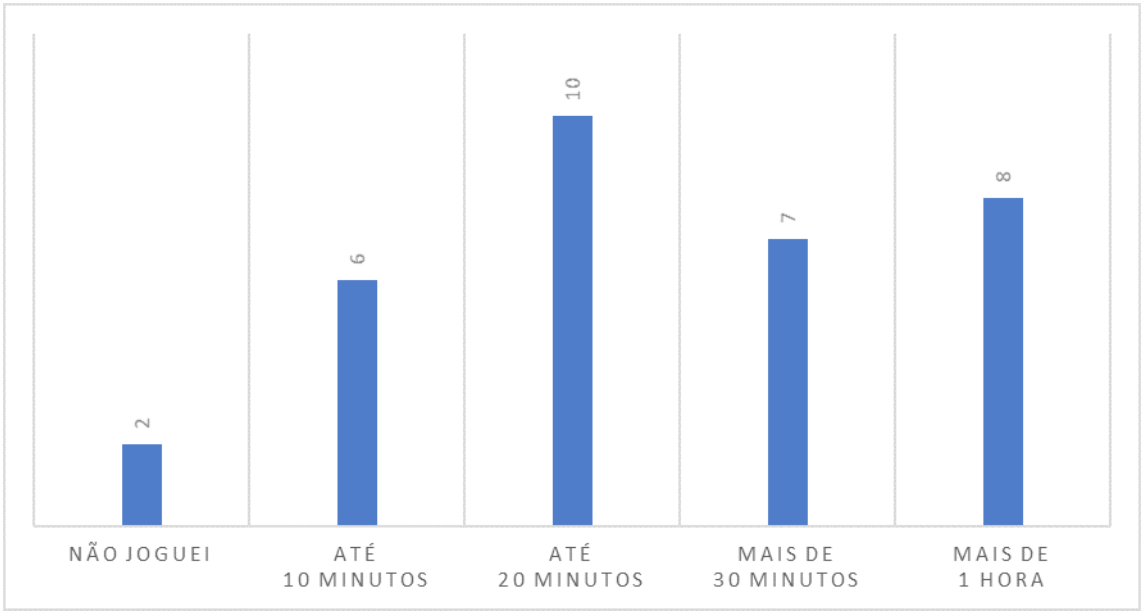
A média de idade foi de 25 anos, o que já era esperado pois a pesquisa só foi divulgada na **FATEC** e entre amigos e colegas dos desenvolvedores.

Figura 55 – Como os participantes ficaram sabendo sobre o jogo



A grande maioria, 75%, como era esperado, é de conhecidos dos desenvolvedores. Entretanto, teve uma pessoa que conheceu o jogo na IV Mostra Acadêmica de Jogos, o que mostra uma ótima oportunidade para apresentação de novos jogos na Mostra e a importância desse evento.

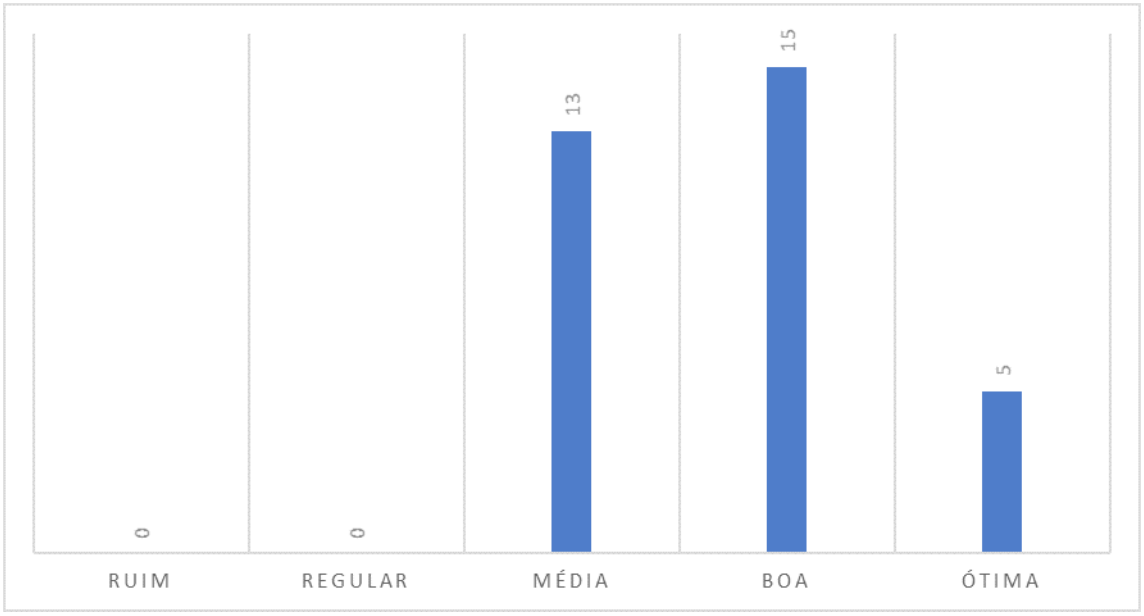
Figura 56 – Tempo de jogo dos participantes



25% das pessoas jogaram por mais de 1 hora. Por ser um jogo ainda não finalizado isso foi surpreendente. Além disso, mais de 50% das pessoas jogaram por

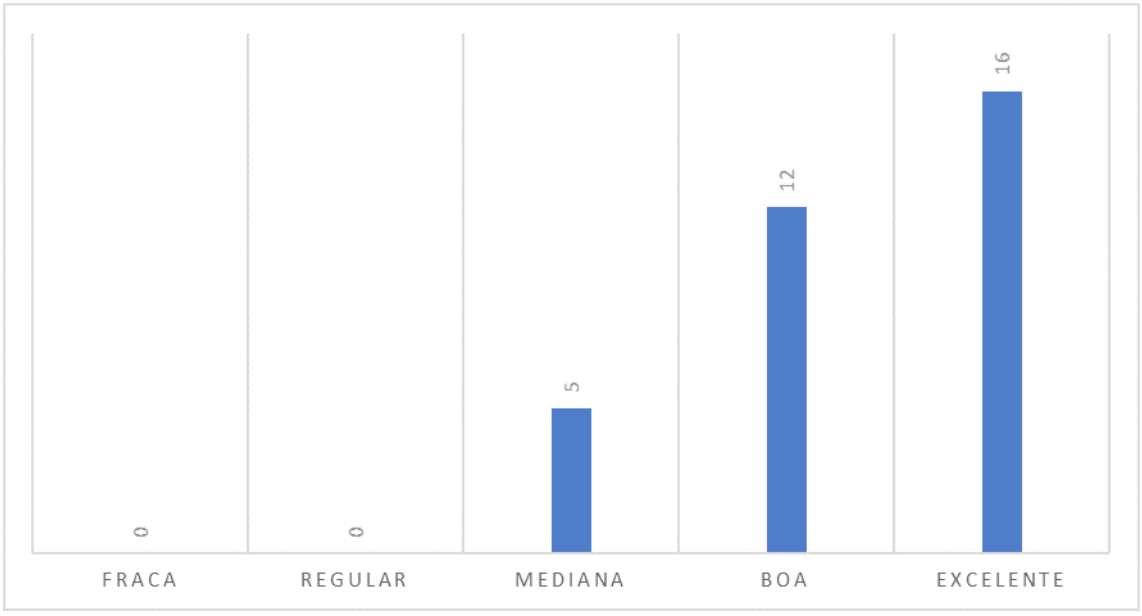
mais de 10 minutos, o que comprova que o jogo não foi só instalado e desinstalado, comprovando um interesse real no jogo.

Figura 57 – Avaliação de jogabilidade dos participantes



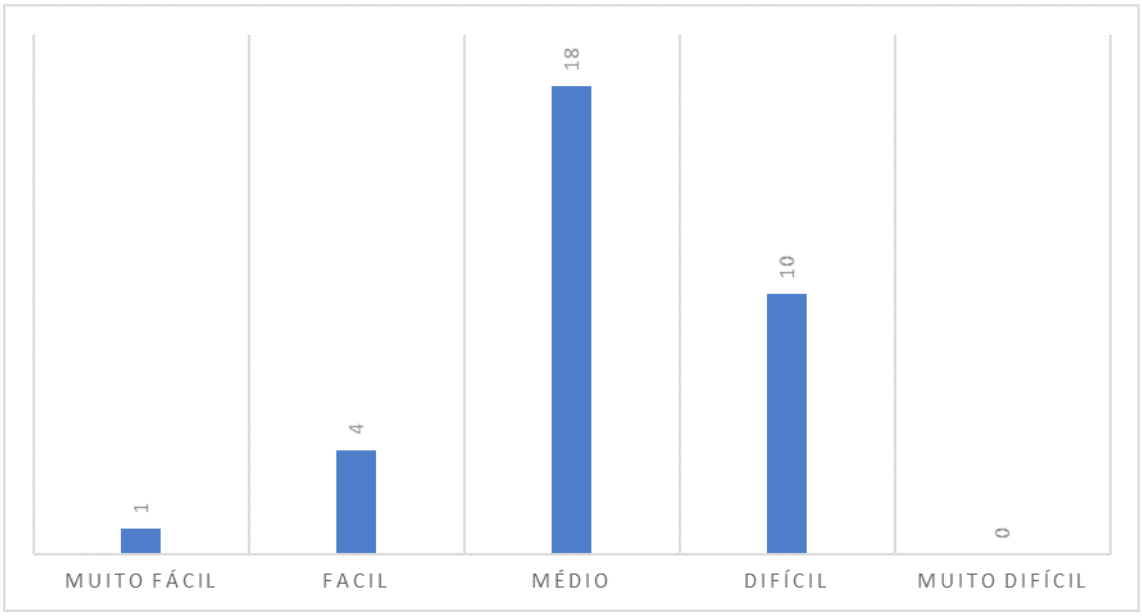
A recepção da jogabilidade foi muito boa, mesmo sendo um jogo não finalizado. As notas 3 e 4 representam 75% do total.

Figura 58 – Avaliação das artes pelos participantes



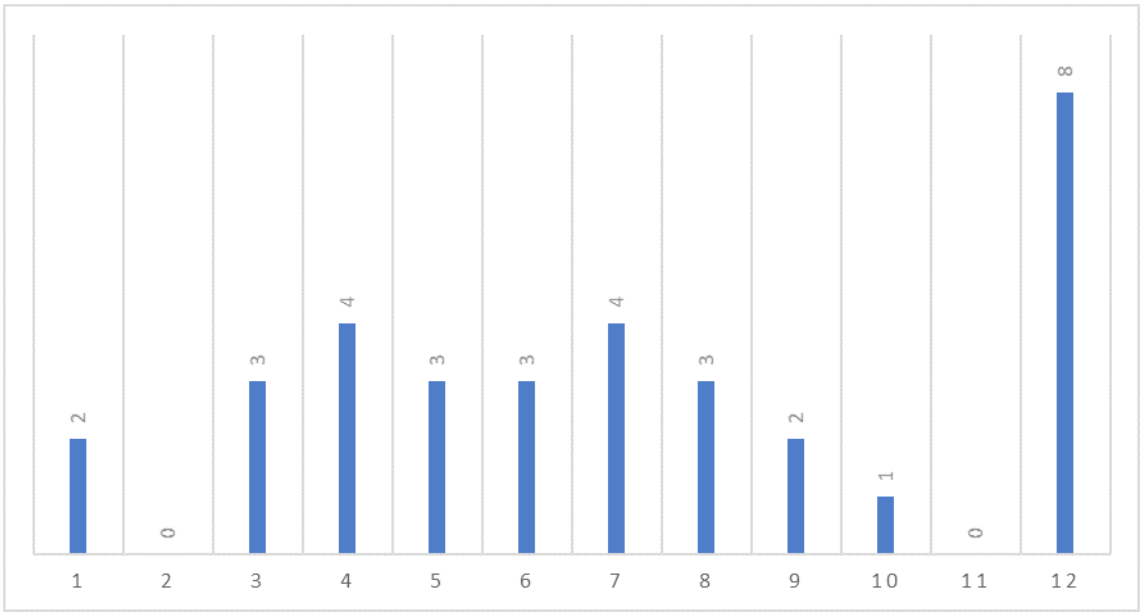
A recepção das artes foi ótima, e a maioria dos participantes deu nota máxima.

Figura 59 – Avaliação da dificuldade pelos participantes



A dificuldade foi classificada como mediana por mais da metade dos participantes, e isso foi um bom resultado já que é um jogo de desafios, não devendo ser nem tão simples para ser chato e nem muito difícil para ser frustrante. Além disso, a dificuldade aumenta gradualmente e será maior em níveis mais avançados.

Figura 60 – Até qual sala os participantes chegaram



Muitos resultados diferentes aqui, mas mais de 25% dos participantes conseguiram chegar até o final. Como avaliamos o tempo médio por sala de 5 minutos e apenas 25% das pessoas realmente jogaram mais de 1 hora, como visto na Figura 56, isso foi comprovado neste resultado.

Figura 61 – O participante jogaria mais, se o jogo fosse finalizado?

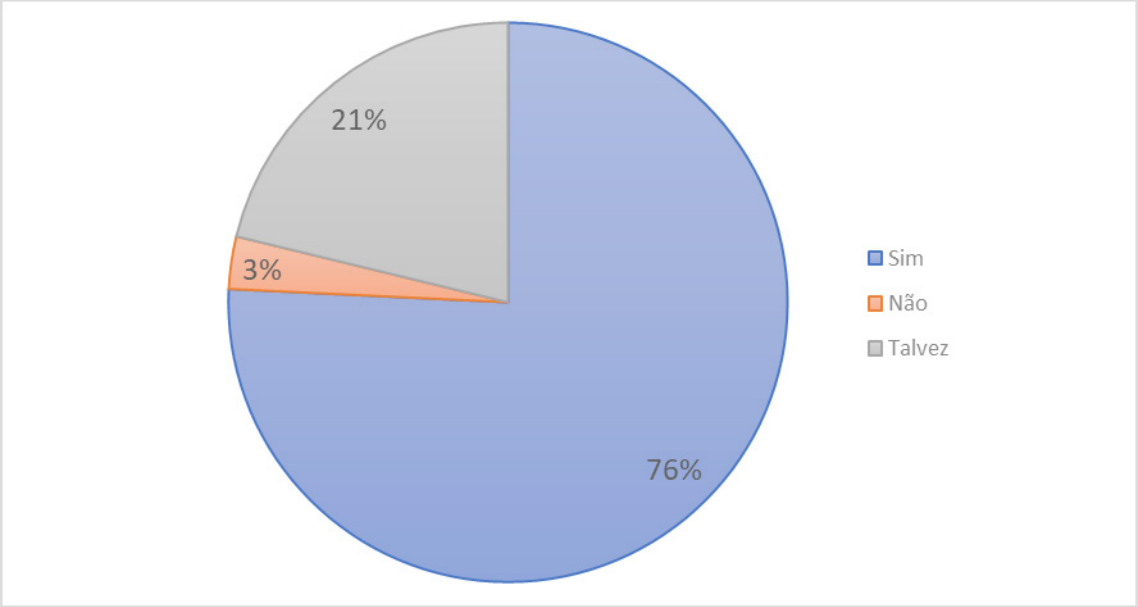
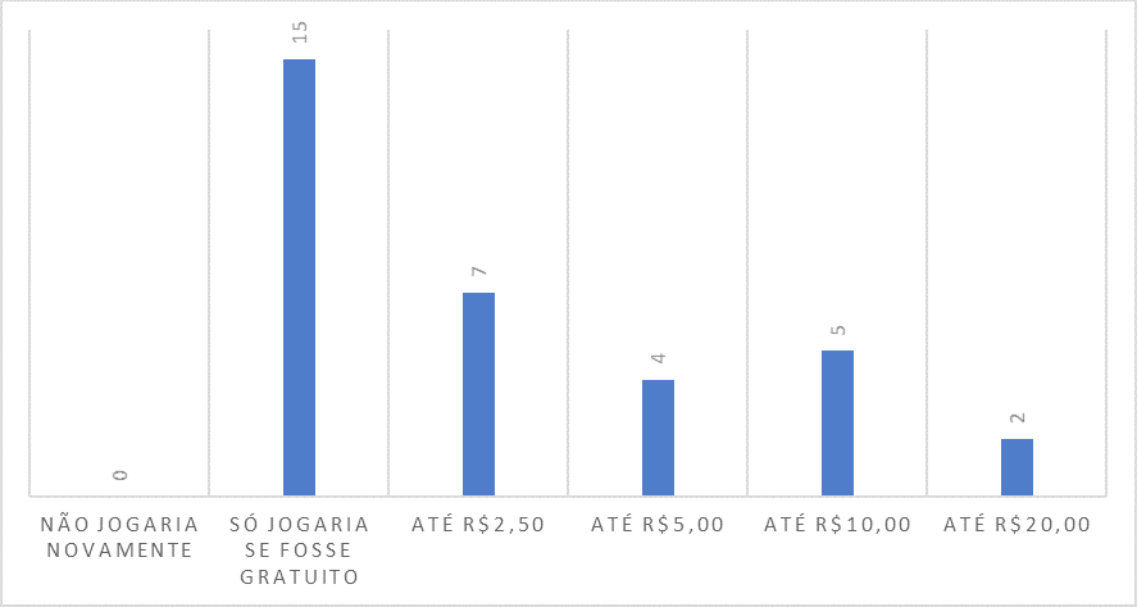


Figura 62 – Até quanto o participante acha que é um preço justo pelo jogo



A grande maioria jogaria, 75% gostariam de ver o jogo finalizado, e mesmo que a maior parte só jogaria se o jogo fosse gratuito, quase 20% pagaria até R$2,50, mas surpreendentemente alguns pagariam até R$20 pelo jogo. O que demonstra um grande interesse das pessoas pelo Lumenide como um jogo casual.

A questão sobre problemas e *bug*s encontrados era uma questão aberta e opcional, apenas alguns participantes comentaram, e os problemas mais comentados foram:

- Problemas ao passar pela porta;

- Problemas com a movimentação;
- Falta de botão de saída do aplicativo;
- O participante não tem celular *Android*, apenas *iOS*.

Fizemos também uma pergunta aberta com o intuito novas sugestões, e tivemos alguns *feedback*s muito interessantes, como:

- Falta de tutorial;
- Novos elementos de jogabilidade, alguns já presentes no conceito original;
- Jogo com potencial de mercado;
- Desafiador.

4.7. Correções de *bugs* e implementações

Novas correções foram implementadas depois da pesquisa e a versão que se encontra atualmente na Google PlayStore é a mais atualizada.

Alguns dos elementos corrigidos e adicionados são, por exemplo, os problemas ao passar pela porta, onde o personagem ficava em uma localização estranha, algumas melhorias nos menus do jogo, como o botão de saída, chefe de final da fase atualizado e funcional, animações de derrota, novos sons e músicas adicionadas e alguns outros *bug*s menores. Foi também adicionado um tutorial, facilitando a jogabilidade de jogadores menos experientes.

5. CONSIDERAÇÕES FINAIS

O conceito inicial parecia muito longe e difícil de ser atingido, mas depois de tanto trabalho ficamos muito felizes pelo que foi alcançado. O tempo foi muito curto, apenas quatro meses do conceito inicial e GDD até o projeto ser realmente finalizado e lançado. Muitas ideias de conceito tiveram que ficar de fora por conta do tempo, mas conseguimos fazer mais o que pensávamos. Em um dado momento pensamos em fazer apenas 5 ou 6 salas, e no fim conseguimos fazer todas as 11 salas inicialmente pensadas no conceito inicial e mais a sala do chefe de fase, que também ficaria de fora.

Conseguimos chegar a esse ponto com muito trabalho em equipe, e foco e determinação de todos os membros, cada um especializado em suas tarefas. Tivemos também ajuda dos professores, principalmente no final do projeto, quando a experiência e o conhecimento foram fatores preponderantes.

O principal fator limitador foi o tempo. Utilizamos todo o tempo livre possível para implementar ideias, muitas vezes discutimos sobre o projeto durante as aulas ou nos intervalos. Também é importante citar o fator conhecimento, já que algumas das ideias eram muito boas, mas para implementar foi necessário estudo e testes. Além disso, tivemos que considerar a publicação do jogo, e a plataforma mais simples para isso foi a escolhida.

A apresentação do jogo na IV Mostra, e os prêmios recebidos, foram muito importantes para a motivação que o grupo precisava no final do processo, além de ter deixado claro o potencial e qualidade que o jogo representa para o mercado. Isso também foi reconhecido na pesquisa que fizemos, onde grande parte dos jogadores apresentaram real interesse pela aquisição de um produto finalizado.

Tivemos que retirar alguns elementos do conceito inicial, mas idealizando colocá-los em fases posteriores do jogo, como por exemplo a Lupa que amplifica o poder do raio de Sol, utilização de um prisma para cores de raios diferentes, além de novos obstáculos como poços onde o jogador pode cair e teias de aranha que deve ser queimadas de alguma forma.

Fazer um jogo completo, desde o início com o GDD até a sua publicação, estudar todo o processo, dificuldades e realizações, feedbacks positivos e negativos de jogadores, correções de *bugs*, frustações por não conseguir implementar algo por falta de conhecimento ou conseguir fazer exatamente o que era esperado, é uma experiência incrível e muito importante para quem escolhe essa área de atuação. Não foi fácil, mas foi gratificante, e valeu todo o esforço no final. Aprendemos muito com isso, e acreditamos que o mercado espera profissionais com essa vivência e experiência.

6. REFERÊNCIAS BIBLIOGRÁFICAS

ARINE, Eduardo. **Aedes Revenge.** Disponível em: <https://play.google.com/store/apps/details?id=br.com.eduardoarine.aedesrevenge&hl=pt_BR>. Acesso em: 04 de junho de 2017.

FIELD, Syd. **Manual do roteiro:** os fundamentos do texto cinematográfico. Rio de Janeiro: Objetiva, 2001.

Fonte **Dalek**. Disponível em: <http://www.netfontes.com.br/view.php/dalek.htm>. Acesso em: 04 de junho de 2017.

LUMENIDE. **Vídeo de apresentação.** Disponível em: <https://www.youtube.com/watch?v=U&video_id=9_uf2_WOAow>. Acesso em: 04 de junho de 2017.

MACLEOD, Kevin. **Curse of the Scarab.** Disponível em: <https://www.youtube.com/watch?v=USDa-WJ-luw>. Acesso em: 04 de junho de 2017.

NADEZHDIN, Cyrill. **CNControls.** Disponível em: <https://www.assetstore.unity3d.com/en/#!/content/15233>. Acesso em: 04 de junho de 2017.

NOVAK, Jeannie. **Introdução ao desenvolvimento de games.** Volume 4. São Paulo: Cengage Learning, 2010.

RABIN, Steve. **Introdução ao desenvolvimento de games:** Entendento o universo dos jogos. Volume 1. São Paulo: Cengage Learning, 2011.

ROGERS, Scott. **Level UP:** Um guia para design de grande jogos. São Paulo: Blucher, 2012.

SALEN, Katie. **Regras do jogo:** Fundamentos do design de jogos: Principais conceitos. Volume 1. São Paulo: Blucher, 2012.

SCHUYTEMA, Paul. **Design de games:** Uma abordagem prática. São Paulo: Cengage Learning, 2013.

SILENT PARTNER. **Belief.** Disponível em: <https://www.youtube.com/audiolibrary>. Acesso em: 04 de junho de 2017.

SIOUX. **Pesquisa Game Brasil 2017.** Disponível em: <https://www.pesquisagame-brasil.com.br/>. Acesso em: 04 de junho de 2017.

The Last Of Us. Naughty Dog. Sony Computer Entertainment, 2013.

The Legend of Zelda. Nintendo. Nintendo, 1986.

7. APÊNDICE

Script 1 – ActiveButton.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

public class ActiveButton : MonoBehaviour {

     [SerializeField] private Sprite OnImage = null;
     [SerializeField] private Sprite OffImage = null;
     [SerializeField] private ValueButton position = 0;
     [Tooltip("True para ativar o botão ao passar por cima e não mais pelo Action Button,
     false para impedir que passe por cima e ativara pelo Active button")]
     [SerializeField] private bool oneKeyPress = false;
     [SerializeField] private bool activeOnStayOver = false;
     [Header("Use Time to ON/OFF")]
     [Tooltip("Caso diferente seja > 0 , será ativado a função")]
     [SerializeField] private float timeToRelease = 0f;
     [SerializeField] private BoxCollider2D blockPassageCollider = null;
     [SerializeField] private UnityEvent callBackActionON = null;
     [SerializeField] private UnityEvent callBackActionOFF = null;
     private enum ValueButton {ON,OFF}
     private float timerCount = 0f;
     private bool changeState = false;
     private bool playerIsOver = false;

     //Audio
     public AudioClip buttonAudio;
     private AudioSource audio;
     private bool playSound = false;

     void Start () {
          playSound = false;
          audio = GetComponent<AudioSource>();

          if (position == ValueButton.OFF) {
               SetOFF (false);
          } else {
               SetON (false);
          }

          blockPassageCollider.isTrigger = activeOnStayOver;
          GetComponents<BoxCollider2D> ()[0].enabled = !activeOnStayOver;
          GetComponents<BoxCollider2D> ()[1].enabled = !activeOnStayOver;

          ShowActiveButton sab = GetComponent<ShowActiveButton>();
          if (sab != null) {
               sab.enabled = activeOnStayOver;
          }
     }

     void Update () {
          playSound = true;

          if (timerCount < 0)
               timerCount = 0;

          ButtonListener ();
     }
```

```
void ButtonListener() {
     // is in cooldown
     if (OnCoolDown ())
          return;

     if (oneKeyPress) {
          if (CnInputManager.GetButtonDown (“Action”) && playerIsOver) {
               SwichButton (true);
          }
     } else {
          if (CnInputManager.GetButtonDown (“Action”) && playerIsOver) {
               SwichButton (true);
          } else if(CnInputManager.GetButtonUp(“Action”) && playerIsOver) {
               SwichButton (true);
          }
     }
}

void SwichButton(bool changeState) {
     if (position == ValueButton.OFF) {
          SetON (changeState);
     } else {
          SetOFF (changeState);
     }
}

public void SetON(bool changeState) {
     // is in cooldown
     if (OnCoolDown ())
          return;

     if(OnImage)
          GetComponent<SpriteRenderer> ().sprite = OnImage;

     position = ValueButton.ON;
     PlaySound();
     CallBackON ();

     if (changeState)
          StartCoolDown();
     }

public void SetOFF(bool changeState) {
     // is in cooldown
     if (OnCoolDown ())
          return;

     if(OffImage)
          GetComponent<SpriteRenderer> ().sprite = OffImage;

     position = ValueButton.OFF;
     PlaySound();
     CallBackOFF ();

     if (changeState)
          StartCoolDown();
     }

void StartCoolDown() {
     if (timeToRelease <= 0f) {
          this.changeState = false;
          return;
     }

     this.changeState = true;
     timerCount = 0f;
}
```

```
    void StartCoolDown() {
        if (timeToRelease <= 0f) {
            this.changeState = false;
            return;
        }

        this.changeState = true;
        timerCount = 0f;
    }

    bool OnCoolDown() {
        if (!changeState)
            return false;

        timerCount += Time.deltaTime;

        if (timerCount >= timeToRelease) {
            changeState = false;
            SwichButton (false);
            return false;
        }
        return true;
    }

    void OnTriggerEnter2D(Collider2D collider) {
        if (isPlayerCollider(collider)) {
            playerIsOver = true;
        }
    }

    void OnTriggerExit2D(Collider2D collider) {
        if (isPlayerCollider(collider)) {
            playerIsOver = false;
        }
    }

    bool isPlayerCollider(Collider2D collider) {
        if (collider.gameObject.tag == "Player")
            return true;
        return false;
    }

    void CallBackON() {
        if (callBackActionON  != null)
            callBackActionON.Invoke();
    }

    void CallBackOFF() {
        if (callBackActionOFF  != null)
            callBackActionOFF.Invoke();
    }

    void PlaySound() {
        if (!playSound)
        return;

        audio.clip = buttonAudio;
        audio.Play();
    }
}
```

Script 2 – ActiveOnOver.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

public class ActiveOnOver : MonoBehaviour {
     [SerializeField] private bool activeOneTime = false;
     [SerializeField] private UnityEvent callBackOnOver = null;
     [SerializeField] private UnityEvent callBackOnExit = null;
     [SerializeField] private List<string> tagFilter;
     private string tagEnter = "";
     private bool isActiveEnter = false;
     private bool isActiveExit = false;

     void Start () {
          if (tagFilter == null)
               tagFilter = new List<string> ();
     }

     void OnTriggerEnter2D(Collider2D collider) {
          if (ValidateTagList (collider) && tagEnter.Equals("") && !isActiveEnter) {
               if (callBackOnOver != null)
                    callBackOnOver.Invoke ();

               tagEnter = collider.gameObject.tag;

          if (activeOneTime) {
               isActiveEnter = true;
               }
          }
     }

     void OnTriggerExit2D(Collider2D collider) {
          if (ValidateTagList (collider) && !isActiveExit) {
               if (!tagEnter.Equals (collider.gameObject.tag)) {
                    return;
               }
               tagEnter = "";

               if (callBackOnExit  != null)
                    callBackOnExit.Invoke ();

               if (activeOneTime) {
                    isActiveExit = true;
               }
          }
     }

     bool ValidateTagList(Collider2D collider) {
          if (tagFilter.Count <= 0)
               return true;

          bool retorno = false;
          foreach(string value in tagFilter) {
               if (value.Equals (collider.gameObject.tag)) {
                    retorno = true;
                    break;
               }
          }
          return retorno;
     }
}
```

Script 3 – BossAnimation.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BossAnimation : MonoBehaviour {
     [SerializeField] private Move _ Set moveSet = 0;
     [SerializeField] private float speed = 1f;
     private enum Move _ Set { LEFT _ RIGHT }
     private bool inCollision = false;
     private bool lockDirection = false;
     private Vector3 dir = Vector3.left;
     public bool pauseMoviment = false;

     void Start () {
          if (moveSet == Move _ Set.LEFT _ RIGHT) {
               dir = Vector3.left;
          }
     }

     void Update () {
          MoveSet();
     }

     private void MoveSet() {
          if (pauseMoviment) {
               return;
          }
          if (moveSet == Move _ Set.LEFT _ RIGHT) {
               MSLeftRight();
          }
     }

     private void MSLeftRight() {
          transform.position += dir * speed * Time.deltaTime;
     }

     private void OnTriggerEnter2D(Collider2D collision) {
          if (moveSet == Move _ Set.LEFT _ RIGHT) {
               if (collision.gameObject.tag.Equals("Sala")) {
                    dir = dir * -1;
               }
          }
     }
}
```

Script 4 – CameraFollow.cs

```
using UnityEngine;
using UnityEngine.Events;
using System.Collections;

public class CameraFollow : MonoBehaviour {
     public Transform target;
     public GameObject limitObject = null;
     public float smoothing = 5f; // quanto desliza  a camera
     private Vector3 lfMin,lfMax;
     private CamLimitBounds limiteFase; //componente q define o tamanho da faze.
     private UnityEvent eventOnPositon;
     private Vector3 lastPosition = Vector3.zero;
     private float stopTime = 0f;

     void Start () {
          GetLimits();
     }
```

```
void GetLimits() {
     if (limitObject != null)
     limiteFase = limitObject.GetComponent<CamLimitBounds>();

     if (limitObject != null) {
          limiteFase.CalcLimits();
          lfMin = limiteFase.LimitMin;
          lfMax = limiteFase.LimitMax;
     }
}

public void ChangeTargetEvent(GameObject newTarget, UnityEvent func) {
     ChangeTarget(newTarget);
     eventOnPositon = func;
}

public void ChangeTarget(GameObject newTarget) {
     target = newTarget.transform;
}

public void ChangeScene(GameObject newLimits, GameObject newTarget) {
     if (newTarget != null) {
          target = newTarget.transform;
     }
     limitObject = newLimits;
     GetLimits();
}

void FixedUpdate() {
     float x = transform.position.x;
     float y = transform.position.y;

     if (Mathf.Abs (x - target.position.x) > 0) {
          x = Mathf.Lerp(x,target.position.x,smoothing * Time.deltaTime);
     }

     if (Mathf.Abs (y - target.position.y) > 0) {
          y = Mathf.Lerp(y,target.position.y,smoothing * Time.deltaTime);
     }

     float cameraHalfWidth = Camera.main.orthographicSize *
          ((float) Screen.width/Screen.height);

     x = Mathf.Clamp (x, lfMax.x + cameraHalfWidth, lfMin.x - cameraHalfWidth);
     y = Mathf.Clamp (y, lfMin.y + Camera.main.orthographicSize,
          lfMax.y - Camera.main.orthographicSize);

     float distCovered = Time.time * smoothing;
     float fracJourney = distCovered / Vector3.Distance(transform.position,
          new Vector3(x, y, transform.position.z)); ;
     transform.position = Vector3.Lerp(transform.position,
          new Vector3(x, y, transform.position.z),Time.deltaTime);

     if (CheckCamOnTargetPositon()) {
          if(eventOnPositon != null) {
               eventOnPositon.Invoke();
               eventOnPositon = null;
          }
     }
     lastPosition = transform.position;
}
```

```
     public bool CheckCamOnTargetPositon() {
          bool retorno = false;

          if(Vector3.SqrMagnitude( lastPosition - transform.position) < 0.0001f)     {
               if (stopTime >= 1f) {
                    retorno = true;
               }
               stopTime += 1f;
          } else {
               stopTime = 0f;
          }
          return retorno;
     }
}
```

Script 5 – CamLimitBounds.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CamLimitBounds : MonoBehaviour {
     [SerializeField] private float width;
     [SerializeField] private float height;
     private Vector3 limitMax = Vector3.zero;
     private Vector3 leftPoint1 = Vector3.zero;
     private Vector3 rightPoint0 = Vector3.zero;
     private Vector3 limitMin = Vector3.zero;
     private Vector3 topPoint0 = Vector3.zero;
     private Vector3 topPoint1 = Vector3.zero;
     private Vector3 botPoint0 = Vector3.zero;
     private Vector3 botPoint1 = Vector3.zero;

     public Vector3 LimitMax {
          get {
               return limitMax;
          }
          set {
               limitMax = value;
          }
     }

     public Vector3 LimitMin {
          get {
               return limitMin;
          }
          set {
               limitMin = value;
          }
     }

     public void CalcLimits() {
          Vector3 centerPosition = this.transform.position;
          Color cor = Color.red;
          limitMax = new Vector3(centerPosition.x - width, centerPosition.y + height, -1f);
          leftPoint1 = new Vector3(centerPosition.x - width,
               centerPosition.y - height, -1f);

          rightPoint0 = new Vector3(centerPosition.x + width,
               centerPosition.y + height, -1f);
          limitMin = new Vector3(centerPosition.x + width, centerPosition.y - height, -1f);

          topPoint0 = new Vector3(centerPosition.x + width, centerPosition.y + height, -1f);
          topPoint1 = new Vector3(centerPosition.x - width, centerPosition.y + height, -1f);

          botPoint0 = new Vector3(centerPosition.x + width, centerPosition.y - height, -1f);
          botPoint1 = new Vector3(centerPosition.x - width, centerPosition.y - height, -1f);
     }
```

```
     private void OnDrawGizmos() {
          CalcLimits();
          Gizmos.color = Color.red;
          Gizmos.DrawLine(limitMax, leftPoint1);
          Gizmos.DrawLine(rightPoint0, limitMin);
          Gizmos.DrawLine(topPoint0, topPoint1);
          Gizmos.DrawLine(botPoint0, botPoint1);
          Gizmos.DrawWireSphere(transform.position, 1f);
     }
}
```

Script 6 – Door.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.Events;
using UnityEngine;

public class Door : MonoBehaviour {
     [Header("Start Position")]
     [SerializeField] private bool isOpen = false;
     [SerializeField] private GameObject topMask = null;
     [SerializeField] private UnityEvent callBackOnOpen = null;
     [SerializeField] private UnityEvent callBackOnClose = null;
     [SerializeField] private UnityEvent callBackEnterAnim = null;
     [SerializeField] private UnityEvent callBackExitAnim = null;
     private DoorAnimationBehavior doorControle;
     private Animator animator;
     public AudioClip openDoorAudio, closingDoorAudio;
     private AudioSource audio;

     void Awake() {
          animator = GetComponent<Animator> ();
          audio = GetComponent<AudioSource>();
          audio.volume = 0f;
     }

     void Start () {
          EnableCollider(!isOpen);
          animator.SetBool("StartOpen", isOpen);
          doorControle = animator.GetBehaviour<DoorAnimationBehavior>();
          doorControle.DoorBh = this;
     }

     void Open() {
          animator.SetTrigger ("Open");
          audio.clip = openDoorAudio;
          audio.Play();
     }

     void Close() {
          animator.SetTrigger ("Close");
          audio.clip = closingDoorAudio;
          audio.Play();
     }

     public void SwithState() {
          if (isOpen) {
               isOpen = false;
               Close ();
          } else {
               isOpen = true;
               Open ();
          }
          animator.SetBool ("StartOpen", isOpen);
     }
```

```
     public void EnableCollider (bool value) {
          GetComponent<BoxCollider2D> ().enabled = value;
          topMask.SetActive (!value);

          if (value) {
               if(callBackOnClose !=null)
                    callBackOnClose.Invoke();
          } else {
               if (callBackOnOpen != null)
                    callBackOnOpen.Invoke();
               }
          }

     public void CallBackEnterAnim() {
          if (callBackEnterAnim != null)
               callBackEnterAnim.Invoke();
     }

     public void CallBackExitAnim() {
          if (callBackExitAnim != null)
               callBackExitAnim.Invoke();
     }
}
```

Script 7 – DoorAnimationBehavior.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DoorAnimationBehavior : StateMachineBehaviour {
     private Door doorMonoBehavior;
     public Door DoorBh {
          set {
               this.doorMonoBehavior = value;
          }
     }

     public override void OnStateEnter(Animator animator,
          AnimatorStateInfo animatorStateInfo, int layerIndex) {

          if(animatorStateInfo.IsName("CloseDoor") || animatorStateInfo.IsName("OpenDoor")){
               doorMonoBehavior.CallBackEnterAnim();
          }
          if(animatorStateInfo.IsName("CloseDoor")) {
               doorMonoBehavior.EnableCollider(true);
          }
     }

     override public void OnStateExit (Animator animator, AnimatorStateInfo stateInfo,
          int layerIndex) {

          if (stateInfo.IsName ("OpenDoor")) {
               doorMonoBehavior.EnableCollider(false);
          }
          if (stateInfo.IsName("CloseDoor") || stateInfo.IsName("OpenDoor")) {
               doorMonoBehavior.CallBackExitAnim();
          }
     }
}
```

Script 8 – EventPlayer.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EventPlayer : MonoBehaviour {
     [SerializeField]
     private Direction direction = 0;
     private enum Direction {LEFT,RIGHT,UP,DOWN }
     private MovimentPlayer movscr = null;

     private void Start() {
          if (movscr == null)
          movscr = GameObject.FindGameObjectWithTag("Player").
               GetComponent<MovimentPlayer>();
     }

     public void StartAutoMoviment() {
          LevelManager.instance.LastEventPlayer = this;
          movscr.EnableMoviment(false);
          movscr.AutoMoviemnt(GetDirection());
     }

     public void StopAutoMoviment() {
          LevelManager.instance.LastEventPlayer = this;
          movscr.EnableMoviment(true);
     }

     private Vector3 GetDirection() {
          Vector3 direct = Vector3.zero;
          if(direction == Direction.LEFT) {
               direct = Vector3.left;
          } else if (direction == Direction.RIGHT) {
               direct = Vector3.right;
          } else if (direction == Direction.UP) {
               direct = Vector3.up;
          } else if (direction == Direction.DOWN) {
               direct = Vector3.down;
          }
          return direct;
     }
}
```

Script 9 – GameManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;
using UnityEngine;

public class GameManager : MonoBehaviour {
     public static GameManager instance = null;
     private SaveGame saveGame = null;

     public SaveGame SaveGame {
          get {
               return saveGame;
          }
     }

     private void Awake() {
          if (instance == null)
               instance = this;
          else if (instance != this)
               Destroy(gameObject);

          DontDestroyOnLoad(gameObject);
```

```
        SaveGame save = new SaveGame();
        if (!PlayerPrefs.HasKey("SaveGame")) {
            save.currentLevel = 1;
            save.currentStage = 1;
            save.muteMusic = false;
            save.muteMusic = false;
            save.tutorialDoneList = new List<int>();
            save.listaReliquias = new List<ItemReliquia>();

            PlayerPrefs.SetString("SaveGame", save.ToJson());
        } else {
            save.Load(PlayerPrefs.GetString("SaveGame"));
        }
        saveGame = save;
    }

    private void Start() {
        instance.LoadGame(saveGame);
    }

    public void SaveCompleteLevel(int newLevel, int newStage) {
        if(newLevel > saveGame.currentLevel)
            saveGame.currentLevel = newLevel;

        if (newStage > saveGame.currentStage)
            saveGame.currentStage = newStage;
        Save();
    }

    public void SaveConfig() {
        saveGame.muteSFX = SoundManager.instance.MusicMuted;
        saveGame.muteMusic = SoundManager.instance.SfxMuted;
        Save();
    }

    public void AddSaveTutorial(int index) {
        if(!saveGame.tutorialDoneList.Contains(index))
            saveGame.tutorialDoneList.Add(index);
    }

    public void SaveTutorial() {
        Save();
    }

    public bool TutorialIsDone(int index) {
        return saveGame.tutorialDoneList.Contains(index);
    }

    public void SalvarReliquia(ItemReliquia item) {
        if (saveGame.listaReliquias == null)
            saveGame.listaReliquias = new List<ItemReliquia>();

        saveGame.listaReliquias.Add(item);
        Save();
    }

    public bool VerificarPedraPega(int sala) {
        if (instance.SaveGame == null)
            return false;

        bool retorno = false;
        if (instance.SaveGame.listaReliquias != null) {
            foreach (ItemReliquia ir in instance.SaveGame.listaReliquias) {
                if (ir.numeroSala == sala && ir.numeroLevel ==
                    LevelManager.instance.SelectedLevel) {
                    retorno = true;
                    break;
                }
            }
        }
        return retorno;
    }
```

```
     private void Save() {
          PlayerPrefs.SetString("SaveGame", saveGame.ToJson());
     }

     public void LoadGame(SaveGame save) {
          SoundManager.instance.MuteSFX(save.muteSFX);
          SoundManager.instance.MuteMusic(save.muteMusic);
     }

     public void LoadLevel(string nameLevel) {
          StartCoroutine(ChangeScene(nameLevel));
     }

     public void LoadLevel(int sala) {
          LevelManager.instance.CurrentRoon = sala;
          int selLevel = LevelManager.instance.SelectedLevel < 1 ? 1 :
               LevelManager.instance.SelectedLevel;
          string level = selLevel < 10 ? "0"+selLevel.ToString() : selLevel.ToString() ;
          StartCoroutine(ChangeScene("Level _ " + level));
     }

     IEnumerator ChangeScene(string levelName) {
          float fadeTime = Camera.main.gameObject.GetComponent<Fading>().BeginFade(1);
          yield return new WaitForSeconds(fadeTime);
          SceneManager.LoadScene(levelName);
     }

     public void QuitRequest() {
          Application.Quit();
     }

     public void StopTime(bool value) {
          Time.timeScale = value ? 0f : 1f;
     }
}
```

Script 10 – LevelManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class LevelManager : MonoBehaviour {
     public int numeroSalaTeste = 1;
     public static LevelManager instance = null;
     private List<LevelBehaviour> levels = null;
     private EventPlayer lastEventPlayer = null;
     private bool bossComplete = false;
     public bool isChangeLevel = false;
     private int selectedLevel = -1;

     public int SelectedLevel {
          get {
               return selectedLevel;
          }
          set {
               selectedLevel = value;
          }
     }

     public int CurrentRoon {
          get {
               return currentRoon;
          }
          set {
               currentRoon = value;
          }
     }
```

```
public List<LevelBehaviour> Levels {
     get {
          return levels;
     }
     set {
          levels = value;
     }
}

public EventPlayer LastEventPlayer {
     get {
          return lastEventPlayer;
     }
     set {
          lastEventPlayer = value;
     }
}

public bool BossComplete {
     get {
          return bossComplete;
     }
     set {
          bossComplete = value;
     }
}
private int currentRoon = -1;

private void Awake() {
     if (instance == null)
          instance = this;
     else if (instance != this)
          Destroy(gameObject);

     DontDestroyOnLoad(gameObject);
}

public void ChangeLevel(int roomIndex) {
     DesabilitaSalas();
     LevelBehaviour level = (LevelBehaviour)levels.ToArray().GetValue(roomIndex -1);
     GameObject mainCam = Camera.main.gameObject;
     mainCam.GetComponent<CameraFollow>().ChangeScene(level.LevelBounds, null);
}

public void LoadStage(List<LevelBehaviour> salas, int salaprateste) {
     if (selectedLevel == -1)
          selectedLevel = 1;

     this.numeroSalaTeste = salaprateste;
     LoadStage(salas);
}

public void LoadStage(List<LevelBehaviour> salas) {
     levels = salas;
     GameObject player = GameObject.FindGameObjectWithTag("Player");
     GameObject mainCam = Camera.main.gameObject;
     currentRoon = currentRoon < 1 ? numeroSalaTeste : currentRoon;
     DesabilitaSalas();
     LevelBehaviour level = (LevelBehaviour)levels.ToArray().GetValue(currentRoon - 1);
     player.transform.position = level.SpawnPositionPlayer.transform.position;
     Vector3 pos = player.transform.position;
     mainCam.transform.position = new Vector3(pos.x,pos.y, mainCam.transform.position.z);
     mainCam.GetComponent<CameraFollow>().ChangeScene(level.LevelBounds, player);
}

public void EnablePlayerMoviment(bool value) {
     GameObject player = GameObject.FindGameObjectWithTag("Player");
     player.GetComponent<MovimentPlayer>().EnableMoviment( value);
}
```

```
     public void SetSelectedRoon(int roon) {
          currentRoon = roon + 1;
     }

     private void DesabilitaSalas() {
          foreach (LevelBehaviour sala in levels) {
               if(levels.IndexOf(sala) != currentRoon - 1)
                    sala.gameObject.SetActive(false);
               else
                    sala.gameObject.SetActive(true);
          }
     }
}
```

Script 11 – Loader.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Loader : MonoBehaviour {
     public GameObject gameManager = null;
     public GameObject soundManager = null;
     public GameObject levelManager = null;
     public GameObject menuManager = null;

     private void Awake() {
          if(GameManager.instance == null) {
               Instantiate(gameManager);
          }
          if (LevelManager.instance == null) {
               Instantiate(levelManager);
          }
          if(SoundManager.instance == null) {
               Instantiate(soundManager);
          }
          if (MenuManager.instance == null) {
               Instantiate(menuManager);
          }
     }
}
```

Script 12 – MenuManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MenuManager : MonoBehaviour {
     public static MenuManager instance = null;
     private GameObject currentUIPainelOpen = null;

     private void Awake() {
          if (instance == null)
               instance = this;
          else if (instance != this)
               Destroy(gameObject);

          DontDestroyOnLoad(gameObject);
     }

     public void AbriPainel(GameObject painel) {
          if(currentUIPainelOpen == null || painel != currentUIPainelOpen) {
               if(currentUIPainelOpen != null)
                    currentUIPainelOpen.SetActive(false);

               painel.SetActive(true);
               currentUIPainelOpen = painel;
          }
     }
```

```
     public void FechaPainel() {
          if(currentUIPainelOpen != null) {
               currentUIPainelOpen.SetActive(false);
               currentUIPainelOpen = null;
          }
     }
}
```

Script 13 – MovimentPlayer.cs

```
using UnityEngine;
using CnControls;
using System.Collections;

public class MovimentPlayer : MonoBehaviour {

     [SerializeField] private float speed = 3;
     [SerializeField] private BoxCollider2D fixColliderPushObj = null;
     private Animator animator;
     [HideInInspector] public bool isDead = false;
     private float speedMod;
     private float h;
     private float v;
     private float currentSpeed;
     private string[] lockAxis;
     private bool ispullPush;
     private float lastHorizontal;
     private float lastVertical;
     private bool isEnable = true;

     void Start () {
          animator = GetComponent<Animator> ();
          if (animator == null)
               Debug.LogWarning ("Inserir um Animator ao personagem.");
     }

     void FixedUpdate () {
          if(isEnable) {
               h = CnInputManager.GetAxisRaw ("Horizontal");
               v = CnInputManager.GetAxisRaw ("Vertical");
               ClampEightDirectionsAxis(h,v);
          }
          Moviment ();
          Animation ();
          ispullPush = false;
     }

     private void ClampEightDirectionsAxis(float axisH, float axisV) {
          if (axisH >= -0.6f && axisH <= -0.5f && axisV >= 0.9f) {
               h = -1f;
               v = 1f;
          } else if (axisH >= -0.6f && axisH <= -0.5f && axisV <= -0.9f) {
               h = -1f;
               v = -1f;
          } else if (axisH >= 0.5f && axisH <= 0.6f && axisV >= -0.9f) {
               h = 1f;
               v = 1f;
          } else if (axisH >= 0.5f && axisH <= 0.6f && axisV <= 0.9f) {
               h = 1f;
               v = -1f;
          } else if(axisH != 0 || axisV != 0) {
               h = Mathf.Round(axisH);
               v = Mathf.Round(axisV);
          }
     }

     public void EnableMoviment(bool value) {
          isEnable = value;
          h = 0f;
          v = 0f;
          ispullPush = false;
     }
```

```
    void Moviment() {
        Vector3 direction = new Vector3 (h, v, 0f);
        currentSpeed = (speed + speedMod);
        transform.position += direction * currentSpeed * Time.deltaTime;
        if (h != 0 || v != 0) {
            lastVertical = v;
            lastHorizontal = h;
        }
    }

    void Animation() {
        fixColliderPushObj.enabled = ispullPush;
        animator.SetBool ("PullPush", ispullPush);
        animator.speed = Mathf.Abs(currentSpeed) * 0.2f;
        animator.SetFloat ("Horizontal",  h);
        animator.SetFloat ("Vertical", v);
        animator.SetBool ("Walking", (h != 0f || v != 0f));
        animator.SetFloat ("LastHorizontal", lastHorizontal);
        animator.SetFloat ("LastVertical", lastVertical);
    }

    public void AutoMoviemnt(Vector3 direction) {
        h = direction.x;
        v = direction.y;
    }

    void OnCollisionStay2D(Collision2D collision) {
        if (!collision.gameObject.tag.Equals("MovingObject"))
            return;
        ispullPush = true;
    }

    public void DeadAnimation() {
        if (isDead)
            return;

        animator.SetTrigger("Dead");
        isDead = true;
        EnableMoviment(false);
    }

    public void OnDeadAnimationEnd(float time) {
        StartCoroutine(WaitThenDoThings(time));
    }

    IEnumerator WaitThenDoThings(float time) {
        yield return new WaitForSeconds(time);

        GameObject.FindGameObjectWithTag("UI").GetComponent<PainelPerdeu>().
            ActivePainel(true);
    }
}
```

Script 14 – PainelBehavior.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI.Extensions;

public class PainelBehavior : MonoBehaviour {
    [SerializeField] private GameObject painelStart = null;
    [SerializeField] private GameObject painelLevelSelect = null;
    [SerializeField] private GameObject painelExit = null;

    private void Awake() {
        painelStart.SetActive(false);
        painelLevelSelect.SetActive(false);
    }
```

```
     private void Start() {
          AbrirPainelStart();
     }

     private void Update()   {
          if (Input.GetKeyDown(KeyCode.Escape))
               painelExit.SetActive(true);
     }

     private void Open(GameObject UI) {
          MenuManager.instance.AbriPainel(UI);
     }

     public void SetLevel(int level) {
          if(level < 0) {
               if (GetComponent<HorizontalScrollSnap>() != null) {
                    LevelManager.instance.SelectedLevel =
                         GetComponent<HorizontalScrollSnap>().CurrentPage +1;
               }
          } else {
               LevelManager.instance.SelectedLevel = level +1;
          }
     }

     public void AbrirPainelStart() {
          Open(painelStart);
     }

     public void AbrirPainelLevelSelect() {
          Open(painelLevelSelect);
          LevelManager.instance.SelectedLevel = painelLevelSelect.
               GetComponent<HorizontalScrollSnap>().CurrentPage +1;
     }

     public void FecharPainel(GameObject painel) {
          painel.SetActive(false);
     }

     public void AbrirPainel(GameObject painel) {
          painel.SetActive(false);
     }

     public void ExitGame() {
          GameManager.instance.QuitRequest();
     }
}
```

Script 15 – PauseMenu.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class PauseMenu : MonoBehaviour {
     public GameObject painelPause = null;
     public GameObject painelSair = null;
     public Toggle tgSoundSFX = null;
     public Toggle tgSoundMusic = null;

     public void Open() {
          GameManager.instance.StopTime(true);
          MenuManager.instance.AbriPainel(painelPause);
          carregaConfiguracao();
     }

     public void OpenNormal() {
          painelPause.SetActive(true);
          carregaConfiguracao();
     }
```

```
     public void FechaNormal() {
          painelPause.SetActive(false);
     }

     public void Close() {
          GameManager.instance.StopTime(false);
          MenuManager.instance.FechaPainel();
     }

     void carregaConfiguracao() {
          tgSoundMusic.isOn = SoundManager.instance.MusicMuted;
          tgSoundSFX.isOn = SoundManager.instance.SfxMuted;
     }

     public void OffSoundFX(bool value) {
          SoundManager.instance.MuteSFX(value);
     }

     public void OffSoundMusic(bool value) {
          SoundManager.instance.MuteMusic(value);
     }

     public void RestartLevel() {
          GameManager.instance.StopTime(false);
          GameManager.instance.LoadLevel(LevelManager.instance.CurrentRoon);
     }

     public void Sair() {
          GameManager.instance.StopTime(false);
          GameManager.instance.LoadLevel("MainMenu");
     }

     public void OpenPainelSair() {
          painelSair.SetActive(true);
     }

     public void ClosePainelSair() {
          painelSair.SetActive(false);
     }
}
```

Script 16 – ReceptorLuz.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

public class ReceptorLuz : MonoBehaviour {
     private Animator anim;
     [SerializeField] private UnityEvent callBackActionON = null;
     [SerializeField] private GameObject timerUI= null;
     private bool isActivate = false;
     private float lastTime = 0f;
     private bool startCount = false;

     private void Awake() {
          anim = transform.parent.gameObject.GetComponent<Animator>();
          if (anim == null)
               Debug.LogWarning("Faltando Animator.");
          if (timerUI == null)
               Debug.LogWarning("Faltando TimerUI.");
          timerUI.SetActive(false);
     }

     private void Update() {
          CheckTimer();
     }
```

```
    void OnTriggerStay2D(Collider2D collider) {
        if (isActivate)
            return;
        if (collider.gameObject.tag.Equals ("SolarRay")) {
            startCount = true;
        }
    }

    private void OnTriggerExit2D(Collider2D collision) {
        if (collision.gameObject.tag.Equals("SolarRay")) {
            startCount = false;
        }
    }

    void CheckTimer() {
        if(startCount) {
            timerUI.SetActive(true);
            if (timerUI.GetComponent<TimerUI>().IsDone()) {
                isActivate = true;
                ActiveReceptor();
                timerUI.SetActive(false);
                startCount = false;
            }
        } else {
            timerUI.SetActive(false);
        }
    }

    void ActiveReceptor() {
        anim.SetTrigger("ON");
        if (callBackActionON != null)
            callBackActionON.Invoke();
    }

    public void SetStartCount(bool value) {
        startCount = value;
    }
}
```

Script 17 – ReflectionSumRay.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[ExecuteInEditMode]
public class ReflectionSumRay : MonoBehaviour {
    [SerializeField] private GameObject raioLuzPrefab;
    [SerializeField] private ReflectionType reflectionType = 0;
    [SerializeField] private NormalsDirection mirrorFace = 0;
    private enum NormalsDirection {UP,DOWN,LEFT,RIGHT,UP _ RIGHT,DOWN _ RIGHT,
        UP _ LEFT,DOWN _ LEFT}
    [Header("Imagens")]
    [SerializeField] public Sprite imgUp;
    [SerializeField] public Sprite imgDown;
    [SerializeField] public Sprite imgLeft;
    [SerializeField] public Sprite imgRight;
    [SerializeField] public Sprite imgUpRight;
    [SerializeField] public Sprite imgUpLeft;
    [SerializeField] public Sprite imgDownRight;
    [SerializeField] public Sprite imgDownLeft;
```

```
private enum ReflectionType {Mirror,Redirecionador}
private SunRay sunRay = null;
private Vector3 orignalPositionRay;
private float originalOrderLayerRay;
private bool rayCollide;
private bool isReflecting = false;
private GameObject raioLuzInstance = null;
private GameObject rayPointIN;
public GameObject RaioLuzInstance {
     get {
          return this.raioLuzInstance;
     }
}
private NormalsDirection anterior = 0;
private GameObject reflectObject = null;

//Audio
[Header("Audio")]
public AudioClip turnMirrorAudio;
private AudioSource audio;

void Start () {
     audio = GetComponentInParent<AudioSource>();
     sunRay = new SunRay (transform);
}

void Update () {
     if (anterior != mirrorFace) {
          anterior = mirrorFace;
          ChangeDirectionMirror (anterior);
     }

     GetDirectionNormals ();
     if (!Application.isPlaying) {
          return;
     }

     if (isReflecting && rayPointIN != null) {
          if (RayIsOnScene(rayPointIN)) {
               ReflectSumRay(rayPointIN);
          }
     } else {
          if(rayPointIN == null && isReflecting) {
               CheckDestroyObject();
          }
     }
}

bool RayIsOnScene(GameObject ray) {
     if(ray.scene.name.Equals("")) {
          CheckDestroyObject();
          return false;
     }
     return true;
}

void ReflectSumRay(GameObject rayPoint) {
     Vector3 originRayPoint = rayPoint.GetComponent<LineRenderer> ().GetPosition (0);
     Vector3 newDirection = CalculeReflection (originRayPoint);
     if (newDirection == Vector3.zero) {
          if (rayPointIN != null)
               rayPointIN = null;
          return;
     }
     if(this.raioLuzInstance == null)
          this.raioLuzInstance = Instantiate(raioLuzPrefab);

     this.raioLuzInstance.transform.parent = transform;
     sunRay.CastSumRayReflection (this.raioLuzInstance,newDirection);
}
```

```
Vector3 CalculeReflection(Vector3 originalPosition) {
     Vector3 retorno = Vector3.zero;
     switch (reflectionType) {
     case ReflectionType.Mirror:
          retorno = reflectionMirror (originalPosition);
          break;
     case ReflectionType.Redirecionador:
          retorno = GetDirectionNormals();
          break;
     }
     if (retorno == Vector3.zero)
          return retorno;

     return transform.position + (retorno * 50f);
}

Vector3 reflectionMirror(Vector3 positionIncident) {
     Vector3 heading = positionIncident - transform.position;
     float distance = heading.magnitude;
     Vector3 inDirection = heading / distance;
     float angleIncident = Mathf.Round(Vector3.Angle(inDirection,
          GetDirectionNormals()) );

     if (CheckReflection (angleIncident)) {
          float reflexionAngle = 360 - angleIncident;
          return Reflecao (inDirection);
     }

     return Vector3.zero;
}

Vector3 Reflecao(Vector3 vf) {
     Vector3 ve = GetDirectionNormals ();
     vf.Normalize ();
     ve.Normalize ();
     float xf = vf.x;
     float yf = vf.y;
     float xe = ve.x;
     float ye = ve.y;

     float xr = xf * (Mathf.Pow (ye,2f) - Mathf.Pow (xe,2f)) - 2 * yf * xe * ye;
     float yr = -2 * xf * xe * ye + yf * (Mathf.Pow (xe,2f) - Mathf.Pow (ye,2f));

     return new Vector3 (xr, yr, 0f).normalized * -1;
}

bool CheckReflection(float angleIncident) {
     bool retorno = false;

     if (angleIncident >= 5 && angleIncident < 85) {
          retorno = true;
     }

     return retorno;
}

void OnTriggerStay2D(Collider2D collider) {
     if (collider.gameObject.tag.Equals("SolarRay")) {
          rayPointIN = collider.transform.parent.gameObject;

          if (rayPointIN == null)
               return;

          isReflecting = true;
     }
}
```

```
void OnTriggerExit2D(Collider2D collider) {
     if (!collider.gameObject.tag.Equals ("SolarRay"))
          return;
     if (this.raioLuzInstance !=null) {
          CheckDestroyObject();
     }
}

public List<GameObject> VerifyParent() {
     List<GameObject> destroyList = new List<GameObject> ();
     GameObject espelho = sunRay.ReflectingObject;
     ReflectionSumRay script;
     if (espelho != null) {
          script = espelho.GetComponent<ReflectionSumRay> ();
          if (script != null) {
               destroyList.AddRange(script.VerifyParent ());
          }
     }
     if (this.raioLuzInstance != null) {
          destroyList.Add (this.raioLuzInstance);
          this.raioLuzInstance = null;
          sunRay.ReflectingObject = null;
          this.isReflecting = false;
     }
     return destroyList;
}

void CheckDestroyObject() {
     List<GameObject> destroyList = VerifyParent ();

     if (destroyList != null) {
          for (int i = 0; i < destroyList.Count; i++) {
               if(!destroyList[i].scene.name.Equals(""))
                    Destroy (destroyList[i]);
          }
     }
     isReflecting = false;
}

Vector2 GetDirectionNormals() {
     Vector2 retorno = Vector2.zero;
     Sprite setarImagen = null;
     switch (mirrorFace) {
     case NormalsDirection.UP:
          retorno = Vector2.up;
          setarImagen = imgUp;
          break;
     case NormalsDirection.DOWN:
          retorno = Vector2.down;
          setarImagen = imgDown;
          break;
     case NormalsDirection.LEFT:
          retorno = Vector2.left;
          setarImagen = imgLeft;
          break;

     case NormalsDirection.RIGHT:
          retorno = Vector2.right;
          setarImagen = imgRight;
          break;
     case NormalsDirection.UP_RIGHT:
          retorno = Vector2.right + Vector2.up;
          setarImagen = imgUpRight;
          break;
     case NormalsDirection.DOWN_RIGHT:
          retorno = Vector2.right + Vector2.down;
          setarImagen = imgDownRight;
          break;
```

```
        case NormalsDirection.UP_LEFT:
            retorno = Vector2.left + Vector2.up;
            setarImagen = imgUpLeft;
            break;
        case NormalsDirection.DOWN_LEFT:
            retorno = Vector2.left + Vector2.down;
            setarImagen = imgDownLeft;
            break;
        }

    public void ChangeDirectionMirrorAction() {

        NormalsDirection newDirection = 0;
        ChangeBACKDirectionMirrorAction((int)mirrorFace);

        switch (mirrorFace) {
        case NormalsDirection.UP:
            newDirection = NormalsDirection.UP_LEFT;
            break;
        case NormalsDirection.UP_LEFT:
            newDirection = NormalsDirection.LEFT;
            break;
        case NormalsDirection.LEFT:
            newDirection = NormalsDirection.DOWN_LEFT;
            break;

        case NormalsDirection.DOWN_LEFT:
            newDirection = NormalsDirection.DOWN;
            break;
        case NormalsDirection.DOWN:
            newDirection = NormalsDirection.DOWN_RIGHT;
            break;
        case NormalsDirection.DOWN_RIGHT:
            newDirection = NormalsDirection.RIGHT;
            break;
        case NormalsDirection.RIGHT:
            newDirection = NormalsDirection.UP_RIGHT;
            break;
        case NormalsDirection.UP_RIGHT:
            newDirection = NormalsDirection.UP;
            break;
        }

        ChangeDirectionMirror (newDirection);
        PlaySound();
    }

        SpriteRenderer componente = transform.parent.GetComponent<SpriteRenderer> ();
        if(componente != null)
            componente.sprite = setarImagen;

        return retorno;
    }

    public void ChangeBACKDirectionMirrorAction(int numberDirection) {
        NormalsDirection oldDirection;
        oldDirection = (NormalsDirection)numberDirection;
        ChangeDirectionMirror(oldDirection);
    }

    void ChangeDirectionMirror(NormalsDirection newDirectionFace) {
        mirrorFace = newDirectionFace;
        GetDirectionNormals ();
        if (this.raioLuzInstance != null) {
            Destroy (this.raioLuzInstance);
            this.raioLuzInstance = null;
        }
    }
```

```
     void PlaySound() {
          if (audio != null) {
               if (turnMirrorAudio == null)
                    return;

               audio.clip = turnMirrorAudio;
               audio.volume = 1f;
               audio.Play();
          }
     }
}
```

Script 18 – Sala.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

[ExecuteInEditMode]
public class Sala : MonoBehaviour {
     [SerializeField] private Image imgMask = null;
     [SerializeField] private int numeroSala = 0;
     [SerializeField] private Text numero = null;
     [SerializeField] private bool isEnable = false;
     private bool isComplete = false;

     public bool IsComplete {
          get {
               return isComplete;
          }
          set {
               isComplete = value;
          }
     }

     public bool IsEnable {
          get {
               return isEnable;
          }
          set {
               isEnable = value;
          }
     }

     private void Start() {
          if (imgMask == null)
               Debug.LogWarning("Mascara do Botão Sala sem imagen.");

          StartMask();
     }

     private void StartMask() {
          Color cor = imgMask.color;
          if (isEnable) {
               cor.a = 0f;
          } else {
               cor.a = 150f;
          }
          imgMask.color = cor;
     }

     private void Update() {
          if (!Application.isPlaying)
               StartMask();

          SetNumero();
          AlphaAnimation();
     }
```

```
     public void OpenLevel() {
          if(isEnable)
               GameManager.instance.LoadLevel(numeroSala);
     }

     private void SetNumero() {
          if (numero == null)
               return;

          numero.text = numeroSala.ToString();
     }

     private void AlphaAnimation() {
          Color cor = imgMask.color;

          if (isEnable) {
               if (cor.a - Time.deltaTime > 0f) {
                    cor.a -= Time.deltaTime;
                    imgMask.color = cor;
               }
          } else {
               if (cor.a + Time.deltaTime < 150f) {
                    cor.a += Time.deltaTime;
                    imgMask.color = cor;
               }
          }
     }
}
```

Script 19 – SaveGame.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SaveGame : SaveUtils {
     public int currentStage;
     public int currentLevel;
     public bool muteSFX;
     public bool muteMusic;
     public List<int> tutorialDoneList;
     public List<ItemReliquia> listaReliquias;
}
```

Script 20 – SoundManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.Audio;

public class SoundManager : MonoBehaviour {
     public static SoundManager instance = null;
     public AudioMixer masterMixer = null;
     private bool sfxMuted = false;
     private bool musicMuted = false;
```

```
    public bool SfxMuted {
        get {
            return sfxMuted;
        }
        set {
            sfxMuted = value;
        }
    }

    public bool MusicMuted {
        get {
            return musicMuted;
        }
        set {
            musicMuted = value;
        }
    }

    private void Awake() {
        if (instance == null)
            instance = this;
        else if (instance != this)
            Destroy(gameObject);

        DontDestroyOnLoad(gameObject);
        SceneManager.sceneLoaded += LevelWasLoaded;
    }

    private void LevelWasLoaded(Scene scene, LoadSceneMode mode) {

    }

    public void MuteMusic(bool value) {
        if (masterMixer == null)
            return;

        this.musicMuted = value;
        masterMixer.SetFloat("musicVolume", (value ? -80f : -15f));
        GameManager.instance.SaveConfig();
    }

    public void MuteSFX(bool value) {
        if (masterMixer == null)
            return;

        this.sfxMuted = value;
        masterMixer.SetFloat("sfxVolume", (value ? -80f : 0f));
        GameManager.instance.SaveConfig();
    }

    public void SetVolumeBossFight() {
        masterMixer.SetFloat("musicVolume", 0f);
    }
}
```

Script 21 – StageController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class StageController : MonoBehaviour {
     [SerializeField] private int numeroLevel = 1;
     [SerializeField] private List<Sala> salas = null;
     [SerializeField] private Image bossColorObj = null;
     private int currentstage = -1;

     private void Awake() {
          bool levelComplete = numeroLevel < GameManager.instance.SaveGame.currentLevel;
          currentstage = levelComplete ? salas.Count :
               GameManager.instance.SaveGame.currentStage;

          foreach (Sala sala in salas) {
               sala.IsEnable = false;
          }
          CheckAvaliableStages();
     }

     private void CheckAvaliableStages() {
          if(salas !=null && salas.Count > 0) {
               Sala[] arraySalas = salas.ToArray();
               Sala sala = null;
               for(int i = currentstage -1; i >= 0; i--) {
                    sala = arraySalas[i];
                    sala.IsEnable = true;
               }
          }
     }
}
```

Script 22 – SunRay.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SunRay  {
     private Vector3 oringinalPosicion;
     private Transform originPoint;
     public Transform OriginPoint {
          get {
               return this.originPoint;
          }
          set {
               this.originPoint = value;
          }
     }
     private GameObject reflectingObject;
     public GameObject ReflectingObject {
          get {
               return this.reflectingObject;
          }
          set {
               this.reflectingObject = value;
          }
     }
```

```
     private LineRenderer ray;
     private Transform endPoint;
     private LayerMask layerHit;

     public SunRay (Transform originPoint, LayerMask layerToCollider) {
          this.originPoint = originPoint;
          this.layerHit = layerToCollider;
     }

     public SunRay (Transform originPoint) {
          this.originPoint = originPoint;
     }

     Vector3 checkRayHit (Vector3 toPosition) {
          float maxDistance = Vector3.Distance(originPoint.position, this.endPoint.position);
          Vector3 direction = toPosition - originPoint.position;
          RaycastHit2D hit;
          LayerMask layerAux;

          if (originPoint.gameObject.tag.Equals ("SolarRayEmitter")) {
               hit = Physics2D.Raycast (originPoint.position + (direction * 0.02f),
                    direction,maxDistance,this.layerHit);
          } else {
               layerAux = ( (1 << 2) | (1 << 8) | (1 << 11) | (1 << 12));
               layerAux = ~layerAux;
               hit = Physics2D.Raycast (originPoint.position + (direction * 0.02f),
                    direction,maxDistance,layerAux);
          }

          if (hit.collider != null) {
               if (hit.transform.gameObject.tag.Equals ("RefractoryObject")) {
                    this.reflectingObject = hit.transform.gameObject;
               } else {
                    this.reflectingObject = null;
               }

               toPosition =  hit.point;
          } else {
               this.reflectingObject = null;
          }
          return toPosition;
     }

     public void FixOrderLayerLineRender(Vector3 toPosition, float orderInLayer) {
          this.ray.sortingLayerName = "Default";
          this.ray.sortingOrder = Mathf.RoundToInt ((toPosition.y -300) * 100f) * -1;
     }

     public void CastSumRayReflection(GameObject raioSolarInstance, Vector3 toPosition) {
          SetRayPoint (raioSolarInstance);
          raioSolarInstance.transform.position = originPoint.position;
          this.endPoint.position = checkRayHit(toPosition);
          FixOrderLayerLineRender (this.endPoint.position, 0);
          this.ray.SetPosition (0, originPoint.position);
          this.ray.SetPosition (1, this.endPoint.position);
     }

     public void CastSumRayEmitter(GameObject raioSolarInstance, Vector3 toPosition,
          float orderInLayer) {
          SetRayPoint (raioSolarInstance);
          raioSolarInstance.transform.position = originPoint.position;
          FixOrderLayerLineRender (toPosition, orderInLayer);
          checkRayHit (toPosition);
          this.endPoint.position = toPosition;
          this.ray.SetPosition (0, originPoint.position);
          this.ray.SetPosition (1, toPosition);
     }

     void SetRayPoint(GameObject raioSolarInstance) {
          this.ray = raioSolarInstance.GetComponent<LineRenderer> ();
          this.endPoint = raioSolarInstance.transform.GetChild (0);
     }
}
```

Script 23 – SunRayEmitter.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[ExecuteInEditMode]
public class SunRayEmitter : MonoBehaviour {
     [SerializeField] private float distance = 0f;
     [SerializeField] private float angle = 0f;
     [SerializeField] private GameObject raioLuzPrefab = null;
     [SerializeField] private float orderInLayer = 0f;
     [SerializeField] private LayerMask layerToCollide;
     private SunRay sunRay = null;
     private GameObject raioLuzInstance = null;

     void Start () {
          if (raioLuzPrefab == null)
               Debug.LogWarning ("Componente RaioLuz não setado corretamente.");

          sunRay = new SunRay (transform,layerToCollide);
          Update();
     }

     void Update () {
          transform.rotation = Quaternion.Euler (new Vector3 (0, 0, angle));
          Vector3 point1 = transform.position + transform.right * distance;
          point1 = new Vector3 (point1.x, point1.y, transform.position.z);

          if (!Application.isPlaying) {
               CastRayDebug (point1);
          } else {
               CastRay(point1);
          }
     }

     void CastRay(Vector3 toPoint) {
          if(raioLuzInstance == null)
               raioLuzInstance = Instantiate (raioLuzPrefab, transform);
          if (sunRay != null) {
               sunRay.CastSumRayEmitter (raioLuzInstance, toPoint, orderInLayer);
          }
     }

     void CastRayDebug(Vector3 toPoint) {
          Debug.DrawLine (transform.position, toPoint, Color.red);
     }
}
```

Script 24 – TimerUI.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class TimerUI : MonoBehaviour {
     [SerializeField] private float minTimeToActive = 0f;
     [SerializeField] private Image timerUIFill;
     private float timer = 0f;
     private bool active = false;

     void Update () {
          if (active) {
               timer += Time.deltaTime;
          }
          ShowUI(active);
     }

     private void OnEnable()      {
          timerUIFill.fillAmount = 0;
          active = true;
          timer = 0f;
     }
```

```
     void ShowUI(bool value) {
          if (timerUIFill == null)
               return;
          if (value) {
               timerUIFill.fillAmount = timer / minTimeToActive;
          } else {
               timerUIFill.fillAmount = 0;
          }
     }

     public bool IsDone() {
          if (timer >= minTimeToActive) {
               return true;
          }
          return false;
     }
}
```

Script 25 – Tiro.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Tiro : MonoBehaviour {
     [SerializeField] private Transform spownBullet;
     [SerializeField] private GameObject bulletPrefab;
     [SerializeField] private float bulletSpeed = 1f;
     [SerializeField] private float timeBetwinBullets = 0.5f;
     [SerializeField] private Direction direction = 0;
     private enum Direction { LEFT, RIGHT, UP, DOWN }
     private float timer = 0f;
     public bool pauseTiro = false;

     void Update () {
          if (pauseTiro)
               return;

          timer += Time.deltaTime;
          if (timer >= timeBetwinBullets) {
               GameObject obj = Instantiate(bulletPrefab, transform);
               obj.transform.localScale = transform.localScale;
               obj.GetComponent<Bullet>().Create(bulletSpeed, spownBullet.position,
                    GetDirection());
               timer = 0f;
          }
     }

     public void SerPauseTiro(bool value) {
          pauseTiro = value;
     }

     private Vector3 GetDirection() {
          Vector3 direct = Vector3.zero;
          if (direction == Direction.LEFT) {
               direct = Vector3.left;
          } else if (direction == Direction.RIGHT) {
               direct = Vector3.right;
          } else if (direction == Direction.UP) {
               direct = Vector3.up;
          } else if (direction == Direction.DOWN) {
               direct = Vector3.down;
          }
          return direct;
     }
}
```