UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Guilherme Raimondi

O uso de uma engine de jogos para criação de um jogo digital educativo integrado em uma plataforma de acompanhamento de aprendizagem

Uberlândia, Brasil

2023

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Guilherme Raimondi

O uso de uma engine de jogos para criação de um jogo digital educativo integrado em uma plataforma de acompanhamento de aprendizagem

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Rafael Dias Araújo

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2023

Guilherme Raimondi

O uso de uma engine de jogos para criação de um jogo digital educativo integrado em uma plataforma de acompanhamento de aprendizagem

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 01 de dezembro de 2023:

Rafael Dias Araújo
Orientador

Renan Gonçalves Cattelan
Membro da Banca

Renato de Aquino Lopes
Membro da Banca

Uberlândia, Brasil
2023

Dedico este trabalho à minha família, pelo amor e apoio. Aos amigos pelos risos e dificuldades compartilhadas. Aos professores que me encorajaram e ensinaram. E ao meu orientador por ter me guiado e auxiliado.

Agradecimentos

Gostaria de expressar minha gratidão a todas as pessoas que contribuíram para a realização deste trabalho e para a conclusão bem-sucedida deste projeto acadêmico.

Primeiramente, agradeço ao meu orientador por me auxiliar e guiar em todo o processo, por me incentivar a segui-lo, bem como pela empatia e prestatividade. Sua orientação foi essencial para o desenvolvimento e qualidade deste trabalho.

Quero agradecer também a todos os professores, que me ensinaram e inspiraram durante o curso. Sem o conhecimentos passados por eles eu não teria as competências necessárias para realizar este trabalho.

Aos colegas de curso, agradeço pela troca de ideias e experiências, pela colaboração constante e pelo apoio mútuo nos momentos desafiadores. Tudo isso foi essencial para o avanço no curso e para culminar na apresentação deste trabalho.

Também gostaria de estender minha gratidão à Faculdade de Ciência da Computação e à Universidade Federal de Uberlândia como um todo, por fornecerem os recursos necessários para a realização desta pesquisa e pela infraestrutura que tornou possível a condução deste estudo.

À minha família, amigos e namorada, que sempre me apoiaram ao longo desta jornada acadêmica. Seu amor, incentivo e compreensão foram fundamentais para me manter no caminho, superar os desafios e chegar até aqui.

Não menos importante, agradeço a mim mesmo pela dedicação e esforço em, mesmo diante das dificuldades e desvios de rota, ter chegado à concretização deste trabalho. Este momento representa não apenas uma conquista acadêmica, mas a conclusão de uma etapa de vida e o início de outra.

Por fim, a todos aqueles que, de alguma forma, contribuíram para esta jornada, meu sincero obrigado. Este trabalho não teria sido possível sem o apoio e colaboração de cada um de vocês.

Resumo

Os jogos emergem como uma abordagem eficaz para potencializar o processo de ensino, proporcionando engajamento, interatividade e estimulando a aprendizagem ativa. Neste contexto, o presente trabalho faz parte de um projeto maior, a plataforma de aprendizagem de química QuimiCot Games, que busca auxiliar no ensino através de ferramentas lúdicas como os jogos com caráter educativo. Com o intuito de auxiliar nesse processo, esta monografia tem o objetivo de abordar aspectos práticos da criação de um jogo para apoio no ensino de química e a integração com a referida plataforma. Serão também avaliados particularidades técnicas como dificuldades encontradas, bem como pontos positivos e negativos do uso de uma *engine* nesse processo a fim de entender as nuncias desse ferramental e poder auxiliar em futuras outras integrações.

Palavras-chave: jogo digital educativo, educação em Química, plataforma de ensino, engine de jogos, Unity

Lista de ilustrações

Figura 1 – Pirâmide da aprendizagem de William Glasser 14
Figura 2 – Exemplo de vários *assets* em um projeto Unity 23
Figura 3 – Exemplo de organização e estruturação de uma cena no Unity 24
Figura 4 – Quatro tipos diferentes de Game Object: um personagem animado, uma luz, uma árvore e uma fonte de áudio 25
Figura 5 – Game Object representando o personagem do jogador, com vários componentes ligados à ele 26
Figura 6 – Exemplo de Prefab de árvore 28
Figura 7 – Diagrama contendo as etapas iterativas para criação do jogo 31
Figura 8 – Tabela periódica. Fonte: (IUPAC, 2023). 33
Figura 9 – Esboço do funcionamento geral do level 1 35
Figura 10 – Esboço do funcionamento geral do level 2 36
Figura 11 – Esboço do funcionamento geral do level 3 36
Figura 12 – Input Actions 38
Figura 13 – Tilemaps configurados para este jogo 41
Figura 14 – Exemplo de *Serialize Field* 42
Figura 15 – Criação da armadilha 44
Figura 16 – Criação do inimigo 45
Figura 17 – Criação da moeda vermelha 48
Figura 18 – Exemplo de sprite múltipla 51
Figura 19 – Exemplo de sprite utilizado no objeto do jogador 51
Figura 20 – Tilemap para criação de plataformas 52
Figura 21 – Animação de correr 53
Figura 22 – Animator Controller do personagem do jogador 54
Figura 23 – Configuração da ação atrelada ao clique do botão de realizar escolhas . 60
Figura 24 – Mecânica de diálogo e NPC 60
Figura 25 – Mecânica de questionário 64
Figura 26 – Abismo do nível 1 65
Figura 27 – Mecânica da ponte e do abismo no nível 1 66
Figura 28 – Mecânica do raio e do guarda-chuva no nível 2 68
Figura 29 – Mecânica principal do último nível concluída 73
Figura 30 – Configuração de build do jogo para WebGL 79
Figura 31 – Arquivos gerados pelo build de um jogo feito em Unity para WebGL . 80
Figura 32 – Captura de tela da plataforma QuimiCot Games com o jogo proposto integrado 81
Figura 33 – Loja de Assets do Unity 87

Figura 34 – Visual Scripting no Unity 88

Lista de códigos fonte

1 Exemplo de um script base em Unity . . . 27
2 Função que implementa a ação de mover . . . 39
3 Base para o script que controla a HP do jogador . . . 42
4 Script que torna-o um ouvinte do evento de dano . . . 43
5 Função que reduz a HP do jogador . . . 43
6 Função que realiza a ação de morte do jogador . . . 43
7 Script causador de danos . . . 44
8 Script de movimentação lateral automática . . . 45
9 Implementação da inversão de direção do movimento . . . 46
10 Script de coletáveis . . . 48
11 Evento de coleta . . . 48
12 Evento de coleta de moedas . . . 48
13 Campo de quantidade atual de moedas na UI . . . 49
14 Implementação para ouvir o evento de coleta de moedas . . . 49
15 Função que aumenta o score de moedas . . . 49
16 Parâmetro de clip de áudio . . . 55
17 Implementação para reproduzir áudio em um ponto . . . 56
18 Script para carregar próximo nível . . . 56
19 Exemplo de função trigger para próximo nível . . . 57
20 Exemplo de linguagem Ink para narrativa de jogos . . . 57
21 Uso de colisão trigger para alterar booleano . . . 58
22 Função base para entrar no modo diálogo . . . 58
23 Função atualizada para entrar no diálogo . . . 59
24 Função para sair do diálogo . . . 59
25 Função para mostrar as opções de diálogo . . . 59
26 Função para realizar escolha em um diálogo . . . 60
27 Script para retornar questionário . . . 62
28 Função para começar questionário . . . 62
29 função para criar questionário . . . 62
30 Função para enviar resposta fixa do questionário . . . 63
31 Função para adicionar gabarito ao questionário . . . 64
32 Função para disparar o questionário . . . 64
33 Script para habilitar objeto diante do evento de resposta correta . . . 65
34 Script para calcular distância entre o chão e a nuvem . . . 67
35 Script para criar raio . . . 67
36 Função para encerrar raio . . . 67

37 Script para criar objeto coletável 68
38 Base do script para polos que abrirão a porta 69
39 Funções que controlam o aparecimento de um objeto visual para indicar interatividade. 69
40 Função que recebe input para interagir com polo 70
41 Script de gerenciador de polos para ativação de porta 70
42 Função para gerenciar a ativação de polos 71
43 Trecho de função que lida com conexão bem sucedida de polos 72
44 Trecho de função que lida com conexão mal sucedida de polos 72
45 Trecho de script que lida com tentativas de conexões de polos 72
46 Script que valida token da plataforma QuimiCot Games 75
47 Script que loga informação na plataforma QuimiCot Games 75
48 Exemplo de implementação de log de informações 76
49 Função que obtém questionário da plataforma QuimiCot Games 76
50 Função atualizada de criação de questionário 77
51 Função que envia resposta do questionário à plataforma QuimiCot Games . 77
52 Função que adiciona resposta da plataforma QuimiCot Games ao questionário 78

Lista de abreviaturas e siglas

2D *Two-Dimensional* (Bidimensional)

3D *Three-Dimensional* (Tridimensional)

AI *Artificial Intelligence* (Inteligência Artificial)

API *Application Programming Interface* (Interface de Programação de Aplicações)

CPU *Central Processing Unit* (Unidade Central de Processamento)

CSS *Cascading Style Sheets* (Folhas de Estilo em Cascata)

GPU *Graphics Processing Unit* (Pontos de Vida)

HDR *High Dynamic Range* (Alto Alcance Dinâmico)

HP *Health Points* (Unidade de Processamento Gráfico)

HTML *Hypertext Markup Language* (Linguagem de Marcação de Hipertexto)

HTTP *Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto)

ID *Identification* (Identificação)

JSON *JavaScript Object Notation* (Notação de Objeto JavaScript)

NPC *Non-Player Character* (Personagem Não-Jogável)

UI *User Interface* (Interface do Usuário)

URL *Uniform Resource Locator* (Localizador Uniforme de Recursos)

Sumário

1 INTRODUÇÃO . . . 14
1.1 Justificativa . . . 16
1.2 Objetivos . . . 16
1.3 Contribuições . . . 16
1.4 Organização da Monografia . . . 17

2 REVISÃO BIBLIOGRÁFICA . . . 18
2.1 Jogos digitais educacionais . . . 18
2.2 Game Engines . . . 19
2.2.1 Principais engines para desenvolvimento de jogos . . . 20
2.2.2 Por que Unity? . . . 21
2.2.3 Entendendo os conceitos básicos do Unity . . . 23
2.2.3.1 Assets . . . 23
2.2.3.2 Cenas . . . 24
2.2.3.3 Objetos de Jogo (Game Objects) . . . 25
2.2.3.4 Componentes . . . 25
2.2.3.5 Scripts (e MonoBehaviour) . . . 26
2.2.3.6 Prefabs . . . 27
2.3 Trabalhos Relacionados . . . 28
2.3.1 Comparativo entre *engines* de jogos . . . 28
2.3.2 Unity para o desenvolvimento de jogos . . . 29
2.3.3 Jogos educativos no ensino de Química . . . 29
2.3.4 Plataformas para monitoramento de aprendizagem em jogos . . . 30

3 DESENVOLVIMENTO . . . 31
3.1 Metodologia . . . 31
3.2 Proposta . . . 32
3.3 Concepção do jogo . . . 33
3.3.1 Visão geral da primeira fase . . . 34
3.3.2 Visão geral da segunda fase . . . 34
3.3.3 Visão geral da terceira e última fase . . . 34
3.4 Esboço do jogo . . . 35
3.4.1 Esboço da primeira fase . . . 35
3.4.2 Esboço da segunda fase . . . 36
3.4.3 Esboço da terceira e última fase . . . 36
3.5 Prototipagem das principais mecânicas do jogo . . . 37

3.5.1 Movimentação do jogador . . . 37
3.5.2 Configuração dos *tilemaps* para construção dos mapas de cada níveis . . . 39
3.5.3 Conceito de vidas e mortes para o jogador . . . 41
3.5.4 Armadilhas e inimigos . . . 43
3.5.5 Coletáveis . . . 46
3.5.6 Arte . . . 49
3.5.7 Animação . . . 52
3.5.8 Áudio . . . 54
3.5.9 Gerenciador de níveis . . . 56
3.5.10 Sistema de diálogo . . . 57
3.5.11 Sistema de questionário (*quiz*) . . . 61
3.6 Criação dos níveis e de suas especificidades . . . 65
3.6.1 Nível 1 - ponte e abismo . . . 65
3.6.2 Nível 2 - raios . . . 66
3.6.3 Nível 3 - porta e conexão de fios entre diferentes polos . . . 69

4 RESULTADOS E DISCUSSÕES . . . 74
4.1 Integração do jogo com as APIs da plataforma QuimiCot Games . . . 74
4.1.1 API de verificação da autorização do jogador . . . 74
4.1.2 API para salvar informações genéricas . . . 75
4.1.3 APIs para solicitar, enviar resposta e receber o gabarito do Quiz . . . 76
4.2 Inserção do jogo na plataforma QuimiCot Games . . . 78
4.2.1 Build do jogo para a plataforma Web . . . 78
4.2.2 Publicação do jogo na plataforma de jogos educativos QuimiCot Games . . . 80
4.3 Desafios encontrados ao longo do trabalho . . . 81
4.3.1 Entender o ecossistema de uma *engine* . . . 81
4.3.2 Aplicar princípios de padrões de arquitetura e design de software . . . 82
4.3.3 Integração com serviços externos não é intuitiva . . . 83
4.3.4 Bugs na própria *engine* . . . 84
4.4 Vantagens de utilizar uma *engine* de jogos . . . 85
4.4.1 Ferramentas e funcionalidades pré-construídas . . . 85
4.4.2 Loja de assets . . . 86
4.4.3 Rápida prototipagem . . . 87
4.4.4 Visual Scripting . . . 88
4.4.5 Multiplataforma . . . 89
4.4.6 Comunidade e suporte . . . 89
4.4.7 Ferramentas de depuração . . . 90
4.4.8 Gerenciamento de recursos e otimização . . . 90
4.4.9 Gráficos e efeitos visuais de alta qualidade . . . 91
4.4.10 Fácil adaptação à diferentes controles . . . 91

4.4.11 Plano gratuito para jogos educativos 92
4.5 Desvantagens de utilizar uma *engine* de jogos 92
4.5.1 Dependência com a própria *engine* 93
4.5.2 Flexibilidade limitada 93
4.5.3 Desempenho 93
4.5.4 Multiplataforma 94
4.5.5 Curva de aprendizagem 94
4.5.6 *Scripting* limitado à algumas linguagens de programação 95

5 CONCLUSÃO 96
5.1 Principais contribuições 96
5.2 Limitações 97
5.3 Trabalhos futuros 98

REFERÊNCIAS 99

1 Introdução

As metodologias de aprendizagem são os modelos utilizados pelos educadores para que os alunos sejam capazes de se desenvolverem e ampliarem os seus conhecimentos (BARBOSA; MOURA, 2013). Elas podem variar de acordo com os objetivos educacionais, o público-alvo e o contexto em que são aplicadas, mas todas têm em comum o objetivo de tornar o processo de ensino mais efetivo.

Em geral, as instituições de ensino utilizam majoritariamente a forma passiva de aprendizado, na qual o professor por meio de aulas expositivas ensina e o aluno é, de certa forma, o sujeito passivo, recebendo essas informações (BARBOSA; MOURA, 2013). Já as estratégias que promovem aprendizagem ativa podem ser definidas como sendo atividades que ocupam o aluno em fazer alguma coisa e, ao mesmo tempo, o leva a pensar sobre as coisas que está fazendo (SILBERMAN, 1996), ou seja, na metodologia ativa o estudante é desafiado a pensar, resolver problemas, tomar decisões e aplicar o conhecimento de forma prática, com o intuito de engajá-lo, torná-lo mais ativo e independente no processo de aprendizagem.

Existem certos métodos de ensino que facilitam a assimilação de conhecimento. De acordo com a teoria do psiquiatra americano William Glasser, os alunos aprendem cerca de 10% lendo; 20% escrevendo; 50% observando e escutando; 70% discutindo com outras pessoas; 80% praticando; e 95% ensinando. Assim, podemos concluir que os métodos mais eficientes de aprendizagem estão inseridos nas metodologias ativas (Jovem Pan, 2023), conforme Figura 1.

Figura 1 – Pirâmide da aprendizagem de William Glasser.
Fonte: www.abrafi.org.br/index.php/site/noticias/ver/4349

O recente avanço e popularização da tecnologia fez com que os jogos ganhassem cada vez mais espaço como forma de aprendizado. Além de lazer, elas também demonstraram ser uma ferramenta eficaz para engajar os alunos e promover a aprendizagem de forma lúdica e divertida, permitindo a construção de novos conhecimentos de forma mais efetiva e significativa (ALVES; BIANCHIN, 2010).

A utilização de jogos na educação se enquadra como metodologia ativa de aprendizagem, apresentando diferentes tipos de desafios e objetivos, que podem ser utilizados para estimular o pensamento crítico, a resolução de problemas, a colaboração, a comunicação, entre outras habilidades importantes. Além disso, pode ser aplicada em diversas áreas do conhecimento, desde a matemática e a física até as ciências sociais e humanas, seja para a educação infantil ou até para a formação de profissionais e, assim, podem ser adaptados de acordo com os objetivos educacionais e o público-alvo.

Dessa forma, os jogos como uma metodologia ativa de aprendizagem têm se mostrado cada vez mais eficazes, sendo utilizados em escolas e universidades ao redor do mundo, contribuindo para uma educação mais dinâmica, eficiente e engajadora (ALVES; BIANCHIN, 2010).

Porém, são necessários mecanismos para aferir e auxiliar esse processo de aprendizado e aqui entram as plataformas de jogos educacionais. Elas permitem que os professores monitorem o desempenho dos alunos em tempo real, o que possibilita um acompanhamento mais preciso e uma intervenção mais eficiente quando necessário. Além disso, as plataformas de jogos educacionais podem ser utilizadas para complementar ou substituir atividades tradicionais de ensino, tornando o processo de aprendizagem mais dinâmico e eficiente.

No âmbito deste trabalho, destaca-se a utilização da plataforma QuimiCot Games enquanto um ecossistema de software que possibilita esse monitoramento e acompanhamento do desempenho dos alunos. Nessa plataforma os professores atuam como criadores diretos de versões personalizadas de jogos digitais com o objetivo de apoiar o ensino contextualizado da disciplina de Química. Além disso, permite que os alunos utilizem esses jogos durante seu processo de aprendizagem, gerando informações que são enviadas de volta para o professor para monitoramento do aprendizado (DAIREL et al., 2021).

A plataforma QuimiCot Games atualmente possui alguns jogos desenvolvidos nativamente na linguagem de programação Javascript e a proposta deste trabalho é avaliar as dificuldades, vantagens e desvantagens da criação de um jogo e integração à plataforma utilizando uma *engine* de jogos, que é um motor de jogos que oferecem facilidades para acelerar o desenvolvimento, publicação e demais processos envolvidos.

1.1 Justificativa

Na criação de um jogo cada ferramenta possui suas vantagens, desvantagens, limitações e facilitadores que devem ser analisados caso a caso. A avaliação da melhor ferramenta é crucial por vários motivos. A escolha de uma ferramenta adequada pode influenciar diretamente na qualidade final do jogo, na jogabilidade e na experiência do jogador. Pode também influenciar no tempo de desenvolvimento e no tempo pós desenvolvimento para manutenção e correção de erros, gerando possíveis custos adicionais.

Outro fator a ser considerado é o nível de complexidade do jogo e a plataforma que esse jogo vai ser executada, pois, caso seja multiplataforma e a ferramenta não ofereça isso, pode gerar retrabalho e, no pior cenário, ser necessário desenvolver tudo novamente para essa outra plataforma.

Ademais, quando pretende-se integrar com uma plataforma de jogos é necessário levar em conta as habilidades de desenvolvimento da equipe, bem como a compatibilidade do sistema atual com a ferramenta escolhida.

1.2 Objetivos

Nesse sentido, o objetivo geral do trabalho é criar um jogo digital educativo usando uma *engine* de jogos e integrá-lo com a plataforma QuimiCot Games. Com o intuito de alcançar o objetivo geral e para que ele seja claro e mensurável, o presente trabalho possui os seguintes objetivos específicos:

- elencar as etapas gerais para o desenvolvimento de um jogo.
- demonstrar como integrar com uma plataforma externa de monitoramento de aprendizagem utilizando a *engine*.
- elencar os desafios, vantagens e desvantagens de utilizar uma *engine* pra o processo de desenvolvimento e integração do jogo.

1.3 Contribuições

Um jogo desenvolvido como parte dessa monografia pode contribuir para o campo de aprendizado, oferecendo uma abordagem prática para a criação de conteúdo educacional em formato de jogo. Além disso, utilizar uma *engine* para esse desenvolvimento oferece uma série de vantagens e desvantagens e a escolha de utilizá-la pode contribuir significativamente em todo o processo, hipótese que será analisada no final do trabalho.

Além disso, a escolha de uma boa ferramenta para o caso em questão pode gerar impacto também na qualidade final do produto e na experiência do usuário. Quando

pensa-se que o mercado de jogos está cada vez mais aquecido, jogos que ofereçam jogabilidade e experiências melhores tendem a reter mais usuários e isso é ainda mais importante no que tange a jogos educativos, que são focados em gerar aprendizado. Quanto mais o usuário quiser interagir, mas eficaz será o jogo e maior será a probabilidade de gerar conhecimento a partir dele.

1.4 Organização da Monografia

O restante da monografia está organizado da seguinte maneira. O segundo capítulo, de revisão bibliográfica, tem como objetivo apresentar um panorama crítico e abrangente das principais teorias, pesquisas e conhecimentos existentes relacionados ao tema da pesquisa. Aqui serão abordados temas como jogos educacionais, *game engines* e trabalhos relacionados, além trabalhar alguns elementos e conceitos gerais.

Já no terceiro capítulo, de desenvolvimento, estarão presentes os principais tópicos relacionado às etapas da criação do jogo em si, desde concepção, esboço, prototipagem e criação de cada nível. Esse capítulo proporcionará uma visão aprofundada do processo prático de criação do jogo, vinculando a teoria à prática.

Depois disso, no quarto capítulo, de Resultados e Discussões, será onde, pós concluído o desenvolvimento do jogo, será realizada a integração com a plataforma de jogos educacionais QuimiCot Games e assim, serão analisados e apresentados os dados obtidos a partir deste trabalho, demonstrando as contribuições para a área de estudo. Alguns temas abordados serão as principais dificuldades observadas ao longo do trabalho, bem como as vantagens e desvantagens gerais de se trabalhar com uma *engine* de jogos na criação e integração com uma plataforma existente.

Por fim, no quinto e último capítulo serão abordadas as conclusões da monografia, demonstrando as contribuições do trabalho, as limitações, a fim de dar uma visão equilibrada e realista do estudo, bem como as oportunidades e tópicos possíveis de pesquisas futuras relacionados ao que foi aqui trabalhado.

2 Revisão Bibliográfica

2.1 Jogos digitais educacionais

Os jogos tem sido uma forma de proporcionar entretenimento e diversão há séculos. Os jogos digitais por sua vez tiverem início na década de 1950 quando pesquisadores e cientistas estavam começando a criar os computadores. Ao explorar essas ideias e formas de interação, surgiu o primeiro jogo digital conhecido como "Tênis para Dois"(Tennis for Two), criado em 1958 por William Higinbotham no Laboratório Nacional de Brookhaven, nos Estados Unidos. Esse jogo permitia que dois jogadores controlassem barras em uma tela e jogassem uma partida de tênis virtual (Wikipédia, 2023b). Posteriormente, na década de 1970, a indústria de jogos começou a se desenvolver com o lançamento do primeiro console de videogame doméstico, o Magnavox Odyssey, o que permitiu acesso da população e impulsionou sua expansão e rápido crescimento na popularidade (Luann Motta Carvalho, 2023).

A maior parte dos jogos são os conhecidos como jogos de entretenimento e são projetados com esse objetivo: entreter, divertir, trazer uma experiência cativante e lúdica. Todavia, recentemente uma outra categoria de jogos tem ganhado força: os jogos educativos. Estes tentam trazer todas as características supracitadas assim como serem uma ferramenta para auxiliar no aprendizado, combinando educação e diversão.

É interessante ressaltar que os jogos educativos apesar de terem reconquistado força nos dias atuais, possuem origem antiga, atrelada com o próprio surgimento dos jogos. Segundo o Portal da Educação, pesquisas revelam que a origem dos jogos surgiu no século XVI na Roma e na Grécia e possuíam o propósito de ensinar letras. Porém, com o avanço do cristianismo, o interesse decresceu, pois tinham um visão de educação mais rígida, disciplinadora, com foco em obediência e voltada para memorização (Portal Educação, 2023).

Segundo Silveira Barone, “os jogos podem ser empregados em uma variedade de propósitos dentro do contexto de aprendizado. Um dos usos básicos muito importante é a possibilidade de construir-se a autoconfiança. Outro é o incremento da motivação. (...) um método eficaz que possibilita uma prática significativa daquilo que está sendo aprendido. Até mesmo o mais simplório dos jogos pode ser empregado para proporcionar informações factuais e praticar habilidades, conferindo destreza e competência” (SILVEIRA; BARONE, 1998).

Dessa forma, podemos ver que os jogos educativos podem ser um ótimo recurso didático para auxiliar no processo de ensino e aprendizagem. Lara afirma que os jogos,

ultimamente, vêm ganhando espaço dentro das escolas, numa tentativa de trazer o lúdico para dentro da sala de aula. Acrescenta que a pretensão da maioria dos professores com a sua utilização é a de tornar as aulas mais agradáveis com o intuito de fazer com que a aprendizagem torne-se algo mais fascinante; além disso, as atividades lúdicas podem ser consideradas como uma estratégia que estimula o raciocínio, levando o aluno a enfrentar situações conflituantes relacionadas com o seu cotidiano (LARA, 2004).

Vale ressaltar que para um jogo educacional ser eficaz, são necessários algumas peculiaridades que outros categorias não possuem. Muitos jogos exploram conceitos extremamente triviais e não tem a capacidade de diagnosticar as falhas do jogador. A maneira com que os jogos educacionais contornam estes problemas é fazendo com que o aprendiz, após uma jogada que não deu certo, reflita sobre a causa do erro e tome consciência do erro conceitual envolvido na jogada errada, tornando o processo de aprendizado mais eficaz (VALENTE; ALMEIDA, 1997).

Diante dessas análises, percebe-se a importância da interação e mediação dos educadores, fundamentais neste processo para que os objetivos dos jogos não passem a ser unicamente concluir o jogo, deixando de lado as questões de aprendizagem. Além disso, faz-se necessária uma relação próxima entre os desenvolvedores de jogos educativos e os profissionais da educação, com o objetivo de trabalharem junto e encontrarem formas de combinar o prazer de jogar com as melhores práticas educacionais a fim de trazer esse conhecimento específico.

Portanto, os jogos além de proporcionarem diversão e enfatizarem o aspecto lúdico podem ser utilizados de forma pedagógica, auxiliando na construção e na fixação de conhecimento. Os jogos são destacados como uma ferramenta adicional a ser desenvolvida e explorada com os alunos, contribuindo positivamente para o processo de ensino e aprendizagem. Quando utilizados corretamente e com a mediação dos educadores, eles se tornam uma ferramenta poderosa e transformadora na educação.

2.2 Game Engines

Criar um jogo é um grande desafio, tanto para a empresa ou instituição, quanto para os engenheiros de software e desenvolvedores atuando nele. Trata-se de um processo extremamente complexo, que envolve vários fatores, desde trabalhar na criação de mecânicas do jogo, simular física entre objetos, cenário e personagens, integrar a arte com o jogo, criar animações, trabalhar com iluminação, gerir recursos computacionais, trabalhar em obter uma boa performance para o sistema operacional que irá executar o jogo, entre diversos outros.

No entanto, uma boa ferramenta de desenvolvimento pode ser crucial nesse processo tornando-se uma grande aliada. E, por isso, a primeira dúvida seria se vale a pena

ou não utilizar uma *engine* de jogos para auxiliar nesse processo como um todo.

Uma *engine* de desenvolvimento de jogos é uma ferramenta de software que oferece uma série de recursos e funcionalidades para auxiliar na criação de jogos. Elas fornecem uma interface gráfica, bibliotecas de código, recursos de física, animação, áudio, entre outros, facilitando o processo de criação do jogo. As *engines* de desenvolvimento de jogos mais populares incluem Unity, Unreal Engine, Godot, GameMaker Studio e Cocos2d. Essas *engines* permitem criar jogos para diversas plataformas, como PC, consoles, dispositivos móveis e até mesmo realidade virtual.

Ademais, o uso de uma *engine* oferece uma ampla variedade de recursos e ferramentas para facilitar o trabalho dos desenvolvedores, auxiliando na produtividade, facilidade na criação de *assets* (como gráficos e sons) e suporte para várias plataformas. Além disso, essas *engines* possuem comunidades ativas, tutoriais e recursos disponíveis, facilitando o aprendizado e o compartilhamento de conhecimentos.

Já um desenvolvimento nativo, por exemplo, envolve a criação do jogo usando as linguagens de programação e ferramentas específicas para cada plataforma em que o jogo será executado. Por exemplo, para a criação de um jogo na plataforma iOS pode-se usar Swift com as ferramentas de desenvolvimento da Apple, como o Xcode. Já para a criação de um jogo Web pode-se usar HTML 5 com Javascript, etc.

O desenvolvimento nativo oferece controle total sobre o jogo e pode permitir melhor otimização e desempenho, especialmente para jogos mais complexos. Porém, o contrário também é possível, nem sempre existe esse ganho de performance só por ser nativo. Por isso, ele requer conhecimento mais aprofundado das linguagens de programação e das peculiaridades de cada plataforma. Isso pode exigir mais tempo e esforço para desenvolver e manter o jogo em diferentes plataformas.

2.2.1 Principais engines para desenvolvimento de jogos

Existem várias engines populares para criar jogos, cada uma com suas próprias características e recursos. Apesar dos recursos serem similares, atualmente as engines de jogos apresentam características distintas e a escolha entre elas torna-se um ponto importante no desenvolvimento de um projeto.

Algumas das principais engines que estão em alta no mercado são:

1. Unity: é uma das engines mais populares e amplamente utilizadas na indústria de jogos. Ela suporta o desenvolvimento de jogos 2D e 3D, possui uma grande comunidade de desenvolvedores e oferece recursos abrangentes para criação de jogos para várias plataformas, incluindo PC, consoles, dispositivos móveis e realidade virtual.

2. Unreal Engine: Desenvolvida pela Epic Games, também é uma engine de alto

desempenho usada para criar jogos em 2D e 3D. Ela oferece recursos avançados de renderização, física e animação, e é conhecida por seu poderoso mecanismo gráfico. A Unreal Engine também suporta várias plataformas, incluindo PC, consoles, dispositivos móveis e realidade virtual.

3. Godot: é uma engine de código aberto e gratuita que ganhou popularidade nos últimos anos. Ela é conhecida por sua facilidade de uso e é adequada para desenvolvedores iniciantes e experientes. O Godot suporta o desenvolvimento de jogos 2D e 3D e possui uma linguagem de script própria chamada GDScript, além de suportar outras linguagens de programação, como C#.

4. GameMaker Studio: é uma engine popular para desenvolvimento de jogos 2D. Ela oferece uma interface visual intuitiva e uma linguagem de script fácil de aprender chamada GML (GameMaker Language). O GameMaker Studio é conhecido por sua facilidade de uso e é adequado para desenvolvedores independentes e iniciantes.

5. Cocos2d: é uma engine de código aberto focada em jogos 2D para dispositivos móveis. Ela oferece suporte a várias plataformas, incluindo iOS, Android e HTML5. O Cocos2d é escrito em C++ e possui uma interface simples e poderosa, tornando-o uma escolha popular para desenvolvedores de jogos mobile.

Essas são apenas algumas das principais engines disponíveis, e a escolha da melhor depende das suas necessidades, preferências e do tipo de jogo que deseja-se criar. Cada engine tem suas vantagens e desvantagens, então é importante avaliar os cenários específicos dos desenvolvedores e do projeto para encontrar a mais adequada.

2.2.2 Por que Unity?

Mesmo com o grande número de engines de jogos disponíveis, algumas se destacam por características que facilitam o desenvolvimento. Para este trabalho, a engine escolhida foi o Unity, que é um motor ou uma plataforma de desenvolvimento de jogos criado pela Unity Technologies. O Unity teve sua primeira versão lançada em 2005 após seu anúncio na Apple Worldwide Developers Conference e desde então tem mudado e evoluído constantemente com auxílio dos desenvolvedores da empresa e da comunidade.

O Unity é atualmente uma das engines mais utilizadas no mundo pela eficiência e facilidade de desenvolvimento apresentada (ALVES, 2020). Existem grandes marcos anunciado pelas própria empresa (Unity Technologies, 2023j) que demonstram a sua ampla adesão no mercado, como:

- Mais de 4 bilhões de downloads por mês de aplicativos são feitos com o Unity.
- 70% dos 1.000 principais jogos para dispositivos móveis foram feitos com o Unity.
- Mais de 17 plataformas diferentes executam as criações desenvolvidas no Unity

O fato de ser uma engine com grande respaldo e uso no mercado demonstra um bom nível de maturidade, além de uma base de usuários alta. Isso influencia diretamente na facilidade de achar materiais de apoio, suporte, dúvidas e exemplos de implementações. Estes quesitos, apesar de parecerem simples, fazem toda a diferença na facilidade de usar uma ferramenta para que ela não seja mais um empecilho do que aliada durante o desenvolvimento. Quando comparamos Unity enquanto engine mais utilizada com o segundo colocado Unreal Engine, ambas as engines têm uma enorme comunidade de usuários ativos, mas o Unity atualmente representa quase 50% da participação de mercado, em comparação com 13% do Unreal (Simran Kaur Arora, 2023).

De forma análoga, levando em consideração o quesito facilidade de uso, o Unity foi considerado mais adequado para iniciantes porque possui uma interface de usuário mais simples, fornece muitos tutoriais e exemplos, além de existirem muitos assets disponíveis e sua instalação não requerer hardware de alto desempenho. Em relação, por exemplo, ao desenvolvimento de jogos móveis para Android o desenvolvimento é bastante simples e o processo de exportação é mais fácil (CHRISTOPOULOU; XINOGALOS, 2017).

Além disso, a quantidade de recursos oferecidos pela engine é um importante fator para levar em conta, à medida que facilita o trabalho e oferece maiores funcionalidades direto da engine. Segundo estudo, Unity e Unreal Engine 4 são os motores de jogo mais poderosos, que suportam quase todos os recursos incluídos na estrutura usada (CHRISTOPOULOU; XINOGALOS, 2017). Nesse sentido, o Unity destaca-se pela sua ampla gama de recursos dedicados ao desenvolvimento de jogos, especialmente 2D, que será o estilo adotado para o jogo do presente trabalho.

Outro ponto acessório, ou seja, não tão relevante, é que o Unity trabalha com a linguagem C# (C Sharp) que é uma linguagem de programação moderna, orientada a objetos e de propósito geral desenvolvida pela Microsoft. Por ser orientada à objetos facilita pra quem, como eu, vem de outras linguagens com propósitos similares como Java e Kotlin. Fora a isso, a documentação fornecida pela Microsoft pra essa linguagem é vasta, o que facilita o processo de desenvolvimento. Dito isso, vale enfatizar que a escolha da engine não foi feita pela linguagem de programação, mas ela ser próxima da que tenho mais experiência tornará a curva de aprendizado menos acentuada e o desenvolvimento do jogo mais rápido.

Portanto, escolher o Unity em detrimento de outras engines de desenvolvimento de jogos se baseia em uma combinação de fatores como maturidade no mercado, facilidade de uso, maior comunidade ativa, ampla gama de recursos oferecidos, interface amigável, por ser mais recomendado para iniciantes e pela versatilidade e suporte que essa plataforma oferece.

2.2.3 Entendendo os conceitos básicos do Unity

Nesta sessão estudaremos alguns conceitos básicos do Unity utilizando o proprio manual fornecido pelo Unity (Unity Technologies, 2023h). Compreender esses conceitos antes de iniciar o desenvolvimento é um passo fundamental que contribui significativamente para a otimização de tempo e sucesso do projeto.

A compreensão desses conceitos de forma prévia estabelece uma base sólida, permitindo aos desenvolvedores navegar de forma eficiente pelo ambiente de desenvolvimento. Ao entender como organizar cenas, utilizar *assets*, criar *scripts* e manipular objetos e etc. os desenvolvedores economizam tempo e minimizam desafios durante o processo de construção como um todo. Em última análise, investir tempo na assimilação dos conceitos básicos do Unity não apenas agiliza o desenvolvimento, mas também proporciona uma base sólida para a criação eficiente de um jogo utilizando essa ferramenta.

2.2.3.1 Assets

Um *asset* é qualquer item que possa ser usado em um projeto do Unity na criação de um jogo ou aplicativo. Os *assets* podem representar elementos visuais ou de áudio em seu projeto, como modelos 3D, texturas, sprites, efeitos sonoros ou música. Um *asset* pode vir de um arquivo criado fora do Unity, como um modelo 3D, um arquivo de áudio ou uma imagem. Também é possível criar alguns tipos de *assets* no editor do Unity, como um controlador de animação, um mixer de áudio ou uma textura de renderização (Unity Technologies, 2023a) por exemplo.

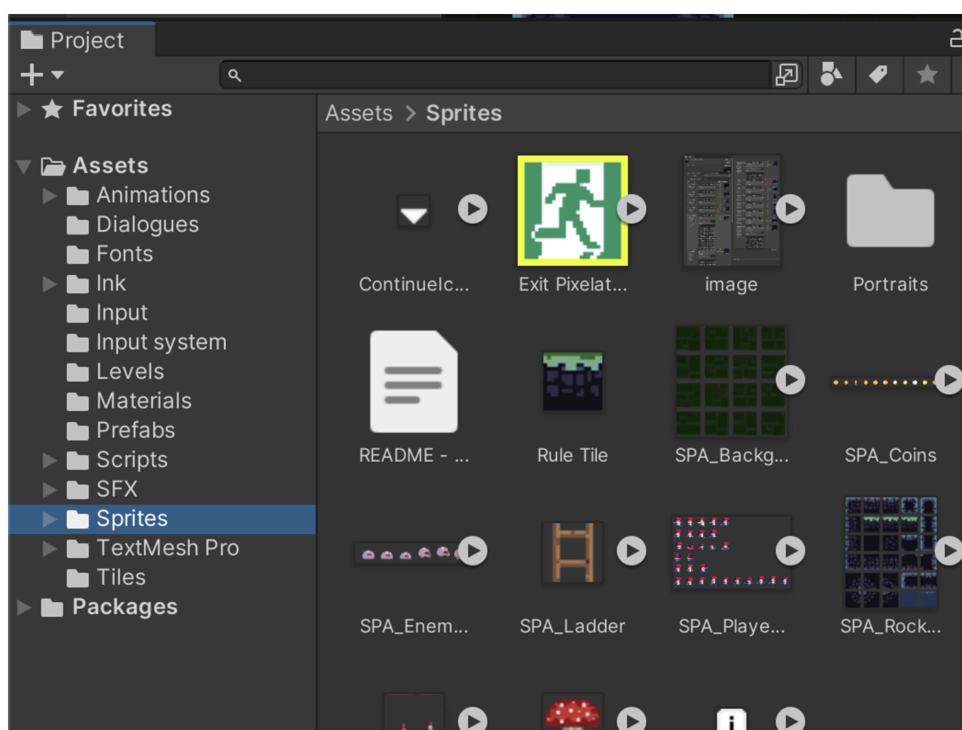

Figura 2 – Exemplo de vários *assets* em um projeto Unity

Em resumo, são elementos fundamentais do Unity utilizados na construção de um jogo e que podem assumir diversos usos. Eles facilitam a reutilização de recursos e promovem facilidade na colaboração entre membros da equipe ao abstrair esses conceitos. Vale ressaltar também que o Unity oferece uma loja de *assets* (*Unity Asset Store*) que amplia e facilita essa oferta, proporcionando acesso a uma vasta gama de recursos prontos para uso.

2.2.3.2 Cenas

Uma cena é basicamente a maior unidade oferecida pelo Unity para a organização dos objetos de um jogo. As cenas são onde trabalha-se com conteúdo no Unity. São *assets* que contêm todo ou parte de um jogo ou aplicativo. Por exemplo, pode-se construir um jogo simples em uma única cena, enquanto que para um jogo mais complexo, pode-se usar uma cena por nível, cada uma com seus próprios ambientes, personagens, obstáculos, decoração e UI (Unity Technologies, 2023i), conforme exemplo na Figura 3.

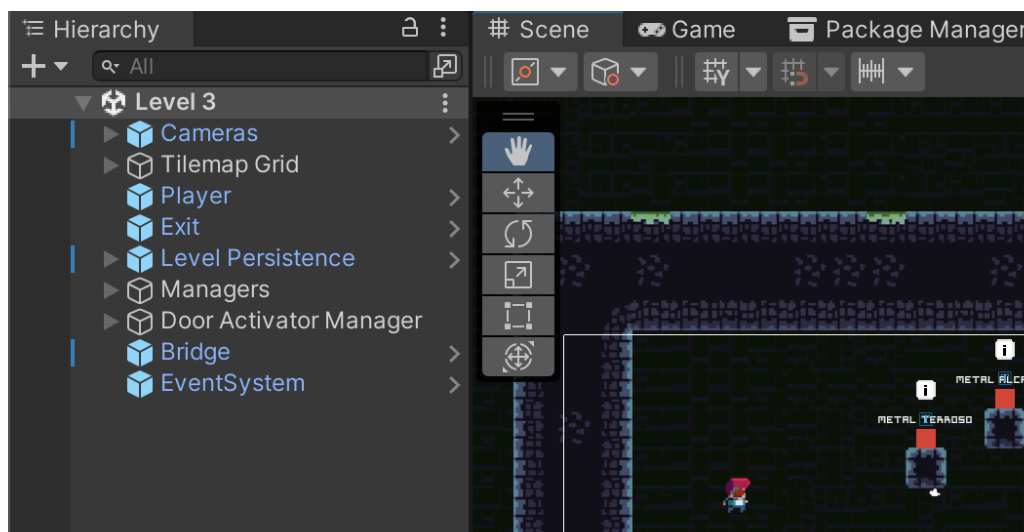

Figura 3 – Exemplo de organização e estruturação de uma cena no Unity

Isso significa que as cenas são cruciais para organizar e modularizar um projeto, permitindo que os desenvolvedores gerenciem eficientemente diferentes partes do seu jogo. De forma prática, em geral, uma cena é usada para representar um único nível ou unidade lógica de um jogo. Por isso, ao construir um jogo utilizando-se de várias cenas, é possível distribuir melhor os tempos de carregamento e testar diferentes partes do jogo individualmente. Ademais, novas cenas costumam ser feitas separadamente de uma cena de jogo já existente, a fim de prototipar ou testar uma parte nova ou funcionalidade em potencial.

As cenas no Unity também são cruciais para controlar o fluxo do jogo, a medida que podem ser carregadas e manipuladas usando as visualizações Hierarquia e Cena oferecidas pela engine. Isso permite que o jogo seja dividido em diferentes ambientes, níveis ou

seções e seja possível gerenciar a transição entre cada uma delas por meio da classe *SceneManager*. Vale ressaltar ainda, que ao navegar entre cenas, o Unity reinicia os objetos. Porém, em vários cenários é interessante preservar dados entre cenas, como quantidade de vidas do jogador, entre outros objetos ou atributos. Para isso o Unity oferece a classe *DontDestroyOnLoad* que mantém certos objetos ativos durante a transição de cenas.

2.2.3.3 Objetos de Jogo (Game Objects)

Cada objeto em um jogo é um Game Object, desde personagens e itens colecionáveis até luzes, câmeras e efeitos especiais. Entretanto, um Game Object não pode fazer nada sozinho; é preciso dar-lhe propriedades antes que ele se torne um personagem, um ambiente ou um efeito especial. Nesse sentido, GameObjects são os objetos fundamentais no Unity que representam personagens, adereços e cenários. Eles não realizam muito por si mesmos, mas agem como recipientes para Componentes, que implementam a funcionalidade (Unity Technologies, 2023c), conforme exemplo na Figura 4.

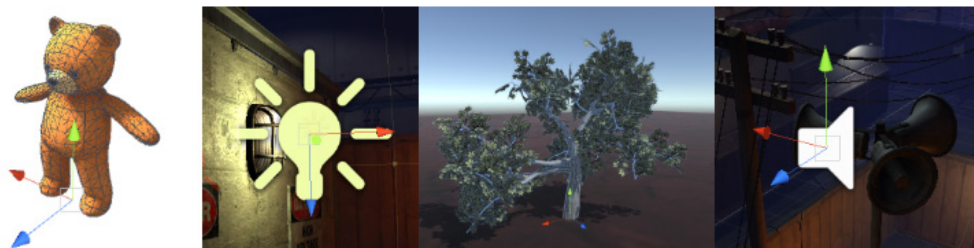

Figura 4 – Quatro tipos diferentes de Game Object: um personagem animado, uma luz, uma árvore e uma fonte de áudio

Em resumo, os Game Objects servem como entidades versáteis que compõem a base estrutural de qualquer projeto Unity, proporcionando a fundação para a criação de jogos de forma eficiente.

2.2.3.4 Componentes

Componentes são as peças funcionais de cada Game Object. Os componentes contêm propriedades que podem ser editadas para definir o comportamento de um Game Object (Unity Technologies, 2023d), ou seja, são peças contendo funcionalidade que podem ser adicionadas aos Game Objects para definir seu comportamento, aparência e interatividade, entre outros. Cada Component é responsável por uma tarefa específica e pode ser combinado com outros para criar objetos complexos.

Na figura abaixo temos um Game Object que representa o personagem do jogador e nele estão anexados alguns componentes que dão certas funcionalidades ao jogador. Aqui vale ressaltar o aspecto geral de cada um para servir de exemplo e dar um contexto das possibilidades de funcionalidades que um componente oferece. O primeiro é o componente de Transform, presente em todos os objetos do Unity e que determina a localização, rotação

e escala do GameObject. Depois temos o Sprite Renderer que controla como ele aparecerá visualmente em um cena. Já o Animator, gerencia as lógicas e estados de animação. Por fim, o Capsule Collider 2D é um componente capaz de dar funcionalidades relacionadas a colisão enquanto o Rigidbody é um componente que fornece uma forma para controlar o movimentação do objeto baseado em conceitos da física, conforme exemplo na Figura 5.

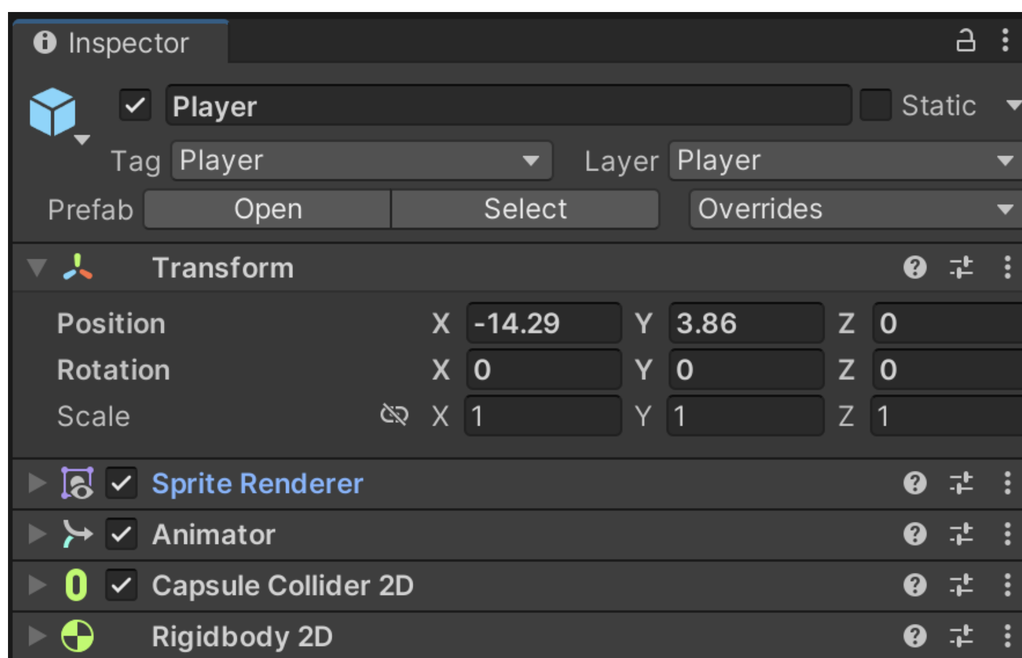

Figura 5 – Game Object representando o personagem do jogador, com vários componentes ligados à ele

Portanto, a combinação e configuração desses componentes permite aos desenvolvedores criar uma ampla variedade de comportamentos e interações em seus jogos. A modularidade dos componentes é fundamental para o paradigma de desenvolvimento no Unity, tornando possível segregar funcionalidades e permitir criar jogos complexos de forma organizada e escalável.

2.2.3.5 Scripts (e MonoBehaviour)

Conforme visto anteriorment, o comportamento de Game Objects é controlado pelos componentes que estão ligados a eles. Embora os componentes integrados do Unity possam ser muito versáteis, logo será perceptível que é necessário ir além do que eles podem fornecer para implementar seus próprios recursos de jogo. Unity permite criar novos componentes usando Scripts. Eles permitem que sejam acionados eventos de jogo, modifiquem-se propriedades de componentes ao longo do tempo e respondam às entradas do usuário da maneira que se desejar (Unity Technologies, 2023b).

Um script faz sua conexão com o funcionamento interno do Unity implementando uma classe que deriva da classe interna chamada *MonoBehaviour*. A classe serve como uma espécie de modelo para criar um novo tipo de Componente que pode ser anexado aos Game Objects. A classe MonoBehaviour fornece a estrutura que permite anexar seu

script a um Game Object no editor, além de fornecer ganchos para eventos úteis, como Start e Update(Unity Technologies, 2023e).

```
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour
{
    // Use this for initialization
    void Start() {

    }

    // Update is called once per frame
    void Update() {

    }
}
```

Código 1 – Exemplo de um script base em Unity

Nesse sentido, a função *Update* é o local para colocar o código que tratará da atualização que ocorre a cada frame no Game Object. Isso inclui movimento, desencadear ações e responder à entrada do usuário, etc., basicamente qualquer coisa que precise ser tratada dinamicamente no decorrer do jogo. Para permitir que a função *Update* faça seu trabalho, muitas vezes é útil poder configurar variáveis, ler preferências e fazer conexões com outros GameObjects antes que qualquer ação do jogo ocorra. A função *Start* será chamada pelo Unity antes do início do jogo (ou seja, antes da função *Update* ser chamada pela primeira vez) e é um local ideal para fazer qualquer inicialização (Unity Technologies, 2023b).

Em conclusão, os scripts desempenham um papel central no desenvolvimento de jogos no Unity, capacitando os desenvolvedores a criar cada particularidade que seu jogo possui. A flexibilidade dos scripts, combinada com a modularidade proporcionada pelos componentes, forma a base estrutural da programação no Unity. Ao compreender como se usa os scripts, os desenvolvedores têm a capacidade de transformar ideias e funcionalidades do jogo em realidade.

2.2.3.6 Prefabs

O sistema de Prefabs do Unity permite criar, configurar e armazenar um Game Object completo com todos os seus componentes, valores de propriedade e Game Objects filhos como um asset reutilizável. O asset Prefab atua como um modelo a partir do qual pode-se criar novas instâncias do Prefab na cena (Unity Technologies, 2023f).

Quando for necessário reutilizar um Game Object em vários lugares da cena, como inimigos, armadilha, coletáveis, etc., vale convertê-lo para um Prefab. Isso é melhor do que simplesmente copiar e colar o Game Object, porque o sistema Prefab permite manter automaticamente todas as cópias sincronizadas. Quaisquer edições feitas em um Prefab

são refletidas automaticamente nas instâncias dele, permitindo que se faça facilmente amplas alterações em todo o projeto sem ter que fazer repetidamente a mesma edição em cada cópia do asset (Unity Technologies, 2023f). Além disso, é possível criar variações desses Prefabs que terão suas peculiaridades mantidas mesmo com alguma atualização do Prefab base, conforme sumarizado na Figura 6.

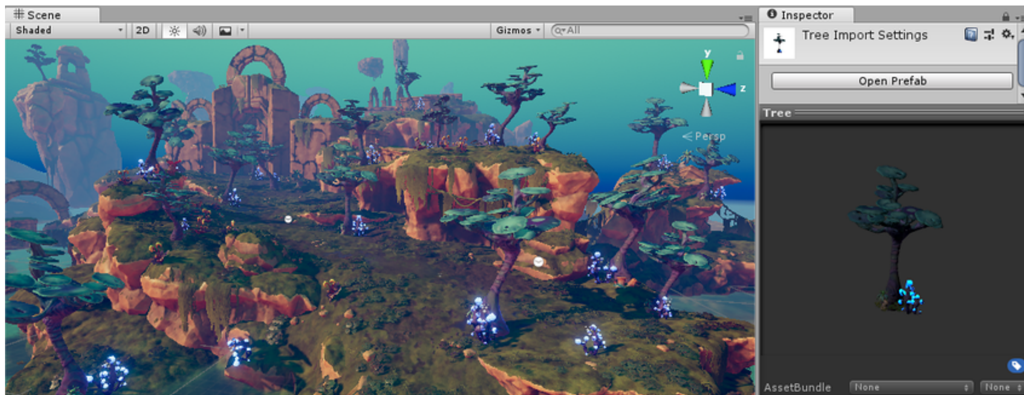

Figura 6 – Exemplo de Prefab de árvore

Portanto, os Prefabs no Unity desempenham um papel central na eficiência do desenvolvimento, permitindo a criação e reutilização rápida de objetos complexos. Essa funcionalidade não apenas simplifica o processo, mas também promove a manutenção consistente em todo o projeto.

2.3 Trabalhos Relacionados

Nesta seção serão avaliados alguns trabalhos anteriores que possuem a temática próxima a deste. O intuito é criar um contexto sólido para o presente trabalho, demonstrando onde e como ele se insere no cenário já existente de pesquisa na área, bem como esclarecendo em que aspecto ele pretende contribuir para o avanço desses conhecimentos.

2.3.1 Comparativo entre *engines* de jogos

Existem diversos estudos cujo tema central é desenvolver um comparativo estruturado entre algumas *engines* de jogos. Alguns como o (CAVALCANTE; PEREIRA, 2018) focam na análise entre duas *engines*, realizando um comparativo aplicado ao caso concreto de um design de jogo específico, onde se pretende determinar a escolha de uma *engine* mais adequada e que facilite o desenvolvimento do jogo em questão a ser implementado. Também fazem um comparativo aplicado à um caso concreto, mas focando principalmente em métricas considerando aspectos técnicos medíveis durante a fase de desenvolvimento do jogo (ALVES, 2020).

Já outros, focam em uma análise mais geral entre as principais *engines* do mercado a partir de uma abordagem empírica. Nesse sentido, trabalhos como (CHRISTOPOULOU;

XINOGALOS, 2017) não focam no jogo em questão a ser desenvolvido, mas utilizam o seu desenvolvimento para colher métricas gerais entre as principais *engines*, como recursos oferecidos, experiência do usuário, performance, etc.

Dessa forma, os primeiros trabalhos mencionados se aproximam da presente monografia no sentido de desenvolver um jogo e observar, durante este processo, aspectos específicos que sejam relevantes para a escolha de uma ou outra *engine*.

2.3.2 Unity para o desenvolvimento de jogos

Existem também alguns estudos, em destaque o (BERTO, 2017), que focam, assim como este, nas etapas de desenvolvimento de jogo analisando os ganhos e prejuízos com um olhar mais específico para uma *engine*, como o Unity. Nesse sentido, os referidos trabalhos possuem uma proposta e temática semelhantes, e neste trabalho haverá um adicional ao passo que serão discutidos em aspectos relacionados à jogos educacionais e a integração com uma plataforma existente que irá colher métricas e monitorar o aprendizado difere deste trabalho que pretende avaliar, através do uso dessa *engine*, a experiência e os recursos de forma genérica.

2.3.3 Jogos educativos no ensino de Química

Nesse contexto, vários trabalhos como (FILHO et al., 2019) focam na criação de um jogo educacional voltado ao ensino de Química, alguns como (PORTZ; EICHLER, 2013), (SIQUEIRA et al., 2023), (BARROS; SOUSA; VIANA, 2022), (ROMANO et al., 2017), focam de forma ainda mais específica, assim como no nesta monografia, na criação de um jogo para ensino da Tabela Periódica. Esses trabalhos exploram como os jogos podem ser eficazes na promoção da aprendizagem de conceitos químicos, estimulando o engajamento dos alunos e melhoria na qualidade de ensino.

Apesar do contexto em que esses trabalhos estão, ou seja, o desenvolvimento de um jogo educacional para o ensino de Química, ser exatamente o mesmo deste trabalho, eles possuem objetivos distintos. A contribuição desses trabalhos reside na avaliação da eficácia desse modelo de aprendizagem, bem como na identificação de melhores práticas, desafios a serem superados e no fornecimento de visões valiosas para o desenvolvimento de jogos educativos que sejam eficazes no processo de ensino de Química. Em contrapartida, aqui o foco é entender aspectos práticos relevantes ao uso de uma *engine* na criação e integração de jogos educativos com plataformas como a QuimiCot Games para que assim tenha-se maior consciência desses ferramentais a fim de auxiliar nesse processo que culminará no mesmo objetivo, oferecer jogos com o potencial de auxiliar no aprendizado de Química.

2.3.4 Plataformas para monitoramento de aprendizagem em jogos

Cientes da relevância dos jogos digitais como ferramentas para potencializar a educação, exitem também pesquisas com foco na outra ponta, em plataformas que agem como ferramentas para o apoio no ensino e avaliação do aprendizado, levantando-se a questão da necessidade de ter uma percepção mais ampla do que acontece dentro de um jogo a fim de aferir de maneira concreta a eficácia do ensino e mecanismos que podem auxiliar nele.

Diante da dificuldade em obter instrumentos que ofereçam apoio aos professores para avaliar o que um determinado aluno aprendeu, trabalhos como (VICTAL; MENEZES, 2016), (DAIREL et al., 2021), (VICTAL; MENEZES, 2015), (JUNIOR; MENEZES, 2015), entre outros, se propõe a apresentar um ambiente ou plataforma para coleta desses dados, bem como suporte à analise e aferição do aprendizado. Nesses casos, o foco é em prover mecanismos para que os professores consigam acompanhar as atividades dos estudantes com o intuito de monitorar a aprendizagem e identificar potenciais dificuldades (DAIREL et al., 2021). Esses mecanismos possibilitam realizar inferências ou adaptações no jogo, à depender da plataforma, no momento exato em que percebe que o aluno está em dificuldade, a fim de tornar o aprendizado mais efetivo.

Esse tema possui forte relação com o trabalho em questão, pois será realizada integração do jogo aqui desenvolvido com a plataforma QuimiCot Games (plataforma proposta inclusive no trabalho acima), que oferece esses recursos a fim de possibilitar e auxiliar as análises de aprendizado. Dessa forma, apesar de estarem correlacionados, este trabalho não foca em analisar e prover esses mecanismos, mas sim na parte prática de criação e posterior integração de um jogo com esse ambiente, entendendo as dificuldades, vantagens e desvantagens de utilizar uma engine nesse processo.

3 Desenvolvimento

3.1 Metodologia

Este trabalho faz parte de um projeto de pesquisa que visa criar um ecossistema de software de apoio ao ensino contextualizado da Tabela Periódica da disciplina de Química e monitoramento da aprendizagem por meio de jogos digitais. Esse ecossistema foi nomeado de QuimiCot Games[1]. A construção do sistema é centrada nos usuários que participam efetivamente do processo de construção (QuimiCot Games, 2023).

Nesse sentido, o jogo educativo criado neste trabalho tem o intuito de contribuir com a plataforma, oferecendo um novo jogo que visa trabalhar a Tabela Periódica de forma lúdica, possuindo toda a jogabilidade intrinsecamente ligada com esse tópico, inserindo e esperando que conceitos de química sejam usados para solucionar problemas e passar entre os níveis, além de possuir o padrão da plataforma que é aplicar questionários entre os níveis a fim de auxiliar a plataforma a medir a aprendizagem. Por isso, ambos possuem objetivos alinhados, que é a melhoria da qualidade na educação, envolvimento do aluno e ganho de aprendizado.

Para tornar possível a criação do jogo, o trabalho em questão foi desenvolvido de forma iterativa, contando com as seguintes etapas: Desenvolvimento, Integração e Análise. A Figura 7 mostra a sequência das etapas e as atividades realizadas em cada uma.

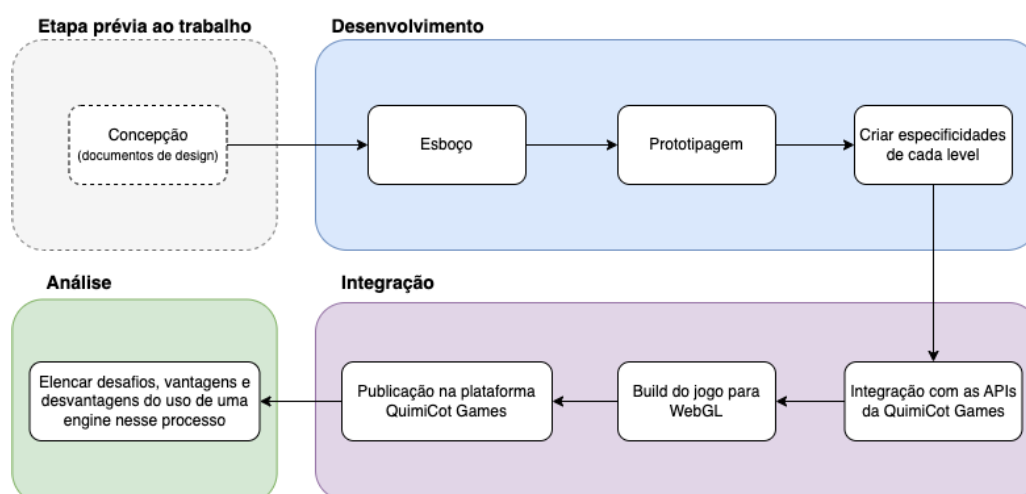

Figura 7 – Diagrama contendo as etapas iterativas para criação do jogo

A primeira etapa foi a concepção da temática do jogo, desenvolvida previamente ao trabalho pelos professores, alunos e administradores da plataforma QuimiCot Games.

[1] https://quimicotgames.com/

Cada fase do jogo foi pensada a fim de trazer abordagens específicas e foi descrito exatamente o que se espera de cada uma no documento de design.

Após estudada a concepção feita por eles, criou-se um esboço de cada um dos níveis do jogo. Geralmente o esboço é feito no formato de rascunho, seja papel ou digital, e possui o intuito de melhorar a visualização da jogabilidade e como as mecânicas propostas pelo documento de design serão transcritas no jogo.

Com o esboço pronto e devidamente validado pelos administradores, o passo seguinte foi iniciar o desenvolvimento e prototipagem das principais mecânicas que um jogo de plataforma contém, como a movimentação do jogador, mecanismos para facilitar a criação do mapa, tratar vida e morte do jogador, armadilhas e inimigos, coletáveis, arte, animação, áudio, gerenciamento de níveis, sistema de diálogo e o próprio questionário que a plataforma padroniza nos jogos.

Depois da prototipagem, iniciou-se um processo iterativo de construção do jogo, com a implementação de cada nível de jogo com entregas periódicas. Finalizada a implementação do jogo, inicia-se a fase de integração do jogo com plataforma mencionada, de forma a receber e enviar dados de uso do jogo. Finalmente, o passo seguinte foi *buildar* e publicar ou disponibilizar o jogo dentro da plataforma QuimiCot Games.

Com o jogo finalizado, serão apresentados e discutidos alguns resultados do trabalho e do impacto obtido e esperado. E, assim, será finalizado com uma conclusão, condensando os principais aspectos aqui abordados.

3.2 Proposta

De modo a auxiliar os métodos tradicionais de ensino e propiciar diferentes recursos para a aprendizagem, um jogo educativo foi concebido com o intuito de ser uma ferramenta a mais no ensino de Química. O referido jogo foi concebido de forma colaborativa por alguns professores de química de Uberlândia e os administradores da plataforma QuimiCot Games com o intuito de auxiliar especificamente no ensino da Tabela Periódica, que é um modelo que agrupa todos os elementos químicos conhecidos e suas propriedades, possuindo 118 elementos químicos, no qual cada quadrado especifica o nome do elemento químico, seu símbolo e seu número atômico (CISCATO; PEREIRA; CHEMELLO, 2015), conforme mostra a Figura 8.

A Tabela Periódica é uma ferramenta básica e fundamental para o conhecimento dos elementos químicos e suas propriedades, porém ela pode ser intimidadora de início e gerar uma certa barreira para os alunos, prejudicando o aprendizado.

Figura 8 – Tabela periódica. Fonte: (IUPAC, 2023).

3.3 Concepção do jogo

Nesta etapa, define-se o conceito e a visão geral do jogo. O jogo desenvolvido neste trabalho foi concebido como pertencente ao gênero plataforma *side-scrolling* bidimensional (2D). Em um jogo de plataforma, o jogador controla um avatar que se movimenta e salta entre plataformas e obstáculos, enfrentando inimigos e desafios, e coletando bônus (SMITH; CHA; WHITEHEAD, 2008). No caso de um jogo *side-scrolling* e 2D o avatar é visto de uma câmera lateral e, geralmente, começa do lado esquerdo do mapa e avança pelo caminho para o direito da tela cumprindo seus objetivos.

O presente jogo foi concebido para o auxílio no ensino de Química e possui seu eixo central no tema de metais da Tabela Periódica, possuindo temática especificamente elétrica. Os principais aspectos, mecânicas e objetivos foram concebidos de forma participativa por estudantes do Ensino Médio à partir de uma pesquisa realizada anteriormente (ARAúJO; NUNES; REZENDE, 2019), e a história de jogo foi idealizada por outros dois estudantes por meio de um projeto de iniciação científica do Ensino Médio. O referido jogo deve contar com questionários entre cada fase que possuem perguntas relacionadas ao tema em questão e aos quais os professores poderão aferir o aprendizado do jogador, no caso o estudante. O conceito do jogo foi pensado dividindo-o em três fases, conforme os requisitos abaixo.

3.3.1 Visão geral da primeira fase

O texto abaixo apresenta a visão geral da história, cenário e mecânica da primeira fase, produzido por estudantes do Ensino Médio no contexto de um projeto de iniciação científica júnior:

"Inicialmente, o jogador irá se deparar com um abismo a sua frente e uma ponte do outro lado, uma pequena caixa de texto aparecerá informando a mecânica que deve ser utilizada: para passar por este obstáculo, terá que fazer com que a ponte funcione, porém houve uma queda de energia e agora será necessário abrir a caixa geral para reativar a energia da ponte. Cuidado, se a chave correta não for ativada, o personagem será eletrocutado.A chave correta está marcada logo acima com a sigla de um Metal Alcalino, se necessário, uma tabela periódica poderá ser utilizada para seu auxilio. Após a leitura do texto, o jogador irá interagir com a caixa geral, tendo agora uma visão mais aproximada da caixa de forças. Como já informado, é necessário clicar na chave correta para que a ponte seja ativada, caso clique em uma das erradas, será eletrocutado e o personagem morrerá. Após atravessar a ponte, um *checkpoint* será setado, caso o jogador perca mais a frente, voltará para este ponto".

3.3.2 Visão geral da segunda fase

O texto abaixo apresenta a visão geral da história, cenário e mecânica da segunda fase, produzido por estudantes do Ensino Médio no contexto de um projeto de iniciação científica júnior:

"Na segunda fase deverá haver uma placa e nela estará escrito: está trovejando e chovendo muito, escolha um dos seguintes guarda-chuvas para lhe proteger pelo caminho, mas cuidado, escolha o guarda-chuva feito com o elemento ideal, pois senão acabará sendo eletrocutado por um raio. Dito isso, haverá três opções de escolha, sendo elas respectivamente Potássio, Platina e Nióbio, sendo o Potássio a escolha correta. Após o jogador escolher seu guarda-chuva, ele ficará com ele aberto sobre sua cabeça e terá liberdade para atravessar o campo sem sofrer dano. Em certo momento da travessia, um raio cairá acertando a ponta do guarda-chuva e, caso o jogador tiver escolhido o material correto (Potássio), ele não será morto. Todavia, caso tenha escolhido o errado, será eletrocutado e terá que voltar ao último *checkpoint*".

3.3.3 Visão geral da terceira e última fase

O texto abaixo apresenta a visão geral da história, cenário e mecânica da terceira fase, produzido por estudantes do Ensino Médio no contexto de um projeto de iniciação científica júnior:

"Na última fase, deverá haver uma porta fechada separando o jogador do término da fase. Logo uma caixa de texto aparecerá para o jogador, com o seguinte texto: a porta a frente funciona a base de eletricidade, porém ela não está funcionando. Para que consiga reativá-la e concluir a fase, terá que colocar a mão na massa e concertar a porta. Abra o painel logo ao lado e tente arrumar os fios, mas cuidado ao ligar fios de polos distintos (positivos e negativos), um curto circuito ocorrerá. Os fios positivos estão representados por uma sigla de um Metal de transição, enquanto os fios negativos estão representados por uma sigla de um Metal Alcalino Terroso. Para seu auxílio, uma tabela periódica poderá ser utilizada se necessário. Caso três curtos circuitos ocorram, o personagem será morto e retornará ao último *checkpoint*".

3.4 Esboço do jogo

Uma etapa importante é o esboço, no caso aqui um desenho feito de forma digital, com o intuito de visualizar, planejar e assegurar que os elementos-chave do jogo estão bem compreendidos antes de iniciar o desenvolvimento e protótipo do jogo propriamente dito.

3.4.1 Esboço da primeira fase

O foco aqui foi desenhar o design do primeiro nível, no qual o principal elemento é o abismo que o jogador irá se deparar, impossibilitando a passagem. Também será analisada a mecânica da ponte que irá possibilitar o avanço do jogador ao responder corretamente a pergunta.

Por fim, foi enumerada a mecânica da caixa de diálogo com a temática da pergunta e as possibilidades de resposta, aliado com o resultado delas afim de garantir que estivessem corretamente compreendidas, conforme mostrado na Figura 9.

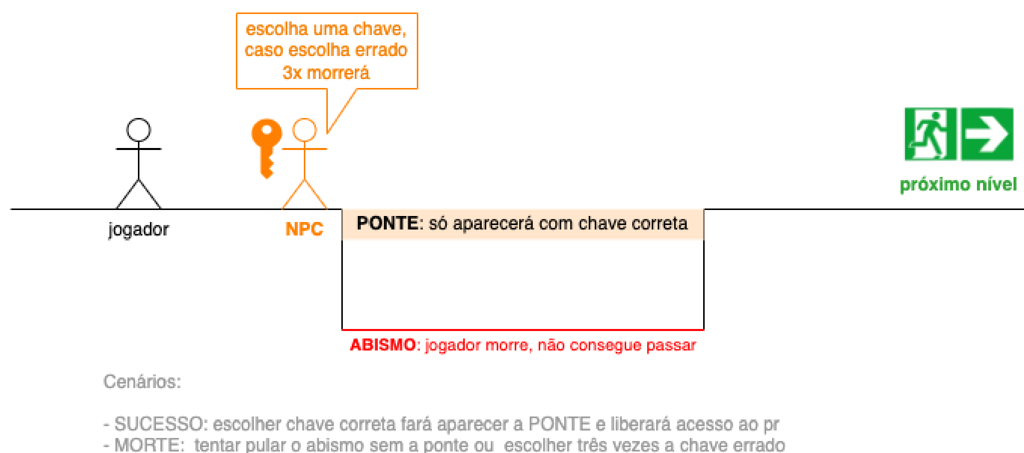

Figura 9 – Esboço do funcionamento geral do level 1

3.4.2 Esboço da segunda fase

Novamente, o foco aqui foi desenhar o design do referido nível, como mostra a Figura 10. O principal aspecto deste nível são os raios que irão eletrocutar o jogador caso tente prosseguir, impossibilitado seu progresso. Além da mecânica da entrega do guarda-chuva após responder um questionário, que serão feitos com materiais específicos que determinarão a possível proteção contra os raios.

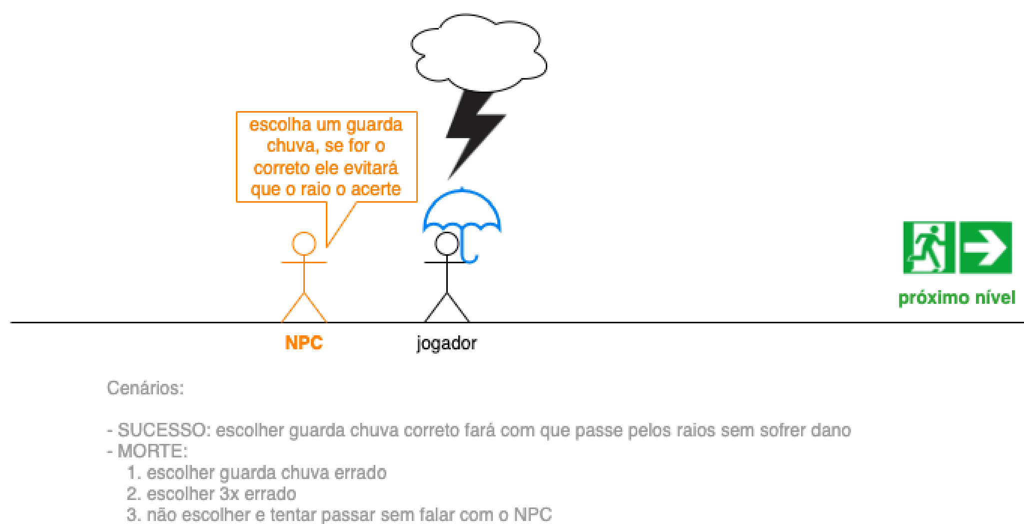

Figura 10 – Esboço do funcionamento geral do level 2

3.4.3 Esboço da terceira e última fase

Para a última fase o objetivo do esboço também era desenhar o design do nível, na qual o principal elemento é a porta fechada impedindo o progresso do jogador e os polos para serem conectados com o objetivo de retornar energia para a porta fazendo-a abrir. Aqui foi desenhado um esboço genérico para a mecânica da ligação dos fios elétricos, mostrado na Figura 11. Entretanto, foram pensadas diferentes possibilidades para a mecânica acima mencionada.

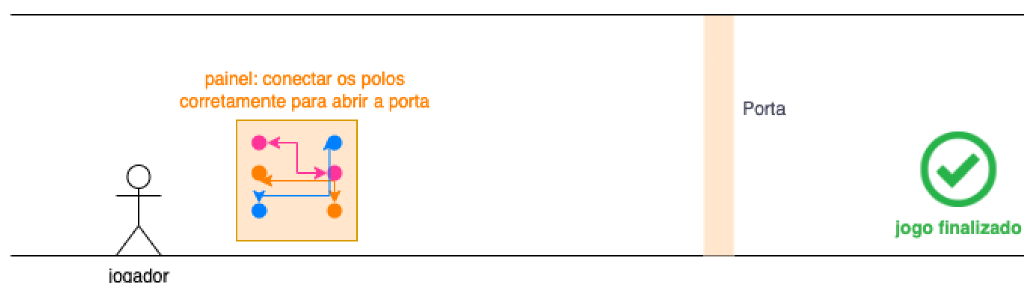

Figura 11 – Esboço do funcionamento geral do level 3

Como primeira ideia teve-se a utilização da própria tela de diálogo com o texto mostrando os polos, permitindo reorganizá-los para tetar fazer as ligações corretas.

A segunda ideia foi usar uma outra tela na qual apareceriam os diferentes polos e jogador iria manualmente clicar e arrastar o mouse para fazer as conexões.

Estas ideias foram descontinuadas em função de prejudicarem a jogabilidade de um jogo plataforma e consequente a dinâmica do jogo. Então, a terceira ideia foi unir a mecânica de ligação dos polos com a plataforma, na qual o jogador continuaria pulando obstáculos, movimentando e interagindo com elementos como interruptores que agiriam aqui como os polos. Ao interagir com um, aguarda-se a interação com o segundo e caso os elementos sejam corretos, a conexão será bem sucedida.

3.5 Prototipagem das principais mecânicas do jogo

Um protótipo é uma versão inicial de um produto a partir da qual serão desenvolvidas versões futuras. Engenheiros e desenvolvedores geralmente criam essas versões iniciais de um novo produto, serviço ou dispositivo antes de lançá-lo. Os protótipos não são o produto ou serviço final. Em vez disso, fornecem uma forma de testar uma ideia, validar o processo operacional e identificar formas de melhorar o item antes de lançá-lo ao público (Paulo Kirvan, 2023).

O Unity oferece uma ampla gama de recursos para agilizar o desenvolvimento de jogos. Isso inclui suporte eficiente para gráficos, modelos 2D e 3D, efeitos de partículas, editor de cenas, simulação de física, gerenciamento de áudio, animação, manipulação de entrada e saída, entre outros. Isso acelera o processo de desenvolvimento, reduzindo significativamente o tempo para criar as funcionalidades básicas e reduz também os gastos necessários para a criação desses protótipos.

3.5.1 Movimentação do jogador

A primeira etapa na criação de um jogo é trabalhar com a movimentação do jogador, com ela podemos dar início às funcionalidades básicas que um jogo possui.

Para construir a movimentação de um jogador em Unity é preciso inicialmente criar um objeto que irá representar o jogador. A princípio será necessário adicionar nesse objeto componentes que trabalhem com física que serão responsáveis por controlar a movimentação, além de componente de colisão, com o intuito de delimitar um espaço de corpo física para trabalhar com colisões entre jogador, chão, paredes, plataformas e demais elementos de um jogo, a fim de que essa movimentação esteja restrita a determinado padrão.

Após configurado o básico, é necessário capturar os *inputs*, ou seja, entradas do

jogador, que para esse jogo seriam os botões ou teclas pressionadas. Para isso o Unity recentemente disponibilizou um novo sistema para centralizar e controlar isso, chamado de *Player Input System* (Unity Technologies, 2023l). De acordo com a própria documentação, esse novo sistema surgiu pra simplificar flexibilizar o trabalho com as entradas do jogador, oferecendo suporte simples à diversos dispositivos de entrada como teclado, controles, touch, etc.

Para isso o *Player Input System* trabalha com a ideia de ações genéricas, chamadas de *Input Actions*, que poderão ser executadas pelas entradas do jogador. Estas ações podem variar de acordo com cada jogo. Por meio da Figura 12 é possível ver as ações configuradas para o referido jogo:

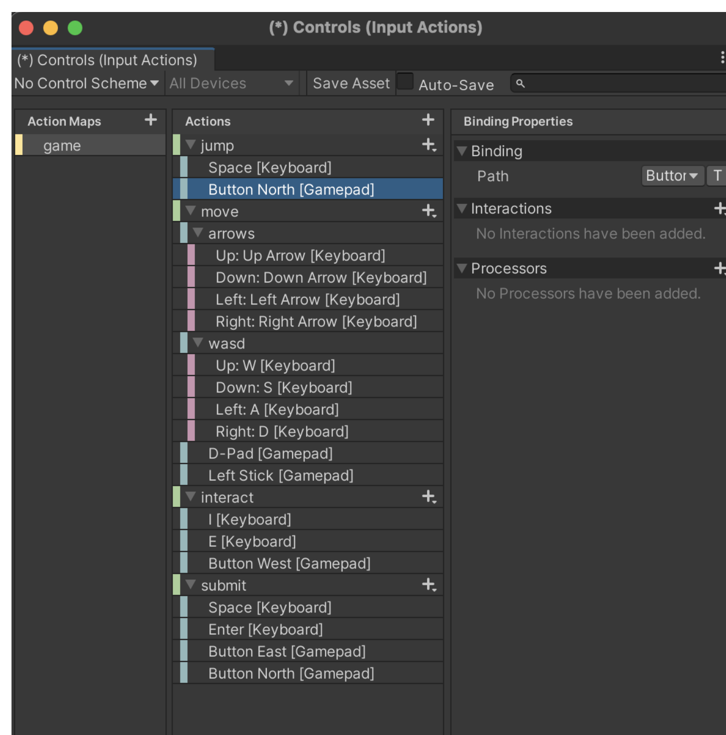

Figura 12 – *Input Actions*

As ações acima configuradas para o jogador são:

- *jump*: ação atrelada ao mecanismo de pular ou saltar.
- *move*: ação atrelada ao mecanismo de movimentação do jogador, seja para a esquerda, direita, cima e baixo (os dois últimos para movimentar em escadas por exemplo).
- *interact*: ação atrelada ao mecanismo de interação do jogador com objetos e diálogos ou qualquer outro mecanismo.
- *submit*: ação atrelada ao envio de uma resposta a um questionário ou à um diálogo por exemplo.

Todavia, adicionar esses componentes no objeto que representa o jogador e configurar as possíveis ações realizadas por meio de uma entrada do jogador não fazem com que tudo isso automaticamente funcione. É imperativo o uso de um *script*, ou seja, um código que controlará o funcionamento de todos esses componentes e do sistema de entradas.

Esse *script*, aqui nomeado de *PlayerController*, é um código na linguagem C# que a cada frame validará se houve alguma dessas ações realizadas pelas entradas do jogador, seja por meio de eventos ou de *callbacks*, que, ao serem acionados, irão manipular os componentes de física do objeto fazendo com que o jogador se mova.

O código abaixo é um exemplo de uma implementação da ação *move*:

```
1  public void MovePressed(InputAction.CallbackContext context)
2  {
3      moveDirection = context.ReadValue<Vector2>();
4      moveVector = new Vector2(
5          moveDirection.x * runSpeed,
6          body.velocity.y
7      );
8
9      body.velocity = moveVector;
10 }
```

Código 2 – Função que implementa a ação de mover

Com isso, percebe-se que a implementação das funções básicas de movimentação do jogador são simples e flexíveis, podendo ser rapidamente desenvolvidas por meio dos recursos oferecidos pela plataforma Unity, sendo necessário apenas uma pequena curva de aprendizado para entender os conceitos envolvidos.

3.5.2 Configuração dos *tilemaps* para construção dos mapas de cada níveis

O sistema *Tilemap* do Unity foi desenvolvido para facilitar a criação e a iteração de design de níveis. É uma ferramenta não obrigatória para isso, porém é um facilitador. Torna o processo de prototipagem e iteração mais rápido e acessível para desenvolvedores, designers e artistas, podendo rapidamente a construir e editar mapas para fases em jogos 2D.

Segundo a Unity (Unity Technologies, 2023k): "Um Tilemap consiste em uma sobreposição de grades e vários outros componentes trabalhando juntos. Todo o sistema permite "pintar"níveis usando a ferramentas de pincel para definir regras de como os *tiles* se comportam. Com o *Tilemaps*, pode-se criar plataformas com bordas/limites dinâmicos, blocos com animação, posicionamentos aleatórios de blocos e muito mais. O sistema *Tilemap* da Unity é perfeito para projetos 2D que contenham níveis de jogabilidade, pois permite ao usuário prototipar níveis que podem ser testados imediatamente no mecanismo do jogo."

Dito isso, percebe-se que o principal e maior motivo para usar a ferramenta de *Tilemaps* é a facilidade de criação e edição de mapas para níveis. Porém, esse não é o único motivo. Existem uma série de outros motivadores, desde performance à experiência do usuário. Abaixo estão listados alguns (Unity Technologies, 2023k):

- Eficiência de memória: Usar um sistema baseado em *Tilemaps* permite que se represente grandes áreas do jogo com um conjunto relativamente pequeno de elementos gráficos. Em vez de ter uma representação individual para cada objeto no nível, pode-se usar *tiles* para preencher áreas maiores, economizando memória e melhorando o desempenho.

- Performance otimizada: O uso de *Tilemaps* em Unity permite que o mecanismo do jogo otimize a renderização e o processamento das células de forma mais eficiente. O mecanismo pode agrupar elementos visuais semelhantes em lotes, reduzindo o número de chamadas de renderização e melhorando o desempenho geral.

- Colisão simplificada: O uso de *tilemaps* simplifica a configuração de colisões em um jogo 2D. atribui-se colisores às células do *tilemap*, permitindo que o personagem do jogador e outros objetos interajam com o ambiente de forma eficiente e precisa.

- Alinhamento e encaixe automático: Com os *Tilemaps*, os tiles se encaixam automaticamente em uma grade, garantindo que tudo fique alinhado corretamente. Isso facilita a criação de níveis consistentes e evita problemas de posicionamento impreciso dos objetos.

- Suporte a diferentes resoluções: Os *Tilemaps* em Unity facilitam o dimensionamento dos elementos do jogo para diferentes resoluções de tela. Os tiles podem ser ajustados automaticamente para se adaptarem a diferentes proporções e tamanhos de tela, garantindo uma experiência consistente para os jogadores.

Diante de todas essas vantagens, optou-se por criar alguns *Tilemaps* para o jogo em questão, conforme Figura 13.

- *Platform Tilemap*: responsável por criar grids de plataforma, nas quais o usuário pode colidir, servindo como chão, parede, plataforma, etc.

- *Background Tilemap*: responsável por criar grids de fundo, não colidíveis, os quais aparecerão atrás dos demais para transmitir a atmosfera geral do mapa.

- *Water Tilemap*: responsável por criar grids de água, não colidíveis, os quais aparecerão no mesmo nível das plataformas, apenas como elemento visual.

- *Hazard Tilemap*: responsável por criar grids de armadilhas, elementos interagíveis, que causam dano ao personagem.

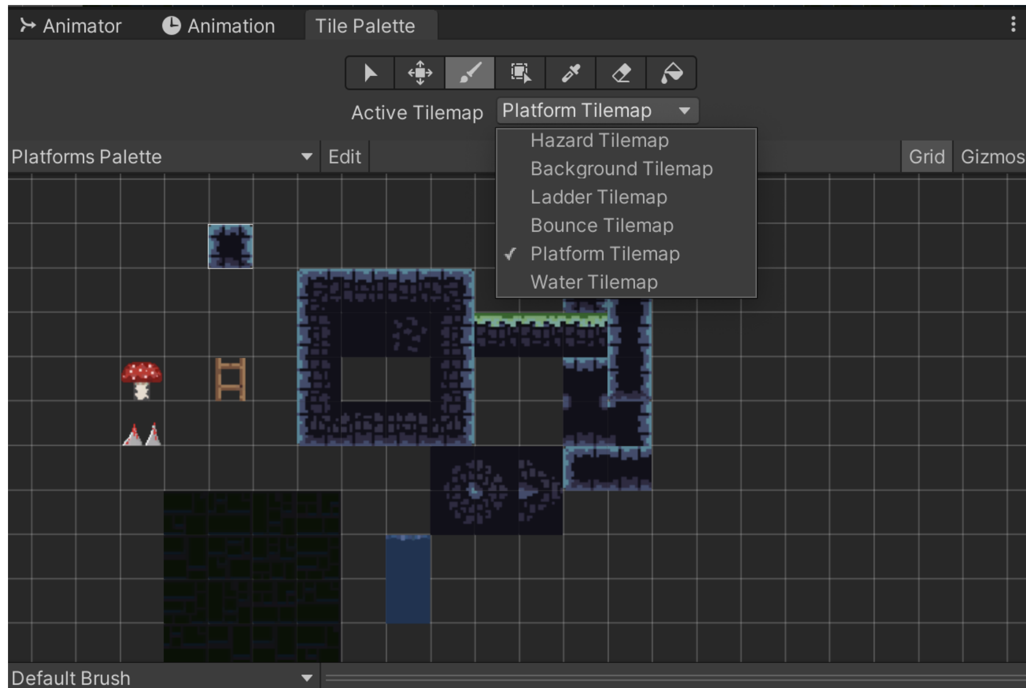

Figura 13 – *Tilemaps configurados para este jogo*

- *Ladder Tilemap*: responsável por criar grids de escadas, elementos interagíveis, os quais auxiliam a movimentação vertical do jogador.

- *Bounce Tilemap*: responsável por criar grids de objetos que interagíveis, que permitem ao jogador pular grandes distâncias.

A criação é extremamente simples e, claro, é possível criar vários outros de acordo com a necessidade, como Tilemaps para inimigos, moedas e etc. Sua utilização trouxe enormes vantagens para a prototipagem das fases, como acelerar o processo, diminuindo tempo de configuração e ganho de praticidade. Esses ganhos são proporcionalmente maiores quanto mais densidade do mesmo elemento precisar existir em um mapa, além da praticidade e facilidade de os adicionar e remover.

3.5.3 Conceito de vidas e mortes para o jogador

Em um jogo o conceito de vida e morte está relacionado à mecânica de jogabilidade. A vida é, em geral, um número que representa a quantidade pontos de vida do personagem controlado pelo jogador, por isso é conhecida também como *Health Points* (HP). Quando sofre dano, os pontos de vida do jogador são reduzidos, e caso esses pontos cheguem a zero, ocorre a morte do personagem.

Esta mecânica contribui adicionando vários elementos que tornam o jogo mais interessante. O primeiro seria o desafio, uma vez que a morte obriga o jogador a recomeçar de um ponto anterior ao que estavam, resultando na perda de progresso. Isso compele o

jogador a analisar a situação, adotar estratégias e aprimorar sua habilidades para que consiga sobreviver e continuar avançando no jogo.

Ademais, a mecânica traz um ambiente de tensão e perigo no jogo. Qualquer erro ou falta de perícia pode resultar na perda de pontos de vida e consequente morte do personagem, o que aumenta e imersão e emoção ao jogar. E, por fim, a sobrevivência, especialmente após situações desafiadoras, também traz a sensação de satisfação e recompensa, tornando a experiência do jogo mais divertida e gratificante.

É importante salientar que existem outra mecânicas que podem auxiliar nesses elementos de tensão e perigo, mas o conceito de vida e morte é a principal e mais comumente utilizado para trazê-los. Sem isso, o jogador poderia passar pelos níveis de qualquer forma, possivelmente tornando a experiência maçante e sem emoção.

Para a implementação dessa mecânica no jogo do referido trabalho, será criado um *script* que controlará a vida do jogador. Nele haverá uma variável que representará os pontos de vida:

```
1  public class PlayerHealthController : MonoBehaviour
2  {
3      [SerializeField] int healthPoints = 3;
4  }
```

Código 3 – Base para o script que controla a HP do jogador

Vale ressaltar que o atributo *SerializeField* é uma funcionalidade do Unity que permite configurar facilmente um campo pela interface gráfica, além de ser visualmente fácil acompanhar no modo *debug*, conforme exemplificado na Figura 14. Ao adicionar o *script* ao objeto do jogador, um campo parametrizável é aberto, que irá substituir o padrão que está no *script* (3).

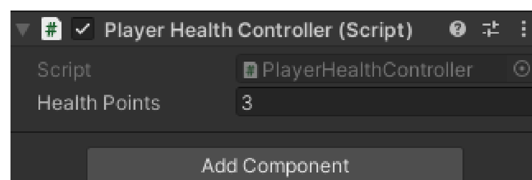

Figura 14 – *Exemplo de Serialize Field*

O conceito de dano pode ser feito de várias maneiras, aqui optou-se por utilizar uma solução orientada a eventos do Unity. Foi criado um evento chamado *OnDamage* que recebe um valor inteiro, representando o dano recebido. Depois, por meio do método *Start* da própria Unity, que é chamado na abertura do jogo, antes da primeira atualização de *frame* do jogo, será incluída uma função que tornará o objeto do jogador um ouvinte deste evento. Isso significa que sempre que o evento *OnDamage* for disparado por alguém, o*script* irá receber e executar uma ação representada por essa função:

```
public static UnityEvent<int> OnDamage { get; private set; }

// Start is called before the first frame update by Unity engine
void Start()
{
    OnDamage.AddListener(decreaseHP);
}
```

Código 4 – Script que torna-o um ouvinte do evento de dano

Após isso, torna-se necessária a implementação da função *decreaseHP* que irá processar a redução dos pontos de vida do jogador.

```
void decreaseHP(int damageReceived)
{
    healthPoints -= damageReceived;
    if (healthPoints <= 0) { die(); }
}
```

Código 5 – Função que reduz a HP do jogador

E por fim, no caso de não existir mais pontos de vida, a função *die* deverá ser chamada para processar a morte do jogador. A morte do jogador é obtida chamando duas outras funções, uma que irá parar os movimentos do jogador (*StopMovement*) e outra que fará o jogador voltar ao início da fase (ResetGameLevel). Para isso, será utilizado o método *FindObjectOfType* da *engine* Unity que busca na cena por outro *script* ue possuem essas funções implementadas, que não serão para não sobrecarregar o referido trabalho.

```
void die(){
    FindObjectOfType<PlayerMovementController>().StopMovement();
    FindObjectOfType<GameSessionController>().ResetGameLevel();
}
```

Código 6 – Função que realiza a ação de morte do jogador

Agora, com tudo configurado, resta apenas criar os componentes que irão disparar os eventos *OnDamage* para completar a referida funcionalidade.

3.5.4 Armadilhas e inimigos

Em um jogo, as armadilhas e inimigos são elemento importantes da jogabilidade, projetados, em geral, para causar dano ao personagem do jogador, dificultando e trazendo desafio ao seu progresso.

As armadilhas são dispositivos ou obstáculos colocados no ambiente do jogo podendo ser elementos estáticos, como espinhos, buracos ou quedas, ou até armadilhas ativadas por movimento, pressão ou outros mecanismos.

Para isso é necessário um novo objeto *Spike* que representará um espinho que, ao entrar em contato com o personagem do jogador, o causará dano total, levando-o à morte imediata. Todavia, para causar dano é necessário outro *script* nomeado de *DamageDealer*

que irá publicar o evento de *OnDamage* criado anteriormente, passando um valor inteiro que representa o dano causado. Este script será adicionado à todos os componentes capazes de causar dano ao jogador.

```
[RequireComponent(typeof(Collider2D))]
public class DamageDealer : MonoBehaviour
{
    [SerializeField] int damage = 1;

    void OnTriggerEnter2D(Collider2D other)
    {
        PlayerHealthController.OnDamage.Invoke(damage);
    }
}
```

Código 7 – Script causador de danos

Agora, será preciso adicionar um componente de colisão à armadilha e configurá-lo como *trigger*. Com isso, ao atravessar os limites da colisão (representado pela linha verde na figura abaixo), um evento (*OnTriggerEnter2D*) criado pela engine da Unity será disparado e utilizando-o, será possível publicar o evento de *OnDamage*. Por fim, basta sobrescrever o valor padrão de dano de um para três, que representa o total de ponto de vidas do nosso jogador, a fim de causar sua morte instantânea, conforme sumarizado na Figura 15.

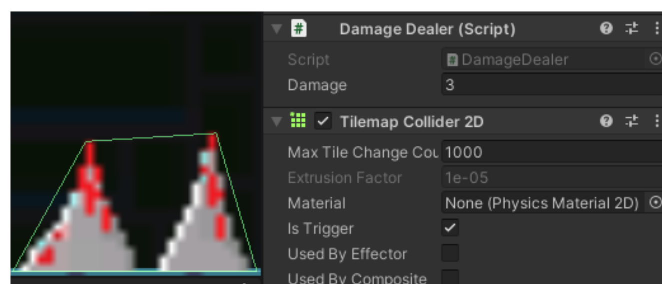

Figura 15 – *Criação da armadilha*

Já os inimigos são personagens controlados pelo jogo que, apesar de possuírem o mesmo princípio de causar dano ao personagem, não necessariamente o levarão á morte imediata. Cada inimigo possui suas peculiaridades e podem variar em termos de comportamento, habilidades, força e estratégias de ataque.

Para este jogo inicialmente será criado apenas um inimigo simples que irá andar de um lado por outro da plataforma e ao entrar em contato com o jogador o causará apenas um ponto de dano.

O processo é o mesmo do anterior, criar um objeto, que será chamado de *EnemySlime*, adicionar um elemento de colisão *trigger* para saber quando entra em contato com o jogador, bem como o já criado script *DamageDealer*, conforme Figura 16.

Todavia, diferente das armadilhas (*Spike*), o inimigo deverá se movimentar sozinho. Para isso, torna-se necessário criar outro script que será chamado de *AutoSideToSideMover*

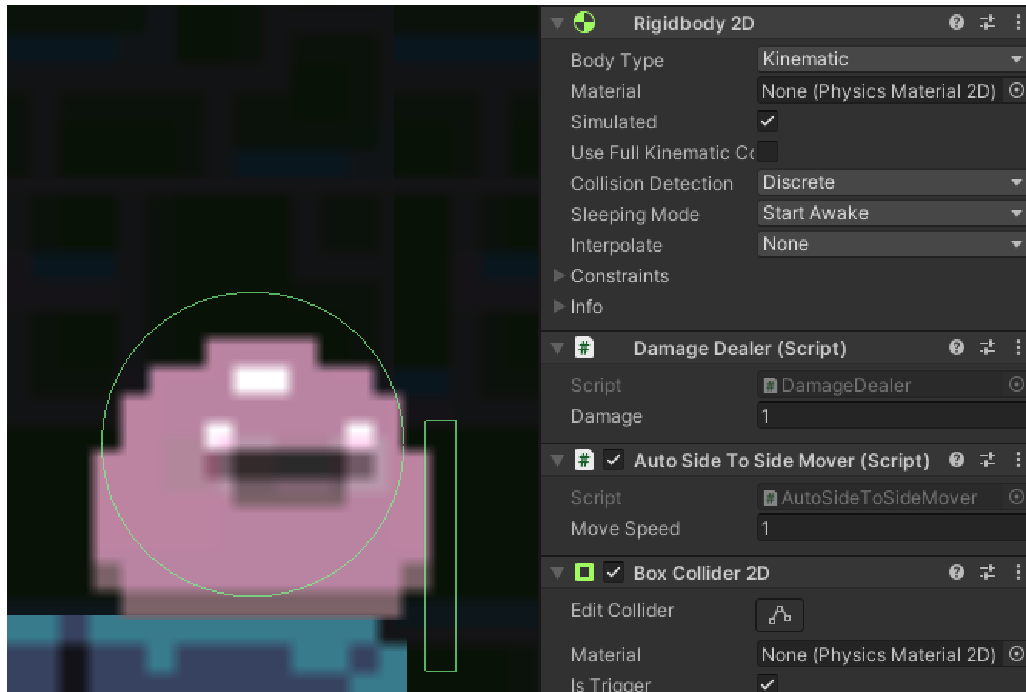

Figura 16 – *Criação do inimigo*

e que irá adicionar a funcionalidade de movimentação lateral automática.

```
public class AutoSideToSideMover : MonoBehaviour
{
    [SerializeField] float moveSpeed = 1f;

    // Update is called once per frame
    void Update()
    {
        GetComponent<Rigidbody2D>().velocity = new Vector2(moveSpeed, 0f);
    }
}
```

Código 8 – Script de movimentação lateral automática

Neste script, inicialmente haverá uma variável que controlará a velocidade de movimento do objeto, no caso o inimigo. Todavia, para que exista movimento é necessário adicionar outro elemento, Rigidbody2D, que permitirá controlar todo o motor de física 2D da Unity.

Para isso, será utilizada a função *Update* do próprio Unity que é chamada uma vez a cada frame do jogo, será possível configurar a velocidade lateral do objeto para a velocidade que foi configurada na variável *moveSpeed*. Com isso, o objeto no início do jogo irá começar a se movimentar. Entretanto, essa movimentação ainda está configurada para apenas um lado e, caso não seja melhorado, o objeto andará infinitamente para o referido lado.

Dessa forma, resta a adição de mais uma funcionalidade, a de trocar de lados automaticamente. Isto pode ser obtido de diversas formas, neste caso será utilizado mais um componente de colisão *trigger*. Este componente estará em contato com o chão prati-

camente o tempo todo que o objeto estiver se movimentando. Ocorre que, ao ao sair da colisão, seja ao se deparar com um penhasco ou uma parede, o objeto irá trocar a direção do movimento e, para isso o seguinte código deve ser adicionado ao de cima:

```
void OnTriggerExit2D(Collider2D other)
{
    moveSpeed = -moveSpeed;
}
```

Código 9 – Implementação da inversão de direção do movimento

3.5.5 Coletáveis

Coletáveis em jogos são itens ou objetos especiais que os jogadores podem encontrar e adquirir durante a progressão do jogo. Eles são projetados para incentivar a exploração, oferecer recompensas extras e aprofundar a experiência do jogador. Os coletáveis podem variar em forma, função e propósito, dependendo do design do jogo.

Ao coletar esses itens associa-se diretamente à um sentimento de recompensa e para aumentar ainda mais isso e incentivar a exploração, esses coletáveis podem estar escondidos em áreas secretas, localizados em locais de difícil acesso ou requerer que o jogador resolva problemas para serem obtidos.

A diversidade de coletáveis dependerá do gênero, estilo de jogo e escolhas de design específicas. Alguns exemplos de coletáveis comumente encontrados em jogos são:

- Moedas ou gemas: são itens pequenos e brilhantes espalhados pelo mundo do jogo. Os jogadores podem coletá-los para acumular pontos ou ganhar dinheiro virtual para ser utilizado na compra de itens e melhorias.
- Corações ou itens de vida: são coletáveis que restauram ou adicionam pontos de vida ao jogador. Eles são frequentemente encontrados em locais estratégicos ou usados como recompensa após a conclusão de desafios ou tarefas.
- Chaves: são objetos necessários para abrir portas trancadas ou desbloquear áreas secretas. Em geral, nesses casos os jogadores precisam encontrar e coletar as chaves para progredir no jogo.
- Estrelas: em geral são coletáveis raros, dificilmente encontrados, ou seja, com poucas reincidências nas fases. Sua coleta comumente desbloqueia áreas adicionais, níveis secretos ou finais alternativos.
- *Power-ups*: são itens temporários que concedem ao jogador habilidades especiais por um período limitado de tempo. Isso pode incluir aumento de velocidade, invencibilidade, maior força de ataque, entre outros.

- Itens equipáveis: são itens coletáveis como armas, roupas, escudos, armaduras, etc. que geralmente influenciam no ataque, defesa e visuais do personagem. Estes itens contribuem com a progressão no jogo, possibilitando enfrentar desafios maiores e dão um senso de customização ao personagem no jogo.
- Pergaminhos ou relíquias: são coletáveis que fornecem informações adicionais sobre a história do jogo ou oferecem informações sobre o mundo do jogo. Coletá-los pode aprofundar a imersão do jogador na narrativa.
- Itens colecionáveis: geralmente se refere à itens gerais como cartas, selos, figurinhas ou qualquer outro tipo de item que faça parte de uma coleção maior dentro do jogo. A conclusão da coleção pode desbloquear recompensas especiais ou alcançar objetivos secundários.

Diante de todas essas possibilidades, para um protótipo inicialmente será desenvolvido apenas o coletável mais comumente usado, as moedas. Entretanto, o script que será criado possibilitará coletar qualquer tipo de item, possibilitando facilmente a futura adição de outros coletáveis.

Como já visto, é preciso adicionar um componente de colisão à ao objeto que será coletável e configurá-lo como *trigger*. Com isso, ao atravessar os limites da colisão, o evento (*OnTriggerEnter2D*) criado pela engine da Unity será disparado e possibilitará a execução de ações pertinentes a coleta de itens.

Nesse sentido, é necessário identificar o jogador como único capaz de coletar, senão os inimigos, outros coletáveis ou qualquer outro objeto poderia interagir com ele. Para isso será adicionada uma *tag* "Player"para identificar o objeto como jogador.

Depois, será necessária uma variável de controle *wasCollected* para evitar que seja coletado mais de uma vez até sua coleta ser processada.

Ao adicionar o script acima em um componente, por exemplo que representa a moeda, deve-se criar um evento específico para a coleta de moedas para que possa ser ouvido por algum outro script e tratado conforme melhor atender ao caso.

Aqui, ao tentar serializar, ou seja, parametrizar via scrip um evento Unity mostrou-se impossível uma vez que a própria engine nativamente não permite isso. Dessa forma, torna-se necessário criar um evento genérico que herda da classe UnityEvent e adicionar a opção de serialização através do atributo *System.Serializable*, conforme:

Agora é possível criar eventos de coleta e passá-lo via parâmetro do método, conforme exemplo do evento criado para coletar moedas:

Feito isso, o script precisa apenas enviar o evento passado como parâmetro e depois destruir o objeto para que saia de cena.

```
[RequireComponent(typeof(Collider2D))]
public class Collectable : MonoBehaviour
{
    [SerializeField] PickupEvent pickupEvent;
    [SerializeField] int points = 1;
    bool wasCollected = false;

    void OnTriggerEnter2D(Collider2D other)
    {
        if(!whoTriggeredIsThePlayer(other) || wasCollected) { return; }
        markAsCollected();
        pickupEvent.Invoke(points);
        Destroy(gameObject);
    }

    bool whoTriggeredIsThePlayer(Collider2D other)
    {
        return other.tag == "Player";
    }

    void markAsCollected()
    {
        wasCollected = true;
        gameObject.SetActive(false);
    }
}
```

Código 10 – Script de coletáveis

```
[System.Serializable]
public class PickupEvent : UnityEvent<int>{};
```

Código 11 – Evento de coleta

```
public class CoinPickupEvent : PickupEvent{};
```

Código 12 – Evento de coleta de moedas

Também é valido criar um parâmetro, por exemplo *points* que expressa a quantidade de pontos que essa coleta representa. O padrão de pontos é um, entretanto, com isso será possível criar variações de itens, como uma moeda vermelha que vale mais pontos que a normal, sendo preciso apenas especificar os pontos q elá concederá conforme 17:

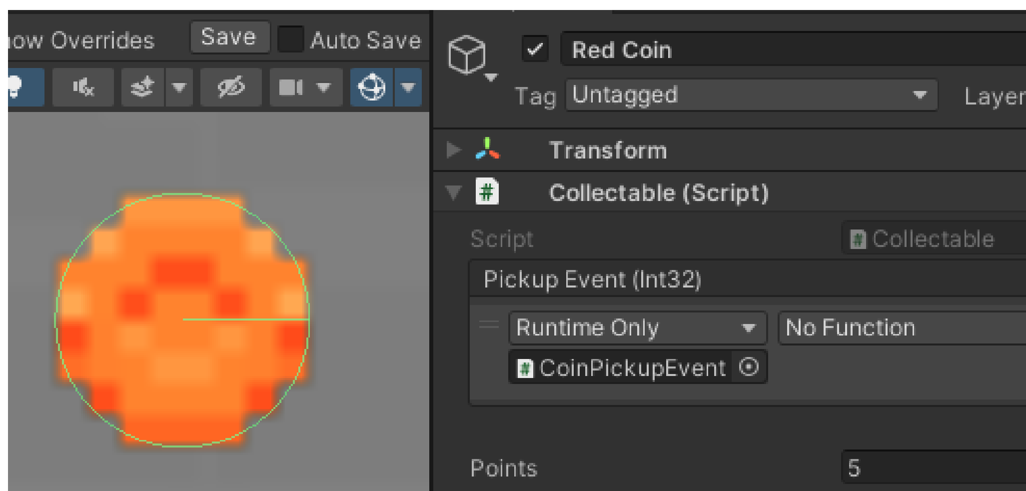

Figura 17 – *Criação da moeda vermelha*

Por fim, resta que o componente *GUIController*, responsável pela apresentação das informações visuais na interface gráfica apresente a quantidade de itens coletados na tela.

Para isso inicialmente será adicionado um campo de texto na interface gráfica:

```
[SerializeField] TextMeshProUGUI currentCoins;
```

Código 13 – Campo de quantidade atual de moedas na UI

Depois é preciso começar a ouvir os eventos disparados no momento em que uma moeda é coletada (*CoinPickupEvent*) e configurar o valor inicial para esse campo:

```
void Start()
{
    currentCoins.text = "0";
    CoinPickupEvent.AddListener(increaseCoinScore);
}
```

Código 14 – Implementação para ouvir o evento de coleta de moedas

Finalmente, resta apenas implementar a função que será chamada cada vez que um evento de moeda coletada for recebido:

```
public void increaseCoinScore(int coinsToBeAdded)
{
    int total = int.Parse(currentCoins.text) + coinsToBeAdded;
    currentCoins.text = total.ToString();
}
```

Código 15 – Função que aumenta o score de moedas

Agora já está tudo configurado para coletar itens como as moedas e mostrá-los na interface gráfica da tela do jogador, além de ser facilmente possível implementar outros coletáveis usando o script aqui desenvolvido.

3.5.6 Arte

A arte é um elemento crucial para criar uma experiência visualmente atraente em um jogo. Ela abrange uma variedade de aspectos visuais que incluem personagens, cenários, ambientes, interfaces e efeitos visuais. A arte é uma das primeiras coisas que os jogadores notam e uma arte visualmente atraente pode despertar o interesse dos jogadores e incentivá-los a começar a jogar, bem como uma arte não tão boa, pode afastar os jogadores, mesmo que a jogabilidade seja boa.

Existem diferentes tipos de artes que podem ser usados para criar os visuais e a atmosfera do jogo. Cada estilo desses de arte ajudam a criar a identidade visual do jogo e podem variar desde os mais simples e minimalistas até os mais detalhados e elaborados.

Um dos estilos mais populares para jogos 2D é o pixel art, no qual são utilizados pixels individuais que são pequenas unidades quadradas que representam uma única cor. Compondo vários pixels dentro de uma grade com tamanho específico é possível criar arte para personagens, fundo, etc. Esse estilo de arte ganhou popularidade nos primórdios dos videogames, quando a tecnologia limitada dos consoles e computadores exigia o uso de gráficos de baixa resolução. Hoje, com o avanço da tecnologia, o poder computacional aumentou, mas esse estilo de arte ainda perdura. Alguns dos principais motivos são a nostalgia, remetendo aos jogos clássicos, a redução de custos e tempo de desenvolvimento diante de sua simplicidade.

Nesse sentido, diante dos motivos supracitados para como simplicidade e possuir ainda um grande apelo visual, o estilo escolhido para o jogo deste trabalho será o pixel art. Existem várias formas de criar pixel art, a primeira é fazê-la de forma manual, desenhando pixel a pixel. Apesar de ser a melhor para criar uma identidade visual única e permitir a criação de artes específicas para o jogo, é também a mais trabalhosa.

Com o intuito de auxiliar nesse aspecto, a Unity oferece, integrado em sua plataforma, a possibilidade de obter pacotes de artes, tanto pagos quanto gratuitos e facilmente integrá-los ao jogo. Para a função de prototipagem e consequentemente para acelerar o desenvolvimento deste trabalho escolheu-se por usar um pacote de artes oferecidos na loja.

Após obter o pacote, é preciso preparar todas as artes contidas nele para estarem aptas a serem usadas de forma consistente no jogo, colocando-as no mesmo tamanho, definido a proporção de pixels por unidade, formato da imagem etc.

Com tudo configurado, agora é necessário criar os *sprites*, ou seja, os elementos gráficos 2D que representam imagens, personagens, objetos, fundos e etc. Eles são usados para criar a arte e a animação de objetos em um ambiente 2D e são a base para a criação de cenas e jogabilidade em jogos desse estilo. Ao colocar as imagens na pasta *Sprites*, o Unity entende se tratarem disso e automaticamente configura o como Sprite 2D, como mostra a Figura 18.

Como o grid do Unity exige um tamanho para cada bloco, é padronizado o tamanho de 32x32 e, para isso, é preciso definir o campo *Pixels Per Unity* como 32 e, como se trata de uma imagem simples, não é preciso fazer mais nada.

Existem casos em uma imagem vêm por meio dos artistas agrupando múltiplas imagens nela. Um exemplo de sprite múltiplo no pacote obtido para este jogo é o do jogador, contendo todos os frames de sua animação.

Dessa forma, é possível quebrá-las em várias ao selecionar o *Sprite Mode* como *multiple* e utilizar o Sprite Editor do Unity para definir um método de corte, que converterá essa imagem grande em várias pequenas imagens que futuramente serão usadas para criar

as animações, conforme 18.

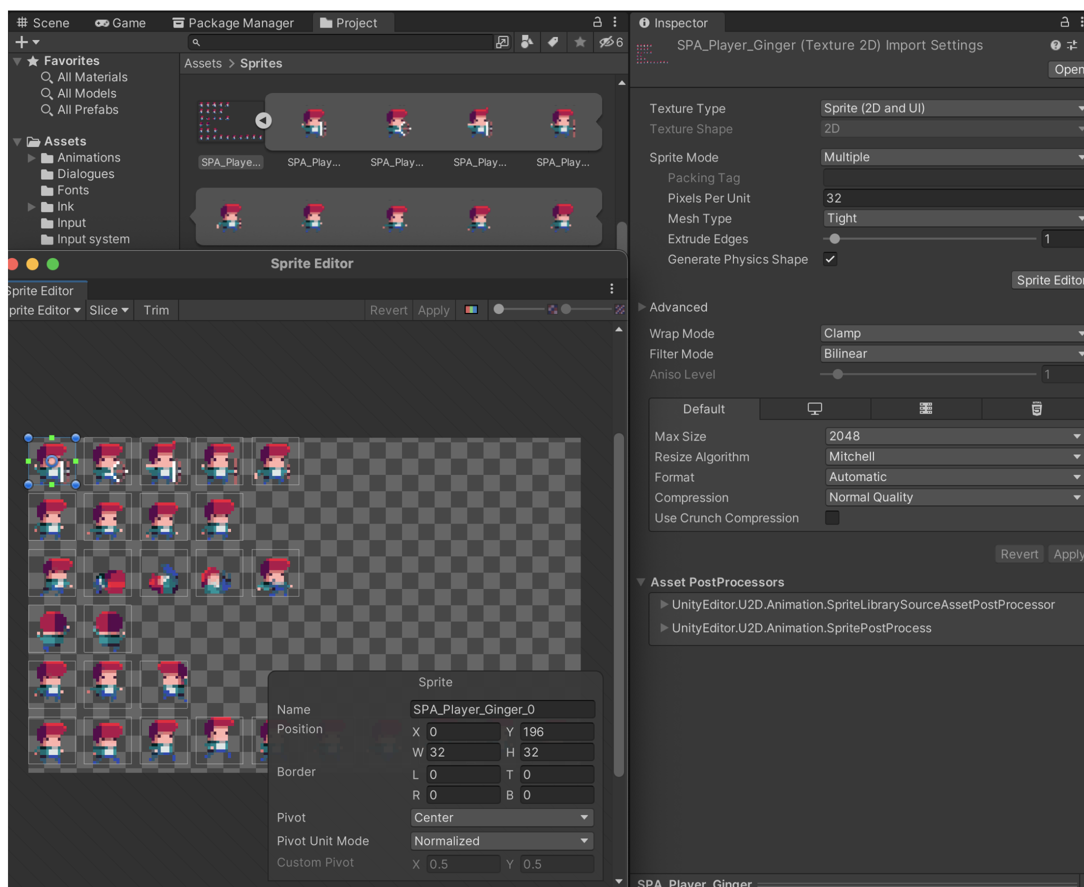

Figura 18 – *Exemplo de sprite múltipla*

Com os sprites configurados, basta arrastar a imagem para um objeto do jogo e um componente será adicionado a ele, possibilitando que a arte seja renderizada no jogo, conforme 19.

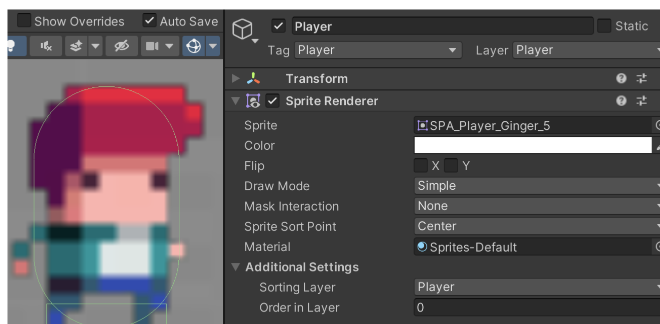

Figura 19 – *Exemplo de sprite utilizado no objeto do jogador*

Por fim, vale a pena voltar para o *Tilemap* de criação de plataformas e criar regras pra esse *Tilemap* configurar cada imagem do sprite automaticamente. As regras em questão dependem de cada arte disponível e de cada jogo a ser desenvolvido. Para este jogo, foram criadas as seguintes regras 20 para configuração automática de cada imagem.

Figura 20 – *Tilemap para criação de plataformas*

Dessa forma, a artes das plataformas tornam-se dinamicamente e automaticamente adaptáveis umas às outras, de acordo com as regras acima definidas, o que acelera o desenvolvimento e prototipagem das fases que, não fosse isso, teriam de ser feitas à mão.

3.5.7 Animação

A animação é uma parte essencial da criação de jogos em Unity, permitindo dar vida aos personagens, objetos e elementos visuais do jogo. O Unity oferece um sistema de animação intuitivo e flexível que facilita a criação de animações complexas.

Para criar uma animação basta selecionar as imagens que irão compor cada frame, clicar com o botão direito e selecionar a opção *Create Animation* conforme exemplo de animação de correr na Figura 21.

Nesse caso, foi configurado para que a animação rode em *loop* infinito, repetido enquanto o personagem do jogador estiver correndo.

Depois de criada as animações de um objeto, é necessário criar um *Animator Controller* que tem a função de controlar os estados de animação e como transitar entre

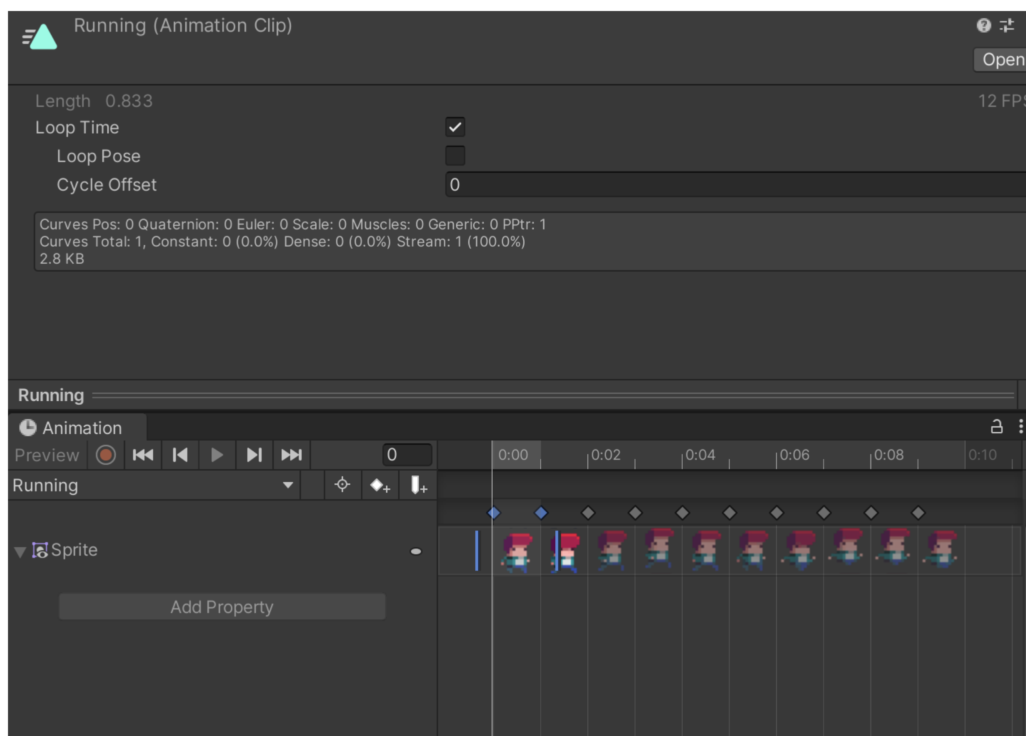

Figura 21 – *Animação de correr*

elas, assim como uma Máquina de Estados Finita.

Uma máquina de estados finita (FSM - do inglês Finite State Machine) ou autômato finito é um modelo matemático usado para representar programas de computadores ou circuitos lógicos. O conceito é concebido como uma máquina abstrata que deve estar em um de um número finito de estados. A máquina está em apenas um estado por vez, este estado é chamado de estado atual. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada num estado, no início do sistema, até o momento presente. Uma transição aqui indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Já um ação é a descrição de uma atividade que deve ser realizada num determinado momento (Wikipédia, 2023a).

Dessa forma, existem três estados pré existentes: *Entry* é estado inicial, disparado quando o jogo é iniciado. Já *Exit* representa o estado final, caso queira-se encerrar a lógica de animações. Por fim, *Any State* é um facilitador que permite a qualquer outro estado chegar até este.

Nesse sentido, mais estados podem ser criados, porém para o presente trabalho os demais estados que serão adicionados são relativos às animações criadas para determinado objeto, como a Figura 22 do personagem do jogador:

Aqui, a partir do estado *Entry* existe uma transição sem condição para a animação de *Idling* que representa o personagem parado com pequenas movimentações. De *Idling* existe uma transição para a animação *Climbing* com a condição de que a variável booleana "isClimbing"seja "true", e voltando pra *Idling* quando "false". Da mesma forma temos uma

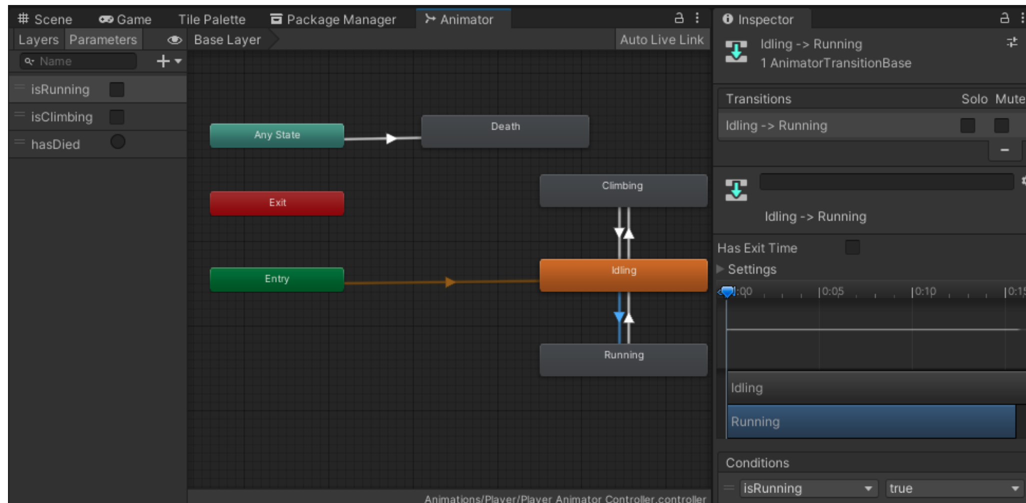

Figura 22 – *Animator Controller do personagem do jogador*

transição de *Idling* para *Running* (animação criada na figura 15) com a condição de que a variável booleana "isRunning"seja "true", e voltando pra *Idling* quando "false". Por fim, será configurado que a partir de qualquer estado (*AnyState*) seja possível ir ao estado de *Death* que representa a animação de quando o personagem do jogador morre, na qual a transição se dá através do momento em que a variável booleana "hasDied"assume o valor "true".

As variáveis acima mencionadas podem ser acessadas dentro das funções, como a que trata das movimentações do jogador por exemplo, ou seja, quando ele estiver em movimento é possível mudar o valor da variável "isRunning"para "true"e quando parar para "false"e assim por diante, gerenciando quando e como as transições ocorrerão. Dito isso, com tudo configurado, basta adicionar ao objeto que deseja executar essas animações o componente *Animator* passando um *Animator Controller* para que todas essas animações e estados sejam automaticamente gerenciadas e executadas.

3.5.8 Áudio

O áudio em jogos desempenha um papel crucial na criação de uma experiência imersiva e envolvente para os jogadores. Ele pode estabelecer atmosfera, transmitir emoção e melhorar a experiência, tornando-a mais divertida e dinâmica.

Geralmente, ao tratar-se de áudio em jogos, resume-se em dois principais tipos: trilha sonora e efeitos sonoros. A trilha sonora de um jogo é composta por músicas e melodias ambientes, que são reproduzidas durante o jogo. Ela ajuda a definir o tom e a atmosfera do jogo, adicionando emoção e enfatizando momentos importantes. A música de fundo pode variar de tranquila e serena a intensa e empolgante ou ainda tensa e assustadora, de acordo com a situação e a ambientação desejada para o jogo.

Já os efeitos sonoros são sons individuais usados para representar ações, interações

e eventos no jogo. Eles podem incluir sons de itens coletados, passos, disparos de armas, impactos, explosões, ruídos ambientais, entre outros. Os efeitos sonoros contribuem para a imersão do jogador e ajudam a transmitir informações auditivas importantes, como a localização de um inimigo ou a proximidade de perigos.

Na Unity existem várias formas de adicionar áudio aos jogos, desde soluções mais simples até gerenciadores completos de áudio. Para soluções mais simples pode-se usar o componente *AudioSource* que permite reproduzir um áudio específico e pontual em um objeto no jogo, podendo ser usado por exemplo para efeitos sonoros. Seria possível também criar uma solução mais completa, com um componente gerenciador de áudio, que centralizaria toda a lógica relacionada a áudio do jogo. Este gerenciador de Áudio gerenciaria eventos de áudio, reproduziria áudio quando necessário e manteria tudo organizado e padronizado, permitindo inclusive alterar grupos de áudios, ao invés de apenas isoladamente.

Todavia, para esse protótipo será utilizada a primeira solução, por ser mais simples e rápida de implementar, focando apenas em efeitos sonoros. Para utilizar o componente *AudioSource* existem duas possibilidades de funções:

PlayOneShot é um método que necessita de uma instância, ou seja, de um componente *AudioSource* existente no objeto para reproduzir um determinado clipe de áudio. O clipe é reproduzido na posição de qualquer objeto ao qual o componente *AudioSource* esteja anexado. Para esta solução seria preciso, então, adicionar um componente de *AudioSource* no objeto.

Todavia, existe a função estática *PlayClipAtPoint* que pode ser usada sem a necessidade de adicionar um componente *AudioSource* ao objeto. Para isso, ao chamar o referido método um *AudioSource* é criado um componente de áudio em determinada posição e usado para reproduzi-lo, para ao encerrar ser automaticamente destruído. Este método, apesar de ser menos performático, é mais simples de adicionar e permite que a remoção dessa lógica e futura evolução da solução de áudio para um gerenciador centralizado seja mais simples, utilizando inclusive os eventos criados anteriormente como o *CoinPickupEvent*.

Dito isso, para reproduzir efeitos sonoros com a solução do *PlayClipAtPoint*, são necessárias apenas duas etapas. A primeira etapa é parametrizar o áudio que será reproduzido:

```
1 [SerializeField] AudioClip audioClip;
```

Código 16 – Parâmetro de clip de áudio

E, finalmente, chamar o método *PlayClipAtPoint* passando o áudio e a posição do jogo a qual será reproduzido. Como o jogo é 2D e não teremos áudio 3D, todos os áudios

serão reproduzidos no centro da câmera, conforme:

```
AudioSource.PlayClipAtPoint(audioClip, Camera.main.transform.position);
```

Código 17 – Implementação para reproduzir áudio em um ponto

E assim, com apenas duas linhas, é possível reproduzir áudio para qualquer objeto no jogo.

3.5.9 Gerenciador de níveis

A função de um gerenciador de níveis em um jogo é controlar a transição e o fluxo entre diferentes níveis ou fases do jogo. Esse componente é responsável por coordenar o carregamento de cenários, recursos e elementos específicos de cada fase do jogo.

Para isso, foi criado um script simples *LevelManager* que conterá apenas um método público *LoadNextLevel*. Para controlar os níveis, chamados no Unity de cenas, será necessário criar mais de uma cena e usar o *SceneManager* da própria Unity para transitar entre eles. A lógica será simples, carregar o próximo nível ou, no caso de ser o último, voltar para o primeiro nível.

```
public void LoadNextLevel() { StartCoroutine(loadNextLevel()); }
private IEnumerator loadNextLevel()
{
    yield return new WaitForSecondsRealtime(levelLoadDelayInSeconds);
    if (isFinalLevel()) { SceneManager.LoadScene(0); }
    else {
        int currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
        FindObjectOfType<LevelPersistence>().ResetLevelPersist();
        SceneManager.LoadScene(currentSceneIndex + 1); }
}
bool isFinalLevel(){
    return (SceneManager.GetActiveScene().buildIndex + 1) == SceneManager.sceneCountInBuildSettings;
}
```

Código 18 – Script para carregar próximo nível

Agora resta escolher uma forma de identificar o final de um nível para chamar o método *LoadNextLevel* recém criado. Por questões de simplicidade será criado o objeto *LevelExit* que representará o final do nível e a transição para o próximo, seja uma porta, portal ou qualquer outra coisa.

Nele será adicionado novamente um componente de colisão *trigger* para identificar o acionamento dessa funcionalidade quando o jogador se aproximar. No script desse objeto, será chamado o referido método quando entrar no alcance do *trigger*.

Pronto, agora já existe a funcionalidade básica de passar de um nível para o outro, gerenciado pelo recém criado *LevelManager*.

```
void OnTriggerEnter2D(Collider2D other)
{
    LevelManager.GetInstance().LoadNextLevel();
}
```

Código 19 – Exemplo de função trigger para próximo nível

3.5.10 Sistema de diálogo

Um sistema de diálogo em jogos é projetado para permitir que os jogadores interajam com personagens não jogáveis (NPC) e outros elementos do jogo por meio de conversas e trocas de informações. O objetivo é criar uma experiência de jogo mais imersiva e envolvente, fornecendo narrativa, contexto e opções de escolha ao jogador, que podem ser utilizados nos mais diversos cenários, dependendo de cada jogo.

O sistema de diálogo é composto por diálogos estruturados, onde cada interação é apresentada como uma sequência de texto entre o jogador e o personagem do jogo. Durante as interações, o jogador é apresentado com opções de escolha, geralmente na forma de botões, que representam diferentes respostas ou ações possíveis. Essas escolhas podem desencadear reações no jogo, como morte, ganho de algum item, entre outros.

Para criação dos diálogos será usado o *Ink*, uma linguagem de script para narrativa para jogos, conforme exemplo:

```
-> main
A ponte parou de funcionar
A chave correta para arrumar a ponte esta marcada com a sigla de um METAL ALCALINO.
Escolha uma das opcoes:
    + [Metal nao alcalino]
        -> chosen("Metal nao alcalino")
    + [Metal alcalino]
        -> chosen("Metal alcalino")
    + [Metal nao alcalino]
        -> chosen("Metal nao alcalino")

Voce escolheu: {opcao}!
-> END
```

Código 20 – Exemplo de linguagem Ink para narrativa de jogos

Cada linha do Ink representa uma parte do diálogo e a divisão em linhas é feita para quebrar o texto em pequenas partes, permitindo ao jogador ir para o próximo texto pressionando um botão qualquer. Já os textos após o símbolo "+"representam as opções de resposta e o que vem após o símbolo " ->"está atrelado apenas ao código, salvando e repassando o valor escolhido pra cada opção, sendo possível acessá-lo no final, conforme exemplo "{opcao}".

Após criados todos os diálogos resta compilar o código que irá gerar um arquivo *JSON* para cada um deles, para depois ser utilizado no jogo. Com esses arquivos, resta criar um sistema para gerenciar e disparar os diálogos, e aqui serão o *DialogueManager* e o *DialogueTrigger* respectivamente.

Inicialmente é preciso que o jogador esteja no alcance do NPC para ser possível interagir com ele e começar o diálogo. Para isso, será utilizado novamente o componente de colisão *trigger*. Será criada uma variável booleana *playerIsInRange* e ao entrar na colisao assumirá o valor "true"e na saída voltará para "false".

```
1  private void OnTriggerEnter2D(Collider2D otherCollider)
2  {
3      playerIsInRange = true;
4  }
5
6  private void OnTriggerExit2D(Collider2D otherCollider)
7  {
8      playerIsInRange = false;
9  }
```

Código 21 – Uso de colisão trigger para alterar booleano

Dado que está ao alcance, é necessário aguardar o *input* do jogador para iniciar o diálogo. Para isso, como já foi criado o sistema que recebe as entradas do jogador, basta aguardar por essa entrada no método *Update* do Unity:

```
1  [SerializeField] private TextAsset inkJSON;
2
3  void Update()
4  {
5      if(playerIsInRange && !DialogueManager.GetInstance().dialogueIsPlaying)
6      {
7          if(InputManager.GetInstance().GetInteractPressed())
8          {
9              DialogueManager.GetInstance().EnterDialogueMode(inkJSON);
10         }
11     }
12 }
```

Código 22 – Função base para entrar no modo diálogo

O *trigger* que irá disparar o diálogo está pronto, agora basta implementar o *DialogueManager*, script que irá gerenciar qualquer diálogo. Antes disso, será criado um painel UI (User Interface, Interface do usuário) que irá apresentar os textos do diálogo.

Depois será criado nesse script o método *EnterDialogueMode* que dará inicio ao diálogo, de forma a basicamente criar uma nova história com a biblioteca Ink. Esse método receberá como parâmetro o arquivo *JSON* gerado para cada script Ink e que será posteriormente transformado efetivamente em diálogo no jogo:

Começando pelo final, será criada uma *Coroutine* que é nada mais do que uma rotina assíncrona que permite pausar e retomar sua execução, passando o método *ExitDialogueMode* que basicamente encerra o diálogo, limpando os textos alterados pelo diálogo em questão e aguardando 200 milissegundos para não fechar o painel abruptamente ao realizar a escolha:

Todavia, para ser um diálogo é preciso que ambas as partes interajam. O NPC já produziu o texto contendo sua fala e agora falta a interação do jogador. Para isso, o

```
public void EnterDialogueMode(TextAsset inkJSON) {
    currentStory = new Story(inkJSON.text);
    dialogueIsPlaying = true;
    dialoguePanel.SetActive(true);

    ContinueStory();
}

private void ContinueStory()
{
    if (currentStory.canContinue) {
        //set text for the current dialogue line
        dialogueText.text = currentStory.Continue();
        // display choices, if any, for this dialogue line
        DisplayChoices();
    }
    else { StartCoroutine(ExitDialogueMode()); }
}
```

Código 23 – Função atualizada para entrar no diálogo

```
private IEnumerator ExitDialogueMode() {
    yield return new WaitForSecondsRealtime(0.2f);

    dialogueIsPlaying = false;
    dialoguePanel.SetActive(false);
    dialogueText.text = "";
}
```

Código 24 – Função para sair do diálogo

método *DisplayChoices* apresentará botões previamente criados, com todas as opções de interação para que o jogador possa realizar a escolha.

```
private void DisplayChoices(){
    List<Choice> currentChoices = currentStory.currentChoices;
    int index = 0;
    foreach (Choice choice in currentChoices) {
        // enable and initialize each choice
        choices[index].gameObject.SetActive(true);
        choicesText[index].text = choice.text;
        index++;
    }
}
```

Código 25 – Função para mostrar as opções de diálogo

Para que isso funcione corretamente, o botão deve ter uma ação atrelada ao clique dele, ou seja, ao clicar nele deve-se efetivamente realizar a escolha dessa opção e o diálogo prosseguir ou ser finalizado, conforme Figura 23.

Aqui o clique do botão aponta para a função *MakeChoice* do *DialogueManager* que será executada passando como parâmetro o índice do botão clicado, para assim ser possível saber qual escolha foi feita.

Dado que o índice da escolha foi informado, resta validar se é o mesmo índice da opção correta, previamente configurada no *DialogueManager* e, caso seja, enviar um evento de resposta respondida com sucesso ou um evento de resposta errada. Por enquanto não existe nenhum script ouvindo esses eventos, todavia mais tarde será possível criá-los

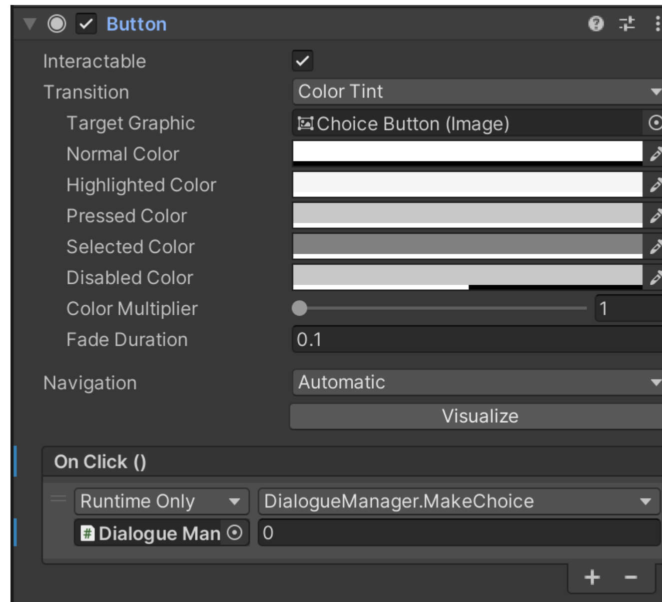

Figura 23 – *Configuração da ação atrelada ao clique do botão de realizar escolhas*

para tomares ações pertinentes a cada um deles.

```
public void MakeChoice(int choiceIndex)
{
    currentStory.ChooseChoiceIndex(choiceIndex);
    if (choiceIndex == correctChoiceIndex) { choiceCorrectlyAnswered.Invoke(); }
    else { choiceWronglyAnswered.Invoke(); }
}
```

Código 26 – Função para realizar escolha em um diálogo

E assim encontra-se finalizado um protótipo de um sistema simples de diálogo, conforme exemplo na Figura 24, que permite gerenciar qualquer comunicação entre um NPC e um jogador. Esse sistema pode ser uma ferramenta a ser utilizada para diversos fins, seja para aprofundar a narrativa do jogo, possibilitar que o jogador passe por determinada parte do nível, receba itens ou etc. Nas próximos sessões serão trabalhadas algumas dessas possibilidades ao inicializar a criação dos níveis propriamente ditos.

Figura 24 – *Mecânica de diálogo e NPC*

3.5.11 Sistema de questionário (*quiz*)

A última funcionalidade geral prototipada para esse jogo foi a criação de um sistema de questionário. Na prototipagem, os textos das perguntas e respostas são fixos, mas ressalta-se que haverá uma integração com a API da QuimiCot Games para obter essas questões de forma dinâmica, cujas questões são criadas pelos professores de Química associados à cada turma.

Vale lembrar também que a necessidade de um questionário neste jogo está diretamente atrelado ao caráter educativo dele e, também, à forma com que a plataforma QuimiCot Games optou para avaliar e medir a aprendizagem e desempenho dos jogadores.

Nesse sentido, os administradores da plataforma QuimiCot Games podem avaliar o conhecimento do aluno sobre os conceitos e conteúdos abordados no referido jogo, permitindo identificar pontos de falhas, de acertos e onde cabem melhorias para futuramente serem trabalhadas.

Outra função é a de reforçar o conteúdo. Questões bem formuladas incentivam a revisão do conteúdo, permitindo que o aluno relembre informações importantes, o que é essencial para a retenção dessa informação e para que ela se transforme em conhecimento. Existem vários outros fatores e possibilidades de usos que podem ser realizados em cada cenário específico.

Neste jogo, os questionários serão mostrados antes da transição para o próximo nível, e neles serão abordados temas diversos, que serão obtidos através da QuimiCot Games. Por hora, para simplificar será feito um um questionário fixo, apenas para prototipar a mecânica, o qual se tornará dinâmica a partir da futura integração com a plataforma.

O script *QuizProvider* irá fornecer um questionário em formato de tela UI, possibilitado interação do jogador através da interface de usuário do Unity. Isso será inicializado por meio do método *StartQuiz*, contando também com a criação posterior de alguns métodos auxiliares.

O primeiro deles será o método auxiliar *GetQuiz* que retornará um objeto *Quiz* representando o questionário e passará uma função *callback*, que após concluído o processo do referido método irá executá-lo. O valor real desse objeto futuramente será obtido a partir da chamada direta ao serviço da plataforma Quimicot Games. Todavia, conforme dito anteriormente, no momento em questão será retornado um objeto fixo com o intuito de prototipar esta funcionalidade:

Agora basta definir o método *StartQuiz* chamando o método *GetQuiz*, que ao concluir chamará o método *CreateQuizUI* como *callback*:

Depois será implementado o método *CreateQuizUI*, que será responsável por criar a interface com o usuário, baseado no objeto *Quiz*, previamente obtido.

```
[Serializable]
public class Quiz {
    public string quiz_id;
    public string pergunta;
    public List<Alternativa> alternativas;
}
[Serializable]
public class Alternativa {
    public string id;
    public string descricao;
    public Alternativa(string id, string descricao)  { this.id = id; this.descricao = descricao; }
}

private IEnumerator GetQuiz(Action<Quiz> onGetQuizCompleteAction) {
    quiz = new Quiz;
    quiz.quiz_id = "123";
    quiz.pergunta = "Qual é um metal alcalino?";
    alternativas = new List<Alternativa>();
    alternativas.Add(new Alternativa("1", "elemento 1"));
    alternativas.Add(new Alternativa("2", "elemento 2"));
    alternativas.Add(new Alternativa("3", "elemento 3"));
    quiz.alternativas = alternativas;
    onQuizCompleteAction(quiz);
}
```

Código 27 – Script para retornar questionário

```
public void StartQuiz() { StartCoroutine(getQuiz(createQuizUI)); }
```

Código 28 – Função para começar questionário

```
private void CreateQuizUI(Quiz quiz) {
    quizUI.SetActive(true); // habilitar Quiz UI
    quizQuestionText.text = quiz.pergunta; // configurar texto da pergunta

    // criar todos os botoes com as alternativas do questionário
    for (int i = 0; i < quiz.alternativas.Count; i++) {
        GameObject currentButton = Instantiate(quizAnswerButtonPrefab, quizUI.transform);
        var initialPosY = (quizUI.transform.position.y + 50);
        currentButton.transform.position = new Vector3(quizUI.transform.position.x, initialPosY-(i*60), 0)

        currentButton.name = quiz.alternativas[i].id;
        TextMeshProUGUI buttonText = currentButton.GetComponentInChildren<TextMeshProUGUI>();
        buttonText.text = quiz.alternativas[i].descricao;

        currentButton.GetComponent<Button>().onClick.AddListener(delegate { SendUserQuizAnswer(); });
        currentButton.GetComponent<Button>().enabled = true;
    }
}
```

Código 29 – função para criar questionário

Para criar a UI do questionário, serão precisos três passos. O primeiro é habilitar a UI criada anteriormente. Depois, substituir o texto da pergunta pelo que veio do objeto Quiz. Por último, criar cada um dos botões que representam as alterativas em uma altura, um abaixo do outro e, para cada botão, atualizar o texto da alterativa e adicionar uma ação que será disparada ao ouvir o evento de clique desse botão.

A ação disparada no clique em uma das alternativa será o método *SendUserQuizAnswer*, que logo mais será implementado. Este método futuramente enviará a resposta à plataforma QuimiCot Games e como retorno desse envio terá o gabarito do questionário, junto com a justificativa do porquê cada alternativa está correta ou errada. Como nesse

primeiro momento não serão realizadas a chamada à plataforma, será criar novamente um objeto fixo, simulando o retorno deles com o gabarito:

```
[Serializable]
public class QuizAnswerSheet
{
    public int correto;
    public string idDaResposta;
    public List<AlternativasJustificada> alternativasJustificadas;
}

[Serializable]
public class AlternativasJustificada
{
    public int id;
    public string quiz;
    public int alt_correta;
    public string descricao;
    public string justificativa;
}

public void SendUserQuizAnswer()
{
    answerId = EventSystem.current.currentSelectedGameObject.name;

    quizAnswerSheet = new QuizAnswerSheet;
    quizAnswerSheet.correto = true;

    alternativasJustificadas = new List<AlternativasJustificada>();
    alternativasJustificadas.Add(new AlternativasJustificada(1,"quizId",0,"descricao1","justificativa1"));
    alternativasJustificadas.Add(new AlternativasJustificada(2,"quizId",1,"descricao2","justificativa2"));
    alternativasJustificadas.Add(new AlternativasJustificada(3,"quizId",0,"descricao3","justificativa3"));

    quizAnswerSheet.alternativasJustificadas = alternativasJustificadas;

    addAnswerSheetToQuiz(answerId, quizAnswerSheet);
}

```

Código 30 – Função para enviar resposta fixa do questionário

Aqui vale ressaltar que será obtido o ID (identificador) da resposta pelo nome do botão que foi configurado na criação da UI do questionário, utilizando o *EventSystem* do Unity para descobrir qual botão que enviou o evento de clique. Depois disso, serão adicionados ao texto dos atual dos botões essas justificativas recentemente obtidas, além de informar ao jogador se acertou ou não ao escolher a alterativa. Após isso, já será possível avançar para a próxima fase.

Aqui o botão será buscado, desativando sua ação de clique, adicionado a justificativa ao texto do botão e, caso a resposta esteja correta alteramos a cor do fundo do botão para verde, senão para vermelho. Por fim, será criado um botão de "continuar", adicionando nele a ação de *LoadNextLevel* criada anteriormente no *LevelManager* que fará com que, ao ser clicado, avance para o próximo nível.

Agora, no objeto *LevelExit* que representa o final do nível e transição para o próximo, será trocada a chamada do *LevelManager* que muda para o próximo nível, pela chamada do método *StartQuiz* do *QuizManager* que iniciará o questionário e culminará na mesma funcionalidade anterior de avançar para o próximo nível.

```
private void addAnswerSheetToQuiz(string asnwerId, QuizAnswerSheet quizAnswerSheet)
{
    foreach (var quizJustifiedAnswer in quizAnswerSheet.alternativasJustificadas)
    {
        GameObject choiceButton = quizUI.transform.Find(quizJustifiedAnswer.id.ToString()).gameObject;

        choiceButton.GetComponent<Button>().enabled = false;

        TextMeshProUGUI buttonText = choiceButton.GetComponentInChildren<TextMeshProUGUI>();
        buttonText.text += ("\n" + quizJustifiedAnswer.justificativa);

        if (quizJustifiedAnswer.alt_correta == 1)
        {
            choiceButton.GetComponent<Image>().color = Color.green;
        }
        if (asnwerId == quizJustifiedAnswer.id.ToString() && quizJustifiedAnswer.alt_correta == 0)
        {
            choiceButton.GetComponent<Image>().color = Color.red;
        }
    }

    GameObject continueButton = Instantiate(quizContinueButtonPrefab, quizUI.transform);
    continueButton.GetComponent<Button>().onClick.AddListener(
        delegate { LevelManager.GetInstance().LoadNextLevel();
    });
}
```

Código 31 – Função para adicionar gabarito ao questionário

```
void OnTriggerEnter2D(Collider2D other)
{
    QuizProvider.GetInstance().StartQuiz();
}
```

Código 32 – Função para disparar o questionário

Por fim, vale ressaltar que não é preciso acertar o questionário para progredir, basta respondê-lo e clicar no botão 'continuar' que aparecerá ao fim do processo para prosseguir normalmente, conforme Figura 25. Vale lembrar que o questionário não possui o objetivo de dificultar a passagem entre níveis, mas sim servir como ferramenta auxiliar para avaliar o conhecimento do jogador.

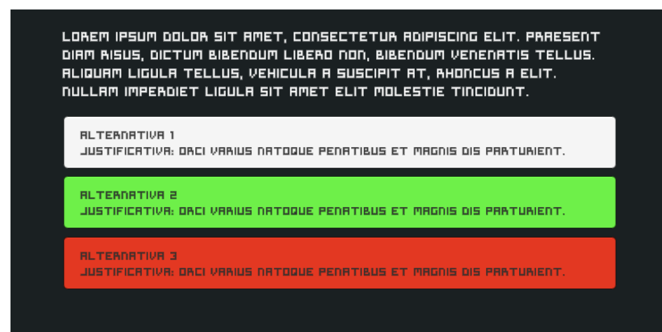

Figura 25 – *Mecânica de questionário*

3.6 Criação dos níveis e de suas especificidades

3.6.1 Nível 1 - ponte e abismo

A mecânica principal desse nível é ter um abismo que impeça o jogador de passar para o outro lado, a menos que escolha a chave correta. Nesse caso, uma vez escolhida a chave correta, uma ponte deverá ser ativada permitindo o progresso no nível.

Para criar o conceito acima de abismo é simples, basta utilizar os *tilemaps* que criados anteriormente. O primeiro é o *tilemap* de plataforma, criando um mapa alto como um precipício, com distancia significativa entre os lados a fim de evitar que o usuário consiga passar para o outro lado sem a ponte. Por fim, basta usar o *tilemap* de armadilhas no fundo do abismo, que, no caso do personagem do jogador cair ocasionará sua ocasionando a morte imediata, impedindo o progresso, conforme Figura 26.

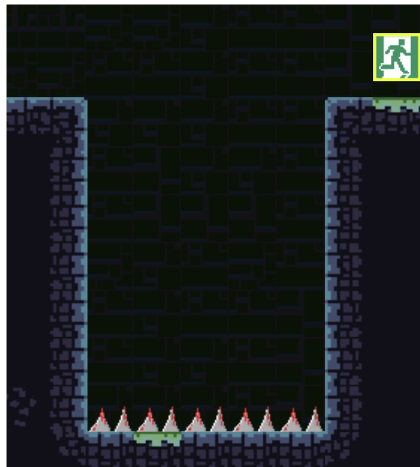

Figura 26 – *Abismo do nível 1*

Com o abismo pronto, resta criar a mecânica da ponte para passar dele. Para isso, será criado um NPC que irá ativar um diálogo para esse nível, que será gerenciado pelo já criado *DialogueManager*. No caso de o jogador escolher uma opção correta, o referido script irá enviar o evento *choiceCorrectlyAnswered*.

```
1  public class SetActiveByChoiceAnsweredCorrectlyEventListener : MonoBehaviour
2  {
3      [Header("Target Object")]
4      [SerializeField] private bool shouldBeSetToActive = true;
5
6      void Start()
7      {
8          gameObject.SetActive(!shouldBeSetToActive);
9          DialogueManager.choiceCorrectlyAnswered.AddListener(SetActive);
10     }
11     private void SetActive()
12     {
13         gameObject.SetActive(shouldBeSetToActive);
14     }
15 }
```

Código 33 – Script para habilitar objeto diante do evento de resposta correta

O próximo passo agora é criar um objeto que represente a ponte e deixar ele escondido no cenário, ou seja, desabilitado. Será adicionado a este objeto um novo script *SetActiveByChoiceAnsweredCorrectlyEventListener* que ouvirá pelo evento acima mencionado e, quando for disparado, habilitará o objeto da ponte, permitindo que apareça na cena e possa ser usado para passar pelo abismo, conforme Figura 27.

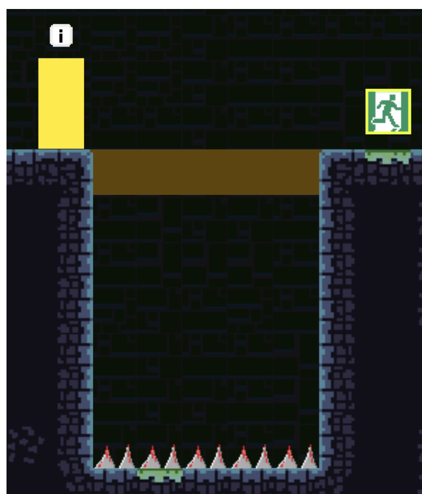

Figura 27 – *Adição da mecânica da ponte no nível 1*

3.6.2 Nível 2 - raios

Nesta fase é necessário criar um mecanismo de raios que eletrocutem o jogador caso ele tente passar. Para progredir é necessário escolher um guarda-chuva feito com o elemento ideal que irá repelir os raios.

Para a mecânica dos raios foram estudados algumas formas de design do jogo. A primeira seria os raios caírem aleatoriamente em diversos pontos do mapa do nível, mas isso permitiria a possibilidade de o jogador progredir na fase sem escolher o guarda chuva correto e, portanto, foi descartada. A forma escolhida foi a de criar nuvens em posições específicas do mapa que ativarão o raio caso o jogador tente passar por ela.

Colocando em prática, será criado um objeto *Cloud* que representará a nuvem e terá um componente de colisão *trigger* responsável por identificar quando o usuário passou pela nuvem e futuramente ativará o raio. Para criar o raio, será necessário outro script, chamado aqui de *ActivateLigthing* que será colocado no objeto acima mencionado. Este script será dividido em três principais funcionalidades.

A primeira será a de, a cada *frame* do jogo (utilizando a função *FixedUpdate* do Unity que é executada a cada frame do jogo), calcular a distância entre a nuvem e um objeto de colisão, seja o chão ou o jogador. Isso permite que posicionemos a nuvem em qualquer posição e em qualquer altura para que o raio criado por ela seja do tamanho exato da distancia entre a nuvem e o objeto que irá se chocar.

```
public class ActivateLigthingVisual : MonoBehaviour {
    int distance = 0;

    void FixedUpdate() {
        RaycastHit2D hit = Physics2D.Raycast(transform.parent.position, -Vector2.up);
        if (hit.collider == null) { return; // não colidiu com nada }
        distance = Mathf.FloorToInt(hit.distance);
    }
}
```

Código 34 – Script para calcular distância entre o chão e a nuvem

Para isso será criada uma variável *distance* que guardará o valor atual da distância e para obtê-la será utilizada a funcionalidade *Raycast* do Unity, o qual detecta objetos que estão ao longo do caminho entre o ponto de origem e uma direção e é conceitualmente como disparar um feixe de laser na cena e observar quais objetos são atingidos por ele (Unity Technologies, 2023g). Dessa forma, o objeto irá disparar uma linha para baixo e registrar informações da colisão, como a distância entre eles, que será útil para este caso.

O segundo passo é utilizar a função *OnTriggerEnter2D* para criar o raio quando o jogador passar embaixo da nuvem. Para representar o raio será criado o objeto *Lightning* que conterá o script *DamageDealer* anteriormente criado e, assim, causará dano ao jogador ao entrar em contato com ele.

Este objeto será passado como parâmetro e sua altura será alterada para ser a mesma da distância que foi pré calculada e salva na variável *distance*.

```
[SerializeField] private GameObject lightningPrefab;

void OnTriggerEnter2D(Collider2D other)
{
    GameObject lightning = Instantiate(lightningPrefab, this.transform.parent);
    lightning.GetComponent<RectTransform>().pivot = new Vector2(transform.parent.position.x, distance);

    StartCoroutine(EndLightning(lightning));
}
```

Código 35 – Script para criar raio

Criado o raio, o último passo é implementar a rotina *EndLightning* que, passado determinado tempo, irá destruir o raio utilizando o método *Destroy* do Unity.

```
IEnumerator EndLightning(GameObject lightning)
{
    yield return new WaitForSecondsRealtime(.2f);
    Destroy(lightning);
}
```

Código 36 – Função para encerrar raio

Com a mecânica dos raios pronta, falta apenas criar um NPC que irá fornecer os guarda-chuvas para que seja possível progredir e passar pelos raios sem sofrer dano. Para isso, será criado novamente um diálogo e reutilizado o prefab do objeto NPC criado

para o nível anterior. Este NPC em específico terá a adição de um script nele, *CollectableSummonerOnEvent*, que será responsável por criar um objeto guarda-chuva com o elemento que irá repelir os raios no caso de o jogador escolher uma opção correta, por meio do envio do evento *choiceCorrectlyAnswered* já mencionado. E, caso escolha uma opção incorreta, criará um objeto guarda-chuva que não irá bloquear os raios, utilizando o evento *choiceWronglyAnswered* também enviado pelo *DialogueManager*.

```
public class CollectableSummonerOnEvent : MonoBehaviour
{
    [SerializeField] private GameObject nextLevelKeyObjectToBeSummoned;
    [SerializeField] private GameObject dummyObjectToBeSummoned;

    void Start()
    {
        DialogueManager.choiceCorrectlyAnswered.AddListener(SummonNextLevelKeyObject);
        DialogueManager.choiceWronglyAnswered.AddListener(SummonDummyKeyObject);
    }

    private void SummonNextLevelKeyObject()
    {
        Summon(nextLevelKeyObjectToBeSummoned);
    }

    private void SummonDummyKeyObject()
    {
        Summon(dummyObjectToBeSummoned);
    }

    private void Summon(GameObject objectToBeSummoned)
    {
        Vector3 positionToBeSummoned = new Vector3(transform.position.x + 3, transform.position.y - 0.5f, 0f);
        Instantiate(objectToBeSummoned, positionToBeSummoned, Quaternion.identity);
    }
}
```

Código 37 – Script para criar objeto coletável

Por fim, resta apenas criar os objetos de guarda chuva. O primeiro é um objeto sem nenhum componente de colisão, o que não evita o raio. O segundo tem um componente de colisão que faz com que bloqueie o raio, não encostando e, consequentemente, não causando dano no jogador. A Figura 28 mostra uma captura de tela com o raio (verde) e o guarda-chuva (rosa) no nível 2.

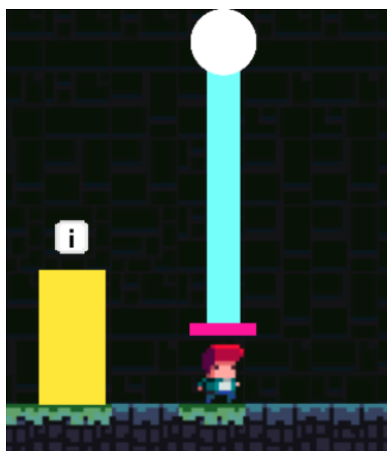

Figura 28 – *Mecânica do raio e do guarda-chuva no nível 2*

3.6.3 Nível 3 - porta e conexão de fios entre diferentes polos

Na última fase, o mecanismo principal é o da conexão entre fios com polos distintos a fim de religar a energia da porta e fazer com que ela se abra, permitindo o progresso pelo nível e encerramento do jogo.

Lembrando que a ideia discutida na concepção foi a de unir a referida mecânica com a plataforma, com o objetivo de o jogador continuar movimentando por plataformas e interagindo com interruptores que se comportarão como os polos. Ao interagir com um, aguarda-se a interação com o segundo e caso os elementos sejam corretos, a conexão é bem sucedida, senão o jogador perderá pontos de vida.

Inicialmente é criado um objeto contendo um componente de colisão que representará a porta e também vários objetos que representarão os polos. Agora para dar a funcionalidade a esses polos criaremos o script *InteractableDoorPole* habilitando cada polo de possuir um sinal (positivo ou negativo) identificando-o e um nome do elemento. Serão também capazes de conectar-se um ao outro e culminar na abertura da porta, mas esta parte será trabalhada mais a frente.

```
1  public class InteractableDoorPole : MonoBehaviour
2  {
3      [SerializeField] public bool isPositivePole;
4      [SerializeField] private GameObject visualCue;
5      private bool playerIsInRange = false;
6
7      public static UnityEvent<GameObject> activatedDoorPole { get; private set; }
8  }
```

Código 38 – Base do script para polos que abrirão a porta

Aqui estarão apenas os atributos principais que um polo possui, o sinal do polo, ou seja, se é positivo ou negativo, representado pela variável booleana *isPositivePole*. Além disso, conterá um objeto visual (*visualCue*) que indicará ao jogador, ao se aproximar (identificado pela variável booleana *playerIsInRange*), que é possível interagir com o polo. Por fim, o evento *activatedDoorPole* que será disparado quando o jogador interagir com o polo em questão.

```
1  void Update()
2  {
3      if(playerIsInRange) { visualCue.SetActive(true);}
4      else { visualCue.SetActive(false); }
5  }
6
7  private void OnTriggerEnter2D(Collider2D otherCollider) { playerIsInRange = true; }
8  private void OnTriggerExit2D(Collider2D otherCollider) { playerIsInRange = false; }
```

Código 39 – Funções que controlam o aparecimento de um objeto visual para indicar interatividade.

Para ativar o objeto visual (*visualCue*) que indicará a possibilidade de interação, é simples. Para isso, basta adicionar um componente de colisão *trigger* e, ao entrar ou sair

do alcance, alterar a variável *playerIsInRange*, para que assim seja possível mostrá-lo ou escondê-lo.

Com o objeto visual pronto, basta alterar o código para aguardar por uma interação do jogador quando ele estiver no alcance do polo, ouvindo pela ação de interação *GetInteractPressed* e enviando o evento anteriormente criado.

```
1  if(playerIsInRange)
2  {
3      visualCue.SetActive(true);
4      if(InputManager.GetInstance().GetInteractPressed())
5      {
6          activatedDoorPole.Invoke(this.transform.parent.gameObject);
7      }
8  }
```

Código 40 – Função que recebe input para interagir com polo

Agora, resta criar um objeto com o script *DoorActivatorManager* que será responsável por gerenciar toda essa mecânica da porta e da interação entre os polos.

```
1  public class DoorActivatorManager : MonoBehaviour
2  {
3      [Header("Connection Points")]
4      [SerializeField] private int connectionsToBeStablished;
5      [SerializeField] private int maxWrongConnections;
6
7      [Header("Line Renderer")]
8      [SerializeField] private LineRenderer lineRendererPrefab;
9
10     private GameObject firstPoleActivated;
11     private GameObject secondPoleActivated;
12
13     private int connectedCorrectly;
14     private int connectedWrongly;
15
16     void Start()
17     {
18         InteractableDoorPole.activatedDoorPole.AddListener(HandlePoleActivation);
19     }
20 }
```

Código 41 – Script de gerenciador de polos para ativação de porta

A base do script serão algumas variáveis de controle como quantas conexões serão possíveis de realizar (*connectionsToBeStablished*), número máximo de conexões ou de tentativas erradas (*maxWrongConnections*), quantas conexões já foram bem sucedidas ou não (*connectedCorrectly* e *connectedWrongly* respectivamente) e os dois polos em questão (*firstPoleActivated* e *secondPoleActivated*).

Além disso, será utilizado o *Line Renderer* que é um componente do Unity capaz de desenhar uma linha entre dois ou mais pontos, a fim de ilustrar e dar um feedback visual para o jogador quando os polos estiverem conectados.

Ademais, no início do script ele começará a ouvir pelo evento de ativação de um polo (*activatedDoorPole*) que foi criado no script anterior e realizará o gerenciamento dessa ativação por meio da função *HandlePoleActivation*.

```
private void HandlePoleActivation(GameObject currentPole)
{
    if (firstPoleActivated == null) {
        firstPoleActivated = currentPole;
        firstPoleActivated.GetComponent<SpriteRenderer>().color = Color.yellow;
    } else {
        secondPoleActivated = currentPole;
        if (IsPoleSignDifferent()) { handleSuccessfulConnection(); }
        else { handleUnuccessfulConnection(); }
    }
    Clean();
}

private bool IsPoleSignDifferent() {
    InteractableDoorPole firstPole=firstPoleActivated.GetComponentInChildren(typeof(InteractableDoorPole));
    InteractableDoorPole secondPole=secondPoleActivated.GetComponentInChildren(typeof(InteractableDoorPole));

    return firstPole.isPositivePole != secondPole.isPositivePole;
}

private void Clean() {
    firstPoleActivated = null;
    secondPoleActivated = null;
}
```

Código 42 – Função para gerenciar a ativação de polos

A primeira parte dessa função é identificar o primeiro polo clicado e dar um feedback visual. Para isso serão utilizadas cores: a cor vermelha é a padrão, identifica que um polo está aguardando interação; a amarela identifica que um dos polos foi clicado, está ativo e aguardando uma interação com outro polo para tentativa de conexão; já a cor verde identifica que foram conectados com sucesso.

A segunda parte é o cerne da funcionalidade, avaliar o sinal dos polos (representado pela função *IsPoleSignDifferent*) e no caso de serem sinais diferentes, o que significa uma conexão bem sucedida, acionar a função de *handleSuccessfulConnection* que irá gerenciar as ações atreladas a essa conexão bem sucedida, senão acionar a função *handleUnuccessfulConnection* que irá tratar uma conexão mal sucedida. Por fim, resta apenas limpar as variáveis (representado pela função *Clean*) para, assim, aguardar as próximas tentativas de conexão.

Agora serão desenvolvidas as funções de conexões bem e mal sucedidas. A primeira (*handleSuccessfulConnection*) irá desenhar uma linha verde entre os polos, representando visualmente a conexão bem sucedida. Além disso, irá desativar esses polos, impedindo a interação com eles novamente. Por fim, será computado que uma conexão correta foi realizada.

Já a função *handleUnuccessfulConnection* irá desenhar uma linha vermelha entre os polos e computar uma conexão mal sucedida.

Por fim, resta implementar a lógica que irá efetivamente gerenciar essas informações. Essa avaliação será feita a cada frame na própria função *Update* do Unity, conforme:

De acordo com o código acima, no caso de o jogador conseguir estabelecer todas

```
private void handleSuccessfulConnection()
{
    LineRenderer lineRenderer = Instantiate(lineRendererPrefab, this.transform);
    firstPoleActivated.GetComponent<SpriteRenderer>().color = Color.green;
    firstPoleActivated.transform.GetChild(0).gameObject.SetActive(false);
    firstPoleActivated.transform.GetChild(1).gameObject.SetActive(false);
    firstPoleActivated.transform.GetChild(2).gameObject.SetActive(false);

    secondPoleActivated.GetComponent<SpriteRenderer>().color = Color.green;
    secondPoleActivated.transform.GetChild(0).gameObject.SetActive(false);
    secondPoleActivated.transform.GetChild(1).gameObject.SetActive(false);
    secondPoleActivated.transform.GetChild(2).gameObject.SetActive(false);

    lineRenderer.SetPosition(0, firstPoleActivated.transform.position);
    lineRenderer.SetPosition(1, secondPoleActivated.transform.position);

    connectedCorrectly++;
}
```

Código 43 – Trecho de função que lida com conexão bem sucedida de polos

```
private void handleUnuccessfulConnection()
{
    firstPoleActivated.GetComponent<SpriteRenderer>().color = Color.red;
    secondPoleActivated.GetComponent<SpriteRenderer>().color = Color.red;

    connectedWrongly++;
}
```

Código 44 – Trecho de função que lida com conexão mal sucedida de polos

```
void Update()
{
    if (connectedCorrectly == connectionsToBeStablished)
    {
        connectedCorrectly = 0;
        DialogueManager.choiceCorrectlyAnswered.Invoke();
    }
    if (connectedWrongly == maxWrongConnections)
    {
        connectedCorrectly = 0;
        PlayerController.OnDamage.Invoke(999);
    }
}
```

Código 45 – Trecho de script que lida com tentativas de conexões de polos

as conexões bem sucedidas será invocado o evento de resposta bem sucedida *choiceCorrectlyAnswered* que será então ouvido pelo objeto da porta e irá desativá-la, cumprindo a função de "abri-la"e permitindo que o jogador prossiga. Ou, no caso de atingir o máximo de conexões erradas, irá invocar o evento de *OnDamage* que irá causar dano total ao jogador, ocasionando sua morte imediata e fazendo-o iniciar a fase novamente.

Falta agora apenas adicionar o script *SetActiveByChoiceAnsweredCorrectlyEventListener* já criado anteriormente e usado pela ponte no nível 1, que irá ouvir o evento de resposta bem sucedida e desativar o objeto em questão, conforme exemplificado na Figura 29.

Assim, todas as funcionalidades essenciais e também as específicas desse jogo encontram-se concluídas.

Figura 29 – *Mecânica principal do último nível concluída*

4 Resultados e Discussões

4.1 Integração do jogo com as APIs da plataforma QuimiCot Games

Muitas plataformas de jogos oferecem APIs (Interface de Programação de Aplicativos ou, em inglês, *Application Programming Interface*) que basicamente são um conjunto de regras e protocolos que permite que diferentes softwares interajam entre si. Nesse caso, as APIs da plataforma QuimiCot Games permitem a integração do jogo com recursos específicos da plataforma, como autorização de jogadores, obtenção de questionários (quizzes) e o salvamento de informações gerais, que podem ser usadas caso a caso.

Com isso, o jogo recém criado estará integrado a uma ferramenta capaz de acompanhar e auxiliar na construção e na fixação de conhecimento, disponibilizando todos os benefícios pedagógicos já citados.

4.1.1 API de verificação da autorização do jogador

Autenticação é o processo de verificar a identidade de um usuário. Esse processo costuma usar informações de um usuário já logado na plataforma, como nome de usuário e senha, para gerar um token, que basicamente é um código exclusivo de verificação que dura um tempo limitado.

Ao tratar de uma plataforma a autorização é geralmente feita na própria plataforma, como ocorre na QuimiCot Games, no momento do login, ou seja, no acesso do jogador na plataforma. Esse processo tem como produto o token acima mencionado, que é repassado para todos os serviços chamados pela plataforma, como no caso a execução dos jogos.

A QuimiCot Games envia esse token pelo cabeçalho http chamado "aluno"e, é preciso verificar no começo do jogo que esse token está válido para que o jogador possa acessar o jogo, processo esse conhecido como autorização. Para isso, será feita a primeira integração com a API da QuimiCot Games por meio do verbo HTTP GET na URL "https://apichemical.quimicotgames.com/aluno", bastando passar o cabeçalho "*authorization*"com o valor do token. Esta API responderá com sucesso caso esteja válido e assim não será tomada nenhuma ação, permitindo que o jogador prossiga e acesse o jogo. Caso a API retorne um erro, significa que o token está inválido e o jogo deverá ser encerrado, impedindo o prosseguimento de um usuário não autorizado.

```
1  IEnumerator ValidateAuthorizationToken(string tokenToBeValidated)
2  {
3      using (UnityWebRequest webRequest = UnityWebRequest.Get(QUIMICOT_URI + "/aluno"))
4      {
5          webRequest.SetRequestHeader("authorization", "Bearer " + tokenToBeValidated);
6          yield return webRequest.SendWebRequest();
7
8          if (webRequest.result != UnityWebRequest.Result.Success)
9          {
10             Application.Quit();
11         }
12     }
13 }
```

Código 46 – Script que valida token da plataforma QuimiCot Games

4.1.2 API para salvar informações genéricas

A segunda integração com a plataforma de jogos QuimiCot Games será com a API que salva informações genéricas. Essa integração é opcional e fica a cargo de cada caso salvar as que sejam pertinentes, de acordo com a necessidade. Nesse jogo, serão salvos o local e momento em que o jogador morreu, para que seja possível analisar locais com maior incidência de mortes por exemplo e futuramente alterar o mapa para uma experiência melhor, além de outras possibilidades.

Para realizar o salvamento, basta criar um objeto de acordo com o contrato da API e enviar uma requisição POST para a URL acima mencionada, do serviço da plataforma. Para isso, será criada novamente uma função que retorna um *IEnumerator* a fim de aguardar (por meio do método *WaitForSecondsRealtime*) durante um segundo neste caso, a requisição ser completada. No caso de passar esse tempo e não haver sido concluída, a requisição será cancelada. Por último, será utilizado o *UnityWebRequest* para efetuar a requisição, repassando o token pelo header "authorization".

```
1  public IEnumerator LogInfoOnBackend(Info info)
2  {
3      yield return new WaitForSecondsRealtime(1f);
4      string infoJson = JsonUtility.ToJson(info);
5
6
7      using (UnityWebRequest webRequest = UnityWebRequest.Post(QUIMICOT_URI + "/aluno/log", infoJson))
8      {
9          webRequest.uploadHandler = new UploadHandlerRaw(Encoding.UTF8.GetBytes(infoJson)) as UploadHandler
10         webRequest.SetRequestHeader("authorization","Bearer "+WebParamsProvider.GetInstance().GetToken());
11         webRequest.SetRequestHeader("Content-Type", "application/json");
12
13         yield return webRequest.SendWebRequest();
14     }
15 }
```

Código 47 – Script que loga informação na plataforma QuimiCot Games

Agora resta adicionar a chamada desta função no *PlayerController* que trata a morte do jogador pelo método *die*. Para isso, será passada para a API o tipo de informação como "morte"e informada a posição do jogador por meio do componente *RigidBody*, avaliando sua posição nos eixos X e Y.

```
void die(){
    Info deathInfo = new Info(
        WebParamsProvider.GetInstance().GetFaseClass(),
        "morte",
        $"[FASE {SceneManager.GetActiveScene().name}] jogador morreu na posição
            x: {rigidBody.position.x}, y: {rigidBody.position.y}",
        DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"),
        DateTime.Now.AddSeconds(1).ToString("yyyy-MM-dd HH:mm:ss"), "{}");

    StartCoroutine(InfoLogger.GetInstance().LogInfoOnBackend(deathInfo));

    animator.SetTrigger("hasDied");
    FindObjectOfType<PlayerMovementController>().StopMovement();
    FindObjectOfType<GameSessionController>().ResetGameLevel();
}
```

Código 48 – Exemplo de implementação de log de informações

Salvar informações genéricas pertinentes em um jogo é de extrema importância, pois contribui para um acompanhamento por meio da plataforma, e pode auxiliar em diversos cenários, auxiliando na identificação de problemas com o jogo propriamente dito, com lacunas de conhecimento do jogador e possibilitando a ação da paltaforma nesses casos.

4.1.3 APIs para solicitar, enviar resposta e receber o gabarito do Quiz

Como já foi comentado, os questionários ou *quizes* são uma importante ferramenta escolhida pela QuimiCot Games a fim de obter insumos dos jogadores para saber se estão aprendendo ou não a temática relacionada. Dessa forma, será feita a integração e apresentado o questionário sempre no final de cada nível.

Para isso, será criado um objeto *QuizManager* que conterá o script *QuizProvider*. Esse script receberá uma interface gráfica com a estrutura geral do questionário previamente criadio e apenas serão subtituidos os textos de acordo com o que a API retornar. Logo depois, será iniciada uma nova rotina que chamará o método *GetQuiz* a fim de realizar uma requisição a API da plataforma QuimiCot Games para obtenção do questionário.

```
private IEnumerator GetQuiz(Action<Quiz> onQuizCompleteAction)
{
    using (UnityWebRequest webRequest = UnityWebRequest.Get($"{QUIMICOT_URI}/aluno/turmas/fases/{
    "WebParamsProvider.GetInstance().GetFaseClass()}/quiz"))
    {
        webRequest.SetRequestHeader("Authorization", "Bearer " + WebParamsProvider.GetInstance().GetToken());
        webRequest.SetRequestHeader("Content-Type", "application/json");

        yield return webRequest.SendWebRequest();
        if (webRequest.result == UnityWebRequest.Result.Success)
        {
            Quiz quiz = JsonConvert.DeserializeObject<Quiz>(webRequest.downloadHandler.text);
            quizId = quiz.quiz_id;
            onQuizCompleteAction(quiz);
        }
    }
}
```

Código 49 – Função que obtém questionário da plataforma QuimiCot Games

Com os dados do questionário, o método acima irá converter a resposta em um objeto local chamado de *Quiz* e chamar a função callback *CreateQuizUI* que será responsável por popular a estrutura de UI pré-existente com esses dados.

```
private void CreateQuizUI(Quiz quiz)
{
    // enable Quiz UI
    quizUI.SetActive(true);

    // set question text
    quizQuestionText.text = quiz.pergunta;


    // create all answers buttons
    for (int i = 0; i < quiz.alternativas.Count; i++)
    {
        GameObject currentButton = Instantiate(quizAnswerButtonPrefab, quizUI.transform);
        var initialPosY = (quizUI.transform.position.y + 50);
        currentButton.transform.position = new Vector3(quizUI.transform.position.x, initialPosY-(i*60), 0);

        currentButton.name = quiz.alternativas[i].id;
        TextMeshProUGUI buttonText = currentButton.GetComponentInChildren<TextMeshProUGUI>();
        buttonText.text = quiz.alternativas[i].descricao;

        currentButton.GetComponent<Button>().onClick.AddListener(delegate { SendUserQuizAnswer(); });
        currentButton.GetComponent<Button>().enabled = true;
    }
}
```

Código 50 – Função atualizada de criação de questionário

Esta função inicialmente reativará a UI previamente criada, alterará o texto da pergunta para o texto da pergunta que foi recebida da QuimiCot Games e, por fim, criará todos os botões de resposta, adicionando um ouvinte ao evento de clique do botão, que disparará a função *SendUserQuizAnswer* responsável por enviar a resposta do usuário à plataforma.

```
private IEnumerator sendUserQuizAnswer(string answerId)
{
    QuizUserAnswer quizAnswer = new QuizUserAnswer(answerId, quizId,
    DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"));
    string json = JsonUtility.ToJson(quizAnswer);

    using (UnityWebRequest webRequest = UnityWebRequest.
Post($"{QUIMICOT_URI}/aluno/turmas/fases/{WebParamsProvider.GetInstance().GetFaseClass()}/quiz", json))
    {
        webRequest.uploadHandler = new UploadHandlerRaw(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(quizAnswer)));
        webRequest.SetRequestHeader("Authorization", "Bearer "+WebParamsProvider.GetInstance().GetToken());
        webRequest.SetRequestHeader("Content-Type", "application/json");

        yield return webRequest.SendWebRequest();
        if (webRequest.result == UnityWebRequest.Result.Success)
        {
            QuizAnswerSheet quizAnswerSheet =
                JsonConvert.DeserializeObject<QuizAnswerSheet(webRequest.downloadHandler.text);

            addAnswerSheetToQuiz(answerId, quizAnswerSheet);
        }
    }
}
```

Código 51 – Função que envia resposta do questionário à plataforma QuimiCot Games

Já a função *SendUserQuizAnswer* usará novamente o UnityWebRequest para enviar uma requisição POST à plataforma e, em caso de sucesso, receberá como resposta o gabarito com o resultado e as justificativas de cada alternativa. Nesse caso, será chamada a função privada *addAnswerSheetToQuiz* para adicionar o gabarito à tela.

```
private void addAnswerSheetToQuiz(string asnwerId, QuizAnswerSheet quizAnswerSheet)
{
    foreach (var quizJustifiedAnswer in quizAnswerSheet.alternativasJustificadas)
    {
        GameObject choiceButton = quizUI.transform.Find(quizJustifiedAnswer.id.ToString()).gameObject;
        choiceButton.GetComponent<Button>().enabled = false;

        TextMeshProUGUI buttonText = choiceButton.GetComponentInChildren<TextMeshProUGUI>();
        buttonText.text += ("\n" + quizJustifiedAnswer.justificativa);

        if (quizJustifiedAnswer.alt_correta == 1)
        {
            choiceButton.GetComponent<Image>().color = Color.green;
        }
        if (asnwerId == quizJustifiedAnswer.id.ToString() && quizJustifiedAnswer.alt_correta == 0)
        {
            choiceButton.GetComponent<Image>().color = Color.red;
        }
    }

    GameObject continueButton = Instantiate(quizContinueButtonPrefab, quizUI.transform);
    continueButton.GetComponent<Button>().onClick.AddListener(delegate { LevelExit.GetInstance().LoadNextLevel(); });
}
```

Código 52 – Função que adiciona resposta da plataforma QuimiCot Games ao questionário

Para adicionar o gabarito à tela, é necessário percorrer todos os botões do questionário, desabilitá-los para evitar que enviem a resposta novamente, acrescentar o texto da justificativa ao botão e mudar a cor da resposta do usuário para verde caso tenha sido correta ou vermelho caso incorreta.

4.2 Inserção do jogo na plataforma QuimiCot Games

Concluído o desenvolvimento do jogo, a etapa final é a sua disponibilização na plataforma. Para isso, foram necessários três passos: *build* do jogo, hospedagem e publicação na plataforma QuimiCot Games, que serão discutidos abaixo. Esses passos possibilitam tornar o jogo disponível na plataforma para que os estudantes possam acessá-lo e jogá-lo, além dos administradores e professores poderem colher dados para fins educativos. Isso envolve a disponibilização do jogo em um local onde os jogadores possam baixá-lo, jogá-lo online ou acessá-lo de alguma outra forma.

4.2.1 Build do jogo para a plataforma Web

Este processo envolve a compilação e preparação do jogo para que possa ser executado em uma plataforma de destino específica, seja Android, iOS, direto no computador, console ou mesmo pelo próprio navegador de internet.

Como o jogo em questão foi desenvolvido utilizando o motor Unity que é multiplataforma, ou seja, permite a partir de um mesmo código a geração de um executável em diversas plataformas e dispositivos, basta escolher a plataforma de destino a qual o jogo irá rodar. Atualmente a Quimicot Games está presente apenas em versão Web, e, portanto, a plataforma de destino será apenas a Web.

Para isso, o jogo é compilado utilizando o WebGL (Web Graphics Library) que é uma API do JavaScript para renderizar gráficos 3D e 2D dentro de um navegador web compatível sem o uso de *plug-ins* (MDN Web Docs, 2023).

Dessa forma, basta acessar o menu "File"(Arquivo) e selecionar "Build Settings"(Configurações de Build). Depois disso, deve-se selecionar todos o níveis criados e escolher a plataforma WebGL como destino. É possível, ainda, mexer em configurações adicionais como resolução, versões de tecnologias específicas e etc., porém não serão feitas nenhuma configuração adicional por questões de simplicidade.

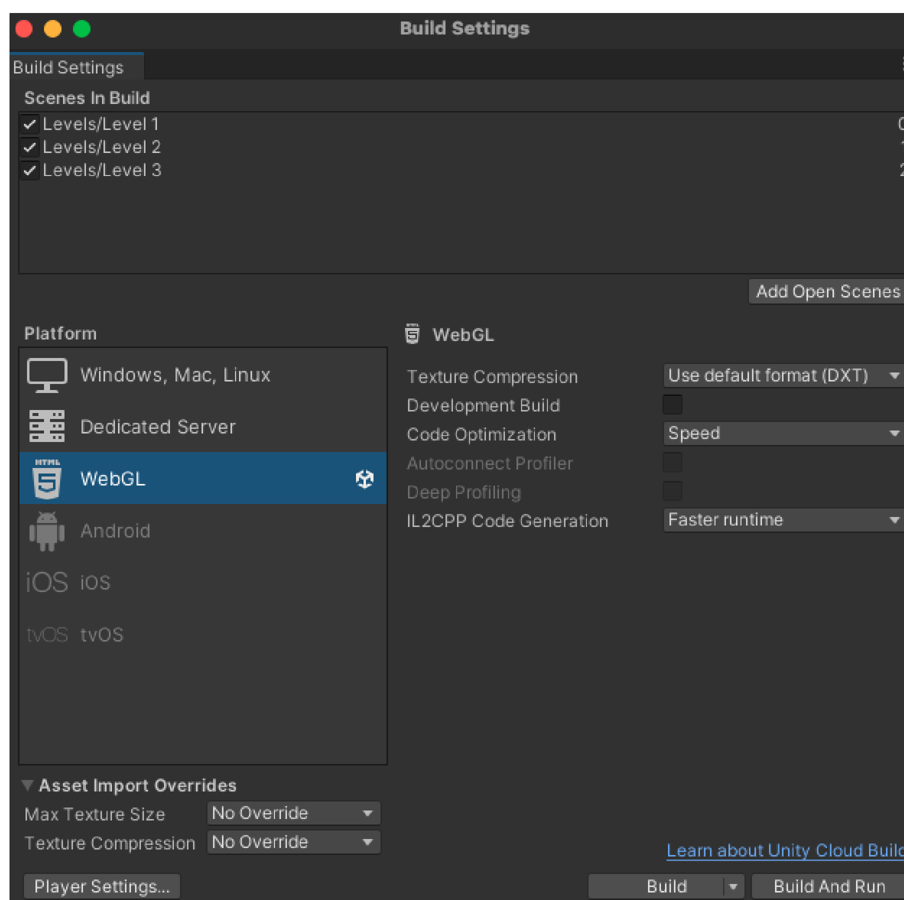

Figura 30 – *Configuração de build do jogo para WebGL*

Tudo configurado, agora é só clicar no botão “Build” (Construir) para iniciar o processo de compilação conforme Figura 30. Dependendo da plataforma de destino isso gerará o executável do jogo ou um pacote contento vários arquivos. No caso de WebGL, o Unity irá gerar um arquivo compactado contendo um arquivo principal HTML que irá

executar o jogo no navegador, além de outros recursos como JavaScript, CSS e arquivos de dados, conforme Figura 31.

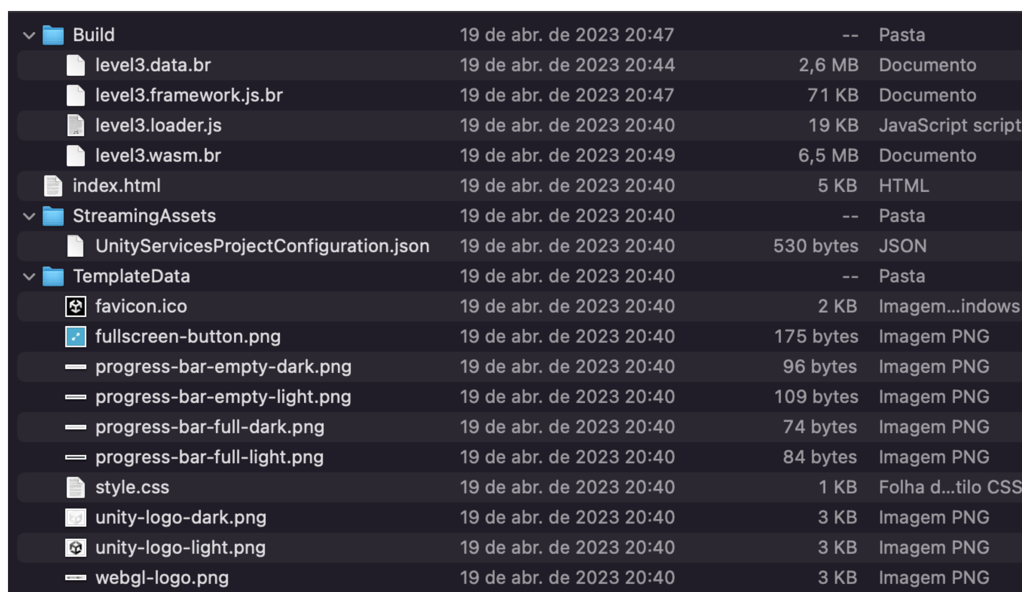

Figura 31 – *Arquivos gerados pelo build de um jogo feito em Unity para WebGL*

Portanto, ao invés de um único arquivo executável como em outras plataformas, os jogos em WebGL são executados com base em vários arquivos (HTML, JavaScript, CSS e recursos gráficos). Isso ocorre devido à natureza da tecnologia WebGL e à forma como os navegadores web funcionam, uma vez que serão carregados no navegador web e interpretados por ele para exibir o jogo via *Canva* no contexto de uma página web.

4.2.2 Publicação do jogo na plataforma de jogos educativos QuimiCot Games

A publicação de um jogo em uma plataforma é um processo crucial para tornar o jogo disponível para os jogadores. O processo varia muito dependendo da plataforma de destino, seja uma loja de aplicativos, uma plataforma de distribuição de jogos ou, como é o caso, uma plataforma de jogos educativos. Cada plataforma tem seu processo, algumas facilitando o passo a passo e outras nem tanto. Em geral, elas oferecem uma interface gráfica que solicita algumas informações sobre o jogo como nome, categoria em que se encontra etc., além do próprio executável do jogo em si, costumeiramente de forma compactada.

Atualmente, não existe uma maneira de realizar essa inserção diretamente na própria plataforma QuimiCot Games. Por isso, foi necessário inserir os arquivos manualmente no servidor em que a plataforma está hospedada, colocar os arquivos na pasta que contém os jogos já existentes e nomear a pasta com o nome do jogo, que será apresentado na plataforma. Dessa forma, ela automaticamente busca esses arquivos do servidor e o jogo está disponibilizado para os estudantes acessarem diretamente no site da QuimiCot Games e para os professores e administradores colherem as métricas dele. A Figura 32 apresenta

uma captura de tela da plataforma que mostra a fase disponível para ser jogada (final da página, botão 'Jogar') e as estatísticas do estudante logado para um elemento químico específico.

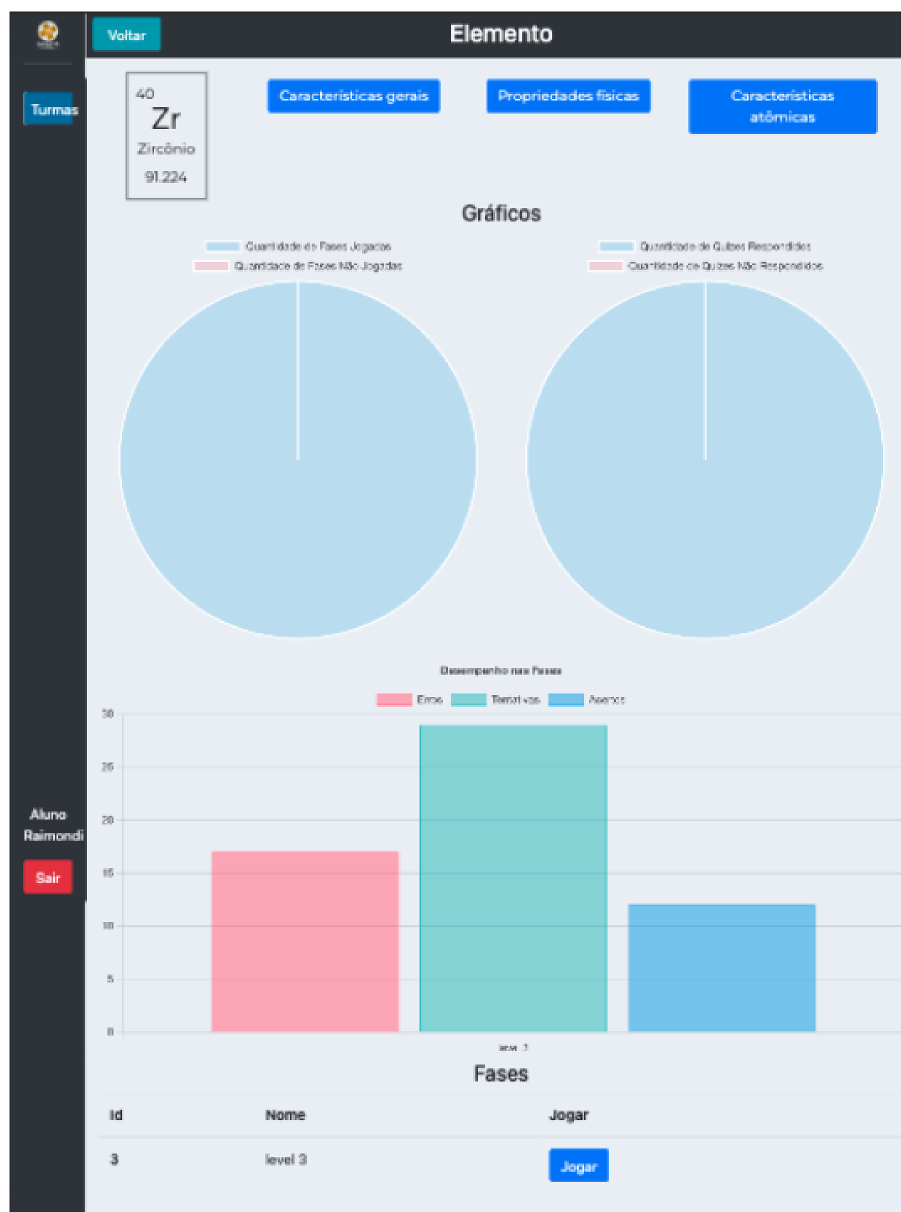

Figura 32 – *Captura de tela da plataforma QuimiCot Games com o jogo proposto integrado*

4.3 Desafios encontrados ao longo do trabalho

Neste capítulo, serão explorados alguns desafios que surgiram ao longo da pesquisa e desenvolvimento deste trabalho de conclusão de curso.

4.3.1 Entender o ecossistema de uma *engine*

Uma das maiores dificuldade encontrada no trabalho sem dúvidas foi entender o contexto geral de todo o ecossistema Unity até finalmente ser possível navegar e produzir

algo efetivamente.

Engines como o Unity são muito poderosas e oferecem vários recursos para facilitar a vida dos desenvolvedores. Todavia, para que existam esses recurso exige-se uma complexidade inseparável e, claro, várias *engines* trabalham para tornar isso cada vez mais intuitivo, porém é um processo extremamente complexo, especialmente para quem nunca teve contato com jogos ou com alguma *engine*.

No começo foi difícil entender até por onde começar e, nesse ponto a escolha de usar o Unity foi ótima, pois existem muitos tutoriais oferecidos pela própria empresa e pela comunidade, inclusive a maioria gratuitos. Após entender o básico do ecossistema, foi necessário seguir alguns tutoriais que ensinavam como criar um primeiro jogo, a fim de adquirir certa experiência. Assim, seguiu-se a criação de três jogos 2D com o Unity e que serviram como base para começar a desenvolver o jogo deste trabalho.

Assim que inicia-se o processo de criar o jogo sozinho, percebe-se vários pontos de dúvida. Algumas pesquisas no Google resolveram a maioria, diante do grande material oferecido pela comunidade. Entretanto, temas mais complexos como sistema de física levaram mais tempo para entender, por exemplo como esses recursos se relacionam e o que cada parametrização e opção oferecida faz, para assim conseguir criar comportamentos realistas e fluidos.

Em suma, houve muitos momentos em que nada fazia sentido e foram necessárias extensas pesquisas para conseguir sair dos problemas em questão. Contudo, conforme adquire-se experiência torna-se mais fácil navegar melhor no ecossistema da engine, começa a ser um processo mais dinâmico e o desenvolvimento flui melhor do que fazendo tudo manualmente, sem utilizar uma engine por exemplo. De forma análogo, adquirindo mais prática também já é possível identificar problemas parecidos e ser mais eficaz na identificação e resolução deles.

4.3.2 Aplicar princípios de padrões de arquitetura e design de software

Princípios de padrões de arquitetura e de design de software demonstraram ser difíceis de aplicar ao desenvolver jogos na Unity devido à natureza peculiar do desenvolvimento de jogos, especialmente vinculados à uma *engine*.

Ao tentar definir uma arquitetura previamente, depara-se com os conceitos oriundos da própria *engine* de jogos. Foram necessárias adaptações até que chegou um momento do trabalho que não foi possível continuar tentando seguir a arquitetura planejada. Nesse momento foi alterada a abordagem, decidiu-se por aplicar alguns princípios de design de software durante o próprio desenvolvimento, mas sem ter um desenho de arquitetura pré-montado. Essa abordagem não é a melhor e costuma gerar uma série de problemas como re-trabalhos. Grande parte disso deve-se a dois principais fatores: a inexperiência

do autor deste trabalho com o desenvolvimento de jogos e com o Unity, bem como nos próprios modos como a *engine* trabalha e sua natureza.

Nesse sentido, aplicar também padrões de design de software como SOLID – conjunto de cinco princípios de design de software que visam criar código mais flexível, extensível e fácil de manter (GAMMA et al., 2004) – foram dificultados pela natureza da *engine*, a qual possui conceitos interligados e componentes que possuem dependência entre si. Essa interdependência dificulta alcançar um baixo acoplamento e seguir o Princípio do Isolamento de Interface (ISP) do SOLID e, como também não oferecem nativamente suporte à Injeção de Dependência (DI), torna difícil aplicar princípio da Inversão de Dependência (DIP).

Com isso, durante o trabalho foi observado que sem um planejamento de arquitetura e boas práticas de design de software o projeto estava se tornando um código espaguete (Spaghetti code), que é uma frase pejorativa para um código-fonte não estruturado e de difícil manutenção (BMC, 2023). A melhor forma encontrada para estruturar e diminuir, mas não resolver, o acoplamento entre os componentes e scripts foi utilizar uma arquitetura orientada a eventos.

A arquitetura orientada a eventos (EDA) é um padrão de arquitetura moderno criado com base em serviços pequenos e sem acoplamento que publicam, consomem ou encaminham eventos. Um evento representa uma mudança de estado ou uma atualização (AWS, 2023). Um exemplo criado neste trabalho foi no caso de uma resposta de um questionário, dependendo da resposta é gerado um evento de resposta correta ou incorreta e cada consumidor interage com esses eventos e reagem de forma específica, inexistindo dependência entre eles.

Dessa forma, as definições de arquitetura e design de software que costumam ser feitas no desenvolvimento de softwares convencionais serão possivelmente adaptadas quando trata-se de desenvolvimento de jogos. Isso não quer dizer que seja impossível de serem alcançados, mas existem certas dificuldades oriundas de estar atreladas à arquitetura e conceitos da própria *engine* que devem ser lavadas em consideração.

4.3.3 Integração com serviços externos não é intuitiva

A integração com APIs externas é uma prática essencial para muitos jogos. Isso permite que os eles acessem serviços externos, como autenticação, obtenção e processamento de dados, entre outros.

A maneira mais comum e fácil de trabalhar com uma API é de forma síncrona, o que significa que o programa aguarda a conclusão dessa operação, ficando retido, antes de prosseguir para a próxima. Todavia, em jogos as chamadas para APIs externas são frequentemente assíncronas para evitar bloqueios no fluxo principal do jogo à medida

que permite ao programa continuar executando outras operações enquanto espera por uma resposta. Trabalhar de forma assíncrona, apesar de mais performático, é bem mais trabalhoso e depende muito do cenário para que se consiga aproveitar e realizar outras operações enquanto aquela está sendo processada.

No Unity, a chamada de APIs externas em geral são feitas usando a estrutura *IEnumerator* e que está muitas vezes associado ao modelo assíncrono baseado em co-rotinas (coroutines). No presente trabalho, algumas APIs da plataforma QuimiCot Games trabalharam bem com esse modelo, principalmente as que não esperam nenhum retorno, como as APIs de autenticação e envio de informações. Todavia, a maior parte das APIs da plataforma acima mencionada tem o funcionamento síncrono, não sendo possível realizar outras tarefas enquanto estiver aguardando seu retorno, como no caso da obtenção, processamento e demais funcionalidades dos questionários.

Nesse caso, como o Unity não oferece um modelo síncrono, foi necessário trabalhar com a estrutura assíncrona que é o *IEnumerator*, naturalmente mais complexa de lidar e usá-la de forma síncrona, bloqueando a execução aguardando o retorno da API. Dessa forma, foi necessário trabalhar com uma estrutura sem aproveitar os benefícios dela, mas sendo necessário passar por suas complexidades, o que tornou a integração bem mais complexa e morosa.

Fora a isso, a estrutura *IEnumerator* não permite o retorno de algum objeto, que seria extremamente útil em cenários onde espera-se um retorno, como o da obtenção questionário, processamento de resposta e obtenção do gabarito. Nesses casos, foi preciso trabalhar com *callbacks functions*, que são funções passada como argumento para outra função e executada após a conclusão da primeira. Isso gerou uma série de funções aninhadas, uma chamando a outra e assim por diante, situação que tornou o desenvolvimento mais complexo e extremamente contra intuitivo.

Por fim, percebe-se que o Unity não está muito desenvolvido nesse quesito. A estrutura de integração com APIs não é muito flexível e não oferece as facilidades que costuma-se encontrar em bibliotecas para desenvolvimento de aplicativos por exemplo, cenário em que o consumo de APIs é um tema central. Em suma, foi possível realizar a integração, mas ela foi mais rudimentar e engessada no único modelo oferecido pela *engine*.

4.3.4 Bugs na própria *engine*

Quando surgem problemas técnicos, a capacidade de identificar e solucioná-los de forma eficaz é crucial. Isso requer habilidades de resolução de problemas e, às vezes, pesquisa extensiva. Quanto mais iniciante se é, mais difícil esse processo se torna, conforme comentado no primeiro tópico de desafio encontrado no trabalho.

Entretanto, o problema agrava-se muito quando é um bug não em algo que está sendo desenvolvido, mas no código da própria *engine*. Como a maioria das *engines* são de códigos fechados, não é possível ter acesso a ele para validar a lógica e então o único recurso é pesquisar e, caso ninguém tenha reportado isso, abrir um *ticket* solicitando o suporte da empresa.

No trabalho em questão ocorreu uma situação dessas, na qual foram despendidos dias tentando compreender onde estava o problema no código desenvolvido, até encontrar nos tickets abertos para a própria empresa um reporte desse *bug* e lá conter um aviso que seria corrigido em alguma futura versão. Nesse caso foi necessário uma solução alternativa, buscando outra estratégia para alcançar a mesma funcionalidade.

Para evitar que isso ocorra, os desenvolvedores de *engines* implementam processos robustos de teste, tanto automatizados quanto manuais. E ainda no caso de ocorrerem, documentam os *bugs* e contam com o reporte da comunidade para auxiliar na identificação e futura correção. Outro ponto a ressaltar é que isso é comum utilizando qualquer tecnologia fornecida por terceiros e há possibilidade também, de ser inclusive maior, da existência de *bugs* no caso de serem criadas manualmente pelo próprio desenvolvedor do jogo.

Portanto, *bugs* na própria *engine* apesar de extremamente incomuns, são cenários possíveis e vale ter esse ponto de atenção. A existência de *bugs* em uma *engine* é comum e a resolução é um esforço contínuo das empresas e da comunidade, visando melhorar a estabilidade e a confiabilidade para os desenvolvedores que a utilizam.

4.4 Vantagens de utilizar uma *engine* de jogos

4.4.1 Ferramentas e funcionalidades pré-construídas

A maior vantagem de utilizar uma *engine* como o Unity para a criação de jogos é a variedade de ferramentas e funcionalidades pré-construídas, ou seja, que já vem embutidas na *engine* e que já foram testadas, refinadas e possuem uma qualidade e segurança bem estabelecidas. Seguem algumas das principais funcionalidades que costumam ser oferecidas:

- Editor Gráfico de Cenas: editor gráfico que permite aos desenvolvedores criar e organizar cenas do jogo de forma visual. Isso inclui a disposição de objetos, câmeras e efeitos visuais.
- Motor de Renderização: motores de renderização cuidam da renderização de gráficos, texturas e efeitos visuais. Isso inclui suporte para gráficos 2D e 3D, sombreamento, pós-processamento e renderização em tempo real.

- Física Integrada: um sistema de física integrada permite simular o comportamento realista de objetos no jogo, como colisões, gravidade e dinâmica de fluidos.
- Iluminação: recursos de iluminação são desde fontes de luz, luzes direcionais, locais, pontuais e de área, capazes de produzir sombras realistas em tempo real.
- Gerenciamento de *assets*: ferramentas para importar, organizar e gerenciar *assets* do jogo, como modelos 2D/3D, texturas, áudio e animações.
- Ferramentas de Animação: sistemas de animação permitem criar e controlar animações de personagens, objetos e demais elementos de um jogo.
- Áudio e Música: recursos para gerenciar áudio e música no jogo, incluindo a reprodução de efeitos sonoros, trilhas sonoras e mixagem de áudio.
- IA e Comportamento de NPCs: recursos para criar comportamentos de NPCs (Personagens Não-Jogáveis) e sistemas de IA, permitindo que os desenvolvedores criem personagens controlados pela inteligência artificial.
- Integração de Redes: muitas *engines* suportam a integração de redes, o que facilita a criação de jogos multijogador online.

Claro que é possível criar tudo isso manualmente, entretanto isso demandaria uma quantidade de tempo e recursos adicionais consideráveis. Ademais, a qualidade das ferramentas oferecidas pela *engine* são consistentes, já foram testadas e usadas extensivamente por vários usuários e empresas, ajudando a garantir a qualidade final do produto e a estabilidade dessas funcionalidades, o que reduz significativamente as chances de erros e problemas.

Também é importante reconhecer que mesmo assim a criação manual de funcionalidades é necessária para atender a requisitos exclusivos do jogo. No entanto, a maioria dos desenvolvedores aproveita as ferramentas e funcionalidades genéricas pré-construídas para acelerar o desenvolvimento e focar na criação apenas das que são específicas para o seu jogo.

4.4.2 Loja de assets

Os *assets* nada mais são do que elementos individuais do jogo que podem ser baixados e utilizados na própria *engine*, como arte, áudio, IA ou até scripts que executam certa funcionalidade específica. Esses *assets* podem ser gratuitos ou pagos, a depender do caso.

Todavia, principalmente no que tange à arte e áudio, é preciso levar em consideração que por serem abertas à comunidade algumas delas provavelmente terão sido

utilizadas por outros jogos, o que pode limitar a originalidade do jogo na visão de quem está jogando. Por isso, é importante ressaltar que, caso haja necessidade, esses *assets* podem ser personalizados até certo grau por quem os utilizará, o que auxilia a encaixar nas necessidades do projeto e inclusive a diferenciar um pouco do pacote original.

Além de acelerar o processo de desenvolvimento, é muito útil para equipes limitadas, como o caso em questão da plataforma QuimiCot Games, que não conta com artistas, músicos e possuem poucos desenvolvedores. No presente trabalho foi utilizada a própria loja do Unity, conforme ilustrado na Figura 33, para adquirir a arte do personagem e do cenário, além do áudio para os efeitos sonoros por exemplo.

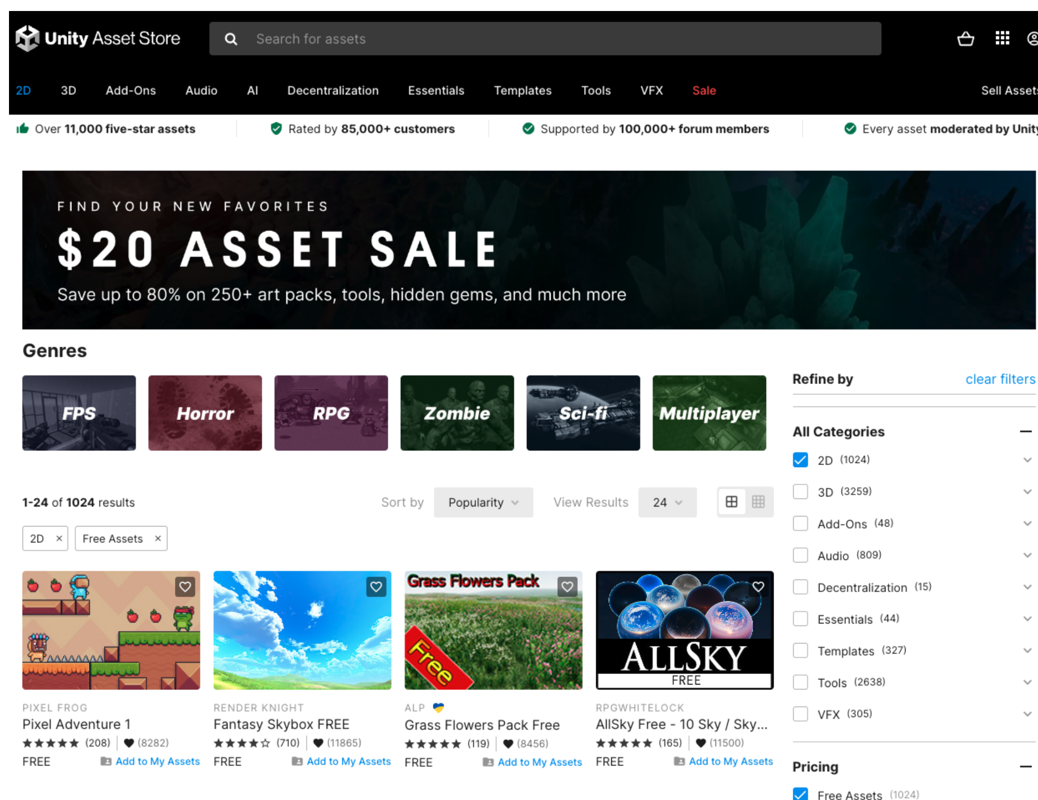

Figura 33 – *Loja de Assets do Unity*

4.4.3 Rápida prototipagem

Um protótipo é uma versão básica do sistema a ser construído, testado e então retrabalhado conforme necessário até que a partir de seu teste funcional o sistema ou produto completo possa ser desenvolvido. Dessa forma, foi possível construir os conceitos chaves, principais mecânicas e objetivos do jogo utilizando funcionalidades pré-construídas pela *engine*, tornando possível validar toda a ideia de forma fácil e rápida, antes de seguir com o desenvolvimento completo.

Isso oferece insumos para que, caso necessário, a rota seja recalculada no início do processo, alterando objetivos e afins. Caso em que otimiza o processo, evitando que isso

seja feito quando já existe toda uma complexidade envolvida, o que geraria um possível grande retrabalho e gastos desnecessários de tempo e recursos.

Utilizando as ferramentas pré-construídas, interfaces gráficas e demais facilidades oferecidas por uma engine de jogos como o Unity é possível ter um protótipo em questão de horas ou poucos dias. Neste trabalho foram necessários algumas semanas para criar todo o básico e validar a viabilidade do conceito do jogo.

4.4.4 Visual Scripting

Os scripts visuais permitem que os criadores desenvolvam mecânicas de jogo ou lógicas de interação usando um sistema visual baseado em gráficos ao invés de escrever linhas de código tradicional (Unity Technologies, 2023m), ou seja, é uma abordagem de programação que permite criar lógicas de software por meio de representações visuais que lembram grafos, utilizando caixas e setas que indicam o fluxo, ao invés de escrever código tradicionalmente textual, conforme Figura 34.

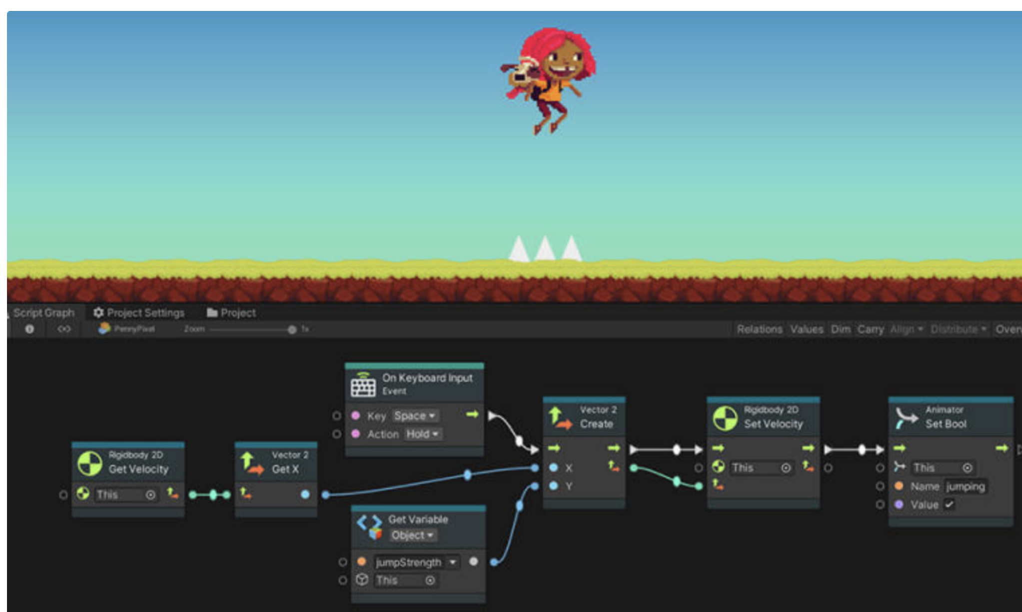

Figura 34 – *Visual Scripting no Unity*

O Visual Scripting é uma ferramenta de certa forma nova e não tão conhecida, mas muito poderosa para acelerar o desenvolvimento de jogos, como no caso de jogos educacionais. Essa ferramenta não foi utilizada neste trabalho em função do autor do trabalho já possuir conhecimentos de programação. Porém é uma grande vantagem para se ter no radar, especialmente para uma plataforma de jogos educativos como a QuimiCot Games, pois no caso em questão a equipe é reduzida e nem sempre será possível ter um programador experiente para desenvolver os jogos, possibilitando, assim que outras pessoas possam auxiliar nesse processo, ou seja, torna a criação de lógica de jogo mais acessível a uma ampla variedade de criadores, independentemente de sua experiência em programação tradicional.

4.4.5 Multiplataforma

Uma das maiores vantagens de utilizar uma engine de jogos, e a principal razão pelo destaque que vem obtendo no mercado, é a capacidade de ser multiplataforma. Essa característica torna possível que os desenvolvedores criem jogos e aplicativos uma vez e os distribuam em uma ampla variedade de plataformas, abrangendo desde dispositivos móveis, Web, até consoles de videogame e sistemas de realidade virtual. Claro que nem todas as plataformas e dispositivos estão disponíveis e cada *engine* oferece suporte para algumas em específico.

Existem diversas vantagens em ser multiplataforma, a primeira é a ampla cobertura de mercado, tornando possível atingir diferentes nichos, dispositivos, plataformas e, consequentemente, aumentando o público em potencial e permitindo que o jogo alcance audiências diversificadas e explorem novos mercados de maneira eficiente. Outra grande vantagem para os desenvolvedores e o time de engenharia é a eficiência no desenvolvimento, contando com uma base única de código para várias plataformas, não sendo necessário refazer e dar manutenção para vários códigos. Isso diminui muito a complexidade, economiza tempo de desenvolvimento e recursos, além de garantir que os jogadores tenham uma experiência consistente e semelhante, independentemente do dispositivo que estão usando.

Em resumo, a característica multiplataforma não apenas simplifica o processo de desenvolvimento, mas também potencializa a capacidade do jogo atingir outras plataformas de maneira simples. Mesmo que inicialmente não tenha-se a pretensão de publicá-lo em outras plataformas, é sem dúvidas um trunfo significativo para a eventualidade de surgi-la.

4.4.6 Comunidade e suporte

Uma comunidade ativa e o suporte facilitado são vantagens significativas de usar uma *engine* de jogos, especialmente no caso do Unity que é a plataforma de jogos mais usada no mercado, possuindo uma das maiores e mais engajadas comunidades.

Ao começar a estudar Unity por exemplo existe uma dificuldade inicial de entender os conceitos básicos e todo o ecossistema da *engine* conforme já mencionado. Todavia, existem diversos tutoriais, artigos, vídeos e recursos educacionais como treinamentos, cursos e etc. oferecidos tanto pela comunidade quanto pelos próprios desenvolvedores do Unity, o que facilitou muito a aprendizagem desses conceitos. Outra importante ferramenta nesse processo foram as documentações da própria *engine*, que explicam cada recurso e dão exemplos de uso da ferramenta e de desenvolvimento via código.

Além disso, com certeza surgirão problemas e dúvidas durante a criação do jogo e, como a comunidade é grande, provavelmente muitos desenvolvedores já passaram por

isso e fizeram questionamentos nos fóruns ou páginas de perguntas e respostas da comunidade deixando o registro dessas informações. Diversas dificuldades surgiram durante este trabalho e as respostas foram encontradas facilmente.

Por fim, as *engines* costumam valorizar o *feedback* da comunidade, já que são seus principais consumidores, e muitas vezes incorporam sugestões e realizam atualizações com o intuito de resolver problemas comuns e facilitar processos.

4.4.7 Ferramentas de depuração

As *engines* incluem consigo ferramentas de depuração que ajudam os desenvolvedores a encontrar e corrigir erros e problemas no jogo durante o desenvolvimento. Essa depuração pode ser feita de forma mais rudimentar usando o console ou de forma integrada com o próprio ecossistema da *engine*.

A depuração via console permite analisar erros e informações logadas pelo próprio serviço ou código, mas é a maneira mais comum e simples de análise. A grande vantagem encontra-se na ferramenta integrada que a maioria das *engines*, como o Unity, oferecem, permitindo aos desenvolvedores por exemplo pausar a execução do jogo em determinado ponto e avaliar os objetos da cena, seus status, variáveis, identificar erros ou problemas e avaliar o fluxo de execução do jogo no geral. Também é possível rastrear a execução do código em si, inspecionar variáveis e depurar linha a linha do script, como outras IDEs já fazem.

Em resumo, o depurador é uma ferramenta poderosa capaz de oferecer controle e visibilidade em tempo real sobre todos os elementos do jogo e sobre cada passo de sua execução, integrando tanto os objetos da cena, ambientes, juntamente com o próprio o código. Ele é um excelente aliado para resolver problemas, testar e garantir que o jogo funcione conforme o esperado, otimizando o tempo e facilitando a avaliação de erros.

4.4.8 Gerenciamento de recursos e otimização

O gerenciamento de recursos e otimização de um jogo é uma etapa fundamental para garantir que os jogadores tenham uma experiência satisfatória e positiva. Fora que permitem a democratização do jogo para diferentes plataformas e hardwares menos ou mais avançados.

Várias *engines* oferecem soluções semelhantes, no caso do Unity existe o *Profiler* que é uma ferramenta integrada de análise de desempenho que permite aos desenvolvedores identificarem e solucionarem problemas de desempenho em seus jogos. Ele oferece informações detalhadas sobre como os recursos utilizados pelo jogo como CPU, GPU, memória, entre outros.

Com o uso desse tipo de ferramenta é possível identificar possíveis gargalos em pontos específicos do jogo onde a execução é mais lenta, o que pode ser gerado, por exemplo, por funções de script que consomem muita CPU, renderização intensiva na GPU, problemas de memória, etc. Ademais, pode ser usado também para avaliar o desempenho de diferentes cenas, configurações ou plataformas, facilitando a otimização para diferentes casos de uso.

Tudo isso contribui para o uso eficiente dos recursos da máquina em que o jogador está rodando e impacta diretamente na experiência, retenção e satisfação do jogador, sem mencionar na competitividade e possibilidade de atingir mais usuários com diferentes características de hardware.

4.4.9 Gráficos e efeitos visuais de alta qualidade

O Unity, assim como outras *engines* de jogos, oferece uma ampla gama de recursos gráficos avançados, permitindo que os desenvolvedores alcancem maior qualidade gráfica em seus jogos, tornando-os mais visualmente atraentes.

Alguns desses recursos são os efeitos de pós-processamento, como *tonemapping*, *antialiasing*, profundidade de campo e correção de cores, a possibilidade de usar texturas de alta resolução, suporte a renderização HDR (High Dynamic Range), que oferece uma ampla gama de cores e contraste para visuais mais realistas, a criação de partículas e efeitos visuais complexos, como fumaça, fogo, explosões, fluidos, etc., além do uso de iluminação e sombras avançadas, contribuindo em um impacto grande para o visual do jogo.

Tudo isso pode ser usado para aprimorar a qualidade gráfica, visual e aumentar os detalhes do jogo em si, tornando a experiência mais satisfatória e imersiva.

4.4.10 Fácil adaptação à diferentes controles

A adaptação à diferentes tipos de controles é uma dificuldade enfrentada por todo desenvolvedor de jogos, especialmente os que trabalham com jogo multiplataforma diante da necessidade de mapear e se adaptar à diferentes controles. Por isso, as *engines* criaram soluções que buscam padronizar e facilitar essa integração.

Conforme demonstrado no capítulo de desenvolvimento, o Unity é conhecido por sua flexibilidade e facilidade de adaptação a diferentes tipos de controles, especialmente diante da funcionalidade recém lançada que foi usada no presente trabalho. A forma como o Unity trabalha é bem interessante pra isso, abstraindo em eventos as ações de *input* do usuário e possibilitando adicionar controles que executem essa mesma ação, necessitando apenas de um mapeamento do botão que executa cada uma.

Isso representa uma grande vantagem, como no jogo desenvolvido para este trabalho, à medida que facilita o processo. Um bom exemplo é que muitos jogadores de PC dão preferência a opção de jogar pelo controle tradicional de console ao invés do mouse e teclado. No geral também é um grande diferencial para o desenvolvimento de jogos em várias plataformas, incluindo PC, consoles, dispositivos móveis e até mesmo realidade virtual ou aumentada.

4.4.11 Plano gratuito para jogos educativos

Algumas plataformas oferecem um plano gratuito, como no caso do Unity o *Unity Personal*, para indivíduos, pequenas empresas de jogos com baixa renda anual e para projetos de ensino e educação como a plataforma QuimiCot Games de jogos educativos.

Vale ressaltar também que esse plano não é válido para apenas um determinado período de tempo e depois torna-se pago. Na verdade enquanto a pessoa, empresa ou instituição estiver nos critérios de elegibilidade poderá usar ele durante um tempo indefinido. Inclusive é permitido a monetização desde que não ultrapasse um certo limite de renda anual, caso em que o jogo alcançou um certo nível de sucesso e possuem renda consideradas suficiente para começar a entrar no plano pago.

Essa é uma enorme vantagem, pois possibilita acesso gratuito à uma *engine* poderosa. O que possibilita e estimula o aprendizado, testes, criação de projetos pessoais e educacionais, além de experimentação para avaliar os benefícios da *engine*.

O *Unity Personal* por exemplo é uma escolha popular para projetos educacionais em escolas e instituições de ensino, permitindo que estudantes e professores utilizem uma *engine* poderosa para criar conteúdo educacional. Para projetos educacionais como a QuimiCot Games funciona muito bem, pois em geral não existe um capital muito grande de investimento e mesmo assim é possível colher todos os benefícios de utilizar uma poderosa *engine* de jogos.

Em suma, a existência de um plano gratuito é uma vantagem significativa, pois democratiza o acesso à tecnologia de desenvolvimento de jogos. Isso ajuda a inspirar a próxima geração de desenvolvedores, auxilia pequenas empresas de jogos e plataformas de ensino e educação.

4.5 Desvantagens de utilizar uma *engine* de jogos

Embora o uso de uma *engine* de jogos, como o Unity, ofereça muitas vantagens, não é isento de desvantagens. Interpretar se tais problemas possuem relevância devem ser feitos caso a caso, levando em consideração os cenários e especificidades do próprio projeto. Nesta seção serão feitas algumas considerações sobre as desvantagens e alguns

pontos de atenção ao usar uma *engine* de jogos.

4.5.1 Dependência com a própria *engine*

A escolha de trabalhar com a uma *engine* de jogos pode gerar uma alta dependência do jogo com os sistemas, estruturas e recursos fornecidos pela *engine* em questão. Situação em que todo o desenvolvimento estará fortemente atrelado à forma como a *engine* trabalha. Essa dependência pode tornar inviável a migração ou portabilidade do jogo para outra *engine*.

Existem maneiras de mitigar isso, como boas práticas de arquitetura de software, isolando camadas de forma a torná-las independente da tecnologia em questão. Vale ressaltar que nem sempre isso será um problema, e aqui torna-se extremamente importante a análise prévia para uma escolha adequada da *engine*, minimizando esses futuros transtornos.

4.5.2 Flexibilidade limitada

Engines de jogos servem extremamente bem para a vários dos cenários, porém para casos extremamente específicos é possível que seja necessário customizar algo para o caso de uso em questão e torne-se inviável ou perca o sentido usar a *engine*, uma vez que algumas partes do código central da *engine* podem ser inacessíveis, o que limita a capacidade de personalização.

As *engines* também incluem sistemas e mecânicas de jogo como física, colisões e etc. que podem limitar a possibilidade de criar mecânicas completamente diferentes. A maioria das *engines* oferecem um alto grau de flexibilidade e personalização, mas em alguns casos isso pode não ser suficiente.

Vale ressaltar que essa desvantagem depende muito do caso em questão exigir tal flexibilidade, pois nos cenários comuns essas funcionalidades podem ser obtidas através da personalização oferecida pela *engine*, criação de scripts personalizados que executem essa lógica, etc. Em todo caso, é algo que desenvolvedores devem considerar ao escolher a melhor ferramenta para criar seus jogos.

4.5.3 Desempenho

As *engines* oferecem uma série de vantagens como produtividade e facilidade de desenvolvimento, mas essas mesmas vantagens também podem impor uma sobrecarga de desempenho. Isso pode ocorrer por vários fatores e costumam estar atrelados à própria natureza da *engine*.

Um dos principais é o fato da natureza universal e multiplataforma de algumas *engines*, ou seja, são projetadas para suportar uma ampla variedade de dispositivos e plataformas. Isso faz com que não sejam tão eficientes quanto construir algo nativo, cenário em que seria possível extrair e otimizar ao máximo cada caso em específico.

Além disso, pode haver uma sobrecarga em função da complexidade oriunda de todas essas funcionalidades pré existentes e possibilidade de personalização que veem junto com a própria *engine* como se fosse um grande pacote. Então para casos extremamente simples e que necessitem de um desempenho muito bom, é possível que encontrem maior dificuldade usando uma *engine*, caso em que talvez seja necessário um trabalho mais profundo para alcançar esse desempenho.

Um ponto interessante é que as próprias *engines* estão cientes dessa sobrecarga e que geralmente vem incorporadas a elas e começaram a quebrar melhor suas funcionalidades em pacotes menores, oferecendo uma solução básica mais minimalistas e simples, permitindo que os pacotes isolados sejam adicionados posteriormente de acordo com a necessidade do desenvolvedor, melhorado assim o presente problema.

4.5.4 Multiplataforma

Na sessão anterior evidenciou-se os benefícios de *engines* como o Unity serem multiplataforma, ou seja, possibilitar desenvolver uma vez e ter a capacidade de publicá-lo em diversas plataformas. Entretanto, existem também algumas desvantagens relacionadas a este mesmo tópico.

O primeiro seria a otimização. O desempenho pode variar muito entre plataformas, o que depende tanto do desenvolvimento do jogo quanto da própria *engine* na forma como realizam essa transformação. Otimizar um jogo para funcionar suavemente em todas elas pode ser desafiador e possivelmente dependerá de uma série de ajustes.

Ademais, à medida que faz-se um jogo multiplataforma, ele perde a possibilidade de utilizar facilmente os recursos específicos de cada uma delas. Essas restrições podem impor limitações e não permitir que se aproveite ao máximo as funcionalidades que cada plataforma oferece. De toda forma não é impossível, a Unity por exemplo oferece mecanismos que permitem a personalização e adaptação do código para atender às características exclusivas de diferentes plataformas, mas é necessário fazer isso manualmente para cada uma e não é um processo tão simples.

4.5.5 Curva de aprendizagem

A curva de aprendizado ao utilizar uma *engine* de jogos se refere ao tempo e ao esforço que os desenvolvedores precisam investir para se familiarizar com a *engine* e aprender a utilizá-la de forma eficaz.

A curva de aprendizado pode variar consideravelmente, dependendo da complexidade da *engine*, das habilidades prévias dos desenvolvedores e da natureza do projeto. Alguns fatores podem contribuir para melhorar esse processo como a existência de uma boa documentação, tutoriais e treinamos de qualidade, fatores que variam entre as *engines*.

Em geral essa curva é mais íngreme quando se trata de *engines* de jogos. É uma parte inevitável do processo, uma vez que elas possuem seu próprio ecossistema e quem ainda não teve contato levará certo tempo para se ambientar. Todavia, apesar do início ser mais lento, as funcionalidades pré-existentes e demais facilidades oriundas delas costumam compensar o investimento inicial. À medida que essa fase inicial é passada e os desenvolvedores tornam-se mais proficientes na utilização da *engine*, eles conseguem aproveitar melhor suas funcionalidades e o restante do processo tende a ser bem mais simples e rápido do que sem usá-la.

4.5.6 *Scripting* limitado à algumas linguagens de programação

Por fim, uma desvantagem que serve mais como ponto de atenção é que cada *engine* oferece suporte à um conjunto limitado de linguagens de programação para desenvolver scripts.

Em geral isso pode ser relevante quando é preciso levar em consideração a experiência prévia da equipe de desenvolvimento. Pode ser que a linguagem que possuem maior domínio não esteja disponível em determinada *engine* e seja necessário um tempo ainda maior para aprendê-la em conjunto com a *engine*. Dependendo da disponibilidade de recursos e prazo talvez não seja possível arcar com esse tempo adicional.

5 Conclusão

Neste capítulo, serão apresentadas as principais conclusões obtidas a partir do desenvolvimento e pesquisa referentes ao presente trabalho, que teve como objetivo avaliar o uso de uma *engine* como ferramenta para a criação de um jogo digital educacional, as vantagens, desvantagens e desafios relacionados, bem como a forma de integrar com uma plataforma de acompanhamento de aprendizagem. Além disso, serão discutidas algumas limitações do estudo e sugestões para trabalhos futuros.

5.1 Principais contribuições

O trabalho em questão introduziu alguns conceitos fundamentais de jogos eletrônicos educacionais, *engine* de jogos, entre outros, a fim de deixar claro a relevância, os temas centrais que seriam posteriormente trabalhados e evidenciar que a criação de jogos educativos possui um grade potencial à medida que é uma forma lúdica usada para estimular e promover o aprendizado. Seguiu-se de uma revisão bibliográfica dos conceitos supracitados e trabalhos relacionados, para que então fosse possível utilizar o que já foi analisado por outros trabalhos e ter uma base sólida para a pesquisa e início do desenvolvimento propriamente dito.

Diante disso, com esta monografia compreendeu-se as principais etapas envolvidas na criação de um jogo, desde as funcionalidades gerais até as mais específicas. Foram destrinchados todos processos, as principais mecânicas e como integrá-lo com uma plataforma de acompanhamento de aprendizagem em jogos, tornando possível avaliar as vantagens e desvantagens de realizar tudo isso utilizando uma *engine* como o Unity.

Em última análise, percebe-se que a escolha de usar uma *engine* para desenvolver um jogo depende das necessidades do projeto, da experiência dos desenvolvedores e dos recursos disponíveis e é importante estar ciente das vantagens e possíveis limitações ou desafios associados ao seu uso. Porém, tornou-se claro que mesmo com as dificuldades, especialmente no começo, ainda é uma boa escolha em virtude das vantagens oferecidas.

Para os cenários em que se beneficiem das vantagens mencionadas e nas quais as desvantagens e dificuldades não sejam superiores à elas, as *engines* mostraram ser poderosas aliadas para acelerar e simplificar muitos dos aspectos técnicos durante o desenvolvimento e subsequente publicação de um jogo.

Além disso, demostrou-se o modo de realizar a integração de jogos com plataformas educacionais existentes, mostrando-se necessário além do conhecimento técnico, constante cooperação entre desenvolvedores e administradores de sistemas. Enfatizou-se, também,

a importância dessas integrações para tornar o jogo efetivo no processo educacional.

Portanto, o conhecimento dos principais processos envolvidos na criação de um jogo, as vantagens e desvantagens do uso de uma *engine* e da maneira de realizar a integração com uma plataforma de acompanhamento de aprendizagem em jogos educacionais como a QuimiCot Games, visam auxiliar a comunidade e facilitar que outros profissionais da área possam fazer uma análise prévia de forma consciente e objetiva para escolher a melhor ferramenta para este cenário. Tudo que foi trabalhado aqui reforça o vasto potencial dos jogos educacionais, de uma plataforma como a QuimiCot Games e da importância da pesquisa contínua no campo, a fim de tornar o aprendizado mais envolvente e eficaz por meio de ferramentas como os jogos educativos.

5.2 Limitações

É imprescindível abordar as limitações deste trabalho para fornecer uma visão equilibrada e realista do estudo. Dito isso, primeiramente é importante reconhecer que o escopo aqui analisado é limitado, o que significa que nem todos os cenários e aspectos possíveis foram abordados. Isso ocorre devido a restrições de tempo, uma vez que tratar todos os cenários seria inviável e talvez impossível neste trabalho. Por isso, podem haver alguns aspectos que não tenham sido abordados ou considerados neste estudo.

De forma análoga, os resultados observados, apesar de tentarem ser gerais, podem não caber para todas as situações e contextos. Nesse sentido, as conclusões aqui obtidas talvez se apliquem ao contexto específico do presente estudo, ou a um desenvolvimento mais superficial de um jogo como aqui feito, podendo não se estender à outras situações.

Quanto aos passos para a criação de um jogo, foram abordados de forma superficial e utilizando exemplos do jogo em questão. Isso implica que alguns passos adicionais podem ser necessários, bem como aspectos mais profundos de alguns deles podem não ter sido cobertos e talvez necessitem de uma análise mais detalhada.

Já no que tange às vantagens e desvantagens de utilizar uma *engine* de jogos foram colhidas a partir de uma escolha de *engine* em específico, no caso o Unity. Apesar de ter sido feito um esforço para generalizar os casos, vale ressaltar que depende muito dos recursos e formas como cada *engine* os oferece, sendo possível obter diferenças em cada uma. Ademais, a experiência dos desenvolvedores com determinada tecnologia pode variar muito e tornar amplamente diferentes as escolhas e resultados aqui obtidos, tornando-se importante avaliar esses cenários no contexto particular de cada caso.

Por fim, a integração de um jogo com uma plataforma foi avaliada usando a QuimiCot Games e, portanto, outras plataformas podem oferecer desafios técnicos e de compatibilidade diferentes e que não foram totalmente abordados no estudo.

5.3 Trabalhos futuros

Este estudo avaliou o uso da *engine* Unity como uma ferramenta para a criação de jogos educacionais e sua integração com plataforma de acompanhamento de aprendizagem em jogos educacionais. No entanto, considerando as várias áreas associadas à educação e a tecnologia, existem diversas oportunidades e tópicos possíveis de pesquisas futuras.

Uma delas seria analisar de forma mais específica e aprofundada as principais *engines* de jogos do mercado, a fim de obter uma análise comparativa dos benefícios, desvantagens e situações de uso recomendadas pra cada um. Ademais, no que tange ao jogo, poderia ser avaliado o aprimoramento da experiência do usuário, considerando aprofundar em maneiras de tornar os jogos mais interessantes e atraentes aos estudantes, para que assim seja ampliado o alcance ao público alvo e a possibilidade de impactar no aprendizado como um todo.

Outro tema promissor e relacionado seria o de analisar técnicas mais efetivas para integrar nos jogos a fim de rastrear, analisar, compreender e monitorar o progresso do aluno, aumentando novamente a capacidade de aprendizado obtida. Poderiam também ser estudadas técnicas e mecânicas de integração que estejam mais próximas da jogabilidade, sendo possível colher esse insumos de artigos, jogos educacionais existentes ou até mesmo por coleta de *feedback* de alunos, professores e jogadores num geral.

Além disso, algumas pesquisas futuras podem abordar o uso de algoritmos de aprendizado de máquina e IA para personalizar a experiência de jogo, tornando-os mais dinâmicos e capazes de se adequar às necessidades individuais dos estudantes, levando em consideração estilos de aprendizagem, níveis de habilidade e preferências específicas por exemplo.

Outro aspecto importante é a exploração da acessibilidade e da inclusão em jogos educacionais. Pesquisas futuras podem se concentrar em tornar os jogos mais acessíveis para alunos com deficiências, como implementar recursos de áudio e legendas para alunos com deficiência auditiva ou até mesmo designs adaptados para alunos com deficiências motoras por exemplo.

Em suma, pesquisas futuras pode aprofundar e expandir as descobertas deste estudo, explorando várias abordagens relacionadas conforme algumas acima mencionadas. Essas novas pesquisas contribuirão para a contínua evolução e eficácia dos jogos educacionais como uma ferramenta valiosa no campo da educação.

Referências

ALVES, G. B. Estudo comparativo entre engines de desenvolvimento de jogos 2d. 2020. Citado 2 vezes nas páginas 21 e 28.

ALVES, L.; BIANCHIN, M. A. O jogo como recurso de aprendizagem. **Revista Psicopedagogia**, Associacao Brasileira de Psicopedagogia, v. 27, n. 83, p. 282–287, 2010. Citado na página 15.

ARAúJO, R.; NUNES, I.; REZENDE, H. Concepção de um jogo digital educativo usando design participativo para ensino contextualizado da tabela periódica. In: **Anais dos Workshops do Congresso Brasileiro de Informática na Educação**. [S.l.: s.n.], 2019. p. 524–533. ISSN 2316-8889. Citado na página 33.

AWS. **O que é a EDA (arquitetura orientada a eventos)?** 2023. Disponível em: <https://aws.amazon.com/pt/what-is/eda/>. Acesso em: 09 de novembro 2023. Citado na página 83.

BARBOSA, E. F.; MOURA, D. G. de. Metodologias ativas de aprendizagem na educação profissional e tecnológica. **Boletim Técnico do Senac**, v. 39, n. 2, p. 48–67, 2013. Citado na página 14.

BARROS, G. C.; SOUSA, J. K. C.; VIANA, D. Jornada química genial: um jogo sério para o ensino da tabela periódica e seus elementos. In: SBC. **Anais do XXXIII Simpósio Brasileiro de Informática na Educação**. [S.l.], 2022. p. 473–484. Citado na página 29.

BERTO, G. B. **Utilizando o Unity para desenvolvimento de jogos 2D**. 2017. Citado na página 29.

BMC. **Spaghetti code**. 2023. Disponível em: <https://www.bmc.com/blogs/spaghetti-code>. Acesso em: 09 de novembro 2023. Citado na página 83.

CAVALCANTE, C. H. L.; PEREIRA, M. L. A. **Comparativo entre Game Engines como Etapa Inicial para o Desenvolvimento de um Jogo de Educação Financeira**. [S.l.]: Jun, 2018. Citado na página 28.

CHRISTOPOULOU, E.; XINOGALOS, S. Overview and comparative analysis of game engines for desktop and mobile devices. 2017. Citado 2 vezes nas páginas 22 e 29.

CISCATO, C. A. M.; PEREIRA, L. F.; CHEMELLO, E. **Química 1: Química Geral**. São Paulo: Moderna, 2015. v. 1. Citado na página 32.

DAIREL, J. G. de M.; TUPINAMBÁ, R. C.; SILVA, Y. G. P.; ARAÚJO, R. D. Em direção a um ecossistema de software para apoio ao ensino de química por meio de jogos digitais. In: SBC. **Anais Estendidos do XX Simpósio Brasileiro de Jogos e Entretenimento Digital**. [S.l.], 2021. p. 689–692. Citado 2 vezes nas páginas 15 e 30.

FILHO, E. B.; SANTOS, C. G. P. dos; CAVAGIS, A. D. M.; BENEDETTI, L. P. dos S. Desenvolvimento e aplicação de um jogo virtual no ensino de química. **Informática na educação: teoria & prática**, v. 22, n. 3 Set/Dez, 2019. Citado na página 29.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Padrões de Projeto–Soluções Reutilizáveis de Software Orientado a Objetos, 2004, Ed**. [S.l.]: Bookman–Porto Alegre, 2004. Citado na página 83.

IUPAC. **Tabela Periódica de Elementos**. 2023. International Union of Pure and Applied Chemistry. Disponível em <https://iupac.org/what-we-do/periodic-table-of-elements/>. Citado 2 vezes nas páginas 6 e 33.

Jovem Pan. **Novos tempos da educação abrem espaço para metodologias inovadoras de ensino**. 2023. Disponível em: <www.abrafi.org.br/index.php/site/noticias/ver/4349>. Acesso em: 13 de novembro 2023. Citado na página 14.

JUNIOR, H.; MENEZES, C. Modelo para um framework computacional para avaliação formativa da aprendizagem em jogos digitais. **XIV Simpósio Brasileiro de Games e Entretenimento Digital SBGames, Trilha da Cultura, Teresina**, p. 819–828, 2015. Citado na página 30.

LARA, I. C. M. Jogando com a matemática de 5ª a 8ª série. **São Paulo: Rêspel**, p. 170, 2004. Citado na página 19.

Luann Motta Carvalho. **Magnavox Odyssey: primeiro console da história foi lançado há 50 anos**. 2023. Disponível em: <https://olhardigital.com.br/2022/10/04/games-e-consoles/magnavox-odyssey-primeiro-console-da-historia-foi-lancado-ha-50-anos/>. Acesso em: 11 de novembro 2023. Citado na página 18.

MDN Web Docs. **WebGL**. 2023. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/WebGL_API>. Acesso em: 22 de agosto 2023. Citado na página 79.

Paulo Kirvan. **Definition of prototype**. 2023. Disponível em: <https://www.techtarget.com/searcherp/definition/prototype>. Acesso em: 03 de novembro 2023. Citado na página 37.

Portal Educação. **Origem dos Jogos e Brincadeiras**. 2023. Disponível em: <https://blog.portaleducacao.com.br/origem-dos-jogos-e-brincadeiras>. Acesso em: 11 de novembro 2023. Citado na página 18.

PORTZ, L. G.; EICHLER, M. L. Uso de jogos digitais no ensino de química: um super trunfo sobre a tabela periódica. **Encontro de Debates sobre o Ensino de Química**, 2013. Citado na página 29.

QuimiCot Games. **QuimiCot Games**. 2023. Disponível em: <https://www.quimicotgames.com>. Acesso em: 25 de outubro 2023. Citado na página 31.

ROMANO, C. G.; CARVALHO, A. L.; MATTANO, I. D.; CHAVES, M. R. M.; ANTONIASSI, B. Perfil químico: um jogo para o ensino da tabela periódica. **Revista Virtual de Química**, v. 9, n. 3, p. 1235–1244, 2017. Citado na página 29.

SILBERMAN, M. **Active Learning: 101 Strategies To Teach Any Subject.** [S.l.]: ERIC, 1996. Citado na página 14.

SILVEIRA, R. S.; BARONE, D. A. C. Jogos educativos computadorizados utilizando a abordagem de algoritmos genéticos. **Universidade Federal do Rio Grande do Sul. Instituto de Informática. Curso de Pós-Graduação em Ciências da Computação**, 1998. Citado na página 18.

Simran Kaur Arora. **Unity vs Unreal Engine**. 2023. Disponível em: <https://hackr.io/blog/unity-vs-unreal-engine>. Acesso em: 11 de novembro 2023. Citado na página 22.

SIQUEIRA, C. C. d. et al. A tabela periódica segundo a cosmoquímica: um jogo digital no ensino de química. Universidade Federal de Itajubá, 2023. Citado na página 29.

SMITH, G.; CHA, M.; WHITEHEAD, J. A framework for analysis of 2d platformer levels. In: **Proceedings of the 2008 ACM SIGGRAPH symposium on Video games**. [S.l.: s.n.], 2008. p. 75–80. Citado na página 33.

Unity Technologies. **Asset Workflow - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/Manual/AssetWorkflow.html>. Acesso em: 12 de novembro 2023. Citado na página 23.

_____. **Creating and Using Scripts - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html>. Acesso em: 12 de novembro 2023. Citado 2 vezes nas páginas 26 e 27.

_____. **Game Objects - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/Manual/GameObjects.html>. Acesso em: 12 de novembro 2023. Citado na página 25.

_____. **Introduction to Components - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/Manual/Components.html>. Acesso em: 12 de novembro 2023. Citado na página 25.

_____. **MonoBehaviour - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/Manual/class-MonoBehaviour.html>. Acesso em: 13 de novembro 2023. Citado na página 27.

_____. **Prefabs - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/2023.3/Documentation/Manual/Prefabs.html>. Acesso em: 13 de novembro 2023. Citado 2 vezes nas páginas 27 e 28.

_____. **Raycast - Unity**. 2023. Disponível em: <https://docs.unity3d.com/ScriptReference/RaycastHit2D.html>. Acesso em: 22 de agosto 2023. Citado na página 67.

_____. **Scenes - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/Manual.html>. Acesso em: 12 de novembro 2023. Citado na página 23.

_____. **Scenes - Unity Manual**. 2023. Disponível em: <https://docs.unity3d.com/Manual/CreatingScenes.html>. Acesso em: 12 de novembro 2023. Citado na página 24.

_____. **Unity - companhia**. 2023. Disponível em: <https://unity.com/pt/our-company>. Acesso em: 01 de julho 2023. Citado na página 21.

_____. **Unity Engine - Introduction to Tilemaps**. 2023. Disponível em: <https://learn.unity.com/tutorial/introduction-to-tilemaps>. Acesso em: 04 de julho 2023. Citado 2 vezes nas páginas 39 e 40.

_____. **Unity Engine - Player Input System**. 2023. Disponível em: <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/UnityEngine.InputSystem.PlayerInput.html>. Acesso em: 02 de julho 2023. Citado na página 38.

_____. **Unity Visual Scripting**. 2023. Disponível em: <https://unity.com/features/unity-visual-scripting>. Acesso em: 03 de novembro 2023. Citado na página 88.

VALENTE, J. A.; ALMEIDA, F. J. D. Visão analítica da informática na educação no brasil: a questão da formação do professor. **Revista Brasileira de Informática na educação**, v. 1, n. 1, p. 45–60, 1997. Citado na página 19.

VICTAL, E.; MENEZES, C. de. Um ambiente para apoio à avaliação da aprendizagem em jogos digitais. In: **Anais dos Workshops do Congresso Brasileiro de Informática na Educação**. [S.l.: s.n.], 2016. v. 5, n. 1, p. 477. Citado na página 30.

VICTAL, E. R. D. N.; MENEZES, C. S. Avaliação para aprendizagem baseada em jogos: Proposta de um framework. **XIV Simpósio Brasileiro de Jogos e Entretenimento Digital**, p. 970–977, 2015. Citado na página 30.

Wikipédia. **Máquina de estados finita**. 2023. Disponível em: <https://pt.wikipedia.org/wiki/MÃąquina_de_estados_finita>. Acesso em: 19 de julho 2023. Citado na página 53.

_____. **Tenis for two**. 2023. Disponível em: <https://pt.wikipedia.org/wiki/Tennis_for_Two>. Acesso em: 11 de novembro 2023. Citado na página 18.