UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Mateus Meneses Fittipaldi

Comparação entre os motores de jogo Unity, Godot e GameMaker: Um Estudo de Caso

Recife

2023

Mateus Meneses Fittipaldi

Comparação entre os motores de jogo Unity, Godot e GameMaker: Um Estudo de Caso

Monografia apresentada ao Curso de Bacharelado em Ciências da Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciências da Computação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciências da Computação

Orientador: Leandro Marques do Nascimento

Recife

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

F547c Fittipaldi, Mateus Meneses
Comparação entre os motores de jogo Unity, Godot e GameMaker: um estudo de caso / Mateus Meneses Fittipaldi. - 2024.
107 f. : il.

Orientador: Leandro Marques do Nascimento.
Inclui referências.

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, , Recife, 2024.

1. Motor de jogo. 2. Desenvolvimento de jogos. 3. Unity. 4. Godot. 5. GameMaker. I. Nascimento, Leandro Marques do, orient. II. Título

CDD

MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
(UFRPE) BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

http://www.bcc.ufrpe.br

FICHA DE APROVAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO

Trabalho defendido por Mateus Meneses Fittipaldi às 08 horas do dia 07 de março de 2024, no link https://meet.google.com/tfa-acnq-mca, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado Comparação entre os motores de jogo Unity, Godot e GameMaker: Um Estudo de Caso, orientado por Leandro Marques do Nascimento e aprovado pela seguinte banca examinadora:

Leandro Marques do Nascimento
DC/UFRPE

Kellyton Brito
DC/UFRPE

Agradecimentos

Agradeço à minha família pelo suporte que sempre me deram para continuar crescendo. Aos meus amigos pelo apoio e ajuda para concluir os projetos do meu trabalho. Aos meus professores por acreditarem em mim ao longo do curso. Ao meu orientador, o professor Leandro Marques pelo incentivo e contribuição ao longo deste trabalho. E acima de tudo a Deus por me dar forças para chegar até aqui.

“Nosso maior medo não é que sejamos inadequados. Nosso maior medo é que sejamos poderosos além da medida. É nossa luz, não nossas trevas, que mais nos apavora.”

(Marianne Williamson)

Resumo

Jogos eletrônicos fazem parte do cotidiano de milhões de pessoas e representam um mercado que movimenta bilhões de dólares anualmente. Além do entretenimento proporcionado aos jogadores, a atividade de criação de jogos também traz diversos benefícios. Para facilitar a criação desses produtos, os motores de jogo surgem como resultado de décadas de desenvolvimento na indústria de jogos. Por meio deles, criar jogos nunca foi tão fácil e acessível como é hoje. No entanto, diante da ampla variedade de opções disponíveis, escolher o motor mais adequado para um projeto requer compreensão de suas funcionalidades, capacidades e adequação ao tipo de jogo desejado. Nesse contexto, foram selecionados três dos motores de jogo gratuitos mais populares - *Unity*, *Godot* e *GameMaker* - para uma análise detalhada de suas capacidades e recursos. A avaliação ocorreu por meio do desenvolvimento do jogo *"Fox vs Plants"*, em cada um dos motores selecionados, e da participação em três *game jams*, quando foi possível desenvolver mais dois jogos, aumentando assim a validade do estudo. Os resultados da comparação revelam que cada motor possui seus pontos fortes e fracos: *Unity* possui mais recursos, *Godot* é mais simples de usar e *GameMaker* é aprendida mais facilmente.

Palavras-chave: motor de jogo, desenvolvimento de jogos, programação, *Unity*, *Godot*, *GameMaker*, *game jam*.

Abstract

Electronic games are part of the daily lives of millions of people and represent a market that generates billions of dollars annually. In addition to providing entertainment to players, game development also brings various benefits. To facilitate the creation of these products, game engines emerge as a result of decades of development in the gaming industry. Through them, creating games has never been as easy and accessible as it is today. However, given the wide variety of options available, choosing the most suitable engine for a project requires understanding its functionalities, capabilities, and suitability for the desired type of game. In this context, three of the most popular free game engines - Unity, Godot, and GameMaker - were selected for a detailed analysis of their capabilities and features. The evaluation took place through the development of the game ”Fox vs Plants” in each of the selected engines, and participation in three game jams, where it was possible to develop two more games, thereby increasing the validity of the study. The comparison results reveal that each engine has its strengths and weaknesses: Unity has more resources, Godot is easier to use, and GameMaker is learned more easily.

Keywords: game engine, game development, programing, Unity, Godot, GameMaker, game jam.

Lista de ilustrações

Lista de tabelas

Lista de *snippets* de código

Lista de abreviaturas e siglas

2D	2 dimensões
3D	3 dimensões
UI	*User Interface*
RPG	*Role Playing Game*
IDE	*Integrated development environment*
GDScript	*Godot Script*
GDExtension	*Godot Extension*
GML	*GameMaker Language*
XR	*Extended reality*

Sumário

1 Introdução

Videogames estão presentes no cotidiano de boa parte da sociedade. Apesar de não ter uma estatística global, apenas nos Estados Unidos mais de metade da população joga pelo menos uma hora por semana (Entertainment Software Association, 2023). Por ser uma atividade tão popular, também, é muito rentável. Nos últimos anos, o mercado de jogos está cada vez mais em alta. Em 2020, foi avaliado em 159 bilhões de dólares, em contrapartida com os mercados de música e cinema que foram avaliados em 19 e 41 bilhões respectivamente (DIVERS, 2023). Esse número cresceu ainda mais durante a pandemia de 2020 e atualmente os jogadores relatam que jogam praticamente tanto quanto ou até mais em relação àquela época (Entertainment Software Association, 2023), por causa dos benefícios que sentem.

Jogar é uma atividade de lazer, ou seja, ajuda a entreter e desestressar, mas além disso, também aproximam as pessoas. Tendo videogames como um ponto em comum, as pessoas conversam sobre o que gostam e trabalham em equipe jogando juntos. É uma atividade que fortalece habilidades motoras, de resolução de problemas, comunicação etc. E não existem restrições para quem pode jogar, praticamente qualquer pessoa pode encontrar um jogo que lhe traga alguma diversão, seja um jovem jogando um jogo de *puzzle* ou até um idoso um de ação, pois são tantos tipos de jogos e cada um com objetivos diferentes. Existem aqueles para divertir, competir, ensinar, treinar e/ou simular (MATTOS et al., 2018; COSTA; MEDEIROS, 2020). No fim, todos podem ser uma ferramenta com diversos benefícios. Inclusive, criar jogos eletrônicos também é uma atividade que pode desenvolver boas habilidades.

Desafiando os desenvolvedores a raciocinar e elaborar um jogo, estudantes têm sua mente exercitada, principalmente na área de computação, onde os conceitos de programação são explorados (COSTA; MEDEIROS, 2020; COMBER et al., 2019). O problema, até então, era justamente a dificuldade de criar um jogo do zero, mas isso foi endereçado com o desenvolvimento de motores de jogos. O uso dessas ferramentas tornou a criação de jogos uma atividade muito mais simples, pois facilitam a implementação de recursos comuns em produtos do gênero, como renderizar as imagens, simular a física e criar um *loop* de jogo (ANDRADE, 2015). Hoje, existem dezenas de ferramentas que ajudam no desenvolvimento, a barreira para começar a trabalhar na área foi bastante reduzida, ainda mais com os motores gratuitos. Qualquer pessoa com os recursos de *hardware* pode baixar um motor e começar a aprender. Entretanto, com tantas opções, o novo problema passa a ser qual motor escolher.

Esse tópico já foi explorado em outras pesquisas (VOHERA et al., 2021), porém

os objetivos variam. Em uma análise, pode se buscar o melhor motor para desenvolver um jogo sério também conhecido como jogo educativo (PAVKOV; FRANKOVIć; HOIć-BOžIć, 2017), em outra o melhor para um curso de desenvolvimento de jogos (DICKSON et al., 2017) e em uma terceira, qual motor produz jogos com melhor desempenho (ŠMíD, 2017). Cada uma dessas pesquisas vai fornecer resultados diferentes. Os motores analisados são outros, os objetivos mudam, o método de pesquisa varia e a época em que pesquisa é realizada pode ter anos de diferença. Motores de jogos evoluem muito rapidamente, tanto que, em poucos meses uma nova versão do mesmo motor com mais recursos pode ser lançada, mas isso não deve desencorajar pesquisas na área, pois ainda demora até que as atualizações da ferramenta mudem significativamente a forma como os usuários a utilizem.

Dessa maneira, após um processo de escolha baseado na popularidade e acessibilidade, medida pela quantidade de jogos criados com cada motor e disponibilizados nas plataformas *Steam* e *itch.io*, foram selecionados para comparação três motores de jogo gratuitos: *Unity*, *Godot* e *GameMaker*. Levando ao objetivo de encontrar as características que os tornam tão populares e como se diferenciam entre si. Como método de avaliar as ferramentas foram desenvolvidos:

- Dois jogos com *Unity*, um durante uma *game jam* e outro independente;
- Dois jogos com *Godot*, um durante uma *game jam* e outro independente;
- Um jogo com *GameMaker* de forma independente.

Os três jogos desenvolvidos de forma independente foram os mesmos, afim de comparar os resultados de trabalhar com cada motor igualmente.

A estrutura desta monografia tem continuidade com o capítulo 2, Contexto, trazendo um pouco da história dos jogos que culmina no desenvolvimento dos motores que temos atualmente, além de detalhes sobre as ferramentas escolhidas. Seguindo com o capítulo 3, Metodologia de Trabalho, que relata o processo de pesquisa e o aprendizado dos motores. O capítulo 4, Desenvolvimento, aborda todo o decorrer dos projetos, apresentando detalhes do trabalho com cada motor e fazendo uma comparação entre os três. E por fim, o capítulo 5, Conclusão, traz as considerações finais da pesquisa junto com ideias para trabalhos futuros.

2 Contexto

Antes de se aprofundar nas características de um motor de jogo, vale entender os conceitos fundamentais de jogos. Nesse contexto, existem os brinquedos ou brincadeiras que são uma forma de livre expressão, comportamento e interpretação. Jogos, por outro lado, são uma brincadeira regrada com objetivos pré-determinados (GROH, 2012). Jogos eletrônicos, portanto, são uma brincadeira com regras e objetivos presentes e regidos por um ambiente virtual.

Para criar uma experiência atraente e engajadora, os *games*, utilizando elementos que estabelecem uma conexão com o usuário, apresentam desafios adequados com objetivos claros e oferecem *feedback* constante, enquanto mantêm a autonomia do jogador, sem obrigá-lo a realizar tarefas específicas. Tais recursos também os tornam excelentes ferramentas para se trabalhar em conjunto com outras áreas. Inclusive, algumas aplicações são consideradas ”gamificadas” por utilizarem elementos de jogos que as tornam mais atrativas e engajadoras. Porém, é importante notar que há uma diferença sutil entre conter elementos de um jogo e ser, de fato, um jogo.

A estrutura de um jogo eletrônico geralmente é composta por:

- Objetivo que apresente claramente o que deve ser feito;
- Conjunto de regras limitando as ações dos jogadores ao mesmo tempo que lhe fornece as ferramentas necessárias para alcançarem o objetivo;
- Jogador ou jogadores dependendo se é um jogo *single-player* ou *multiplayer*;
- Mecânica de jogo, que é a forma como os jogadores interagem com o programa;
- Desafios para os jogadores superarem, que não podem ser demasiadamente fáceis nem difíceis, pois podem tornar o jogo chato ou estressante;
- Progressão conforme os jogadores avançam para evitar repetição, pode ser através de aumento da dificuldade ou apresentação de novos elementos e cenários;
- Narrativa é um elemento opcional, mas é uma forma de manter os jogadores engajados com o jogo;
- Gráficos e áudio fornecem uma melhor experiência de jogo e é através desses elementos que os jogadores recebem *feedback* de suas ações;
- Interface de usuário (UI) que inclui menu de jogo, ícones, medidores, pontuação, que são os elementos pelos quais os jogadores interagem com o jogo e verificam seu progresso;

- Sistema de recompensas é a forma como os jogos motivam os jogadores, seja com itens do jogo ou pontuação para competirem entre si;
- Final do jogo para os jogadores saberem a condição de vitória e derrota e quando o jogo termina.

Embora, muitos não possuam todos esses elementos, eles são bastante comuns entre os jogos eletrônicos (GROH, 2012).

Implementar esses recursos difere um pouco de criar um *software* tradicional. Em um jogo, o fluxo não segue um padrão. O jogador inicia o programa, e a partir daí pode acontecer qualquer coisa, o jogo pode começar de imediato, pode haver um menu, podem haver múltiplas formas diferentes de jogar, cada videogame criado é único nesse sentido. Mas, apesar disso, muitos elementos dos jogos podem ser reutilizados de um projeto para outro. Como forma de facilitar a implementação, os motores de jogos trazem um conjunto de recursos padrões dos *games* já implementados. Embora seja necessário aprender a programar e utilizar a ferramenta, esses *softwares* possibilitam a criação de jogos mais complexos em menos tempo.

Neste capítulo será abordado em mais detalhes o que é um motor de jogo, como melhoram o desenvolvimentos de jogos, sua evolução ao longo dos anos, quais são os motores mais populares atualmente e uma introdução às ferramentas que foram analisadas nesta pesquisa.

2.1 Motores de jogos

Existem vários produtos conhecidos como motores de jogos ou pelo seu termo em inglês, *game engines*, porém não há uma definição concreta de motor de jogo. Essas ferramentas são geralmente reconhecidas como *softwares* projetados para facilitar o desenvolvimento de jogos, visando abstrair funcionalidades básicas que podem ser facilmente reutilizadas em diversos projetos (ANDRADE, 2015).

Algumas características comuns dos *engines* de jogos incluem:

- Motor de renderização, responsável por exibir elementos visuais na tela do usuário;
- Controlador de dispositivos de entrada, como mouse, teclado ou *joystick*, para reconhecer diferentes comandos;
- *Loop* de jogo, que estabelece uma rotina de eventos processados a cada atualização de quadro da tela;

- Motor de física, usado para simular fenômenos físicos e tratar colisões e eventos relacionados;
- Sistema de áudio para incorporar elementos sonoros no jogo;
- Ambiente de desenvolvimento com cena gráfica, o espaço onde os desenvolvedores gerenciam elementos gráficos e suas interações;
- Animação para aprimorar a representação de imagens 2D ou modelos 3D;
- Gerenciamento de memória para acomodar múltiplos elementos no jogo;
- Encadeamento de processos, permitindo a execução paralela de múltiplos processos gerados pelos elementos do jogo;
- Inteligência artificial, utilizada para gerenciar personagens não controlados pelo jogador;
- Sistema de rede para permitir conexão entre jogadores nos jogos *online*;
- Capacidade de publicação em múltiplas plataformas, ampliando o alcance do produto.

Não são todos os recursos que estão disponíveis em todos os *engines*, e alguns podem ser mais complexos de usar do que outros. Mas essas ferramentas tiveram um forte impacto na indústria de jogos. Comparando com um trabalho de manufatura, como a carpintaria, onde é necessário estabelecer um espaço de trabalho, adquirir as ferramentas necessárias e, em seguida, encontrar clientes para oferecer serviços, os motores de jogos fornecem um ambiente de trabalho completo, reúnem as ferramentas necessárias e simplificam a distribuição do produto. Isso permite que os desenvolvedores concentrem seus esforços em melhorar seus jogos. Entretanto, levou anos de evolução até os *game engines* atingirem o estágio de desenvolvimento atual.

2.1.1 História dos jogos e motores de jogos

Ao explorar as origens dos jogos eletrônicos, percebemos algumas criações marcantes para sua história. Seguindo a linha do tempo mostrada na Figura 1, o primeiro computador dedicado para jogar, criado na década de 40, possuía apenas um jogo, Nim, que consistia de remover palitos e quem puxar o último, perde. O jogador só podia jogar contra a máquina, que vencia a maioria das vezes. E por muitos anos, a experiência de jogos eletrônicos se resumia a jogos tradicionais como Nim, Xadrez e *Blackjack* contra o computador. Até o final da década de 50 com a criação do famoso, *Pong*, que se popularizou apenas na década de 70 com o lançamento das máquinas

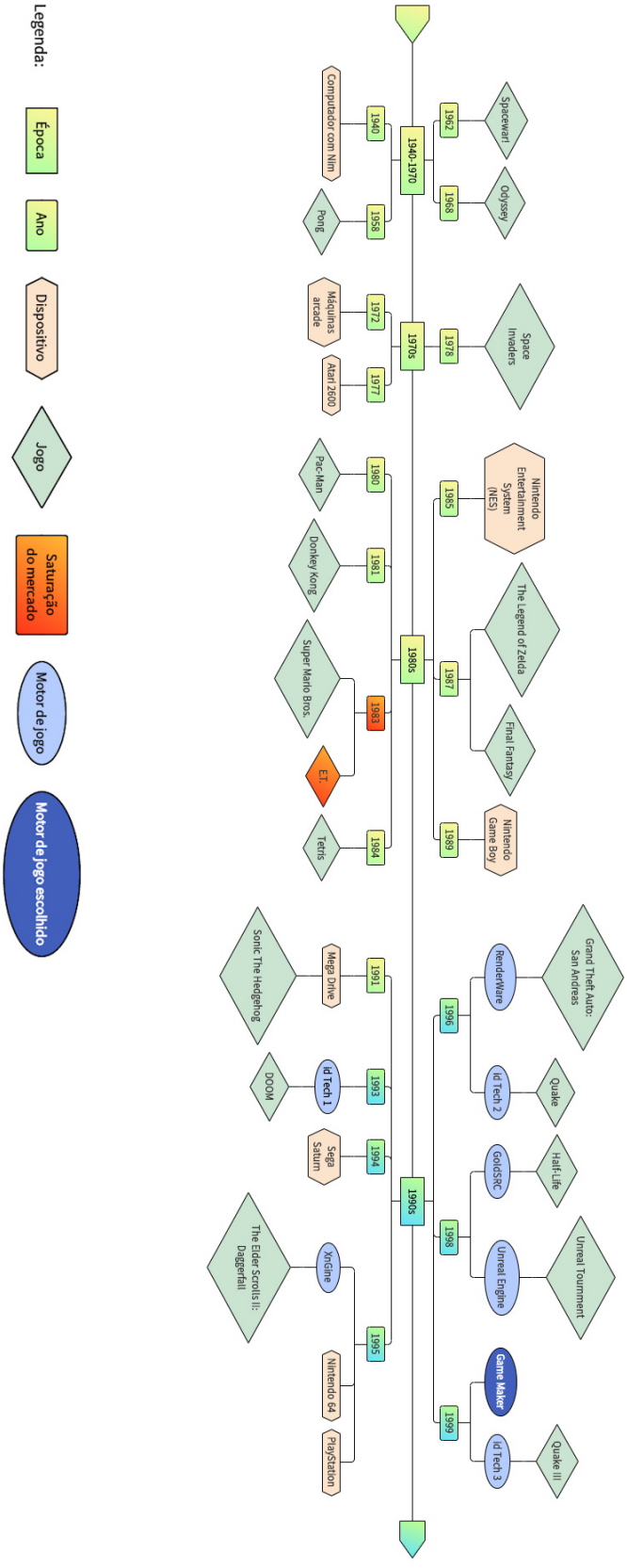

Figura 1 – Linha do tempo dos jogos e motores de jogos de 1940-1999

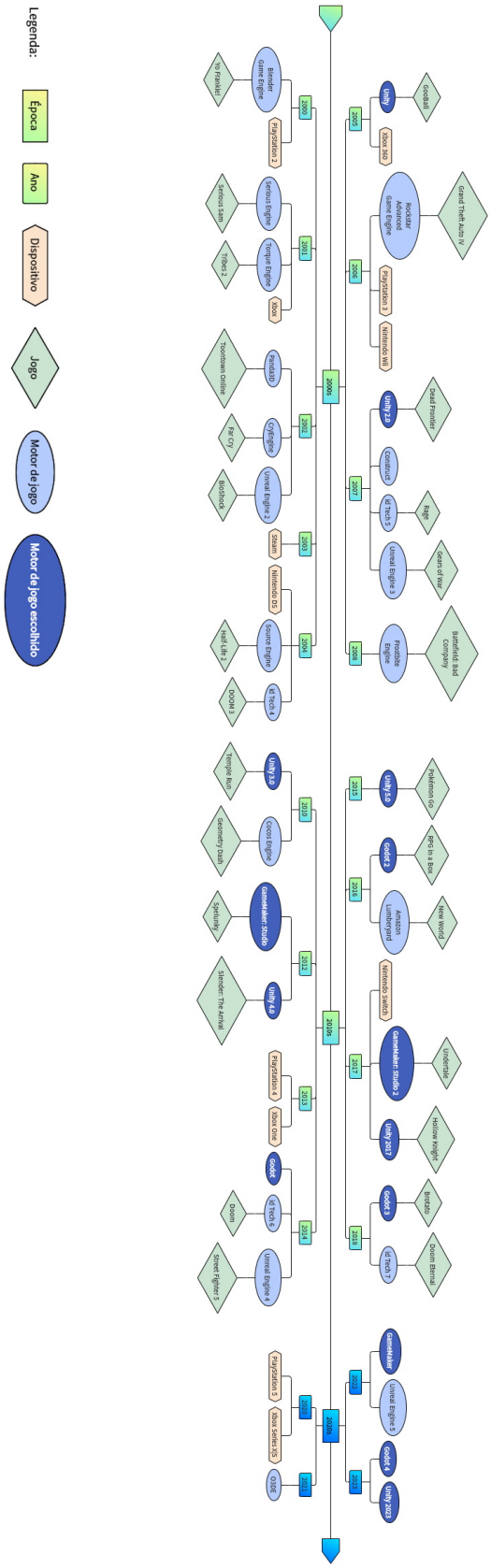

Figura 2 – Linha do tempo dos jogos e motores de jogos de 2000-2023

arcade. Nesse meio tempo, também foi criado o primeiro jogo eletrônico para computador, *Spacewar!*, que foi amplamente compartilhado pelos computadores nos EUA (The Strong National Museum of Play, 2023).

Nessa mesma época também surgiram os primeiros consoles de videogame como *Odyssey*, *Atari 2600* e *Intellivision*. E dando continuidade à linha das máquinas *arcade*, vieram os famosos jogos: *Pac-Man*, *Donkey Kong* e *Super Mario Bros.*, que foram seguidos por uma saturação no mercado de jogos na década de 80 após o lançamento do jogo *E.T.*, marcado de laranja na Figura 1. Por alguns anos, a área de jogos digitais parou de receber grandes investimentos, mas alguns projetos que foram lançados durante esse período conseguiram alavancar o mercado rumo a uma nova ascensão. No final dos anos 80 e início dos 90, o mercado de consoles começou sua competição com os aparelhos da *Nintendo*, o *Nintendo Entertainment System* (NES) e o *Nintendo Game Boy* (NGB), e a *Sega* bateu de frente lançando o *Genesis*, ou *Mega Drive*, e o *Sega Saturn*. Tudo isso trouxe à vida jogos mundialmente conhecidos, como *The Legend of Zelda*, *Final Fantasy* e *Sonic The Hedgehog* (The Strong National Museum of Play, 2023).

A tecnologia continuou evoluindo, novos jogos e consoles foram lançados, e um problema foi observado. Havia a falta de padrões entre diferentes tipos de *hardware*. Cada console e computador tinha sua própria arquitetura, linguagem de programação e especificações técnicas. Portanto, portar um jogo para diferentes plataformas significava recriá-lo em um ambiente completamente diferente. Isso não envolvia apenas ajustar o código, mas também otimizar e adaptar o jogo para garantir um desempenho adequado. Esse processo era caro e demorado. Com a chegada dos sistemas operacionais, esse problema foi em parte mitigado. Em vez de desenvolver jogos para uma ampla variedade de hardwares, os desenvolvedores passaram a focar em sistemas específicos, como o *Microsoft DOS*, *Mac OS* e alguns consoles. No entanto, ainda era necessário realizar muitas modificações ao mudar de uma plataforma para outra. Além disso, as rápidas evoluções na computação frequentemente introduziam novos padrões de sistema.

Essa situação começou a mudar com o surgimento dos motores de jogo. Muitos programas com recursos voltados para jogos começaram a ser desenvolvidos e, embora não seja possível dizer qual foi o primeiro motor de jogo, por não haver uma definição exata, um dos primeiros marcos notáveis que se encaixa na ideia de um motor de jogo é o *id Tech 1*, criado para o jogo *DOOM* pela *id Software* em 1993. Essa ferramenta revolucionou o desenvolvimento de jogos ao separar o código principal, como o sistema de renderização, dos elementos de arte, mundo e regras do jogo (LOWOOD, 2014). Embora não seja um motor de jogo 3D puro, o *id Tech 1*, também chamado *DOOM engine*, conseguia renderizar *sprites* 2D de objetos, personagens e cenários

em uma velocidade impressionante, criando a ilusão de um ambiente 3D.

O primeiro motor 3D verdadeiro foi o *XnGine*, desenvolvido pela *Bethesda* em 1995. A *id Software* só lançou seu primeiro *engine* 3D em 1996, com o *id Tech 2*, ou *Quake engine*, que introduziu a inovadora técnica de *Z-buffering* para determinar quais áreas seriam visíveis e renderizar apenas essas partes (PAUL; GOON; BHATTACHARYA, 2012). A partir daí, mais e mais *engines* foram desenvolvidos.

Outros motores notáveis incluem:

- *RenderWare*, *Criterion Software* (1996) - Títulos das séries *Mortal Kombat*, *Tony Hawk Pro Skater*, *Grand Theft Auto*, *Sonic*, *Pro Evolution Soccer*. Não gratuito (RenderWare - Wikipedia, 2023).
- *GoldSRC*, *Valve* (1998) - *Half-Life*. Gratuito para criar modificações dos jogos da *Valve* (Valve Corporation, 2023).
- *Unreal Engine*, *Epic Games* (1998) - *Unreal*, *Unreal Tournment*, *Fortnite*. Licença gratuita desde 2015 (JENSEN, 2023).
- *GameMaker*, *YoYo Games* (1999) - *Undertale*. Licença para produção de jogos não comerciais gratuita desde 2021 (GameMaker - Wikipedia, 2023).
- *id Tech 3*, *id Software* (1999) - *Quake III*. Código aberto desde 2005 (id Tech 3 - Wikipedia, 2023).
- *Blender Game Engine*, *Blender Foundation* (2000)/*UPBGE* (2021) - *Yo Frankie!*. Código aberto (Blender Game Engine - Wikipedia, 2023).
- *Torque engine*, *GarageGames* (2001) - *Tribes 2*. Código aberto desde 2012 (Torque (game engine) - Wikipedia, 2023).
- *Serious engine*, *Croteam* (2001) - Jogos da série *Serious Sam*. A primeira versão do motor possui código aberto desde 2016 (Croteam, 2023).
- *CryEngine*, *Crytek* (2002) - *Far Cry*, *Crysis*. Gratuita desde 2016 (CryEngine - Wikipedia, 2023).
- *Panda3D*, *The Walt Disney Company* (2002) - *Toontown Online*. Código aberto (Panda3D - Wikipedia, 2023).
- *id Tech 4*, *id Software* (2004) - *DOOM 3*. Código aberto a partir de 2011 (id Tech 4 - Wikipedia, 2023).
- *Source engine*, *Valve* (2004) - *Half-Life 2*. Gratuito para criar modificações dos jogos da *Valve* (Valve Corporation, 2023).

- *Unity*, *Unity Technologies* (2005) - *Pokémon Go*. Licença gratuita desde 2009 (Unity (game engine) - Wikipedia, 2023).
- *Rockstar Advanced Game Engine* (RAGE), *Rockstar San Diego* (2006) - *Grand Theft Auto IV*. Não disponível ao público (Rockstar Advanced Game Engine - Wikipedia, 2023).
- *Construct*, *Scirra Ltd* (2007) - Principalmente jogos de navegador. Licença gratuita com recursos limitados (Construct (game engine) - Wikipedia, 2023).
- *Frostbite Engine*, *DICE* (2008) - Jogos das séries *Battlefield*, *Need for Speed*, *FIFA*. Não disponível ao público (Frostbite (game engine) - Wikipedia, 2023).
- *Cocos engine*, *Xiamen Yaji* (2010) - *Geometry Dash*. Código aberto (Cocos, 2023).
- *Godot*, *Godot Foundation* (2014) - *Brotato*. Código aberto desde o lançamento (Godot Foundation, 2024).
- *Amazon Lumberyard*, *Amazon Games* (2016) - *New World*. Gratuita (Amazon Lumberyard - Wikipedia, 2023).
- *O3DE*, *Linux Foundation* (2021) - Continuação do *Amazon Lumberyard*. Código aberto (Amazon Game Tech Team, 2021).

Alguns dos motores listados foram descontinuados, outros recebem atualizações até hoje, e ainda existem muitos mais que foram desenvolvidos ao longo dos anos, seja para uso interno de empresas, licenciamento por terceiros, pagos ou gratuitos. A disponibilidade de licenças gratuitas ajudou a disseminar o desenvolvimento de jogos, facilitando o aprendizado e o trabalho de várias empresas e desenvolvedores independentes.

A linha do tempo apresentada nas Figuras 1 e 2, foi construída a partir das referências citadas ao longo desta sessão. A evolução dos jogos e dos motores foi observada em (The Strong National Museum of Play, 2023) e (PAUL; GOON; BHATTACHARYA, 2012). Para completar com os motores mais atuais, foram realizadas pesquisas simples no *Google* por ”motores de jogo” e ”motores de jogo populares”. Mais informações sobre o lançamento e jogos populares de cada motor foram coletadas pesquisando separadamente por cada ferramenta encontrada.

2.1.2 Motores gratuitos

Após o lançamento de *DOOM* em 1993, a *id Software* observou muitos jogadores querendo tentando modificar o jogo, seja alterando algo que havia sido criado ou adicionando mais conteúdo. Alguns conseguiram acessar seu código e implementar

suas ideias e isso chamou atenção dos líderes por trás do *id Tech 1*, que resolveram tornar aberto o código base do motor de jogo (LOWOOD, 2014).

Liberar o seu trabalho para outras pessoas foi uma grande mudança no mercado de jogos. Desenvolvedores independentes começaram a surgir criando e modificando projetos com o *engine* da *id Software*. Isso chamou a atenção de outras empresas que começaram a trabalhar mais nos motores em vez dos jogos e para atrair mais usuários, liberaram versões gratuitas de suas ferramentas.

Como motor gratuito, será considerado um *software* dedicado a desenvolvimento de jogos e que esteja disponível para baixar e utilizar sem necessidade de gastos monetários. Isso abrange os motores que são completamente gratuitos e aqueles que possuem licenças de uso gratuitas, pois esses também oferecem planos pagos que garantem mais recursos para os desenvolvedores e maior suporte da empresa responsável. Dentre esses motores, existem aqueles mais utilizados, seja por sua simplicidade de uso, variedade de recursos ou capacidade de produzir jogos de alta qualidade gráfica, o que os tornam excelentes ferramentas para o desenvolvimento.

2.2 Trabalhos relacionados

Diante de tantas informações, é difícil analisar tudo e comparar igualmente duas ou mais ferramentas. Além de que existem dezenas de motores de jogos disponíveis. Escolhendo apenas os gratuitos já limita as opções, mas ainda há uma grande variedade. Outras pesquisas já se propuseram a solucionar esse problema ou contribuir com uma análise de motores jogos.

O estudo de (VOHERA et al., 2021) compara *Unity*, *GameMaker*, *Unreal* e *CryEngine*, focando nas diferenças técnicas e os recursos importantes para desenvolver jogos que cada motor oferece. Através de uma análise da arquitetura geral de um motor e como cada um dos escolhidos se destaca, a pesquisa conclui que não há um melhor motor, pois todos são boas ferramentas, dependendo mais do projeto que será desenvolvido e da habilidade dos usuários.

Com um menor escopo e objetivo diferente, o trabalho de (DICKSON et al., 2017) compara qual o melhor motor entre *Unity* e *Unreal*, para utilizar como ferramenta em um curso de desenvolvimento de jogos. O autor se baseia em fatores relevantes para utilizar os motores na sala de aula e experimenta cada motor ao longo de dois semestres. Com o depoimento dos alunos, a pesquisa conclui que ambos os *engines* são excelentes opções, mas *Unity* é menos complicado de aprender por possuir mais recursos *online*, e *Unreal* atinge resultados visualmente mais satisfatórios. No fim, o motor escolhido depende da intenção do educador.

Em um nível mais aprofundado, as pesquisas de (SINGH; KAUR, 2022), (HA, 2022), (SALMELA, 2022), (PöNNI, 2021) e (COSTA, 2017) mantém o foco da pesquisa em apenas um motor e relatam o processo de criar um projeto com ele. Apesar do objetivo mudar um pouco entre elas, essas pesquisas apresentam os motores analisados, trazem detalhes técnicos e fornecem uma base de como é trabalhar com essas ferramentas.

Seguindo os dois pontos de vista de comparação e prática, a pesquisa de (PAVKOV; FRANKOVIć; HOIć-BOžIć, 2017) faz um comparação entre alguns motores para encontrar o melhor para o desenvolvimento de um jogo sério. São desenvolvidos protótipos do jogo em cada ferramenta e no fim, os pesquisadores optam por utilizar *GameMaker* por ser a que mais se encaixa no escopo deles. E a pesquisa de (ŠMíD, 2017) faz uma comparação de desempenho desenvolvendo o mesmo jogo em *Unity* e *Unreal*. Tendo o mesmo jogo como ponto de comparação, é possível observar com mais precisão as diferenças de cada motor. O autor conclui que *Unity* é mais simples, pois *Unreal* trabalha com C++, que é considerada uma linguagem complexa para aprender, o motivo pelo qual ele escolheu trabalhar com o sistema de programação visual do motor. Mas, apesar de ser mais difícil, o jogo da segunda ferramenta apresentou uma taxa de quadros por segundo maior em praticamente todas as situações do jogo.

Essa linha de raciocínio mostra os benefícios de comparar motores de jogo, como um modo de avaliá-los e facilitar a escolha de uma ferramenta e utiliza o desenvolvimento de jogos como método de levantar detalhes de cada *engine*. É necessário apenas realizar uma seleção inicial de motores para serem analisados.

2.3 Processo de escolha dos motores

O tamanho da comunidade envolvendo um motor jogo é um forte indicador de sua qualidade. Porém, popularidade não é um fator que possa ser facilmente medido, afinal inclui muitas variáveis, como número de instalações do motor e de jogos desenvolvidos, atividade em fóruns *online*, material disponibilizado, tanto pela empresa responsável quanto pelos seus usuários. Em muitas pesquisas, os resultados são parecidos, mas como base para escolha dos motores, este projeto analisou os motores utilizados para desenvolver jogos lançados nas plataformas *Steam* e *itch.io*. A primeira, por ser uma das mais populares para jogos de computador e a segunda por ser muito utilizada para divulgação de jogos e/ou protótipos feitos por desenvolvedores independentes. É possível publicar jogos em ambas, sendo a maior diferenças entre elas, a necessidade de passar por um processo de aprovação para ter o jogo posto na *Steam*, o que agrega mais valor aos jogos aceitos e torna a plataforma um alvo para grandes títulos.

Figura 3 – Jogos por *game engine* na plataforma *Steam* (SteamDB, 2023)

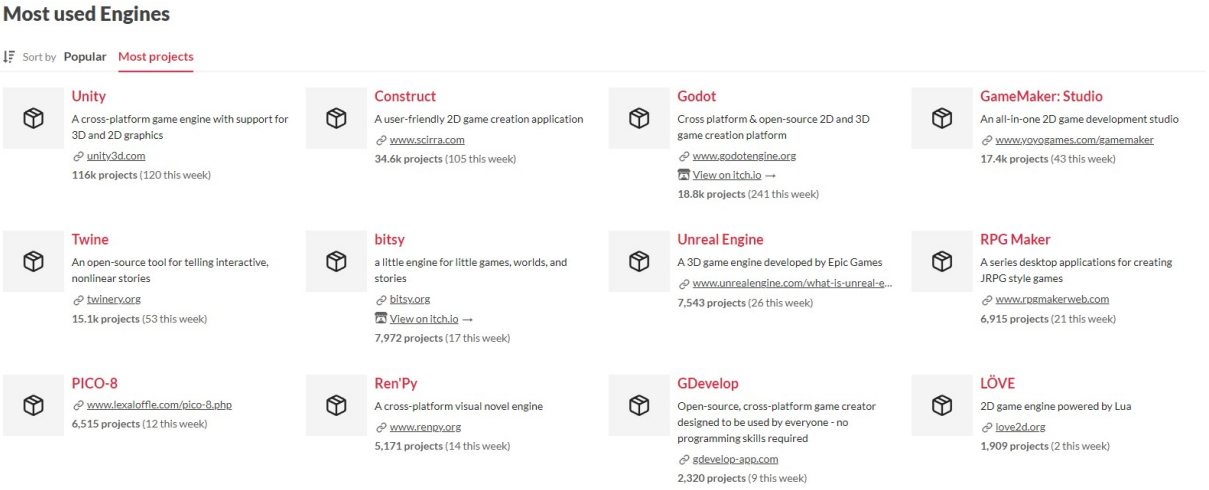

Figura 4 – Jogos por *game engine* na plataforma *itch.io* (itch.io, 2023)

Como observado nas Figuras 3 e 4, juntando o total de jogos desenvolvidos nas duas plataformas, *Unity* é o motor mais utilizado por uma larga margem de diferença dos demais. *GameMaker*, *Godot* e *Unreal* completam o top 4 estando bem próximos um do outro. O quinto lugar do pódio pertence ao *RPGMaker*, que apesar de ser um motor de jogo com capacidade de criação vasta, seu foco está, como o nome sugere, no gênero RPG.

Diante dos resultados, este trabalho selecionou os motores *Unity*, *GameMaker* e *Godot* para comparação, pois são os motores gratuitos mais populares. *Unreal Engine* também entraria na pesquisa, mas precisou ser deixado de lado. Por causa de sua alta exigência de *hardware*, não havia dispositivos disponíveis que cumprissem seus requisitos durante a pesquisa. Além disso também ser um fator que pode afastar desenvolvedores iniciantes sem máquinas de alto desempenho.

2.4 Introdução a *Unity*

Em 2002, em um fórum de *OpenGL*, os criadores do *Unity Engine* se encontraram durante uma busca por colaboradores para um compilador de *shader* com código aberto para desenvolvedores de jogos para *mac* (KORANNE, 2021). A partir daí, começaram o plano de ter sua empresa e criar jogos com seu próprio e avançado motor de jogo, que poderiam também licenciar para outros desenvolvedores. Conforme a equipe aumentava e a ferramenta chegava mais perto de se concretizar, o foco do projeto passou a ser os pequenos desenvolvedores, que não possuíam acesso às mesmas ferramentas que as grandes empresas. Com essa ideia em mente, a primeira versão do *Unity Engine* foi lançada em junho de 2005 como um motor de jogo para *Mac OS X* durante a *Apple Worldwide Developers Conference* (Unity (game engine) - Wikipedia, 2023).

A versão *Unity* 2.0 foi lançada em 2007 com diversas melhorias gráficas, otimizações e novos recursos, como a adição de uma camada de rede para facilitar a criação de jogos *multiplayer online*. Um outro ponto de destaque foi o lançamento da *App Store* da *Apple* em 2008, quando *Unity* rapidamente adicionou suporte a *iPhone* e por muito tempo era a única opção para desenvolver jogos para esse aparelho (Unity (game engine) - Wikipedia, 2023). Até então, a ferramenta só possuía versões pagas, sendo a mais acessível a *Unity Indie* pelo preço de 199 dólares, mas em 2009, ela foi substituída por uma edição gratuita conhecida apenas por *Unity* (HELGASON, 2009).

A terceira grande versão do motor chegou em 2010, com recursos gráficos melhorados para computadores e consoles, além de introduzir suporte a *Android*. Seguido da *Unity* 4.0 em 2012, que trouxe suporte suporte à *DirectX* 11 e *Adobe Flash* (Unity (game engine) - Wikipedia, 2023). Nesta época, o motor da *Unity Technologies* já era considerado a melhor ferramenta para desenvolver jogos para aparelhos móveis.

Em 2015, com a chegada da quinta versão, o *engine* trouxe grandes melhorias de áudio, iluminação, motor de física, efeitos de partícula e performance geral da ferramenta. Além de melhorar a usabilidade de forma que o motor se tornasse mais acessível, também chegou o suporte a *WebGL* para que os desenvolvedores pudessem facilmente adicionar seus jogos a navegadores web compatíveis. Outra mudança nessa versão foi a remoção do suporte a linguagem de programação *Boo*, tornando C#, a linguagem primária do motor (Unity (game engine) - Wikipedia, 2023).

Por fim, desde 2017, *Unity* passou a utilizar o ano de desenvolvimento para numerar as novas versões, de modo que *Unity* 5.6 foi seguido de *Unity* 2017. No decorrer dos anos, o motor continuou sendo aprimorado, recebendo inúmeras melhorias de desempenho e recursos cada vez mais avançados para que os desenvolvedores consigam criar os melhores jogos possíveis. Tudo isso levou o *engine* até sua ver-

são atual: *Unity* 2023.2.11, lançada em 20 de fevereiro de 2024 (Unity Technologies, 2024b).

Até então, diversos projetos já foram criados utilizando *Unity*. A própria empresa promove alguns dos sucessos recentes de sua ferramenta em seu site, mas para ilustrar melhor o potencial do motor, foi realizada uma rápida pesquisa no *Google* pelos jogos mais populares feitos com *Unity*. E o resultado mostrado na Figura 5 trás apenas 51 de seus muitos sucessos.

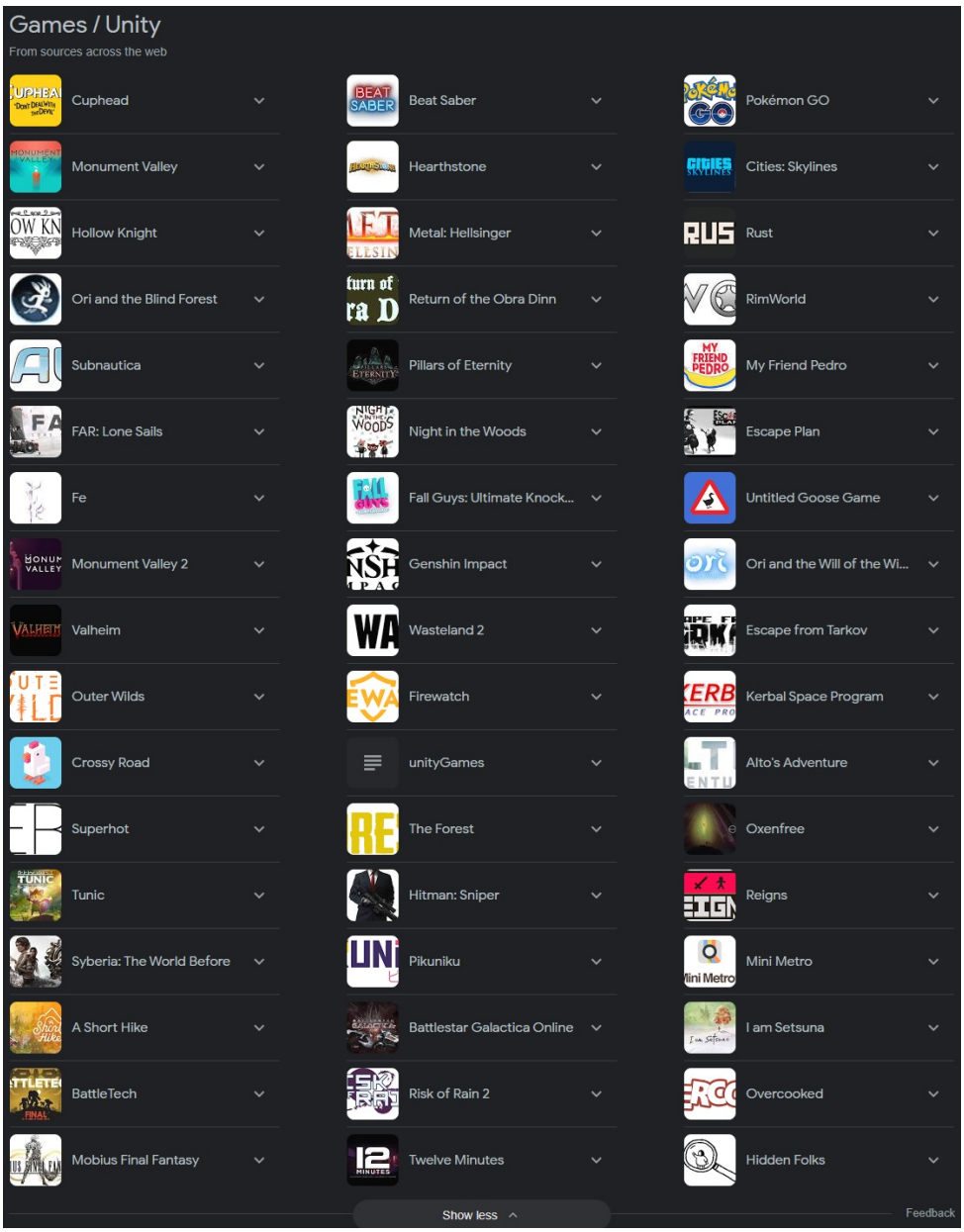

Figura 5 – Resultados da pesquisa: *most popular unity games* (Google, 2023)

Embora sejam projetos grandes que geram bastante lucro, todos começaram pelo mesmo processo simples: instalar o motor de jogo. A *Unity Technologies* recomenda baixar o *Unity Hub*, que permite os usuários terem um maior controle sobre as

versões do motor baixadas, seus projetos e qual versão utilizar em cada projeto. Na Figura 6, é apresentado a tela principal do *Hub* com o menu localizado na lateral esquerda com as abas ”*Projects*”, onde o usuário cria e gerencia seus projetos, ”*Installs*”, local para baixar e gerenciar as versões do *engine*, ”*Learn*”, que trás conteúdo para o usuário aprender e praticar, e ”*Community*”, onde os usuários veem e compartilham materiais desenvolvidos para usar nos projetos.

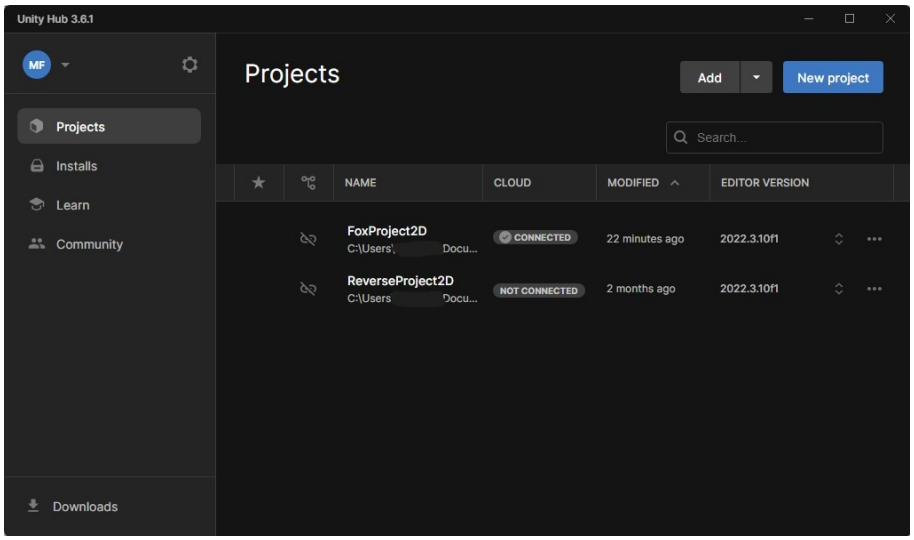

Figura 6 – *Unity Hub*

Após criar um projeto e iniciá-lo, o usuário se depara com o ambiente de desenvolvimento do motor mostrado na Figura 7. Como exemplo, foi criado um projeto do zero utilizando o *template* de projeto 2D oficial do *Unity*. Passando pelas principais funções, há o nome do projeto com a versão da ferramenta utilizada no canto superior esquerdo. Abaixo, uma barra de ferramentas com opções para configurar o projeto e o editor, importar pacotes ou arquivos para o projeto, gerenciar as abas mostradas no editor e realizar algumas ações relacionadas aos componentes do projeto. Explicando brevemente o modelo de projeto do motor de jogo, todo o conteúdo é mostrado através das cenas de jogo, que são as diferentes telas vistas pelo jogador. Dentro de uma cena são adicionados vários objetos para construir o jogo e assim cada cena trás um conjunto de elementos e ações diferentes para o jogador.

No centro da Figura 7, é mostrada a cena de jogo na qual o usuário está trabalhando, basicamente todos os elementos do jogo são mostrados e controlados nessa área. Acima, estão os controles para iniciar e para o teste do jogo. Junto ao nome da aba ”*Scene*”, existe a aba ”*Simulator*”, que permite o usuário testar o jogo em diferentes simuladores de dispositivos, no exemplo em questão, o dispositivo demonstrado é de um celular *Android*. À esquerda da visualização da cena, a aba ”*Hierarchy*” mostra todos os objetos presentes na cena e a hierarquia entre eles. A cena sempre será o primeiro objeto e todos os outros, seus filhos. No exemplo, o único objeto presente é

a câmera principal, necessária para indicar a área do jogo que deve ser renderizada no dispositivo. Abaixo, estão as abas ”*Project*”, com todos os arquivos do projeto e pacotes que estão disponíveis no computador, e ”Console”, onde é mostrado qualquer erro, aviso ou mensagem que o usuário programe para aparecer durante a execução do jogo, é utilizado principalmente para ajudar na depuração do código. Concluindo, a aba da direita da tela é o inspetor que apresenta os detalhes e configurações de qualquer elemento que o usuário selecionar, seja um arquivo ou objeto do jogo.

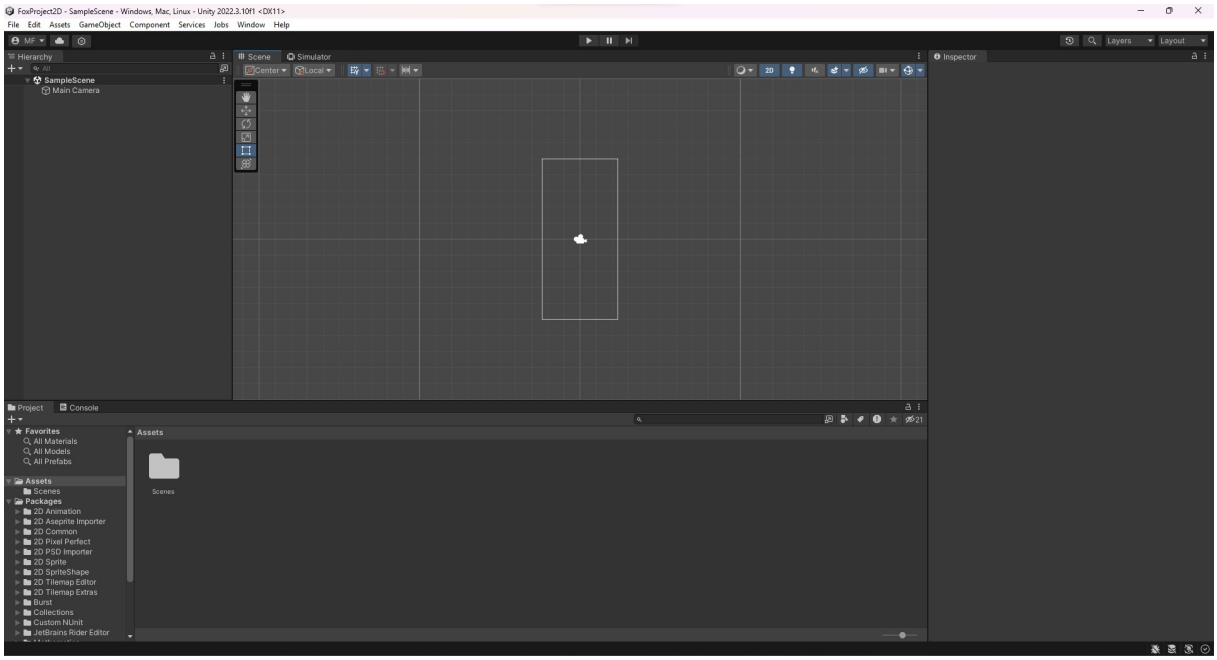

Figura 7 – *Unity IDE*

Nesta pesquisa, o tempo dedicado para trabalhar com *Unity* foi menor em comparação aos outros dois motores, pois já havia uma experiência prévia de trabalho aproveitada durante a análise da ferramenta.

2.5 Introdução a *Godot*

Godot engine é um motor de jogos com código aberto mantido pela *Godot Foundation*. O projeto foi criado com intuito de fornecer uma ferramenta para quebrar barreiras no desenvolvimento de jogos e permitir que qualquer pessoa possa criar videogames tanto 2D quanto 3D (Godot Foundation, 2023a). O desenvolvimento do motor de jogo se iniciou em 2001 e passou por diversas mudanças de nome, estética, gráficos até ser disponibilizado para o público em 2014 (LINIETSKY, 2014), desde então recebe suporte através de doações de indivíduos e organizações.

Sua primeira versão foi lançada como um motor de jogo semelhante a *Unity*, mas com código aberto. Permitia criar jogos 2D ou 3D, apresentou uma linguagem de programação conhecida como *GDSCript* que lembrava *Python*, rodava em *Windows*,

OSX e *Linux* e criava jogos para *iOS*, *Android*, *Desktops*, *Googles' NaCL*, *PlayStation3* e *Vita* (GameFromScratch, 2023). Definitivamente era possível desenvolver jogos com *Godot*, mas ainda faltavam muitos recursos para que o motor pudesse competir com os demais.

Passando por diversas mudanças como melhorias motor de física 2D, *debugger* e instanciação de cenas, novo sistema de busca nos arquivos e edição de múltiplas cenas, além de avanços na sua linguagem de programação, *Godot* 2.0 foi lançado em 2016 (Godot (game engine) - Wikipedia, 2023). Não houveram mudanças drásticas na estrutura do motor, mas várias implementações de qualidade de vida, performance e pequenos recursos adicionais estavam gradativamente melhorando a ferramenta.

Com a chegada de *Godot* 3.0, em 2018, o motor ganhou destaque trazendo sistemas de renderização melhorados, usando *OpenGL ES* 2.0 e 3.0, compatibilidade com realidade virtual, suporte a Mono e C#, troca de motor de física 3D, novo motor de áudio, suporte a *WebAssembly* e *WebGL* 2.0, melhorias significativas para *GDScript*, além de diversas outras pequenas mudanças (GameFromScratch, 2023). Seguido da versão 3.5, que adicionou mais alguns recursos, melhorou o desempenho e corrigiu diversos *bugs*, *Godot* 3 elevou o nível do motor, ganhando bastante espaço entre os desenvolvedores de jogos.

Em 2019, o time de desenvolvimento foi divido em 2, um para continuar o trabalho com *Godot* 3.0 e outro para trabalhar na quarta versão do motor, lançada oficialmente em março de 2023. Essa versão, mais uma vez, elevou o nível da ferramenta para desenvolver jogos, apresentando novos sistemas de renderização utilizando *Vulkan*, *Direct3D 12* e uma nova versão legado de *OpenGL*, melhorias para quase todos os recursos e sistemas do motor e a volta do motor de física 3D próprio de *Godot*. A versão 4.1 melhorou a performance geral da ferramenta e trouxe pequenos aprimoramentos a vários recursos, em especial melhorando a capacidade das linguagens de programação que o motor pode usar: *GDScript*, *VisualScript*, C# e *GDExtension*, uma tecnologia de *Godot* para compatibilizar linguagens não suportadas sem a necessidade de recompilar o motor (GameFromScratch, 2023).

Atualmente, *Godot* está nas versões 4.2.1, lançada em 12 de dezembro de 2023, e 3.5.3, lançada 25 de setembro do mesmo ano (Godot Foundation, 2024). Até então, muitos jogos foram criados, principalmente por desenvolvedores independentes. Embora não tenha tantas criações populares quanto *Unity*, a equipe da *Godot Foundation* destaca algumas mostradas na Figura 8.

Iniciar um projeto com *Godot* é ainda mais simples, pois não é necessário instalar nada, o arquivo baixado já é o motor de jogo, bastando apenas, descompactar a pasta e executar o programa. A primeira tela, mostrada na Figura 9, apresenta o Gerenciador de Projetos, com as abas Projetos Locais, onde o usuário pode administrar

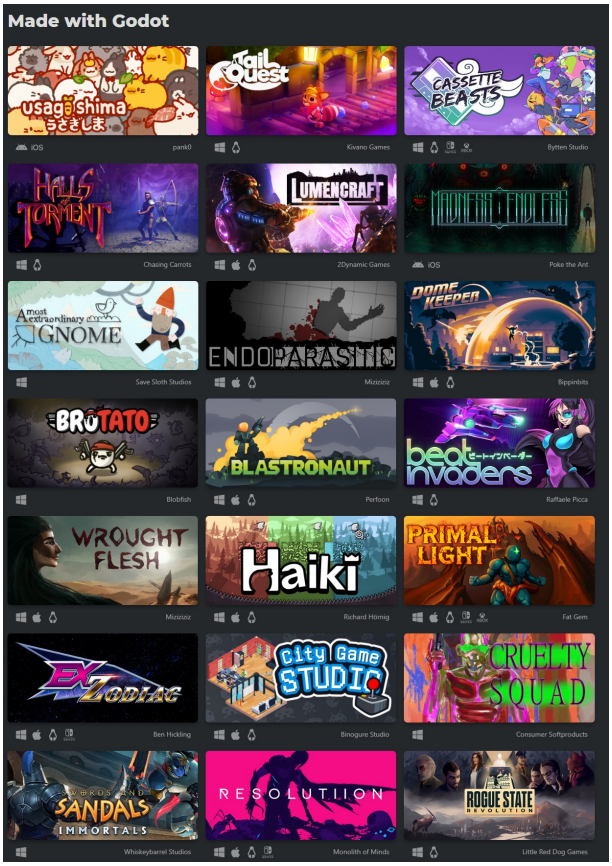

Figura 8 – *Godot Showcase*
(Godot Foundation, 2023b)

seus projetos, criar ou adicionar novos e escolher um para começar a trabalhar, e a aba Projetos da Biblioteca de Recursos, que trás demos, projetos e *templates* de projetos criados e compartilhados por outros usuários. Além de ser uma ferramenta mais leve, é possível perceber mais duas diferenças em relação ao *Unity*. Não é necessário criar uma conta para utilizar o motor de jogo e existe uma tradução em português do programa.

Para exemplificar a primeira visualização do ambiente de desenvolvimento, novamente foi criado um projeto de exemplo. Abrindo um projeto recém criado, o usuário enxerga a tela mostrada na Figura 10. *Godot* segue o mesmo estilo de *Unity*. No canto superior esquerdo há o nome do projeto e do programa. Abaixo, a barra de ferramentas com opções para configurar o projeto, o editor, as cenas de jogo, a depuração e abrir páginas de assistência no navegador *web*. O motor também utilza o modelo de cenas de jogo, porém em vez de objetos e componentes, *Godot* usa nós ou *nodes*, e cada cena precisa ter um node inicial, por isso a janela da esquerda com a aba ”Cena” dá a opção de ”Criar Nó Raiz”. Nessa mesma aba são mostrados todos os nós presentes na cena e a hierarquia entre eles. Na mesma janela, também existe a aba ”Importar” que configura a importação dos arquivos para o projeto. Mais abaixo, há a janela com os arquivos do projeto, a pasta principal é reconhecida como ”res://” e por padrão se

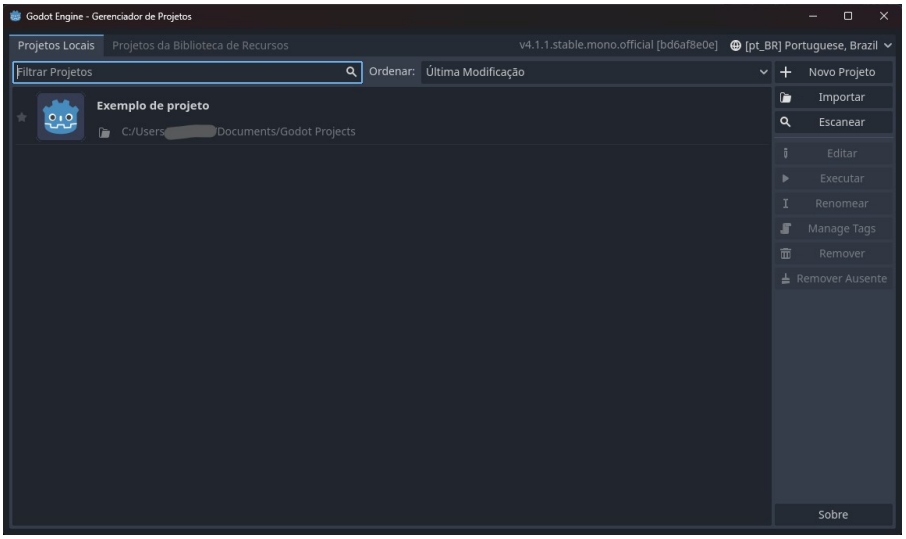

Figura 9 – *Godot Engine* - Gerenciador de Projetos

inicia com uma imagem do ícone do *Godot Engine*.

No centro da tela, assim como *Unity* é mostrada a cena na qual o usuário está trabalhando, mas acima há espaço para abrir mais cenas e alterar rapidamente entre elas. E ainda mais acima, há opções para alterar a visualização da cena entre 2D e 3D, abrir o editor de código e acessar a biblioteca de recursos compartilhados por outros usuários do motor. Os controles para executar os testes estão localizados no canto superior direito. Na barra inferior da tela, à direita é mostrada a versão do motor e à esquerda abas com mensagens de saída do projeto, seja de compilação ou execução, configurações e detalhes de depuração, áudio, animação e *shader*. Terminando a explicação inicial da IDE, na direita da tela estão as abas ”Inspetor” para gerenciar as propriedades dos nós, ”Nó” que administra conexões entre os nós e ”Histórico” com todas as ações que o usuário realiza no editor e permite retornar para estados antes de cada ação.

2.6 Introdução a *GameMaker*

GameMaker é um motor de jogo especializado em jogos 2D, desenvolvido em 1999, originalmente pelo nome *Animo*. Na época, seu propósito era apenas ser uma ferramenta gráfica. Nas versões seguinte, seu nome mudou para *Game Maker* e o foco passou a ser desenvolvimento de jogos 2D. Recebeu recursos como sistema de partículas e conexão de rede na versão 5.0 e a possibilidade de utilizar gráficos 3D na versão 6.0 (GameMaker - Wikipedia, 2023). Até a versão atual, é possível criar jogos 3D com *GameMaker*, porém é uma funcionalidade limitada e complexa do motor.

Em 2007, o criador começou uma parceria com a empresa *YoYo Games* e juntos lançaram a sétima versão do *software*, que continuou passando por mudanças ao

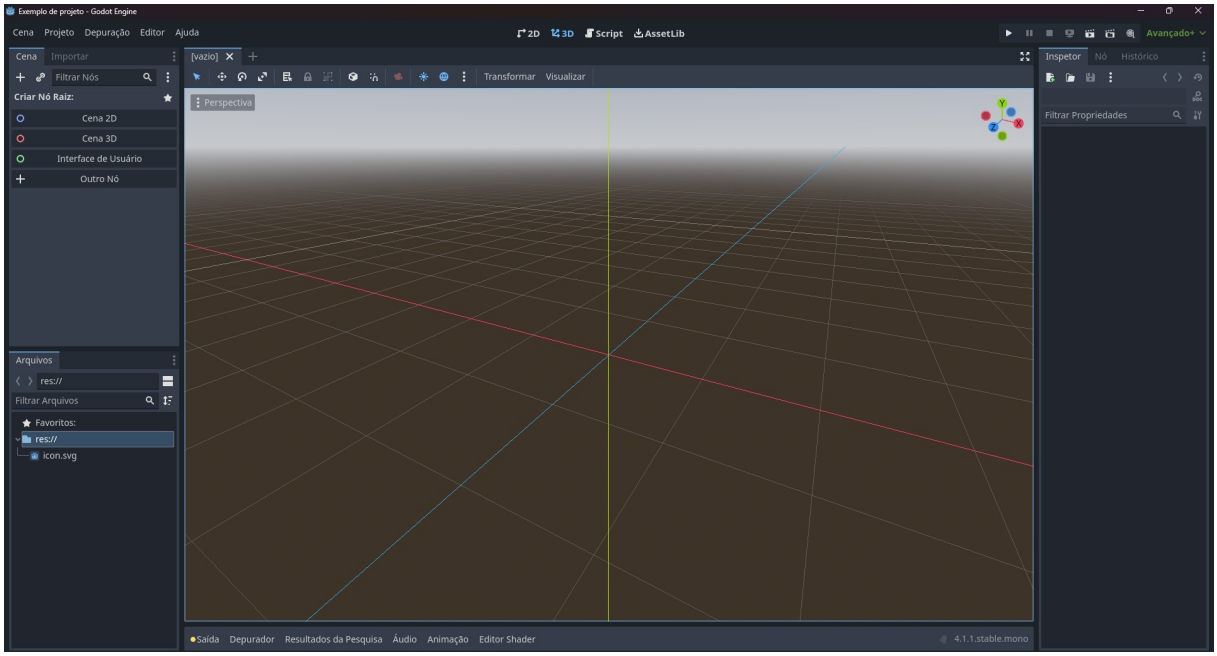

Figura 10 – *Godot Engine IDE*

longo dos anos, tendo seu nome alterado para *GameMaker* (sem espaço) em 2011 e *GameMaker Studio* em 2012. Também houve uma aquisição da *YoYo Games* em 2015 pela *Playtech*, que continou o desenvolvimento da ferramenta. Houve uma grande atualização, em 2017, na qual o editor do motor foi restruturado e recebeu diversas melhorias junto da sua linguagem de programação. Essa versão foi chamada de *GameMaker Studio 2*.

Em 2021, a empresa foi vendida novamente, dessa vez para a empresa *Opera*, que estava iniciando no mercado de jogos. O desenvolvimento do motor não sofreu impacto por isso, exceto pela liberação de uma versão gratuita completa, cuja única limitação é a plataforma de exportação dos jogos. O nome do motor foi alterado novamente para *GameMaker*, em 2022, e suas novas versões passaram a ser numeradas de acordo com o ano e mês de lançamento.

A versão mais atual de *GameMaker*, 2023.11.1.129, foi lançada em 14 de dezembro de 2023 e trouxe uma série de melhorias de performance e novidades a alguns de seus recursos (YoYo Games, 2024). Se sobressaindo no gênero 2D, na Figura 11, *GameMaker* promove jogos bastante populares criados com o motor.

Diferente dos outros motores analisados, *GameMaker* tenta trabalhar com apenas uma versão do motor, notificando quando há uma nova versão para que o usuário já instale e continue trabalhando no mesmo projeto, mas com o motor atualizado. Ao iniciar o programa, como observado na Figura 12, há opções para gerenciar os projetos. É possível iniciar um projeto do zero ou escolher um dos *templates* utilizados para aprendizado. Também há botões que direcionam o usuário para algum conteúdo

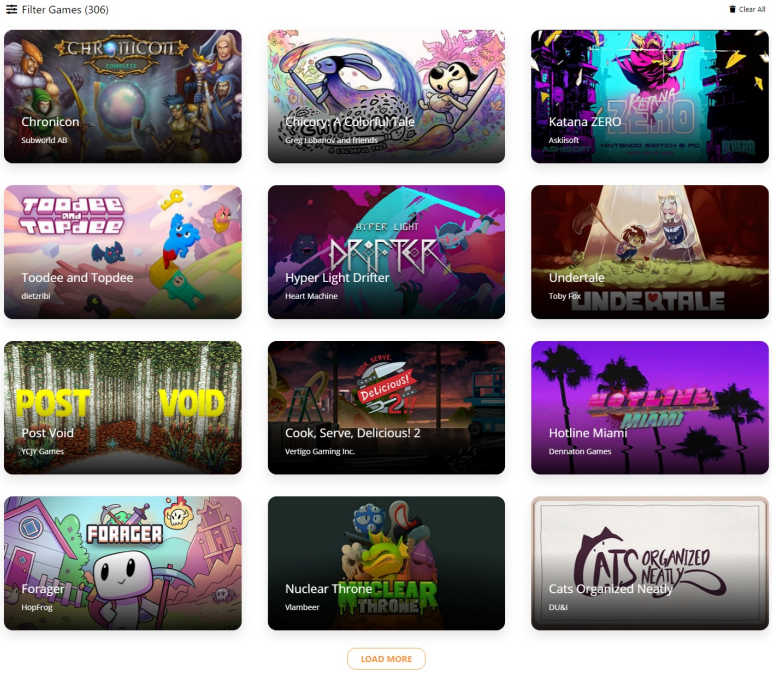

Figura 11 – *GameMaker Showcase*
(YoYo Games, 2023b)

do site do *GameMaker*, como tutoriais, dicas e o fórum do motor. Esta página inicial apresenta uma barra de ferramentas no topo, pois o programa já é o motor de jogo, abrir um projeto apenas muda o foco da página inicial para a área de trabalho do *engine*. E assim como *Godot*, *GameMaker* possui tradução em português, porém muitos componentes, incluindo as pastas do projeto, são traduzidos, o que pode tornar difícil seguir alguns tutoriais que só estão disponíveis em inglês.

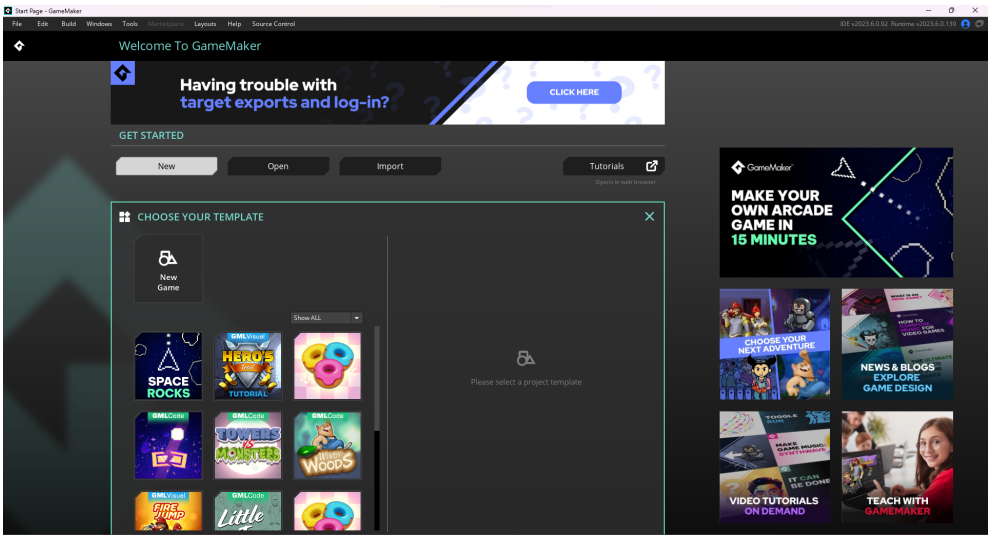

Figura 12 – *GameMaker* - Página Inicial

Ao abrir um projeto pela primeira vez, a tela da Figura 13 é apresentada para o usuário. Do mesmo modo que seus concorrentes, *GameMaker* mostra o nome do projeto e do programa no canto superior esquerdo, com a barra de ferramentas abaixo.

Nessa barra, é possível acessar as configurações do projeto, do editor, gerenciar as janelas expostas, executar o projeto e gerar a *build* do mesmo. Abaixo, há uma outra barra com atalhos para algumas funções do motor, como iniciar ou carregar um projeto, executá-lo, abrir as configurações do jogo, fechar o projeto e retornar para o menu, dentre outras opções. Em uma breve explicação, o modelo de programar de *GameMaker* é o que mais se diferencia dentre os três motores. É possível utilizar sua linguagem própria para programar tanto com *GML Code* quanto *GML Visual*, mas o principal do *engine* é mostrado de forma visual.

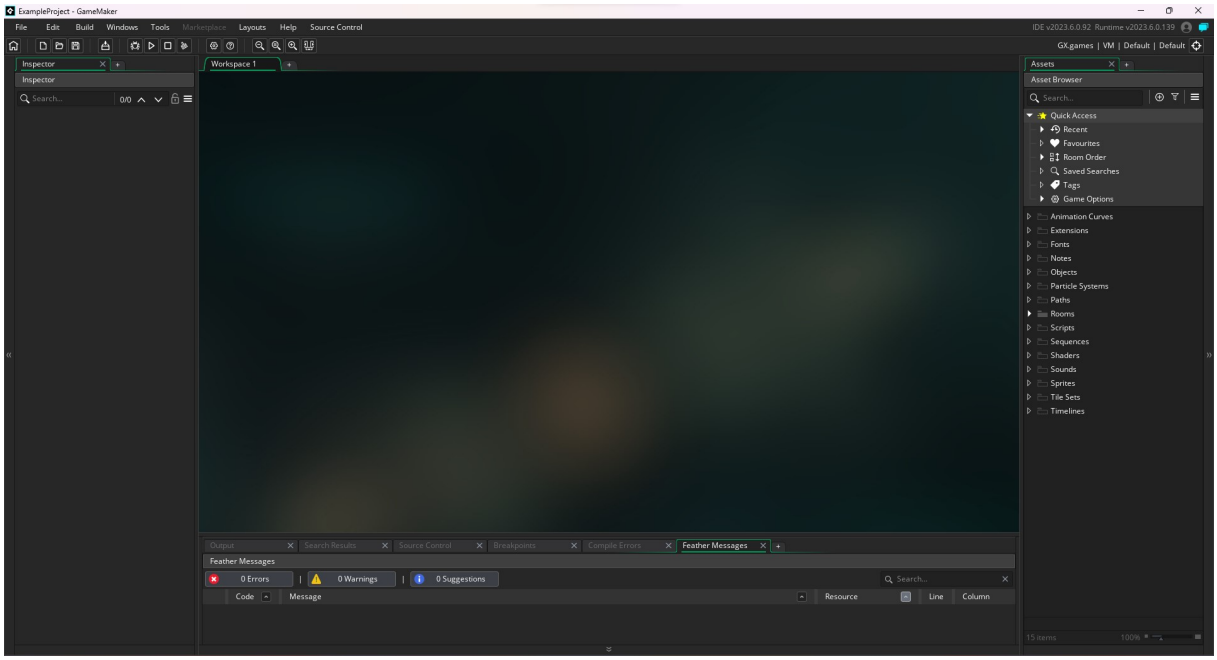

Figura 13 – *GameMaker IDE*

Na área central, de imediato não é mostrada a cena atual do jogo e sim uma área de trabalho onde tudo é programado e organizado. Ao criar um objeto, ele é exposto nessa área e qualquer programação relacionada a ele é feito em janelas que são exibidas nessa área e ligadas por conexões entre o objeto e o código. As cenas de jogo em *GameMaker* são conhecidas como ”*Rooms*” ou ”Salas” e já é criada uma inicial por padrão, mas o usuário precisa abri-la para poder organizar os elementos do jogo. Ao abrir uma ”*Room*”, ela aparece na área central no lugar da área de trabalho, e é possível trocar entre essas e outras janelas pelas abas acima do área central. Ao lado direito da tela, há a janela de ”*Assets*” do projeto com todos os objetos e arquivos. O motor cria por padrão um modelo de gerenciamento de arquivos base com todas as pastas necessárias para o projeto. No lado esquerdo da tela, existe um inspetor como nos outros motores, que mostra configurações dos elementos selecionados. Por fim, na área inferior estão as janelas utilizadas para depuração: saídas do projeto, *breakpoints*, mensagens de erro etc. Inicialmente, o fato da IDE não mostrar a cena de jogo diferencia bastante dos outros motores, ainda mais quando a área de trabalho é

preenchida com vários objetos e pedaços de código, mas manipular os elementos em uma ”*room*” é similar aos outros *game engines*.

2.7 Comparativo inicial dos motores

Coletando informações básicas dos motores em seus próprios sites (Unity Technologies, 2023a; Godot Foundation, 2024; YoYo Games, 2023c), é possível comparar os seguintes pontos mostrados na Tabela 1.

	Unity	***Godot***	***GameMaker***
Empresa	*Unity Technologies*	*Godot Foundation*	*YoYo Games*
Ano de lançamento	2005	2014	1999
Versão atual	Unity 2023.2.11	Godot 4.2.1 e 3.5.3	GameMaker 2023.11.1.129
Licença gratuita	*Student* e *Personal*	Sim	*Free*, mas com restrições de exportação do projeto
Licença paga	*Unity Pro*, *Unity Enterprise* e *Unity Industry*	Não	*Creator*, *Indie* e *Enterprise*
Linguagem de programação	C#, JavaScript	GDScript, VisualScript, C#, GDExtension	GML Code, GML Visual
IDE	Sim	Sim	Sim
Editor de código	Não	Sim	Sim
Plataformas do editor	*Windows*, *macOS* e *Linux*	*Windows*, *macOS*, *Linux*, *Android* e *Web*	*Windows* e *macOS*

Plataformas de distribuição	*Android*, *ChromeOS*, *iOS/iPadOS*, *tvOS*, *PlayStation 4* e *5*, *Xbox One* e *Series X|S*, *Nintendo Switch*, *Windows 7+*, *Mojave 10.14+*, *Ubuntu*, *CentOS 7*, *WebGL* e diversas plataformas *XR* (Unity Technologies, 2023b)	*Android*, *iOS*, *HTML5*, *Mac OSX*, *Universal Windows Platform*, *Windows Desktop* e *Linux/X11* (LINIETSKY; MANZUR; CONTRIBUTORS, 2023a)	*iOS*, *Android*, *Amazon Fire*, *Android TV*, *tvOS*, *Nintendo Switch*, *PlayStation 4* e *5*, *Xbox One* e *Series X|S*, *HTML5*, *Windows*, *Mac* e *Ubuntu* (BRAMBLE, 2023)
Público alvo	Desenvolvedores independentes, pequenas e grandes empresas de jogos, cinema, arquitetura, automóveis, manufatura, energia, aeroespaço e mais, estudantes e educadores (Unity Technologies, 2023a)	Desenvolvedores independentes, estúdios profissionais, estudantes e educadores (Godot Foundation, 2024)	Desenvolvedores independentes, estúdios profissionais, estudantes e educadores interessados em desenvolvimento de jogos 2D (YoYo Games, 2023c)

Tabela 1 – Comparação básica dos motores de jogos

Pela Tabela 1, é possível observar algumas vantagens e desvantagens de cada motor. *Unity* possui versões gratuitas com acesso total ao motor, não possui linguagem de programação própria o que torna mais fácil encontrar conhecimento sobre as linguagens em outras fontes, e também não possui um editor de código, sendo necessário utilizar um editor externo. Pode ser utilizado nos sistemas operacionais *Windows*, *macOS* e *Linux*, possui uma lista extensa de plataformas para exportar os projetos e é amplamente utilizado em diversas áreas.

Godot, por sua vez, é o motor mais recente dos três, mas sua rápida evolução o torna digno de comparação e é o único motor completamente gratuito. Possui uma linguagem de programação própria, *GDScript*, mas é compatível com C# e, graças a tecnologia da *GDExtension*, também pode ser usado com várias outras linguagens. É um motor leve e acessível, disponível para as mesmas plataformas do *Unity* mais *An-*

droid e navegador *web*. Porém, não possui suporte oficial à exportação para consoles, se limitando a *desktop*, *mobile* e *web*. Não atua em tantas áreas, mas possui usuários com diversos níveis de aprendizado.

Por fim, *GameMaker*, é o mais antigo dos três motores e, apesar de possuir uma versão gratuita, as plataformas de exportação são limitadas e não é possível monetizar os projetos criados nessa versão. Sua única linguagem de programação é a *GameMaker Language*, sendo possível utilizar através de código ou programação visual com *GML Code* e *GML Visual*, respectivamente. Dos três motores, é o que está disponível em menos plataformas, apenas *Windows* e *macOS*. Mas nas plataformas de distribuição se estende quase tanto quanto *Unity* deixando de fora apenas as plataformas *XR*. Sua área de atuação é focada principalmente no desenvolvimento de jogos 2D.

Apesar de terem algumas diferenças, todos os três motores se apresentam como boas ferramentas para desenvolver jogos. Os três possuem um ambiente de desenvolvimento integrado (IDE), estão disponíveis em mais de uma plataforma e podem exportar jogos para diversas outras, além de serem amplamente utilizados por usuários com variados níveis de conhecimento. Esta pesquisa tem a intenção de aprofundar essa comparação, visando observar diferenças mais sutis de trabalhar com cada um desses motores. O capítulo seguinte dá continuidade explicando a metodologia utilizada para analisar os três *game engines*.

3 Metodologia de Trabalho

Utilizando a estratégia de desenvolver jogos como forma de analisar os motores de jogo, esta pesquisa esta pesquisa desenvolveu cinco jogos com os três motores alvos: *Unity*, *Godot* e *GameMaker*. Apresentados na Figura 14, três deles são o mesmo jogo desenvolvido em cada motor e os outros dois foram experimentos extras desenvolvidos em maratonas de criação de jogos conhecidas como *game jams*. Neste capítulo, será abordado o processo de aprender e trabalhar com cada motor, alguns detalhes gerais dos motores e informações sobre *game jams* e de quais houve participação.

Figura 14 – Jogos criados em cada motor por ordem de desenvolvimento

3.1 Aprendizado de *Godot*

Como mencionado no capítulo anterior, já havia uma base de experiência com *Unity* e para evitar qualquer tendência, foi escolhido começar a pesquisa com *Godot*.

Estudar uma ferramenta nova sempre é um desafio, sendo necessário descobrir qual versão da ferramenta utilizar, como criar um projeto, configurar o ambiente de desenvolvimento, e como de fato desenvolver o jogo desejado. *Godot* facilita bastante esse procedimento inicial, pois o motor está disponível em diversas plataformas, como *Steam*, *itch.io* e *Google Play*, além do próprio site da ferramenta, onde é mais fácil de ver as versões disponível para *download*, suas especificações e a documentação do motor. Um detalhe importante é observar se a versão baixada possui .NET, pois é necessário para programar em C#, que foi o caso nesta pesquisa. No início deste trabalho, a versão mais atual era a 3.5.1, que foi a utilizada para desenvolver o jogo comparativo. Alguns meses depois, na época da primeira *game jam*, já havia sido lançada a versão 4.0.3, portanto neste trabalho foram utilizadas as duas versões citadas, cada uma para um projeto diferente.

Como primeira experiência com *Godot*, foi escolhida uma demo de projeto disponível na biblioteca de recursos do próprio motor mostrado na Figura 15. O projeto ”2.5D Demo” desenvolvido com *GDScript* consiste em um ambiente simples de plataforma 2D, onde é possível alterar o ponto de vista entre o 2D clássico, 2.5D, *top-down*, isométrico e oblíquo, além de poder alterar os controles do jogo entre o movimento padrão nos eixos X e Y ou um movimento isométrico nas diagonais. Apesar de ser uma demo simples, sem conhecer o motor de jogo nem a linguagem de programação, entender como o projeto funciona se torna difícil, de modo que essa foi uma experiência apenas para conhecer a interface do motor e se familiarizar com o ambiente. Para aprender de fato a utilizar *Godot*, uma série de tutoriais *online* em conjunto com a documentação do motor foram consultados e a partir desse aprendizado, foi criado o projeto utilizado como ponto de comparação entre os motores.

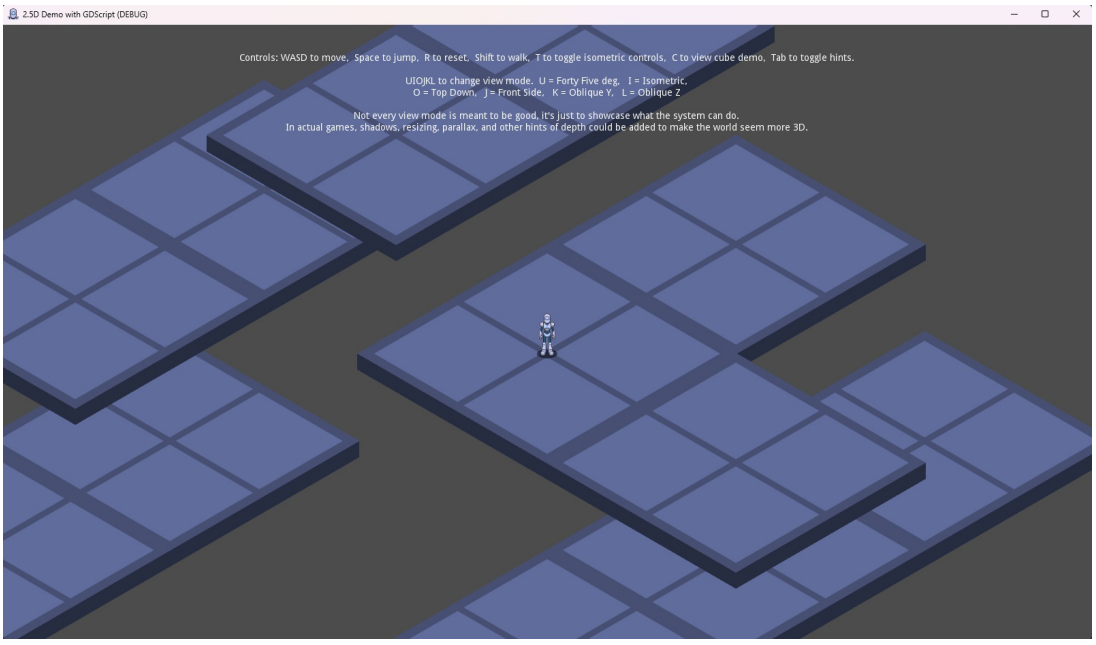

Figura 15 – *Godot 2.5D Demo*

As referências para aprender o motor de jogo *Godot* foram as seguintes:

- Site e canal do *YouTube*, GDQuest (GDQuest, 2023) para aprender conceitos gerais do motor;
- *Playlist* do *YouTube*, *Godot Action RPG Series* (Heartbeast, 2020) para praticar o desenvolvimento de jogos com o motor;
- *Playlist* do *YouTube*, C# *Godot Tutorials* (Abdullah, 2020) para tirar dúvidas sobre programação no motor com C#;
- Documentação de *Godot* (LINIETSKY; MANZUR; CONTRIBUTORS, 2023b) para consultas sobre os recursos do motor e código.

O aprendizado foi fruto das referências citadas acima, sendo a mais utilizada a *playlist Godot Action RPG Series*. De forma geral, reproduzir as implementações demonstradas nos vídeos e pesquisar soluções na documentação ou na série de tutoriais com C#, foi a rotina padrão para aprender *Godot*.

3.2 Aprendizado de *Unity*

O segundo motor experimentado foi *Unity*. Após experimentar *Godot*, trabalhar com este segundo permitiu analisá-lo de forma menos enviesada, mas ainda foi uma experiência mais acelerada devido ao conhecimento prévio da ferramenta.

Começar a trabalhar com *Unity* é simples, mas demorado em comparação a *Godot*. É necessário baixar o motor através do site da *Unity Technologies* ou do *Unity Hub*, instalar e esperar o programa carregar todos os seus recursos, o que pode demorar alguns minutos dependendo do *hardware* utilizado. Durante a instalação, é possível instalar o editor de código *Visual Studio* em conjunto, facilitando a integração com o motor, porém, por preferência pessoal, o editor escolhido foi *Visual Studio Code*, o que tornou necessário um passo extra para configurar o motor para o reconhecer.

O tempo de trabalho com *Unity* foi bem mais reduzido, afinal não houve um tempo de aprendizado. Por uma semana foi praticada uma implementação básica do projeto de comparação utilizando a versão 2022.3.4f1, lançada em 26 de junho de 2023. Logo, a implementação foi interrompida para trabalhar no projeto da *game jam*, que ocorreu em um fim de semana, durando das 12 horas de uma sexta-feira até as 12 horas do domingo. Após esse evento, foi iniciado o aprendizado de *GameMaker*, pois o interesse maior era conseguir explorar os três motores de jogos, caso o tempo para desenvolver todos os projetos não fosse suficiente. O desenvolvimento com *Unity* foi retomado um tempo depois com a versão 2022.3.16f1 de *Unity*, lançada dia 19 de dezembro de 2023.

Embora já houvesse experiência com bastante do conteúdo do motor, algumas consultas foram realizadas nas seguintes referências:

- Canal do *YouTube Brackeys* (Brackeys, 2012) para aprender a implementação de alguns recursos;
- Canal do *YouTube Blackthornprod* (Blackthornprod, 2017) para aprender a implementação de alguns recursos;
- Site *Unity Learn* (Unity Technologies, 2024a) do próprio motor com tutoriais para aprender a implementação de alguns recursos;
- Documentação de *Unity* (Unity Technologies, 2023c) para consultas sobre o código do motor.

Os canais do *YouTube* mencionados são bastante conhecidos por trazerem conteúdo que ajuda no aprendizado de *Unity*, sendo excelentes referências para aprender, ou pelo menos ajudar, a implementar mecânicas dos jogos. E os tutoriais, em conjunto com a documentação, foram extremamente úteis para descobrir como utilizar alguns componentes dos objetos do motor.

3.3 Aprendizado de *GameMaker*

Após o término da segunda *game jam*, *GameMaker* foi selecionado como o terceiro motor de jogo da pesquisa. Além de sua popularidade, a proposta de um jogo 2D para comparação e os requisitos de *hardware* da ferramenta, fizeram dela uma excelente opção para completar o trio de motores.

Assim como nos outros casos, a experiência se iniciou com o *download* do instalador da versão mais atual do motor. Abrindo o *software* pela primeira vez, são dadas as opções de iniciar um novo projeto a partir de um *template* ou importar um. Sem conhecer nada do motor, criar um projeto seguindo os tutoriais de *GameMaker* é uma excelente pedida. Neste trabalho, o projeto *Space Rocks*, mostrado na Figura 16, foi escolhido para ser a introdução ao *engine*. Seguindo o vídeo tutorial (GameMaker, 2022), foi possível aprender o básico do motor, como é sua estrutura, a organização de arquivos, como criar objetos, programar eventos e tratar colisão.

Com o básico do conhecimento adquirido, o mesmo projeto desenvolvido com *Godot* foi recriado na versão v2023.6.0.92 de *GameMaker*, lançada em 19 de julho de 2023. Durante o desenvolvimento, foram utilizadas as seguintes referências:

- Blog de *GameMaker* (YoYo Games, 2023a) para aprender a implementar alguns recursos;

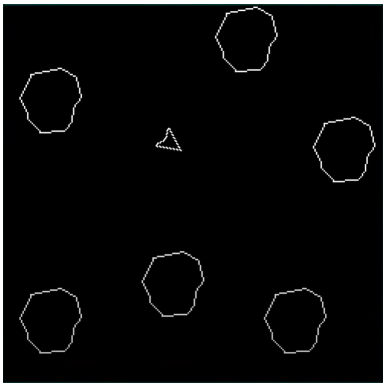

Figura 16 – *GameMaker Space Rocks*

- Fórum de *GameMaker* (XenForo, 2023) para tirar dúvidas sobre código;
- Documentação de *GameMaker* para tirar dúvidas sobre código;

Através do processo de aprender a trabalhar com *GameMaker* foi possível observar diversas diferenças em relação aos outros motores, que serão abordadas no capítulo 4. Mas, por enquanto, vale ressaltar que, num primeiro contato, *GameMaker* trás mais conteúdo para iniciantes na ferramenta do que os demais *engines*.

3.4 *Game jams*

Game jams são bastante conhecidas na área de desenvolvimento de jogos como uma excelente forma de praticar ou aprender algo novo. São eventos que consistem em desenvolver um pequeno projeto dentro de determinado tempo e muitas vezes possuem regras extras para desafiar os desenvolvedores. As características comumente encontradas em *game jams* são:

- Trabalho em equipe: os participantes são encorajados a trabalhar em grupos, normalmente de 2 a 5 pessoas;
- Multidisciplinaridade: também são incentivados a trabalhar com pessoas que possuem diferentes conjuntos de habilidades;
- Restrição de tempo: há um limite de tempo definido para finalizarem um protótipo do projeto, geralmente de em fim de semana ou poucos dias;
- Restrição de tema: o escopo do projeto deve se limitar a um determinado tema, que pode ser algo específico ou muito abrangente. Serve como um desafio à criatividade dos participantes;

- Local físico: a maioria das *jams*, atualmente, ocorrem de forma *online*, contudo, é normal ocorrerem presencialmente em uma localização preparada para receber os participantes pelo tempo do evento;
- Contexto de jogos: existem eventos similares para desenvolver projetos de *softwares* gerais, mas *game jams*, como o nome indica, se limitam a desenvolver jogos dos mais diversos gêneros;
- Objetivo de aprendizado: a ideia por trás das *game jams* não é desenvolver um projeto até o final, apenas um protótipo para praticar alguma habilidade ou ilustrar uma ideia. Caso seja de interesse dos envolvidos, é possível continuar trabalhando para finalizar o projeto após o evento.

Dependendo do evento, todos esses pontos podem ser abordados ou apenas alguns (FOWLER et al., 2016). E também podem haver regras que limitem as ferramentas utilizadas ou proíbam algumas ações, como utilizar inteligência artificial para gerar código. Mas de modo geral, tais eventos consistem em aprender em conjunto com outros desenvolvedores de forma prestativa e respeitosa.

Nesta pesquisa, foram utilizadas três *jams* como etapas do processo de aprendizado dos motores de jogos, mas uma delas não foi bem sucedida. A primeira com *Godot* foi a "*Godot Wild Jam #58*", que ocorreu de 09/06/2023 a 18/06/2023 (Godot Wild Jam, 2023). Essa *jam* ocorre todo mês e foca em desenvolver jogos com *Godot*. Nessa edição, o tema foi "*rain or shrine*", e embora uma ideia tenha sido bolada, houveram dificuldades na hora de implementar.

Na mesma época, começou uma outra *game jam* que ocorreu de 17/06/2023 a 27/06/2023, a "*Learn You a Game Jam: Pixel Edition*" (Captain Coder, 2023). Apesar de ser um evento com menos participantes, a proposta de aprender algo novo durante a *jam* estava mais alinhada com esta pesquisa. O tema proposto foi "*You are the Monster*" e uma das regras era escrever um *dev-log* relatando o que foi aprendido durante o evento. Dessa vez, foi possível concluir o protótipo utilizando *Godot* com um pouco de ajuda na parte artística, apesar de ainda terem tido alguns problemas. O jogo criado durante este evento foi nomeado "*Slimus The City Destroyer*".

A última *jam* foi a mais popular das três, com milhares de participantes e submissões. A "*GMTK Game Jam 2023*", que ocorreu de 07/07/2023 a 09/07/2023, teve como tema: "*roles reversed*" (Game Maker's Toolkit, 2023). O projeto foi desenvolvido em apenas um fim de semana com *Unity* e, diferente dos outros eventos, este foi feito por uma equipe de quatro pessoas, três programadores e uma artista. Mais detalhes serão abordados no próximo capítulo, mas por enquanto vale mencionar que esse projeto foi concluído com sucesso e ainda foi aprimorado para uma mostra de jogos

que ocorreu no ”Festival REC’n’Play 2023” no dia 21/10/2023. Essa oportunidade foi promovida pelos grupos ”Jogo Coletivo” e ”Game Jam das Minas” que abriram uma seleção para que os interessados enviassem um formulário com o jogo desenvolvido e a equipe participante. Após o prazo estabelecido, selecionaram os jogos que seriam apresentados no evento conhecido por ”Mostra teu Jogo”. Como visto na Figura 17, dentre eles, o jogo ”*Follow Me - A ghost game*”, desenvolvido neste projeto durante a *GMTK Game Jam 2023* sob o nome da equipe ”MiniVan Jogos”, foi escolhido para a mostra.

Figura 17 – Lista de jogos selecionados para o evento ”Mostra teu Jogo” (Jogo Coletivo, 2023)

4 Desenvolvimento

O principal ponto de comparação entre os motores foi a experiência de desenvolver um jogo que recebeu o nome de *Fox vs Plants*. A ideia foi repetir o mesmo processo em cada *engine* com poucas alterações entre eles. O jogo foi idealizado a partir do aprendizado de *Godot*, seguindo o tutorial da série de vídeos *Godot Action RPG Series* (Heartbeast, 2020), utilizando os *assets* de arte, efeitos sonoros e música fornecidos por ele e realizar algumas alterações na implementação do projeto do tutorial. Isso possibilitou explorar bastante do motor e chegar no jogo comparativo descrito abaixo.

4.1 *Fox vs Plants*

Como uma raposa aventureira usando uma capa (Figura 18), o jogador precisa destruir todas as moitas que encontrar pela frente (Figura 19). Para atingir seu objetivo, ele tem a sua disposição uma espada e a habilidade de atirar folhas que alteram a cor das moitas, pois para poder destruí-las, é necessário que estejam na cor correspondente à grama abaixo delas. Além disso, é preciso ser rápido, porque existe um tempo limite para que todas as moitas sejam destruídas.

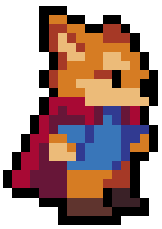

Figura 18 – Raposa

Figura 19 – Moitas

Utilizando teclado e mouse, o jogador move a raposa e pode mirar e atacar. O desafio está em conseguir sincronizar as cores das moitas com a grama a tempo, sendo que o chão pode alterar para uma cor aleatória após um certo intervalo de tempo. Conforme passam as fases do jogo, há mais moitas para serem destruídas e a frequência com a qual a grama troca de cor aumenta.

A câmera de jogo mostra uma visão de cima do cenário, conhecida pelo termo *top down*. A raposa se move para cima, baixo, esquerda e direita de acordo com as

entradas do teclado ”W”, ”A”, ”S” e ”D”, ou setas direcionais. A posição do mouse define para onde o projétil da folha (Figura 20) será lançado utilizando o botão direito do mouse, e para qual direção a raposa atacará com a espada utilizando o botão esquerdo do mouse. As ações de movimento e ataque são independentes, de modo que o jogador pode andar e atacar ao mesmo tempo.

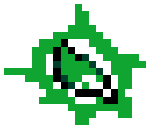

Figura 20 – Folhas atiradas pela raposa

O cenário do jogo é composto por rochas que limitam o espaço para se mover, moitas que precisam ser destruídas, grama atrás mostrando a cor que a moita precisa sincronizar e uma grama mais ao fundo mostrando a fase em que o jogador está (Figura 21). A interface do usuário consiste em apenas dois elementos: um relógio com o tempo restante e um contador mostrando quantas moitas restam no mapa. Por causa de recursos como a fonte de texto de cada motor de jogo, alguns detalhes variam entre as versões do jogo de cada motor.

Figura 21 – Cenário de jogo na versão de *Unity*

O jogo consiste em apenas três fases e, embora, seja o mesmo jogo desenvolvido em três motores diferentes, todas as fases são diferentes, pois as noções de *design* e balanceamento foram se aprimorando durante o aprendizado das ferramentas. Na versão de *Unity*, como uma das mudanças de *design*, a terra utilizada nas rochas do cenário foi reaproveitada para criar um caminho que serve tanto para guiar o jogador, quanto para lançar um desafio extra ao limitar a visão da grama, dificultando a visualização da cor na qual a moita precisa estar.

Além disso, também existem três telas adicionais: menu, tela de vitória e tela de derrota. Isso forma o seguinte fluxo de jogo: iniciando pelo menu, pressionar ”*Play*” muda para a primeira fase. Destruindo todas as moitas do mapa, passa para a segunda fase, em seguida para a terceira e, a completando, a tela de vitória é exibida, onde há apenas um botão para retornar ao menu e reiniciar o fluxo. Caso o tempo acabe em algum nível, a tela de derrota é exibida, de modo que o jogador pode pressionar ”*Try Again*” para repetir a fase perdida ou ”*Menu*” para reiniciar o fluxo de jogo.

4.2 *Godot*

O primeiro motor utilizado nesta pesquisa. Devido a sua leveza em termos de consumo de memória e processamento, criar e abrir um projeto levam apenas poucos segundos, mesmo em computadores menos potentes. Não houve necessidade de configurar nada no motor, assim que carregou a interface, o desenvolvimento pôde iniciar. O editor de código foi o único problema, pois *Godot* possui um editor embutido que reconhece apenas sua linguagem nativa, a *GDScript*. Ainda é possível programar em C# nesse espaço, mas para ter mais suporte de um editor, foi utilizado o *Visual Studio Code*, que reconhece facilmente os arquivos de *Godot*.

4.2.1 *Nodes*

Abrindo o projeto no motor de jogo pela primeira vez, o desenvolvedor se depara com uma cena vazia com uma opção para criar um ”nó” raiz (Figura 22). ”Nós” ou ”*nodes*”, em *Godot*, são os objetos de jogo utilizados para montar as cenas e o jogo como um todo. Cada ”nó” possui um tipo, que corresponde à sua função, podendo ser exibir imagens, animações, elementos da UI, reproduzir sons, definir um raio de colisão e muito mais. Além disso, ”nós” podem ser transformados em cenas de jogo e reutilizados em outras cenas, como um modo de utilizar várias vezes o mesmo objeto. E cada um só pode ter um *script* associado.

Figura 22 – *Godot* - criação do ”nós” raiz

Apesar de possuírem vários tipos, os *nodes* se dividem em três principais: 2D, 3D e *Control*. Os demais tipos, exceto por alguns poucos, herdam de algum deles, possuindo todas as suas características, métodos e variáveis. Mesmo assim, é comum esses ”nós” principais serem utilizados para agrupar os outros. Para criar um *node*, basta selecionar o ícone de mais no canto superior esquerdo ou clicar com o botão direito do mouse em um outro ”nó” para que o novo já seja instanciado como filho do existente.

4.2.2 Movimento do jogador

O primeiro *node* criado, foi do tipo ”*KinematicBody2D*” para ser o objeto principal do jogador. Esse ”nó” herda de ”*Node2D*” e ”*CollisionObject2D*”, portanto suas configurações também estão disponíveis em seu inspetor (Figura 23). O nome do ”nó” foi alterado para ”*Player*” e o *script* mostrado no *Snippet* 1 foi criado e associado a ele.

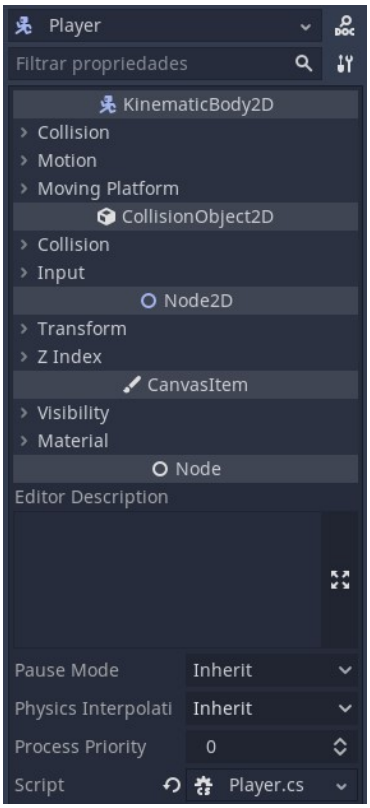

Figura 23 – *Godot - Node KinematicBody2D*

```
1 public class Player : KinematicBody2D
2 {
3     [Export] public int Speed = 200;
4
5     public override void _Process(float delta) {
6         MoveInput();
7     }
8     private void MoveInput() {
```

```
9         Vector2 velocity = Vector2.Zero;
10        if (Input.IsActionPressed("right")) velocity.x += 1;
11        if (Input.IsActionPressed("left")) velocity.x -= 1;
12        if (Input.IsActionPressed("up")) velocity.y -= 1;
13        if (Input.IsActionPressed("down")) velocity.y += 1;
14        if (velocity != Vector2.Zero) velocity = velocity.Normalized();
15        MoveAndSlide(velocity * Speed);
16    }
17 }
```

Snippet 1 – Código de movimentação do jogador

O código possui o método ”*Process*”, que é executado a cada atualização de quadro do jogo e nele é chamado o método ”*MoveInput*”, que altera o vetor de velocidade de acordo com a entrada do usuário. Em seguida, o movimento do ”*KinematicBody2D*” é realizado através do seu método ”*MoveAndSlide*”, que recebe um vetor e realiza o movimento com a força e direção dele. Esse método também faz o objeto deslizar caso colida com algo, de modo que seu movimento não seja interrompido. Por padrão, a entrada do usuário para realizar as ações de cima, baixo, esquerda e direita são as teclas direcionais, mas é possível alterar ou adicionar novas teclas pela configuração ”Mapa de Entrada” na janela de ”Configurações do Projeto”.

Para dar um imagem ao *node* ”*Player*”, um ”*Sprite*” foi adicionado como filho. E para reutilizá-lo nos diferentes níveis do jogo, seu ”nó” foi transformado em uma cena de jogo.

4.2.3 Colisão entre objetos

Implementar colisão entre ”nós” é um processo simples, bastando colocar um *node* ”*CollisionObject2D*” com um filho ”*CollisionShape2D*” para dar forma ao colisor. Isso garante que o objeto esteja pronto para colidir com outros, mas para permitir que a colisão ocorra, é necessário para cada ”nó” configurar a ”*Layer*” a qual o *node* pertence e a ”*Mask*”, que se refere à camada que ele está detectando (Figura 24). Por exemplo: um ”nó” da ”*Layer*” 1, com ”*Mask*” 2, irá colidir com ”nós” da segunda camada e de nenhuma outra. É possível configurar para que um objeto pertença a mais de um ”*Layer*” e detecte múltiplas camadas em sua ”*Mask*”, incluindo sua própria.

O *node* ”*Player*” já herda de ”*CollisionObject2D*”, apenas adicionar um ”*CollisionShape2D*” garante que sua colisão esteja pronta. Só é necessário adicionar ”nós” adicionais e prestar atenção nas configurações de colisão de cada um.

Figura 24 – *Godot* - configurações do *CollisionObject2D*

4.2.4 Animação

Em *Godot*, animações são reproduzidas através de dois *nodes*. ”*AnimationPlayer*” cria as animações que serão reproduzidas, enquanto o ”*AnimationTree*” controla as transições entre elas. Cada animação consiste em alterar configurações dos ”nós” em cena. Para animar a raposa, por exemplo, é necessário separar os quadros de animação em seu ”nó” ”*Sprite*”, definindo a quantidade de *frames* que ela possui.

Em seguida, clicar no ”*AnimationPlayer*” abre uma nova aba, onde as animações são criadas (Figura 25). Enquanto essa aba estiver aberta, um ícone de chave aparece ao lado de todas as configurações dos *nodes*. Ele é usado para criar uma faixa de animação para controlar a propriedade correspondente. Assim, é possível adicionar a configuração ”*Frame*” de ”*Sprite*” para realizar a troca de quadros para cada animação.

Figura 25 – *Godot* - janela ”Animação”

Passando para o ”*AnimationTree*”, seu *node* funciona como uma máquina de estados. Através da aba ”Árvore de Animação”, que é aberta ao utilizar esse ”nó”, é possível criar novos estados com animações (Figura 26). Cada estado pode corresponder a uma ou mais animações, caso utilize uma estrutura como *BlendSpace2D*, que utiliza duas variáveis para controlar as animações presentes nela. A raposa utiliza três estruturas desse tipo, sendo cada uma com quatro animações correspondendo aos quatro possíveis lados de realizar as ações ”*Idle*”, ”*Run*” e ”*Attack*”.

Realizar uma transição de animação, necessita de um código como o apresen-

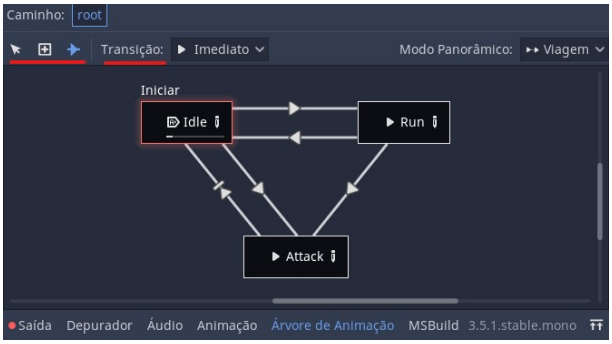

Figura 26 – *Godot* - janela ”Árvore de Animação”

tado no *Snippet* 2, onde o *node* ”*AnimationTree*” do ”*Player*” é utilizado para alterar os valores das variáveis dos *BlendSpaces*. E através do método ”*Travel*” as transições entre estados são realizadas.

```
1  public class Player : KinematicBody2D
2  {
3    [...]
4    AnimationTree animTree = new AnimationTree();
5    AnimationNodeStateMachinePlayback animState;
6
7    public override void _Ready() {
8      animTree = GetNode<AnimationTree>("AnimationTree");
9      animState = (AnimationNodeStateMachinePlayback)animTree.Get("parameters
       /playback");
10   }
11   [...]
12   private void MoveInput() {
13     [...]
14     if (velocity != Vector2.Zero) {
15       animTree.Set("parameters/Idle/blend_position", velocity);
16       animTree.Set("parameters/Run/blend_position", velocity);
17       velocity = velocity.Normalized();
18       animState.Travel("Run");
19     } else {
20       animState.Travel("Idle");
21     }
22     MoveAndSlide(velocity * Speed);
23   }
24 }
```

Snippet 2 – Código do jogador com animações

4.2.5 *TileMap*

O *node* ”*TileMap*” é utilizado para criar cenários de jogo usando *tiles*, ou seja, pequenos blocos com pedaços do mapa. Mas para isso, precisa de um ”*TileSet*”, que

é um conjunto de *tiles* pré-definidos usados para construir o ambiente de jogo no "*TileMap*". Em suas propriedades, há uma opção para criar um novo "*TileSet*". Isso abre uma nova janela, onde pode ser colocado um arquivo de imagem com o conjunto de "*tiles*" (Figura 27).

Em seguida, dentre as opções disponíveis, neste projeto foi utilizado um *Autotile*, pois serve para desenhar o mapa escolhendo automaticamente os *tiles* a partir de regras declaradas. Esse processo cria o "*TileSet*", mas para poder usá-lo corretamente, é preciso configurar o modo da *bitmask* para corresponder ao padrão da imagem utilizada.

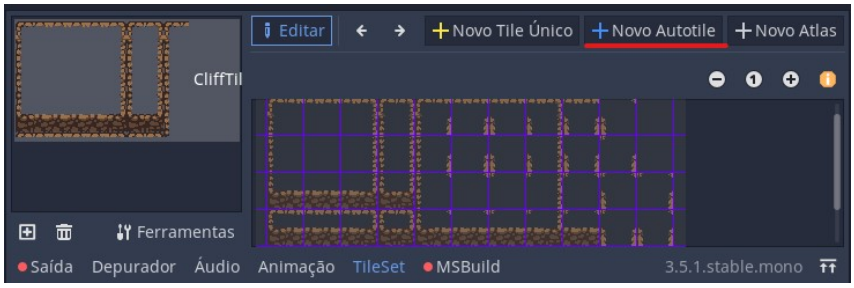

Figura 27 – *Godot* - janela do "*TileSet*"

Para os dois conjuntos de *tiles* desse projeto o modo ideal é "3x3 *(minimal)*". Isso permite ir na aba "*Bitmask*" e desenhar como deve ser aplicada a regra do *autotile* (Figura 28). Funciona como uma máscara 3x3 para cada *tile*, onde cada quadrado vermelho corresponde a onde deve haver mapa para que o *tile* central seja desenhado. Outra configuração realizada para apenas um dos "*TileSets*" foi para adicionar colisão. O processo é similar a desenhar a *bitmask*, basta navegar para a aba "Colisão" e escolher os *tiles* que terão colisão, porém é um processo lento, pois precisa selecionar cada *tile* separadamente.

Figura 28 – *Godot* - janela do *TileSet* com a *Bitmask*

4.2.6 *Signals*

Godot possui um recurso para criar interações entre *nodes* chamado "*Signals*", ou "Sinais". Todos os "nós", por padrão, já possuem alguns sinais implementados. Eles

são disparados quando uma certa ação ocorre e invocam um método que esteja conectado a eles. Para visualizar os *signals* de um *node*, basta selecionar a aba ”Nó” ao lado do ”Inspetor”.

O *node* ”*Sprite*”, por exemplo, possui dois sinais (Figura 29). Eles são disparados quando houver uma troca de *frame* ou textura. Clicando duas vezes em um sinal, é aberta uma janela de configuração que permite conectá-lo a um método de um *node* com *script* (Figura 30). O nome do método precisa ser inserido manualmente, mas pode ser implementado posteriormente.

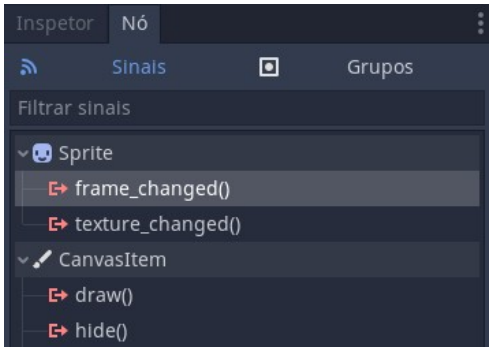

Figura 29 – *Godot* - sinais do ”nó” ”*Sprite*”

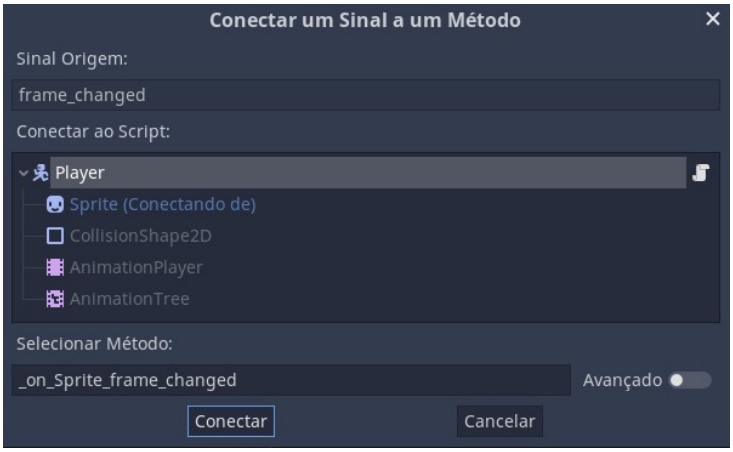

Figura 30 – *Godot* - janela para conectar um sinal a um método

Sinais também podem ter parâmetros, chamar métodos com parâmetros iguais aos seus e também é possível criá-los e conectá-los por código. O *script* apresentado no *Snippet* 3, trás um exemplo de criação e conexão de um sinal utilizando código, onde basta declarar o sinal e chamar o método ”*Connect*” com o nome do sinal, do *node* alvo e seu método.

```
1 public class Teste : Node2D
2 {
3   [Signal]
4   public delegate void SignalName();
5
```

```
6       public override void _Ready() {
7           this.Connect("SignalName", this, "OnSignalActivation");
8       }
9       public void OnSignalActivation() {
10          GD.Print("O SINAL FOI ATIVADO");
11      }
12  }
```

Snippet 3 – Código de exemplo da implementação de sinais

4.2.7 Ataques do jogador

Para implementar os ataques do jogador, foi utilizado um conceito de *hurtboxes* e *hitboxes*. São cenas de jogo criadas com um *node* ”*Area2D*”, que usa uma área para detectar colisores que entrarem nela, e um ”*CollisionShape2D*” sem forma. Isso agiliza um pouco o desenvolvimento, pois criando uma dessas cenas, basta adicionar uma forma para ter uma área de colisão pronta.

O ataque com espada do jogador é implementado basicamente utilizando apenas de animações. Foi utilizado uma *hitbox* para detectar as colisões e um *node* ”*Position2D*” para definir sua posição. Em cada animação de ataque, a posição e rotação do *node* foram ajustados e o ”*CollisionShape2D*” foi habilitado e desabilitado para corresponder à animação da espada (Figura 31). E no código do jogador, bastou utilizar o ”*AnimationTree*” para transicionar para o estado de ataque.

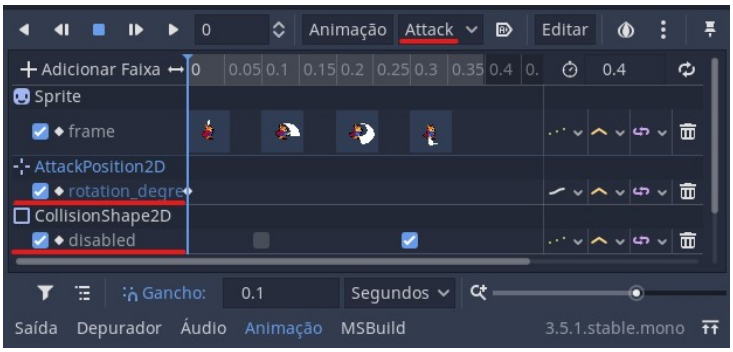

Figura 31 – *Godot* - animação controlando o ataque com espada

A implementação do projétil do jogador foi realizada criando uma cena com uma *hitbox* como base, mais uma *sprite* e o *script* mostrado no *Snippet* 4 para mover o objeto após ele ser instanciado. Em seguida, bastou conectar o sinal ”*body_entered*” da ”*Area2D*” com o método ”*OnAreaEntered*” do *script* para destruir o projétil quando ele colidir com algo. Para instanciar essa cena, entretanto, foi necessário alterar o código do jogador para pegar a referência do projétil, instanciá-lo e ajustar sua posição e rotação, como também visto no *Snippet* 4.

```
1 public class Projectile : Area2D
```

```
{
  [Export] public int ProjectileSpeed = 250;

  public override void _Process(float delta) {
    Vector2 direction = Vector2.Right.Rotated(Rotation);
    GlobalPosition += direction * ProjectileSpeed * delta;
  }
  public void OnAreaEntered(Node body) {
    QueueFree();
  }
}

public class Player : KinematicBody2D
{
  [...]
  PackedScene projectile;

  public override void _Ready() {
        [...]
    projectile = GD.Load<PackedScene>("res://Player/LeafProjectile.tscn");
  }
  [...]
  private void AttackInput() {
    if (Input.IsActionJustPressed("attack")) {
      animTree.Set("parameters/Attack/blend_position", GlobalPosition.
    DirectionTo(GetGlobalMousePosition()));
      animState.Travel("Attack");
    }
    if (Input.IsActionJustPressed("attack2")) {
      var instance = projectile.Instance();
      this.GetParent().AddChild(instance);
      Node2D leaf = (Node2D)instance;
      leaf.GlobalPosition = GlobalPosition;
      float leafRotation = GlobalPosition.DirectionTo(
    GetGlobalMousePosition()).Angle();
      leaf.Rotation = leafRotation;
    }
  }
}
```

Snippet 4 – Código do projétil e alteração no do jogador

4.2.8 Moita

As moitas são objetos estáticos e possuem um *sprite*, *hurtbox* e um *node* "*AnimatedSprite*", que é utilizado para reproduzir animações quando não há muitas e sem a necessidade de criar transições. Além disso, seu *script*, mostrado no *Snippet* 5, pos-

sui *arrays* com as diferentes texturas e nomes das animações dos *sprites* de cada cor da moita.

Seu método para detectar colisões verifica qual o objeto encontrado: espada, grama de fundo ou projétil. Se for a espada, a moita é destruída caso esteja na mesma cor da grama, que só é detectada uma vez no início da cena. Nesse caso, o sinal para troca de textura da grama é conectado a um método que troca a cor alvo da moita. E se detectar o projétil, a cor atual é trocada para a próxima do *array* de texturas.

```
public class Bush : Node2D
{
  AnimatedSprite effect; Sprite sprite;
  Texture[] bushColors = {null, null, null, null, null};
  String[] bushAnimations = {"white", "green", "purple", "red", "yellow"};
  int currentColor;
  public int currentWorldColor = 0;
  DynamicBackground currentBg;

  public override void _Ready() {
    effect = GetNode<AnimatedSprite>("AnimatedSprite");
    sprite = GetNode<Sprite>("Sprite");
        // Repeat for every color:
    bushColors[0] = GD.Load<Texture>("res://.import/Bush_color.png-<id>.
   stex");
        [...]
    currentColor = (int)(GD.Randi() % 5);
    sprite.Texture = bushColors[currentColor];
  }
  public void OnBushAreaEntered(Area2D area) {
    if (area.CollisionLayer == 4) {
      if (currentColor == currentWorldColor) {
        OnBushDestroyed();
      }
    } else if (area.CollisionLayer == 64) {
      currentBg = area.GetNode<DynamicBackground>("..");
      Sprite bgSprite = area.GetNode<Sprite>("../Sprite");
      bgSprite.Connect("texture_changed", this, "OnBgTextureChanged");
      currentWorldColor = currentBg.currentColor;
    } else {
      currentColor++;
      if (currentColor >= bushColors.GetLength(0)) {
        currentColor = 0;
      }
      sprite.Texture = bushColors[currentColor];
    }
  }
  [...]
```

```
}
```

Snippet 5 – Código da moita

4.2.9 Grama

A grama de fundo possui um *sprite*, uma área de colisão e um *timer*, que é um *node* com temporizador. Seu *script*, apresentado no *Snippet* 6, consiste em escolher uma cor aleatória a partir do *array* de texturas, conectar o *timer* a um método para trocar a cor da grama e iniciar o cronômetro. Sempre que o tempo definido acabar, um sinal é disparado, uma cor aleatória é definida e o temporizador reseta. Pela IDE, o desenvolvedor pode ajustar o tempo e a cor inicial da grama.

```
public class DynamicBackground : Node2D
{
  [Export]
  public float _WaitTime = 5;
  [Export]
  public int currentColor = -1;
  Sprite bgSprite;
  Texture[] bgColors = { null, null, null, null, null };
  Timer clock;

  public override void _Ready() {
    bgSprite = GetNode<Sprite>("Sprite");
    clock = GetNode<Timer>("Clock");
    [...]
    if (currentColor < 0) currentColor = (int)(GD.Randi() % 5);
    if (_WaitTime > 0) {
      clock.Connect("timeout", this, "ChangeColor");
      clock.WaitTime = _WaitTime;
      clock.Start();
    }
    bgSprite.Texture = bgColors[currentColor];
  }
  public void ChangeColor() {
    int newColor = (int)(GD.Randi() % 5);
    if (newColor == currentColor) newColor++;
    if (newColor >= bgColors.GetLength(0)) newColor = 0;
    currentColor = newColor;
    bgSprite.Texture = bgColors[currentColor];
    clock.WaitTime = _WaitTime;
    clock.Start();
  }
}
```

Snippet 6 – Código da grama

4.2.10 Câmera de jogo

Godot não precisa de um *node* de câmera para renderizar a cena de jogo, mas para ter mais controle sobre a visão do jogador, é possível utilizar um ”nó” ”*Camera2D*”. Isso permite selecioná-lo para ser a câmera atual de jogo, customizar seu movimento e, utilizando um *node* ”*RemoteTransform2D*”, fazê-la seguir o jogador. Esse outro ”nó”, precisa ser filho do objeto que se deseja seguir e dentro de suas propriedades, é necessário definir o *node* da câmera no campo ”*Remote Path*”.

4.2.11 Música e efeitos sonoros

Reprodução de sons em *Godot* é realizada utilizando um *node* ”*AudioStreamPlayer*” para cada som do jogo. Com ele, é possível referenciar um arquivo de áudio e alterar algumas configurações de reprodução, como se ela deverá ocorrer assim que a cena for carregada (Figura 32). Nas propriedades de cada arquivo também é possível ajustar configurações mais específicas, como ativar a reprodução em *loop*. E para reproduzir um efeito sonoro utilizando código, basta chamar o método ”*Play*” do *node* correspondente.

Figura 32 – *Godot* - configurações do ”*AudioStreamPlayer*”

4.2.12 Interface do usuário (UI)

Todos os elementos de UI de *Godot* herdam do tipo de *node* ”*Control*”. Para mostrar as informações da fase para o jogador, foi criada um cena com *nodes* para o relógio, o contador de moitas e um marcador de nível, criado apenas nesta versão do jogo. Cada elemento possuindo uma imagem e um texto, exceto no caso do relógio, onde também foi usado um *timer* e um *script*, mostrado no *Snippet* 7, que inicia o tempo e o converte em um texto formatado para ser exibido na UI.

```
public class LevelClock : Control
{
  [Export]
  public int _TotalTime = 70;
  Label timeLabel;
  Timer timer;

  public override void _Ready() {
    timeLabel = GetNode<Label>("TimeLabel");
    timer = GetNode<Timer>("Timer");
    timer.WaitTime = _TotalTime;
    UpdateTimeText();
    timer.Start();
  }
  [...]
  void UpdateTimeText() {
    float min = (int)timer.TimeLeft / 60;
    float sec = (int)timer.TimeLeft % 60;
    String minutes = "";
    String seconds = "";
    if (min < 10) minutes = "0";
    if (sec < 10) seconds = "0";
    minutes += min;
    seconds += sec;
    timeLabel.Text = (minutes + ":" + seconds);
  }
}
```

Snippet 7 – Código do relógio da UI

Além desses elementos, foram utilizados botões nas cenas de menu, vitória e derrota. Para usá-los, basta conectar seu sinal, que dispara quando for pressionado, à ação que deseja executar. Neste projeto, todos os botões foram conectados a um *node* de áudio, que por sua vez, foram conectados a métodos para trocar de cena quando o efeito sonoro terminar.

Elementos da interface do usuário permanecem estáticos onde forem posicionados. Para que sigam a câmera de jogo, é necessário utilizar um *node* "*CanvasLayer*", que cria um camada para os elementos de UI sobre a câmera de jogo. Dessa forma, qualquer objeto que estiver como filho do "*CanvasLayer*" irá seguir o movimento da câmera.

4.2.13 Fluxo de jogo

O fluxo do jogo é gerenciado por um *node* com um *script* chamado "*GameController*", usado para instanciar novas cenas, atualizar o contador de moitas da UI

e registrar o progresso do jogador, marcando quantas moitas existem no nível para passar de nível quando chegar em zero.

Em *Godot*, é necessário que sempre exista uma cena de jogo, por isso existe uma cena principal apenas com o ”*GameController*” e um *node* para a música de jogo, de modo que todas as telas são carregadas como filhos dessa cena. O código do *Snippet* 8, mostra apenas um pedaço do controlador do jogo para exemplificar como ocorre o carregamento de uma cena. Cada uma tem sua referência armazenada em uma ”*PackedScene*”, a atual é salva na variável ”*currentScene*” e toda vez que uma cena precisar ser carregada, o método ”*LoadScene*” é utilizado para destruir a antiga, carregar a nova e trocar a referência da cena atual.

```
1  public class GameController : Node2D
2  {
3    PackedScene menuScene;
4    Node currentScene;
5      [...]
6    public override void _Ready() {
7      menuScene = GD.Load<PackedScene>("res://UI/Menu.tscn");
8          [...]
9    }
10   void LoadScene(PackedScene sc) {
11     currentScene.QueueFree();
12     var instance = sc.Instance();
13     AddChild(instance);
14     currentScene = GetChild(2);
15   }
16     [...]
17 }
```

Snippet 8 – Código do ”*GameController*”

4.2.14 Exportação de projeto

O primeiro passo para exportar um projeto, é baixar os *templates* de exportação de *Godot*. Isso pode ser realizado acessando a janela ”Gerenciar Modelos de Exportação” da Figura 33, onde há um botão para baixar e instalar todos os modelos disponíveis.

Para publicar o jogo no site *itch.io*, que foi a plataforma alvo utilizada neste trabalho, é necessário subir um arquivo de exportação no modelo ”HTML5” para que o jogo possa ser executado no navegador. Pela opção ”Exportar”, a janela de exportação da Figura 34 é aberta, onde o desenvolvedor pode adicionar uma nova predefinição para um dos modelos disponíveis e configurá-la como desejar. Em seguida, basta selecionar a opção ”Exportar Projeto” e escolher um local para salvar os arquivos.

Figura 33 – *Godot* - janela ”Gerenciador de Modelos de Exportação”

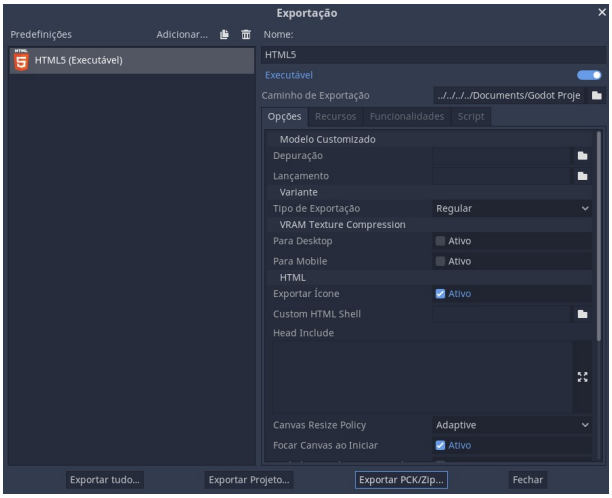

Figura 34 – *Godot* - janela ”Exportação”

Especificamente para o *itch.io*, é necessário que o arquivo ”html” do jogo tenha o nome ”index.html”. Além disso, todos os arquivos devem estar em uma pasta compactada do tipo ”zip”. Após ajeitar esses detalhes nos arquivos do projeto, ele está pronto para ser publicado [1].

4.3 *GameMaker*

Recriar o mesmo jogo com *GameMaker* foi uma experiência mais rápida, pois a ideia já estava pronta, de modo que os pontos mais demorados foram apenas para aprender alguns conceitos e mudar o raciocínio de algumas implementações. Não houve necessidade de nenhuma configuração prévia no motor, porque o ambiente de desenvolvimento já é bem completo, possuindo, inclusive, um editor de *sprites* próprio.

O projeto foi desenvolvido utilizando código na própria IDE do motor. Mas bastante da sua lógica ainda utiliza elementos de programação visual. E, no início do desenvolvimento, não foram utilizados os mesmos *assets* do projeto em *Godot*, por isso, alguns elementos possuem nomes que não correspondem ao objeto real.

[1] <https://zub40.itch.io/fox-vs-plants-godot-version>

4.3.1 Organização geral do projeto

As pastas de um projeto em *GameMaker* já possuem uma organização própria, sendo divididas pelo tipo do arquivo. Além disso, o motor reconhece cada arquivo pelo prefixo em seu nome. *Sprites* por exemplo, devem começar com ”spr_”.

A área de trabalho de *GameMaker* é caracterizada por um *workspace*, onde os elementos de jogo são configurados, e cenas de jogo, conhecidas como *rooms*. E, recursos mais específicos, como um *sprite*, são abertos em uma aba separada para serem editados.

4.3.2 Objetos e eventos

Os objetos de jogo em *GameMaker* não possuem um tipo específico, todos são apenas ”objetos”, mas possuem várias configurações. Podem receber um *sprite*, variáveis e eventos, que são basicamente métodos chamados em momentos específicos. O evento ”*Create*”, por exemplo, ocorre quando a cena de jogo é criada, e o evento ”*Step*”, toda vez que há uma atualização de quadro.

4.3.3 Movimento do jogador

Criando um objeto ”*Player*”, uma janela com suas configurações é aberta no espaço de trabalho (Figura 35). Adicionando um evento ”*Step*”, é possível programar o movimento do jogador utilizando funções para receber entradas do teclado, como mostrado no *Snippet* 9. Não há teclas específicas para nenhuma ação em *GameMaker*, sendo necessário escolher exatamente as que se deseja utilizar. Após, receber os valores de entrada, basta ajustar cada um para corresponder ao lado que o jogador irá se mover, adicionar um valor de velocidade e chamar a função ”*move_and_collide*”, que também recebe um objeto para a raposa colidir. Neste caso, ”obj_box”, que se refere às moitas do jogo.

```
1  var _hor0 = keyboard_check(ord("D"))-keyboard_check(ord("A"));
2  var _hor = keyboard_check(vk_right)-keyboard_check(vk_left);
3  var _ver0 = keyboard_check(ord("S"))-keyboard_check(ord("W"));
4  var _ver = keyboard_check(vk_down)-keyboard_check(vk_up);
5
6  if (abs(_hor) < abs(_hor0)) _hor = _hor0;
7  if (abs(_ver) < abs(_ver0)) _ver = _ver0;
8
9  var _speed = 2;
10 var _xvec = _hor * _speed;
11 var _yvec = _ver * _speed;
12
```

```
13 move_and_collide(_xvec, _yvec, obj_box);
```

Snippet 9 – Código do evento ”*Step*” do objeto ”obj_player”

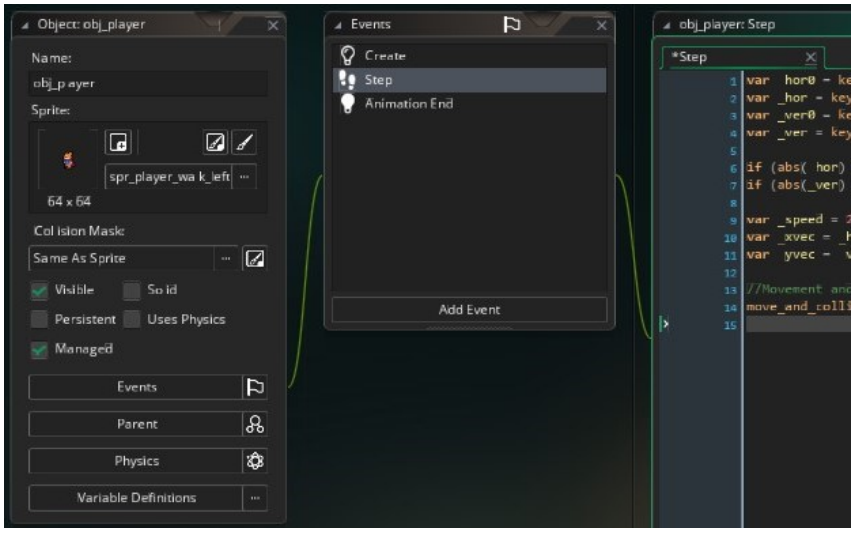

Figura 35 – *GameMaker* - ”obj_player” e seu evento ”*Step*”

4.3.4 Ataques do jogador

Os objetos de ataques do jogador são instanciados em seu evento ”*Step*”, mostrado no *Snippet* 10. Em seus próprios eventos, o objeto do projétil foi programado para, após criado, se mover até a posição do cursor do mouse. E o objeto da espada, é instanciado em uma posição específica correspondente ao lado mais próximo do mouse, e seu código apenas o destrói após passar um segundo.

```
1  [...]
2  if mouse_check_button_pressed(mb_right) {
3    instance_create_layer(x, y, "Instances", obj_projectile);
4  }
5  if mouse_check_button_pressed(mb_left) {
6    var _angle;
7    var _xval = mouse_x - x;
8    var _yval = mouse_y - y;
9    [...]
10   var _atk = {
11     image_angle : image_angle + _angle
12   };
13   instance_create_layer(_xval, _yval, "Instances", obj_attack, _atk);
14 }
```

Snippet 10 – Código do evento ”*Step*” do objeto ”obj_player” com instanciação dos ataques

Além disso, *GameMaker* permite definir o formato dos colisores a partir do *sprite* de cada objeto. E para lidar com colisões, a um evento específico que pode ser criado

para colidir com cada objeto de jogo. No caso dos ataques do jogador, ambos possuem eventos para serem destruídos ao colidir com uma moita.

4.3.5 Moita

O objeto para as moitas, chamado ”obj_box”, é controlado através de cinco eventos. O primeiro é o evento ”*Create*”, que define uma cor aleatória para ela a partir de um *array* global de cores. O segundo é a colisão com a grama para definir sua cor alvo igual à cor que estiver abaixo. O seguinte lida com a colisão com um projétil, trocando a cor da moita. O quarto é chamado ao colidir com a espada e verifica se está na cor certa para ser destruída. E o último, destrói o objeto após concluir a animação de destruição.

```
// Create event
sprite_counter = random(5);
destroy_color = 0;
box_bg = pointer_null;
image_blend = global.colors[sprite_counter];

// Collision with background event
box_bg = other;
destroy_color = box_bg.image_blend;

// Collision with obj_projectile event
instance_destroy(other);
sprite_counter++;
if sprite_counter >= array_length(global.colors) {
  sprite_counter = 0;
}
image_blend = global.colors[sprite_counter];

// Collision with obj_attack event
if image_blend == box_bg.image_blend {
  sprite_index = spr_forest_bush_destroyed;
}

// Animation end
if (sprite_index == spr_forest_bush_destroyed) {
  instance_destroy();
}
```

Snippet 11 – Código dos eventos do objeto ”obj_box”

4.3.6 Grama

A implementação da grama se trata apenas de trocar sua cor de acordo com um intervalo de tempo pelo seu evento ”*Step*”. As variáveis com o tempo e a cor da grama foram definidas fora de seus eventos, na configuração ”*Variable Definitions*” do objeto (Figura 36). Isso permite alterar seus valores dentro de uma cena de jogo para customizar seu comportamento em cada local.

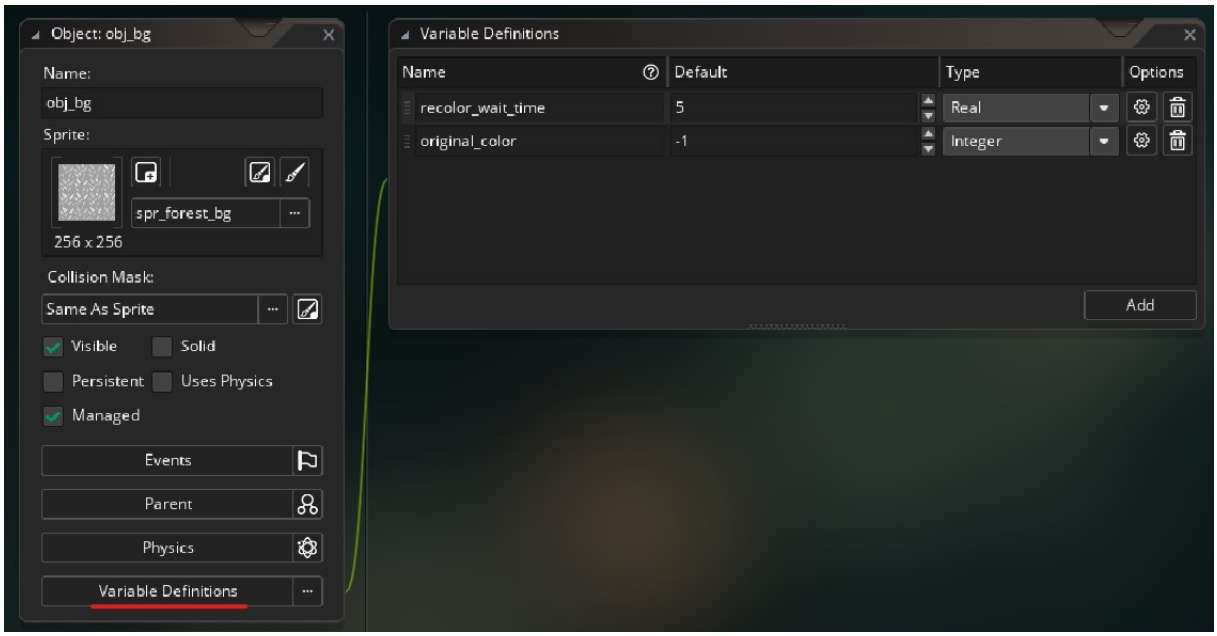

Figura 36 – *GameMaker* - variáveis definidas de ”obj_bg”

4.3.7 *TileSet*

Criando um novo ”*TileSet*” em *GameMaker*, suas configurações permitem adicionar uma imagem com o conjunto de *tiles* (Figura 37). É possível ajustar suas propriedades para que cada *tile* seja bem delimitado. Além disso, a janela de um ”*TileSet*” possui três editores: ”*Brush Builder*” para construir um padrão com um certo conjunto dos *tiles*; ”*Tile Animation*” para utilizar *sprites* com animação; e ”*Auto Tiling*” para construir um padrão automático de *tiles*.

Assim como no projeto em *Godot*, foram criados *autotiles* para os dois ”*TileSets*” do projeto. Isso abre a janela da Figura 38, onde pode ser selecionado um padrão de 47 ou 16 *tiles*. Em seguida, para cada padrão do *autotile*, foi escolhida uma imagem correspondente.

O conjunto de *tiles* das rochas precisa de colisão, porém isso não é tão simples de implementar em *GameMaker*. Foi necessário alterar o código de movimento do jogador, como mostrado no *Snippet* 12, para detectar se a região onde ele pretende se mover está preenchida na camada de *tiles* da cena de jogo. Isso é realizado, utilizando

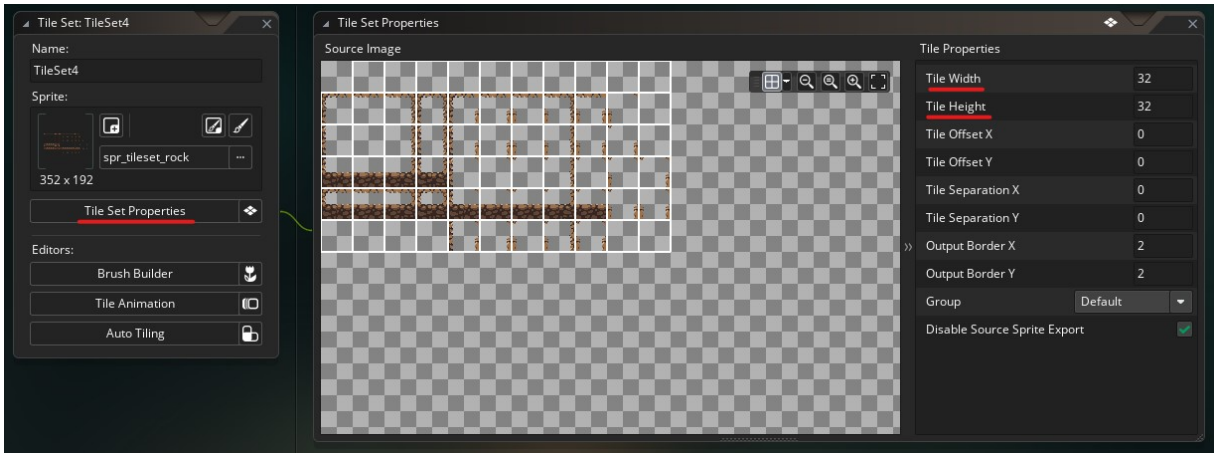

Figura 37 – *GameMaker* - ”*Tile Set*” e suas propriedades

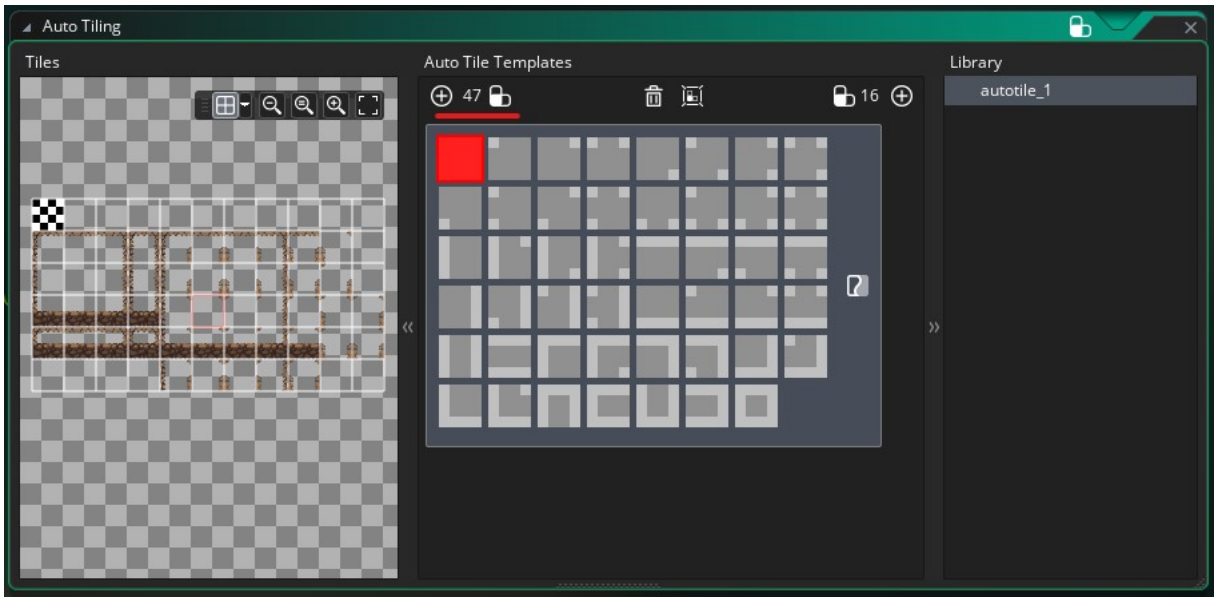

Figura 38 – *GameMaker* - configurações de ”*Autotile*”

as bordas do ”obj_player” em conjunto com uma função que detecta *tiles* em uma posição específica de uma dada camada. Caso haja cenário nessa posição, o movimento é impedido.

```
1  [...]
2  var _bbox_side;
3
4  if (_xvec > 0)
5      _bbox_side = bbox_right;
6  else
7      _bbox_side = bbox_left;
8  if (tilemap_get_at_pixel("Tiles_1",_bbox_side+_xvec,bbox_top) != 0) || (
       tilemap_get_at_pixel("Tiles_1", _bbox_side+_xvec,bbox_bottom) != 0) {
9    _xvec = 0;
10 }
11 if (_yvec > 0)
```

```
12      _bbox_side = bbox_bottom;
13  else
14      _bbox_side = bbox_top;
15  if (tilemap_get_at_pixel("Tiles_1",bbox_left,_bbox_side+_yvec) != 0) || (
        tilemap_get_at_pixel("Tiles_1",bbox_right,_bbox_side+_yvec) != 0) {
16    _yvec = 0;
17  }
18  move_and_collide(_xvec, _yvec, obj_box);
19  [...]
```

Snippet 12 – Código adaptado do evento ”*Step*” de ”obj_player”

4.3.8 Câmera de jogo

As *rooms* de *GameMaker* possuem suas próprias configurações de visualização. É possível ajustar o tamanho da cena e ativar ”*Viewports*”, que funcionam como câmeras de jogo (Figura 39). As cenas de menu, vitória e derrota têm apenas o seu tamanho ajustado, mas as dos níveis do jogo possuem ”*Viewports*”.

Utilizar tal recurso permite ajustar as configurações de visualização da câmera (Figura 40). Primeiro, é importante deixar a *viewport* visível para que ela seja utilizada. As demais configurações ajustam seu tamanho e controlam seu movimento. Para determinar um objeto para seguir, como deve acontecer com o jogador, basta selecioná-lo na área ”*Object Following*”.

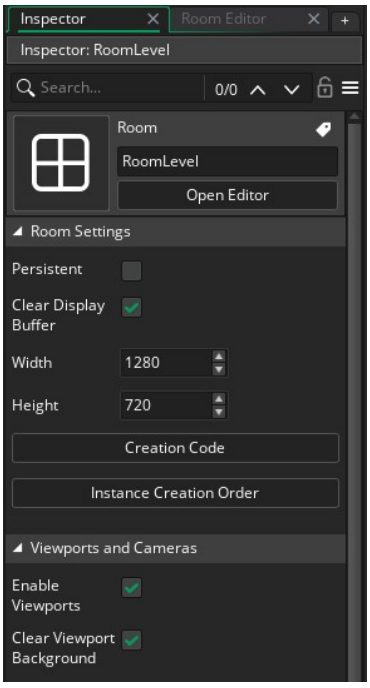

Figura 39 – *GameMaker* - inspetor de ”*RoomLevel*”

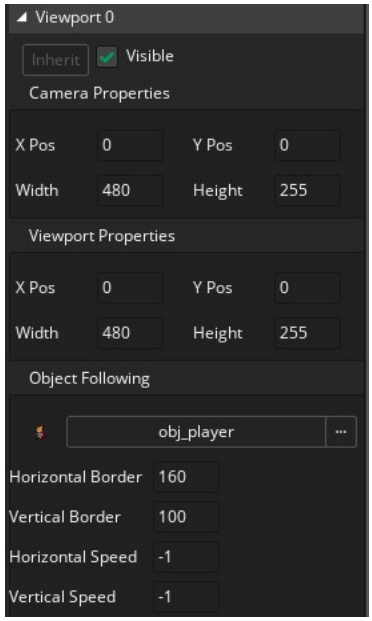

Figura 40 – *GameMaker* - configurações do "*Viewport 0*" de "*RoomLevel*"

4.3.9 Interface do usuário (UI)

Os elementos de UI precisam de dois eventos específicos para serem renderizados corretamente: "*Draw*" e "*DrawGUI*". O primeiro, não precisa de nenhum código, mas é necessário criar o evento para sobrescrever o comando de renderização padrão. O segundo, utiliza o método "*draw_self*" para renderizar o elemento na camada de UI, de modo que esteja sempre acompanhando o *viewport* principal da cena.

Os dois elementos de UI utilizados possuem estruturas de eventos semelhantes: uma função que atualiza o texto exibido e outra que troca de cena quando a condição de vitória ou derrota for alcançada. Ambas as condições são verificadas durante o evento "*Step*" de cada elemento. Porém, o relógio do jogo também registra a passagem de tempo durante esse evento. O contador de moitas, mostrado no *Snippet* 13, utiliza uma função que conta a quantidade de objetos de um determinado tipo na cena.

```
1  // Create event
2  total_boxes = instance_number(obj_box);
3  box_counter = total_boxes;
4
5  function update_counter(){
6    var _text = "";
7    box_counter = instance_number(obj_box);
8    _text += string(box_counter);
9    _text += "/";
10   _text += string(total_boxes);
11   return _text;
12 }
13 function win() {
14   room_goto_next();
```

```
}
// Draw GUI event
draw_self();
draw_set_font(fnt_button);
draw_set_halign(fa_center);
draw_set_valign(fa_middle);
draw_set_color(c_white);
draw_text(x, y, update_counter());
```

Snippet 13 – Código dos eventos do objeto ”obj_box_counter”

4.3.10 Animações

Animações em *GameMaker* não possuem um arquivo ou objeto próprio, sendo necessário utilizar *sprites* com múltiplos *frames* e alterar as propriedades do objeto que será animado. Usar ”*sprite_index*”, troca o arquivo de imagem que está sendo utilizado, ”*image_index*” troca o *frame* do *sprite* atual, e ”*image_speed*” a taxa de quadros por segundo.

Controlando essas propriedades, as animações podem ser reproduzidas, mas para gerenciar a transição entre elas, é necessário utilizar variáveis do tipo *boolean* para trabalharem como alterações de estado que definirão qual animação deve ser reproduzida. O código do jogador, exibido no *Snippet* 14, mostra um pouco do controle de animações, onde o evento ”*Create*” declara cada animação e estado, e o evento ”*Animation End*” realiza uma transição para resetar as animações. E todo o gerenciamento das transições de estados e reprodução das animações ocorre no evento ”*Step*”, ocupando boa parte do código do jogador.

```
// Create event
[...]
image_speed = 0;
anim_idle = spr_player_walk_right;
anim_walk_left = spr_player_walk_left;
[...]
anim_atk_left = spr_player_atk_left;
[...]
walking = false;
attacking = false;

// Animation End
image_speed = 0;
sprite_index = anim_idle;
attacking = false;
```

Snippet 14 – Código dos eventos de ”obj_player” adaptado para gerenciar animações

4.3.11 Música e efeitos sonoros

Arquivos de áudio em *GameMaker* possuem o prefixo ”snd_” e as configurações mostradas na Figura 41. Essa imagem, corresponde ao efeito sonoro de um botão, que pode ser reproduzido através da função ”*audio_play_sound(snd_button, 0, false)*” em qualquer código do jogo. O primeiro argumento é o arquivo escolhido, o segundo o canal de som, que por padrão é zero, e o terceiro é uma verificação se o som deve ser reproduzido em *loop* ou não.

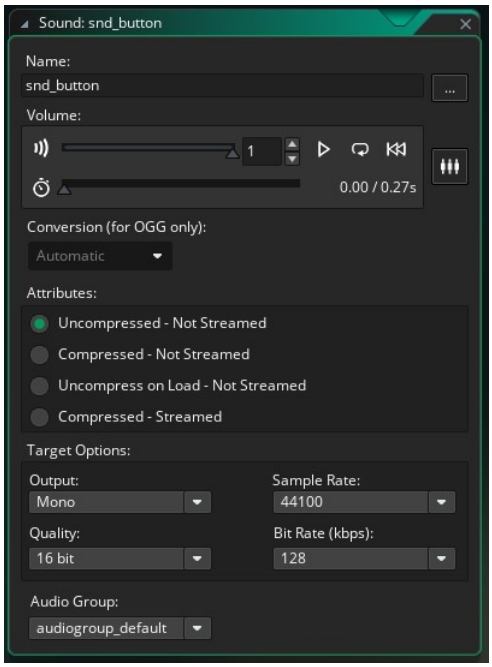

Figura 41 – *GameMaker* - configurações do arquivo ”snd_button”

4.3.12 Fluxo de jogo

As *rooms* do jogo podem ser gerenciadas pela janela ”*Room Manager*”, que pode ser aberto clicando ao lado esquerdo do arquivo de alguma cena (Figura 42). Nesse espaço, é possível reordenar as *rooms* para escolher qual é inicial e planejar uma ordem para as trocas de cena. Também é possível duplicar uma *room* ou criar heranças entre elas.

As transições entre as telas do jogo ocorrem de acordo com a ordem estabelecida no ”*Room Manager*”. As fases do jogos realizam trocas de cena quando a condição de vitória ou derrota for cumprida. Já as demais cenas, realizam a troca através de botões, que são criados a partir de um modelo ”obj_parent_button” com o código do *Snippet* 15. Isso providencia uma função para ser executada quando o botão for pressionado e cria um comportamento baseado na posição e no clique do botão esquerdo do mouse.

```
1 // Create event
```

```
2  hovering = false;
3  clicked = false;
4
5  function activate_button() {
6    // Do something
7  }
8  // Step event
9  hovering = position_meeting(device_mouse_x_to_gui(0), device_mouse_y_to_gui
       (0), id);
10 if (hovering && mouse_check_button_pressed(mb_left)) {
11   clicked = true;
12 }
13 if (mouse_check_button_released(mb_left)) {
14   clicked = false;
15   if (hovering) {
16     activate_button();
17   }
18 }
19 if (clicked) image_index = 2;
20 else if (hovering) image_index = 1;
21 else image_index = 0;
```

Snippet 15 – Código dos eventos do objeto ”obj_button_parent”

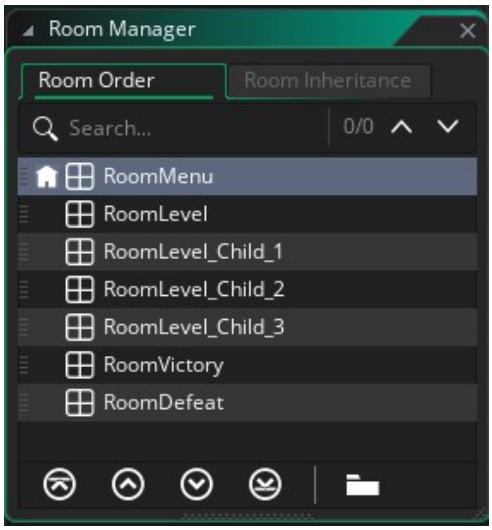

Figura 42 – *GameMaker* - janela ”*Room Manager*”

4.3.13 Efeitos visuais extras

O motor fornece uma variedade de efeitos adicionais para serem utilizados. Neste projeto foi utilizado um efeito de folhas voando no plano de fundo, que pode ser ativado pelas configurações do *layer* ”*Background*” de uma *room*. Além disso, também foi criado um efeito de animação para o título do jogo, utilizando um recurso cha-

mado ”Sequência”, que permite adicionar elementos e configurar suas propriedades para serem alteradas durante um intervalo de tempo.

4.3.14 Geração de *build*

Utilizando uma licença gratuita de *GameMaker*, os jogos só podem ser publicados na plataforma ”*gx.games*” da empresa *Opera*. Pela opção ”*Build*”, o desenvolvedor pode selecionar ”*Create Executable*”, isso abre uma janela que iniciará a exportação, antes é necessário logar em um conta ”*Opera*”, caso ainda não esteja logado. Quando estiver terminado, será possível pressionar ”*Edit Game on Opera*” para configurar a exibição do jogo na plataforma (Figura 43).

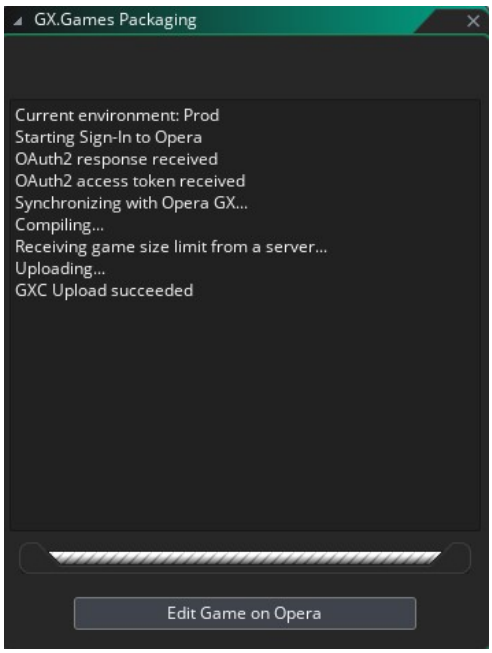

Figura 43 – *GameMaker* - janela ”*GX.Games Packaging*”

No navegador, é aberta uma página do site ”*gx.create*” com as informações do jogo criado. É possível alterar o título do jogo, a descrição, adicionar imagens e, ao clicar na opção ”*Publishing*”, uma nova aba é aberta, onde a visibilidade do jogo pode ser alterado e a publicação pode ser concluída [2].

4.4 *Unity*

O conhecimento de *Unity* acumulado com a experiência de desenvolver o mesmo jogo duas vezes, permitiram que o trabalho com esse motor fosse o mais ágil dos três. Mas *Unity* também é a ferramenta mais pesada dentre as escolhidas, de modo que levava um pouco mais de tempo para abrir o projeto e compilar o código.

[2] <https://gx.games/games/dwvi03/fox-vs-plants-gamemaker-version-/>

O motor não possui um editor de código próprio, mas possui uma integração fácil com *Visual Studio*. Porém, por preferência pessoal, o editor escolhido foi o *Visual Studio Code*, que apesar de possuir integração com o motor, necessita de um pacote disponível no *Unity Package Manager* e de uma configuração nas preferências do *Unity* para alterar o editor de *scripts* externo.

4.4.1 *Scenes, GameObjects* e *Components*

Unity trabalha com cenas de jogo, compostas por "*GameObjects*", que por sua vez, utilizam "*Components*" dos mais variados tipos. Geralmente, cenas são as diferentes telas ou fases do jogo, objetos são os diferentes elementos, como a raposa e as moitas, e componentes são o que definem o comportamento e aparência de cada elemento. Por trabalhar tanto com jogos 2D quanto 3D, os componentes para os jogos em duas dimensões costumam ter o sufixo "2D" para serem diferenciados.

4.4.2 Movimento do jogador

Iniciando a implementação, há apenas uma cena criada chamada "*SampleScene*" com a "*Main Camera*" (Figura 44). Nesse espaço, é criado um *GameObject* do tipo "*Sprite*" com o nome "*Player*". Em seguida, o componente "*Rigidbody2D*" é adicionado para que jogador possa se mover. Para o contexto de jogo *top-down*, é necessário alterar as propriedades desse componente para que a escala de gravidade seja zero e a rotação do eixo "Z" seja congelada (Figura 45).

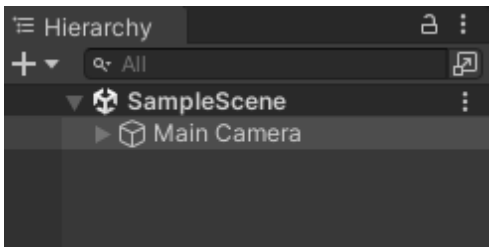

Figura 44 – *Unity* - aba *Hierarchy*

Com o "*Rigidbody2D*" configurado, basta adicionar o *script* "*PlayerMovementScript*", mostrado no *Snippet* 16 para fazer o jogador se mover a partir da entrada do usuário. O código utiliza o método "*Update*" para receber as interações com as teclas configuradas para alterar os eixos horizontais e verticais, que por padrão são "W", "A", "S", "D" e as setas direcionais. Em seguida, o método "*FixedUpdate*" é utilizado para de fato mover o objeto. É importante utilizar esse outro método, porque o movimento é realizado com uma variável da passagem de tempo entre cada atualização, sendo que "*Update*" utiliza a mudança de *frame* atualizar e "*FixedUpdate*" um valor fixo do sistema de física. Isso garante que as ações relacionadas à física do jogo sejam mais precisas.

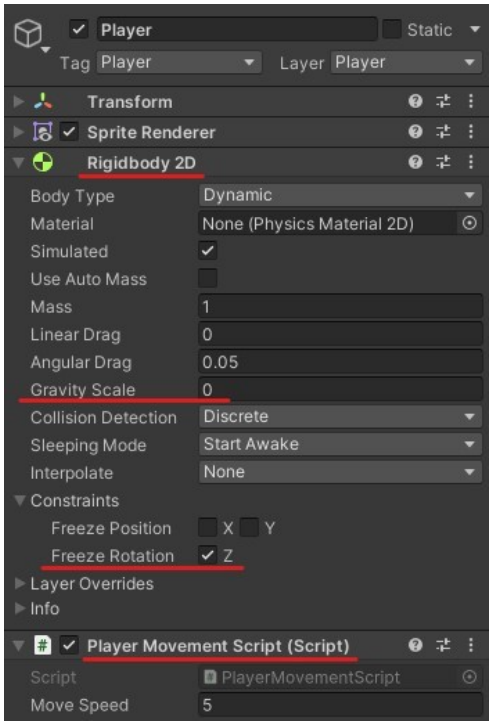

Figura 45 – *Unity* - inspetor do objeto *Player*

```
public class PlayerMovementScript : MonoBehaviour
{
    public float moveSpeed = 5f;
    Rigidbody2D rb;
    Vector2 movement;

    // Start é chamado antes da primeira atualização de frame
    void Start() {
        rb = GetComponent<Rigidbody2D>();
    }
    // Update é chamado uma vez por frame
    void Update() {
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");
    }
    // FixedUpdate é chamado de acordo com a frequência definida no sistema de física
    void FixedUpdate() {
        rb.MovePosition(rb.position + movement.normalized * moveSpeed * Time.fixedDeltaTime);
    }
}
```

Snippet 16 – Código de movimento do jogador

4.4.3 Ataque do jogador

Os ataques do jogador são objetos criados da mesma forma, possuem os componentes ”*Sprite*” e ”*Rigidbody2D*”, mas são transformados em *prefabs*. Isso é um recurso de *Unity* que permite armazenar um objeto como um arquivo do projeto para poder reutilizá-lo criando quantas cópias desejar. Uma vez criado, objetos que se tornaram *prefabs* são identificados pela cor azul. Isso permite instanciar os ataques do jogador por código.

Apesar de poder utilizar o primeiro *script*, um novo foi criado, pois isso é considerado uma boa prática para facilitar a leitura de código e modularizar o projeto, simplificando a edição e reaproveitamento de código. O novo arquivo, mostrado no *Snippet* 17, consiste em detectar quando o usuário apertar o botão esquerdo ou direito direito do mouse e instanciar o ataque na direção do cursor. Para a espada, o objeto é instanciado para algum dos quatro lados, detectando qual mais se aproxima da posição do mouse e aplicando a rotação necessária ao ataque. Já para a folha, o método ”*AddForce*” é utilizado para aplicar uma força com a direção para onde o objeto deve ser lançado.

```
1  public class PlayerAttackScript : MonoBehaviour
2  {
3      public Camera _worldCamera;
4      public GameObject leafProjectile;
5      public float projectileSpeed = 5f;
6      Vector2 lookDirection;
7
8      void Update() {
9          lookDirection = _worldCamera.ScreenToWorldPoint(Input.mousePosition
       ) - transform.position;
10         lookDirection = lookDirection.normalized;
11         if(Input.GetButtonDown("Fire1")) SwordAttack();
12         if(Input.GetButtonDown("Fire2")) CreateProjectile();
13     }
14     void SwordAttack() {
15         Vector3 pos;
16         float rot = 0;
17         [...]
18         GameObject sword = Instantiate(swordAttack, transform.position +
       pos/2, transform.rotation);
19         Rigidbody2D rbSword = sword.GetComponent<Rigidbody2D>();
20         rbSword.rotation = rot;
21     }
22     void CreateProjectile() {
23         GameObject leaf = Instantiate(leafProjectile, transform.position,
       transform.rotation);
24         Rigidbody2D rbProj = leaf.GetComponent<Rigidbody2D>();
```

```
25          rbProj.AddForce(lookDirection * projectileSpeed, ForceMode2D.
    Impulse);
26      }
27  }
```

Snippet 17 – Código dos ataques do jogador

4.4.4 Colisão

Colisões em *Unity* precisam de duas condições para acontecerem. A primeira é que ambos os objetos envolvidos possuam um componente "*Collider2D*" e pelo menos um deles deve ter um componente "*Physics2D*" ou algum outro que herde dele, como "*Rigidbody2D*". A segunda condição é configurar a "*Layer*" de cada objeto e permitir que colidam na "*Layer Collision Matrix*".

A camada de um objeto define onde ele está para que possa ser detectado por outros objeto dentro da cena de jogo. E a matriz de colisão de *layers* é o que gerencia quais camadas podem colidir entre si. Objetos de uma *layer* podem ver e serem vistos por outros de várias camadas incluindo sua própria. Por padrão, *Unity* já cria os projetos com algumas *layers* e nesse projeto foram adicionadas mais conforme a necessidade (Figura 46).

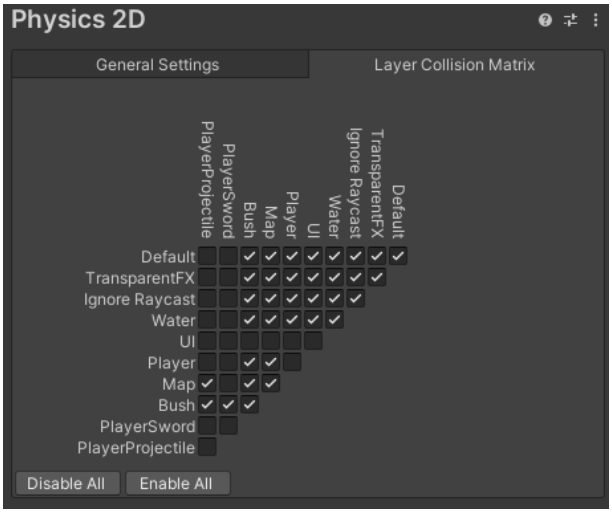

Figura 46 – *Unity - Layer Collision Matrix*

Com as condições cumpridas, os objetos podem colidir entre si, atrapalhando seus movimentos ou utilizando os métodos: "*OnCollisionEnter2D*", "*OnCollisionStay2D*" e "*OnCollisionExit2D*" para lidar com as colisões em diferentes casos. Isso permite utilizar o *script*, mostrado no *Snippet* 18, para destruir os projéteis quando detectarem uma colisão.

```
1 public class LeafProjectileScript : MonoBehaviour
2 {
```

```
3     // Detecta quando inicia uma colisão
4     void OnCollisionEnter2D(Collision2D col) {
5         GameObject.Destroy(gameObject);
6     }
7 }
```

Snippet 18 – Código para autodestruição do projétil em uma colisão

4.4.5 Moitas

Utilizando colisores, as moitas podem ser implementadas para detectarem os ataques do jogador. O *script* do *Snippet* 19 mostra sua inicialização no método "*Start*", onde uma cor aleatória é escolhida e aplicada ao seu *sprite*. Em seguida, seu método "*OnCollisonEnter2D*" é utilizado para detectar as colisões com o projétil e a espada. O primeiro apenas troca sua cor e o segundo, verifica se está na cor alvo para ser destruída. A variável contendo essa informação poderá ser trocada pelo objeto da grama.

```
1 public class BushScript : MonoBehaviour
2 {
3     SpriteRenderer bSprite;
4     Color[] bColors;
5     int colorCount = 0;
6     public Color target = Color.white;
7 
8     void Start() {
9         bSprite = gameObject.GetComponent<SpriteRenderer>();
10        Color nColor = new Color(0.75f,0.75f,0.75f);
11        bColors = new Color[] {Color.white, Color.green * nColor, Color.
      magenta * nColor, Color.red * nColor, Color.yellow};
12        colorCount = Random.Range(0, bColors.Length - 1);
13        bSprite.color = bColors[colorCount];
14    }
15    void OnCollisionEnter2D(Collision2D col) {
16        if (col.gameObject.tag == "PlayerProjectile") {
17            colorCount++;
18            if (colorCount >= bColors.Length)
19                colorCount = 0;
20            bSprite.color = bColors[colorCount];
21        }
22        if (col.gameObject.tag == "PlayerAttack") {
23            if (target == bColors[colorCount])
24                GameObject.Destroy(gameObject);
25        }
26    }
27 }
```

Snippet 19 – Código da moita

4.4.6 Grama

A grama do jogo utiliza apenas um *sprite* e um *script*. Seu código, mostrado no *Snippet* 20, detecta todas as moitas sobre a grama utilizando o método "*Physics2D.OverlapBoxAll*" que detecta todos os colisores de uma dada *layer* na região. Em seguida, altera a cor alvo de cada uma dessas moitas. Sempre que passar um certo intervalo de tempo, a cor da grama é trocada e suas moitas são atualizadas para alvejarem a nova cor. Assim como o projeto dos outros motores, o desenvolvedor pode alterar as variáveis da cor da grama e do intervalo de tempo até a troca, podendo escolher uma cor aleatória ou desabilitar a troca de cores.

```
1  public class GrassBackgroundScript : MonoBehaviour
2  {
3      [...]
4      SpriteRenderer rend;
5      Collider2D[] bushes;
6      public int colorCount = 0;
7      public float colorSwitchDelay = 5.0f;
8      float currentTime = 0.0f;
9      public LayerMask bushLayer;
10
11     void Start() {
12         [...]
13         Vector2 boxSize = rend.bounds.size;
14         bushes = Physics2D.OverlapBoxAll(transform.position, boxSize, 0,
       bushLayer);
15         if (colorCount < 0)
16             UpdateColor();
17         else
18             rend.color = bColors[colorCount];
19         UpdateBushes();
20         if (colorSwitchDelay <= 0.0f)
21             enabled = false;
22     }
23     void Update() {
24         currentTime += Time.deltaTime;
25         if(currentTime >= colorSwitchDelay) {
26             currentTime = 0.0f;
27             UpdateColor();
28             UpdateBushes();
29         }
30     }
31     void UpdateColor() {
32         int color = Random.Range(0, bColors.Length - 1);
33         if(color == colorCount) {
34             colorCount++;
35             if (colorCount >= bColors.Length)
```

```
36                  colorCount = 0;
37          } else {
38              colorCount = color;
39          }
40          rend.color = bColors[colorCount];
41      }
42      void UpdateBushes() {
43          foreach (Collider2D bush in bushes) {
44              if (bush != null) {
45                  BushScript bushScript = bush.gameObject.GetComponent<
    BushScript>();
46                  bushScript.target = bColors[colorCount];
47              }
48          }
49      }
50  }
```

Snippet 20 – Código da grama

4.4.7 *Tilemap*

Para criar os cenários de jogo, *Unity* utiliza objetos do tipo "*Tilemap*" para desenhar conjuntos de *tiles*. Criar um *tilemap*, trás junto um objeto "*Grid*", que basicamente cria o espaço para desenhar os *tiles*. Interagir que um desses *gameObjects* permite abrir a janela "*Tile Palette*". Ela é utilizada para criar paletas de desenho com os conjuntos de *tiles*. Criando uma nova paleta, o próximo passo é adicionar um arquivo com o *tile set* desejado (Figura 47), isso cria um novos arquivos para cada bloco do conjunto.

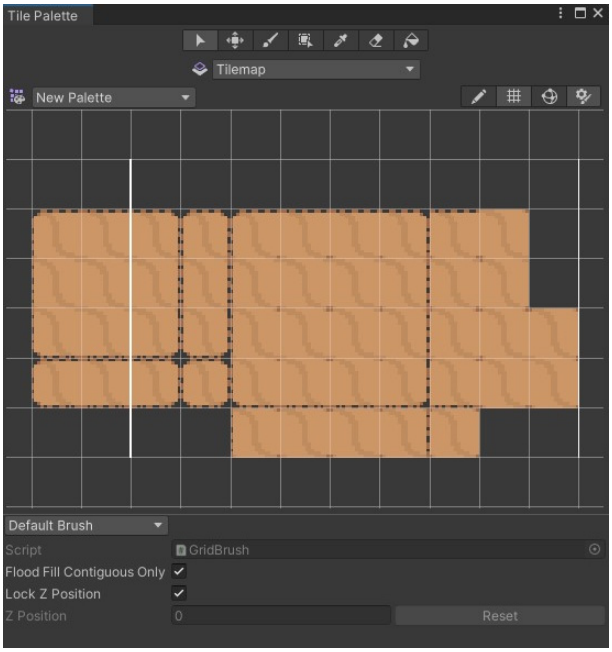

Figura 47 – *Unity* - janela da paleta de *tiles* com *tiles*

Esse processo já permite começar a desenhar o cenário, porém, para facilitar o trabalho, pode ser utilizado um recurso chamado ”*Rule Tile*”, que é basicamente um *autotile* como os dos outros motores. Ele pode criado em alguma pasta do projeto através da opção de criação de *assets*. Em seguida, é possível ir nas suas propriedades e adicionar o conjunto de *tiles* para aplicar as regras de automatização, como mostrado na Figura 48. Utilizando a ideia de matriz 3x3, o ponto central corresponde ao *tile* em questão, as setas verdes a onde precisa haver um *tile* e os ”X” vermelhos representam os espaços que não podem ter cenário. Terminando de configurar o *rule tile*, ele pode ser adicionado a uma paleta no lugar do *tile set* padrão.

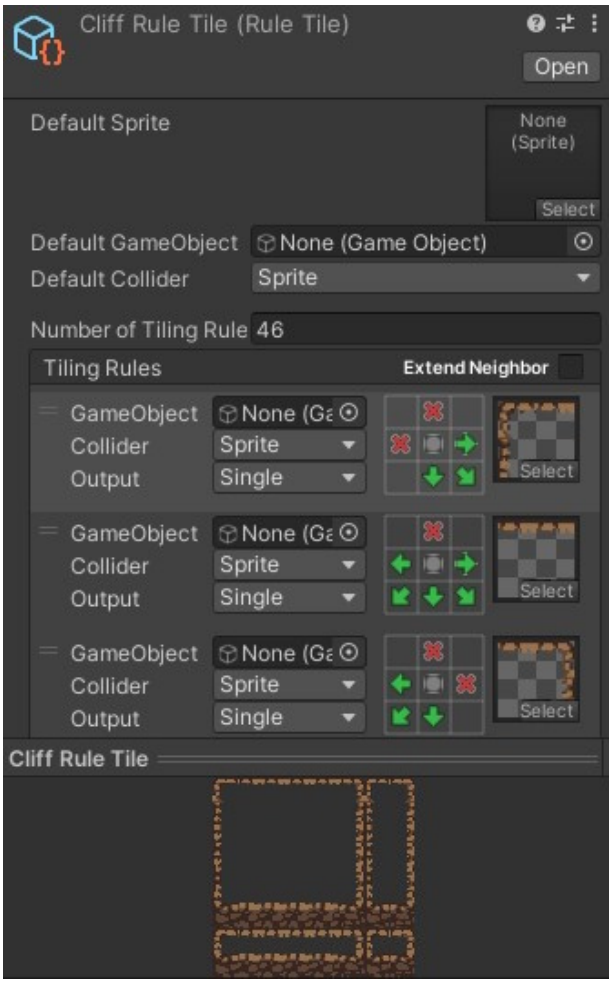

Figura 48 – *Unity* - inspetor do *Rule Tile*

Neste projeto, foram criados dois *tilemaps*, um para as rochas do cenário e outro para a terra, pois cada um desses objetos funciona como uma camada, de modo que não é possível desenhar dois blocos diferentes no mesmo *tilemap*. Para o cenário da rocha, também foi adicionado um componente ”*TilemapCollider2D*” para que seus *tiles* possam colidir com outros objetos.

4.4.8 Animações

Adicionar imagens estáticas a objetos de jogo é simples, basta associar o arquivo de imagem ao *sprite* do objeto. Animações, por outro lado, precisam de duas coisas: um componente "*Animator*", usado para reproduzir as animações do objeto e que recebe um *asset* "*AnimatorController*", responsável por armazenar e gerenciar as transições entre as animações. Com ambos adicionados ao objeto, é possível criar novas animações através da janela "*Animation*" (Figura 49).

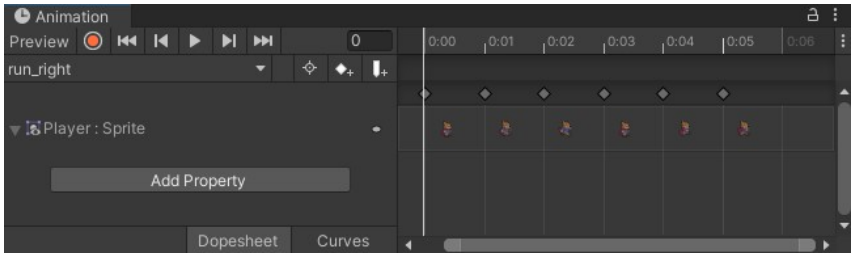

Figura 49 – *Unity* - janela *Animation*

Em *Unity*, é possível controlar a mudança de várias propriedades dos *gameObjects* através de animações, mas neste caso, só foram adicionados os *sprites* de cada animação da raposa. Após todas serem criadas, acessar a janela "*Animator*" possibilita criar transições entre todas as animações (Figura 50). Funcionando como uma máquina de estados, também é possível criar uma *blend tree* para criar transições fáceis entre múltiplos elementos. Neste projeto, foram criadas três *blend trees* com o tipo "*2D Simple Directional*" para utilizar duas variáveis para controlar as transições dentro delas.

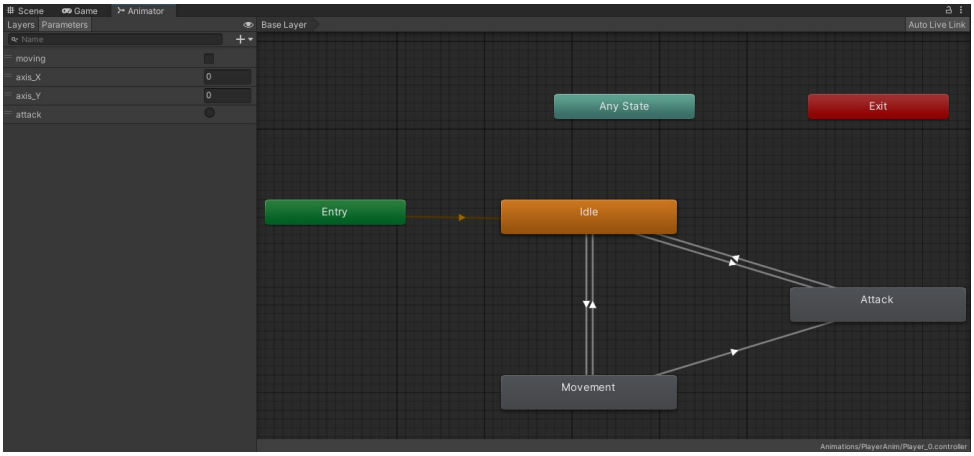

Figura 50 – *Unity* - controlador de animações do jogador

Trabalhar com animações em *Unity* é bastante similar a *Godot*, de modo que as *blend trees* foram utilizadas para as três animações da raposa para cada um dos quatro lados possíveis de serem reproduzidas. Mas uma diferença, é a possibilidade de configurar a transição entre estados, ajustando o tempo e as variáveis que a efetuam.

Isso permite que apenas alterando o valor dessas variáveis de animação através de código, as transições ocorram. Como demonstrado no *Snippet* 21, para as transições da animação de movimento da raposa, a variável "*moving*" é alterada e para ajustar o lado para onde ela deve ser reproduzida são usadas "*axis_X*" e "*axis_Y*".

```
1  public class PlayerMovementScript : MonoBehaviour
2  {
3      [...]
4      Animator animator;
5
6      void Start() {
7          [...]
8          animator = GetComponent<Animator>();
9      }
10     void Update() {
11         [...]
12         if (movement.magnitude != 0.0f) {
13             animator.SetBool("moving", true);
14             animator.SetFloat("axis_X", movement.x);
15             animator.SetFloat("axis_Y", movement.y);
16         } else {
17             animator.SetBool("moving", false);
18         }
19     }
20     [...]
21 }
```

Snippet 21 – Código do jogador alterado para controlar as animações

4.4.9 Música e efeitos sonoros

Para reprodução de sons, *Unity* possui um componente "*AudioSource*", que recebe um arquivo de áudio e configura sua reprodução (Figura 51). Como um modo ter mais controle sobre recursos de áudio, foi criado um *script* "*SoundManager*" em um *gameObject* para gerenciar todos os sons do jogo. Este objeto recebe os arquivos de áudio, cria um "*AudioSource*" para cada um e possui métodos para iniciar ou para a reprodução de um som específico. Além disso, seu código foi implementado utilizando o padrão *Singleton* para que o controle dos efeitos de som permaneça acessível após uma troca de cena.

4.4.10 Câmera de jogo

Todas as cenas em *Unity* já possuem uma câmera de jogo e, embora seja possível fazê-la se mover por código, um componente que facilita muito o controle sobre ela é o "*Cinemachine*", disponível no "*Package Manager*" do motor. Esse pacote permite

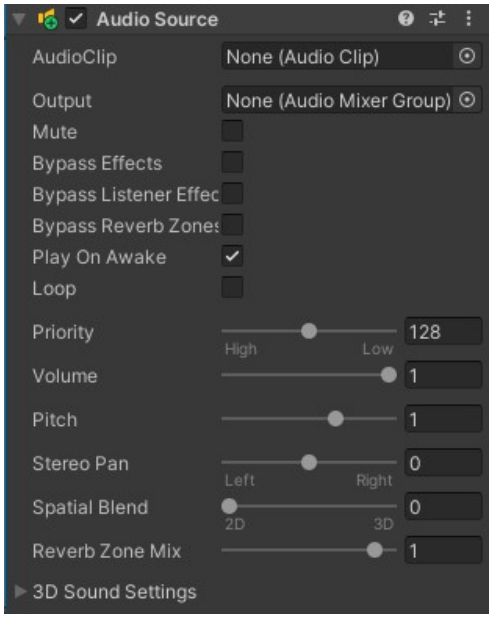

Figura 51 – *Unity* - componente *AudioSource*

criar diversos tipos de câmera de jogo: "*2D Camera*", "*ClearShot Camera*", "*Dolly Camera with Track*", "*Dolly Track with Cart*", "*FreeLook Camera*", "*State-Driven Camera*", dentre outros. Utilizar um deles, associa um componente "*CinemachineBrain*" à câmera principal para que ela seja controlada.

Neste projeto, foi utilizada uma "*2D Camera*", que cria um objeto com o componente "*CinemachineVirtualCamera*" já configurado para usar em um jogo 2D. Isso permite referenciar o objeto do jogador para que a câmera o siga, além de poder configurar detalhes do seu movimento na aba "*Body*" do componente.

4.4.11 Interface do usuário (UI)

Os elementos de UI são criados dentro de um objeto "*Canvas*", que precisa ser filho da "*Main Camera*" para acompanhá-la durante seu movimento. Os *gameObjects* de cada elemento possuem um *sprite* e um *Text (TMP)*, que faz parte do pacote *Text Mesh Pro*, utilizado para facilitar a customização de textos. Para gerenciar o andamento do jogo e atualizar a interface do usuário, foi criado um objeto com um *script* "*LevelController*".

Seu código, apresentado no *Snippet* 22, utiliza referências ao textos do relógio e do contador de moitas para atualizar cada texto nos método "*UpdateTimer*" e "*UpdateBushes*". Além disso, também registra a passagem de tempo do relógio e detecta um *gameObject* chamado "*Bushes*", que contém todas as moitas da fase como filhos. E quando detectar as condições de vitória ou derrota, realiza a devida troca de cena.

```
1 public class LevelController : MonoBehaviour
2 {
```

```
3     public string nextScene = "VictoryScene";
4     public float levelTime = 120.0f;
5     float currentTime = 0.0f;
6     GameObject bushParent;
7     int totalBushes;
8     int remainBushes;
9     public TextMeshProUGUI timeText;
10    public TextMeshProUGUI bushText;
11
12    void Start() {
13        bushParent = GameObject.Find("Bushes");
14        totalBushes = bushParent.transform.childCount;
15        UpdateBushes();
16    }
17    void Update() {
18        currentTime += Time.deltaTime;
19        UpdateTimer();
20        UpdateBushes();
21        if (currentTime >= levelTime)
22            OnSceneChange("DefeatScene");
23    }
24    void UpdateTimer() {
25        int minutes = (int)(levelTime - currentTime)/60;
26        int seconds = (int)(levelTime - currentTime)%60;
27        if (seconds >= 10)
28            timeText.text = minutes + ":" + seconds;
29        else
30            timeText.text = minutes + ":0" + seconds;
31    }
32    void UpdateBushes() {
33        remainBushes = bushParent.transform.childCount;
34        bushText.text = remainBushes + "/" + totalBushes;
35        if (remainBushes == 0)
36            OnSceneChange(nextScene);
37    }
38 }
```

Snippet 22 – Código do *LevelController*

4.4.12 Fluxo de jogo

As cenas de jogo são gerenciadas por um objeto com o *script* "*GameController*". Usando o padrão *Singleton*, é ele quem realiza as trocas de cena através do código, mostrado no *Snippet* 23, que contém apenas um método para carregar uma cena específica e outro para voltar à cena anterior.

```
1 public class GameController : MonoBehaviour
```

```
{
    public static GameController instance;
    string previousScene;
    [...]
    public void OnSceneChange(string sceneName) {
        previousScene = SceneManager.GetActiveScene().name;
        SceneManager.LoadScene(sceneName);
    }
    public void OnRestart() {
        SceneManager.LoadScene(previousScene);
    }
}
```

Snippet 23 – Código do *GameController*

Por ser um objeto com instância única, sua referência é perdida quando há uma troca de cena. Portanto, é necessário utilizar um outro objeto sem *Singleton* com um *script* que chame os métodos tanto do ”*GameController*” quanto do ”*SoundManager*”. Para as cenas de menu, vitória e derrota, foi utilizado o objeto ”*MenuController*” com o *script* do *Snippet* 24. Esse mesmo código também foi adicionado ao ”*LevelController*”.

```
public class MenuController : MonoBehaviour
{
    public void PlaySound(string name) {
        SoundManager.instance.Play(name);
    }
    public void OnSceneChange(string sceneName) {
        GameController.instance.OnSceneChange(sceneName);
    }
    public void OnRestart() {
        GameController.instance.OnRestart();
    }
}
```

Snippet 24 – Código do *MenuController*

Para realizar a chamada a esses métodos na cena de menu e similares, foram utilizados botões. Em *Unity*, as configurações de um botão permitem alterar sua visualização e utilizar a área ”*OnClick*” para selecionar métodos de outros objetos para serem executados. Basta selecionar o *gameObject* desejado, escolher um método dele e passar o parâmetro necessário (Figura 52).

4.4.13 Geração de *build*

Concluindo o projeto com *Unity*, o último passo é gerar uma *build* do jogo para que ele possa ser compartilhado [3]. Na IDE, basta ir na opção *Build Settings* ou usar

[3] <https://zub40.itch.io/fox-vs-plants-unity-version>

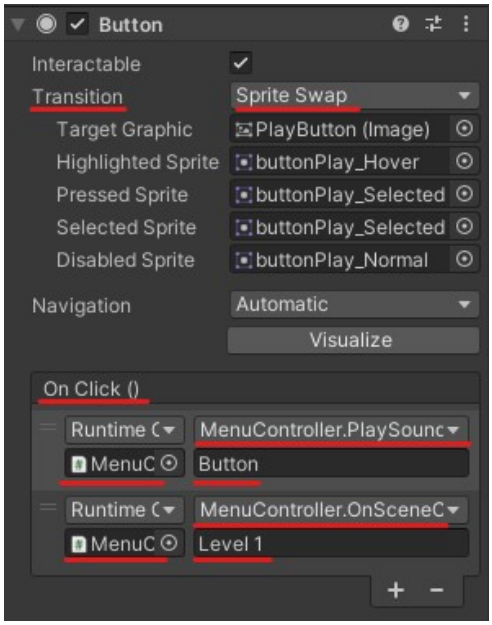

Figura 52 – *Unity* - componente *Button*

o atalho ”Ctrl+Shift+B” para abrir uma janela com as configurações para gerar *build* (Figura 53). É possível selecionar todas as cenas de jogo que serão utilizadas e a plataforma para qual o jogo será disponibilizado. *Unity* já possui todos os arquivos necessários instalados, portanto, com tudo configurado, basta clicar em *Build* e escolher um local para salvar o arquivo. Uma vez terminado, o jogo está pronto para ser compartilhado. Para colocá-lo no *itch.io*, é necessário somente colocar os arquivos da *build* em uma pasta compactada do tipo ”zip” e carregar essa pasta no projeto dentro do site.

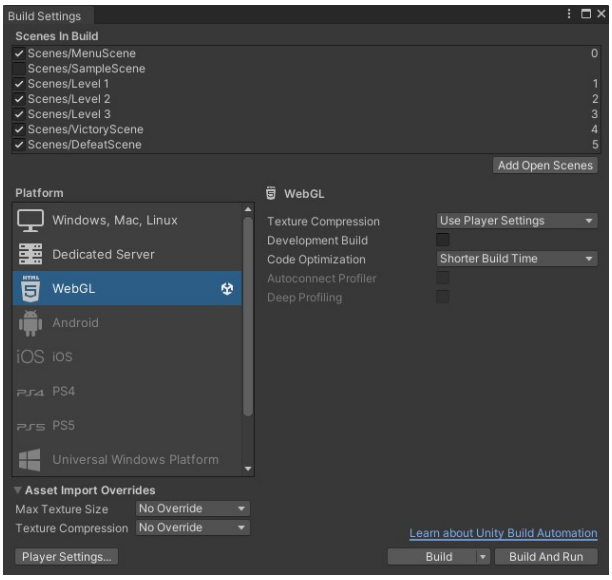

Figura 53 – *Unity* - janela *Build Settings*

4.5 *Game Jams*

4.5.1 Primeira jam - *Godot Wild Jam*

Para continuar a experiência com *Godot*, foi iniciado um projeto para a "*Godot Wild Jam* #58". O desenvolvimento iniciou com a busca por ideias para o tema "*rain or shrine*" e busca por *assets* para utilizar no jogo. Após alguns dias, foi decidido implementar um jogo 2D *sidescroller* de plataforma. A ideia seria criar um personagem para andar por um cenário chuvoso enquanto utilizava um orbe para se proteger, e seu objetivo seria alcançar um altar e depositar o orbe.

Porém, o tempo para desenvolver o projeto acabou ainda nas fases iniciais de desenvolvimento. Só havia sido adicionado o personagem com movimento e animação, um *tile set* básico e um NPC com um *script* para patrulhar uma área. Os motivos que impediram a conclusão desse projeto foram:

- Necessidade de aprender as novidades da versão 4 de *Godot*;
- Excesso de trabalho na ideia de jogo;
- Gerenciamento de tempo.

Apesar do jogo não ter sido concluído, a *game jam* foi um aprendizado que pôde ser levado para os outros eventos. Essa experiência permitiu aprender sobre a nova versão do motor e saber como se planejar melhor para outros eventos.

4.5.2 Segunda jam - *Learn You a Game Jam Pixel Edition*

Essa *game jam* possuía uma proposta mais educativa, incentivando os participantes a aprendem algo novo, utilizarem *assets* de *pixel art* e registrarem o desenvolvimento e aprendizado em um blog, vlog ou dev-log. Ela durou do dia 17 de junho de 2023, enquanto ainda estava ocorrendo a primeira *jam*, a 27 de junho, de modo que alguns dias de desenvolvimento foram perdidos, mas ainda foi possível concluir o projeto também utilizando *Godot*.

Pelo tema "*You are The Monster*", foi planejado criar um jogo isométrico, onde o jogador controlaria um monstro para destruir a cidade. Os detalhes foram decididos no decorrer do projeto, de modo que logo se iniciou o desenvolvimento criando uma imagem de *tile* isométrico para o jogo e um *sprite* para o monstro.

A implementação de um *tilemap* isométrico é idêntica a de *tiles* quadrados, sendo necessário apenas ajustar a distância entre os blocos do mapa para que fiquem bem encaixados. O movimento do jogador também não é um problema. Utilizando o código de movimento do projeto "*Fox vs Plants*", bastou ajustar o valor de cada eixo

para ter um movimento diagonal. O desafio foi aplicar a ideia de centralizar o jogador nos *tiles* para que ele se movimentasse apenas um bloco por vez. No fim, a referência ao *node* do *tilemap* foi utilizada para detectar se havia um *tile* abaixo do jogador, caso houvesse, a posição dele era ajusta para centralizar no bloco, caso não, o movimento era negado e ele retornava ao bloco onde estava.

Nesse ponto, surgiu a seguinte ideia: o monstro precisa destruir a cidade para recuperar vida, mas toda vez que se move ele enfraquece. Desse modo, foi implementado um valor de pontos de vida para o jogador, sendo um máximo de cem, e toda vez que se move perde dez. E foram criados prédios no mapa, pelos quais o jogador pode passar por cima para recuperar vinte pontos de vida. Eles foram implementados utilizando *tilemap* em uma camada diferente do mapa de jogo.

Perto de terminar o prazo, ainda foi possível implementar uma barra de vida na UI sincronizada com o valor de pontos de vida. Isso finalizou um protótipo de jogo (Figura 54), mas para deixar uma ideia mais fechada, foi criado um algoritmo para gerar novos prédios aleatoriamente sempre que todos do mapa fossem destruídos. Isso providenciou um fluxo de jogo, de certa forma, infinito.

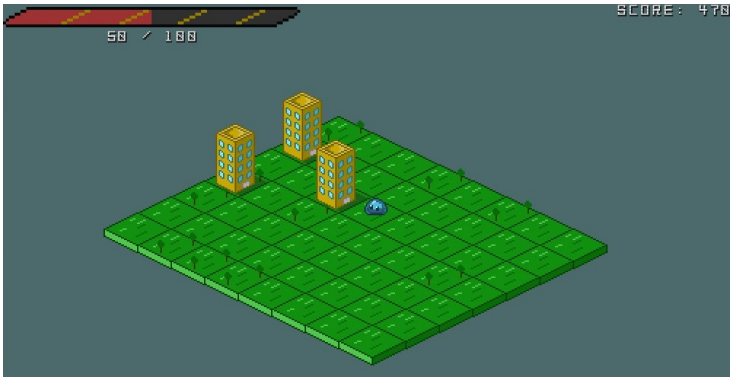

Figura 54 – *Slimus The City Destroyer* - tela de jogo

Nos dois últimos dias, ainda foram implementados: um sistema de pontuação exibido na UI, uma tela de menu (Figura 55), uma segunda fase de jogo selecionável (Figura 56) e uma animação do monstro devorando os prédios, que foi adquirida graças a colaboração de um colega. Único detalhe que faltou implementar foram os efeitos sonoros para o jogo.

O projeto, que recebeu o nome de ”*Slimus The City Destroyer*” [4], foi submetido pouco antes do prazo acabar. Mas durante a submissão, houve um problema: os *templates* para *build web* da versão 4.0.3 de *Godot* ainda não estavam disponíveis, por isso foi necessário submeter versões para *Windows* e *Linux*, de modo que quem desejasse testar precisaria baixar o jogo. No fim não foi um grande problema, pois o projeto ainda foi avaliado pelos juízes do evento e por alguns dos participantes. No to-

[4] <https://zub40.itch.io/slimus-the-city-destroyer>

Figura 55 – *Slimus The City Destroyer* - tela de ”Menu”

Figura 56 – *Slimus The City Destroyer* - tela de ”Seleção de Nível”

tal essa *game jam* teve 42 submissões e o resultado da avaliação foi bem satisfatório, conseguindo a 28ª colocação e sendo bastante elogiado pela abordagem do tema e pelos visuais. E o ponto mais criticado foi a falta de elementos sonoros.

Analisando essa experiência, o projeto pôde ser concluído com sucesso, um produto mínimo foi entregado e *Godot* foi bem testado. O motor se provou uma boa ferramenta para uma *game jam*, exceto pelo problema dos *templates web*. E a participação de um evento do gênero possibilitou perceber que é necessário um certo preparo e planejamento para poder cumprir seu curto prazo, sendo o ideal já estar acostumado com as ferramentas utilizadas e com o ritmo de trabalho, que são dois pontos possíveis de se conquistar através da prática.

4.5.3 Terceira jam - *GMTK Game Jam 2023*

A *jam* promovida pelo canal do *YouTube*, ”*Game Maker's Toolkit*”, cujo foco é passar conhecimento sobre desenvolvimento de jogos, durou 48 horas. Apesar de ser um tempo bem reduzido em comparação às outras duas *jams*, trabalhar com uma equipe permitiu concluir o projeto com sucesso. Pelo conhecimento dos membros do grupo e pela intenção deste projeto, o motor escolhido para o evento foi *Unity* com a versão 2022.3.4f1.

Num primeiro momento, o grupo se reuniu para fazer um *brainstorm* de ideias. Ficou decidido que seria um jogo de labirinto, onde o jogador controlaria um fantasma e precisaria ajudar uma menina a escapar do local. Isso abordaria o tema "*roles reversed*" utilizando o fantasma para ajudar, em vez de assustar, e alterando o controle padrão de jogo para que o objetivo fosse guiar, em vez de controlar, a personagem. Além da ideia inicial, no primeiro dia foram coletados alguns *assets* gratuitos para utilizar no jogo e foi implementada uma mecânica para um objeto seguir o cursor do mouse.

Por serem apenas 48 horas, o trabalho foi realizado até a hora de conclusão do evento, afim de conseguir finalizar um protótipo. A implementação prosseguiu com a adição de um *tilemap* e o pacote "*Universal Render Pipeline (URP)*" de *Unity* para utilizar um sistema de luzes 2D. Isso possibilitou criar um cenário escuro e trabalhar com a ideia de que a menina precisaria evitar o escuro para não sentir medo. Um ponto de luz foi adicionado ao cursor do mouse para guiar a personagem. E foi implementada uma barra de choro para ser preenchida enquanto a menina estivesse no escuro.

Como adição à mecânica do jogo, foram adicionados pontos de luz espalhados pelo mapa de jogo para servirem como pontos seguros para a menina, de modo que a ideia final do projeto foi: em um labirinto, o fantasma precisa guiar a personagem, que é atraída pela luz (Figura 57). Enquanto estiver no escuro, sua barra de choro é preenchida. Perto de um ponto de luz estático, ela é reduzida. E perto do fantasma, se mantém estagnada. Com isso, a condição de vitória seria a menina alcançar a saída do labirinto e a de derrota a barra de choro ficar cheia.

Figura 57 – *Follow Me - A ghost game* - tela de jogo

Restando ainda várias horas, toda a mecânica foi concluída com sucesso, de forma que o restante do tempo foi utilizado para trabalhar o fluxo de jogo. Foram criadas uma cena de menu, *pop-ups* pós jogo para indicar se o jogador venceu ou perdeu (Figura 58) e botões para realizar a transição das cenas. Objetos "*GameController*" e "*SoundManager*" foram utilizados para facilitar o controle do jogo, e o sistema de

iluminação também foi aplicado ao menu, criando um visual mais diferenciado (Figura 59).

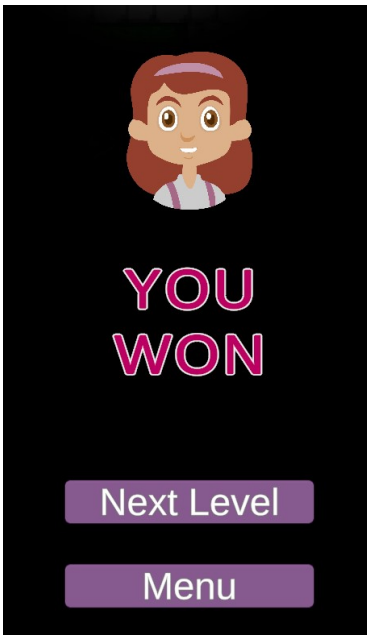

Figura 58 – *Follow Me - A ghost game - pop-up* de ”Vitória”

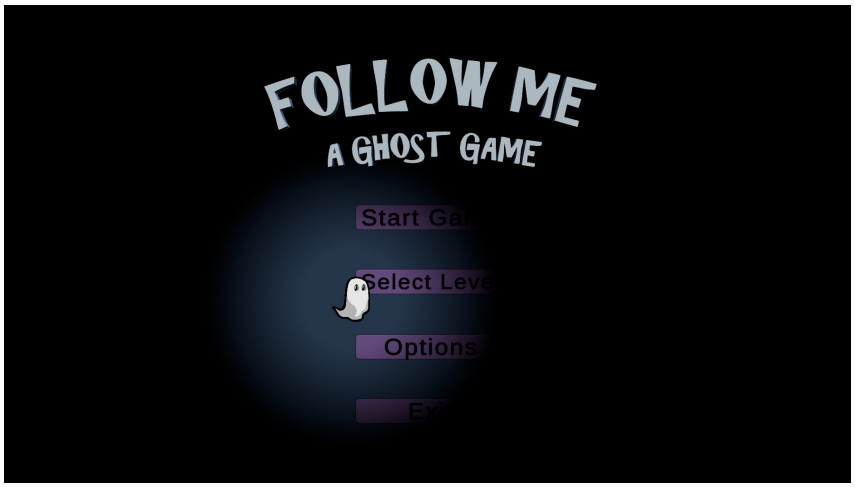

Figura 59 – *Follow Me - A ghost game* - tela de ”Menu”

Um menu de opções foi implementado para pausar o jogo e permitir que o jogador alterasse o volume. Por fim, foram criadas artes para o fantasma, a menina, que recebeu o nome de Aurora, e a UI do jogo. Uma fase de jogo foi desenvolvida com um objetivo para ser alcançado e o projeto foi concluído com sucesso. Houve apenas um problema na geração da *build*, pois todas as operações com *Unity* eram muito lentas no *hardware* utilizado originalmente e, após finalmente terminar de compilar o projeto, ocorreu um erro que impedia o jogo de ser aberto no navegador web. Felizmente, outro membro da equipe conseguiu gerar a *build* para publicar o jogo dentro do prazo.

Toda a experiência dessa *game jam* foi um grande sucesso. Desenvolver um jogo em apenas 48 horas é um desafio, forçando os participantes a serem mais assertivos com as ideias e críticos com as tomadas de decisão. Mas, trabalhando em equipe, é possível explorar muito mais opções para o projeto e implementá-lo com bastante agilidade. Cada membro contribui trazendo ideias, conhecimentos e auxiliando os colegas que tiverem dúvidas.

O projeto foi submetido com o nome "*Follow Me - A ghost game*" [5] e, diferente da *game jam* com *Godot*, que era mais focada no aprendizado, a "*GMTK Game Jam 2023*" possuía muitos profissionais da área de jogos participando, de modo que a análise dos projetos foi mais criteriosa. Apesar disso, os resultados da avaliação foram bastante satisfatórios. Dentre as 6.774 submissões, esse jogo obteve a posição 3518, ficando próximo da média. Segundo os comentários de outros participantes, a jogo possuía uma boa proposta, visuais bonitos e era divertido, porém tinha alguns *bugs*.

Após essa *game jam*, surgiu a oportunidade de compartilhar o projeto na mostra de jogos que ocorreu no "Festival REC'n'Play 2023". Para participar, o projeto foi aprimorado por alguns membros da equipe, adicionando: monstros no labirinto para funcionarem como um obstáculo para a personagem; novos níveis; e uma história escondida em páginas de diário espalhadas pelos mapas. Também melhoraram o desempenho geral do jogo corrigindo alguns *bugs*. Essas melhorias foram iniciadas antes da seleção dos jogos para evento e concluídas pouco antes dele acontecer no dia 21 de outubro. Por fim, essas mudanças foram adicionadas, atualizando o jogo no *itch.io*.

De forma presencial, várias pessoas puderam testar os jogos da mostra e dar suas opiniões. A intenção do evento era incentivar o desenvolvimento de jogos para os desenvolvedores locais, permitindo iniciantes a compartilharem suas criações e receberem *feedback* construtivo. Dentre os visitantes e os demais desenvolvedores presentes, "*Follow Me - A ghost game*" foi bastante elogiado por eles.

Todo esse trabalho da *game jam* permitiu aprender bastante de *Unity*. Isso contribuiu para acelerar o desenvolvimento do projeto "*Fox vs Plants*" com o motor, que ocorreu alguns meses depois. Muitos recursos foram aproveitados de um jogo para o outro e os conceitos da ferramenta foram aprendidos.

4.6 Análise e comparação dos motores

Implementar o mesmo jogo em três motores permite observar suas diferenças, pontos fortes, fracos e o que têm em comum. A experiência com *Godot* e *Unity* definitivamente foi mais extensa devido às *game jams*, mas a maior parte dos recursos de cada motor pôde ser bem observada durante o desenvolvimento do "*Fox vs Plants*".

[5] <https://zub40.itch.io/follow-me-a-ghost-game>

4.6.1 Análise de *Godot*

Godot ainda é um motor bem recente, mas já possui os recursos necessários para desenvolver um jogo de qualidade. As mecânicas principais dos jogos abordadas neste projeto são facilmente implementadas, como *tilemaps*, UI, câmera de jogo, efeitos sonoros, tudo isso pode ser adicionado com um *node* e poucas linhas de código. Possui um bom sistema de colisão e controlador de animações. Seu ambiente de trabalho permite utilizar múltiplas abas ao mesmo tempo com diferentes cenas de jogo. Talvez o ponto mais complexo do motor seja seus sistema de sinais, mas é um recurso que abre portas para uma grande variedade de implementações.

Tudo isso faz de *Godot* uma excelente ferramenta, mas seu maior diferencial é o fato de ser bastante acessível e simples de usar. É um motor que ocupa muito pouco espaço de memória, possui um bom desempenho e está disponível em múltiplas plataformas. Além disso, possui um *design* mais agradável aos olhos, pois não expõe tantas informações simultâneas e utiliza bastante espaçamento entre seus elementos. Dos três motores, com certeza é o mais fácil de baixar e começar a programar.

Os principais pontos negativos são devido ao fato de ainda ser relativamente novo. Algumas propriedades dos *nodes* são difíceis de serem acessadas utilizando C#, pois o suporte de *Godot* a essa linguagem ainda não é tão grande, sendo mais recomendado utilizar *GDSript* para iniciar o aprendizado. Embora, exista bastante conteúdo para aprender *Godot*, podem haver dúvidas cuja resposta não seja fácil de encontrar na internet, principalmente se for sobre código com linguagens diferentes da sua original.

4.6.2 Análise de *GameMaker*

GameMaker, por sua vez, existe a bastante tempo, se consolidando como uma boa ferramenta para criar jogos 2D. O motor também não exige muito de memória e possui um bom desempenho. Dos três, é o que possui a estrutura mais diferente, utilizando seu sistema de eventos em cada objeto. Por um lado isso facilita a compreensão de cada elemento e requer pouco trabalho, pois geralmente há pouco código em cada evento. O problema é que sem o devido cuidado, o código pode facilmente ficar desorganizado.

O motor permite implementar facilmente muitos dos recursos de jogos 2D, trás uma variedade de efeitos extras e permite editar arquivos como os *sprites* no próprio editor. Entretanto, algumas implementações que são simples nos outros dois motores são mais complicadas com *GameMaker*. Fazer a personagem se mover requer mais código, não existe um controlador de animações e implementar colisão para o *tilemap* é um processo complexo que envolve muitas checagens à posição dos *tiles*. O fato de

sua versão gratuita só permitir publicar jogos no site *gx.games* limita seu alcance, mas graças a isso, o motor foi otimizado para facilmente exportar o projeto para lá.

De modo geral, *GameMaker* permite implementar jogos 2D com bastante facilidade, exceto por um ou outro recurso. Por sua estrutura ser diferente, foi mais difícil compreender alguns de seus conceitos, mas sabendo como o motor funciona e como organizar seus objetos, pode tranquilamente ser utilizado para desenvolver projetos maiores.

4.6.3 Análise de *Unity*

Como o motor gratuito mais popular, *Unity* possui a maior quantidade de recursos dos três motores. O que não existe embutido na ferramenta, pode facilmente ser encontrado em um pacote do motor ou aprendido através de tutoriais e fóruns *online*. É difícil não encontrar uma solução para implementar algo com *Unity*. Seu maior ponto de destaque é a quantidade de informação e material disponível na internet.

Todos os recursos dos projetos puderam ser implementados sem grandes dificuldades. O motor possui um bom controlador de animações e sistema de colisão. Dos três é o que possui mais plataformas de exportação disponíveis e também permite escolher diferentes resoluções de tela para testar o jogo, possibilitando visualizar como seria executá-lo em diferentes plataformas.

Porém, comparar com os outros dois motores, permite enxergar alguns pontos negativos em *Unity*. É um motor que exige mais processamento do computador, de modo que não possui um bom desempenho em máquinas menos potentes. Seu ambiente de desenvolvimento fica atrás dos outros, pois não possui editor de código embutido e não permite abrir múltiplas cenas simultaneamente. Constantemente recebe novas atualizações, o que é um ponto positivo, mas após algumas versões é difícil atualizar o projeto para ser usado com a versão mais recente do motor. Por fim, sua grande variedade de recursos é um ponto forte, mas por conta disso, muitas vezes um iniciante na ferramenta pode se sentir sobrecarregado pela quantidade de opções disponíveis.

4.6.4 Comparação completa

Comparando pontos mais específicos da implementação, é possível classificá-los de acordo com sua capacidade e complexidade utilizando um sistema de pontuação de 1 a 5. Onde cada nível corresponde a:

1	Com desenvolvimento complexo ou que tenha limitações

2	Mais complexo que o nível mediano
3	Desenvolvimento mediano
4	Desenvolvimento simples
5	Mais simples que o anterior ou que apresenta flexibilidade e versatilidade no seu uso

Tabela 2 – Pontuação da tabela de comparação das funcionalidades

Através desse sistema, as funcionalidades de cada motor podem ser avaliadas pela seguinte tabela:

Funcionalidades	**Unity**	**Godot**	**GameMaker**
Estruturas do motor	*GameObjects*, *Components* e *GameScenes*	*Nodes* e *Scenes*	*Objects*, *Events* e *Rooms*
Movimento do jogador	4	4	3
Sistema de colisões	4	4	5
Instanciar objetos	4	3	4
Implementação e controle de animações	4	4	2
Implementação de *tilemaps*	4	5	3
Câmera de jogo	5	4	4
Interface do Usuário	4	4	4
Áudio e efeitos sonoros	5	5	5
Controle de transição entre cenas	4	4	3
Geração de *build*	4	4	5
Extra	Pacotes de *assets*	Múltiplas abas	Múltiplas abas e efeitos visuais

Tabela 3 – Comparação de funcionalidades e complexidade entre os motores

Os pontos abordados são vistos com mais detalhes nos tópicos abaixo:

- Movimento da personagem - não tão simples de implementar com *GameMaker* por não utilizar vetores;
- Sistema de colisões - todos os motores são flexíveis, pois permitem controlar exatamente o que acontece em uma colisão. *GameMaker* é mais simples que os demais por não usar *layers* de colisão, sendo complexo apenas quando se trata de colisões com *tilemaps*;
- Instanciar objetos - um pouco complexo com *Godot* por precisar trocar a hierarquia do *node* instanciado;
- Implementação e controle de animações - versátil com *Unity* e *Godot* por ser utilizado para alterar outras propriedades além das imagens e possuir um controlador de animações robusto. *GameMaker*, entretanto, não possui recursos que facilitem implementar ou controlar animações;
- Implementação de *tilemaps* - *Godot* permite implementar *tilemaps* de forma bem flexível por trazer diversas configurações de fácil acesso. *Unity* também possui configurações similares, mas difíceis de serem encontradas. E *GameMaker* é tão simples quanto *Godot*, porém adicionar colisão aos *tiles* é um processo mais complicado em relação aos outros motores;
- Câmera de jogo - graças ao pacote "*Cinemachine*", o controle de câmeras com *Unity* é bastante flexível;
- Controle de transição entre cenas - transição de cenas ocorre de forma similar nos três motores, o detalhe é como controlar por código. Com *Unity* foi utilizado o objeto "*GameController*" que permitiu realizar facilmente as trocas de cenas. Para *Godot* também foi usado um objeto similar, mas seu código é mais complexo por controlar todas as transições em diferentes momentos. E *GameMaker* não utiliza um objeto para centralizar o controle das transições, elas ocorrem dentro dos objetos de UI. Essas questões que favorecem *Unity*, podem ser devido a falta de conhecimento com os outros motores quando o desenvolvimento foi iniciado;
- Geração de *build* - simples de realizar com *Unity* e *Godot*, porém houveram muitas configurações não exploradas de ambos. *GameMaker* é o motor que torna essa tarefa a mais simples possível, basta utilizar uma conta da plataforma e apertar o botão de exportação.

Cada motor se destaca de uma forma diferente, de modo que não há um melhor. A pontuação da tabela anterior indica que *Unity* permite implementar com mais facilidade a maioria das funcionalidades exploradas neste projeto, devido à sua ampla

game de recursos, que tornam o motor bastante versátil e flexível. Porém, pode exigir mais familiaridade com conceitos de programação e ter uma curva de aprendizado inicial mais íngreme. *Godot* é quase tão simples de utilizar quanto *Unity*, mas atrai principalmente por ser bastante versátil e leve, providenciando uma experiência tranquila com muitas possibilidade. Sua acessibilidade e estrutura simples fazem dele uma excelente ferramenta para desenvolver jogos sem grandes barreiras. *GameMaker* fica um pouco atrás na pontuação, por causa de alguns recursos mais complexos de serem implementados. Mas trás uma abordagem mais acessível para iniciantes, facilitando bastante a criação de jogos 2D mais simples.

5 Conclusão

Comparar motores de jogos não é novidade, afinal é como se inicia o processo de escolha de uma ferramenta. Pesquisas, como as de (DICKSON et al., 2017) e (PAVKOV; FRANKOVIć; HOIć-BOžIć, 2017), já foram conduzidas para determinar a melhor opção para um objetivo específico e chegam a conclusão de que não um melhor motor de forma geral, e sim, um melhor para cada caso. Poucas dessas pesquisas se propuseram a desenvolver um jogo como forma de avaliar mais de um motor e compará-los entre si. E nenhuma, que tenha sido observada, realizou tal comparação entre *Unity*, *Godot* e *GameMaker*.

Sabendo que escolher um motor de jogo depende do caso de uso, a intenção deste projeto estava em levantar dados relevantes sobre cada um dos motores selecionados, como uma forma de descobrir seu potencial e os recursos oferecidos aos desenvolvedores. Isso foi alcançado após o desenvolvimento do jogo ”*Fox vs Plants*” em cada *engine*, mais a participação em três *game jams*.

Cada motor pode ser utilizado para alcançar o objetivo desejado, basta conhecer a ferramenta, saber o que ela é capaz de criar e se está alinhada com a proposta de projeto. Neste trabalho, foi desenvolvido um jogo 2D *top-down* em três motores diferentes e os três são excelentes para essa proposta. Porém, *GameMaker* é mais apropriado para jogos 2D, se a proposta fosse para criar um jogo 3D, outras opções seriam mais interessante. É importante que a ferramenta escolhida seja capaz de realizar a ideia do desenvolvedor.

Conhecimento e preferência pessoal também são fatores relevantes. Pode ser possível criar o mesmo jogo com a mesma agilidade em dois motores diferentes, mas se o desenvolvedor conhecer mais um deles, ou preferir trabalhar com um em vez do outro, escolher o motor preferido possibilitará alcançar um resultado melhor e mais rapidamente. Durante essa pesquisa, o tempo de desenvolvimento gasto com cada motor não foi um ponto utilizado para avaliá-los, mas foi possível observar que o desenvolvimento com *Unity* foi mais rápido, justamente por haver mais experiência com o motor.

Embora o projeto da ”*Godot Wild Jam #58*” não tenha sido concluído, a experiência serviu como aprendizado para os jogos desenvolvidos com sucesso na ”*Learn You a Game Jam: Pixel Edition*” e ”*GMTK Game Jam 2023*”: ”*Slimus The City Destroyer*” e ”*Follow Me - A ghost game*”, respectivamente. Participar desses eventos proporcionou uma compreensão mais profunda das vantagens e desafios envolvidos. O trabalho em equipe e os prazos limitados impulsionaram a criatividade e o aprendizado rápido.

Além disso, ao final de cada projeto, obteve-se pelo menos um protótipo de jogo, contribuindo para o crescimento do portfólio dos desenvolvedores.

Godot, *GameMaker* e *Unity* foram aprendidos, testados e comprovados como excelentes ferramentas para o desenvolvimento de jogos. Ao longo deste trabalho foi apresentado a estrutura de cada motor, o processo de criação de jogos com cada um e seus destaques. É esperado que isso possa providenciar uma base para conhecer esses *games engines* e incentivar a exploração dessas e outras ferramentas de desenvolvimento de jogos.

5.1 Trabalhos futuros

Existem muitos motores de jogos, este trabalho utilizou apenas três dos mais populares. Embora vários já tenham sido estudados, ainda há muito espaço para explorar outros motores e compará-los entre si e com os três analisados neste projeto. *Unreal Engine*, por exemplo, seria uma ótima adição à comparação.

Além disso, os motores sempre estão mudando e cada atualização é uma oportunidade de estudar o que há de novo para o desenvolvimento de jogos. Como ocorreu com *GameMaker* na versão 2023.8, quando passou a ser possível utilizar funções de colisão com *tilemaps* (GameMaker, 2023), facilitando a implementação de uma forma que não era possível durante a execução deste projeto.

Também há fatores que impactam indiretamente os motores de jogo. Em setembro de 2023, foi anunciada uma mudança na política de monetização dos jogos desenvolvidos com *Unity* (ISAAC; BROWNING, 2023). Isso levou vários desenvolvedores a pararem de utilizar o motor e começarem a aprender outros. Uma mudança que mexe com a economia das empresas de jogos e, a longo prazo, pode impactar a evolução de *Unity* e outros motores.

Por fim, dentro do escopo deste projeto houveram recursos dos motores que não puderam ser explorados. Novas propostas de jogos ou alterações para o jogo desenvolvido podem ampliar o conhecimento de cada motor e revelar mais pontos que mostrem o potencial de cada ferramenta.

Referências

Abdullah. *C Godot Tutorials*. 2020. Acessado: 01 de fevereiro de 2023. Disponível em: <https://www.youtube.com/playlist?list=PLMgDVIa0Pg8XMe1GVc5eg0Rwi-cXqIR6q>. Citado na página 42.

Amazon Game Tech Team. *Built for Builders: AWS and Open 3D Engine – Stable 21.11 Release*. 2021. Acessado em: 10 de dezembro de 2023. Disponível em: <https://aws.amazon.com/pt/blogs/gametech/built-for-builders-aws-and-open-3d-engine-stable-21-11-release/>. Citado na página 23.

Amazon Lumberyard - Wikipedia. *Amazon Lumberyard*. 2023. Acessado em: 10 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Amazon_Lumberyard>. Citado na página 23.

ANDRADE, A. Game engines: a survey. *EAI Endorsed Transactions on Serious Games*, EAI, v. 2, n. 6, 11 2015. Citado 2 vezes nas páginas 14 e 17.

Blackthornprod. *Blackthornprod*. 2017. Acessado em: 09 de julho de 2023. Disponível em: <https://www.youtube.com/@Blackthornprod>. Citado na página 43.

Blender Game Engine - Wikipedia. *Blender Game Engine*. 2023. Acessado em: 10 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Blender_Game_Engine#cite_note-4>. Citado na página 22.

Brackeys. *Brackeys*. 2012. Acessado em: 09 de julho de 2023. Disponível em: <https://www.youtube.com/@Brackeys>. Citado na página 43.

BRAMBLE, R. *WHAT PLATFORMS CAN I EXPORT MY GAME TO WITH GAMEMAKER?* 2023. Acessado em: 17 de novembro de 2023. Disponível em: <https://gamemaker.io/en/blog/export-with-gamemaker>. Citado na página 38.

Captain Coder. *Learn You a Game Jam: Pixel Edition*. 2023. Acessado em: 05 de julho de 2023. Disponível em: <https://itch.io/jam/learn-you-a-game-jam>. Citado na página 45.

Cocos. *Cocos - The world's top 2D3D engine, game / smart cockpit /AR/VR/ virtual character / education*. 2023. Acessado em: 10 de dezembro de 2023. Disponível em: <https://www.cocos.com/en>. Citado na página 23.

COMBER, O. et al. Engaging students in computer science education through game development with unity. In: *2019 IEEE Global Engineering Education Conference (EDUCON)*. [S.l.: s.n.], 2019. p. 199–205. Citado na página 14.

Construct (game engine) - Wikipedia. *Construct (game engine)*. 2023. Acessado em: 10 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Construct_(game_engine)>. Citado na página 23.

COSTA, A. F. *GAMEMAKER: STUDIO COMO FERRAMENTA DE APRENDIZAGEM DE PROGRAMAÇÃO*. Tese de Bacharelado, 2017. Acessado em: 11 de abril de 2023. Disponível em: <https://www.comp.uems.br/~ricardo/PFCs/PFC%20188.pdf>. Citado na página 25.

COSTA, Y. Y. K. d.; MEDEIROS, L. F. d. Ensino de programação: relato de experiência sobre desenvolvimento de jogos digitais no ensino superior. *Revista Intersaberes*, v. 15, n. 34, abr. 2020. Disponível em: <https://www.revistasuninter.com/intersaberes/index.php/revista/article/view/1821>. Citado na página 14.

Croteam. *Technology - Croteam*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <http://www.croteam.com/technology/>. Citado na página 22.

CryEngine - Wikipedia. *CryEngine*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/CryEngine>. Citado na página 22.

DICKSON, P. E. et al. An experience-based comparison of unity and unreal for a stand-alone 3d game development course. In: . New York, NY, USA: Association for Computing Machinery, 2017. (ITiCSE '17), p. 70–75. ISBN 9781450347044. Disponível em: <https://doi.org/10.1145/3059009.3059013>. Citado 3 vezes nas páginas 15, 24 e 100.

DIVERS, G. *Gaming Industry Dominates as the Highest Grossing Entertainment Industry*. 2023. Acessado em: 23 de agosto de 2023. Disponível em: <https://gamerhub.co.uk/gaming-industry-dominates-as-the-highest-grossing-entertainment-industry/>. Citado na página 14.

Entertainment Software Association. *Essencial Facts About the Video Game Industry*. [S.l.], 2023. Disponível em: <https://www.theesa.com/2023-essential-facts/>. Citado na página 14.

FOWLER, A. et al. Understanding the benefits of game jams: Exploring the potential for engaging young learners in stem. In: *Proceedings of the 2016 ITiCSE Working Group Reports*. New York, NY, USA: Association for Computing Machinery, 2016. (ITiCSE '16), p. 119–135. ISBN 9781450348829. Disponível em: <https://doi.org/10.1145/3024906.3024913>. Citado na página 45.

Frostbite (game engine) - Wikipedia. *Frostbite (game engine)*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Frostbite_(game_engine)>. Citado na página 23.

Game Maker's Toolkit. *GMTK Game Jam 2023*. 2023. Acessado em: 20 de julho de 2023. Disponível em: <https://itch.io/jam/gmtk-2023>. Citado na página 45.

GameFromScratch. *The Evolution/History of the Godot Game Engine*. 2023. Acessado em: 13 de novembro de 2023. Disponível em: <https://gamefromscratch.com/the-evolution-history-of-the-godot-game-engine/>. Citado na página 31.

GameMaker. *Como Fazer Um Jogo De Arcade Clássico No GameMaker*. 2022. Acessado em: 21 de julho de 2023. Disponível em: <https://youtu.be/nwlvT-L9vFg?si=AWocYXgvzxxPCKCS>. Citado na página 43.

GameMaker. *Easy Tile Collisions | GameMaker*. 2023. Acessado em: 25 de fevereiro de 2024. Disponível em: <https://youtu.be/XxL4_a2Ci1s?si=lVVHcT-pMTQ5ILIx>. Citado na página 101.

GameMaker - Wikipedia. *GameMaker*. 2023. Acessado em: 15 de novembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/GameMaker>. Citado 2 vezes nas páginas 22 e 33.

GDQuest. *Learn to Make Games - GDQuest*. 2023. Acessado em: 01 de fevereiro de 2023. Disponível em: <https://www.gdquest.com>. Citado na página 42.

Godot Foundation. *Godot Foundation*. 2023. Acessado em: 20 de agosto de 2023. Disponível em: <https://godot.foundation>. Citado na página 30.

Godot Foundation. *Showcase - Godot Engine*. 2023. Acessado em: 12 de dezembro de 2023. Disponível em: <https://godotengine.org/showcase/>. Citado na página 32.

Godot Foundation. *Godot Engine - Free and open source 2D and 3D game engine*. 2024. Acessado em 20 de fevereiro de 2024. Disponível em: <https://godotengine.org>. Citado 4 vezes nas páginas 23, 31, 37 e 38.

Godot (game engine) - Wikipedia. *Godot (game engine)*. 2023. Acessado em: 10 de novembro de 2023. Disponível em: <en.wikipedia.org/wiki/Godot_(game_engine)>. Citado na página 31.

Godot Wild Jam. *Godot Wild Jam #58*. 2023. Acessado em: 05 de julho de 2023. Disponível em: <https://itch.io/jam/godot-wild-jam-58>. Citado na página 45.

Google. *Resultados da Pesquisa: most popular unity games*. 2023. Consulta de Pesquisa: ”most popular unity games”no Google. Acessado em: 19 de dezembro de 2023. Disponível em: <https://www.google.com/search?q=most+popular+unity+games&source=lmns&bih=650&biw=1325&client=opera-gx&hl=en&sa=X&ved=2ahUKEwjCyKOF3JyDAxVcK7kGHaI8ABgQ0pQJKAB6BAgBEAI>. Citado na página 28.

GROH, F. Gamification : State of the art definition and utilization. In: . [s.n.], 2012. Disponível em: <https://api.semanticscholar.org/CorpusID:201785646>. Citado 2 vezes nas páginas 16 e 17.

HA, T. H. *Game Development with Unreal Engine*. Bachelor’s thesis, 2022. Acessado em: 11 de abril de 2023. Disponível em: <https://www.theseus.fi/handle/10024/751778>. Citado na página 25.

Heartbeast. *Godot Action RPG Series*. 2020. Acessado em: 01 de fevereiro de 2023. Disponível em: <https://youtube.com/playlist?list=PL9FzW-m48fn2SlrW0KoLT4n5egNdX-W9a&si=W5-EJthpo0-sq2ZB>. Citado 2 vezes nas páginas 42 e 47.

HELGASON, D. *A free Unity?* 2009. Acessado em: 08 de novembro de 2023. Disponível em: <https://blog.unity.com/technology/a-free-unity>. Citado na página 27.

id Tech 3 - Wikipedia. *id Tech 3*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Id_Tech_3>. Citado na página 22.

id Tech 4 - Wikipedia. *id Tech 4*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Id_Tech_4>. Citado na página 22.

ISAAC, M.; BROWNING, K. How a pricing change led to a revolt by unity's video game developers. *The New York Times*, 2023. Acessado em: 22 de novembro de 2023. Disponível em: <https://www.nytimes.com/2023/10/02/technology/how-a-pricing-change-led-to-a-revolt-by-unitys-video-game-developers.html?smid=url-share>. Citado na página 101.

itch.io. *Most used Engines*. 2023. Acessado em: 23 de outubro de 2023. Disponível em: <https://itch.io/game-development/engines/most-projects>. Citado na página 26.

JENSEN, K. T. *25 Years Later: The History of Unreal and an Epic Dynasty*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://www.pcmag.com/news/25-years-later-the-history-of-unreal-and-an-epic-dynasty>. Citado na página 22.

Jogo Coletivo. *Lista de jogos selecionados para Mostra teu Jogo*. 2023. Instagram. Disponível em: <https://www.instagram.com/p/CyJ3zeJrv_c/?utm_source=ig_web_copy_link&igsh=MzRlODBiNWFlZA==>. Citado na página 46.

KORANNE, H. *History of Unity3D*. 2021. Acessado em: 07 de novembro de 2023. Disponível em: <https://www.linkedin.com/pulse/history-unity3d-harsh-koranne/>. Citado na página 27.

LINIETSKY, J. Godot history in images! 2014. Acessado em: 20 de agosto de 2023. Disponível em: <https://godotengine.org/article/godot-history-images/>. Citado na página 30.

LINIETSKY, J.; MANZUR, A.; CONTRIBUTORS. *Exporting projects*. 2023. Acessado em: 17 de novembro de 2023. Disponível em: <https://docs.godotengine.org/en/stable/tutorials/export/exporting_projects.html>. Citado na página 38.

LINIETSKY, J.; MANZUR, A.; CONTRIBUTORS. *Godot Docs*. 2023. Acessado em: 20 de dezembro de 2023. Disponível em: <https://docs.godotengine.org/en/stable/>. Citado na página 42.

LOWOOD, H. Game engines and game history. In: *History of Games International Conference Proceedings*. [S.l.: s.n.], 2014. p. 2014. Citado 2 vezes nas páginas 21 e 24.

MATTOS, M. et al. Uma pesquisa-ação sobre o desenvolvimento do pensamento computacional com crianças. In: *Anais do XXIV Workshop de Informática na Escola*. Porto Alegre, RS, Brasil: SBC, 2018. p. 421–429. Disponível em: <https://sol.sbc.org.br/index.php/wie/article/view/14354>. Citado na página 14.

Panda3D - Wikipedia. *Panda3D*. 2023. Acessado em: 10 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Panda3D>. Citado na página 22.

PAUL, P.; GOON, S.; BHATTACHARYA, A. History and comparative study of modern game engines. *International Journal of Advanced Computer and Mathematical Sciences*, v. 3, p. 2230–9624, 01 2012. Citado 2 vezes nas páginas 22 e 23.

PAVKOV, S.; FRANKOVIć, I.; HOIć-BOžIć, N. Comparison of game engines for serious games. In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. [S.l.: s.n.], 2017. p. 728–733. Citado 3 vezes nas páginas 15, 25 e 100.

PöNNI, O. *Starting A Beginner Game Project With GameMaker Studio 2*. Bachelor's thesis, 2021. Acessado em: 12 de abril de 2023. Disponível em: <https://www.theseus.fi/handle/10024/495212>. Citado na página 25.

RenderWare - Wikipedia. *RenderWare*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/RenderWare>. Citado na página 22.

Rockstar Advanced Game Engine - Wikipedia. *Rockstar Advanced Game Engine*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Rockstar_Advanced_Game_Engine>. Citado na página 23.

SALMELA, T. *Game Development using the open-source Godot Game Engine*. Bachelor's thesis, 2022. Acessado em: 17 de fevereiro de 2023. Disponível em: <https://www.theseus.fi/handle/10024/746943>. Citado na página 25.

SINGH, S.; KAUR, A. Game development using unity game engine. In: *2022 3rd International Conference on Computing, Analytics and Networks (ICAN)*. [S.l.: s.n.], 2022. p. 1–6. Citado na página 25.

SteamDB. *What are games built with and what technologies do they use?* 2023. Acessado em: 20 de outubro de 2023. Disponível em: <https://steamdb.info/tech/>. Citado na página 26.

The Strong National Museum of Play. *Video Game History Timeline*. 2023. Acessado em: 21 de novembro de 2023. Disponível em: <https://www.museumofplay.org/video-game-history-timeline/>. Citado 2 vezes nas páginas 21 e 23.

Torque (game engine) - Wikipedia. *Torque (game engine)*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://en.wikipedia.org/wiki/Torque_(game_engine)>. Citado na página 22.

Unity (game engine) - Wikipedia. *Unity (game engine)*. 2023. Acessado em: 08 de novembro de 2023. Disponível em: <en.wikipedia.org/wiki/Unity_(game_engine)>. Citado 2 vezes nas páginas 23 e 27.

Unity Technologies. *Plataforma de desenvolvimento em tempo real do Unity | Engine para 3D, 2D, VR e AR*. 2023. Acessado em: 15 de novembro de 2023. Disponível em: <unity.com>. Citado 2 vezes nas páginas 37 e 38.

Unity Technologies. *System requirements for Unity 2023.1*. 2023. Acessado em: 15 de novembro de 2023. Disponível em: <https://docs.unity3d.com/2023.1/Documentation/Manual/system-requirements.html>. Citado na página 38.

Unity Technologies. *Unity Documentation*. 2023. Acessado em: 09 de julho de 2023. Disponível em: <https://docs.unity.com>. Citado na página 43.

Unity Technologies. *Learn game development w/ Unity | Courses tutorials in game design, VR, AR Real-time 3D | Unity Learn*. 2024. Acessado em: 28 de janeiro de 2024. Disponível em: <https://learn.unity.com>. Citado na página 43.

Unity Technologies. *Unity download archive*. 2024. Acessado em: 28 de fevereiro de 2024. Disponível em: <https://unity.com/releases/editor/archive>. Citado na página 28.

Valve Corporation. *Valve Developer Community*. 2023. Acessado em: 08 de dezembro de 2023. Disponível em: <https://developer.valvesoftware.com/wiki/Main_Page>. Citado na página 22.

VOHERA, C. et al. Game engine architecture and comparative study of different game engines. In: *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. [S.l.: s.n.], 2021. p. 1–6. Citado 2 vezes nas páginas 14 e 24.

XenForo. *GameMaker Community*. 2023. Acessado em: 15 de agosto de 2023. Disponível em: <https://forum.gamemaker.io/index.php>. Citado na página 44.

YoYo Games. *GameMaker Notícias, Artigos e Tutoriais*. 2023. Acessado em: 15 de agosto de 2023. Disponível em: <https://gamemaker.io/pt-BR/blog>. Citado na página 43.

YoYo Games. *GameMaker Showcase | Games Made With GameMaker*. 2023. Acessado em: 12 de dezembro de 2023. Disponível em: <https://gamemaker.io/en/showcase>. Citado na página 35.

YoYo Games. *Make 2D Games With GameMaker | Free Video Game Maker*. 2023. Acessado em: 17 de novembro de 2023. Disponível em: <gamemaker.io>. Citado 2 vezes nas páginas 37 e 38.

YoYo Games. *GameMaker Release Notes*. 2024. Acessado em: 20 de fevereiro de 2024. Disponível em: <https://gms.yoyogames.com/ReleaseNotes.html>. Citado na página 34.

ŠMíD, A. *Comparison of Unity and Unreal Engine*. Bachelor Project — Czech Technical University in Prague, 2017. Acesso em: 12 abril 2023. Disponível em: <https://dcgi.fel.cvut.cz/theses/2017/smidanto>. Citado 2 vezes nas páginas 15 e 25.