Análise Comparativa entre Game Engines para jogos 2D: Unity e GameMaker

Christian Ricardo Souza Velani, Jean Gaioto Reis, Valeria Maria Volpe (orientadora)

e-mail:
christian.velani@fatec.sp.gov.br, jean.reis@fatec.sp.gov.br, valeria.volpe@fatec.sp.gov.br

Resumo: Atualmente o acesso à informação e a conhecimentos se tornou fácil e rápido. Com o mercado de jogos eletrônicos extremamente aquecido, se torna cada vez mais comum encontrar pessoas querendo criar seus próprios jogos. Com isso, diversas ferramentas para desenvolver jogos, de forma ágil, surgiram e estão disponíveis para uso com ou sem custo. Escolher qual ferramenta utilizar é uma tarefa difícil, devido às características de cada uma. Este trabalho procurou realizar uma comparação entre duas *engines* mais populares no mercado, *Unity* e *GameMaker*. O objetivo principal foi comparar os mais variados aspectos de ambas durante o processo de desenvolvimento de um jogo, o qual foi o mesmo em ambas. Buscou-se ter como resultados as vantagens e desvantagens de cada uma das *engines*, tantos nos aspectos funcionais quanto em seu nível de complexidade de uso e compreensão.

Palavras-chave: Desenvolvimento de Jogos usando *Unity* e *GameMaker*. Comparação entre *Unity* e *GameMaker*. Comparação de ferramentas de desenvolvimento de Jogos 2D. Vantagens e Desvantagens de *Unity* e *GameMaker* para desenvolvimento de jogos.

Abstract: *In current times where access to information and knowledge is easy and fast, and with the electronic games market extremely heated, the reality of people wanting to create their own independent games is becoming increasingly common. As a result, several tools for developing games quickly appeared in the community. Choosing which one to use is a challenging task. This work seeks to make a comparison between two most popular engines on the market, Unity and Gamemaker. The main objective is to compare distinct aspects of both during the game development process, which is the same in both. The aim is to obtain as results the advantages and disadvantages of each of the engines, both in functional aspects and in their level of complexity of use and understanding.*
Keywords: *Game Development using Unity and GameMaker. Comparison between Unity and GameMaker. Comparison of 2D Game Development Tools. Advantages and Disadvantages of Unity and GameMaker for Game Development.*

1 Introdução

A indústria de desenvolvimento de jogos eletrônicos é um dos segmentos mais dinâmicos e promissores do mercado de entretenimento, proporcionando experiências interativas de alta qualidade para uma audiência global ávida por inovação e entretenimento. Dentro desse cenário, os jogos 2D têm se destacado como uma forma de expressão artística e criativa, oferecendo aos desenvolvedores e jogadores um ambiente estilisticamente único, muitas vezes marcado pela nostalgia e jogabilidade cativante. No entanto, a criação de jogos

2D eficaz requer ferramentas apropriadas e, nesse contexto, entram as *game engines,* que desempenham um papel importante.

Este trabalho se propôs a realizar uma análise comparativa entre as duas das g*ame engines* mais proeminentes na criação de jogos 2D: *Unity* e *GameMaker*. A escolha das *game engine*s foi motivada pela necessidade de entender as nuances e a capacidade destas ferramentas, e o impacto que elas podem ter no desenvolvimento de jogos 2D. A compreensão dessas *engine*s podem, não apenas informar os desenvolvedores sobre a melhor escolha para seus projetos, mas também contribuir para a literatura acadêmica no campo do desenvolvimento de jogos.

Para que fosse feito essa análise e comparação foram definidos critérios baseados no que o mercado de desenvolvimento de jogos e as pessoas que atuam nessa área normalmente precisam. Por meio desses critérios foi possível notar a necessidade do desenvolvimento de um jogo em cada uma das ferramentas escolhidas, para melhor compreensão do funcionamento delas, além da criação de um quadro comparativo, colocando lado a lado cada aspecto avaliado.

Após a análise e comparação realizada foi possível notar que cada ferramenta tem seus pontos positivos e negativos, aos quais se destacam: a *Unity* é uma ferramenta mais completa pois suporta o desenvolvimento de jogos 2D e 3D além de ser muito mais utilizada no mercado, o que acarreta um caminho de aprendizado mais complexo, porém mais prático; já em relação ao *Game Maker* foi possível constatar que esta é uma ferramenta mais atrativa para iniciantes e por ter o foco somente em jogos 2D, deixa o desenvolvimento mais simples. Também é possível notar que a *GameMaker* é muito mais fácil de utilizar, desde sua instalação até seu interfaceamento, apesar de se ter falta de recursos em suas licenças mais básicas.

Atualmente o mundo dos jogos eletrônicos é um dos mais rentáveis e mais queridos da população, de tal forma, que tem ocasionado um aumento no interesse de desenvolvimento de jogos. Sendo assim, tornou-se interessante realizar a análise dessas duas ferramentas disponíveis no mercado para saber qual é melhor para o projeto, com usuário iniciante.

A escolha das *game engine* impacta muito na produção do jogo e no sucesso dele, de tal forma, que a escolha "incorreta" pode acarretar atrasos ou até mesmo ter que abandonar o desenvolvimento e recomeçar a produção em outra ferramenta. Assim, reforçando ainda mais a necessidade de uma análise criteriosa das *games engine*s disponíveis antes de se iniciar o projeto.

Em busca de conhecer e comparar essas ferramentas foram definidos os seguintes objetivos:

- Comparar os recursos e funcionalidades de cada motor gráfico;
- Comparar o desempenho do motor gráfico;
- Comparar o desempenho do jogo final;
- Comparar a facilidade de uso;
- Comparar o melhor para fazer jogos em 2d;
- Analisar a complexidade de uso e compreensão baseado no nível de conhecimento prévio do usuário.

2 Fundamentação Teórica

Na fundamentação teórica será abordada a teoria necessária para a compreensão e desenvolvimento deste trabalho. Nas Seções 2.1 estão expostos os conceitos gerais de jogos 2D, descrevendo como são jogos em segunda dimensão. Na Seção 2.2, apresenta-se toda a parte conceitual necessária em *engine* de desenvolvimento de jogos. Na seção 2.3 e 2.4 será tratado sobre ambas *games engine*s que foram utilizadas para este trabalho, sendo elas: *Unity* e *GameMaker.*

2.1 Jogo 2D

Os jogos 2D representam uma categoria distinta no mundo dos videogames, caracterizada por ambientes, personagens e elementos gráficos que são representados em um plano bidimensional, sem a profundidade tridimensional típica dos jogos 3D: "[...] um programador faz uma versão muito abstrata dessa mecânica de jogo, talvez em 2D, com formas geométricas simples em vez de personagens animados." Schell (2008, p. 84).

Os jogos 2D têm sido uma parte significativa e querida da indústria de jogos por décadas, destacando-se por sua simplicidade estilística e foco em mecânicas de jogabilidade acessíveis. A estética de um jogo 2D é frequentemente marcada por gráficos bidimensionais, muitas vezes pixelados, que evocam uma sensação de nostalgia para muitos jogadores.

2.2 *Game Engine*

Uma *game engine*, ou motor de jogo, é uma infraestrutura de *software* que permite a criação, desenvolvimento e execução de jogos eletrônicos. Ela serve como a espinha dorsal técnica de um jogo, proporcionando ferramentas, bibliotecas e funcionalidades para simplificar o processo de criação, enquanto gerencia tarefas complexas, como renderização gráfica, física, áudio e interação do jogador. A escolha da *game engine* certa é crucial para o sucesso de um projeto de desenvolvimento de jogos, e a compreensão das características distintivas das diversas *engine*s disponíveis é de suma importância.

Hocking (2015, p. 4) diz sobre as *game engine*:

> Uma *game engine* de jogo fornece uma infinidade de recursos que são úteis em muitos ambientes diferentes jogos, então um jogo implementado usando essas *game engines* obtém todos esses recursos ao adicionar recursos de arte personalizados e código de jogo específico para esse jogo.

2.3 *Unity*

Este trabalho se propõe a realizar uma análise comparativa entre duas *Game Engines* amplamente reconhecidas na indústria de jogos: a *Unity* e o *GameMaker*. A *Unity* é uma das

Game Engines mais populares e versáteis, conhecida por sua capacidade de criar jogos 2D e 3D de alta qualidade: “[...]*Unity* tem simulação de física, mapas normais, oclusão ambiental do espaço da tela, sombras dinâmicas… e a lista continua.” Hocking (2015, p. 4).

Com um poderoso conjunto de ferramentas e suporte multiplataforma, a *Unity* ganhou destaque na criação de jogos para várias plataformas, incluindo PC, consoles, dispositivos móveis e realidade virtual. Sua flexibilidade e vasta comunidade de desenvolvedores a tornaram uma escolha preferencial para muitos estúdios e criadores independentes.

2.4 *GameMaker*

Por outro lado, o *Game Maker* é uma *Game Engine* conhecida por sua acessibilidade e facilidade de uso. Desenvolvido pela *YoYo Games* (1999), é uma escolha popular para iniciantes e desenvolvedores independentes que desejam criar jogos 2D. O *GameMaker* oferece uma interface de arrastar e soltar, permitindo que criadores sem experiência em programação desenvolvam jogos com relativa facilidade. Seu foco em jogos 2D e suas capacidades de prototipagem rápida atraem muitos desenvolvedores que buscam concretizar suas ideias de jogo de forma eficaz: “[...]Game Maker é ideal para aprender o desenvolvimento de jogos, pois permite que você comece fazendo jogos sem ter que estudar um idioma completamente novo.” Habgood e Overmars (2006, p. 14).

. A análise comparativa entre a *Unity* e o *GameMaker* explora as vantagens e desvantagens de ambas as e*ngine*s, considerando fatores como funcionalidades, desempenho, curva de aprendizado e flexibilidade, com o objetivo de auxiliar desenvolvedores na escolha do melhor *game engine* para seus projetos 2D.

3 Trabalhos Similares

3.1 Estudo comparativo entre *engine*s de desenvolvimento de jogos 2D

No trabalho, o autor Gelderson (2021) tinha como objetivo realizar a comparação entre as game *engine*s *Unity* e *Construct 3* através do desenvolvimento de uma game 2D

Para fazer a comparação foi definido como metodologia que primeiro seria feito o GDD (*Game Design Document*) que é as informações do jogo a ser feito e as métricas que seriam analisadas. Após a definição das métricas, o jogo foi construído em ambas as *engine*s, o qual o processo foi descrito detalhadamente para cada uma das métricas.

O resultado obtido foi que a *Construct 3* se mostrou muito simples de se usar e a técnica *drag and drop* se tornou bem evidente durante o processo de criação. Já a *Unity* se mostrou mais completa e com isso o processo de desenvolvimento se tornou mais complexo, mas também disponibilizando mais opções de desenvolvimento.

É enfatizado nas considerações finais o fato de que, para um iniciante, a *Construct 3* é a melhor opção devido sua facilidade e seu recurso de *drag and drop*. Já para quem busca um jogo com mais detalhes e mais controle de aspectos técnicos a melhor opção se torna a *Unity*.

O trabalho em questão se mostrou bem escrito e com boas métricas, além de uma conclusão satisfatória para os objetivos buscados. Além do fato de que a análise das *engine*s proporciona uma boa base de referência para trabalhos futuros semelhantes ao mesmo.

O fato do detalhamento de como foi feito cada métrica dentro das *engine*s proporciona um bom material de estudo para caso seja necessário utilizar uma métrica semelhante, assim tornando um trabalho relevante para quem busca entender e/ou desenvolver jogos.

3.2 Comparativo entre Game Engines como Etapa Inicial para o desenvolvimento de um Jogo de Educação Financeira

O trabalho de Cavalcante e Pereira (2018), começa introduzindo como a educação financeira é importante e como a população não é educada financeiramente, além de mencionar como a gamificação ajuda no processo de aprendizado. É definido como objetivo definir uma game *engine* para ser utilizada para a criação de um jogo voltado para a educação financeira sem conhecimento especializado em programação.

Para o trabalho foi escolhido as game *engine*s *Unity*, *Godot* e *Phaser*, e a metodologia utilizada foi que as 3 game *engine*s iram ser avaliadas através de 3 etapas, sendo elas Análise da Interface e Funcionalidades, Desenvolvimento de um Jogo e Vantagens e Desvantagens.

Após o processo de fazer as 3 etapas nas *engine*s os autores chegaram como conclusão de que, para atingir os objetivos deles, a melhor opção seria o Godot por ser mais leve, ter a possibilidade de exportar jogos para diversas plataformas e sua interface e linguagem de programação serem amigáveis.

Apesar de ser um trabalho não muito longo e que não realizou uma análise muito profunda das *engine*s se torna interessante por ser uma visão diferente delas, por analisar mais superficialmente e em busca de descobrir qual é melhor para utilização de alguém que não é familiarizado com a programação.

O trabalho traz um pequeno “defeito” que é o fato para cada *engine* foi desenvolvido um jogo diferente, o que pode acarretar uma falsa facilidade ou dificuldade de produção, já que mesmo tendo elementos semelhantes nos 3 jogos, por seus “gêneros” serem diferentes pode acontecer de um jogo ser mais simples de ser produzido do que o outro.

Como já relatado antes, devido ao tamanho curto do trabalho que foi realizado para um congresso, ele não traz muitas informações sobre as *engine*s e nem sobre o desenvolvimento de jogos, mas é sempre importante ter uma visão diferente sobre um assunto, ainda mais da área de tecnologia, pois as ferramentas não são produzidas só para quem tem um grande conhecimento.

3.3 Estudo comparativo entre as *game engines* Unity e Ogre

O trabalho tem como objetivo fazer uma comparação entre as game *engine*s *Unity* e *Ogre*, COSTA(2016) , e para isso, primeiro os autores fazem uma introdução desde os primórdios das game *engine*s que eram em código puro, então comentam sobre as ferramentas

conhecidas como editores de *levels* e finalizam a introdução falando sobre os *MODs*(*modifications*) que são modificações feitas por fãs de jogos famosos, as quais algumas delas geraram novos jogos, como no caso do jogo Counter-Strike que nasceu como um *mod* do jogo *Half Life* e que se tornou um grande sucesso que a empresa dona do jogo original comprou o *mod* e o tornou um jogo oficial.

Após a introdução da história das game *engine*s, é explicado sobre as *engine*s que serão analisadas e então é feita uma análise sobre elas, onde é relatado que a *Unity* é uma ferramenta profissional já a *Ogre* é mais voltada para ambientes acadêmicos.

O trabalho em si não enfatiza muito a análise das *engine*s, o foco maior é em contar a história por trás delas, que mesmo sem o foco nas *engine*s em si pode ser de auxílio, pois saber da história por trás de uma ferramenta ou uma série de ferramentas é tão importante quanto entender das próprias ferramentas.

Apesar de ser um bom trabalho sobre o histórico das games *engine*s, o título leva o leitor a outra compreensão sobre o que o texto apresentará, dando a entender que o trabalho fará a comparação entre as ferramentas de forma mais profunda, mas a comparação é curta e superficial.

4 Metodologia

O processo de análise e comparação foi divido em quatro partes: Análise inicial das *game engines,* Desenvolvimento de um jogo dentro delas, Construção de um quadro comparativo.

Para que seja possível a realização de todas as partes foi escolhida uma série de critérios a serem utilizados durante todo o processo, sendo eles podendo ser divididos em algumas categorias.

Os critérios foram divididos entre: Facilidade de Uso, Desempenho, Ferramentas e Recursos, Licenciamentos e custos e outros. Para cada uma das categorias foram escolhidos alguns critérios baseados em pesquisas de mercado e conhecimento prévios dos envolvidos no trabalho.

Os critérios escolhidos para serem utilizados durante o desenvolvimento são:

- Facilidade de Uso: Curva de aprendizado, Documentação e Suporte, Recursos de Aprendizado, Comunidade, Instalação, Interfaceamento intuitivo
- Desempenho: Otimização
- Ferramentas e recursos: Integração com sistema de áudio, Animação de sprites, Recursos de depuração e edição de código, Recursos para agilizar o processo de criação
- Licenciamento e Custos: Preços de Licenças, Recursos de cada licença
- Outros: Reutilização de códigos antigos

4.1 Análise Inicial das *Game Engines*

Nessa etapa do processo irá ser feita uma análise superficial de ambas as *game engines* escolhidas em busca de gerar uma visão inicial sobre cada uma delas. Para isso os critérios de Instalação, Interfaceamento Intuitivo, Preços de Licenças e Recursos de cada licença, foram utilizados. A análise foi feita a partir da experiência adquirida com a pesquisa e utilização das ferramentas.

4.2 Desenvolvimento do Jogo

Nessa etapa foram construídos dois jogos semelhantes, um em cada *game engine* para testar as funcionalidades delas, em busca de preencher os critérios: Documentação e Suporte, Recursos de Aprendizado, Comunidade, Otimização, Integração com sistema de áudio, Animação de *Sprites*, Recursos de depuração e edição de código, Recursos para agilizar o processo de criação, Curva de Aprendizado e Reutilização de códigos antigos.

Por meio dessa etapa foi possível obter conhecimento mais profundo de como cada *game engine* funciona e seus devidos recursos para que assim possa ser gerado uma visão geral melhor sobre elas para o desenvolvimento da comparação.

4.3 Quadro Comparativo

Nessa etapa foi criado um quadro comparativo dando valores numéricos para como cada ferramenta, para que se possa identificar se cada *game engine* está de acordo com a visão dos autores, no que diz respeito ao processo de análise, além de listar os principais pontos positivos e negativos de cada uma delas. Também foi criado outros métodos para mostrar essa comparação usando gráficos.

5 Desenvolvimento

5.1 Análise Inicial das *game engines*

Nessa etapa foi realizado uma análise inicial dos aspectos não relacionados ao desenvolvimento do jogo em si, e sim das licenças, instalação e o interfaceamento de cada ferramenta em busca de uma visão inicial delas.

5.2 Unity

A Unity é uma das ferramentas mais utilizadas no mercado de desenvolvimento de jogos nos dias de hoje, fornecendo ferramentas que auxiliam no desenvolvimento de inúmeros

tipos de jogos, desde jogos 2D, 3D, Mobile até jogos de VR/AR e é utilizado na indústria dos cinemas.

5.2.1 Licenças

A Unity conta com 4 planos de pagamento de licença:

- *Personal;*
- *Pro;*
- *Enterprise;*
- *Industry*.

Cada um com seus devidos benefícios e seus requisitos para se manter no mesmo.

O plano *Personal* é o plano para pessoas individuais e pequenas empresas com custo zero assim disponibiliza o básico para a criação e disponibilização do jogo, como por exemplo a plataforma de desenvolvimento em tempo real do Unity, Scripts visuais no Unity, também disponibiliza todos os recursos disponíveis no Unity Cloud, exceto pelo Unity DevOps, que é disponibilizado apenas o plano gratuito dele.

Além desses recursos mais básicos, ainda no *Personal* ela disponibiliza *Cloud Diagnostics,* o qual é uma ferramenta para gerar relatórios de erros, também é disponibilizado a integração com ferramentas de colaboração só que somente pode ser escolhida uma ferramenta

Em relação a monetização o Unity Personal, dá direito ao Unity Ads e o plugin de compras no aplicativo, ou seja, maneiras de monetizar o jogo por dentro dele. Já em relação a suporte, a Unity não oferece nenhum tipo de suporte especializado e treinamentos para o plano Personal.

É importante notar o fato de que para se manter no plano *Personal*, caso o indivíduo ou a pequena empresa vá utilizar da Unity para fornecer serviços para terceiros, os clientes tem que ter receitas ou fundos arrecadados nos últimos 12 meses menor que 100 mil dólares, já para aqueles que não fornecem para terceiros, em caso de pequenas empresas, a receita bruta agregada e financiamento inferiores a 100 mil dólares e para indivíduos e amadores o valor gerado em conexão com o suo do Unity for inferior a 100 mil dólares.

Indo para o plano *Pro*, já é um plano um pouco mais robusto, custando R$11.791,00 por ano para ter acesso a mais recursos como a personalização da tela inicial, ajuda na distribuição de jogos em diversas lojas, ferramentas de criação de jogos AR/MR, Havok Physics entre outras coisas que já vem no plano mais algumas que podem ser adicionadas através de um custo a mais.

E o último plano voltado para o desenvolvimento de jogos é o *Enterprise*, feito para equipes sérias de qualquer tamanho com projetos complexos, com valores em média de 200 dólares por mês por pessoa, sendo que o plano só disponibilizado para compra a partir de 20 licenças. Ele permite a utilização de todos os recursos que a Unity disponibiliza.

Dentre eles, será utilizado o personal, devido a não ter custos iniciais e ser o que provavelmente alguém que está começando no mercado agora irá utilizar, ou seja, é o mais "comum" de ser utilizado.

5.2.2 Instalação

Para baixar o Unity é muito simples, no site deles eles deixam de maneira simples de se encontrar o botão de download do instalador, como pode ser visto na figura 1.

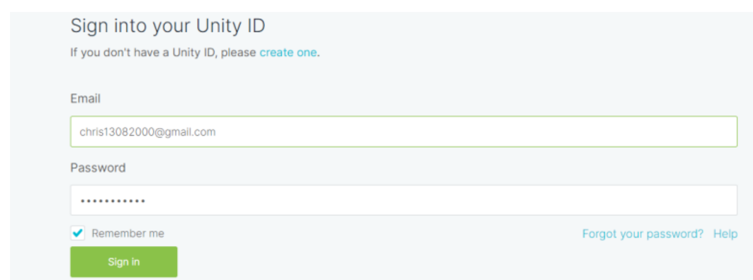
Figura 1 Baixe o Unity

Fonte: Site da Unity

Após baixar o instalador no site da Unity, o processo inicial de instalação é simples, é só seguir avançando no instalador sem muitas escolhas para serem feitas, como pode ser visto nas figuras 2 e 3, somente aceita-se os termos e então já escolhemos onde será instalado e já o instalamos. Ele irá instalar o Unity Hub, que é uma aplicação que permite gerenciar as versões instaladas do Unity Editor além de acessar conteúdos disponibilizados nas abas de aprendizado e comunidade.

Ao terminar a instalação do Hub, você deve entrar com sua conta da Unity para que assim possa acessar o mesmo.

Figura 2 Entrando na Unity



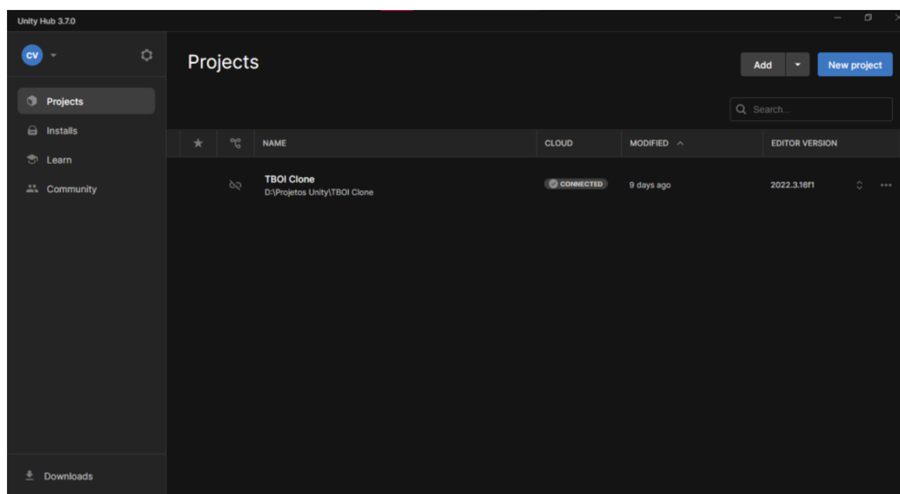
Fonte: Unity Editor

Após entrar na plataforma tem-se acesso ao HUB, onde pode-se então fazer o *download* de alguma versão do Unity Editor, o qual é a ferramenta de criação de jogos.

5.2.3 Interfaceamento

O interfaceamento da Unity HUB é bem simples e intuitivo, sendo dividido em 4 abas principais e a área de download. Já no Editor ela se mostra mais complexa e com muitas abas, além de poder ter seu layout totalmente customizável, deixando com a aparência que o usuário querer

Figura 3 Interface do Unity HUB - Projetos

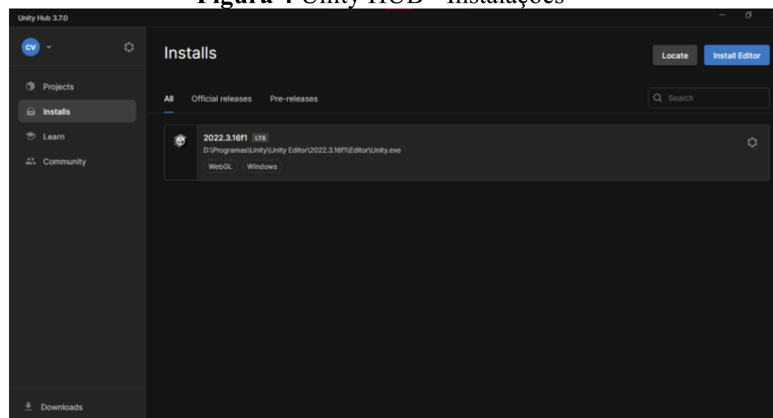


Fonte: Unity Hub - Projetos

A primeira interface que um usuário comum irá ver ao instalar o Unity e fazer o login no mesmo é a do Unity HUB, o gerenciador de instalação do Unity Editor, o qual é dividido em 4 abas: Projetos, Instalações, Aprender e Comunidade.

Na aba projetos, como o próprio nome sugere, é onde gerencia-se os projetos, podendo criar, excluir e editar os já criados. Na aba instalações (figura 4), pode-se gerenciar as versões do Unity Editor, seja as instaladas como as novas instalações.

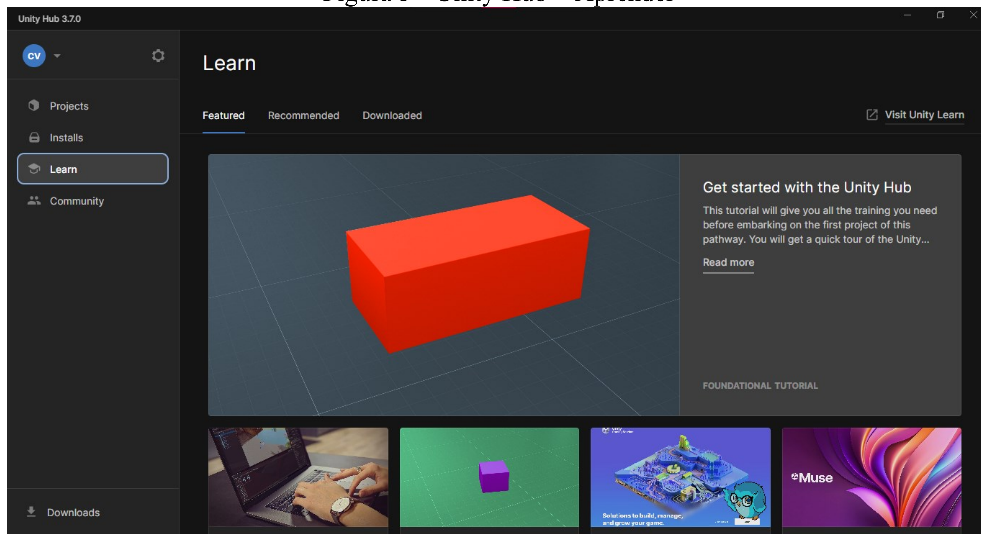
Figura 4 Unity HUB - Instalações



Fonte: Unity HUB - Instalações

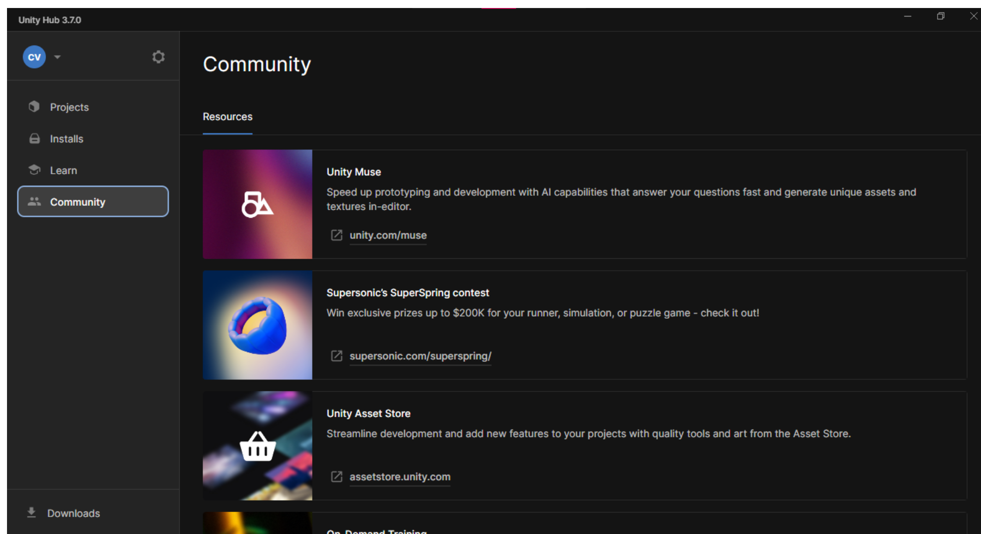
Já na aba aprender (figura 5) temos links úteis relacionados ao Unity Learn, que é a plataforma de aprendizado da Unity, e por último na aba Comunidade (figura 6) temos links úteis no geral para recursos, novidades e outras informações relacionadas a Unity.

Figura 5 - Unity Hub – Aprender



Fonte: Unity HUB - Instalações

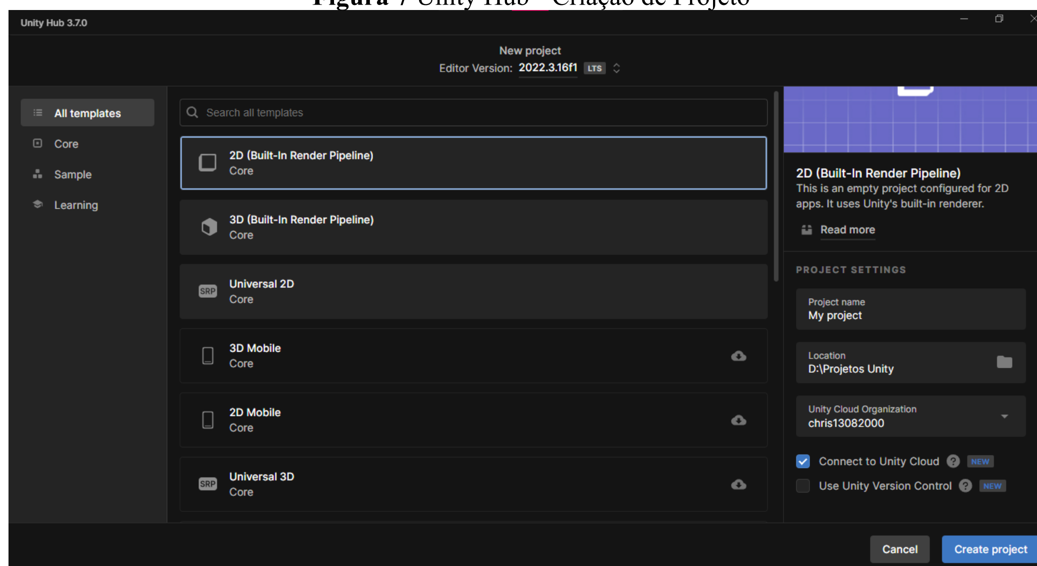
Figura 6 Unity HUB - Comunidade



Fonte: Unity HUB - Comunidade

Ainda no HUB temos a interface de criação de projetos, onde pode-se definir as configurações iniciais do projeto baseado nos *templates* que a Unity oferece, além de o local onde será armazenado e recursos extras como o Unity Cloud e Unity Version Control, como pode ser visto na figura 7.

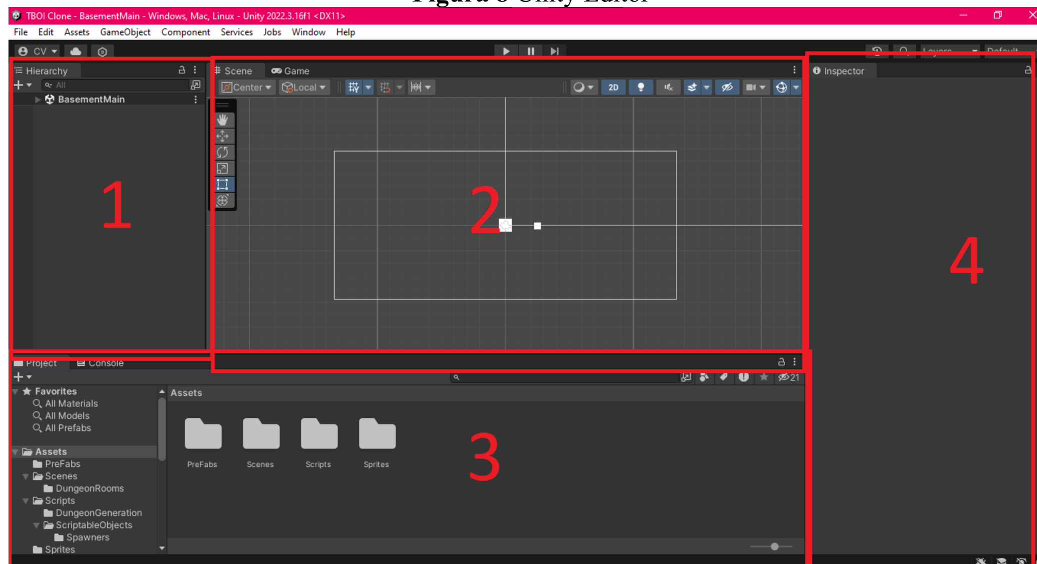
Figura 7 Unity Hub - Criação de Projeto



Fonte: Unity HUB – Comunidade

Outra interface importante de se analisar é a do editor, onde cria-se o jogo. Para fins da realização desse trabalho foi utilizado a versão 2022.3.16f1 do Unity Editor, o qual a interface pode ser vista na figura 8.

Figura 8 Unity Editor



Fonte: Unity Editor

A interface do editor é dividida, inicialmente no seu layout padrão, em quatro setores:

1. Hierarquia – Mostra os componentes que estão na “cena” que está selecionada
2. *Game* e Cena – Pré-visualização de como fica a tela do jogo, onde pode-se jogar o mesmo e tela de edição da cena, para colocar e remover elementos na sala atual
3. Projeto e Console – Pastas e arquivos do projeto e console para análise de erros.

4. Inspetor – Local para editar as propriedades dos componentes da cena

No geral a interface da Unity é bem intuitiva na questão do HUB, só que no editor pode gerar um primeiro impacto, pois são muitas "abas" com muitas opções, de tal maneira que pode deixar meio confuso o entendimento inicial, só que é importante ressaltar que a Unity tem muitos tutoriais, os quais ajudam a compreender melhor o que tem em cada uma dessas abas e para que usar cada uma delas.

5.3 GameMaker

A *Game Maker* é uma ferramenta de desenvolvimento de jogos, ideal para criar jogos 2D para diversas plataformas. Ele oferece uma interface intuitiva, tornando o acessível tanto para iniciantes quanto para desenvolvedores experientes.

5.3.1 Licença

A *Game Maker* possui no total 3 licenças diferentes:

- *Free;*
- *Professional;*
- *Enterprise*;

Onde vai aumentando os benefícios que cada uma vai oferecendo.

Quando é baixado o *Game Maker* pelo próprio site deles, ele já vem com a licença *Free*, a qual é uma licença não comercial, ou seja, o jogo desenvolvido não pode ser usado para a obtenção de lucros. Ele possui acesso a diversas ferramentas e recursos da GameMaker Studio 2, também possui uma criação e exportação de jogos para Windows, Linux, macOS e dispositivos móveis, com um limite de 100.000 downloads por plataforma.

Por ser gratuito ele possui certas desvantagens como: a marca d'água do GameMaker que será exibida na inicialização do jogo, suporte técnico é limitado para fóruns da comunidade, possui o código-fonte fechado e não há acesso a módulos da Steam, consoles entre outros. Sendo mais utilizado por estudantes ou aqueles que desejam experimentar a plataforma.

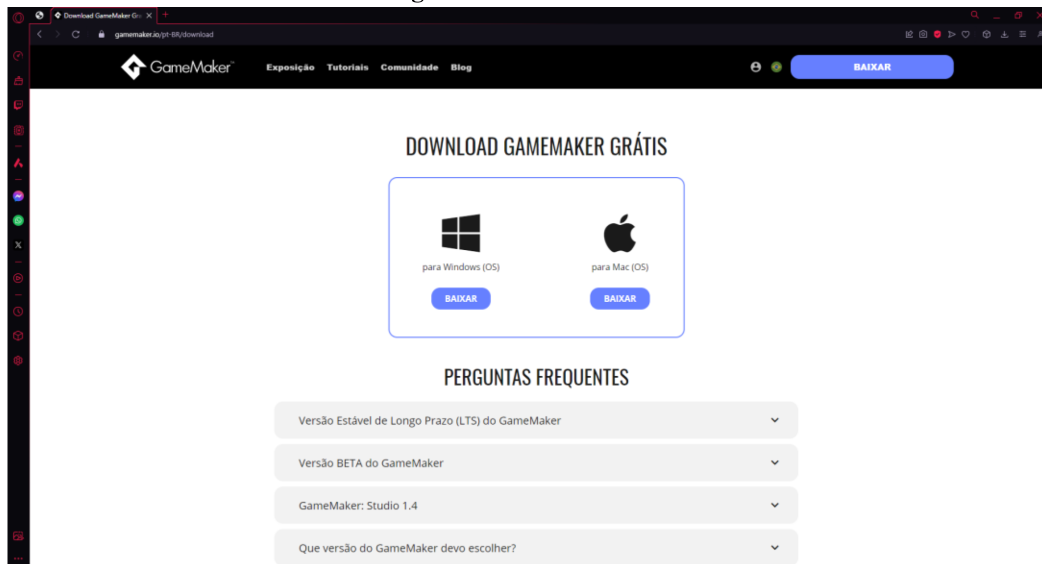
O plano *Professional* é necessário pagar pelo menos uma vez para obtê-lo, custando em torno de R$269,99, possui as mesmas ferramentas e recursos do *Free*, só com a diferença dele ser uma licença comercial, em que é permitido a venda de seus jogos no qual é preciso pagar royalties de 10% sobre a receita bruta acima de R$5000,00 por ano. Nesse plano agora há suporte técnico por e-mail e não possui as desvantagens que a versão gratuita tem.

Tendo as mesmas benefícios que a versão *Professional*, a versão *Enterprise*
é preciso ser paga mensalmente por um valor de R$139,99 ou R$1399,90 ao ano, tem como principal diferença da versão *Professional* o suporte personalizado, a exportação para consoles e não possuindo royalties.

5.3.2 Instalação

Para baixar a *GameMaker* é necessário acessar o site e acessar a parte de *downloads* e escolher para qual plataforma deseja fazer a instalação (Figura 9).

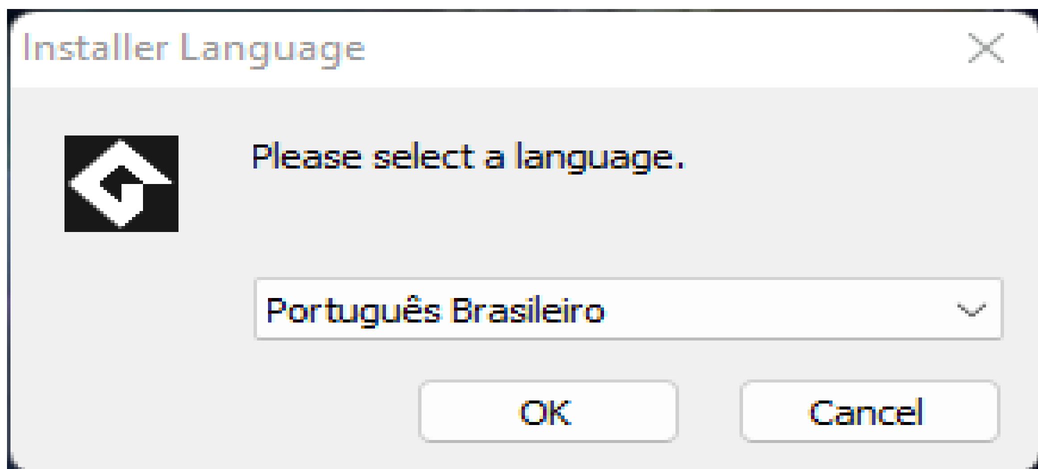
Figura 9 Site GameMaker



Fonte: Site GameMaker

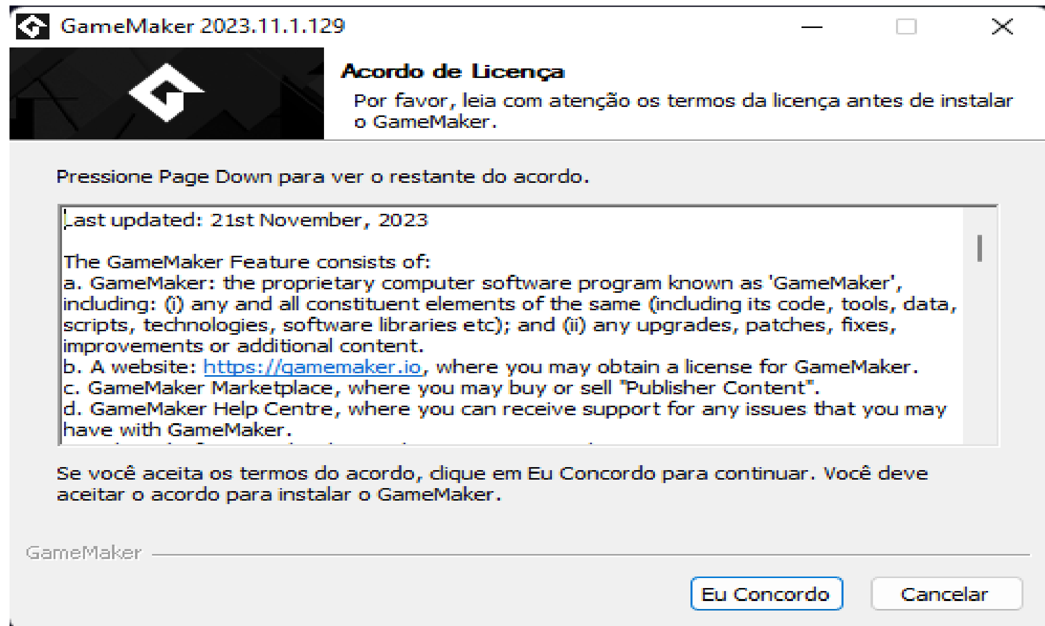
Para começar a instalação primeiro é necessário escolher a língua e ler o termo de licença, assim como esta as figuras 10 e 11.

Figura 10 Instalador GameMaker - Idiomas



Fonte: Instalador GameMaker

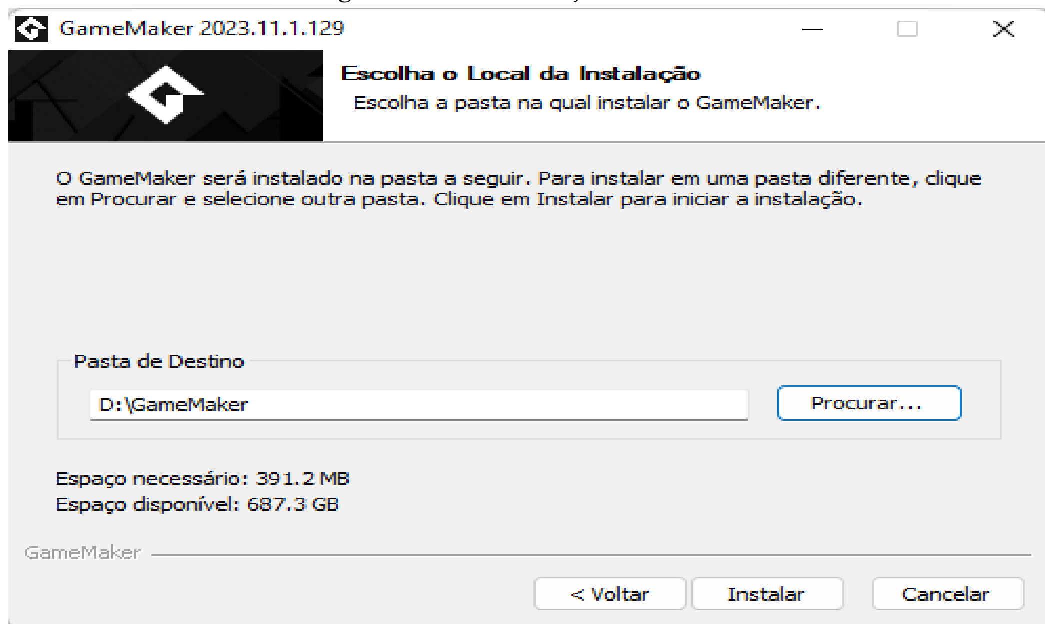
Figura 11 Instalador GameMaker – Termos



Fonte: Instalador Gamemaker

Depois as opções, representadas pela figura 12, devem ser escolhidas de acordo com a preferência do usuário.

Figura 12 Local Instalação GameMaker



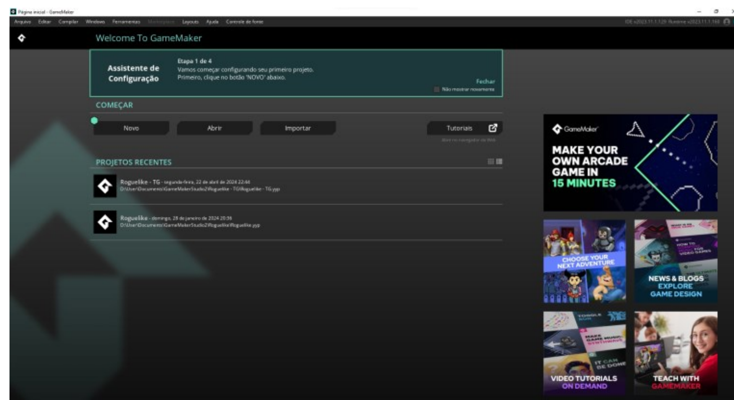
Fonte: Instalador Gamemaker

Assim é concluído todo o processo de instalação do GameMaker.

5.3.3 Interfaceamento

A primeira tela que temos ao abrir a *engine* possui 4 opções centrais, sendo elas: "Novo" onde é para criar os jogos do zero, "Abrir" para acessar jogos que já estão sendo utilizado na máquina do usuário, "Importar" que é capaz de pegar outros jogos que não estão salvos no *GameMaker* do usuário e o "Tutoriais" que te direciona para o site oficial da *GameMaker*, numa aba que possui diversos tutoriais de como utilizá-lo.

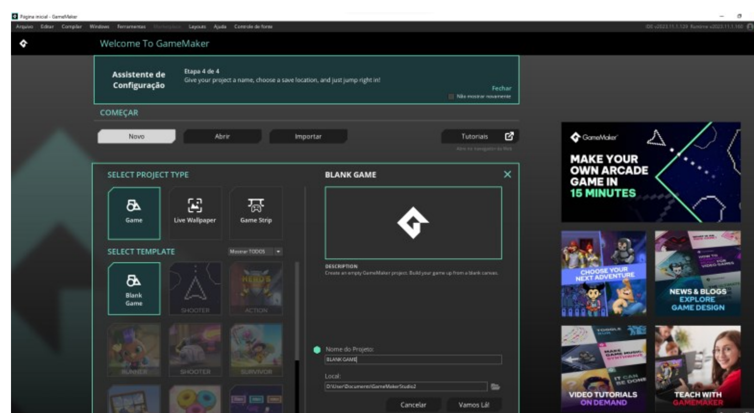
Figura 13 HUD inicial do GameMaker



Fonte: HUD GameMaker

Na aba "Novo" possui três opções diferentes: *Game* onde possui vários jogos feitos de diversos modos diferentes para quem não deseja começar o jogo do zero e a escolha para quem quer começar do início chamada *Blank Game*, as opções *Live Wallpaper* e *Game Strip* estão totalmente relacionadas ao navegador *OperaGx* uma servindo para fazer plano de fundo animado e a outra para criar um jogo para esse navegador.

Figura 14 HUD GameMaker-Novo



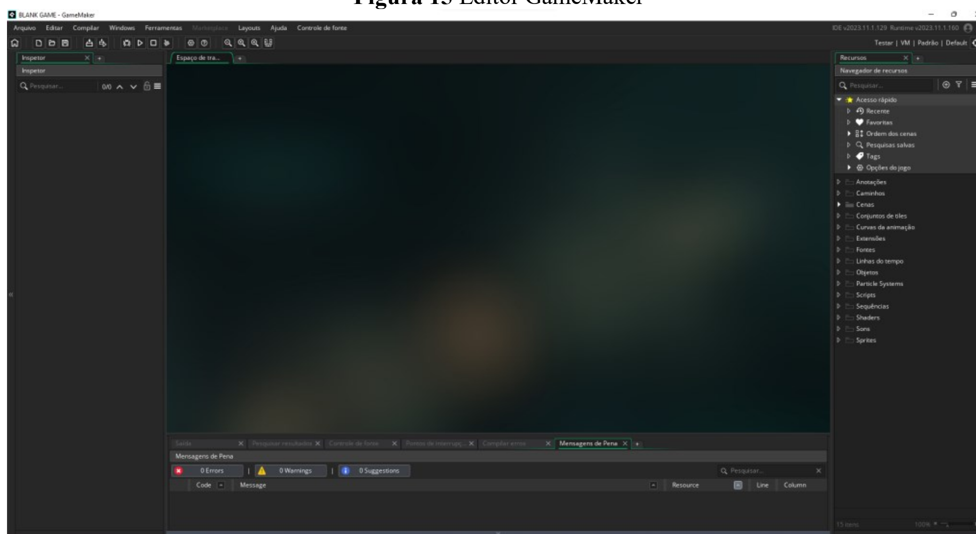
Fonte: HUD GameMaker

Ao entrar na aba do editor é possível se deparar com uma série de recursos, a coluna a direita possui diversas pastas onde será colocado todos os scripts criados para o

desenvolvimento do jogo, além de onde vamos guardas e definir as sprites, sons, cenas, menus entre outras funcionalidades.

Na coluna de esquerda é onde fica as camadas de cenas, a ordem de criação das instâncias e o controle junto com a ativação das câmaras. Na parte central é onde podem ser feitos os códigos do projeto, junto com a edição dos objetos e *sprites*.

Figura 15 Editor GameMaker



Fonte: Editor GameMaker

5.4 Desenvolvimento do Jogo

Foram escolhidas algumas características base as quais os jogos a serem criados teriam, como seu gênero sendo *RogueLike,* estilo gráfico PixelArt 2D, jogos semelhantes a jogos famosos como Enter the Gungeon e The Binding of Isaac, os quais forma as principais inspirações.

5.4.1 Unity

Para construção do jogo na Unity foi inicialmente seguido uma série de vídeos tutorial do canal JVCOB, onde ele cria um clone do jogo The Binding of Isaac no Unity. É uma série de vídeos feitas na versão do Unity Editor de 2019,

5.4.1.1 *Player*

Começando o desenvolvimento cria-se um *sprite* simples através das opções que o editor disponibiliza e um *GameObject* nomeado de Player e adicionamos a ele tanto o *sprite* criado quanto um *RigidBody2D,* um componente disponível no editor da Unity que atribui a sprites propriedades físicas e permite o controle do mesmo a partir de script baseado em forças.

Nesse componente deixamos a escala da gravidade dele para 0, pois o jogo funciona com uma visão de cima, então se a gravidade fosse mantida, o sprite ficaria sempre indo para baixo, o que não pode ocorrer. Também se adiciona um *Box Collider 2D* a ele para que haja colisão com outros sprites.

Com a base visual e alguns componentes simples adicionado ao jogador (Player) cria-se o primeiro script do projeto, o que é chamado de *Player Controller*, que é responsável por adicionar funcionalidades e/ou propriedades ao jogador. Nesse script primeiro define-se 3 variáveis, a *speed* armazena a velocidade que o personagem do jogador se move, e as variáveis horizontais e verticais, que pegam o input do jogador, também tem uma variável a qual pega o *RigidBody2D* do objeto. Com essas variáveis definidas e a *speed* tendo seu valor atribuído no editor, cria-se o código para verificar se houve *input* em algum dos eixos ou em ambos e caso ocorra move o jogador pela tela.

5.4.1.2 *Collection Controller*

No script do jogador, deixa-se assim por enquanto e cria-se outro elemento do jogo, os itens, para isso novamente cria-se um sprite, um *GameObject* para ser a base do item e adicionamos o sprite e um *collider* nele. Já no script do mesmo só é criado inicialmente só o código para que caso a colisão do jogador se encontre com o do item, o item se "destrua".

5.4.1.3 Sistema de tiro

Para que o personagem do jogador possa atirar projéteis, é necessário a configuração de novas teclas as configurações de input, define-se as teclas direcionais com o nome de *ShootVertical* e *ShootHorizontal*, esse é um dos recursos que a Unity disponibiliza, maneiras de associar funções às teclas ou botões de um controle e dar nome as mesmas para ser mais fácil de utilizar dessa captura nos scripts. Definidos quais as teclas irão ser responsáveis por fazer o jogador atirar, volta-se para o script do jogador e adiciona-se uma nova sessão de código que verifica o input das teclas de tiro e chama a função responsável por atirar os projéteis, para isso, deve-se criar o projétil a ser atirado.

O projétil a ser atirado pelo jogador é um novo *GameObject* que é transformado em um *PreFab* que é uma maneira que o Unity permite de armazenar *GameObjects* já configurados para a reutilização. O script do projétil é um código simples que armazena o tempo de vida da bala, ou seja, depois de quantos segundos em tela ela irá ficar antes de desaparecer e a função que faz a bala subir depois desse tempo.

Com o projétil criado e definido como um *PreFab,* pode-se atribuir ele a uma variável no script do jogador e através dessa variável na função de atirar do jogador esse objeto é instanciado, adiciona-se um *RigidBody2D* e adiciona a *velocity* do objeto um Vetor com a mesma direção que o jogador acionou, assim fazendo com que o projétil seja criado e mova-se para frente na cena do jogo.

5.4.1.4 *Enemy Controller*

Agora, cria-se o inimigo, repetindo boa parte do processo feito para criar os outros objetos: sprite, *GameObject* e o script, só que antes do código do script, adicionamos a *tag Enemy* ao objeto, *tags* são uma maneira que o Unity tem de poder classificar objetos, para fácil busca nos scripts, como por exemplo, foi utilizado no código dos itens, onde verifica-se se o colisor que teve contato com o item tinha a *tag player* para só depois excluir o item caso seja. No script do inimigo, utiliza-se de um modelo matemático de computação chamado *Finite State Machine (FSM)*, o qual é uma estrutura que mantém um programa, parte de programa funcionando baseado em um número finito de estados, que nesse caso o inimigo começa com os estados de *Wander* (andando) e *Follow* (seguindo). Para armazenar esses estados é criado tanto um Enum para configurar os estados possíveis e uma variável que mantém o estado atual que o inimigo vai estar. Para esse sistema funcionar utiliza-se de uma estrutura condicional do tipo switch para verificar qual estado o inimigo está e chamar a respectiva função que define o que o inimigo deve fazer naquele estado, e no método *Update* do inimigo é feita uma estrutura condicional baseada na distância do jogador do inimigo para definir em qual estado o inimigo deve estar.

Quando o inimigo está no estado de Andar, escolhe-se uma direção aleatória para o inimigo e através do *Quaternion* para normalizar a rotação do inimigo, ele fica se movendo naquela direção aleatória baseado na sua velocidade, que é definida entre as variáveis do script, até que o jogador entre na área de visão do inimigo, que é outra variável definida no script.

No estado de Seguir, como o nome diz, ele fica indo em direção ao jogador, mas só caso o jogador esteja na "visão" do inimigo.

No script do projétil, adiciona-se uma verificação para caso o colisor da bala atinja o colisor do inimigo, a função que destrói o inimigo e depois o próprio projétil se destrói.

5.4.1.5 *Game Controller*

Alguns status fixos do jogo são necessários de ter um controle maior e para isso é criado um script com o nome de *GameController*, o qual é um script que segue o padrão de projeto chamado *Singleton*, onde só existe uma instância do mesmo ativa o tempo todo, ou seja, toda vez que for necessário utilizar algo relacionado ao mesmo, utiliza-se da instancia já ativa dele ao invés de ser criado uma nova, isso permite que tenhamos status com valores "únicos" e de uso global. Para isso é necessário o entendimento melhor do funcionamento da classe *MonoBehaviour* que é a classe o qual todos os scripts criados até agora herdam sua implementação, ela é uma classe que vem com uma série de métodos prontos que ajudam no desenvolvimento, entre eles temos o *Awake,* que é o método que é chamado assim que o script é inicializado, e é deles que o padrão *Singleton* utilizará, pois dentro dele, verifica se a instância do script, e caso não, instância.

Temos os status do jogador: vida, vida máxima, velocidade de movimento do jogador, velocidade do tiro, *cooldown* de ataque do jogador, definidos como estáticas e propriedades da

classe, ou seja, através dos métodos *get* e *set* que elas são acessadas. Também tem as funções de modificar esses status nesse script, então para que o inimigo possa causar danos no jogador, precisa-se retornar ao script do inimigo onde adiciona-se um novo estado ao mesmo, o *Attack* (Ataque), nesse estado, caso o jogador esteja no inimigo esteja na área de ataque do inimigo, definida sendo menor que a de visão, ele causa dano ao jogador, chamando a função responsável por causar dano ao jogador da instância de *GameController.*

Ainda relacionado ao script principal do jogo, o *GameController*, cria-se o *HUD* do jogo, para isso utiliza-se de dois sprites de coração, um sendo a vida cheia e outra vazia, utilizando as propriedades de edição de sprites que o Unity disponibiliza, assim sobrepõe-se às imagens no *HUD* e define-se a de vida cheia como uma imagem do tipo preenchimento, então no novo script *HealthUIController* onde é feito um cálculo baseado na vida atual do jogador com a vida máxima para definir esse nível de preenchimento, permitindo que cada vez que haja mudança na vida o preenchimento do elemento visual da vida mude também.

Nesse momento, os itens são só elementos visuais, então para que ele tenha funções precisa-se trabalhar mais neles, primeiro cria-se uma classe *Item* que diferente de todas até agora, é criada dentro do script responsável pelos itens, e não herda de *MonoBehaviour*, e sim uma classe que será do tipo *System Serializable,* que permite que objetos que tenham essa classe possam ser armazenados com o valor que eles tiverem naquele momento. Com essa classe definida, pode-se criar de forma visual os itens, dando um sprite a eles e o que eles vão afetar no jogador.

Para terminar essa parte relacionada ao *GameController* “cria-se” um novo tipo de inimigo, o *ranged*, para tal é necessário retornar para o *EnemyController* e definimos um novo Enum que define os tipos de inimigos que haverá no jogo e cria-se uma variável para armazenar um valor desse Enum. Então no método de atacar adiciona-se um switch que verifica o tipo do inimigo e realiza o ataque baseado nisso, onde agora tem-se o novo método de ataque implementado, que se assemelha muito com o que acontece no ataque do jogador, instancia-se a bala e a comando a ir na direção do player.

5.4.1.6 Geração de Masmorras

Toda a ideia de um jogo do estilo *roguelike* baseia-se na ideia de um mundo gerado aleatoriamente e toda vez que o jogador perder, ter que recomeçar, então para que isso possa acontecer é necessário um sistema de geração de salas, ou geração de masmorras. O sistema que será criado para esse jogo é baseado na ideia de escolher uma dentre uma série de mapas/cenas já configurados anteriormente e montar toda a masmorra a partir delas.

Primeiro começa pelo *RoomController*, onde inicialmente é importado a biblioteca de gerenciamento de cenas do Unity, cria-se uma classe chamada de *RoomInfo*, a qual armazena algumas informações importantes que toda sala terá que ter, sendo o nome, x e y. Na classe principal segue-se novamente o padrão *Singleton*, e a primeira coisa que a classe faz, é verificar se a sala já existe, para isso cria-se um método para fazer isso, esse método, através das coordenadas x e y recebidas, verifica se a sala existe em uma lista de *Rooms* e retorna se existe

ou não, só que é necessário então definir as coordenadas x e y na classe *Room* e já pode aproveitar e definir as variáveis que detém os tamanhos da sala, largura e comprimento.

Ainda estando no script da *Room*, dentro do método *start*, verifica-se se existe instância de *RoomController*, pois caso não exista, pode estar ocorrendo um erro da sala errada estar sendo iniciada. Um novo método precisa ser criado também, o *GetRoomCentre*, que assim como o nome diz, ele retorna o centro da sala. Para finalizar momentaneamente as edições no *Room*, cria-se um método para desenhar um Gizmo ao redor da sala para auxiliar visualmente.

No editor, precisa-se agora criar as salas, que no caso são cenas do Unity, nomeamos elas como *BasementMain* e a *BasementStart*, onde na segunda, remove-se todos os objetos que não devem ter na sala inicial do jogo. Nessa sala cria-se um objeto que representa a sala em si, e outro com o nome de *sidewall* que tem as paredes da sala e ele é colocado dentro do objeto principal da cena, o *Room*, que também recebe como componente o script de mesmo nome. Criamos as portas da sala, cada uma em um objeto separado e para finalizar duplica-se essa sala e renomeia-se o nome da duplicata para *BasementEnd.*

De volta para o script *RoomController* onde é criado um método *LoadRoom*, que é responsável por criar uma instância de *RoomInfo* e colocamos o mesmo na fila *LoadRoomQueue*. Outro método a ser criado é o *LoadRoomRoutine* que é uma rotina de carregamento das salas através do método de carregamento de cenas assíncronas disponibilizado pela Unity. Cria-se o método *RegisterRoom*, que é responsável por verificar se a sala a ser registrada já existe, caso ela já exista na lista de salas registradas a *Room* será destruída, ou seja, o objeto *Room* que foi passado para *RegisterRoom* não existirá mais, isso se torna importante em um momento futuro.

No método *LoadRoom* verifica-se se a sala já existe na fila, caso exista a sala nada precisa ser feito, caso não, ela seja adicionada a fila de salas, novamente esse método junto com todo o conjunto se torna de melhor compreensão. É necessário adicionar as salas criadas as configurações de *build* do jogo, assim permitindo que as salas possam ser acessadas pelo código. De volta ao script *RoomController* é adicionado ao método *Start* dele 5 chamadas ao *LoadRoom* o que ocasiona de assim que o jogo ser iniciado 5 salas são criadas. Já no *Update*, outro método especial do Unity que é convocado a cada novo frame do jogo, é colocado uma chamada para o método *UpdateRoomQueue,* que é um método ainda a ser criado. O método em questão é responsável por verificar se já tem alguma sala sendo carregada, ou se a fila de salas está vazia, caso nenhuma das condições sejam verdadeiras, é começado o processo de esvaziar a fila iniciando as mesmas. Na cena *BasementMain*, adiciona-se um novo objeto chamado *RoomController,* onde é adicionado o script o qual tem o mesmo nome. E finalizando essa parte no script *Room*, em seu método *Start* chama-se o método para registrar sala.

5.4.1.7 Câmera

Da maneira que o jogo está atualmente, são criadas 5 salas, e o jogador pode ser mover, só que a câmera do jogo não acompanha ele, o que ocasiona em caso o jogador avance nas salas

não será possível ver onde o personagem está. Para que isso não ocorra é necessário criar um script para controlar isso.

Criando então o *CameraController*, mais um script que segue o padrão *Singleton*, então é necessário que ele se instancie no método *Awake* e toda vez que precisamos alterar algo relacionado ao mesmo, utiliza-se através da instância. Duas variáveis são criadas inicialmente, uma armazena a sala atual onde a câmera está e outra a velocidade em que a câmera se move.

No método *Update*, é necessário chamar *UpdatePosition*, que verifica se não existe sala atual na câmera, que caso não esteja, uma variável *TargetPos* é criada e atribui-se um valor que virá do método *GetCameraTargetPosition.* Esse novo método, verifica novamente se não tem sala atribuída a variável *currRoom* e se não tiver, pega a posição central da sala atual e define a posição da câmera sendo esse valor.

Em *UpdatePosition*, é necessário colocar a função para que a câmera se mova da posição atual para a posição alvo na velocidade desejada. Existe também o método *IsSwitchingScene* que verifica se está acontecendo transição de sala, ou seja, ele verifica se a posição atual da sala é diferente da posição alvo, no caso de ser, a câmera estará em processo de trocar de cena.

De volta ao *RoomController*, cria-se o método *OnPlayerEnterRoom*, o qual recebe uma variável da classe *Room* e nele passamos essa sala para o *currRoom* da câmera, a sala também é atribuída a uma variável de mesmo nome só que pertencente a *RoomController*.

Para que seja possível ser realizada essa troca da câmera para outra sala, é necessário saber quando isso deve ocorrer, então é colocado um colisor nas salas um pouco menor do que será o tamanho da sala e ela é colocada no modo *Trigger* que é uma maneira do colisor não afetar fisicamente os outros colisores, ou seja, ele é capaz de detectar se houve colisão com outro colisor, mas não afetará eles fisicamente.

No script da sala, *Room,* adiciona-se então o método especial *OnTriggerEnter2D,* outro método especial da Unity que é acionado ao um colisor do tipo *Trigger* ter contato com outro colisor, foi feito igual no script dos itens, através desse método que é chamado o método *OnPlayerEnterRoom* que irá atualizar a posição da câmera.

5.4.1.8 Geração aleatória de salas

O código de geração aleatória de salas começa com o *DungeonCrawlerController*, esse script, em uma tradução livre, seria o controlador do rastreador de masmorra, ou seja, ele controla uma parte muito importante do sistema de geração aleatória, os rastreadores. Como já mencionado anteriormente, futuramente existirá uma seção explicando todo o sistema de geração de masmorra aleatória.

No *DungeonCrawlerController* cria-se um enum chamado *Direction*, que define quatro valores, cada um correspondente a uma direção (cima, baixo, esquerda, direita). Dentro da classe principal cria-se um dicionário que liga uma direção a seu vetor correspondente.

Um novo script chamado *DungeonGenerationData* é criado e em sua classe principal mudamos o tipo de herança que ele tem, desde herdar de *MonoBehaviour* ela herda de *ScriptableObject* que é uma maneira de armazenar dados para que não haja repetição de dados.

Essa classe tem três variáveis, o número de rastreadores, o número de iteração mínima e o número de iteração máxima. No geral esse script armazena as principais informações para decidir os padrões de aleatoriedade do sistema gerador de masmorras, onde o número de rastreadores, é o número de elementos escolhedores de direção, o número mínimo de vezes que cada rastreador irá escolher e o número máximo que ele poderá escolher.

Retornando ao *DungeonCrawlerGeneration* para terminar a implementação do método *DungeonGeneration*. Nesse método é feita uma definição aleatória da quantidade de interações que cada *dungeonCrawler* irá realizar, assim cada um realiza suas devidas interações e adiciona as coordenadas de onde deve ser gerada cada uma das salas.

Por fim, no script *DungeonGenerator*, é criado duas variáveis, uma que é do tipo *DungeonGenerationData* e outra do tipo List<Vector2Int>. No método *Start* tem-se a atribuição da lista de coordenadas de salas a serem criadas a partir do sistema de *DungeonCrawler* tendo como parâmetro o *DungeonGenerationData* recém-criado no script à lista previamente criada. E para finalizar o método *Start* é chamado o método *SpawnRooms* utilizando a lista de coordenadas como parâmetro.

O método *SpawnRooms* gera uma sala vazia para cada coordenada da lista que foi recebida como parâmetro.

Para finalizar o sistema de geração de salas, em *RegisterRoom*, é colocado uma verificação se a sala a qual está tentando ser gerada já existe, e caso exista não é gerada.

5.4.1.9 Sistema de geração de masmorras

O sistema de geração de masmorra implementado funciona da seguinte maneira: um ou mais rastreadores são gerados, esses rastreadores escolhem entre 4 direções para gerar uma sala partindo da sala inicial e adicionando as posições que ele visitou a uma lista para que seja futuramente gerada uma sala naquelas coordenadas e esse processo de visitar coordenadas é repetido por cada um dos rastreadores um número de vezes aleatório que varia entre um número mínimo. No geral é um sistema bem simples, pois tudo que ele faz é decidir as coordenadas das salas, depois é passado essas coordenadas para o sistema que gera a sala, ou seja, essas salas são adicionadas a uma lista de salas e depois cada uma é gerada no jogo pelo sistema da Unity de carregar cenas assincronamente.

5.4.1.10 Portas das salas

É necessário criar um sistema para que as portas sejam geradas corretamente, ou seja, para que só exista porta na sala caso tenha salas ao lado dela, e para isso um novo script é criado, o *Door*. Nesse script é criado um enum que terá representado cada uma das direções possíveis e uma variável que armazena qual o tipo da porta e esse script é adicionado aos objetos no editor do jogo.

Em *Room* cria-se 4 variáveis, uma para cada porta e uma lista de portas, a qual pega todas as portas que estão dentro da “sala” para que possa então ser feito a interação por elas e

definir o objeto correto para cada porta direcional. Em outro método é feito a verificação se existe sala em cada uma das direções e caso não exista a porta correspondente aquela direção é desativada e esse método é adicionado ao *RegisterRoom* para que as salas já sejam salvas com as portas desnecessárias desativadas.

5.4.1.11 *Boss Room*

Para a criação da sala do chefe, é necessário revisitar o script *DungeonGenerator,* onde no método *SpawnRooms,* é realizado uma verificação se para verificar se a sala que está sendo gerada é a última da lista, pois se for, ela irá se tornar a sala do boss.

A geração da sala de boss ocorre somente quando o gerador está criando sua última sala, e nesse caso, é pega as coordenadas da última sala a ser gerada e transforma a mesma na sala do chefe.

5.4.1.12 *Item/Enemy Generator*

O sistema de geração de itens é criado através de um sistema que gera um *Grid*, onde grid é uma estrutura de mapeamento das salas que divide as salas em diversos quadrados, e através desse grid, é gerado uma série de coordenadas. Além do *Grid* é necessário o próprio *item spawner* que é um script onde tem uma lista de itens geráveis onde cada item tem um peso que representa a chance de ele ser gerado, e através de um sistema que soma esses pesos para selecionar qual item irá ser gerado. Então o sistema no geral funciona da seguinte maneira: o *Grid* gera uma lista de coordenadas nas salas onde pode ser gerado os itens, o *Item Spawner* decide qual item irá ser gerado, e o sistema escolhe uma posição através da lista gerada pelo grid, sorteia quantos itens serão gerados e instancia o objeto naquela posição.

Semelhante ao sistema de geração de item para gerar os inimigos na sala, ocorre o mesmo processo, geração de lista de coordenadas pelo *Grid,* o sistema sorteia uma posição da lista e instancia os inimigos.

Para a criação desse sistema é criado um objeto não visual que representa o *Grid* para ser o mapa da sala, ou seja, ele é um objeto pertencente a sala do tamanho dela além de outro objeto para ser o *Item Spawner* e o *Enemy Spawner*, com ambos pertencendo a sala, dentro do script dela é possível utilizá-los no sistema de geração.

5.4.1.13 Atualizações no jogo

Primeiro de tudo, o sistema de geração de salas que é ensinado pela série de vídeos é interessante, mas tem um problema, ele gera e instancia as salas, o que pode acarretar em caso de o número de salas ser alto, uma pequena lentidão na hora de gerar as salas, então é feita uma mudança para que as salas sejam geradas e só for instanciadas quando o player precisar acessar ela, e para isso, é necessário adicionar caixa de colisão nas portas e adicionar aos scripts uma

verificação para quando houver contato do player com a porta, a sala é instanciada e o player é movido um pouco para dentro da mesma.

Aos inimigos é adicionado *Rigid Body* e "peso" para que eles possam colidir com as paredes da sala, e não sejam empurrados pelo jogador. Também se adiciona colisores as paredes e uma a *tag*, que é uma maneira de identificar os objetos no script e com essa *tag* é possível fazer com que as balas colidam com ela e desapareçam na colisão.

Também se cria um sistema para que as portas desapareçam das salas caso tenha inimigos e elas só apareçam quando não tiver inimigos, esse sistema funciona através do objeto da sala que seu controlador verifica se há inimigos na sala, caso tenha ele desativa as portas, e quando não tiver nenhum inimigo ele reativa as portas.

5.4.1.14 Outros Sistemas

Cria-se também para o jogo um minimapa, que é um objeto do tipo câmera, que só visualiza elementos que estão em outro *Layer*, então, para todos os objetos que for aparecer no minimapa é necessário adicionar a eles um novo elemento visual que fica nesse *Layer* e adiciona-se essa câmera a sala principal para que ele sempre esteja visual para o jogador.

Além dos sprites padrões que o Unity oferece é possível adicionar sprites externos, e a Unity oferece um sistema para criação de animações e controle dela através de estados, onde esses estados têm variáveis que ao receber algum valor específico chama o próximo estado, ou seja, troca para a animação necessária. Criada a animação e esse sistema de controle de estado, adiciona-se o componente *Animator* ao jogador e assim o player tem sprites animados.

Semelhante ao sistema de animação, a Unity oferece diversos componentes relacionados ao áudio e através da inserção desses componentes aos objetos, e os arquivos de áudio a eles, então é possível controlar esses arquivos de áudio através dos scripts.

5.4.2 GameMaker

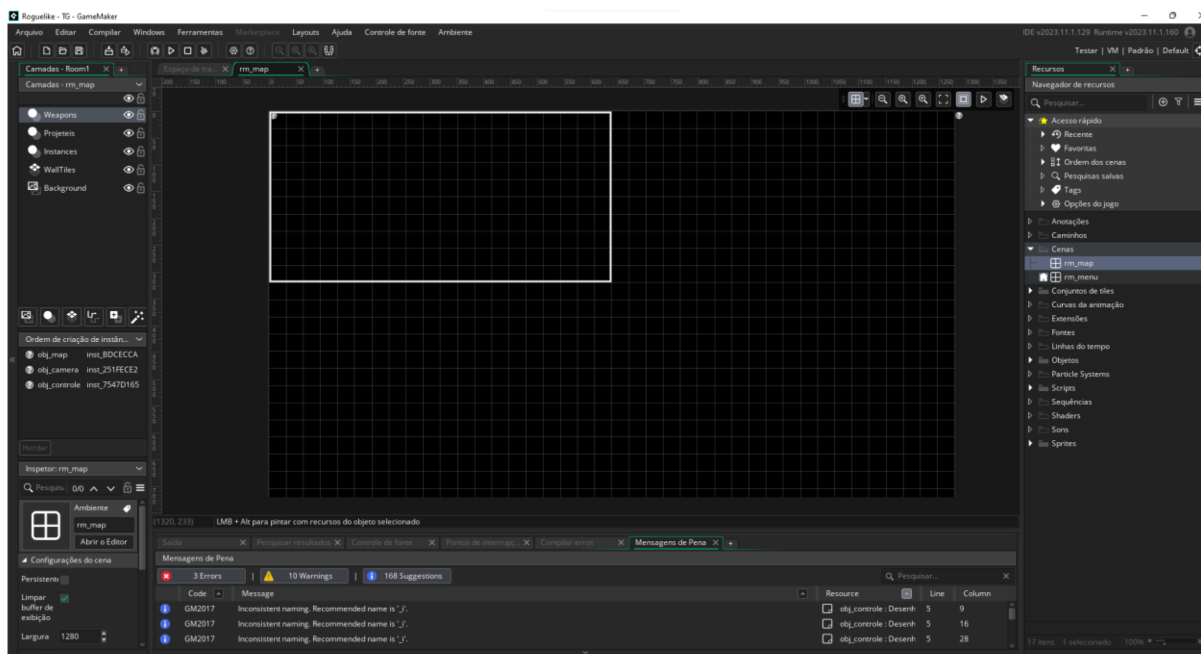
O jogo teve como base os tutoriais dos vídeos do canal GUI e TARI, onde neles é ensinado e explicado o funcionamento de como fazer os vários códigos necessários para produzir um roguelike no estilo shooter.

5.4.2.1 Mapa Procedural

Para começar o jogo primeiro foi criado o mapa em que o jogo vai se passar, como é o roguelike o mapa deve ser diferente para cada vez que iniciar o jogo. Com isso em mente foi utilizado a geração procedural para construir a cena, no qual consiste em uma geração algorítmica de dados ao invés de manual.

Primeira coisa a ser feita foi criar um ambiente nomeado de *rm_map*, ela será a sala onde nosso *player* quando criado vai se mover, como pode se observar o ambiente criado possui vários quadrados, que são chamados de *Grids.*

Figura 16 GameMaker - Ambiente do Jogo



Fonte: Editor GameMaker

Após a criação do ambiente, serão criados os objetos com os nomes de *obj_map*, no qual será utilizado para fazer a geração procedural, e *obj_wall* que será a parede que impedirá a passagem quando no *player* for criado.

Com isso criado, foi feito um script onde é escolhido a célula no meio da *rm_map* e a partir dela é determinada mais outra célula, só podendo ser essa escolhida se ela estiver encostada com a célula previamente sorteada. Dessa forma, é estabelecida uma quantidade de células que vão ser sorteadas, esse número será escolhido conforme o tamanho do mapa desejado, e todas as outras que não foram escolhidas serão preenchidas com o obj_wall.

Para fazer testes foi feito um código onde sempre que apertado a letra R o mapa era redesenhado, assim podendo recomeçar o jogo. Sendo apenas preciso agora ser colocado o *obj_map* no *rm_map* para que ele funcione.

5.4.2.2 *Player*

Para adicionar o jogador onde iremos comandar foi preciso criar um *obj_player* com uma sprite mais simplificada que será alterada no futuro. A primeira coisa a determinar foi o tamanho, a velocidade e as teclas desejadas para se mover, após tudo definido foi criado um script para sua movimentação, onde ele pode se mover para a horizontal, vertical e na diagonal, de maneira que ao apertar duas teclas juntas elas não se somarem suas velocidades.

Como o mapa já era gerado ao iniciar o jogo, para que fosse possível o nosso *obj_player* ser feito junto com a criação da sala, foi estabelecido que ele só nascesse no mapa apenas onde

não possuísse *obj_wall* e de preferência sempre nas laterais da sala. Aproveitando já foi feita uma colisão onde o jogador é impedido de entrar dentro das paredes predefinidas.

5.4.2.3 Inimigos

Da mesma forma que o *player*, foi feito um *obj_inimigo* no qual ele aparece no mapa assim que o jogo é iniciado, mas tendo que estar a uma distância do nosso jogador, para impedir que já comece com você levando dano. O script feito faz com que eles fiquem no estado parados até o obj_player entrar em seu campo de visão (se o jogador estiver atrás de uma parede e perto dele, ainda estará contabilizado no estado de parado), assim eles mudam o estado para o *seguindo_player*, que o faz perseguir o *player* até que o próprio não exista mais.

Com isso foi adicionado vida tanto para o jogador e para os inimigos, assim sempre que o *obj_inimigo* encostasse no *obj_player* ele perdia um de vida, quando essa vida chegasse a zero o jogo era recomeçado. Foi escolhida uma quantidade determinada de inimigos que nascem, junto com seus atributos de vida, velocidade e dano.

5.4.2.4 *Sprite*

Para mudar a aparência de nossos objetos, de maneira no qual fique mais fácil de identificar cada objeto durante o jogo, foi pego *assets* 2D totalmente gratuitos do site ITCH.IO, é preciso apenas escolher e baixar as *sprites* que deseja. O site fornece vários *assets* pagos e gratuitos, mas para o projeto foi utilizado apenas aqueles que não eram necessários pagar.

Para o *obj_wall* foi pego uma sprite que possuía o desenho de parede de vários ângulos diferentes e de um chão, agora voltando ao script do *obj_map* foi adicionado um código para que dependendo do local que a parede está localizada ela pega uma sprite determinada para que pudesse dar uma perspectiva altura ao jogo, assim como mostra a figura abaixo, onde a parte em marrom são paredes e em cinza é o chão.

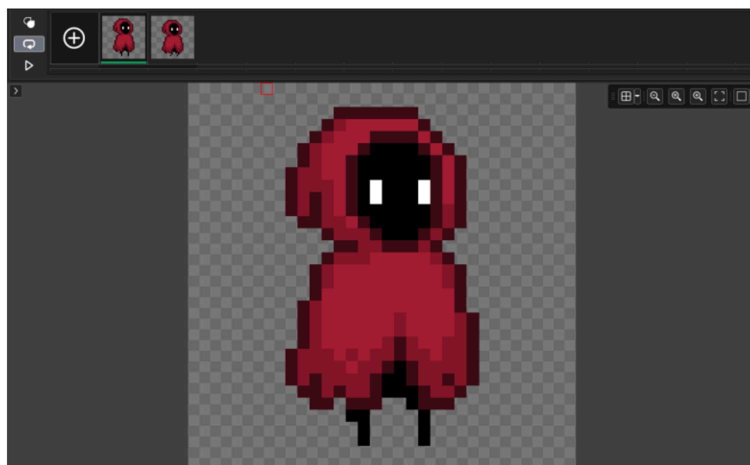
Figura 17 GameMaker - Sprite das Paredes e do Chão

Fonte: Editor GameMaker

Agora para o jogador e para o inimigo foi feita da mesma forma, foram escolhidos dois pacotes de *sprites* onde foi preciso selecionar a sprite desejada para cada um, sendo uma *spr_idle* para quando o objeto estiver parado e a outra *spr_run* quando começar a andar.

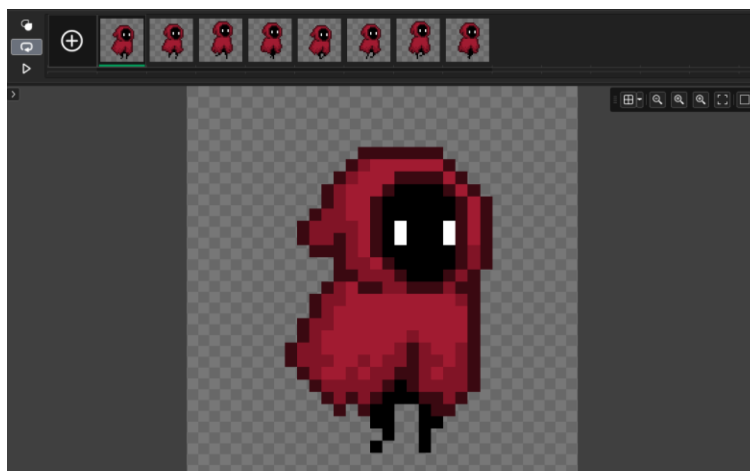
Para que não fique torto é necessário configurar que a *sprite* nos objetos para que ela fique sempre na parte do meio central e separar cada parte dos assets que compõem sua animação, para que fique mais fluido. Também foi um código para que o jogador e o inimigo fiquem com a sprite na direção correta na qual eles estão se movendo.

Figura 18 GameMaker - Sprite Jogador Parado



Fonte: Editor GameMaker

Figura 19 GameMaker - Sprite Jogador Correndo



Fonte: Editor GameMaker

Foi adicionado também uma sombra embaixo desses objetos é um *hit_alpha* que serve para sempre que algum deles tiver sua vida diminuída suas sprites vão ficar totalmente brancas por um breve momento para indicar que levaram dano.

Para que dependendo do *asset* que foi escolhido não deixe o objeto mais fácil ou mais difícil de se atingir, foi preciso criar um *spr_mask* para ficar de maneira padronizada onde será a caixa de acerto para eles, foi preciso apenas selecionar ela como a máscara de colisão.

Figura 20 GameMaker - Sprites dentro do jogo

Fonte: Editor GameMake*r*

5.4.2.5 Armas

Para que o jogador possa causar dano e consiga eliminar os inimigos, foi escolhido a criação de armas para que ele utilize, onde cada uma tenha sua própria característica como: quantidade de munição, velocidade de disparo, dano, se ele vai ser automático e o delay entre os disparos.

Foi criado um *obj_weapon* e para tornar mais fácil a organização foi criado uma lista com diferentes tipos de armas com as características citadas anteriormente, assim foi adicionado *sprite* para cada uma delas para ser mais fácil de diferenciá-las em jogo. A fim que fosse possível a alteração entre elas, foi criado um *obj_weapon_drop* juntamente com um *script* chamado *scr_mudar_arma* tornado o jogador capaz de largar sua arma no chão e pegá-la sempre que ele apertar uma tecla que foi definida no código.

Com as armas já prontas foi feito em seguida os projéteis que vão sair delas e atingir os inimigos, assim foi criado o obj_proj que contém a *sprite* desse projétil e um script para ele ser destruído sempre que encostar em um inimigo ou parede. Voltando no *obj_weapon* foi adicionado uma quantidade de munição para cada arma criada, sendo assim sempre que o contador chegasse a zero não fosse mais possível atirar com ela.

As armas já foram concluídas, mas faltava o jogador ser capaz de atirar com elas, então ao voltar no script do *obj_player* foi adicionado um código que o permitia atirar, nesse mesmo código possuindo a diferença de quando o jogador está usando a função manual da arma apenas apertando o botão ou a função automática onde ele precisa pressionar o botão.

Figura 21 GameMaker - Sprites arma

Fonte: Editor GameMaker

5.4.2.6 Câmera

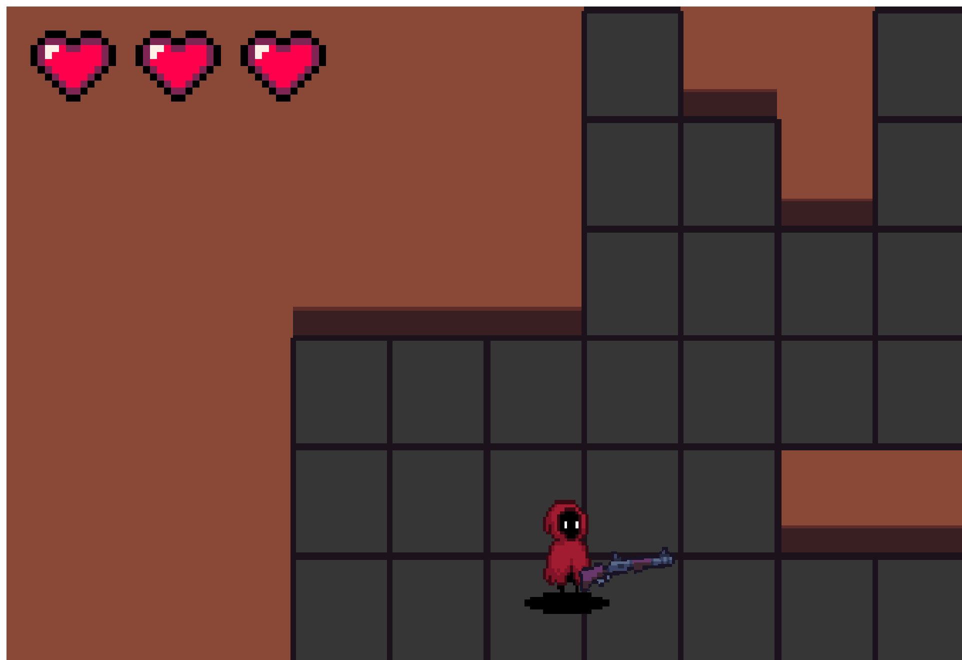
Até o momento, quando o jogo abre, é possível visualizar todo o mapa que foi gerado junto com seus inimigos. Para aumentar a imersão e para que ele fique mais parecido com os jogos do seu gênero, foi criado uma câmera com o foco em seu *player.*

Foi criado um *obj_camera* onde foi dimensionado o tamanho de células que ele vai mostrar, junto foi feito um *script* no qual ele centraliza nosso *obj_player* e o segue conforme ele se movimenta pelo cenário criado.

5.4.2.7 *HUD*

Já terminado os sistemas fundamentais de um jogo *roguelike,* foi feito um *obj_controle*, nele contendo um script para que fosse desenhado na interface do jogador a quantidade de vidas que ele possui, ou seja, sempre que o inimigo causasse algum dano no nosso *player* uma de suas vidas sairia de sua interface. Para melhor visualização foi pego uma *sprite* com um desenho de coração para simbolizar a vida de nosso jogador.

Figura 22 GameMaker - Sprites vida do jogador

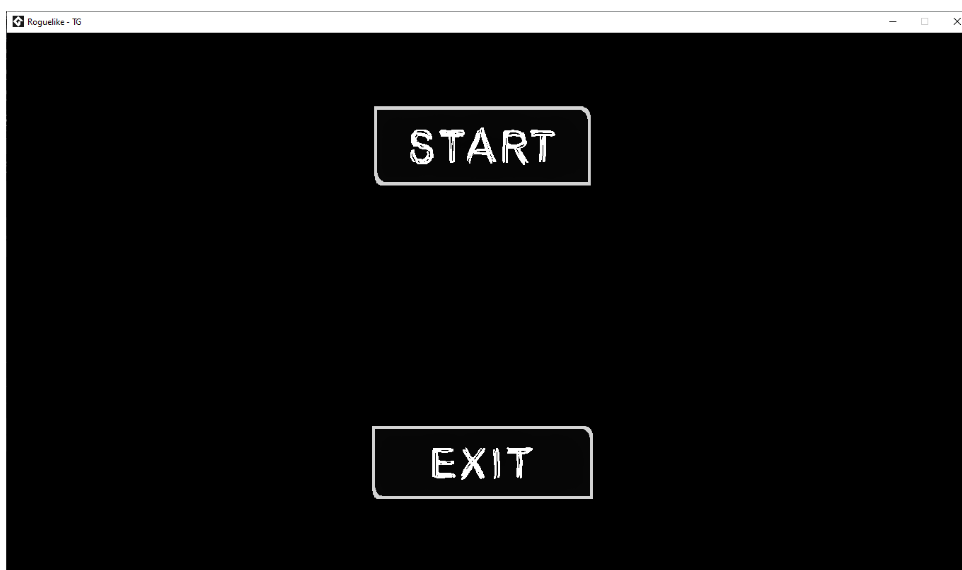


Fonte: Editor GameMaker

5.4.2.8 Menu

De modo que o jogador não entrasse direto no cenário do jogo, foi criado uma nova *room* com o nome de *rm_menu*, nela com dois diferentes botões: *start* e *exit*. O botão do start tem a função de direcionar a pessoa para a *rm_map* onde fica o cenário do jogo criado anteriormente. Já o botão *exit* tem o simples objetivo de encerrar a aplicação.

Figura 23 GameMaker - Menu do Jogo



Fonte: Editor GameMaker

5.5 Comparação final

O quadro comparativo (Tabela 1) busca elucidar como cada ferramenta "desempenhou" em cada critério decidido anteriormente através de um sistema de pontuação de 0 a 5 que representa a opinião gerada pela utilização e análise de cada ferramenta.

Em relação a Unity, ela tem uma curva de aprendizado relativamente difícil, pois apesar de ter muitas funcionalidades e recursos, essa quantidade de recursos acarreta em um impacto inicial forte em um usuário iniciante na área, além de que muitos desses recursos inicialmente não são intuitivos, acarretando em um processo mais lento de se aprender a utilizar a ferramenta, mas após isso e com o entendimento de como boa parte dos sistemas são controlados através dos scripts, a utilização se torna um pouco mais simples.

Já na documentação e suporte, a Unity oferece uma documentação completa de todos seus recursos só que muitos deles utiliza de muitos termos técnicos da área de desenvolvimento de jogos ou de área de matemática computacional, que ocasiona uma leve dificuldade em entender ela.

Para o aprendizado da Unity, é importante mencionar que ela possui uma comunidade gigantesca, que ajuda todos os níveis de pessoas, além da própria empresa disponibilizar diversos tutorias de fácil compreensão, ensinando diversas ferramentas e recursos disponibilizados tanto de forma gratuita quanto paga, além do fato de que mesmo com diversas atualizações, o sistema permanece semelhante, permitindo a utilização de códigos antigos de tal forma que mesmo os conteúdos mais antigos da comunidade em relação a códigos ainda é útil.

Os recursos de sprites e áudio da Unity são extremamente eficientes, pois conseguem importar e configurar os *sprite sheets* e os arquivos de áudio de forma automática precisando de pouca configuração manual, além de que a utilização deles se torna extremamente intuitiva através de poucos minutos de estudos deles, devido ao fato dos diversos componentes que podem ser adicionados aos componentes do jogo.

Tabela 1 Quadro Comparativo

Quadro Comparativo		
Critério/GameEngine	Unity	GameMaker
Curva de Aprendizado	3	4
Documentação e Suporte	4	4
Recursos de Aprendizado	5	5
Comunidade	5	4
Instalação	2	5
Interfaceamento intuitivo	4	5
Otimização	2	4
Integração com sistema de áudio	5	5
Animação de Sprites	5	5
Recursos de Depuração e Edição de Código	3	4

Recursos para agilizar o processo de criação	5	3
Preços de Licenças	2	2
Recursos da Licença	5	3
Reutilização de códigos antigos	5	4

Fonte: próprio autor

A Unity no geral tem uma interface muito intuitiva, e de fácil compreensão, mas seu sistema de instalação, otimização e depuração e edição de código são meio ruins. O processo de instalação é meio confuso, pois primeiro é necessário instalar o Unity Hub, e só através dele é possível instalar o Editor, apesar desse sistema permitir que os usuários instalem diversas versões diferentes do Editor, isso pode ocasionar em uma dúvida de qual usar, ainda mais para o usuário iniciante.

Em relação a otimização, é importante mencionar que cada mínima alteração em qualquer script o editor precisa meio que reimportar tudo, isso unido com o fato de a Unity não possuir editor de código embutido, ocasiona em que cada edição de código, perde-se um tempo esperando o editor fazer esse processo de importação e para finalizar, o sistema de depuração dela consiste em uma conexão entre o editor que for escolhido pelo usuário, normalmente entre Visual Studio e Visual Studio Code e o Editor, o que acarreta as vezes, de que ocorra travamentos e importação infinita do editor, sendo necessário encerrar a conexão para que o editor fique disponível novamente, mas no geral, a depuração de código funciona como em qualquer código feito em editores ou em ide, podendo marcar breakpoints para que o código do jogo rode pausando nesses pontos para analisar o que está ocorrendo.

Outro ponto a ser mencionado sobre o sistema de código da Unity é que ele utiliza uma linguagem de programação que já é muito utilizada no mercado de desenvolvimento no geral, que o C#, assim fazendo com que qualquer desenvolvedor que já tenha conhecimento dela e de linguagens semelhantes a ela, terá uma fácil adaptação a utilizar ela.

As licenças da Unity são bem definidas, e mesmo a mais básica, disponibiliza diversos recursos úteis, só que o preço das outras licenças é relativamente alto para "pouco" recurso extra, o que acarreta somente a utilização das licenças pagas, quando a empresa não se encaixar nos critérios da licença pessoal.

A GameMaker de primeira vista possui diversas opções logo na tela inicial de criação do jogo, mas conforme sendo utilizada a pessoa vai pegando o jeito e entendendo a utilização de cada função, sendo necessário para melhor entendimento a leitura na documentação.

A documentação feita pela GameMaker se mostrou bastante útil durante o processo de criação do jogo, contendo uma quantidade de guias para a criação de diversos recursos. Além de ter um *quick start guide* onde nele é explicado de maneira mais resumida para aqueles que já tem um certo grau de conhecimento.

Como mencionado em cima, na sua documentação possui ótimos recursos para o melhor aprendizado sobre a engine, além de possuir uma comunidade que ajuda uns aos outros e vídeos para melhor compreensão. Mesmo tendo uma comunidade que se ajuda, ela não é tão grande se comparada à de outras engines, como a Unity, então é possível ter uma dificuldade de achar a solução para um problema mais específico, além de que códigos muito antigos raramente ficam obsoletos.

Como mostrado no tópico acima, a instalação numa visão geral é bem simples, sendo necessário só entrar no site, selecionar onde o arquivo será baixado e clicar em continuar.

Na parte de adicionar *sprites* e áudio são bem simples de se utilizar, bastando apenas importar e configurar os *assets* escolhidos e configurar da maneira que desejar, podendo fazer alterações na própria *engine*.

As licenças da GameMaker não possuem um bom custo-benefício, onde elas acabam possuindo um valor mais elevado e não oferecendo tantas melhorias se comparado a licença gratuita, que é uma ótima escolha para aqueles que desejam começar na área e jogos, pois já oferece todos os recursos para desenvolver um.

5.6 Resultados e Discussões

Em relação a Unity, foi possível notar que boa parte de seu sistema é voltado para a utilização dos scripts para controlar o jogo, então mesmo com vários recursos que podem ser controlados através do editor, o sistema do jogo é melhor controlado através dos scripts. A Unity se mostrou uma ferramenta extremamente eficiente e com muitos recursos gratuitos que para um iniciante na área e para os mais avançados. Além de que em sua licença básica já disponibiliza recursos de monetização para os jogos.

No geral a Unity se mostrou uma ferramenta de criação de jogos que tem inúmeros motivos para ser uma das mais utilizada no mercado, mas também mostrou que ainda tem coisas a se melhorar, como implementar um sistema de edição de código interno e/ou uma melhor maneira de conexão com editor de códigos e ide externos, para que não ocorra as travadas, e tantos momentos de espera de reimportação, que pode serem demoradas em máquinas mais fracas.

GameMaker se mostrou uma engine bem útil na criação de jogos, principalmente em 2D, possuindo editor interno de códigos o que facilita demais no desenvolvimento e não tendo vários requisitos para se rodar no computador.

Além de ser uma boa opção de porta de entrada para aqueles que desejam começar no mundo de desenvolvimento de jogos, possuindo uma documentação simples de se entender e em sua forma gratuita já abrange diversos recursos para a criação de seu jogo. Tendo como principais desvantagens sua comunidade não ser tão grande e vídeos muitos antigos ficando obsoletos conforme o tempo vai passando e a *game engine* vai atualizando.

Comparando lado a lado as *game engines* é possível notar que a *GameMaker* tem uma curva de aprendizado menos íngreme, ocasionando em uma maior facilidade de aprendizado. Já em relação a documentação, elas se mostraram muito semelhantes, o que também ocorre nos recursos de aprendizado com as duas empresas disponibilizam tutoriais e guias de como fazer jogos completos ou mecânicas de jogo desde o zero.

Em relação a instalação, a *GameMaker* se mostrou mais prática, sendo simples passos para já poder utilizar a ferramenta, sem precisar de criação de contas. já a Unity é necessária ter uma conta além de que o processo de instalação é mais demorado pois é necessário a instalação de duas ferramentas diferentes antes da utilização dela.

A interface de ambas as ferramentas é fácil de entender, mas a GameMaker se destaca novamente, por ter menos informação na tela e uma melhor separação dos elementos facilitando a compreensão.

No desenvolvimento de jogo ambas mostraram ser extremamente eficiente, com recursos que facilitam muito o processo de criação com meios simples de inserção de *sprites,* áudio, animação dos *sprites* além de recursos de depuração de código. Apesar de ambas terem recurso de depuração de código há uma grande diferença, pois a Unity utiliza da linguagem de programação C# que já extremamente utilizada no mercado de desenvolvimento no geral, o que pode ser mais conveniente para casos de pessoas que já conhecem a mesma, só que existe o fato de que a Unity não tem editor de código interno o que gera mais um ponto positivo para o GameMaker, que apesar de utilizar uma linguagem de programação própria, a GML, tem editor de código interno agilizando o processo de criação do jogo.

As duas *game engines* possuem licenças gratuitas, só que a Unity se destaca no fato de que na sua licença básica já permite a compilação final, monetização e venda do jogo, já a GameMaker exige a compra de outro plano para que o jogo possa ser vendido.

6 Conclusões

Os principais objetivos eram relacionados a ter uma visão ampla e comparar os recursos e utilização de cada *game engine*, de tal forma a poder gerar uma pequena análise que facilitaria para outras pessoas que querem entrar nessa área de desenvolvimento de jogos escolher entre essas ferramentas.

No geral a Unity se demonstrou como uma ferramenta mais versátil, só que menos atrativa para iniciantes, devido a sua complexidade inicial, sua documentação pouco intuitiva, e muitos recursos semelhantes que podem acarretar uma má compreensão de como utilizar os mesmos. Apesar disso é uma ferramenta que permite o desenvolvimento de diversos tipos de jogos, e para jogos 2D em específico, se mostrou extremamente competente, de tal forma que não é necessário muito conhecimento anterior para utilização e criação de jogos na mesma.

O GameMaker em sua grande parte é muito intuitivo principalmente para desenvolvedores que desejam iniciar no mundo dos jogos, possuindo uma interface intuitiva e uma documentação de simples entendimento facilitando a compreensão na hora da criação dos códigos e em sua versão gratuita existem diversos recursos ajudando na criação do jogo. Mas se deseja criar um jogo para comercialização, é necessário a compra das licenças, elas possuem um preço elevado e não oferecem tantos benefícios se comparar com as de outras *games engine*s.

No geral a GameMaker se demonstrou como uma ferramenta mais amigável para iniciantes e não necessita de muitos recursos físicos para utilização da mesma e por ser focada somente em desenvolvimento de jogos 2D, ela se mostra mais eficiente na criação desse tipo de jogos. Já a Unity é uma ferramenta mais versátil e mais utilizada no mercado, além de que com sua licença básica já se tem o suficiente para colocar um jogo para vender no mercado, só que ela é uma ferramenta mais complicada para alguém sem conhecimento nenhum de desenvolvimento de jogos, o que pode ocasionar em muitas revisitas e reestudo para entender algumas das ferramentas e recursos disponibilizados por ela, mesmo tendo uma comunidade maior e mais ativa.

Conclui-se então que a GameMaker é uma ferramenta melhor para quem está iniciando na área e quer aprender enquanto a Unity já é melhor para pessoas que já tem um conhecimento prévio e que tem como objetivo já colocar o jogo para vender no mercado.

REFERÊNCIAS

ALVES, Gelderson Bezerra. **ESTUDO COMPARATIVO ENTRE *ENGINE*S DE DESENVOLVIMENTO DE JOGOS 2D**. 2021. 45 f. Monografia (Especialização) - Curso de Engenharia de Software, Universidade Federal do Ceará, Quixadá, 2021.

CAVALCANTE, Carlos Henrique Leitão; PEREIRA, Maria Luciana Almeida. Comparativo entre Game *Engine*s como Etapa Inicial para o Desenvolvimento de um Jogo de Educação Fina. In: CONGRESSO SOBRE TECNOLOGIAS NA EDUCAÇÃO, 3., 2018, Canindé. **Anais do III Congresso sobre Tecnologias na Educação.** Fortaleza: S.L., 2018. p. 536-542.

COSTA, Diego Passos; GOMES, Fabio Fonseca Barbosa; DUARTE, Ricardo. ESTUDO COMPARATIVO ENTRE AS GAME *ENGINE*S *UNITY* E OGRE. **Computação Aplicada**, S.L., v. 5, n. 1, p. 18-25, jun. 2016.

GREGORY, Jason. **Game *Engine* Architecture**. 3. ed. S.L: A K Peters/Crc Press; 3Rd Edition, 2018.

HABGOOD, Jacob; OVERMARS, Mark; WILSON, Phil. **The Game Maker's Apprentice**: game development for beginners. S.L: Apress, 2006.

HOCKING, Joe. ***Unity* in Action**: multiplatform game development in c#. 3. ed. S.L: Manning, 2022.

SCHEEL, Jesse. **The Art of Game Design**: a book of lenses. S.L: Morgan Kaufmann Publishers, 2008.

TEKINBAS, Katie Salen; ZIMMERMAN, Eric. **Rules of Play**: game design fundamentals. S.L: The Mit Press, 2003.