PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Geração Procedimental em Jogos Eletrônicos: Desenvolvimento de um Dungeon Generator

Thiago do Prado Cerqueira

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC
DEPARTAMENTO DE INFORMÁTICA
Curso de Graduação em Engenharia da Computação

Rio de Janeiro, Dezembro 2023

Thiago do Prado Cerqueira

Geração Procedimental em Jogos Eletrônicos: Desenvolvimento de um Dungeon Generator

Relatório de Projeto Final, apresentado ao programa Engenharia da Computação da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação

Orientador: Augusto Cesar Espíndola Baffa
Departamento de Informática

Rio de Janeiro

Dezembro 2023

“To create a new standard, you have to be up for that challenge and really enjoy it.”

- Shigeru Miyamoto

Agradecimentos

Aos meus pais **Fernando** e **Mariana**, à minha irmã **Giovana**, e ao meu namorado **Daniel** por todo o apoio que me deram durante meus estudos.

Ao meu orientador, **Augusto Baffa**, por toda a atenção e auxílio entregues a este projeto e a mim nesses últimos 2 semestres.

Aos meus amigos dos cursos de Engenharia, **Rogerio Alvarez da Rocha** e **Ana Clara Filgueiras Granato** por todo o suporte e apoio na nossa trajetória pela PUC-Rio.

Aos meus amigos de fora da faculdade, **Nina Ambrosio**, **Gabriel Chuairi**, **Ana Carolina Xavier**, **Anna Laura Ferreira** e **Flávio Lima** por toda ajuda e incentivo que me deram durante o curso.

A todos os membros da Prisma Game Lab por me ajudarem a cultivar ainda mais o meu interesse em jogos eletrônicos nos últimos dois anos.

Ao Departamento de Informática da PUC-Rio por todo o suporte.

Resumo

Cerqueira, Thiago. Baffa, Augusto. **Geração Procedimental em Jogos Eletrônicos: Desenvolvimento de um Dungeon Generator**. Rio de Janeiro, 2023. 30p. Relatório de Projeto Final II – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Na indústria de desenvolvimento de jogos eletrônicos, diversas técnicas de geração procedimental são utilizadas visando reduzir, entre outros fatores, a carga de trabalho de programadores e artistas e os custos de produção. Considerando os vários algoritmos existentes, este projeto apresenta os estudos e apresentações breves de alguns dos mais comuns e as etapas do desenvolvimento de um gerador de dungeon para jogos eletrônicos utilizando um dos algoritmos de geração procedimental. O resultado foi um conjunto de códigos capaz de produzir uma dungeon que utiliza o algoritmo de Wave Function Collapse e que pode ser adaptado e utilizado em projetos de desenvolvimento de jogos completos.

Palavras-chave:
Gerador de Dungeon, Geração Procedimental, Jogos Eletrônicos, Wave Function Collapse

Abstract

Cerqueira, Thiago. Baffa, Augusto. **Procedural Generation in Video Games: Development of a Dungeon Generator**. Rio de Janeiro, 2023. 30p. Relatório de Projeto Final II – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

In the video game development industry, several procedural generation techniques are used, aiming to reduce, among other factors, the programmers' and artists' workload and the production costs. Considering all of the existing algorithms, this project presents the studies and brief presentations of some of the most common and the development steps of a functional dungeon generator for video games using one of the procedural generation algorithms. The result was a collection of code scripts capable of producing a dungeon using the Wave Function Collapse algorithm and that can be adapted and used in full game development projects.

Keywords:
Dungeon Generator, Procedural Generation, Videogame, Wave Function Collapse

Lista de Imagens

Lista de Algoritmos

Índice

1. Introdução

O mundo independente (“indie”) de desenvolvimento de jogos eletrônicos sempre contou com grandes desafios no mercado, em sua maioria em respeito a limitação de recursos, tempo e visibilidade [1], especialmente quando competindo com as maiores empresas do ramo, como a Nintendo, que produzem jogos categorizados como AAA (“*triple-A*”), categoria a qual pertencem jogos de grandes orçamentos, alta publicidade, produzidos e distribuídos por grandes empresas [2]. Contudo, com o passar dos anos, essa disparidade entre desenvolvedores indie e AAA tem sido reduzida cada vez mais, especialmente após o advento de plataformas como a Steam, a App Store e o Google Play, que permitem a publicação de jogos sem a necessidade de terceiros, mas não só isso: a popularização das técnicas de geração procedimental permite que até as menores equipes de desenvolvimento construam jogos com escopo e complexidade muito maiores do que o imaginado, ativamente competindo com aqueles produzidos por grandes estúdios [3].

1.1 Geração Procedimental

De acordo com o Massachusetts Institute of Technology (MIT) [4], geração procedimental é a criação de dados pelo computador. Utilizando um sistema de IA, é possível que um código passe a ser inteiramente responsável por aspectos de um jogo que normalmente recairiam sobre um desenvolvedor humano, economizando tempo e dinheiro que podem ser alocados a outras partes do processo.

Além da redução dos custos, a utilização de algoritmos de geração procedimental também permite ganhos consideráveis em armazenamento, uma vez que não será necessário guardar níveis inteiros na memória do dispositivo, e no fator conhecido como “replay-value” (“replayability” ou ainda “rejogabilidade”, em português). De acordo com Oxford Learner’s Dictionary [5], este é um fator abstrato que define o quão interessante é a proposta de se jogar o jogo mais de uma vez. Considerando os conteúdos gerados de forma dinâmica, dificilmente um jogador terá experiências iguais tão cedo, permitindo que ele explore diversas estratégias.

1.2 *Dungeon*

Segundo o site Engaged Family Gaming, *Dungeons* podem ser definidas como um local, normalmente um conjunto de salas e corredores, que possui desafios de combate ou elementos e *puzzles* de lógica, que normalmente oferecem prêmios mais significativos que outras áreas do jogo [6]. Comumente, a dificuldade dos combates e a complexidade dos *puzzles* lógicos progride conforme o jogador fica mais experiente. São regiões constantemente encontradas em jogos do gênero *Massive Multiplayer Online Role-Playing Game* (MMORPG), como Wizard101 (2008), e *Adventure*, como The Legend of Zelda: Tears of The Kingdom (2023), entre outros. A figura 1.1 mostra a dungeon Blackrock Depths, de World of Warcraft (2004), possivelmente a maior dungeon já vista em qualquer MMORPG, segundo o site GameRant [7].

Figura 1.1 - Blackrock Depths Dungeon (World of Warcraft) [8]

Dungeons também podem desempenhar diferentes papéis em determinados tipos de jogos. Para MMORPGs, por exemplo, é comum que essas regiões sejam pontos em que os jogadores se reúnem para completá-las repetidas vezes, visando obter pontos de experiência ou equipamentos e armas raras para fortalecer seu personagem, estratégia conhecida como *farming* [9].

Um dos jogos mais antigos a utilizar essas técnicas foi o Rogue em 1980, tão revolucionário que dele surgiu um gênero de jogos totalmente novo, conhecido como “roguelike”. Rogue possui um sistema de *Dungeon Generation* que gera um nível totalmente diferente toda vez que o jogador inicia, acoplado a um sistema de morte permanente, no qual uma vez que o jogador perca, todo o

progresso feito no jogo também é perdido, exceto pelos conhecimentos técnicos adquiridos que são reaproveitados ao iniciar o jogo novamente [10].

1.3 Este projeto

Similarmente ao feito por Rogue, este projeto se propõe a implementar um algoritmo para criação procedimental de uma Dungeon e seus elementos, de forma que seja sempre possível atingir o destino final.

Este documento está organizado em 9 seções de conteúdo. A primeira seção é esta introdução, que visa apresentar o tema do projeto e explicitar de forma sucinta seus objetivos. A seção 2, Situação Atual, discorre sobre o presente estado do uso de dungeons e geração procedimental, com um olhar focado na indústria de jogos eletrônicos. A terceira seção entra mais a fundo nos objetivos que se deseja atingir com o projeto, assim como a definição de outras especificações. A seção 4, Plano de Ação, apresenta brevemente diversos conceitos que foram julgados necessários para o entendimento dos processos executados e explicados na seção seguinte. A seção 5 de metodologia explica os algoritmos utilizados no desenvolvimento do projeto, ilustrados com pseudocódigos. A sexta seção explicita os testes que foram realizados para garantir o bom funcionamento dos códigos desenvolvidos e usados na parte anterior. A sétima discorre sobre os resultados finais do desenvolvimento e quais objetivos previstos foram alcançados. A seção 8 é a conclusão do trabalho realizado e a elaboração de possíveis próximos passos caso busquem dar continuidade a este projeto. A nona e última seção lista todas as referências externas utilizadas no decorrer deste documento para fins de consulta ao material fonte.

2. Situação Atual

Desde o surgimento da geração procedimental em videogames na década de 1980, como em Rogue e Pitfall, diversos jogos, sejam eles indies ou de grandes estúdios, utilizam as técnicas para criação dinâmica de conteúdo e, por conta disso, ganharam reconhecimento internacional.

Minecraft (2009), ainda que tenha sido adquirido pela Microsoft em 2014, já era um dos jogos indie mais bem sucedidos, visto o valor de 2.5 bilhões de dólares pagos pela empresa [11], parcialmente graças ao seu uso de geração procedimental. O jogo possui um algoritmo para gerar automaticamente um mundo inteiramente diferente do anterior ao se iniciar um novo jogo. Ele também conta com um sistema de “seeds” que permite que o algoritmo produza um terreno de forma específica. Dada uma mesma “seed”, o mesmo terreno será originado. Essa possibilidade de explorar e se aventurar mundos diferentes é um dos fatores mais atrativos de Minecraft, combinado com o gênero sandbox.

Contudo, Minecraft está longe de ser o único jogo no mercado que se apropria da geração procedimental. Diversos grandes nomes na indústria, como No Man’s Sky (2016) e Stardew Valley (2016), possuem alguma forma de geração dinâmica de conteúdo.

Além da geração procedimental, jogos com um sistema de dungeons continuam extremamente populares em 2023. Considerando a vitória de “The Legend of Zelda: Tears of the Kingdom” na categoria “Most Anticipated Game” do The Game Awards em 2022 [12], e sua indicação a “Game of the Year” no The Game Awards 2023 [13], a maior honra que um jogo pode receber em relação a premiação, fica claro que o público consumidor de videogames ainda espera ansiosamente por jogos com essa característica. A franquia “The Legend of Zelda”, que teve seu primeiro jogo lançado em 1986 no Japão e em 1987 nos Estados Unidos e rapidamente cresceu para se tornar não só uma das maiores da Nintendo, como uma das franquias mais reconhecidas mundialmente, fortemente utiliza sistemas de dungeon de forma consistente em todos os seus principais jogos, como ilustrado na figura 2.1 na página seguinte.

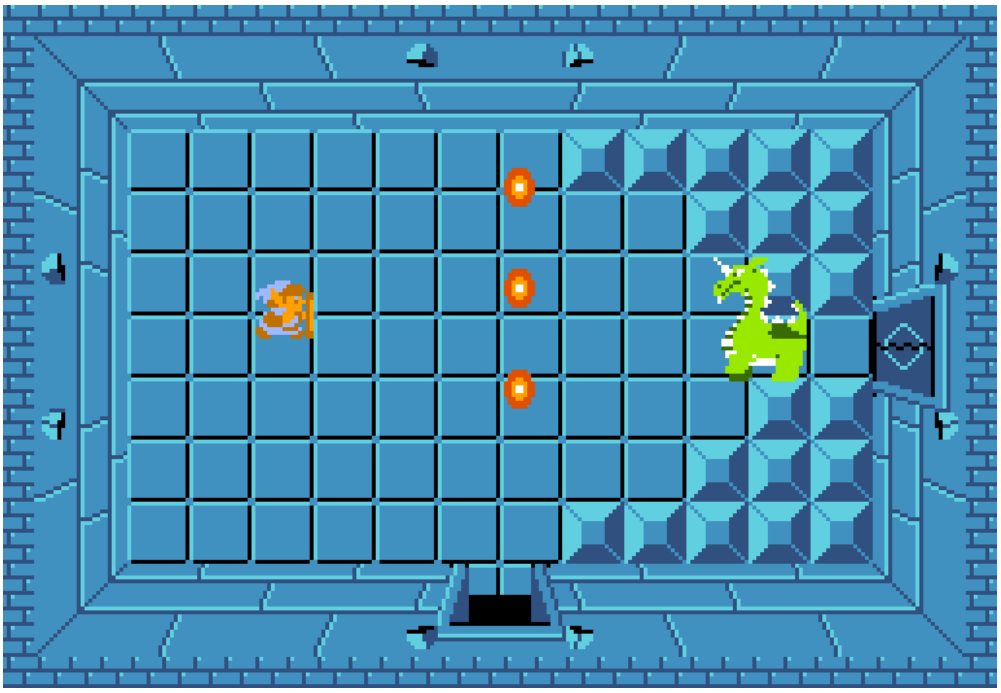

Figura 2.1 - Dungeon de The Legend of Zelda 1987 (Fonte: ZELDA DUNGEON, 2017)

A combinação das técnicas de geração procedimental com um sistema de *dungeon*, inicialmente implementado por Rogue, ainda se mostra bastante popular. Jogos como The Binding of Isaac (2011), sua sequência The Binding of Isaac: Rebirth (2014), Crypt of the NecroDancer (2015) e Hades (2020), todos do gênero Roguelike, continuam recebendo avaliações extremamente positivas de usuários até hoje nas lojas digitais. Hades, da SuperGiant Games, chegou a receber os prêmios de “Best Independent Game” e “Best Action Game” no The Game Awards em 2020 [14], mostrando não só que a combinação funciona de forma excelente, mas também que é muito bem recebida pelo público.

2.1 Algoritmos de Geração Procedimental

A seguir serão detalhadas algumas das soluções existentes e utilizadas atualmente para a realização de geração procedimental.

2.1.1 *Cellular Automata*

O algoritmo de *cellular automata* consiste em uma grade (ou matriz) de células que se encontram em um determinado estado (dentre um número finito pré-estabelecido de estados), normalmente representado por um inteiro [15].

Embora inicialmente simples, é possível adicionar conjuntos de regras e condições para utilizar o *cellular automata* para produzir resultados mais complexos. Dentre os usos mais famosos dessa solução encontra-se o

Conway's Game of Life, um jogo sem jogadores idealizado por John Horton Conway em 1970 [16].

O algoritmo para Conway's Game of Life é composto de apenas 2 estados, vivo ou morto, e a execução segue uma regra muito simples: para uma determinada célula "viva", se menos de 2 ou mais de 3 vizinhas estão "vivas", então ela morre, caso contrário, ela permanece "viva" para o próximo ciclo. Uma célula "morta" volta à vida se exatamente 3 vizinhas estiverem "vivas" [16].

A partir de uma disposição inicial definida pelo usuário, são realizadas diversas etapas de execução até que todas as células se tornem "mortas" ou que o algoritmo convirja para uma execução infinita. A imagem 2.2 abaixo ilustra uma das primeiras disposições iniciais descobertas que fazem o jogo ser infinito, a Gosper Glider Gun, encontrado pelo matemático Bill Gosper ainda em 1970 [17].

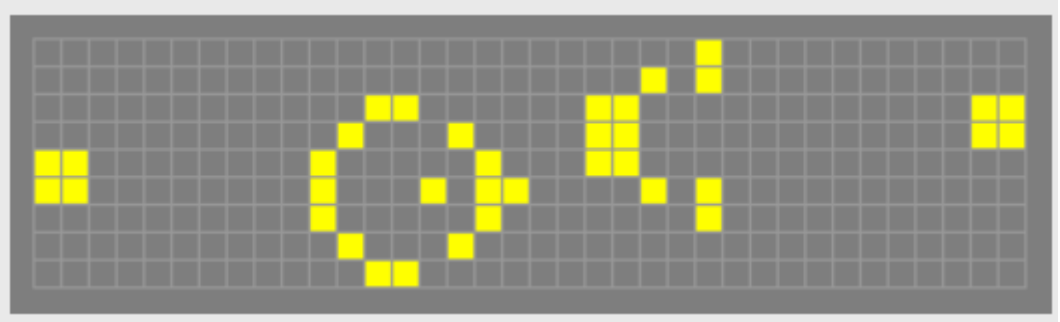

Figura 2.2 - Gosper Glider Gun [17]

2.1.2 *Perlin Noise*

Perlin Noise é um algoritmo popularmente utilizado para a geração de texturas e terrenos que recebe como parâmetros uma determinada quantidade de números *float* dependendo da dimensão. Para o caso mais comum, de duas dimensões, os parâmetros (x e y) podem ser interpretados como posições dos pixels. Para o caso de uma textura, por exemplo, a função de *Perlin Noise* é chamada para cada pixel para determinar qual cor ele vai receber, normalmente em uma escala de cinza [18].

A vantagem que esse algoritmo tem sobre uma geração simples de números pseudo-aleatórios é a suavidade entre os valores gerados. Um gerador simples pode produzir valores extremamente distantes em intervalos curtos de tempo, enquanto no *Perlin Noise* as diferenças são bem menos discrepantes, dando ao resultado uma aparência mais orgânica ou natural. Na geração de terreno, por exemplo, é possível criar ambientes de nível irregular, mas sem uma disparidade brusca entre altitudes que deveriam ser "vizinhas" [19].

Perlin Noise é um algoritmo versátil criado por Ken Perlin em 1980 enquanto trabalhava na produção do filme Tron, sendo utilizado por ele para a geração de efeitos visuais. Em 1997, Perlin recebeu um Academy Award pelo avanço tecnológico por esse trabalho [19].

A figura 2.3 a seguir exemplifica um terreno gerado com *Perlin Noise*.

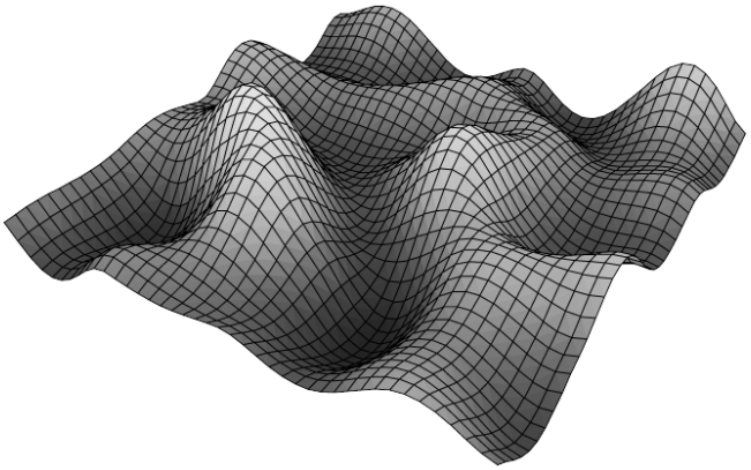

Figura 2.3 - Terreno gerado por Perlin Noise [20]

2.1.3 *Drunkard's Walk*

De acordo com a Noveltech [21], *Drunkard's Walk*, também conhecido como *Random Walk*, é um dos algoritmos de geração procedimental mais simples de ser implementado e pode ser utilizado para fabricação de desenhos interessantes, que podem ser usados como cavernas, por exemplo. Ele consiste na exploração de uma grade ou matriz, com passos equivalentes a uma célula, até que o número desejado de células seja atingido.

Esse objetivo é alcançado de maneira simples através de código, seguindo 3 passos principais. Primeiramente, determina-se a célula atual como "explorada" (ou "vazia"), então se escolhe aleatoriamente uma das quatro direções cardinais (desconsiderando movimentos na diagonal) e se move naquela direção. Por fim reinicia-se o loop até atingir um certo número de células exploradas (op. cit.). A figura 2.4 na página seguinte ilustra um mapa desenhado por meio da implementação de um algoritmo *Drunkard's Walk*.

Figura 2.4 - Resultado de Drunkard's Walk [22]

Ainda segundo a Noveltech, a simplicidade de implementação é um aspecto muito atrativo, considerando que, por isso, é possível facilmente adaptá-lo para algoritmos de geração mais complexos.

Contudo, um dos maiores problemas com esta solução é que, pelo menos em sua forma crua, ela não é totalmente confiável, já que algumas vezes ela pode produzir resultados interessantes, mas também muitas vezes ocasiona em desenhos menos atrativos, pelo fato de ser extremamente aleatório [22].

2.1.4 Finalização

Os algoritmos apresentados estão entre algumas das possibilidades de soluções existentes quando o assunto é geração procedimental. Outro exemplo, mais recente, se trata do Wave Function Collapse, que foi utilizado neste projeto e será detalhado no capítulo 4.

3. Proposta e Objetivos do Trabalho

A ideia do projeto é desenvolver um algoritmo que crie uma *Dungeon* de forma procedimental, implementada com um sistema de *seed* que permita a seleção de uma geração específica, de forma que seja sempre completável e com *layouts* gerais e internos de sala gerados de forma aleatória.

Para isso, foi escolhida a Engine Unity [23] e a linguagem C#, que é a utilizada pela *engine*. A Unity se mostra bem versátil pois tem a possibilidade de exportar o jogo em formatos aceitos por iOS, Windows, Linux e Android, além de apresentar uma gama variada de tutoriais, fóruns e suporte na internet.

As tarefas principais que espera-se ter concluídas são:

- Um algoritmo capaz de gerar uma disposição jogável aleatória para as salas da *dungeon*;
- Um algoritmo que possibilite o preenchimento automático de salas da *dungeon*, de forma que ela seja sempre caminhável e possível de ser concluída;
- Um algoritmo que gere inimigos variados, a partir da combinação de partes;
- Um sistema de *seeds*, que resulte sempre em um mesmo resultado de geração para a mesma *seed*.

4. Plano de Ação

Será utilizada uma abordagem com reuniões quinzenais/semanais com o orientador, para alinhamento do que foi feito na última semana e relatar o que está planejado para a seguinte.

O planejamento envolve principalmente a familiarização e o estudo de algoritmos de geração procedimental e a seleção e implementação daqueles julgados mais adequados para cada tarefa.

Além disso, uma pesquisa para aprofundamento na linguagem C#, mais voltado para seu uso na engine escolhida, e das funções e ferramentas da própria Unity.

4.1 Estudos sobre algoritmo Wave Function Collapse

Segundo a Universidade de Southampton, Wave Function Collapse (WFC) é um evento da física quântica descrito como o momento em que um dado sistema com superposição de múltiplos estados se estabiliza em um deles, proposto pela primeira vez pelo físico alemão Werner Heisenberg [24].

No contexto da computação, WFC inspirou um algoritmo de geração procedimental que recebeu o mesmo nome, idealizado por Maxim Gumin e disponibilizado por ele em um repositório no GitHub em 2016 [25].

O algoritmo recebe uma imagem de entrada do usuário e retorna como saída outra imagem gerada procedimentalmente que seja similar à entrada, como demonstrado pela figura 4.1.

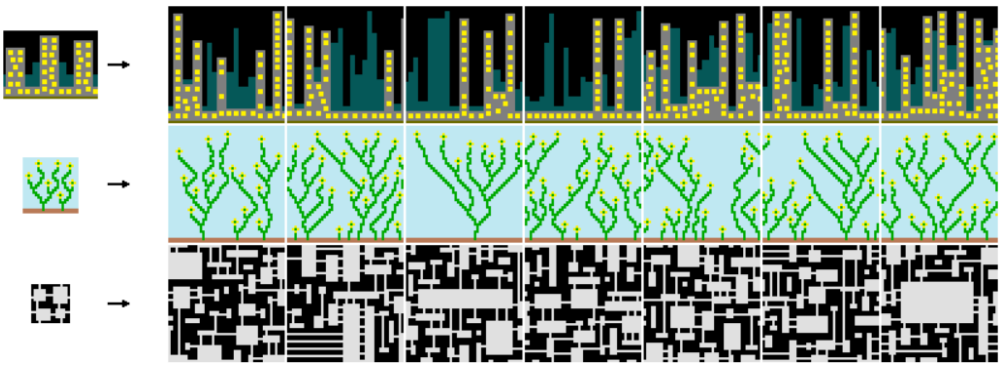

Figura 4.1 - Exemplos de entrada e saídas do Wave Function Collapse [25]

O Wave Function Collapse funciona a partir da superposição de estados. O algoritmo recorta várias regiões de tamanho NxN da imagem de entrada e cada uma dessas regiões é representada como um estado. A saída então inicialmente existe como várias regiões NxN em superposição (ou “não observadas”) até que

cada passo de observação selecione a com menor entropia e colapse em apenas um estado.

Esse evento desencadeia um efeito dominó: ao fixar o estado de uma região, suas vizinhas possuem o número de estados possíveis que se encaixam significativamente reduzido.

O processo é repetido até que a imagem de saída esteja em um estado totalmente observado. O resultado é uma combinação de inúmeros recortes da entrada dispostos e rotacionados para criar uma saída que seja diferente, porém similar ao padrão input do algoritmo, como exemplificado pela figura 4.2 a seguir.

Figura 4.2 - Exemplificação detalhada das regiões no resultado final [25]

O algoritmo foi adaptado para diversas linguagens e plataformas, incluindo a escolhida para esse projeto: a Unity. A versão do WFC para Unity foi publicada pelo usuário “selfsame” na plataforma itch.io [26] e será utilizada para geração interna das salas da dungeon. Seu uso será detalhado em seções mais adiante.

4.1.1 Vantagens e Desvantagens

Em uma palestra sobre o desenvolvimento do *roguelike* “*Caves of Qud*” [27], o desenvolvedor Brian Bucklew utilizou o *Wave Function Collapse* de Gumin e relatou seus pontos positivos e negativos do algoritmo no processo.

Segundo Bucklew, a principal vantagem do WFC é a facilidade com a qual resultados complexos são gerados. Ele cita que para replicar esse nível de output utilizando outros algoritmos conhecidos seria necessário elevar em mesma medida a complexidade do próprio algoritmo.

Entre as maiores desvantagens da utilização do *Wave Function Collapse*, Bucklew destaca 3:

- Homogeneidade: Bucklew notou que independentemente do tamanho definido para a imagem de *output*, a escala do padrão gerado será similar ao da entrada. Ao passar como *input* um quadrado vermelho, por exemplo, a probabilidade de se gerar alguns poucos quadrados vermelhos grandes na saída é exponencialmente menor do que a de gerar diversos quadrados

pequenos. Isso torna o algoritmo não muito confiável se o intuito é a geração de objetos de larga escala, como uma Catedral em um vilarejo.

- Sobreajuste: Entradas muito detalhadas tendem a produzir resultados menos diversos, pela natureza com a qual o algoritmo executa. Se um setor for muito detalhado, já existirão menos possibilidades de combinação que encaixem umas nas outras.
- Conectividade: O algoritmo de WFC não consegue garantir que duas regiões estejam conectadas uma à outra, novamente por conta da forma com a qual a saída é gerada.

Contudo, em um artigo produzido para a SBGames em 2020 [28], Pedro Minini e Joaquim Assunção concluem que as soluções para todos esses problemas são simples e envolvem apenas alguns passos de pré ou pós-processamento.

A solução para homogeneidade seria gerar a estrutura grande que se deseja previamente com outros meios e utilizar o WFC posteriormente para preencher seu interior. Essa solução foi aplicada no decorrer deste projeto, quando gera-se as salas da *dungeon,* como visto na seção 5.3.

Já para os casos de sobreajuste e conectividade, a solução seria em pós-processamento. Detalhes extras como portas e mobílias e a conectividade entre diferentes regiões podem ser adicionadas após a execução do WFC.

4.2 Estudos sobre Unity

A Unity é uma das game engines mais populares no mercado, juntamente com a Unreal, e os métodos em que as partes do jogo interagem são ditados, principalmente, por Objetos, Componentes e Scripts.

4.2.1 Objetos

Um objeto é toda e qualquer instância de uma entidade em uma cena do Unity e é definida pela classe GameObject [29], como jogador, inimigos, NPCs, cenário, partículas, botões, menus e outros elementos de UI. Eles interagem entre si e com a cena através de seus componentes.

4.2.2 Componentes

Os componentes são como blocos que definem características ou comportamentos distintos e podem ser acoplados a um objeto, permitindo uma

alta personalização no desenvolvimento com a engine [30]. Dentre os mais relevantes se encontram o transformador (que determina os parâmetros de posição, rotação e escala), caixas de colisão (que verificam se dois objetos estão em contato ou não) e scripts.

4.2.3 Scripts

Um script é um componente de código, comumente na linguagem C# no caso da Unity [31]. Pode ser entendido como um componente personalizado e que pode coordenar o funcionamento de um objeto e dos componentes atrelados a ele, bem como alterar parâmetros gerais da cena.

A Unity possui um conjunto de bibliotecas prontas com funções gerais para auxiliar no processo de desenvolvimento de código. Algumas das funções mais importantes disponibilizadas pela plataforma são a Start, que é executada uma única vez assim que o objeto é instanciado em cena, a Update, que é executada uma vez a cada frame e a Fixed Update, que é executada a cada frame antes todas as operações do sistema de física.

A Figura 4.3 abaixo retrata a hierarquia entre Objetos (em vermelho), Componentes (em roxo) e Scripts e suas funções (em azul).

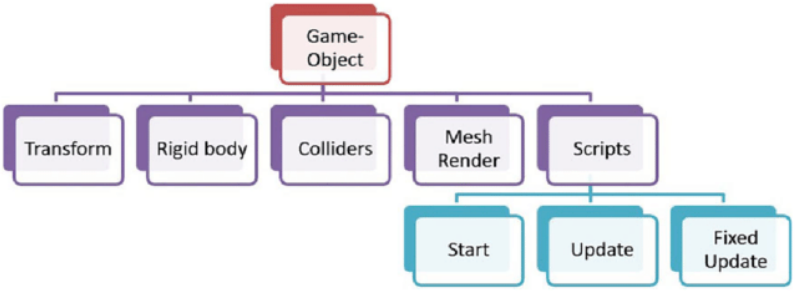

Figura 4.3 - Diagrama hierárquico da Unity [32]

4.2.4 Prefabs

O sistema de Prefabs da Unity permite transformar um Objeto em um Asset reutilizável, que passará a funcionar como uma espécie de modelo [33].

Um prefab é criado quando um Objeto na hierarquia é arrastado para a região dos arquivos do projeto Unity. Este pode então ser instanciado manualmente ou dinamicamente com Scripts para rapidamente gerar múltiplas cópias de um mesmo objeto de forma mais eficiente. No modo de edição de cena, quaisquer modificações realizadas ao Prefab na região dos arquivos são imediatamente transpostas a todas as cópias instanciadas em cena. Contudo, as cópias são totalmente independentes enquanto a cena estiver rodando.

Por exemplo, este recurso é útil para instanciar diversas cópias de um mesmo inimigo. No modo de edição alterar um atributo deste Prefab, como a vida máxima, vai aplicar a mudança a todas as suas cópias existentes. Porém, quando o jogador causa dano a um inimigo quando a cena estiver rodando, as outras cópias deste inimigo não são afetadas.

5. Metodologia

O projeto tem como objetivo principal o desenvolvimento de um sistema de dungeon de forma automática através de artifícios de Geração Procedimental, visando-se obter disposições que sejam sempre passíveis de serem completadas por um jogador.

Para melhor execução, o projeto foi dividido em 3 grandes etapas: a primeira consiste na geração da disposição geral das salas na *dungeon*, a segunda engloba a criação dos espaços internos de cada uma das salas e a terceira trata da instanciação de inimigos em cada uma das salas geradas.

O diagrama de classes apresentado na figura 5.1 exibe as classes que serão detalhadas nas subseções posteriores.

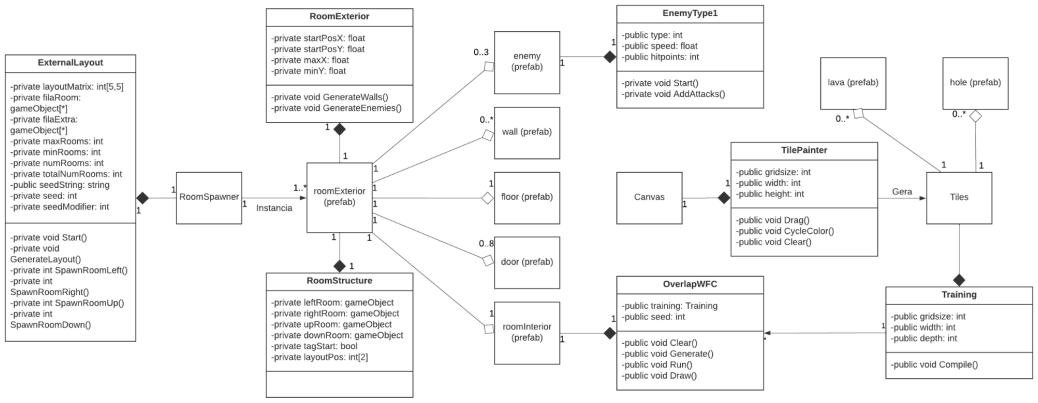

Figura 5.1 - Diagrama de Classes (Fonte: Autor)

5.1 Estrutura da Sala

A estrutura e atributos internos das salas são definidos no *script* "*RoomStructure.cs*".

Cada sala possui quatro variáveis GameObject (*upRoom*, *downRoom*, *leftRoom* e *rightRoom*) responsáveis por guardar referências às salas vizinhas desta nas quatro direções cardinais. Todas essas são iniciadas com valor default "*null*" e recebem a referência caso alguma sala seja criada conectada diretamente a ela. Contudo, como se trata de uma masmorra, ainda que ao final da geração duas salas sejam vizinhas uma da outra, elas podem não ter conexão direta entre si.

A variável *tagStart* é um valor booleano que define se a sala é uma sala inicial ou não. Apenas a primeira sala gerada pelo algoritmo possui esse valor como *"true"*, para evitar que a primeira sala da dungeon gere inimigos ou

interiores variados com Wave Function Collapse. Por ser inicial, é ideal que ela sirva apenas como ponto de partida.

O último atributo deste script é *layoutPos*, um array de inteiros de tamanho 2, cuja função é armazenar as coordenadas (x, y) da sala em relação à matriz geral (detalhada em 5.2).

Todos os atributos são definidos como *private* e só são acessados por meio dos métodos *get* e *set* correspondentes para garantir o encapsulamento do código e evitar atribuições incorretas a essas variáveis.

Como esse script é exclusivo para o manuseio e armazenamento dos atributos de uma sala, ele não possui nenhum outro método além dos já citados, incluindo os métodos *Start* e *Update* padrão de componentes de código.

5.2 Layout da *Dungeon*

A disposição geral das salas envolve uma série de aspectos, como por exemplo o número máximo e mínimo de salas, onde elas serão geradas e com quais outras salas elas estarão conectadas.

Para a solução destas questões foi desenvolvido o *script* "*ExternalLayout.cs*", cujo conteúdo será detalhado a seguir.

Como forma a limitar o espaço em que salas podem ser criadas e poder facilmente verificar se um determinado espaço já está ocupado ou não, utilizou-se uma matriz de inteiros de tamanho 5x5, denominada "*layoutMatrix*". Cada coordenada (x,y) da matriz é traduzida em uma região no espaço físico da cena que pode ser ocupado por uma sala. Espaços vazios são indicados com valor "0" e, conforme vão sendo preenchidos, são substituídos pelo valor "1". A figura 5.2 na página seguinte ilustra um possível conteúdo da matriz antes e depois de uma geração.

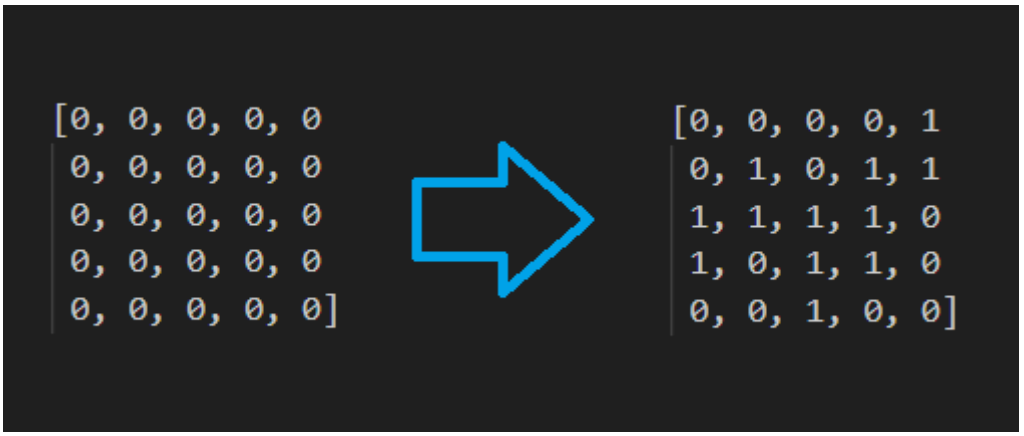

Figura 5.2 - Matriz antes e após a geração (Fonte: Autor)

Para organizar a ordem de geração são declarados dois arrays (*filaRoom* e *filaExtra*) que, como o nome indica, servem como uma fila, ordenando que salas terão vizinhos criados primeiro.

Os quatro atributos *maxRooms*, *minRooms*, *totalNumRooms* e *numRooms* servem para controlar a quantidade de salas a ser gerada em uma execução do código. *maxRooms* e *minRooms* são estáticas e armazenam as quantidades máxima e mínima, respectivamente, de salas que qualquer execução pode gerar. São os limites de salas. *totalNumRooms* é um número sorteado a cada execução entre *maxRooms* e *minRooms* e determina quantas salas devem ser geradas naquela execução apenas. *numRooms* é um contador e armazena quantas salas já foram criadas até o momento em uma dada execução. Essa última variável é usada para determinar quando parar o algoritmo de geração.

O método *Start* desse script inicializa a variável *totalNumRooms* e declara ambos os arrays de fila com tamanho igual a ela antes de chamar o método *GenerateLayout*.

A função *GenerateLayout* é a principal do script e coordena todo o processo de geração e conexão de salas na dungeon. Primeiramente é escolhida de forma aleatória uma posição na matriz para instanciar a sala inicial. Uma referência a essa sala é colocada na primeira posição do array filaRooms e a geração passa para o próximo estágio.

O segundo estágio consiste em percorrer o array *filaRooms* e gerar salas vizinhas para as referenciadas nele. Para cada iteração do for loop, obtém-se os parâmetros necessários da sala no índice atual do *filaRooms* (o script RoomStructure, a posição física da sala e a sua posição na matriz) e sorteamos um valor inteiro para a variável local *spawnProb*. Então são chamadas quatro funções responsáveis por gerar ou não uma sala vizinha, uma para cada direção

cardinal, que recebem todos os parâmetros citados anteriormente. Elas são *SpawnRoomRight*, *SpawnRoomLeft*, *SpawnRoomUp* e *SpawnRoomDown.* O algoritmo para esse estágio está ilustrado no pseudocódigo 5.1 abaixo.

```
Para cada sala em filaRoom
{
    roomsConnected = 0;

    Se a sala for nula ou  se já foi criado o número máximo de salas, encerre o loop;

    currentRoom = componente RoomStructure da sala atual;
    currentRoomPos = posição física da sala atual;
    currentPos = posição na matriz da sala atual;

    spawnProb = Random.Range(1, 6);

    roomsConnected += SpawnRoomRight();
    roomsConnected += SpawnRoomLeft ();
    roomsConnected += SpawnRoomDown ();
    roomsConnected += SpawnRoomUp   ();

    Se a sala atual ainda puder receber vizinhos, adicione-a ao array filaExtra;
}
```

Algoritmo 5.1 - For loop para segunda etapa

As quatro funções cardinais possuem exatamente a mesma estrutura, apenas com as constantes alteradas para corresponder a cada uma das direções. Elas seguem o modelo do pseudocódigo 5.2 a seguir.

```
private int SpawnRoom()
{
    int spawnProbRange;
    GameObject newRoom;

    Se a sala não está na borda da matriz
    {
        Se o espaço vizinho está vazio
        {
            spawnProbRange = Random.Range(1, 11);
            if (spawnProb < spawnProbRange)
            {
                newRoom = Instancia sala no local fisico adequado;
                Armazena a posição na matriz da newRoom;
                Referencie a sala atual no atributo correto na newRoom;
                Referencie a newRoom no atributo correto na sala atual;
                Adicione a newRoom a fila;
                Altere o valor na matriz de "0" para "1";
                Incremente o contador de salas geradas;

                Execute o wave function collapse na newRoom;

                return 1;
            }
        }
    }

    return 0;
}
```

Algoritmo 5.2 - Pseudo-código dos métodos cardinais

O modelo verifica se uma sala pode ser gerada naquela direção por meio de duas checagens: primeiro se confirma a posição da sala na matriz, para garantir que ela não esteja no limite e em seguida verifica-se se o espaço vizinho está vazio. Sabendo-se agora que a sala pode ter um vizinho naquela direção, sorteia-se um número inteiro que é comparado com o valor de *spawnProb* determinado anteriormente. Caso *spawnProb* seja menor, a sala é então gerada naquele espaço. Todas as referências e ajustes de parâmetros são realizados e, em seguida, o interior da nova sala é criado com o algoritmo de Wave Function Collapse.

Os valores de retorno desses métodos são adicionados a variável local *roomsConnected*, que terá, ao fim de cada iteração, quantas salas vizinhas foram geradas para a sala atual. Caso esse número não seja 4, a sala atual é adicionada ao array *filaExtra* para a última etapa.

O terceiro e último estágio da geração repete o mesmo processo da etapa anterior, porém utilizando o array *filaExtra*. Essa etapa existe para aproximar o número de salas geradas (*numRooms*) do valor sorteado para *totalNumRooms* e

reduzir significativamente as chances de uma execução terminar com poucas salas no total.

5.3 Configuração interna das salas

O design interno de cada sala é definido em dois momentos: o primeiro assim em que é instanciada em cena e o segundo com a execução do Wave Function Collapse no script *ExternalLayout.cs*, detalhado em 5.2.

O primeiro momento é quando são geradas as estruturas da sala, sendo elas as paredes, portas e o chão tomando como base as referências que a sala possui das suas vizinhas. Esse processo é realizado no script *RoomExterior.cs*.

Esse script possui os atributos *doorPrefab*, *wallPrefab* e *floorPrefab* que armazenam os prefabs de portas, paredes e chão, respectivamente, para futuramente instancia-los em cena nas posições corretas. Também são declaradas as variáveis *startPosX* e *startPosY*, que armazenam as coordenadas iniciais para a geração das paredes, e *maxX* e *minY*, que guardam os limites de até onde as paredes serão geradas.

O método *Start* nesse script calcula os valores para *startPosX, startPosY, maxX* e *minY*, obtém o componente *RoomStructure.cs* (detalhado em 5.1) da própria sala e chama a função *GenerateWalls*.

GenerateWalls é o principal método desse script e é o responsável por instanciar as paredes, portas e o chão da sala corretamente. O prefab do chão é instanciado na mesma posição que o objeto sala e ampliado de forma a cobrir inteiramente o tamanho da sala (que é padrão para todas as geradas). As paredes e portas são geradas em um loop while, como ilustrado no pseudocódigo 5.3 abaixo.

```
Enquanto a parede não atingir o tamanho certo
{
    Se está no meio da parede e há uma sala conectada a essa nessa direção {
        Instancie prefab de porta;
    } Senão {
        Instancie prefab de parede;
    }

    Incremente tamanho da parede;
}

Resete as variáveis para o próximo while loop;
```

Algoritmo 5.3 - Construção das paredes

Os prefabs da parede e da porta possuem tamanho 1x1. A sala possui tamanho 22x10, o que significa que as paredes horizontais da sala ou possuem

22 prefabs de parede, ou 20 de parede e dois de porta. As verticais, no mesmo raciocínio, possuem ou 10 prefabs de parede ou 8 de parede e dois de porta.

As portas são criadas apenas quando é identificado no script *RoomStructure.cs* se naquela direção existe uma sala conectada a essa. Em caso positivo, ao invés de se gerar os prefabs de parede nas 2 posições centrais, são geradas portas.

O loop ilustrado acima é repetido uma vez para cada uma das quatro paredes da sala, com os ajustes adequados de direção e posição inicial. A primeira a ser gerada é sempre a superior, seguindo em sentido horário.

O segundo momento da configuração do interior da sala ocorre quando o script *ExternalLayout.cs* ativa o algoritmo do Wave Function Collapse.

Como elaborado em 4.1, o algoritmo Wave Function Collapse para Unity utilizado no projeto foi encontrado na internet, como uma adaptação do original. Nessa versão, a imagem de origem inserida pelo usuário é criada na própria cena da Unity utilizando prefabs e, a partir dessa mesma imagem, cada uma das salas executa o algoritmo para modificar o próprio interior. A figura 5.3 retrata, similarmente à figura 4.1, um exemplo de entrada e saídas, desta vez para a engine.

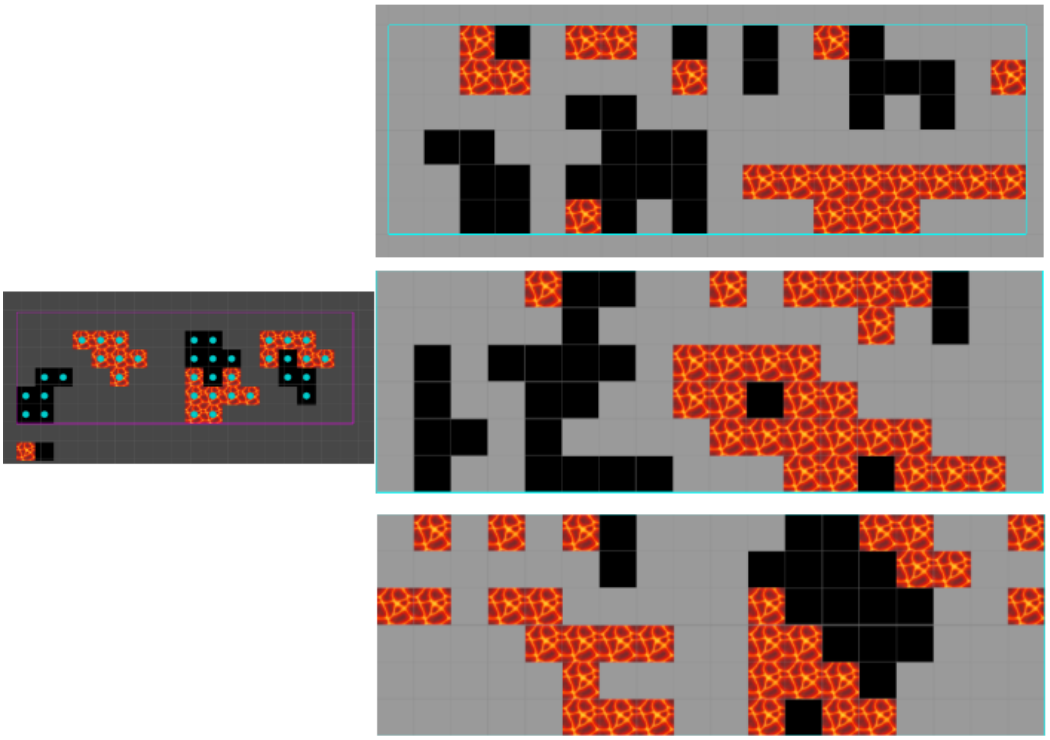

Figura 5.3 - Entrada e saída do WFC no Unity (Fonte: Autor)

Cada prefab de sala é composto por 2 objetos. O objeto *RoomExterior* é o objeto pai. Ele recebe como componentes tanto o script *RoomStructure.cs*

quanto o *RoomExterior.cs* e possui o objeto filho *RoomInterior*, cujo único componente é o script *OverlapWFC.cs* que executa o Wave Function Collapse na área determinada (é possível ver a área delimitada com uma linha azul na figura 5.2), disponível como parte do pacote encontrado online.

A área definida para receber o resultado do Wave Function Collapse foi definida como sendo de tamanho 18x6, ou seja, 2 de dimensão a menos em cada um dos lados. Isso foi feito para garantir que o algoritmo nunca gerasse o design sobre as paredes da sala e de forma a garantir que, independentemente do resultado de geração do WFC, sempre haverá um trecho de chão sem obstáculos contornando a sala, permitindo que um jogador alcance todas as portas sem que estas estejam bloqueadas. A imagem 5.4 abaixo ilustra a diferença entre o tamanho da sala e do design gerado dinamicamente.

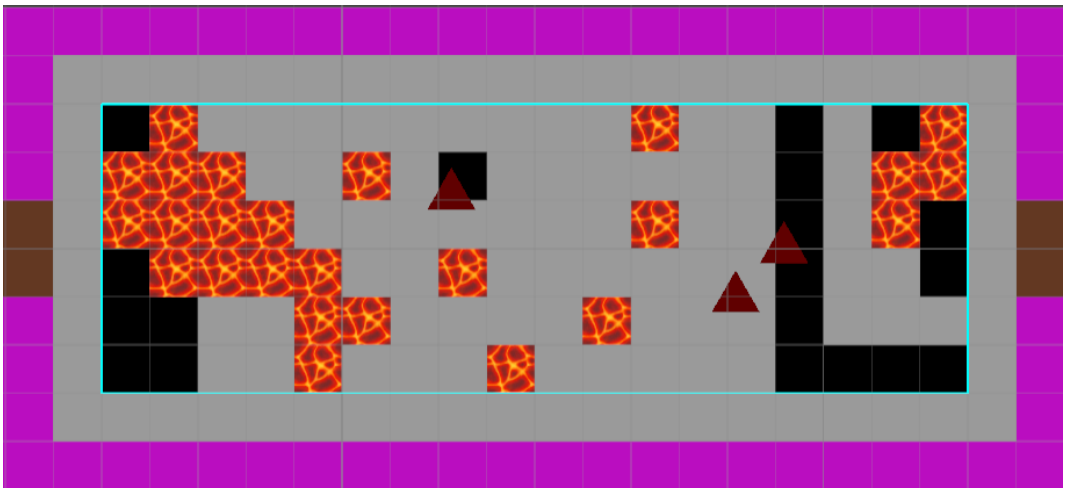

Figura 5.4 - Exemplo de sala completa (Fonte: Autor)

O desenho na Unity, ilustrado como a imagem mais à esquerda na figura 5.2, é criado antes da execução dos códigos, diretamente na própria cena do projeto com o algoritmo *TilePainter.cs*, também parte do pacote. O algoritmo *Training.cs* compila o desenho realizado para que este seja a base para o WFC executado mais tarde.

O atributo *training* do *OverlapWFC.cs* recebe o componente *Training.cs* do objeto canvas (o objeto que armazena o desenho) em tempo de execução logo antes de ser executado. A atribuição e execução ocorrem no script *ExternalLayout.cs*, especificamente no algoritmo 5.2.

5.4 Instanciação de Inimigos

Cada sala em uma dungeon possui inimigos que o jogador precisa derrotar para seguir adiante. Nesse projeto, com exceção da sala inicial, todas as salas

possuem 3 inimigos. A geração deles se dá pelo método *GenerateEnemies*, declarado também no script *RoomExterior.cs*, ilustrado pelo pseudocódigo 5.4 a seguir.

```
Se a sala não for inicial
{

    Enquanto o número de inimigos for menor que 3
    {
        Sorteie uma posição (x,y) dentro da sala;

        Se essa posição não possuir um inimigo, ela é valida;
        Caso contrário, é invalida;

        Se a posição for válida, instancie um inimigo nessa posição e incremente o número de inimigos;
    }
}
```

Algoritmo 5.4 - Instanciação de inimigos

As posições de inimigos já instanciados na sala são armazenadas em um array *enemyPos* e a verificação é feita apenas percorrendo esse array. Se a nova posição não estiver nele, é válida. Se já estiver, então é inválida.

Se a posição for válida, um inimigo será instanciado nela e ela será adicionada ao array, caso contrário o loop reinicia sem instanciar e uma nova posição é sorteada.

Cada inimigo possui um script *EnemyType1* que é responsável por determinar todos os parâmetros (como pontos de vida e velocidade) de uma instância. Os parâmetros são determinados sorteando um inteiro *type* e depois sorteando um inteiro *speed* e outro *hitpoints*, modificados pelo valor de *type*. O ideal é que mais velocidade implique em menos pontos de vida, portanto, *type* reduz o valor sorteado de hitpoints e aumenta o de velocidade.

Para o que seriam os ataques dos inimigos, foram criados 3 scripts em branco (*AttackA*, *AttackB* e *AttackC*). O método *AddAttacks* sorteia um valor para a variável *attacks* e dois dos três scripts em branco são adicionados ao objeto dependendo do valor sorteado, simbolizando a possibilidade de se poder combinar diversos ataques como bem entender. O pseudocódigo abaixo ilustra o funcionamento de *AddAttacks*.

```
int attacks = Random.Range(5, 8);

Se ataque < 7
{
    Adicione ataque A;
}

Se ataque > 5
{
    Adicione ataque C;
}

Se ataque for ímpar
{
    Adicione ataque B;
}
```

Algoritmo 5.5 - *AddAttacks*

Como os valores possíveis para ataque são 5, 6 ou 7, apenas duas das três condições serão ativadas por vez. Caso o valor seja 5, os ataques A e B são adicionados. Para o valor 6, são os ataques A e C. Com 7, B e C são adicionados ao objeto. Este é um trecho de código que pode ser rapidamente alterado para suportar mais variações de ataques.

5.5 Sistema de Seed

Quando se trata de elementos gerados dinamicamente por geração procedimental, um sistema de seeds se torna um aliado interessante, especialmente para fins de teste. O objetivo desse sistema é poder ditar para o código como a geração deve se dar. Sempre que uma mesma *seed* é utilizada, deve-se obter o mesmo resultado.

Como boa parte da aleatoriedade dos scripts discutidos se dá por meio do uso de funções e métodos do módulo Random da própria Unity, é possível aproveitar o sistema interno de seeds dela.

A biblioteca Random inclui um método *InitState* que recebe um número inteiro que será utilizado como a *seed* inicial para todos as funções que realizam geração de números pseudo-aleatórios.

No script *ExternalLayout*, discutido em 5.2, foi declarada uma variável de string *seedString*. Durante o modo editor da plataforma Unity, é possível alterar o valor dessa variável fora do código antes da execução. Qualquer string pode ser inserida para ser convertida em uma *seed* internamente. A conversão se dá pelo método nativo de string *GetHashCode* que retorna um inteiro equivalente ao *Hash Code* daquela string. Este inteiro é usado pela *InitState* como *seed*.

É válido ressaltar que existem muito mais strings possíveis do que números *Hash*, o que significa que, invariavelmente, diversas strings poderão retornar o mesmo código inteiro, mas isso não foi identificado como um empecilho para essa solução.

Em relação ao Wave Function Collapse, seu algoritmo *OverlapWFC* (discutido em 5.3) também apresenta uma variável *seed*. Antes da chamada desse algoritmo pelo *ExternalLayout*, essa variável é inicializada como o valor *Hash* obtido anteriormente adicionado de um inteiro *seedModifier*, que inicia em 1 e é incrementado em 1 sempre que uma sala nova é gerada, garantindo que nenhuma sala naquela execução terá o mesmo interior e que os interiores serão os mesmos sempre para um mesmo valor *Hash*.

6. Testes

Para realização dos testes foi criada uma cena separada e novos scripts que adaptam as funções presentes nos originais. Estes foram simplificados para que os testes foquem nas etapas de decisão "*if*", garantindo que as verificações realizadas são suficientes para que o algoritmo funcione como se espera.

6.1 Funções cardinais

As funções cardinais, detalhadas em 5.2, foram adaptadas para não gerar de fato salas e tiveram o código relativo a isso removido e substituído por uma simples instrução de Debug, a *Debug.Log*, nativa da Unity. Esta função imprime na tela, na aba de avisos da plataforma, qualquer string ou variável passada como parâmetro. Está sendo usada múltiplas vezes nos métodos, agora batizados de *TestSpawnRoom*, para indicar se a geração seria bem sucedida ou falha e imprimir a razão da falha.

Ainda nessas funções, os parâmetros que estão sendo checados são apenas aqueles que compõem as condições dos *if*. São esses: *currentPos* (um array que armazena a posição na matriz de uma sala), *spawnProb*, *numRooms* e *totalNumRooms* (detalhados em 5.2). Para cada um, foram criadas duas variáveis: uma que faz aquele atributo passar na checagem e uma que faz ele falhar na checagem. As checagens são aninhadas umas nas outras, cada uma com a própria chamada de *Debug*. O algoritmo 6.1 na página seguinte ilustra o modelo de função de teste para os métodos *SpawnRoom*.

```
Se ainda puderem ser criadas salas:
{
    Se a sala atual não está na borda da matriz:
    {
        Se o vizinho desejado está vazio:
        {
            Se a probabilidade foi favorável:
            {
                Debug.Log("Sucesso na geração de sala");
                return 1;
            }

            Debug.Log("Falha no spawnProb");
            return 0;
        }

        Debug.Log("Vizinho esta ocupado");
        return 0;
    }

    Debug.Log("Sala esta na borda");
    return 0;
}

Debug.Log("Falha no numero de salas");
return 0;
```

Algoritmo 6.1 - Teste das funções cardinais

Para essas funções, o teste é construído de maneira que, caso alguma das checagens falhe, ele prossiga imediatamente para a instrução de Debug.Log que apresenta o motivo da falha e em seguida retorne, impedindo a execução dos *Debugs* mais externos.

Com o algoritmo dessa forma, é possível verificar todos os casos em que ele deveria funcionar ou não de maneira simples, pois basta trocar os parâmetros que são passados para o método.

Assim como em 5.2, quatro métodos similares foram produzidos, um para cada uma das direções em que se é possível instanciar uma sala (cima, baixo, esquerda e direita), apenas alterando-se alguns valores para que as verificações sejam realizadas para o lado correto.

6.2 Geração de inimigos

De forma análoga ao teste descrito acima, foi elaborada a função *GenerateEnemiesTest*, com o objetivo de verificar a geração de inimigos no interior das salas, garantindo que nenhuma sala gere mais que 3 inimigos, que

nenhum inimigo seja gerado na mesma posição que um já existente e que a sala inicial não ative a geração. O pseudocódigo 6.2 a seguir demonstra o método de teste implementado.

```
public void GenerateEnemiesTest(RoomStructure roomStruct, float x, float y, float[,] enemyPos, int totalEnemies)
{
    Se a sala não for inicial
    {
        bool retry = false;

        Se ainda puderem ser gerados inimigos
        {
            retry = false;

            for (int i = 0; i < 3; i++)
            {
                Se ja existe inimigo nessa posição
                {
                    retry = true;
                    Debug.Log("Nao gerou, posicao ja ocupada");
                    return;
                }
            }

            Se puder gerar inimigo
            {
                Debug.Log("Gerou");
                return;
            }
        }

        Debug.Log("Nao gerou, maximo de inimigos");
        return;
    }

    Debug.Log("Nao gerou, Sala inicial");
    return;
}
```

Algoritmo 6.2 - Teste de geração de inimigos

Este método recebe as variáveis *roomStruct, x, y, enemyPos* e *totalEnemies* que são definidas antes da chamada da função. Cada parâmetro possui duas versões: a que faz o algoritmo retornar com erro e sem gerar o inimigo e a que permite a geração com sucesso.

Para que se obtenha o sucesso é necessário que *roomStruct* e *totalEnemies* sejam as variantes "*Success*" e que pelo menos uma entre *x* e *y* também seja "*Success*". Ambas *x* e *y* "*Fail*" resultam em uma posição já existente em *enemyPos* e falha no processo de geração. Caso *roomStruct* ou *totalEnemies* sejam a variante "*Fail*", a geração irá falhar, independentemente do valor das outras variáveis.

Os resultados da geração também são verificados por meio dos métodos *Debug.Log* ao invés de efetuar a instanciação na cena.

7. Resultados

Durante a fase de planejamento deste projeto foram estipulados diversos objetivos que os algoritmos deveriam atingir, listados no capítulo 3. A seguir serão apresentadas análises para refletir sobre o atingimento ou não de tais metas.

7.1 Layout jogável

Havia sido determinado que a geração de um layout de dungeon que fosse sempre jogável e passível de ser completamente terminado seria um dos objetivos a se alcançar.

Os algoritmos apresentados e os testes realizados indicam que todos os fatores para que isso possa ser considerado como concluído estão presentes:

- O algoritmo não gera salas completamente isoladas sem conexão a nenhuma outra, o que significa que todas as salas podem ser alcançadas por alguma direção;
- O design interno das salas, por uma restrição imposta manualmente, nunca irá gerar cenários em que portas da sala estejam bloqueadas por um obstáculo ou terreno, seja ele qual for.

Dadas essas características, é possível afirmar que o primeiro e segundo objetivos expostos no capítulo 3 foram totalmente atendidos pelo que foi desenvolvido.

7.2 Variação de inimigos

Além dos citados na seção anterior, a geração de inimigos diversos que utilizassem a combinação de partes também era um dos objetivos a serem atingidos pelo projeto.

A forma como estes são criados em cada sala é inteiramente dinâmica e em tempo de execução. Os métodos responsáveis pelos inimigos garantem que:

- Todas as salas, com exceção da inicial, irão possuir o mesmo número de inimigos ao serem instanciadas e que nenhum deles surgirão na mesma posição.

- Cada inimigo terá seu tipo sorteado imediatamente após serem gerados, alterando seus atributos físicos (vida e velocidade) de acordo.
- Um segundo sorteio adiciona dois ataques distintos dos três disponíveis, de forma independente do sorteio de tipo, de modo que dois inimigos do mesmo tipo possam ter ataques diferentes.

Todas essas garantias permitem uma grande variação nos inimigos instanciados, mesmo com poucas opções de personalização. Alguns dos inimigos serão iguais, o que é normal (e até esperado) em uma mesma dungeon, mas a maioria irá possuir alguma diferença dos demais, seja em um dos ataques ou em um dos seus atributos.

Dada a análise, é possível afirmar que esta meta foi alcançada com sucesso.

7.3 *Seed*

O último objetivo macro definido para este projeto envolvia a implementação de um sistema de seed que permitisse a replicação exata de uma geração específica.

O algoritmo desenvolvido garante:

- A pré-determinação da seed no modo de editor da Unity, para que o código utilize a seed inserida durante a geração;
- Que todo o interior das salas gerado pelo Wave Function Collapse seja diferente para cada sala daquela execução;
- Que, ao executar o código mais de uma vez com a mesma *seed*, o mesmo layout externo da dungeon e design interno das salas seja exatamente o mesmo, incluindo os inimigos gerados.

O único aspecto que o sistema atual não atende ocorre quando se executa o algoritmo sem uma *seed* inserida. Nesse caso, o código não é capaz de retornar a *seed* que foi utilizada, caso se deseje utilizá-la para reproduzir o resultado outra vez.

Fora isso, o sistema implementado funciona exatamente como se é esperado, podendo ser considerado concluído.

8. Conclusão

Com base no que foi apresentado nas seções 7.1, 7.2 e 7.3, pode-se afirmar que todos os objetivos previstos para o projeto foram alcançados com sucesso, o que significa que o Dungeon Generator desenvolvido é altamente funcional. A figura 8.1 na página seguinte ilustra o resultado da geração para a *seed* “Teste”.

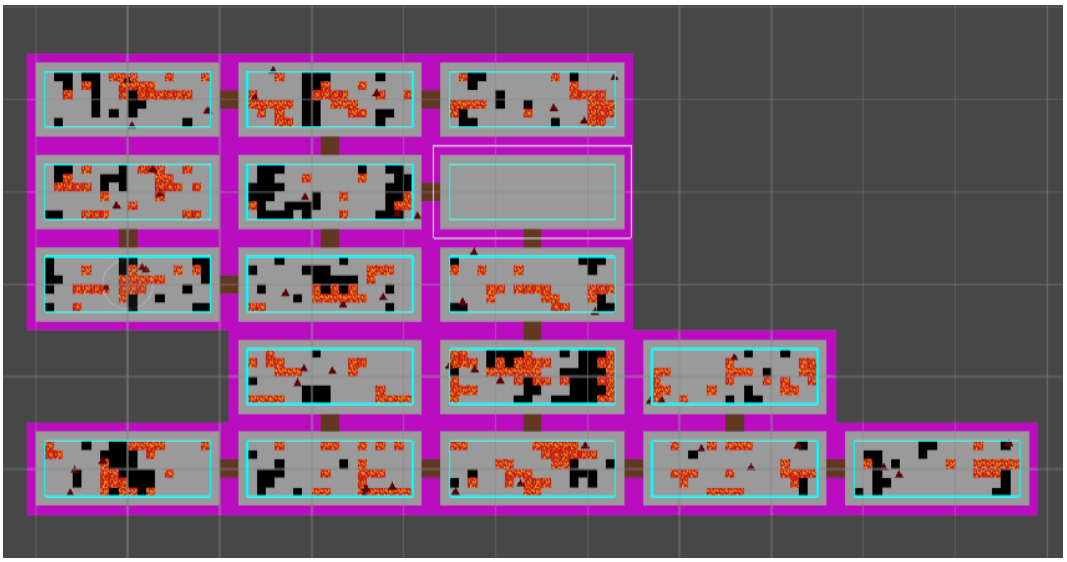

Figura 8.1 - *Dungeon* completa (Fonte: Autor)

Os algoritmos desenvolvidos e utilizados são capazes de produzir:

- Uma dungeon com no mínimo 10 e no máximo 20 salas, todas interconectadas entre si, de modo que é possível alcançar todas as salas a partir de qualquer uma;
- Limitações para garantir que nenhuma passagem entre salas esteja bloqueada;
- Uma geração que garante que o resultado aleatório do design interno das salas nunca irá se repetir em uma mesma geração;
- Três inimigos sistematicamente diferentes por sala;

Todos esses pontos permitem que o que foi desenvolvido possa ser diretamente utilizado no desenvolvimento de jogos e facilmente adaptado para as necessidades específicas (como por exemplo, tamanho da dungeon, designs internos de sala e número de inimigos).

8.1. Próximos Passos

Um próximo passo natural para um Dungeon Generator é a criação dos elementos que o transformem efetivamente em um jogo.

O desenvolvimento de um jogo envolveria a elaboração de diversos novos elementos, como o de um jogador capaz de explorar os terrenos gerados, atacar e derrotar os inimigos que habitam as salas e efetivamente concluir a dungeon. Envolveria a criação de diversos scripts para os próprios prefabs de chão, paredes e portas para detectar colisões e alterar o ambiente (como abrir as portas da sala) quando o jogador realiza determinada ação (como eliminar todos os inimigos da sala). Seria necessário também a própria codificação do comportamento e ataques de inimigos, como fazê-los detectar a presença do jogador. Até mesmo a personalização dos prefabs que compõem o terreno, utilizados pelo Wave Function Collapse, desde simples detecção de colisão até comportamentos mais complexos, como danificar ou reduzir a velocidade do jogador caso esteja sobre ele.

O processo de desenvolver efetivamente um jogo ao redor do Dungeon Generator está fora do escopo deste projeto, mas é uma evolução natural justamente pelo fato do projeto se tratar de uma mecânica específica da indústria de jogos eletrônicos.

Outras melhorias mais pontuais no próprio gerador podem ser realizadas, como por exemplo a redução do número de prefabs utilizados no desenho das paredes. Como os prefabs possuem tamanho 1x1, são necessárias diversas cópias para constituir todas as paredes. E esse número cresce consideravelmente ao se levar em conta o número de salas criadas por execução. É possível, por exemplo, esticar um prefab pequeno para que ele fique do tamanho desejado, como é feito com o chão das salas.

Além disso, pode ser implementado um meio de resolver as carências do sistema de *seed*, evidenciadas em 7.3, de maneira a torná-lo mais completo.

9. Referências

[1] YELLOWBRICK. **Navigating Challenges in the Indie Game Industry**. 2023. Disponível em: https://www.yellowbrick.co/blog/entertainment/navigating-challenges-in-the-indie-game-industry. Acesso em: 19 nov. 2023.

[2] ARM. **AAA Games**. Disponível em: https://www.arm.com/glossary/aaa-games. Acesso em: 17 nov. 2023

[3] BLATZ, Michael; KORN, Oliver. **A Very Short History of Dynamic and Procedural Content Generation**. 2017. Disponível em: https://www.researchgate.net/publication/315863952_A_Very_Short_History_of_Dynamic_and_Procedural_Content_Generation. Acesso em: 17 nov. 2023.

[4] MIT. **Procedural Generation**: creating 3d worlds with deep learning. 2019. Disponível em: https://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html Acesso em: 07 maio 2023.

[5] OXFORD LEARNER'S DICTIONARY. **Replay-value**. Disponível em: https://www.oxfordlearnersdictionaries.com/us/definition/english/replay-value. Acesso em: 17 maio 2023.

[6] ENGAGED FAMILY GAMING. **GAMING DEFINITION OF THE WEEK**: DUNGEON. 2022. Marie Rosa. Disponível em: https://engagedfamilygaming.com/parent-resources/gaming-definition-of-the-week-dungeon/. Acesso em: 11 nov. 2023.

[7] GAMERANT. **The Most Iconic World of Warcraft Dungeons from Each Expansion**. 2023. Disponível em: https://gamerant.com/world-of-warcraft-expansions-iconic-dungeon-designs-history. Acesso em: 11 nov. 2023

[8] WOWHEAD. **Blackrock Depths**. Disponível em: https://www.wowhead.com/classic/zone=1584/blackrock-depths. Acesso em: 11 nov. 2023

[9] TECHOPEDIA. **Farming.** Disponível em: https://www.techopedia.com/definition/19278/farming. Acesso em: 17 nov. 2023

[10] PROCEDURAL CONTENT GENERATION WIKI. **Rogue**. 2016. Disponível em: http://pcg.wikidot.com/pcg-games:rogue. Acesso em: 19 nov. 2023.

[11] MICROSOFT. **Microsoft purchases 'Minecraft'**. 2014. Disponível em: https://news.microsoft.com/announcement/microsoft-purchases-minecraft Acesso em: 09 maio 2023.

[12] THE GAME AWARDS. **Most Anticipated Game**: recognizing an announced game that has demonstrably illustrated potential to push the gaming medium forward. Disponível em: https://thegameawards.com/nominees/most-anticipated-game-presented-by-prime. Acesso em: 09 maio 2023.

[13] THE GAME AWARDS. **Game of the Year**. 2023. Disponível em: https://thegameawards.com/nominees/game-of-the-year. Acesso em: 15 nov. 2023

[14] THE GAME AWARDS. **Rewind**. 2022. Disponível em: https://thegameawards.com/rewind/year-2020. Acesso em: 09 maio 2023.

[15] ZGEB, Bronson. **Procedural Generation with Cellular Automata**. 2022. Disponível em: https://bronsonzgeb.com/index.php/2022/01/30/procedural-generation-with-cellular-automata/. Acesso em: 21 nov. 2023.

[16] MARTIN, Edwin. **Game of Life**: Info. Disponível em: https://playgameoflife.com/info. Acesso em: 21 nov. 2023.

[17] MARTIN, Edwin. **Game of Life**: Lexicon. Disponível em: https://playgameoflife.com/lexicon. Acesso em: 24 nov. 2023.

[18] RAOUF. **Perlin Noise**: A Procedural Generation Algorithm. Disponível em: https://rtouti.github.io/graphics/perlin-noise-algorithm. Acesso em: 21 nov. 2023.

[19] KHAN ACADEMY. **Perlin Noise**. Disponível em: https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise. Acesso em: 21 nov. 2023.

[20] SCRATCHAPIXEL. **Perlin Noise: Part 2**. Disponível em: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/perlin-noise-terrain-mesh.html. Acesso em: 21 nov. 2023.

[21] NOVELTECH. **Generating a 2D map using the Random Walk algorithm**. 2023. Disponível em: https://www.noveltech.dev/procgen-random-walk. Acesso em: 21 nov. 2023.

[22] JRHEAD. **Procedural Dungeon Generation: A Drunkard's Walk in ClojureScript**. Disponível em: https://blog.jrheard.com/procedural-dungeon-generation-drunkards-walk-in-clojurescript. Acesso em: 21 nov. 2023.

[23] UNITY. Disponível em: https://unity.com/pt. Acesso em: 17 maio 2023

[24] ROSS, Douglas. **Wavefunction Collapse**. Disponível em: https://www.southampton.ac.uk/~doug/quantum_physics/collapse.pdf. Acesso em: 24 out. 2023.

[25] GUMIN, Maxim. **Wave Function Collapse**. 2016. Disponível em: https://github.com/mxgmn/WaveFunctionCollapse. Acesso em: 24 out. 2023

[26] SELFSAME. **Unity Wave Function Collapse**. 2017. Disponível em: https://selfsame.itch.io/unitywfc. Acesso em: 24 out. 2023

[27] BUCKLEW, Brian. **Dungeon Generation via Wave Function Collapse**. 2019. Disponível em: https://www.youtube.com/watch?v=fnFj3dOKcIQ. Acesso em: 28 dez. 2023.

[28] MININI, Pedro; ASSUNÇÃO, Joaquim. **Combining Constructive Procedural Dungeon Generation Methods with WaveFunctionCollapse in Top-Down 2D Games**. 2020. Disponível em: https://www.sbgames.org/proceedings2020/ComputacaoShort/207911.pdf. Acesso em: 28 dez. 2023.

[29] UNITY. **Game Object**. 2023. Disponível em: https://docs.unity3d.com/ScriptReference/GameObject.html. Acesso em: 24 out. 2023

[30] UNITY. **Component**. 2023. Disponível em: https://docs.unity3d.com/ScriptReference/Component.html. Acesso em: 24 out. 2023

[31] UNITY. **Scripting**. 2023. Disponível em: https://docs.unity3d.com/Manual/ScriptingSection.html. Acesso em: 24 out. 2023.

[32] ALMOUSA, Amjed; SABABHA, Belal; AL-MADI, Nailah. **UTSim**: a framework and simulator for uav air traffic integration, control, and communication. 2019. Disponível em: https://www.researchgate.net/publication/335744769_UTSim_A_framework_and_simulator_for_UAV_air_traffic_integration_control_and_communication. Acesso em: 24 out. 2023

[33] UNITY. **Prefabs**. 2023. Disponível em: https://docs.unity3d.com/Manual/Prefabs.html. Acesso em: 29 out. 2023.