Algoritmo de geração de mapas para jogos do gênero *Metroidvania* através de técnicas procedurais *

Lucas Oliveira de Souza[1], **Roger Luis Hoff Lavarda**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia
do Rio Grande do Sul – Ibirubá – RS – Brasil

lucasosouza66@gmail.com, roger.lavarda@ibiruba.ifrs.edu.br

Abstract. *The video game genre Metroidvania is characterized mainly by non-linear exploration and progression of player abilities. There is an interdependence relationship between these characteristics, so that the player, by exploring the map, discovers new abilities which will allow them to overcome obstacles and progress throughout the game, and frequently must return to previously inaccessible areas, now with these abilities at their disposal. The task of building a world that presents these characteristics is up to the level designer, and it is crucial for a positive player experience and for the game's success. This work presents an algorithmic method to automate part of the process of designing levels for games of the Metroidvania genre and implements it via a plugin for the Godot game engine. By utilizing controlled randomness to organize predesigned rooms into a cohesive world, the foundation for an exploration experience containing the main characteristics of the genre is concretized in less than a second.*

Resumo. *O gênero de jogos eletrônicos Metroidvania é caracterizado principalmente pela exploração não linear do mundo e progressão das habilidades do jogador. Há uma relação de interdependência entre essas características, de modo que o jogador, ao explorar o mapa, descobrirá novas habilidades que o permitirão superar obstáculos e progredir no jogo, e por muitas vezes deverá retornar a áreas previamente inacessíveis, agora dispondo dessas habilidades. A tarefa de construir um mundo que apresente essas características cabe ao designer de níveis, e é crucial para uma experiência positiva para o jogador e para o sucesso do jogo. Este trabalho apresenta um método algorítmico para automatizar parte do processo de design de níveis de jogos do gênero Metroidvania e o implementa através de um plugin para o motor de jogos Godot. Utilizando aleatoriedade controlada para organizar salas pré-projetadas em um mundo coeso, a fundação de uma experiência de exploração que contém as principais características do gênero é concretizada em menos de um segundo.*

1. Introdução

O gênero *Metroidvania*, derivado das populares franquias *Metroid* e *Castlevania* (Figura 1), é caracterizado pelos elementos de exploração e de progressão das habilidades do jogador (PRADO et al., 2020). Esses elementos são interligados de forma entrelaçada:

*Trabalho de Conclusão de Curso (TCC) do curso Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul – Campus Ibirubá, 2025.

a exploração leva à descoberta de novas habilidades que, por sua vez, serão necessárias para acessar áreas inicialmente indisponíveis.

Figura 1. Captura de tela do jogo *Castlevania: Aria of Sorrow*. Fonte: o autor.

Apesar de haver, em geral, uma ordem para o cumprimento dos objetivos que recompensarão o jogador com novas habilidades – principalmente quando há a necessidade de se estabelecer sequência, como a sequência de eventos que formam a narrativa da história do jogo –, a exploração do mundo (também chamado de "mapa") ocorre de forma não linear. É comum ocorrerem situações nas quais o jogador encontra o local de acesso a áreas projetadas para serem acessadas apenas em um momento posterior. O que assegura que o jogador não navegue por essas áreas antes do planejado é a ausência das habilidades necessárias para isso. Por exemplo: o acesso pode ocorrer através de uma plataforma mais alta do que o protagonista consegue alcançar ao pular.

Ao explorar as áreas disponíveis, o jogador eventualmente obtém habilidades que lhe garantem meios de entrar nas áreas inicialmente inacessíveis. Seguindo o exemplo anterior, uma possível habilidade para alcançar a plataforma seria um segundo pulo que pode ser realizado no ar. Nesse ponto, o jogador deve realizar *backtracking*, ou seja, retornar a áreas já exploradas, agora com a solução ou "chave" para desbloquear uma nova parte do mapa.

A geração procedural de conteúdo é a geração de conteúdo de forma algorítmica em oposição à criação manual. Ela é há muito tempo aplicada no desenvolvimento de jogos como uma ferramenta para a produção de diversos elementos, como regras, histórias, texturas, terrenos e construções (HENDRIKX et al., 2013). Através de técnicas pseudo-aleatórias, como ruídos, combinadas com regras e restrições, é possível gerar pedaços de conteúdo tradicionalmente criados de forma manual por *designers*.

A construção manual de mapas e níveis em jogos do gênero *Metroidvania* é uma tarefa que exige alto grau de criatividade e planejamento por parte do designer, pois en-

volve equilibrar a liberdade de exploração com a progressão de habilidades e desafios. Por outro lado, algumas condições objetivas devem ser observadas, como a progressão linear correta dos desafios, a fim de evitar "furos" através dos quais o jogador possa pular partes da experiência. Todos esses aspectos incorrem em custos de tempo e recursos no processo de desenvolvimento.

A relevância desta obra reside na crescente adoção de técnicas de geração procedural de conteúdo na indústria de jogos. A geração automatizada de conteúdo pode ser aliada à construção manual para agilizar o trabalho mecânico e analítico sem abandonar o controle e a capacidade de polimento, ou ainda, em formas mais completas, ser incorporada no produto final para produzir experiências únicas sob demanda do jogador. Apesar desses benefícios, há poucos registros de sua aplicação para a produção de *Metroidvanias*.

Dessa forma, este trabalho busca oferecer um algoritmo capaz de gerar automaticamente mapas de jogos do gênero *Metroidvania*, empregando típicas estratégias procedurais de aleatoriedade controlada com validação imediata, dirigidas por um processo de tentativa e erro. Para ser considerado bem sucedido, serão avaliados alguns critérios. Primeiramente, o algoritmo deve produzir mapas formados por salas de diferentes tamanhos, em que pares de passagens conectadas sobreponham-se, mas que, à exceção disso, não ocupem simultaneamente o mesmo espaço. Outro critério essencial é a compilação de informações sobre o mapa gerado que indiquem como o conteúdo interativo deve estar organizado para assegurar a progressão correta. Finalmente, o tempo de execução não deve ser superior a alguns segundos, idealmente percebido como instantâneo pelo usuário que o aciona.

O texto está estruturado da seguinte maneira: a Seção 2 apresenta o referencial teórico, abordando os conceitos fundamentais sobre o gênero *Metroidvania*, técnicas de geração procedural e o motor de jogos *Godot Engine*, utilizado na implementação. A Seção 3 discute os trabalhos relacionados, destacando abordagens anteriores e suas limitações. A Seção 4 descreve detalhadamente a metodologia e arquitetura do algoritmo proposto, incluindo suas definições, estrutura e restrições de funcionamento. A Seção 5 apresenta os resultados obtidos e a análise de desempenho do algoritmo. Em seguida, a Seção 6 expõe as perspectivas de trabalhos futuros, propondo aprimoramentos e novas funcionalidades. Por fim, a Seção 7 traz as conclusões, sintetizando as contribuições deste estudo e suas implicações para o desenvolvimento de jogos com geração procedural.

2. Referencial teórico

Nessa seção são apresentados tópicos pertinentes ao desenvolvimento deste trabalho, contextualizando domínios abrangidos e apresentando ferramentas e métodos já existentes sobre os quais a proposta será desenvolvida.

2.1. Gênero *Metroidvania*

A palavra *Metroidvania* refere-se ao gênero de jogos definido por determinadas características, mas também é usada como denominação de um jogo pertencente a esse gênero. Essas características são: a não linearidade da exploração, a falta de uma separação em níveis e a progressão das habilidades do jogador (PRADO et al., 2020).

O gênero deve seu nome a duas franquias iniciadas em 1986: *Metroid* e *Castlevania*. Ambas mantiveram sua relevância ao longo dos anos, recebendo diversas

continuações com as quais interagiram gerações de jogadores. O sucesso do gênero pode ser observado em lançamentos recentes, como *Ori and the Blind Forest* (2015), *Hollow Knight* (2017), *Blasphemous* (2019) e *Ori and the Will of the Wisps* (2020), todos com avaliações majoritariamente positivas na plataforma de distribuição digital *Steam*.

Abaixo, algumas estatísticas sobre o faturamento desses títulos:

- *Ori and the Blind Forest* gerou lucro para a *Microsoft* (dona da publicadora do jogo, a *Xbox Game Studios*) uma semana após seu lançamento, e para a desenvolvedora *Moon Studios* em algumas semanas (MAKUCH, 2015);
- *Hollow Knight* havia vendido 2,8 milhões de cópias em fevereiro de 2019 (ARI, 2019);
- Em março de 2021, *Blasphemous* alcançou 1 milhão de jogadores (DEVORE, 2021);
- Lançado em março de 2020, *Ori and the Will of the Wisps* já havia sido jogado por 2 milhões de pessoas em junho (LIVINGSTON, 2020).

Os números apresentados confirmam a recepção positiva do público a novos lançamentos desse gênero nos dias atuais.

2.2. Geração procedural de conteúdo

A geração procedural de conteúdo, ou *procedural content generation* (*PCG*), segundo Hendrikx et al. (2013), é um tema muito explorado para o desenvolvimento de jogos e em diversos artigos, incluindo um *survey* compreensivo sobre as técnicas de *PCG* e os elementos aos quais ela se aplica. Hendrikx et al. categorizam os aspectos do desenvolvimento de jogos aos quais a *PCG* se aplica:

- *Game Bits*: textura, som, vegetação, construções, comportamento, e fogo, água, pedra e nuvens;
- *Game Space*: mapas interiores, mapas exteriores e corpos aquáticos;
- *Game Systems*: ecossistemas, redes rodoviárias, ambientes urbanos e comportamento de entidades;
- *Game Scenarios*: quebra-cabeças, *storyboards*, história e níveis;
- *Game Design*: *design* de sistemas e *design* de mundo;
- *Derived Content*: notícias e transmissões, e tabelas de classificação.

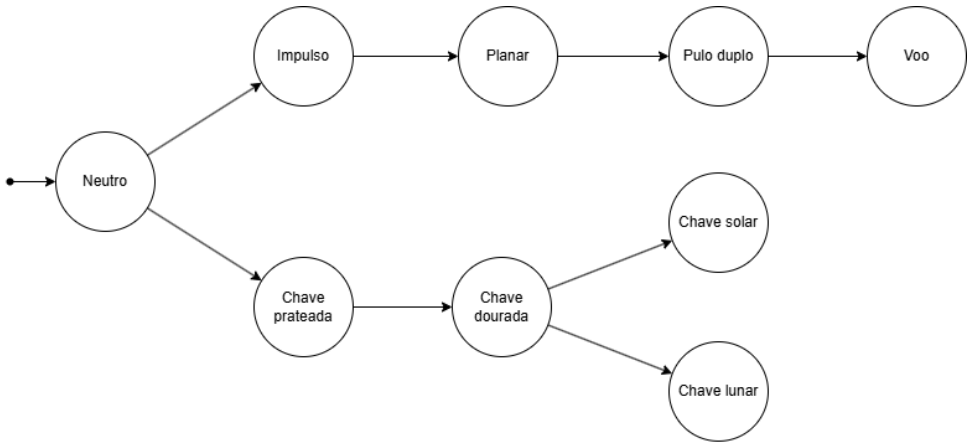
A *PCG* compreende técnicas e algoritmos diversos, tais como: ruído (*value noise*, *perlin noise*, *simplex noise*, *worley noise*), autômatos celulares e inteligência artificial. Essas técnicas são utilizadas e muitas vezes combinadas para gerar conteúdo como texturas e mapas automaticamente, de forma pseudoaleatória.

O algoritmo proposto no presente trabalho, bem como os algoritmos usados para a conclusão de cada passo do processo de geração, aplicam-se à categoria *Game Space*, que engloba os espaços no qual o jogador navegará durante o jogo.

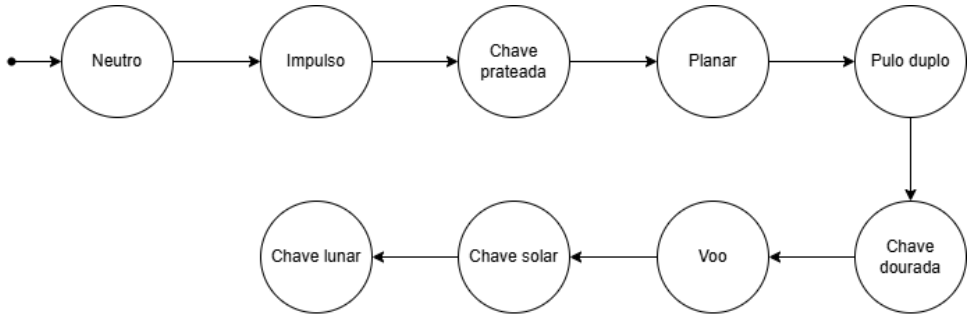
2.3. Geração procedural aplicada a mapas para *Metroidvanias*

Uma implementação demonstrando a geração procedural de mapas que contém as características básicas de um *Metroidvania* foi elaborada por Stalnaker (2020). Ela forma um

caminho de exploração através de salas pré-projetadas, cuja navegação ocorre por passagens trancadas por requisitos específicos chamados “chaves”. Esse algoritmo recebe algumas entradas incluindo nós inicial e final, dimensões do mapa e uma ordem de requisitos ou itens-chave que é transformada em um grafo acíclico dirigido, como exemplificado na Figura 2a, e então em uma ordem linear à qual o nível gerado deverá obedecer. Um possível resultado é representado pela Figura 2b.



(a) Ordem de requisitos ramificada. Fonte: o autor, adaptado de Stalnaker (2020).



(b) Ordem de requisitos linear. Fonte: o autor.

Figura 2. Ordem de requisitos ramificada e um possível resultado de sua transformação em uma ordem linear.

As entradas então são usadas para criar um grafo em forma de grade representando um nível em potencial. Para que seja possível transformá-lo em um nível jogável válido, esse grafo deve ser um grafo planar, ou seja, não deve haver interseções entre arestas. Devido à natureza aleatória da geração, é necessário verificar que o grafo representa um nível possível de ser completo. Um nível falha na validação (e, por consequência, é descartado) caso o nó final não esteja no grafo, ou seja, é impossível alcançar o fim, e caso a obtenção de um item-chave não cause um aumento no número de regiões exploráveis, o que é um sintoma da possibilidade de regiões serem exploráveis antes de seu requisito ser alcançado.

Após uma verificação de sucesso, os itens-chave são posicionados de modo que apenas um seja adquirível em dado momento. Um item-chave é sempre posicionado na área resultante da diferença entre a região atualmente explorável e a região explorável anteriormente à obtenção da chave mais recente.

Stalnaker menciona o fato de que esse processo poderia ser melhorado ao se utilizar um algoritmo mais inteligente, impondo restrições à aleatoriedade e eliminando a necessidade de verificação, por consequência diminuindo o desperdício de poder computacional decorrente da eventual necessidade de se descartar saídas inválidas.

Além disso, enquanto na maioria dos *Metroidvanias* são encontradas salas de diversos tamanhos, esse método apresenta a limitação de suportar apenas salas de tamanho homogêneo.

2.4. *Godot game engine*

Desenvolvido nos anos 2000 por Juan Linietsky e Ariel Manzur, o motor de jogos *Godot* foi usado internamente por empresas em muitos projetos antes de tornar-se *open source* em 2014 (LINIETSKY, 2019).

A *Godot* é desenvolvida principalmente em C++, porém não é necessário que os usuários utilizem essa linguagem: através de uma abordagem orientada a dados, a *engine* opera em arquivos de texto legíveis por humanos e amigáveis a ferramentas de versionamento, além de possuir uma linguagem de *scripting* própria (*GDScript*). Essas características eliminam a necessidade de que seus usuários recompilem o código de seus projetos, possibilitando um processo de desenvolvimento iterativo e ágil.

No entanto, não é excluída a possibilidade de criação de extensões (*GDExtension*) compiladas na forma de bibliotecas dinâmicas, integradas à *engine* através de uma interface binária. Essa aborgadem costuma ser reservada a casos em que a alta performance do código de máquina é necessária, ou quando se deseja trazer uma biblioteca nativa para a *Godot*.

Outra possibilidade, devido a sua licença *open source*, é a modificação e recompilação da *engine* inteira, geralmente motivada pela necessidade de se acessar alguma funcionalidade não exposta para o usuário tradicional.

O download padrão da *Godot* vem acompanhado de um editor cujo objetivo é apresentar uma interface gráfica intuitiva capaz de criar e manipular projetos e os arquivos (geralmente de texto) que os compõem. Um fato interessante é que o editor em si é também um "projeto" executado pela própria *Godot*, o que proporciona a vantagem de suportar a escrita de complementos (*plugins*) para o editor na linguagem *GDScript*, a mesma que o usuário empregará em seus projetos.

O editor também conta com um ambiente de desenvolvimento integrado (*IDE*) para a linguagem *GDScript*, com utilitários de formatação e completação de código, além de integração com a ferramenta de *debug* de projetos em execução.

Essas características e muitas outras, incluindo gerenciamento de entradas, carregamento de *assets* (imagens, sons, modelos 3D, etc.) e animação, fazem da *Godot* um ambiente completo de desenvolvimento de jogos e também uma ferramenta valiosa de rápida prototipação.

3. Trabalhos relacionados

Nesta seção serão analisados quatro trabalhos correlatos, mais especificamente na área de geração procedural. Estes trabalhos foram escolhidos como potencial material de re-

ferência na elaboração do presente trabalho e não necessariamente abrangendo o mesmo tema.

No trabalho de Stalnaker (2020), que teve por objetivo gerar proceduralmente mapas para *Metroidvanias*, grafos são utilizados para modelar a ordem dos requisitos e os mapas, garantindo que obedeçam às características do gênero. A falta de restrições durante a fase de geração resulta na necessidade de realizar uma verificação para garantir que os mapas gerados podem ser "vencidos". Além disso, o algoritmo produz uma disposição das salas com *layout* retangular e denso, muito diferente do que é observado em *Metroidvanias* existentes.

Gutierrez-Rodriguez et al. (2019) apresentam um modelo de geração procedural focado em *design* de níveis para *Metroidvanias* baseado nas preferências e experiências de *designers*. O modelo é aprimorado através de um processo no qual são utilizados dados gerados por inteligência artificial a partir de um conjunto inicial de dados. Devido à natureza iterativa do processo, há a possibilidade de intervenção por um *designer* para filtrar novas soluções. Foi concluído que, dado um banco de dados de experiências amplo e diverso, resultados aceitáveis são obtidos mesmo sem o *feedback* do *designer*.

O objetivo do trabalho de Baron (2017) foi analisar e implementar algoritmos de geração de *dungeons*, algo comumente empregado em jogos do gênero *Roguelike*, sendo uma de suas principais características a geração procedural de mapas a cada nova jogatina. Os algoritmos alvo do estudo são divididos em duas catergorias: posicionamento de salas e posicionamento de corredores. Os da primeira categoria são *Random Room Placement* e *BSP Room Placement*; os da última, *Random Point Connect*, *Drunkard's Walk* e *BSP Corridors*, sendo este último aplicável somente a casos em que *BSP Room Placement* é usado, devido à necessidade de acessar as estruturas de dados intermediárias desse método. Por fim, o código foi aplicado a um projeto 3D na *engine* de jogos *Unreal Engine 4*.

O quarto trabalho, de Lin et al. (2020), tem como área o campo do *design* urbano, e utiliza Redes Neurais Convolucionais (*CNNs*) e *Wave Function Collapse* (*WFC*) para realizar síntese de espaços urbanos. Primeiro, foi estabelecido um banco de dados de *design* urbano que inclui três conjuntos de dados representados por mapas 2D: redes rodoviárias, formas espaciais de quadra e *layout* de função de construção. O banco de dados de redes rodoviárias é a fonte de dados para a síntese pela *WFC*, enquanto as formas espaciais de quadra são extraídas para síntese pela *CNN*. Por fim, o *layout* de função de construção é extraído para gerar múltiplos *layouts* de função de construção. Cada um dos passos é avaliado por um *designer*. Modelos 3D são gerados a partir das diferentes cores que representam alturas distintas na forma espacial de quadra 2D gerada. Os resultados do trabalho demonstram a viabilidade de *WFC* e *CNNs* para rápida prototipagem de *design* urbano.

O Quadro 1 faz um comparativo entre os algoritmos. O primeiro trabalho possui maior similaridade com este trabalho, tanto em objetivo quanto em abordagem, enquanto o segundo compartilha apenas o mesmo objetivo e utiliza técnicas muito diferentes. O terceiro trabalho tem como alvo um gênero diferente, porém os algoritmos demonstrados apresentam propriedades desejáveis. O quarto trabalho demonstra o uso do *Wave Function Collapse*, inicialmente cogitado como solução para o presente trabalho.

Trabalho	Gênero ou área de aplicação	Algoritmo(s) e estrutura(s) de dados
(STALNAKER, 2020)	*Metroidvania*	Grafos dirigidos, validação
(GUTIERREZ-RODRIGUEZ et al., 2019)	*Metroidvania*	Algoritmos evolutivos, redes neurais artificiais
(BARON, 2017)	Geração de *dungeons*	*Random Room Placement*, *BSP Room Placement*, *Random Point Connect*, *Drunkard's Walk*, *BSP Corridors*
(LIN et al., 2020)	*Design* urbano	*Wave Function Collapse*, Redes Neurais Convolucionais
Presente trabalho	*Metroidvania*	Tentativa e erro, grafos dirigidos

Quadro 1. Comparação dos trabalhos relacionados.

O presente trabalho seguiu a mesma linha de investigação do trabalho de Stalnaker em um esforço de aproximação em direção ao *level design* de *Metroidvanias* tradicionais.

4. Metodologia

O cerne do método proposto encontra-se nesta seção, iniciando com a definição de vocábulos utilizados ao longo da obra, seguida da descrição do funcionamento do algoritmo, e finalizando com o relato de implementação e dos artefatos produzidos.

4.1. Definições e Estrutura

Serão apresentados nesta seção termos importantes para a compreensão do presente trabalho, acompanhados de suas definições no contexto da obra. Essa terminologia foi elaborada especificamente para este trabalho, baseando-se em conceitos encontrados na literatura ou familiares a conhecedores do gênero, mas sem o objetivo de encaixar-se em quaisquer possíveis categorizações existentes.

4.1.1. Conceitos básicos do mundo do jogo

O jogador é representado por um personagem que navega um ambiente composto por pisos, plataformas, paredes, escadas e obstáculos, entre outros elementos. Esses ambientes navegáveis são chamados **"salas"**. Os elementos descritos anteriormente são a representação interna de uma sala; além disso, salas ocupam espaço no mundo do jogo conforme descrito por sua representação externa.

O **conteúdo da sala** é a representação interna de uma sala. Contém os objetos com os quais o personagem jogável pode interagir e os espaços pelos quais pode navegar. Na Figura 3, os retângulos internos, bem como todas as ilustrações dentro deles (figura humanoide representando o personagem jogável, plataforma, portas, escadas, espinhos e item colecionável em forma de estrela), são considerados conteúdo de salas.

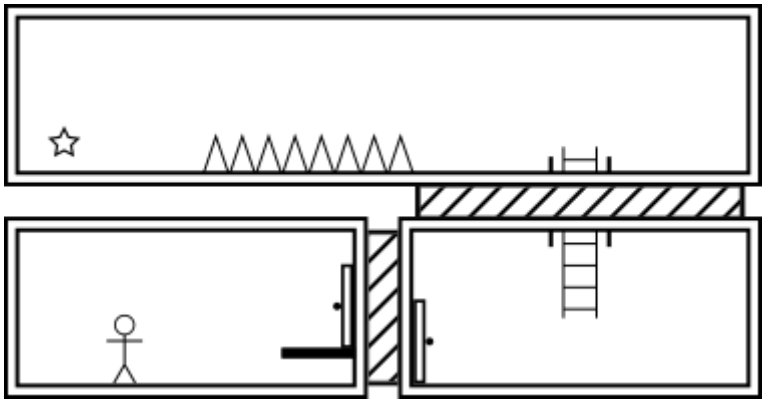


Figura 3. Esquematização da sobreposição entre as representações interna e externa de salas. Fonte: o autor.

A representação externa de uma sala é o **perfil de sala**. Ele delimita o espaço que a sala ocupa no mundo (seu tamanho) e as arestas que a vinculam a outras salas. Corresponde ao que é visto pelo jogador ao abrir o mapa do mundo em *Metroidvanias* que oferecem essa funcionalidade. Na Figura 3, são representadas pelos retângulos externos.

As **arestas** são passagens em uma sala pela qual o jogador pode transitar para fora da sala atual e para dentro de outra. Suas informações são descritas no perfil de sala por uma posição na sala (em unidades-tela) e pelo lado para o qual o personagem jogável as atravessa (norte, sul, leste, oeste). Além disso, informam uma gama de chaves potenciais correspondentes a variações de conteúdo. Cada variação exige que o jogador possua a chave a ela associada para que possa transitar pela aresta e alcançar a sala conectada. Apenas uma variação de cada aresta será apresentada ao jogador por instância da sala. As áreas preenchidas por linhas diagonais na Figura 3 ilustram pares de arestas conectando duas salas. Note a correspondência entre essas arestas externas e elementos internos do conteúdo das salas, como as portas e as escadas na figura.

Em *Metroidvanias* como *Castlevania: Aria of Sorrow*, é possível observar no mapa-múndi salas de diversos tamanhos (vide Figura 4). O conteúdo das salas de menor tamanho ocupa exatamente a tela inteira do dispositivo que executa o jogo. A esse tamanho, daremos o nome de **"unidade-tela"**. Todas as salas são compostas por um número inteiro de unidades-tela.

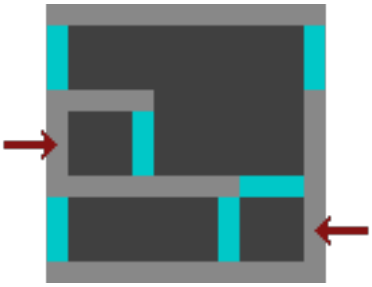


Figura 4. Trecho do mapa-múndi de *Castlevania: Aria of Sorrow*. As salas apontadas por setas possuem o tamanho igual a uma unidade-tela. Fonte: o autor.

4.1.2. Elementos de progressão

A progressão em um *Metroidvania* ocorre pela aquisição de **chaves**, que não necessariamente assumem a forma de chaves de fechadura. Habilidades especiais que o personagem jogável pode aprender são um dos tipos mais comuns de chave. O jogador adquire uma chave ao visitar determinado trecho do mundo e nele completar alguma ação, como interagir com um objeto, derrotar um inimigo chefe ou resolver um enigma. Nesse sentido, o ato de "adquirir" assemelha-se mais a assinalar uma caixa em uma lista de tarefas do que apossar-se um objeto.

O **conjunto de chaves** é o conjunto que contém todas as chaves planejadas para existir no jogo.

A **ordenação de chaves** consiste de um grafo acíclico dirigido no qual cada nó representa uma chave do conjunto de chaves e uma aresta que parte da chave A e aponta para a chave B indica que A deve necessariamente ser adquirida pelo jogador antes de B. Uma chave deve ser designada como **"chave inicial"**. Nenhuma aresta deve apontar para o nó relacionado à chave inicial.

4.1.3. Estrutura e Instanciação

O mundo do jogo consiste em uma coleção de salas. Elas encaixam-se como um quebra-cabeças, com cada aresta conectada a exatamente uma aresta (não podendo conectar-se a si mesma), formando um **mapa-múndi** coeso, onde salas não sobrepõem-se umas às outras. Minoritariamente, podemos encontrar em certos *Metroidvanias* salas que localizam-se em isolamento, isto é, suas arestas não "tocam" aquelas às quais se conectam, fazendo com que transições por elas pareçam teletransportar o jogador pelo mapa-múndi. Esse tipo de ocorrência é desconsiderado pelo trabalho atual.

Quando o conteúdo de uma sala está manifestado no jogo de modo que o usuário consegue vê-la e navegar por ela, dizemos que ela está **instanciada**. Comumente, apenas uma sala está instanciada a qualquer momento do jogo. Quando o jogador transita para outra sala, essa é instanciada e a anterior é destruída.

4.2. Arquitetura do Algoritmo

A operação do algoritmo implementado por este trabalho segue o modelo em lote: o usuário prepara as entradas antes de executar o código, que realiza todo o processamento sem interação do usuário até produzir uma saída completa ou falhar.

4.2.1. Descrição geral do fluxo de geração

O processo de geração pode ser resumido como a inserção de várias salas na forma de regiões. Cada região corresponde a uma chave na ordenação. Uma sala só pode ser acessada pelo jogador após a aquisição da chave correspondente à sua região.

O primeiro passo é posicionar a sala inicial, cujo perfil pode ser escolhido aleatoriamente ou especificado pelo usuário. A partir das arestas da sala inicial, é construída

uma lista de arestas abertas que podem ser usadas como referência para posicionar novas salas.

A chave inicial da ordenação é atribuída à sala inicial e definida como chave da região atual. Após o término da inserção de salas em uma região, todas as chaves imediatamente posteriores à chave da região atual são adicionadas uma lista de chaves potenciais, e uma delas é escolhida aleatoriamente para a próxima região e removida da lista; aquelas que não forem selecionadas permanecem na lista e podem ser sorteadas para as próximas regiões. O Algoritmo 1 fornece uma visão geral dessa etapa.

Algoritmo 1 Nível mais alto da geração.

```
1:  Input: chave_inicial, ordenacao
2:  Escolher e posicionar sala com região chave_inicial                ▷ sala inicial
3:  proximas_chaves ← {chave_inicial}
4:  while not vazia(proximas_chaves) do
5:      chave_atual ← elemento aleatório removido de proximas_chaves
6:      Escolher e posicionar sala com região chave_atual      ▷ primeira sala da região
7:      for 1 to tamanho_regiao do
8:          Escolher e posicionar sala com região chave_atual
9:      end for
10:     p ← ordenacao[chave_atual]          ▷ obter chaves imediatamente posteriores
11:     Concatenar p à lista proximas_chaves
12: end while
```

Uma consequência desse processo sequencial é a linearização ou "achatamento" da ordenação de chaves. A existência de uma região para cada chave na ordenação é garantida; o caso contrário é considerado uma falha de geração.

O posicionamento de salas acessa uma das entradas fornecidas pelo usuário: uma coleção de perfis de sala. Ele ocorre da seguinte forma:

1. uma aresta A na lista de arestas abertas é escolhida aleatoriamente;
2. um perfil de sala é escolhido aleatoriamente a partir da coleção de perfis;
3. uma aresta B nesse perfil é escolhida aleatoriamente;
4. é realizada uma tentativa de de posicionar uma instância do perfil de sala selecionado de modo que A e B encaixem-se, obedecendo às restrições descritas na Sessão 4.2.3.

A aresta B, o perfil de sala e a aresta A serão respectivamente ressorteados quando o posicionamento falhar. A geração como um todo terá falhado quando isso ocorrer para todas as arestas A na lista de arestas abertas.

O número de salas que devem posicionadas em cada região é definido pelo usuário. Após o posicionamento da primeira sala em uma região, a lista de arestas abertas é esvaziada antes de receber as arestas abertas dessa sala, o que garante o agrupamento das salas da região.

4.2.2. Estruturas de dados e representação do mapa

O mapa-múndi é representado por uma lista de salas. Cada sala é uma instância de um perfil sala, e mais de uma sala pode referenciar o mesmo perfil. Arestas e perfis de aresta possuem um relacionamento similar, além de pertencerem a suas contrapartes. Essas estruturas e relações encontram-se diagramadas na Figura 5.

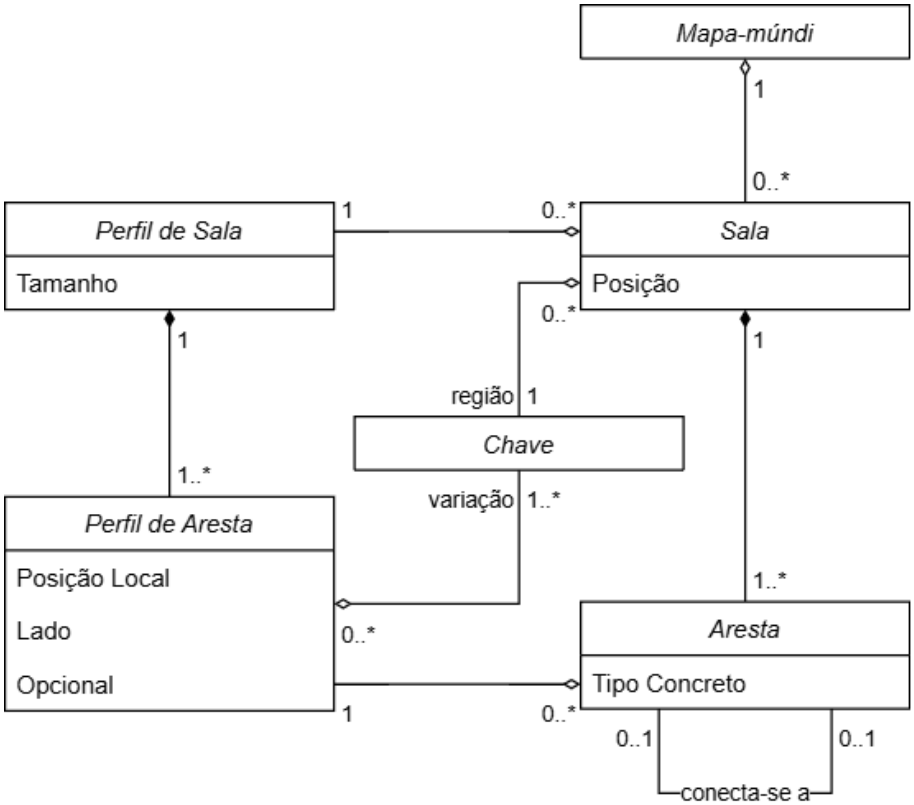


Figura 5. Diagrama de classes do conjunto de estruturas de dados necessárias para representar um mapa gerado. Fonte: o autor.

Considerando que o algoritmo realiza muitas consultas espaciais no mapa-múndi (obtenção de salas a arestas instanciadas em determinada unidade-tela), é possível utilizar uma estrutura de dados otimizada para esse caso de uso. O presente trabalho optou por uma lista dinâmica simples devido à pronta disponibilidade.

4.2.3. Restrições e critérios de validação

Ao realizar uma tentativa de posicionar uma sala no mapa-múndi, é necessário garantir que ela não sobreponha-se a outras salas já existentes, não torne nenhuma aresta insolúvel e não quebre a progressão ditada pela ordenação de chaves.

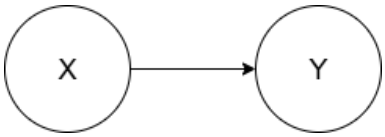
O teste de sobreposição é um teste trivial de intersecção de retângulos entre a sala que será posicionada e todas as salas já presentes no mapa-múndi. Em caso positivo (há intersecção), a tentativa é rejeitada.

Após esse passo, é necessária uma avaliação mais complexa no perímetro da nova

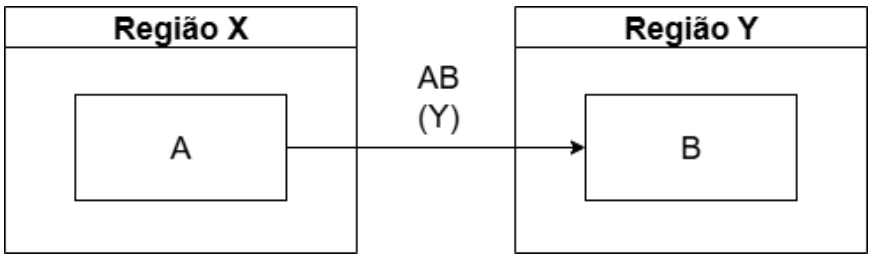
sala para garantir que toda aresta não opcional seja solúvel, isto é, faça frente a outra aresta ou a um espaço vazio onde uma sala compatível pode ser posicionada futuramente. Esse teste também deve ser realizado em salas ao redor, em toda aresta que esteja frente a frente com a nova sala.

Arestas que estejam frente a frente são consideradas conectadas. Essa informação é armazenada para que uma etapa posterior possa implementar a transição entre elas (tal etapa está fora do escopo deste trabalho). Também é nesse momento que, para arestas conectadas, é escolhida a variação que cada aresta apresentará. Dentre todas as variações suportadas pelo perfil, são excluídas dessa escolha aquelas cuja chave ainda não foi selecionada para nenhuma região, pois nesse ponto o jogador ainda não as terá obtido.

Além disso, quando duas arestas conectam diferentes regiões, uma validação especial precisa ocorrer. Devido à ordenação final de chaves ser linear, essas regiões distintas classificam-se em anterior e posterior. Uma aresta que transita para uma região posterior deve obrigatoriamente usar uma variação correspondente à chave dessa região, conforme ilustrado na Figura 6b: a sala A, pertencente à região trancada pela chave X, possui uma aresta AB que transita para a sala B, na região trancada por Y. Considerando a ordenação proposta (Figura 6a) que demanda que Y seja posterior a X, a transição por AB só será permitida ao possuir a chave Y; isso garante a principal propriedade desejável na progressão de um *Metroidvania*. Se a aresta não possuir uma variação que satisfaça a essa condição, o posicionamento é rejeitado.



(a) Exemplo de ordenação: chave X anterior à chave Y.



(b) Representação de trecho de mapa com salas contidas em regiões.

Figura 6. Exemplo de imposição de requisito. Fonte: o autor.

Após a inserção bem-sucedida do número almejado de salas em todas as regiões, a tarefa final é o fechamento de todas as arestas que permaneceram desconectadas. Porém, como apenas arestas opcionais podem ser fechadas, é necessário um mecanismo para prevenir que arestas não opcionais permaneçam abertas. Esse mecanismo consiste em seguir adicionando novas salas a essas arestas até que nenhuma aresta não opcional esteja em aberto.

A solução escolhida utiliza uma coleção especial de perfis de sala, na qual o usuário deve adicionar apenas salas que apresentem grandes chances de sucesso para essa etapa. A sala ideal tem dimensões 1×1 e uma aresta opcional em cada direção.

4.3. Implementação

Nesta subseção serão relatados os passos seguidos durante a investigação, o ambiente de desenvolvimento e os artefatos resultantes, que culminaram na produção de um *plugin* utilizável por desenvolvedores de jogos que trabalhem com a *Godot*.

4.3.1. Ferramentas e linguagem

O ambiente em que o trabalho foi implementado é a *Godot Game Engine* versão 4.4. Tanto o algoritmo quanto as ferramentas auxiliares detalhadas na Seção 4.3.3 foram escritos na linguagem *GDScript*.

As principais razões dessa escolha foram a praticidade de uso das funcionalidades gráficas pré-existentes, as ferramentas de depuração embutidas e a possibilidade de extensão do editor – essa sendo especialmente favorável, pois um motor de jogos é a fundação ideal para integrar uma ferramenta de apoio a desenvolvedores de jogos.

4.3.2. Processo de implementação

A hipótese inicial consistia em um algoritmo que iria posicionar uma sala inicial e, através de tentativa e erro, posicionar novas salas a partir de arestas "abertas". As arestas das novas salas seriam adicionadas à lista de arestas abertas e o processo se repetiria.

Essa primeira versão considerava apenas restrições espaciais, como rejeitar o posicionamento de salas em sobreposição a outras e exigir que arestas sempre se conectassem a outra aresta ou a espaços livres; não levava em consideração chaves potenciais ou opcionalidade das arestas.

Uma vez verificado que a abordagem produzia layouts com o formato desejado, o próximo passo foi incorporar chaves e suas ordenações e, finalmente, o fechamento das arestas abertas ao fim da geração.

O projeto contendo a implementação está disponível em um repositório *Git* no seguinte endereço: ⟨https://github.com/hexagon-0/metroidvania-map-generator⟩.

4.3.3. Mecanismos de visualização, edição e depuração

Inicialmente, os perfis de salas eram construídos através do editor de propriedades padrão da *Godot*, denominado "inspetor", que exibe apenas controles simples como campos numéricos e de texto e caixas de marcação. Um objeto 2D auxiliar foi criado para servir como representação gráfica dos perfis de sala e refletir os ajustes nas propriedades.

Isso rapidamente mostrou-se tedioso e sujeito a erros, incitando o desenvolvimento de um editor visual personalizado (Figura 7). Esse editor, integrado ao editor da *Godot* através de sua interface de *plugins*, elimina a necessidade do objeto 2D e também permite a seleção de arestas para edição diretamente através de interação com a representação visual, além de impedir configurações ilegais de arestas.

Para facilitar a depuração do algoritmo, um mecanismo de exibição passo-a-passo

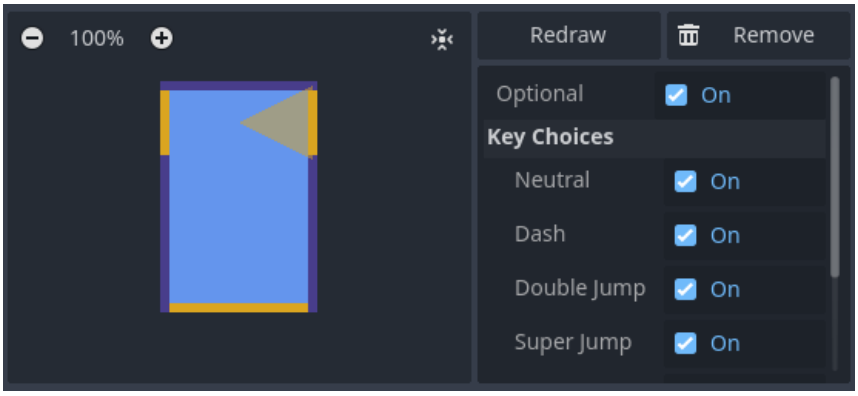


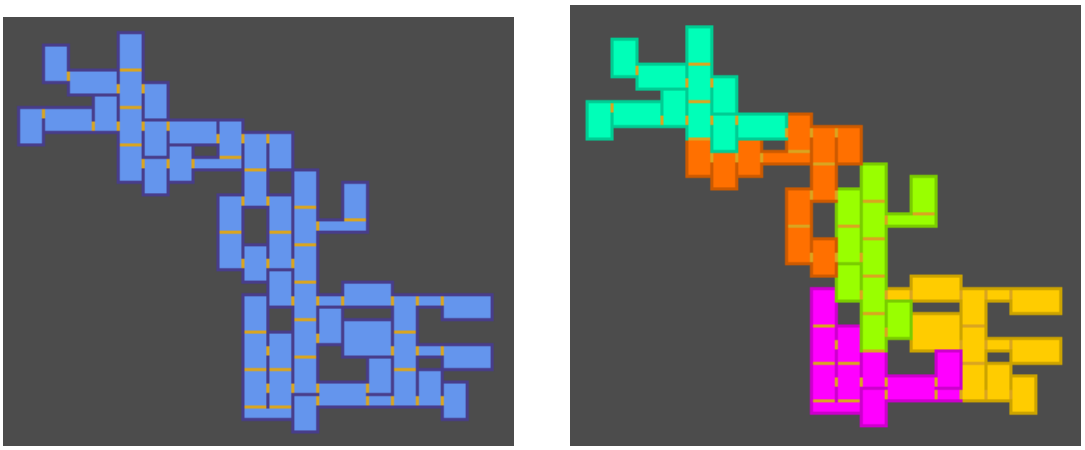
Figura 7. Captura de tela do editor de perfil de sala. Fonte: o autor.

foi inserido em meio ao processo de geração. Isso permite que o usuário avalie as tentativas realizadas e fornece um novo meio de detecção humana de *bugs*, pois a partir disso é possível testemunhar aceites e rejeições que desobedecem a lógica pretendida.

5. Resultados e Discussão

Será realizada nessa seção uma apresentação dos resultados obtidos, seguida de uma análise e comparação com um trabalho relacionado.

Os mapas-múndi resultantes do processo de geração, demonstrados pela Figura 8, obedecem a todas as restrições estipuladas. Não foram constatadas sobreposições, conexões inválidas (toda aresta encontra-se com outra aresta que realiza a transição oposta) ou "furos" na progressão que permitam o avanço do jogador para uma região posterior sem adquirir a chave apropriada.



(a) Mapa-múndi resultado da geração. **(b) Mapa-múndi com regiões colorizadas.**

Figura 8. Exemplo de mapa-múndi gerado pelo algoritmo, com arestas em amarelo. Fonte: o autor.

A chance de o algoritmo completar sua execução e produzir um mapa finalizado é proporcional ao número de opções que lhe são dadas a cada passo. Isso está diretamente relacionado à "qualidade" das entradas, principalmente os perfis de sala: quanto maior a variedade de perfis, a quantidade de variações suportadas por cada perfil e a quantidade de arestas que são opcionais, mais alternativas o algoritmo terá.

Além disso, nota-se que o maior risco de falha total de geração existe logo após o início de cada nova região. Isso ocorre por consequência do esvaziamento da lista de arestas abertas somado à alta chance de sobreposição com outras salas.

Os testes listados no Quadro 2 revelam que um aumento do número de salas por região diminui a taxa de sucesso: cerca de 60.3% para 10 salas, enquanto 50 salas reduzem isso a 33.2% (note que esse parâmetro determina a quantidade de salas além da sala inicial e da primeira sala de cada região, que são obrigatórias).

Os tamanhos de região empregados nos testes foram escolhidos para compreender uma gama significativa dos valores que, através da observação de jogos do gênero, estimou-se encontrar em um jogo real, localizados na casa das dezenas. A quantidade de chaves usada (cinco) é o número intermediário presente nos testes de Stalnaker (2020).

Tam. região	Sucessos	% sucesso	Tempo médio (µs)
10	603	60.3%	17714
25	471	47.1%	62843
50	332	33.2%	187388
75	328	32.7%	390578

Quadro 2. Taxa de sucesso e tempo médio por geração bem-sucedida para tamanhos de região 10, 25, 50 e 75. Foram realizadas 1000 tentativas para cada, com 5 chaves/regiões.

O trabalho de Stalnaker (2020) realiza repetidas tentativas de geração até que um mapa válido seja produzido. A seção de análise lista o número médio de mapas potenciais gerados e tempo médio de geração, categorizados por número de chaves e tamanho do mundo. Para realizar uma comparação, uma abordagem similar foi adotada. O Quadro 3 lista os resultados de ambos. Note uma diferença fundamental entre os dois trabalhos: o de Stalnaker (2020) possui um mapa-múndi de tamanho delimitado, diferente do presente trabalho, o que dificulta a determinação de quais parametrizações devem ser comparadas.

Por fim, o número de salas geradas foi eleito como critério de comparação. Isso tornou necessária a aproximação desse número para o trabalho de Stalnaker (2020), uma vez que o tamanho do mundo, o único parâmetro relacionado fornecido, é apenas um limite superior, pois porções da treliça que representa o mundo podem estar vazias. Assumiremos que toda célula será preenchida com uma sala, isto é, o número de salas geradas será igual a esse limite superior dado por $largura \times altura$. Para o presente trabalho, o total de salas emitido por uma geração bem-sucedida será, no mínimo, $k \times (1 + r) + 1$, onde r é o tamanho da região e k é o número de chaves na ordenação, caso nenhuma sala seja adicionada durante o fechamento.

Como visto no Quadro 3, a quantidade de salas relacionada ao tamanho de região 10, o menor testado pela presente análise, aproxima-se daquela produzida pelo maior tamanho de mundo testado por Stalnaker (2020), 8×8. A média de tentativas aqui obtida é uma ordem de magnitude menor, enquanto o tempo médio é cerca de um terço do reportado pelo outro trabalho.

É importante ressaltar que a comparação através do número de salas geradas é meramente quantitativa e não qualitativa, pois uma extensão exagerada de área opcional

Trabalho	Parâmetros	Salas est.	Média tentativas	Tempo médio (µs)
Presente trabalho	10	56	1.669	23770.859
	25	131	2.115	98213.632
	50	256	2.818	330726.208
	75	381	3.182	706771.938
Stalnaker (2020)	3×3	9	123.84	35701.78
	4×4	16	24.38	15109.587
	5×5	25	18.37	21231.167
	6×6	36	13.1	24761.55
	7×7	49	16.35	44138.813
	8×8	64	12.7	65249.419

Quadro 3. Comparação com os resultados de Stalnaker (2020). Apenas os números obtidos para 5 chaves foram listados. A coluna "Parâmetros" é distinta para cada trabalho.

ou obrigatoriamente explorável pelo jogador pode ser altamente indesejável, sob o risco de tornar a experiência de jogo entediante.

Ultimamente, podemos concluir que confinar o algoritmo de geração a um espaço limitado incorre um aumento de esforço computacional. Além disso, é considerada a hipótese de que o mecanismo de variações de arestas flexibiliza o conjunto de regras que o processo deve respeitar, facilitando a produção de mapas válidos. Ao passo que essa flexibilidade não existe sem certo custo para o *designer*, estima-se que, durante a produção de perfis de sala, a adoção de uma diretriz que dita que cada aresta deve incluir, no mínimo, a chave inicial entre suas variações será suficiente para a maioria dos casos, pois essa chave torna-se inaceitável apenas nas fronteiras entre regiões.

Um obstáculo para a escalabilidade do algoritmo é a utilização de uma lista dinâmica para armazenar as salas, o que torna o tempo de consulta proporcional ao número de salas posicionadas. Uma estrutura de dados especializada poderia eliminar essa correlação.

6. Trabalhos Futuros

Esta seção enumera pontos do trabalho que poderão ser desenvolvidos em obras futuras.

O próximo passo natural é a implementação da instanciação do conteúdo das salas. O algoritmo já providencia todas as informações necessárias para selecionar as variações corretas de cada aresta e seus destinos. A criação das mecânicas do personagem jogável e das chaves está fora do escopo do algoritmo de propósito geral, pois são específicas para cada jogo.

A mitigação de pontos de falha é outro ponto essencial. Conforme descrito anteriormente, uma grande fonte de falhas completas na geração é causada pela modificação de estado ao iniciar o posicionamento de uma nova região – em especial a limpeza da lista de arestas abertas. Tornar essas modificações reversíveis converteria esse tipo de falha completa em falha local: seria possível descartar somente a região incompleta e escolher outra aresta da região anterior para tentar novamente.

Para ampliar o horizonte de tarefas de *design* automatizadas, métricas como

distância entre cada sala e a entrada da região em que se encontra podem ser usadas como critério para o posicionamento de salas especiais, como ponto de salvamento, ponto de viagem rápida, inimigo chefe e local de obtenção de chave, elementos que precisam ser posicionados estrategicamente.

Também é desejável permitir que o usuário especifique o perfil da sala final, onde o jogo se encerrará, e sua posição no mapa-múndi. Para isso, será necessário direcionar a geração de salas para que conecte a sala final às outras.

Existem inúmeros algoritmos de *pathfinding* que poderiam ser usados para traçar uma rota até a sala final, porém esses algoritmos costumam procurar pelo menor trajeto possível, enquanto o progresso do jogador em um *Metroidvania* raramente é expresso por um caminho direto do início ao fim do jogo. Uma possível solução seria realizar a geração do mapa normalmente e, ao fim dela e caso a sala final esteja desconectada das demais, aplicar a técnica de *pathfinding* escolhida.

Pode-se cogitar ainda possibilitar a definição de salas não retangulares. Apesar de não haver restrição alguma que impeça o *designer* de atribuir a uma sala um conteúdo não retangular, toda sala ocupa um espaço retangular no mapa-múndi devido ao formato especificado no perfil ser representado apenas pelos dois componentes escalares "altura" e "largura".

Para que salas externamente não retangulares sejam possíveis, será necessário elaborar uma nova representação para o seu formato (polígonos, coleção de retângulos, etc.) e identificar e modificar todas as operações do algoritmo que atualmente presumem o formato retangular, como a análise de sobreposição.

Atualmente, o algoritmo parte do princípio de que cada aresta de uma sala pode ser alcançada a partir de qualquer uma de suas outras arestas. Isso impossibilita a existência de salas que possuam "túneis" ou seções internas não interconectadas. Essas formas podem ser úteis para compactar passagens entre áreas do mundo, e talvez seja desejável incorporar à ferramenta a capacidade de processá-las corretamente.

Como é usual em técnicas procedurais, seleções aleatórias são feitas em diversos pontos do processo de geração. Um exemplo é a escolha do perfil de sala que será posicionado a seguir durante o posicionamento de salas por região. Exercer algum controle sobre essa aleatoriedade pode ajudar na obtenção de resultados melhores. Nesse caso, seria possível adicionar um peso ao perfil de sala (diretamente nas propriedades do perfil ou através da coleção de perfis), a fim de controlar a frequência com que cada perfil será apresentado.

Por fim, há vias para explorar a melhoria do processo de finalização. O fechamento de arestas restantes após a geração atualmente depende de uma coleção de perfis de sala separada daquela utilizada para o posicionamento de salas, na qual devem estar enumerados perfis de sala criados especialmente para cooperar com essa etapa. Idealmente, essa tarefa deveria ser eliminada, pois onera o usuário com mais trabalho de criação e amplia a superfície de erros caso as características dos perfis na coleção especial desviem-se do ideal.

O método mais amigável ao usuário final do algoritmo seria escolher um perfil de sala da mesma coleção usada anteriormente para gerar o mapa. Porém, para garantir

maiores chances de sucesso e mitigar o risco de que o algoritmo continue adicionando salas indefinidamente sem nunca poder fechar todas as arestas, seria imperativo o emprego de uma heurística para selecionar o melhor perfil de sala: por exemplo, salas com menor tamanho e menor número de arestas não opcionais (idealmente zero) são preferíveis.

7. Conclusão

O presente trabalho obteve êxito em seu objetivo de propor e implementar um método para a geração automatizada de níveis para jogos do gênero *Metroidvania* organizando modelos de conteúdo pré-desenhado de forma procedural. O algoritmo desenvolvido mostrou-se capaz de gerar mapas completos, compostos por salas interconectadas, respeitando critérios como:

- ausência de sobreposições espaciais;
- manutenção da progressão ditada pela ordenação de chaves;
- garantia de conectividade entre regiões distintas;
- utilização de *layouts* compatíveis com a navegação típica de *Metroidvanias*.

Além disso, verificou-se que a implementação, ainda que prototípica, sustenta tempos de execução compatíveis com o uso em ferramentas de apoio ao *design*, possibilitando a visualização rápida de mapas válidos e oferecendo uma base funcional para um sistema mais abrangente de autoria procedural.

O trabalho não implementou a instanciação completa do conteúdo interno das salas e a navegabilidade, restringindo-se a relacionamentos entre salas e suas propriedades externas, como posicionamento no mundo e conexões entre arestas.

Em síntese, o trabalho atingiu seu objetivo principal ao demonstrar a viabilidade de um método procedural para construção de mapas de *Metroidvanias*, mas ainda apresenta oportunidades significativas de aprimoramento. Desenvolvimentos futuros, especialmente com a inclusão de ferramentas de edição mais maduras, refinação do processo de fechamento de arestas e ampliação da análise experimental, podem ampliar sua aplicabilidade prática e seu potencial de adoção na indústria.

Referências

ARI. Hollow knight: Silksong revealed! *Team Cherry Blog*, 2019. Disponível em: ⟨https://web.archive.org/web/20190310231829/http://teamcherry.com.au/hollow-knight-silksong/⟩. Acesso em: 11 de janeiro de 2023.

BARON, J. R. Procedural dungeon generation analysis and adaptation. In: *Proceedings of the SouthEast Conference*. [S.l.: s.n.], 2017. p. 168–171.

DEVORE, J. Over a million people played blasphemous. *Destructoid*, 2021. Disponível em: ⟨https://www.destructoid.com/over-a-million-people-played-blasphemous/⟩. Acesso em: 11 de janeiro de 2023.

GUTIERREZ-RODRIGUEZ, A.; COTTA, C.; FERNÁNDEZ-LEIVA, A. J. Deep evolutionary training of a videogame designer. *EVO* 2019*, p. 6–7, 2019.

HENDRIKX, M. et al. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, ACM New York, NY, USA, v. 9, n. 1, p. 1–22, 2013.

LIN, B.; JABI, W.; DIAO, R. Urban space simulation based on wave function collapse and convolutional neural network. In: *Proceedings of the 11th Annual Symposium on Simulation for Architecture and Urban Design*. [S.l.]: Society for Computer Simulation International, 2020.

LINIETSKY, J. A decade in retrospective and future. *Godot Engine News*, 2019. Disponível em: ⟨https://godotengine.org/article/retrospective-and-future⟩. Acesso em: 9 de janeiro de 2023.

LIVINGSTON, C. 2 million people have played ori and the will of the wisps since march. *PC Gamer*, 2020. Disponível em: ⟨https://www.pcgamer.com/2-million-people-have-played-ori-and-the-will-of-the-wisps-since-march/⟩. Acesso em: 11 de janeiro de 2023.

MAKUCH, E. Celebrated xbox one game ori and the blind forest profitable in one week. *GameSpot*, 2015. Disponível em: ⟨https://www.gamespot.com/articles/celebrated-xbox-one-game-ori-and-the-blind-forest-/1100-6426518/⟩. Acesso em: 11 de janeiro de 2023.

PRADO, C. S. do; LAZARINI, J. P. P. R.; FÁVARO, A. L. O. Análise dos princípios de desenvolvimento de jogos metroidvania. In: *SBGames 2020*. [S.l.: s.n.], 2020.

STALNAKER, T. W. *Procedural generation of metroidvania style levels (thesis)*. 2020.