UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Pedro Probst Minini

EXPLORAÇÃO DE TÉCNICAS ATUAIS DE PROGRAMAÇÃO DE JOGOS E GERAÇÃO PROCEDURAL DE AMBIENTES ATRAVÉS DO DESENVOLVIMENTO DE UM PROTÓTIPO DE JOGO EM RUST

Santa Maria, RS
2021

Pedro Probst Minini

EXPLORAÇÃO DE TÉCNICAS ATUAIS DE PROGRAMAÇÃO DE JOGOS E GERAÇÃO PROCEDURAL DE AMBIENTES ATRAVÉS DO DESENVOLVIMENTO DE UM PROTÓTIPO DE JOGO EM RUST

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**. Defesa realizada por videoconferência.

ORIENTADOR: Prof. Joaquim Vinicius Carvalho Assunção

489
Santa Maria, RS
2021

End. Eletr.: pprobst@insiberia.net

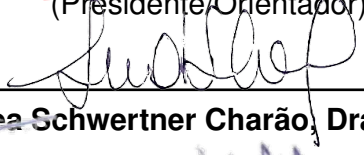
Pedro Probst Minini

EXPLORAÇÃO DE TÉCNICAS ATUAIS DE PROGRAMAÇÃO DE JOGOS E GERAÇÃO PROCEDURAL DE AMBIENTES ATRAVÉS DO DESENVOLVIMENTO DE UM PROTÓTIPO DE JOGO EM RUST

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação**.

Aprovado em 8 de fevereiro de 2021:

Joaquim Vinicius Carvalho Assunção, Dr. (UFSM)
(Presidente/Orientador)



Andrea Schwertner Charão, Dra. (UFSM)

Sergio Luis Sardi Mergen, Dr. (UFSM)

Santa Maria, RS
2021

AGRADECIMENTOS

Seria no mínimo estranho caso eu não agradecesse – em primeiro lugar – ao meu avô Edgar pelos muitos anos de auxílio e encorajamento aos meus estudos. Em seus quase 90 anos, ele continua com uma acuidade mental acima da média ao compará-lo com seus pares. Quando a sua última hora chegar, direi com orgulho que aproveitei cada momento de sua companhia. Entre seus ensinamentos (ou resquícios genéticos), tento seguir rigorosamente: 1) sempre cumprir com suas obrigações o quanto antes, 2) sempre pagar à vista e 3) sempre sair com antecedência.

Durante a minha graduação, tive um bom número de professores memoráveis – felizmente, a grande maioria dos professores da informática são ótimos. No entanto, dois merecem um destaque especial: meu orientador Joaquim Assunção e a professora Andrea Charão. O professor Joaquim cedeu-me um espaço no grupo de pesquisa DMAG, e por ele tive meus semestres mais produtivos na faculdade (e que renderam frutos). Já a professora Andrea – além de dar aulas mágicas – notou um suposto potencial em mim (algo que até agora não encontrei) e ofereceu-me a minha primeira oportunidade de projeto na universidade; embora ela provavelmente não tenha percebido, aprendi muito sobre o convívio humano enquanto foi minha orientadora.

Apesar de sempre gostar de jogos digitais como um meio de entretenimento, nunca os vi como temas interessantes de pesquisa. Isso só mudou em 2019, um pouco antes de eu entrar no DMAG. Foi jogando Caves of Qud que cheguei ao tema deste TG, ao ficar impressionado com a riqueza de interações presentes no jogo, possíveis graças a pipelines complexos de geração procedural de conteúdo criados pelos desenvolvedores. Portanto, é apenas natural que eu também agradeça ao desenvolvedores de Caves of Qud – Brian Bucklew e Jason Grinblat – pela existência deste trabalho. Normalmente, vejo jogos como nada além de entretenimento – tais como filmes ou livros de ficção –, mas alguns são tão especiais que é difícil reduzi-los apenas a isso.

Em retrospecto, é engraçado pensar que o prelúdio deste trabalho teve seu início quando, após uma sessão jogando Qud, pesquisei "resources to learn procedural generation" em um dos oráculos da internet. Acabei caindo num grande buraco de coelho.

Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called data. The evolution of a process is directed by a pattern of rules called a program. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

(Harold Abelson e Gerald Sussman, "Structure and Interpretation of Computer Programs")

RESUMO

EXPLORAÇÃO DE TÉCNICAS ATUAIS DE PROGRAMAÇÃO DE JOGOS E GERAÇÃO PROCEDURAL DE AMBIENTES ATRAVÉS DO DESENVOLVIMENTO DE UM PROTÓTIPO DE JOGO EM RUST

AUTOR: Pedro Probst Minini
ORIENTADOR: Joaquim Vinicius Carvalho Assunção

Em muitas instâncias, jogos digitais são considerados *sandboxes* ideais para a exploração de diversas facetas de programação, como inteligência artificial e geração procedural. Seguindo essa ideia, este trabalho propõe – através do desenvolvimento de um protótipo de jogo 2D – a exploração e análise de diversos temas atualizados de computação: arquitetura de software *entity-component-system* (ECS), design *data-driven* e geração procedural de ambientes; este último combinando algoritmos construtivos tradicionais com uma versão adaptada do inovador algoritmo *WaveFunctionCollapse*. Especificamente, além de demonstrar como aplicar os conceitos mencionados em um jogo, assim como descrever problemas, soluções e especialmente sinergias previamente não documentadas entre diversas técnicas de geração procedural, foram realizadas análises sobre os métodos de geração procedural utilizados, através de medidas de desempenho e definições de possíveis casos de uso para cada algoritmo.

Palavras-chave: Jogos Digitais. Entity-Component-System. Data-Driven Design. Geração Procedural de Conteúdo. Rust.

ABSTRACT

EXPLORATION OF CURRENT GAME PROGRAMMING TECHNIQUES AND PROCEDURAL GENERATION OF ENVIRONMENTS THROUGH THE DEVELOPMENT OF A GAME PROTOTYPE IN RUST

AUTHOR: Pedro Probst Minini
ADVISOR: Joaquim Vinicius Carvalho Assunção

In many instances, digital games are considered to be ideal *sandboxes* for exploring various facets of programming, such as artificial intelligence and procedural generation. Following this idea, this work proposes – through the development of a 2D game prototype – the exploration and analysis of several up-to-date computing subjects: *entity-component-system* (ECS) software architecture, *data-driven design* and procedural generation of environments; the latter combining traditional constructive algorithms with an adapted version of the innovative *WaveFunctionCollapse* algorithm. Specifically, in addition to demonstrating how to apply the mentioned concepts in a game, as well as describing problems, solutions and especially synergies previously undocumented between the various procedural generation techniques, the procedural generation methods utilized were analyzed through performance measures and definitions of possible use cases for each algorithm.

Keywords: Digital Games. Entity-Component-System. Data-Driven Design. Procedural Content Generation. Rust.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

CA	*Cellular Automata*
RW	*Random Walker*
IA	Inteligência Artificial
UI	*User Interface*
BSP	*Binary Space Partitioning*
BMP	*Bitmap*
CPU	*Central Processing Unit*
ECS	*Entity-Component-System*
FOV	*Field of View*
GiB	Gibibyte (1 GiB $= 1024^3$ bytes, 1 GB $= 1000^3$ bytes)
GHz	Gigahertz
PCG	*Procedural Content Generation*
PDG	*Procedural Dungeon Generation*
POO	Programação Orientada a Objetos
RON	*Rusty Object Notation*
RPG	*Role-Playing Game*
RTS	*Real-Time Strategy*
NPC	*Non-Playable Character*
WFC	*WaveFunctionCollapse*
DOTS	*Data-Oriented Technology Stack*
JSON	*JavaScript Object Notation*
TDCL	*Top-Down Cavern-Like*
TDML	*Top-Down Mansion-Like*
UFSM	Universidade Federal de Santa Maria

SUMÁRIO

1 INTRODUÇÃO

O cenário de desenvolvimento de jogos evolui constantemente com o passar do tempo, do mesmo modo como ocorre em outros setores tecnológicos. Em especial, jogos vêm sendo programados seguindo princípios orientados a dados em detrimento da orientação a objetos, através da arquitetura *entity-component-system* (ECS). Como exemplos na indústria, o famoso jogo multijogador de tiro em primeira pessoa *Overwatch* utiliza a arquitetura ECS (FORD, 2017); além disso, ECS vem ganhando suporte inclusive através de *game engines* populares, como a Unity por meio do novo *Data-Oriented Technology Stack* (DOTS)[1]. No entanto, a bibliografia referente a ECS é escassa, com informação dispersa em websites e seminários de compartilhamento de experiências em desenvolvimento de jogos, o que diminui a acessibilidade a essa arquitetura. O protótipo de jogo desenvolvido neste trabalho – implementado na linguagem de programação Rust – tem como base um ECS, e busca ilustrar como a arquitetura é tipicamente utilizada em jogos. Ainda, não apenas é utilizado ECS, como também é feito o uso de design *data-driven* ao elaborar arquivos de dados contendo a descrição de diversas entidades do jogo, que serão interpretadas em código – uma prática comum atualmente.

Ademais, desde a última década vêm se popularizando os jogos dos gêneros *roguelike* e *roguelite* – este último mais acessível ao grande público. Ambos os gêneros têm dois aspectos em comum: morte permanente (*permadeath*) e geração procedural de conteúdo, de modo que cada rodada do jogo seja inédita. Geração procedural fornece um maior grau de valor de rejogabilidade (*replay value*) em jogos – além da redução dos custos de produção. Dependendo do gênero do jogo, geração procedural pode ser utilizado densamente ou raramente, e de várias maneiras: na geração de itens, missões, ambientes, personagens, entre outros. Atualmente, tornou-se relativamente comum algum aspecto de geração procedural estar presente desde títulos independentes (*Spelunky*, *Noita*) até jogos AAA de alto custo de produção (*Civilization*, *Borderlands*). Especificamente, este trabalho também foca em explorar diversas técnicas de geração procedural de ambientes, gerando níveis visualmente diversos com a utilização do inovador *WaveFunctionCollapse* (2016) em conjunto com diferentes algoritmos classicamente utilizados em jogos.

1.1 OBJETIVO PRINCIPAL

O objetivo geral deste trabalho consiste na exploração técnicas de arquitetura de software aplicadas ao desenvolvimento de jogos, juntamente com a elaboração de *pipelines* híbridos (combinando diferentes algoritmos) de geração procedural de ambientes e

[1] <https://unity.com/pt/dots>.

a posterior comparação qualitativa entre os métodos de geração utilizados neste trabalho. Para demonstrar os sistemas explorados atuando em conjunto, foi utilizado como plano de fundo um protótipo de jogo digital 2D do gênero *roguelike*.

1.2 OBJETIVOS ESPECÍFICOS

- Aplicação da arquitetura de software *entity-component-system* – em crescente uso na indústria – em detrimento do paradigma convencional de programação orientada a objetos.
- Desenvolvimento de um sistema *data-driven*, no qual entidades de jogo (cores, equipamentos, personagens não jogáveis etc.) são definidas em arquivos de dados separados do código-fonte, os quais são carregados e interpretados em tempo de execução. Jogos com suporte a modificações de jogadores (*modding*) comumente permitem que os jogadores acessem e modifiquem os arquivos de dados (chamados de *raws*), sendo, portanto, um aspecto importante para jogos que pretendem atingir maiores níveis de rejogabilidade.
- Geração procedural de ambientes utilizando algoritmos construtivos de geração procedural de *dungeons* (como são chamados os mapas/ambientes a serem explorados pelo jogador em alguns jogos) e o *WaveFunctionCollapse* – um algoritmo que retorna uma saída de tamanho maior (neste trabalho, a saída é uma *dungeon* ou parte dela) localmente similar à entrada, que é do mesmo formato, mas de tamanho menor –, realizando uma comparação entre as abordagens e combinando-as para gerar resultados distintos.

1.3 ORGANIZAÇÃO DO TRABALHO

- Capítulo 1: apresenta a introdução do trabalho e os objetivos almejados.
- Capítulo 2: realiza a fundamentação teórica de todos os conceitos requeridos para o entendimento deste trabalho, além de apresentar as ferramentas utilizadas para a sua elaboração.
- Capítulo 3: resume os trabalhos relacionados que auxiliaram a produção desta monografia, em menor e maior grau.
- Capítulo 4: descreve como foi implementado um *entity-component-system* para o protótipo do jogo apresentado.

- Capítulo 5: expõe como foram utilizados os diversos algoritmos de geração procedural de ambientes introduzidos no segundo capítulo.
- Capítulo 6: exibe os diversos aspectos *data-driven* do jogo implementado.
- Capítulo 7: demonstração do protótipo do jogo desenvolvido.
- Capítulo 8: apresenta a conclusão do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordados tópicos essenciais ao entendimento do trabalho. Em específico, são formalizados os diversos conceitos nos quais este trabalho é fundamentado.

2.1 O QUE É UM ROGUELIKE?

A definição precisa de um jogo como pertencente ao gênero *roguelike* gera calorosas discussões em fóruns na internet. Entre comunidades mais conservadoras de jogadores, um jogo é chamado de *roguelike* tradicional quando segue a maioria das diretrizes impostas pela Interpretação de Berlim[1] (acordo sobre a definição das características de um jogo *roguelike*, ocorrido durante a *International Roguelike Development Conference 2008* em Berlim), que consta, entre outras imposições:

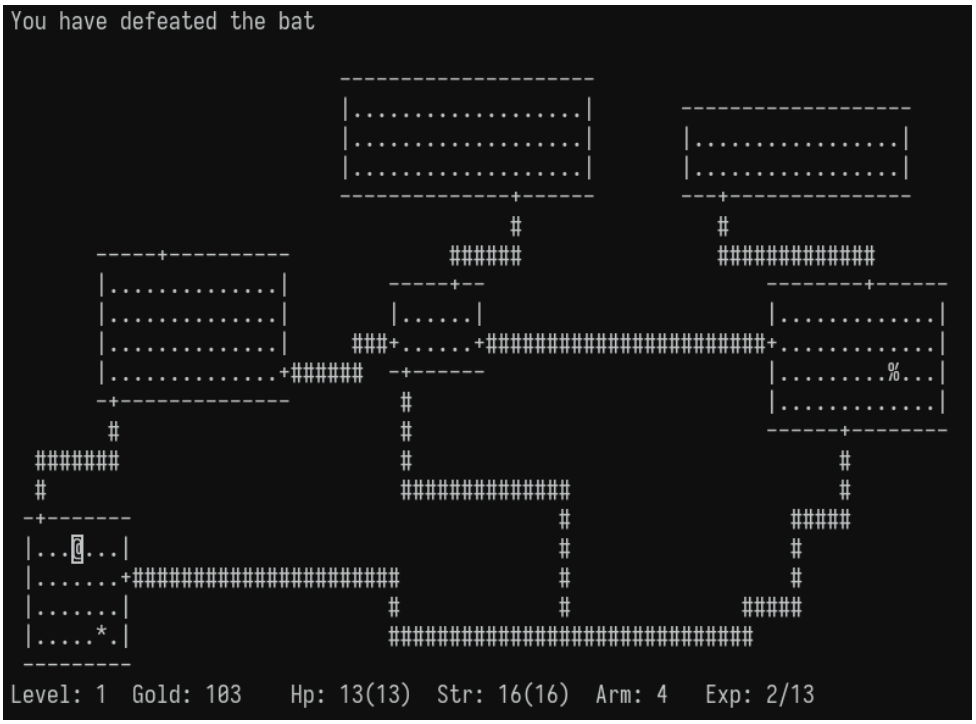
- Ambientes gerados proceduralmente.
- Mortes permanentes (*permadeath)*, ou seja, após a derrota, o jogador precisa recomeçar o jogo do início.
- Basedo em turnos (após uma ação do jogador, os inimigos realizam suas ações).
- Movimento em *grid* (e.g. xadrez), na qual as entidades do jogo movimentam-se apenas ortogonalmente e diagonalmente. Cada célula da *grid* é chamada de *tile*.
- Exploração, combate e gerenciamento de inventário.

O jogo desenvolvido neste trabalho segue as diretrizes mais importantes da Interpretação de Berlim (descritas acima), e, portanto, é classificado como um *roguelike* tradicional – nota-se, entretanto, que nem todos os *roguelikes* tradicionais seguem rigidamente a Interpretação de Berlim, que tem sido descrita como defasada[2].

O termo *roguelike* tem suas origens no jogo *Rogue* (TOY; ARNOLD, 1980) – um jogo com temática de alta fantasia –, desenvolvido inicialmente para *mainframes* baseados em UNIX. *Rogue* era popular entre programadores, e com o passar dos anos muitos desenvolveram jogos com estética e mecânicas semelhantes ao *Rogue* original; daí originou-se o gênero de jogos *roguelike*, que evoluíram à medida que jogos foram avançando como um todo. A Figura 2.1 apresenta uma captura de tela de *Rogue*, enquanto a Figura 2.2 exibe um *roguelike* tradicional com uma temática radicalmente diferente e gráficos em 3D.

[1] <http://www.roguebasin.com/index.php?title=Berlin_Interpretation>.
[2] *Screw the Berlin Interpretation!:* <http://www.gamesofgrey.com/blog/?p=403>.

Figura 2.1 – Captura de tela do jogo *Rogue* (1980) sendo executado em um emulador de terminal moderno.



Fonte: Próprio autor.

Figura 2.2 – Captura de tela do jogo *Jupiter Hell* (2019-2021), um *roguelike* tradicional moderno em 3D com temática de ficção científica inspirada pelo jogo *Doom* (1993).

Fonte: Próprio autor.

2.2 ENTITY-COMPONENT-SYSTEM

O *entity-component-system* (ECS, ou entidade-componente-sistema) é uma arquitetura de software que utiliza a localidade de dados (*data-oriented design*) para tirar vantagem do *caching* da CPU (NYSTROM, 2014b), acelerando o acesso à memória, e o princípio de *composição sobre herança* (GAMMA et al., 1994). Estruturalmente, o ECS é dividido em três eixos principais (MARTIN, 2007):

- *Entities* (entidades): são os blocos conceituais do sistema, que representam objetos concretos do programa (e.g. um elfo ou uma espada em um típico jogo de alta fantasia); em outras palavras, entidades podem ser consideradas *contêineres de componentes* (NYSTROM, 2014a), sendo inúteis por si só.

- *Components* (componentes): são blocos que fornecem funcionalidades às entidades. Por exemplo, em um jogo de alta fantasia uma entidade Dragão pode ter os seguintes componentes rotulando-a: *Nome*, *CospeFogo*, *IAHostil*, *Imunidades* etc.

- *Systems* (sistemas): são os blocos globais, os quais conectam as entidades aos seus respectivos componentes, fornecendo a lógica de cada funcionalidade. Num típico *game loop*, vários sistemas rodam simultaneamente e a todo o tempo. Um sistema de *Fome*, por exemplo, acessa todas as entidades rotuladas com o componente *SenteFome*, retirando 1 ponto de vida da entidade a cada turno enquanto ela estiver rotulada com *SenteFome*.

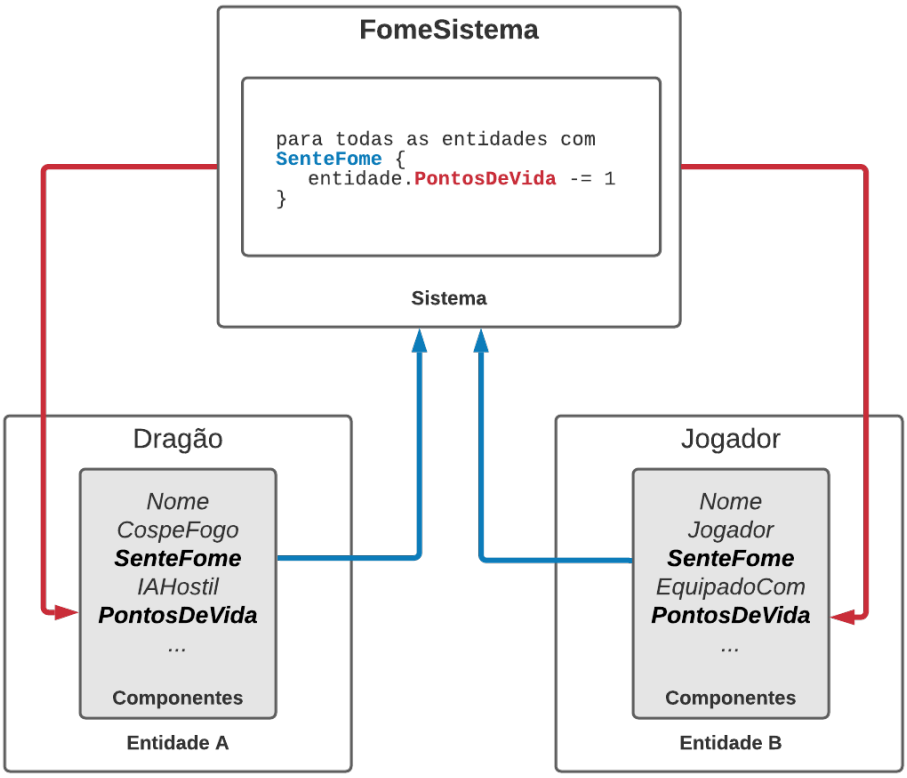
Apesar de ainda subutilizada, há crescente suporte à arquitetura ECS por meio de bibliotecas específicas de linguagens de programação e até mesmo conhecidas *game engines*[3]. A Figura 2.3 contém um exemplo de funcionamento básico do ECS envolvendo duas entidades e um sistema.

Por conta de sua natureza modular – facilitada com a modificação dos componentes de entidades em tempo de execução –, a arquitetura ECS pode evitar diversos *pitfalls* presentes na programação orientada a objetos (POO), como ambiguidade em herança múltipla e problemas de sincronia em paralelização (HARKONEN, 2019). De acordo com Mertens (2018), uma diferença chave entre POO e ECS é que em POO, o comportamento e os dados de uma entidade são parte de um objeto; no ECS, as informações são dissociadas em componentes e sistemas, e as entidades podem ser facilmente estendidas. Por exemplo, suponha que o jogador tenha encontrado um amuleto para o seu personagem que o torne imune ao fogo; neste caso, ao vestir o amuleto, pode ser adicionado um componente de imunidade ao fogo para o jogador de forma dinâmica, e nenhuma função adicional precisaria ser escrita, já que o *sistema* responsável por fornecer funcionamento

[3]Documentação sobre ECS em Unity: <https://docs.unity3d.com/Packages/com.unity.entities@0.14/manual/index.html>.

ao *componente* de imunidade trataria este caso automaticamente, pois o jogador é uma *entidade* que contém o componente de imunidade. Ao retirar o amuleto, o componente de imunidade é removido da entidade, e, caso a entidade não tiver nenhum componente de imunidade restante, ela é "desconectada" do sistema de imunidades.

Figura 2.3 – Exemplo simplificado de um sistema de fome em um jogo utilizando ECS.



Fonte: Próprio autor.

2.2.1 Vantagens do ECS

- Internamente, um ECS é projetado seguindo o design orientado a dados, o que proporciona, de acordo com Llopis (2009):
  - Fácil paralelização, pois os dados são processados em grupos, o que facilita a separação destes em diferentes *threads*.
  - Uso otimizado da *cache*, pois os dados estão dispostos em regiões contíguas na memória.
- POO tem foco em entidades singulares, no entanto, um programador de jogos normalmente manipula múltiplas entidades (LLOPIS, 2009).

- Inexistência de problemas como a ambiguidade em herança múltipla, que podem surgir à medida que árvores de herança crescem.
- Modularidade e simplicidade através do conceito de composição sobre herança.
- Jogos complexos com um grande número de entidades podem tirar maior proveito de ECS e design orientado a dados, como jogos de estratégia em tempo real (RTS) e *role-playing games* (RPGs).

2.2.2 Problemas do ECS

- Por conta do design orientado a dados, o ECS é bastante eficiente ao lidar com grandes quantidades de dados, mas não ao lidar com entidades singulares (HARKONEN, 2019). Portanto, a escrita de sistemas para entidades únicas pode ser redundante.
- Como os sistemas do ECS são essencialmente genéricos, pode se tornar complexo estendê-los para tipos particulares de entidades.
- Pode ser desnecessário no desenvolvimento de jogos simples com poucas entidades (e.g. *Flappy Bird*). Em uma entrevista com Stefan Reinalter[4] realizada por Straume (2019), Reinalter exemplificou dois casos em que design orientado a dados (no qual ECS é fundamentado), preferencialmente, poderia e não poderia ser utilizado, respectivamente:
  - Um sistema de partículas que transforma milhares de partículas a cada *frame*.
  - Um sistema de interface que não transforma mais de 50 *widgets* a cada *frame*.
- É pouco conhecido e não tem suporte nativo em linguagens de programação, ou seja, um ECS precisa ser projetado ou importado de bibliotecas para ser utilizado.

2.3 GERAÇÃO PROCEDURAL DE CONTEÚDO

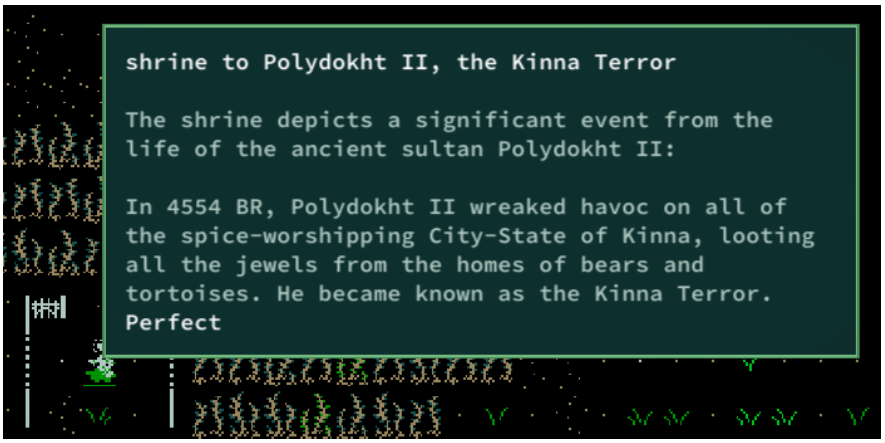
Geração Procedural de Conteúdo (PCG) é a automação da produção de mídia, que seria tipicamente produzida por um humano (BARRIGA, 2019). Na área de jogos digitais, PCG pode ser aplicada de diversos meios, geralmente em forma textual, sonora ou gráfica.

Nota-se que PCG é mais densamente observado na comunidade de jogos independentes, principalmente porque é composto por equipes de desenvolvimento menores. Nos

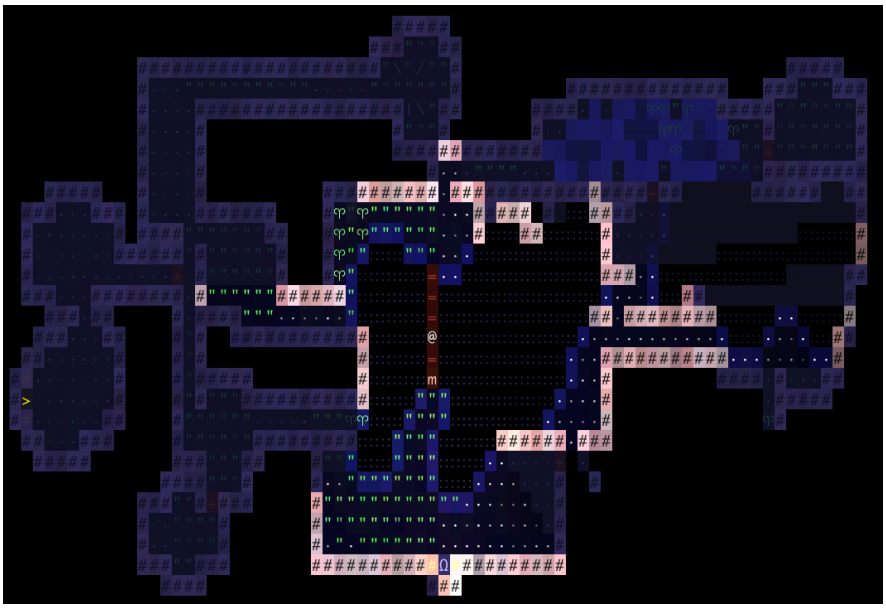
[4] <https://at.linkedin.com/in/stefan-reinalter-a2334b9>.

últimos anos, *roguelikes* e gêneros semelhantes têm sido responsáveis por popularizar e, assim, expandir os limites da geração procedural. Por exemplo, *Dwarf Fortress* – um jogo de construção de colônias – simula toda a história de um mundo fictício como forma de criar uma narrativa única (VELD et al., 2015). O *roguelike Caves of Qud* faz uso criativo de geração procedural de ambientes e narrativa (GRINBLAT; BUCKLEW, 2017) para simular um mundo de temática *dying earth* (um futuro onde o planeta está em seus estágios finais de existência), com suas próprias ruínas, sultões e cultos. A Figura 2.4 contém dois exemplos de aplicações distintas de PCG em jogos.

Figura 2.4 – Duas facetas de PCG: em (a), um fragmento textual gerado proceduralmente em *Caves of Qud*; em (b), uma *dungeon* gerada proceduralmente em *Brogue*.



(a)



(b)

Fonte: Próprio autor.

Devido ao seu amplo espectro de pesquisa, este trabalho enfoca um único aspecto de PCG: a geração procedural de *dungeons* (PDG). Especificamente, mostra-se – de ma-

neira simples – como criar *pipelines* de geração combinando uma implementação genérica do algoritmo *WaveFunctionCollapse* (WFC)[5] com métodos de geração construtivos tradicionais para gerar ambientes esteticamente ou estrategicamente interessantes. Embora *dungeon* (calabouço) seja um termo utilizado extensivamente, ele nem sempre representa um espaço isolado e fechado; informalmente, os métodos de PDG também podem incluir a geração de ambientes menos claustrofóbicos como florestas, vales e desfiladeiros. Neste trabalho, *dungeon*, "mapa" e "ambiente" são termos utilizados de forma intercambiável.

Para o *pipeline* de geração de *dungeons* implementado neste trabalho, utilizam-se duas estratégias de algoritmos classificadas por Shaker, Togelius e Nelson (2016):

1. **Algoritmos construtivos**: baseiam-se tipicamente no posicionamento aleatório de elementos no mapa, gerando conteúdo em tempo fixo e geralmente curto, sem realizar avaliação do conteúdo posteriormente para gerá-lo novamente caso necessário. Encontram-se nesta classe muitos algoritmos baseados em grafos e autômatos celulares. Tipicamente possuem um processo de implementação mais simples.

2. **Algoritmos baseados em busca**: durante ou após a etapa de geração, são realizados testes para assegurar a consistência do conteúdo gerado; por conta disso, esses algoritmos têm tempo de execução bastante variável. Encontram-se nesta classe algoritmos evolutivos e outros algoritmos estocásticos de busca/otimização – inclusive o WFC (que também pode ser considerado como uma abordagem híbrida baseada em solucionador), utilizado neste trabalho. Tipicamente possuem uma implementação mais rigorosa.

Este trabalho contém *pipelines* de geração para dois tipos de ambientes, podendo ser combinados: *top-down mansion-like* (TDML) e *top-down cavern-like* (TDCL) (VIANA; SANTOS, 2019). O primeiro refere-se a ambientes constituídos primariamente por salas conectadas por corredores, enquanto o último refere-se a ambientes mais caóticos e orgânicos, como cavernas ou florestas. O termo *top-down* concerne à visão da câmera do jogo: de cima para baixo. Enquanto os algoritmos utilizados neste trabalho são utilizados especificamente para um jogo *top-down*, variações dos mesmos também podem ser utilizadas em jogos *sidescrollers* (visão lateral, como em *Mario*) e até mesmo em 3D.

2.3.1 Descrições dos algoritmos de geração procedural utilizados

Neste trabalho, os algoritmos selecionados estão resumidos no Quadro 2.1. Os algoritmos construtivos escolhidos são tradicionalmente utilizados em *roguelikes*, enquanto

[5]Originalmente desenvolvido por Maxim Gumin e disponível em: <https://github.com/mxgmn/WaveFunctionCollapse>.

o WFC foi selecionado por conta de suas características inovadoras. Seguindo a taxonomia PCG de Togelius et al. (2011), todos esses algoritmos são estocásticos e gerados durante o tempo de execução do jogo. Eles também têm algum grau de controle por meio de parametrização.

Quadro 2.1 – Breve comparação entre os algoritmos escolhidos.

Algoritmo	Método de Geração	Entrada	Saída	Conectividade	Complexidade
Passeio Aleatório	Construtivo	Preenchida	TDCL	Parcial	Baixa
Autômatos Celulares	Construtivo	Caótica	TDCL	Não	Baixa
BSP Dungeon	Construtivo	Preenchida	TDML	Total	Média
Digger	Construtivo	Preenchida	TDML	Total	Média
WFC	Busca	Qualquer	Similar à entrada	Não	Alta

Uma entrada *preenchida* corresponde a uma lista de *tiles* representando um mapa onde todos os *tiles* são de um único tipo sólido; uma entrada *caótica* representa uma mistura de tipos de *tiles*. *Conectividade* refere-se à capacidade do algoritmo de gerar ambientes totalmente conectados (sem regiões isoladas), enquanto *complexidade* está relacionada à dificuldade de implementação.

2.3.1.1 Construtivos

- **Passeio Aleatório**
  Como o nome sugere, o algoritmo de passeio aleatório (*random walk* ou *drunkard's walk*)[6] opera com base na utilização de agentes (*walkers*) que percorrem n passos em direções aleatórias, gerando uma única caverna (quando os agentes são iniciados na mesma posição) ou um sistema de cavernas (quando os agentes são iniciados em posições diferentes). A vida útil de cada agente e a direção seguida em cada etapa são determinadas aleatoriamente a partir de um intervalo escolhido, e o número de agentes é aumentado a cada iteração do algoritmo, que depende do número de *tiles* "escavados" desejados. No modo de algoritmo para a geração de um sistema de cavernas ao invés de uma única caverna, a conectividade entre as regiões não é garantida, pois os agentes podem ser inicializados em posições diferentes no mapa e nunca se encontrarem. A saída é caótica e visualmente orgânica.

- **Autômatos Celulares**
  O algoritmo de autômatos celulares (*cellular automata*) aplicado à PDG[7] (JOHNSON; YANNAKAKIS; TOGELIUS, 2010) utiliza teoria de autômatos celulares (WOLFRAM, 1983) para modificar entradas caóticas. Com base em regras definidas pelo usuário,

[6] <http://www.roguebasin.com/index.php?title=Random_Walk_Cave_Generation>.

[7] <http://www.roguebasin.roguelikedevelopment.org/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels>.

o mapa é “suavizado” em n iterações; quanto maior o número de iterações, maior a suavização sofrida no mapa.

As regras definidas têm base nas células vizinhas da célula sendo analisada; a vizinhança a ser considerada pode ser a *vizinhança de Moore* (vizinhos ortogonais e diagonais) ou a *vizinhança de von Neumann* (apenas vizinhos ortogonais). Um exemplo de regra arbitrária, por exemplo, poderia ser a seguinte: considerando uma célula de valor 0, ela terá seu valor alterado para 1 caso 4 células vizinhas tenham valor 1. Naturalmente, o programador tem flexibilidade para fazer suas próprias regras. No entanto, quanto mais regras forem inseridas, mais lenta será a execução do algoritmo; dependendo do tamanho do mapa gerado, isso pode impactar o desempenho.

Tal como o algoritmo de passeio aleatório, autômatos celulares também produzem saídas orgânicas. Saídas de ambos os algoritmos podem ser visualizadas na Figura 2.5. Além disso, autômatos celulares têm alta capacidade sinérgica, podendo ser combinados com diversos algoritmos durante o processo de geração.

Figura 2.5 – Mapas gerados por passeio aleatório e autômatos celulares, respectivamente.

Fonte: *Roguebasin.*

- ***Binary Space Partitioning Dungeon***

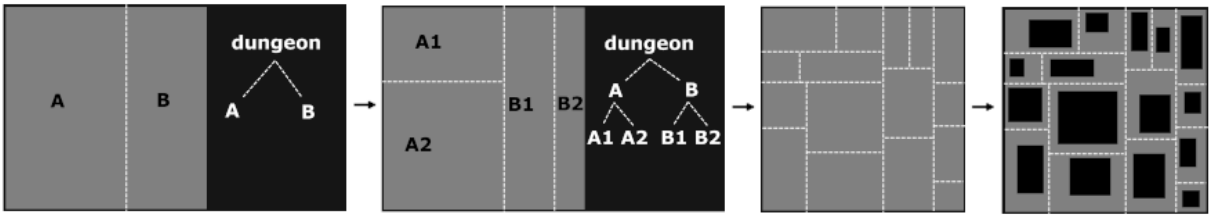
  O algoritmo *Binary Space Partitioning (BSP) Dungeon*[8] é um dos algoritmos TDML mais utilizados em jogos *roguelike*. Ele é baseado na subdivisão do espaço e na subsequente adição de salas nos espaços gerados, com base em uma árvore BSP (SCHUMACHER et al., 1969). No plano 2D, o algoritmo opera como uma *quadtree*; em 3D, como uma *octotree* (SHAKER et al., 2016).

  Após a geração das salas (Figura 2.6), a lista de salas pode ser reordenada por posição, tamanho, entre outros atributos. As conexões entre as salas são geralmente

[8] <http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation>.

feitas sequencialmente; portanto, a ordem das salas na lista determinará como as salas serão conectadas, alterando o resultado final.

Figura 2.6 – Representação visual do algoritmo BSP Dungeon em 2D.



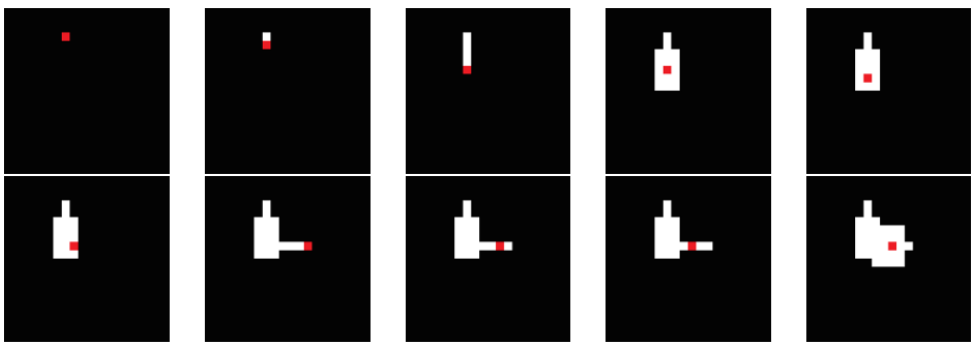
Fonte: *Roguebasin.*

A saída do algoritmo é composta por salas de qualquer formato (tipicamente retangulares) conectadas por corredores.

- ***Digger/Tunneler***

  Além do algoritmo BSP Dungeon, *Digger* (SHAKER et al., 2016) (ou *Tunneler*) é outro algoritmo utilizado para gerar mapas TDML. Nesse algoritmo, as salas geradas são normalmente derivadas de uma sala localizada centralmente. A cada iteração, um índice de borda aleatório da sala sendo analisada é escolhido; e a partir desse índice, outra sala é gerada na mesma direção da borda, escolhida a uma distância determinada aleatoriamente. Em seguida, as duas salas são conectadas por uma "sala estreita", definindo um corredor. O algoritmo termina quando o número máximo de iterações é atingido, o que potencialmente indica a impossibilidade de inserção de salas do tamanho solicitado. Apesar de compartilhar um nome comum, o algoritmo *Digger* tende a ser severamente modificado entre as implementações. Um exemplo do algoritmo pode ser visto na Figura 2.7 – no entanto, diferentemente da versão implementada neste trabalho, os corredores são gerados *anteriormente* à geração de uma sala.

Figura 2.7 – Funcionamento do *Digger* tradicional.

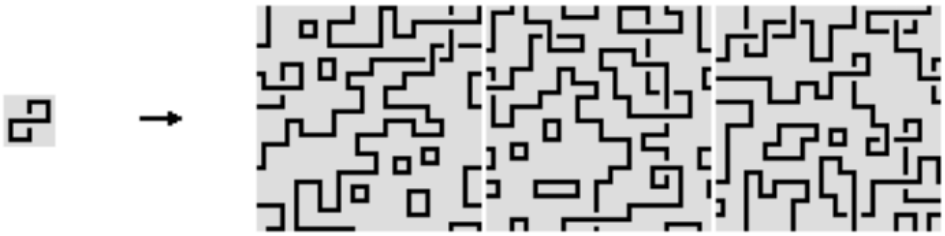


Fonte: Shaker et al. (2016).

2.3.1.2 Baseados em busca

- ***WaveFunctionCollapse***
  O algoritmo *WaveFunctionCollapse*[9] (WFC, Figura 2.8) foi o único algoritmo baseado em busca escolhido, devido aos seus aspectos inovadores e crescente utilização na indústria. O WFC utiliza uma pequena entrada e gera uma saída de maior tamanho, a qual é localmente semelhante à entrada. Originalmente feito para trabalhar com valores de pixel em imagens *bitmap* (BMP), foi adaptado neste trabalho para aceitar *tiles* genéricos.

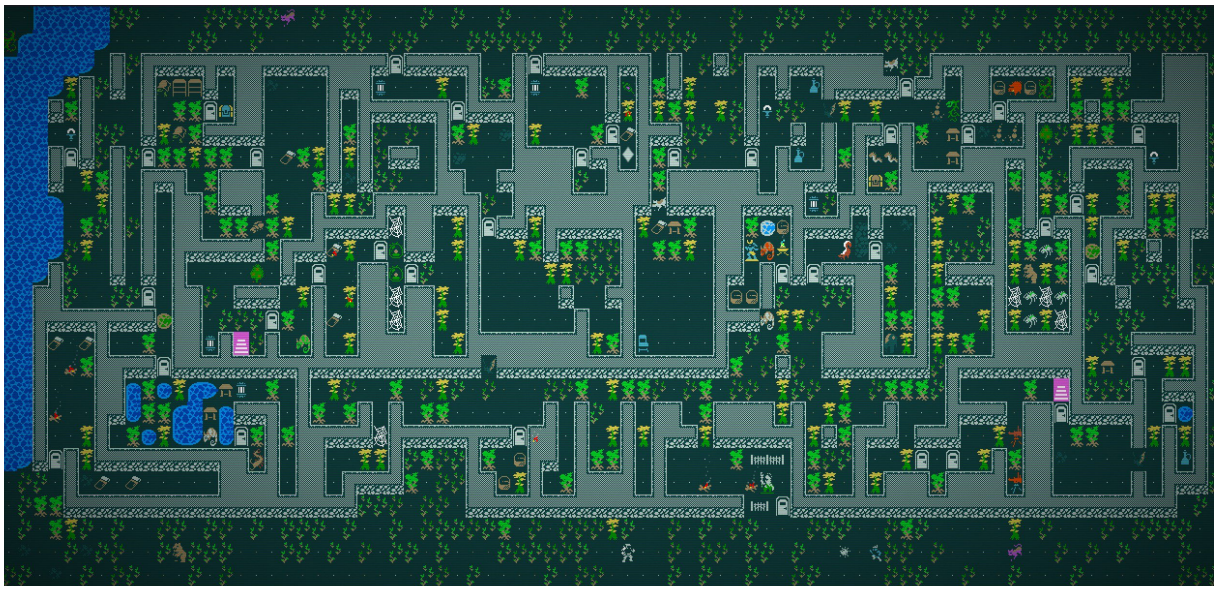
Figura 2.8 – WFC original. Uma entrada e três saídas distintas geradas.



Fonte: Maxim Gumin.

Na indústria, *Caves of Qud* foi um dos primeiros jogos comerciais a utilizar o WFC no *pipeline* de geração de ambientes. Esse jogo é conhecido por conter *dungeons* diversas e arquiteturalmente interessantes; a Figura 2.9 contém um exemplo.

Figura 2.9 – WFC como arquitetura interna em *Caves of Qud*.



Fonte: Brian Bucklew <https://twitter.com/unormal/status/819725864623603712>.

Essencialmente, o WFC é um solucionador complexo (KARTH; SMITH, 2017) – cada célula de tamanho fixo no mapa inicialmente tem a possibilidade de aceitar todos os

[9] <https://github.com/mxgmn/WaveFunctionCollapse>.

padrões de *tiles* gerados a partir das entradas particionadas (que são refletidas, rotacionadas e assim por diante), e a partir das *regras de adjacência* o conjunto de padrões é reduzido até que a célula seja “colapsada", ou seja, quando ela possui apenas um padrão possível. As mudanças são propagadas da célula atual sendo analisada para as células vizinhas, que também têm seus conjuntos de possibilidades reduzidos; por analogia, essa etapa funciona de forma semelhante a um jogo de *Sudoku*: cada vez que um número é inserido numa célula, diminui-se a quantidade de números possíveis que podem ser inseridos nas outras células. A próxima célula escolhida para ser colapsada é aquela com o menor valor de entropia (incerteza). O cálculo da entropia é utilizado para reduzir a chance de remover todas as possibilidades de uma célula (SHERRATT, 2019), o que acarretaria em um estado de contradição.

$$EntropiaWFC = \log_2 W - (\frac{\sum_{i=1}^{n} w_i \log_2 w_i}{W})$$

Onde W é a soma das frequências relativas de cada padrão possível na célula analisada, igual a:

$$W = \sum_{i=1}^{n} w_i$$

O algoritmo termina quando todas as células do mapa são colapsadas; caso contrário, o algoritmo chega a uma contradição – nesse caso, ele pode reiniciar ou retroceder, dependendo da implementação. Cada célula colapsada é então formatada para a saída. Resumidamente, o WFC funciona da seguinte maneira:

1. Pré-processamento:
   (a) Leitura da entrada.
   (b) Particionamento da entrada (coleta de padrões).
   (c) Reflexão e rotação dos padrões (opcional).
   (d) Cálculo das frequências relativas dos padrões.
   (e) Definição das regras de adjacência; em outras palavras, um parâmetro que define a compatibilidade entre padrões. Se dois padrões são suficientemente similares, eles são compatíveis e podem ser vizinhos.
   (f) Inicialização das células do *grid*, que inicialmente podem aceitar todos os padrões coletados na etapa de pré-processamento.
2. Núcleo WFC (repete-se até que todas as células sejam colapsadas):
   (a) Escolhe-se a célula com o menor valor de entropia para ser colapsada. Na primeira iteração, a escolha é aleatória.

(b) A partir das regras de adjacência, a célula escolhida é colapsada, ou seja, aceita apenas um padrão. Na primeira iteração, a escolha é aleatória.

(c) Propaga a mudança para as células vizinhas, que por conta das regras de adjacência precisam ter seus conjuntos de possibilidades de padrão reduzidos. Os valores de entropia são atualizados.

3. Pós-processamento:

(a) O resultado final do WFC é mapeado para a saída.

Mais detalhes sobre o algoritmo são descritos na Seção 5.6.

2.4 DESIGN DATA-DRIVEN

Neste trabalho, design *data-driven* – não confundir com design *data-oriented, ou orientado a dados* – refere-se à prática de definir todas as entidades de jogo (e.g. itens, mobs, equipamentos) separadamente do código-fonte, como forma de facilitar a inserção e modificação de novas entidades no jogo sem alterar porções de código já escritas, o que auxilia no desenvolvimento. Outra vantagem dessa abordagem é permitir a modificação dos arquivos de dados por usuários finais do jogo (*modding*). *Modding* é uma prática que auxilia na rejogabilidade de um jogo, enquanto propicia maior interação na comunidade de jogadores. Existem inúmeras plataformas de *modding* na web, sendo o *Nexus*[10] e o *Mod DB*[11] as duas mais conhecidas.

No desenvolvimento de jogos – especialmente *roguelikes* e gêneros similares – é relativamente comum definir as entidades de jogos separadamente do código fonte. Por exemplo, para acomodar diferentes níveis de complexidade, o jogo *Dwarf Fortress* faz extenso uso do design *data-driven*, chamando de *raws*[12] os arquivos de dados – nomenclatura também adotada neste trabalho. Uma porção de um arquivo *raw* de *Dwarf Fortress* – referente à definição de insetos no jogo – pode ser visualizada na Figura 2.10.

Ainda, os desenvolvedores do jogo *Caves of Qud* elaboraram um sistema combinando um modelo próprio de ECS com design *data-driven* (BUCKLEW, 2015), demonstrando a robustez do sistema para jogos mecanicamente complexos e a sinergia entre as duas abordagens.

[10] <https://www.nexusmods.com/>.

[11] <https://www.moddb.com/>.

[12] Enquanto o código-fonte de *Dwarf Fortress* é fechado, os *raws* estão disponíveis publicamente em: <https://github.com/Qartar/dwarf-fortress/tree/legacy/raw/objects>.

Figura 2.10 – Definição de uma mosca em um *raw* do jogo *Dwarf Fortress*.

```
creature_insects
[OBJECT:CREATURE]
[CREATURE:FLY]
        [DESCRIPTION:A tiny flying insect found around rotting meat and
        garbage. These bugs are widely considered to be a nuisance.]
        [NAME:fly:flies:fly]
        [CASTE_NAME:fly:flies:fly]
        [CREATURE_TILE:250][COLOR:0:0:1]
        [NATURAL]
        [BIOME:NOT_FREEZING]
        [BIOME:ANY_POOL]
        [VERMIN_MICRO]
        [VERMIN_ROTTER]
        [VERMIN_GROUNDER][FREQUENCY:100]
        [VERMIN_HATEABLE]
        [UBIQUITOUS]
        [VERMIN_NOTRAP]
        [POPULATION_NUMBER:2500:5000]
        [CLUSTER_NUMBER:100:200]
        [SMALL_REMAINS]
        [PREFSTRING:ability to annoy]
        [FLIER]
        [DIURNAL]
        ...
...
```

2.5 FERRAMENTAS UTILIZADAS

Esta seção apresenta as principais ferramentas utilizadas para a implementação deste trabalho.

- **A linguagem Rust**
  Apesar de jovem (2015), a linguagem de programação multiparadigma Rust está em crescente uso na indústria de jogos[13] e desenvolvimento de sistemas. Diferentemente de C e C++, Rust promove segurança de memória por design, enquanto possui desempenho comparável a C++[14].

- **specs**
  A *specs*[15] é uma biblioteca escrita em Rust que implementa um ECS. Ela promove fácil paralelismo, alta flexibilidade e bom desempenho.

[13] <https://arewegameyet.rs/>.
[14] <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html>.
[15] <https://github.com/amethyst/specs>.

- **bracket-lib**
  O *bracket-lib*[16] é um *toolkit* escrito em Rust e utilizado primariamente para o desenvolvimento de jogos graficamente simples. Ele contém diversos módulos responsáveis por diferentes funcionalidades, como gráficos, geometria, algoritmos de *pathfinding* etc. O *bracket-lib* foi escolhido por possuir extensa documentação (incluindo tutoriais) e facilidade de integração com o *specs*.

- **Rusty Object Notation**
  Na implementação do design *data-driven* neste trabalho, utiliza-se *Rusty Object Notation* (RON) como o formato apropriado para os arquivos de dados pelos seguintes motivos: melhor legibilidade e distinção entre estruturas de dados, suporte a comentários e uma sintaxe similar à linguagem de programação Rust. Observe a comparação na Figura 2.11 entre, respectivamente, *JavaScript Object Notation* (JSON) e RON:

Figura 2.11 – Comparação entre JSON e RON.

```
{
  "materials": {
    "metal": {
      "reflectivity": 1.0
    },
    "plastic": {
      "reflectivity": 0.5
    }
  },
  "entities": [
    {
      "name": "hero",
      "material": "metal"
    },
    {
      "name": "monster",
      "material": "plastic"
    }
  ]
}
```

```
Scene( // Nome de classe (opcional)
  materials: { // Map
    "metal": (
      reflectivity: 1.0,
    ),
    "plastic": (
      reflectivity: 0.5,
    ),
  },
  entities: [ // Array
    (
      name: "hero",
      material: "metal",
    ),
    (
      name: "monster",
      material: "plastic",
    ),
  ],
)
```

Fonte: RON GitHub, disponível em <https://github.com/ron-rs/ron>.

[16] <https://github.com/thebracket/bracket-lib>.

3 TRABALHOS RELACIONADOS

Diante da miríade de possibilidades de pesquisa referentes a jogos digitais, estes são comumente utilizados como plataformas para explorar diversas aplicações de Ciência da Computação – como inteligência artificial, algoritmos de geração procedural e arquitetura de software, sendo os dois últimos especialmente relacionados a este trabalho. Ainda que os temas explorados aqui pertençam a um pequeno nicho, os trabalhos citados abaixo podem abordar assuntos que, apesar de não diretamente aplicados, serviram de referência ou inspiração para a elaboração deste trabalho.

Silva (2015) desenvolveu um jogo do gênero *roguelike* com o intuito de explorar geração procedural de *dungeons* – utilizando os algoritmos *BSP Dungeon*, autômatos celulares e busca em profundidade aplicado à geração de labirintos – e IA de inimigos com base em máquinas de estados e algoritmos de *pathfinding*. Diferentemente do trabalho desenvolvido nesta monografia, Silva fornece maior enfoque à aplicação de técnicas de IA, explorando em menor grau os algoritmos de PDG utilizados.

Viana (2019), além de realizar uma extensa revisão bibliográfica referente a PDG, utilizou e expandiu uma solução *search-based* – um algoritmo genético – para realizar esboços de *dungeons* com barreiras – mais especificamente, barreiras com chaves (e.g. portas trancadas). Similarmente, Pereira (2019) aplicou algoritmos genéticos para a geração de *dungeons* com barreiras e objetivos (missões), utilizando algoritmos construtivos para o preenchimento do espaço gerado.

Linden, Lopes e Bidarra (2013) forneceram uma visão geral referente às diversas classes de algoritmos utilizados em PDG, incluindo conceitos básicos e exemplos de aplicações na indústria. Similarmente, Shaker et al. (2016) elaboraram um livro introdutório sobre métodos de geração procedural em jogos. Ainda, Viana e Santos (2019) realizaram uma inspeção sobre os trabalhos mais recentes relacionados à PDG, classificando diversos algoritmos com base na taxonomia proposta por Togelius et al. (2011).

Em um dos primeiros artigos publicados sobre o WFC, Karth e Smith (2017) descreveram o algoritmo e seus pontos fortes, formulando-o como um problema de programação de conjunto de respostas (*Answer Set Programming*). Além disso, eles mostraram o WFC sendo utilizado para gerar níveis em jogos e até mesmo poesia.

Na *Roguelike Celebration* de 2019, Bucklew (2019) descreveu parcialmente o *pipeline* de geração de *dungeons* em *Caves of Qud*. Ainda, na *Roguelike Celebration* de 2020, Wolverson (2020) detalhou diversos algoritmos construtivos de geração procedural de *dungeons* aqui utilizados. O trabalho anterior (MININI; ASSUNÇÃO, 2020) do autor desta monografia descreveu brevemente o uso do WFC em conjunto com outros algoritmos de geração procedural para jogos 2D com visão *top-down*. Infelizmente, apesar do uso na indústria, poucos trabalhos na academia estão relacionados à aplicação do WFC.

Enquanto este trabalho apresenta uma implementação genérica do WFC como em Minini e Assunção (2020), outros trabalhos descrevem como ele pode ser expandido: Scurti e Verbrugge (2018) mostraram como o WFC pode ser modificado para criar caminhos aleatórios interessantes para personagens não jogáveis (NPCs) em jogos. Kim et al. (2019) propuseram uma implementação do WFC baseada em grafos ao invés de sua implementação comum baseada em *grid*, com o intuito de facilitar o uso de PDG no espaço de grafos e em mundos tridimensionais de forma eficiente. Sandhu, Chen e McCoy (2019) descreveram uma implementação do WFC integrando restrições de design específicas para gerar níveis, proporcionando assim mais controle do desenvolvedor sobre o algoritmo.

Não foram encontrados trabalhos acadêmicos diretamente relacionados à aplicação da arquitetura de software ECS em jogos. Harkonen (2019) descreveu a estrutura básica de um ECS e demonstrou suas vantagens em relação ao paradigma tradicional de orientação a objetos, enquanto Vagedes et al. (2019) demonstraram o uso da arquitetura ECS em simuladores militares, citando a falta de documentação sobre a arquitetura como um problema. Atualmente, nota-se que a maioria das informações relacionadas à aplicação de ECS em jogos encontra-se alheia à academia, sendo dispersa em páginas web.

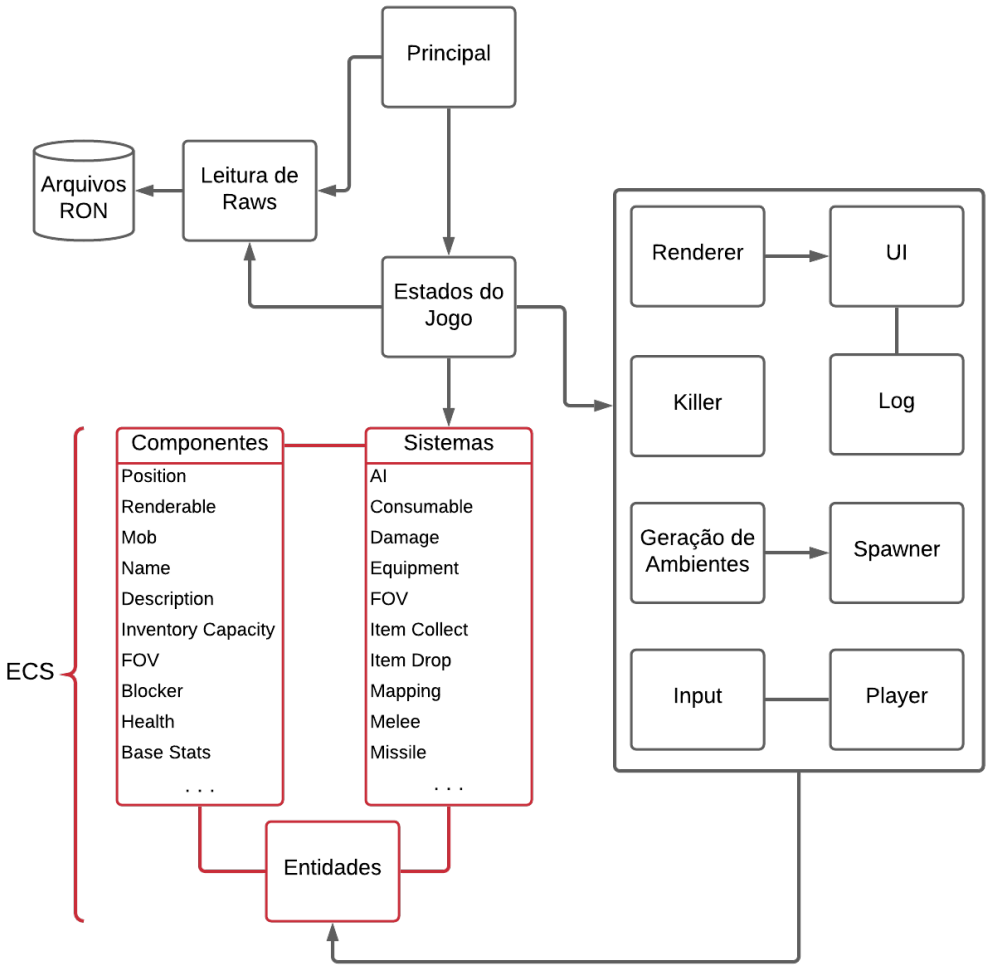
Por fim, este trabalho é relevante pois demonstra as distintas etapas na elaboração de um jogo, combinando diversos temas abordados nas referências acima. Em especial, é demonstrado como aplicar ECS no contexto de jogos digitais, também fazendo uso do design *data-driven* para assistir o desenvolvimento. Ainda, mostra-se como usar uma implementação padrão de WFC (adaptado para *tiles*) de forma prática em conjunto com outros algoritmos comumente utilizados em *roguelikes* – expandindo o trabalho anterior de Minini e Assunção (2020) e as apresentações de Bucklew (2019) e Wolverson (2020).

4 ARQUITETURA DE SOFTWARE

O protótipo de jogo desenvolvido neste trabalho é estruturado a partir de um *Entity-Componenent-System* (ECS), definido na seção 2.2. Para isso, foi utilizada a biblioteca *specs* para a linguagem de programação Rust, juntamente com o *toolkit bracket-lib*. A biblioteca *specs* fornece um ECS inicialmente vazio, sendo de responsabilidade do programador o desenvolvimento de seus próprios componentes e sistemas, que possuem suas escritas facilitadas a partir de diversas funções providas pela biblioteca.

Uma visão da arquitetura atual do programa pode ser visualizada na Figura 4.1. Os sistemas básicos de ECS (em vermelho) e leitura de *raws* foram baseados inicialmente na implementação de Wolverson (2019), e soluções diversas retiradas de outros autores estão creditadas nos arquivos de código-fonte[1].

Figura 4.1 – Arquitetura do jogo utilizando ECS.



Fonte: Próprio autor.

[1] <https://github.com/pprobst/tcc-ufsm-2020>.

Definições referentes à Figura 4.1:

- Principal (`src/main.rs`) - é o arquivo principal do jogo, responsável por carregar os arquivos externos (e.g. *tilesets* e raws) em memória e inicializar: a) um terminal virtual ASCII/CodePage-437[2], b) um "mundo" ECS no qual todos os componentes definidos são inicialmente registrados e c) um *game loop*.
- Leitura de Raws (`src/raws`) - conjunto de arquivos responsáveis por ler e interpretar os arquivos de dados RON.
- Arquivos RON (`resources/raws`) - arquivos de dados que definem as cores dos *tiles*, as entidades do jogo e a tabela de *spawn*.
- Estados do Jogo (`src/state.rs`) - gerencia o *game loop* a partir de uma máquina de estados e inicializa os sistemas do ECS.
- Sistemas (`src/systems`) - arquivos que descrevem o funcionamento dos diversos sistemas do jogo.
- Componentes (`src/components.rs`) - definição dos componentes que fazem parte das entidades definidas nos arquivos de dados.
- Entidades - mobs, itens e equipamentos. Definidos como coleções de componentes e manipulados em sistemas e em outras funções.
- Renderer (`src/renderer.rs`) - renderiza os componenetes da interface e os *tiles* no terminal virtual.
- UI (`src/ui`) - definição dos diversos componentes da interface do usuário (UI).
- Log (`src/log.rs`) - estrutura definida por um vetor de mensagens, as quais são exibidas na tela quando ocorrem eventos relevantes no jogo, como um ataque.
- Killer (`src/killer.rs`) - responsável por remover as entidades do ECS quando as mesmas não são mais necessárias.
- Geração de Ambientes (`src/map_gen`) - arquivos com os diversos algoritmos de geração de ambientes.
- Spawner (`src/spawner.rs`) - responsável por inserir as entidades no ambiente a partir das informações provenientes na tabela de *spawn*, a qual define a chance de *spawning* das entidades.
- Input (`src/input.rs`) - define os controles do jogo para o jogador (*player*).

[2] <https://dwarffortresswiki.org/index.php/Character_table>.

- Player (`src/player.rs`) - descreve as funcionalidades possíveis para o jogador a partir das entradas do usuário (e.g. movimentação, acesso ao inventário, uso de itens etc.).

4.1 UM EXEMPLO DE INTERAÇÃO NO ECS

Por questões de simplicidade, nesta seção é analisado um dos sistemas mais acessíveis do jogo: o sistema de dano. A partir dele, é possível ter um entendimento de como implementar sistemas ECS utilizando a linguagem Rust com a biblioteca *specs*. Para a escrita de um sistema de dano funcional, é preciso:

1. Definir os componentes que serão utilizados (Figura 4.2).
2. Escrever o sistema (Figura 4.3).
3. Executar o sistema a cada *tick* do jogo (Figura 4.4).

O fluxo de cada sistema do jogo é similar ao apresentado na Figura 2.3. Enquanto cada sistema é definido de maneira similar, outras porções de código alheias aos sistemas também podem acessar entidades do ECS; por exemplo, as porções de código que são executadas sob demanda, como componentes da UI, remoção de entidades não utilizadas, criação de entidades a partir dos arquivos de dados, entre outros.

Figura 4.2 – Definição de componentes (`src/components.rs`).

```rust
pub type Position = Point; // Posição da entidade no mapa.

#[derive(Component, Debug)]
pub struct BaseStats { // Componente de status de uma entidade.
    pub health: Health, // Vida.
    pub defense: i32, // Pontos de defesa.
    pub attack: i32, // Pontos de ataque.
    pub god: bool, // Invencível (usado para debugging).
}

#[derive(Component)]
pub struct SufferDamage { // Dano sofrido por uma entidade.
    pub amount: Vec<(i32, bool)>,
}
```

Na Figura 4.3, a lógica do sistema é definida dentro de um laço, no qual são acessadas as entidades que possuem os componentes *SufferDamage*, *BaseStats* e *Position*.

Figura 4.3 – Definição do sistema de dano (`src/systems/damage.rs`).

```
pub struct DamageSystem {}

impl<'a> System<'a> for DamageSystem {
    type SystemData = (
        WriteStorage<'a, SufferDamage>, // Dano sofrido.
        Entities<'a>, // Entidades do ECS.
        ReadExpect<'a, Entity>, // Jogador.
        WriteStorage<'a, BaseStats>, // Pontos de vida etc.
        WriteExpect<'a, Map>, // Mapa registrado no ECS.
        ReadStorage<'a, Position>, // Posição da entidade.
    );

    fn run(&mut self, data: Self::SystemData) {
        let (mut damage, entities, _player, mut stats, mut map,
             position) = data;

        // Para CADA entidade que carrega esses componentes...
        for (damage, _ent, victim_stats, pos) in (&damage,
                                                  &entities,
                                                  &mut stats,
                                                  &position).join()
        {
            // Se a entidade não for invencível, retirar vida
            // da mesma de acordo com o dano total recebido.
            if !victim_stats.god {
                for dmg in damage.amount.iter() {
                    victim_stats.health.hp -= dmg.0;
                }
            }
            // Se a vida da entidade chegar a 0, definir
            // a posição da entidade no mapa como vazia.
            if victim_stats.health.hp <= 0 {
                map.clear_blocker(pos.x, pos.y);
            }
        }
        damage.clear(); // Limpa o vetor de dano recebido.
    }
}
```

É aplicada a quantidade de dano recebido como forma de decremento sobre a quantidade de "vida" das entidades acessadas.

Já na Figura 4.4, o sistema de danos *DamageSystem* é inserido no *game loop* principal do jogo, após os sistemas *MeleeSystem* e *MissileSystem*, referentes ao funcionamento do combate no jogo.

Figura 4.4 – Execução do sistema (`src/state.rs`).

```
fn run_systems(&mut self) {
    let mut vis = FOVSystem {};
    vis.run_now(&self.ecs);

    let mut hostile_ai = HostileAISystem {};
    hostile_ai.run_now(&self.ecs);

    let mut mapping = MappingSystem {};
    mapping.run_now(&self.ecs);

    let mut reload = WeaponReloadSystem {};
    reload.run_now(&self.ecs);

    let mut melee = MeleeSystem {};
    melee.run_now(&self.ecs);

    let mut missile = MissileSystem {};
    missile.run_now(&self.ecs);

    // Aqui!
    let mut damage = DamageSystem {};
    damage.run_now(&self.ecs);

    let mut collect_item = ItemCollectSystem {};
    collect_item.run_now(&self.ecs);

    let mut drop_item = ItemDropSystem {};
    drop_item.run_now(&self.ecs);

    let mut consumable = ConsumableSystem {};
    consumable.run_now(&self.ecs);

    let mut equip = EquipmentSystem {};
    equip.run_now(&self.ecs);

    self.ecs.maintain();
}
```

5 GERAÇÃO PROCEDURAL DE AMBIENTES

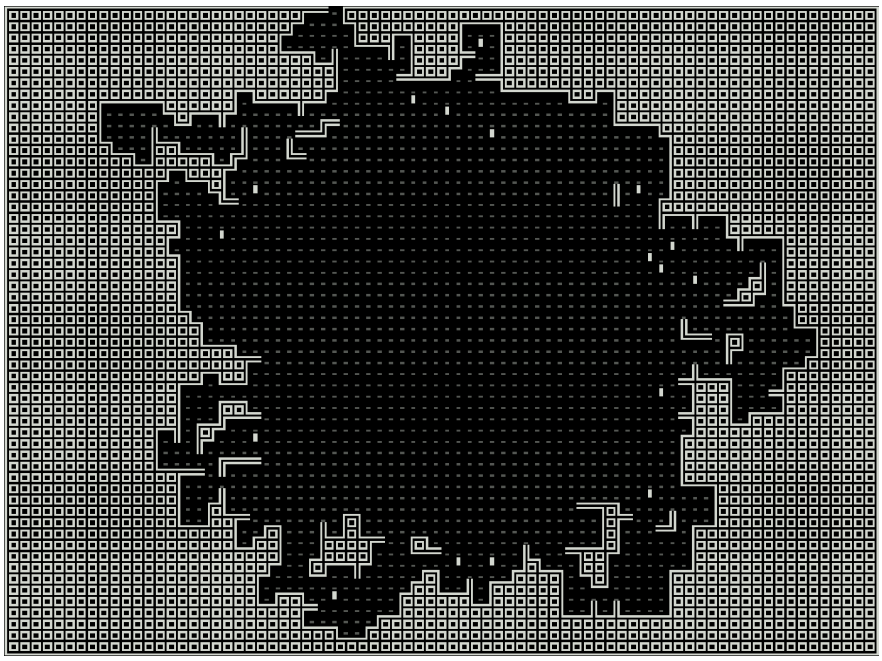
Geração procedural de conteúdo é um dos pilares fundamentais para jogos do gênero *roguelike*; em especial, geração procedural de ambientes – neste caso, comumente chamados de *dungeons*. Como descrito na Subseção 2.3.1, o protótipo de jogo implementado neste trabalho utiliza algoritmos construtivos e algoritmos baseados em busca para gerar cada nível, na maioria das vezes combinando diversas técnicas num mesmo mapa.

As seções a seguir contêm os resultados obtidos com cada algoritmo, executados em mapas de tamanho 80x60 *tiles*. Ao final, são exibidos resultados utilizando combinações dos mesmos. O código-fonte de todos os algoritmos está no *GitHub*[1].

5.1 PASSEIO ALEATÓRIO

O algoritmo de passeio aleatório é considerado um bom algoritmo para a geração de ambientes visualmente orgânicos (ou "naturais"), sendo especialmente útil para a geração de cavernas e sistemas de cavernas – um tipo de ambiente comum em jogos do gênero *roguelike*. Na implementação deste trabalho, utilizam-se agentes inicializados em um único ponto central (Figura 5.1) e agentes inicializados em pontos distintos (Figura 5.2).

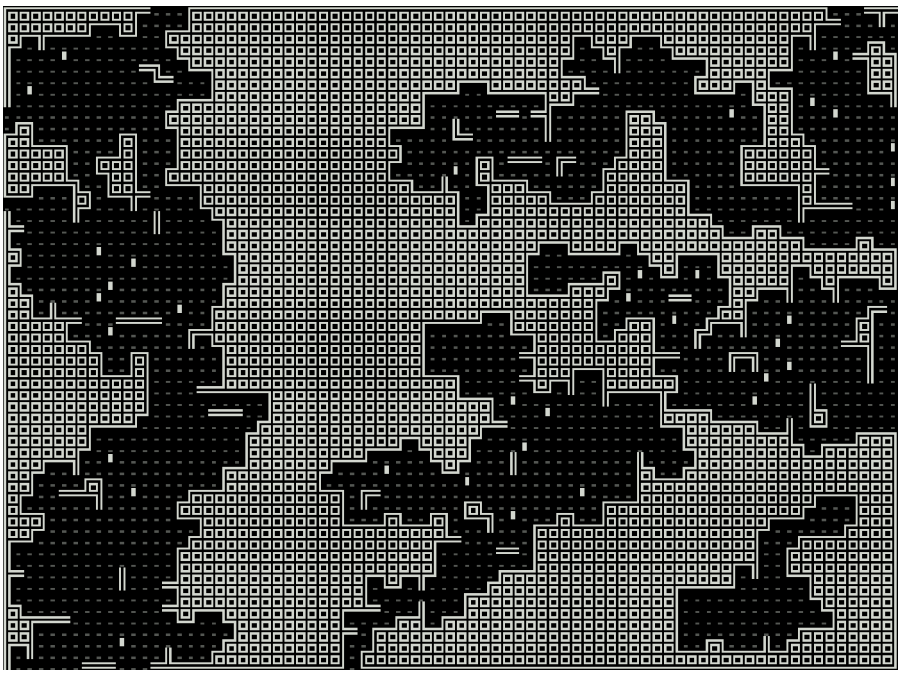
Figura 5.1 – Passeio aleatório com agentes agrupados (40% do mapa escavado).



Utilizando-se agentes agrupados, é gerada uma única caverna totalmente conectada. Entretanto, ao utilizar agentes separados, gera-se um sistema de cavernas que

[1] <https://github.com/pprobst/tcc-ufsm-2020/tree/master/src/map_gen>.

Figura 5.2 – Passeio aleatório com agentes separados (35% do mapa escavado).



frequentemente pode conter regiões isoladas. O problema da *ausência de conectividade* é recorrente em diversos algoritmos, e é discutido na Seção 5.7.

Note que nas duas figuras anteriores, os agentes movem-se apenas ortogonalmente. Naturalmente, também é possível fazê-los com que se movam diagonalmente, como pode ser observado nas Figuras 5.3 e 5.4.

Figura 5.3 – Passeio aleatório com agentes agrupados (40% do mapa escavado), com movimento diagonal.

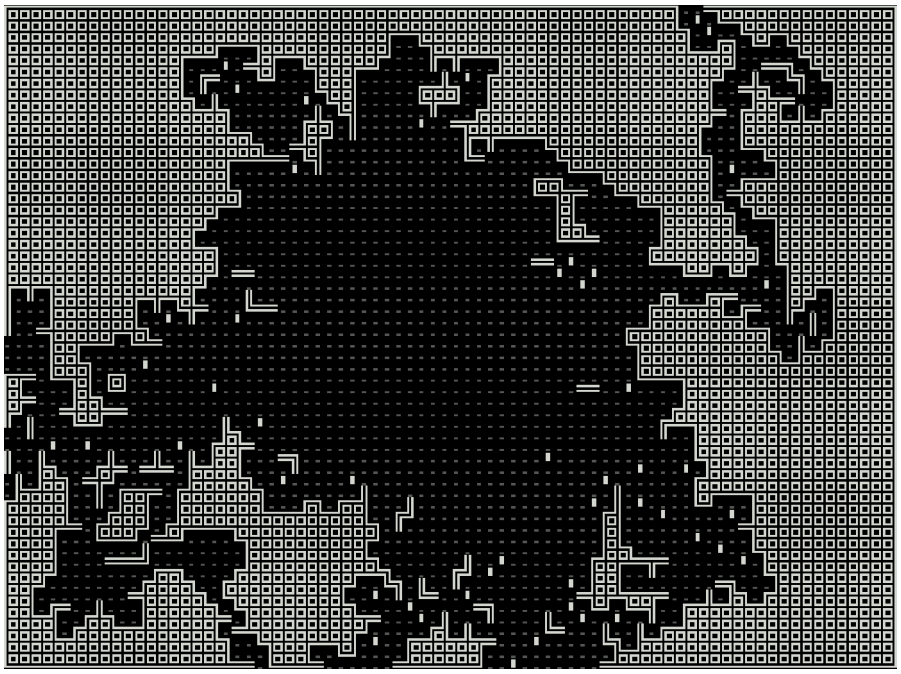
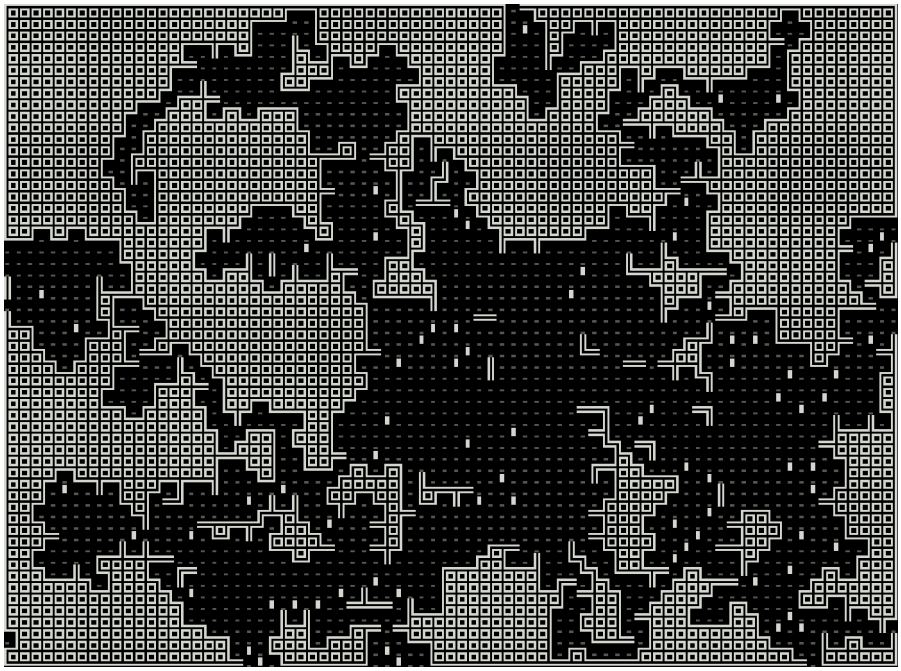


Figura 5.4 – Passeio aleatório com agentes separados (35% do mapa escavado), com movimento diagonal.



Habilitar o movimento diagonal permite que os agentes tenham uma maior variação na escolha aleatória do próximo movimento (8 movimentos possíveis ao invés de apenas 4); logo, o mapa acaba sendo melhor explorado, com uma maior chance de gerar becos. Em contrapartida, pode-se gerar muitos corredores com navegação complicada, o que, em excesso, pode frustrar o jogador.

5.2 AUTÔMATOS CELULARES

Ambientes gerados a partir de autômatos celulares tradicionalmente utilizam um mapa *caótico* como base. Um mapa caótico é aquele no qual um mapa totalmente preenchido com *tiles* sólidos (e.g. “parede”) possui uma porcentagem x (normalmente entre 35% e 50%) de *tiles* vazios, inseridos em posições aleatórias no mapa. Um algoritmo de autômato celular funciona a partir de diversas iterações (ou gerações) sobre o mapa caótico. Quanto maior o número de iterações, maior é a suavização sofrida no mapa.

Na Figura 5.5, “áreas abertas” refere-se a uma pequena modificação nas regras do algoritmo, que tem como efeito criar regiões menos estreitas se comparadas às regiões da Figura 5.6. A modificação é a seguinte: *quando o número de vizinhos com tile do tipo “parede” for menor do que o número da regra a ser aplicada, mas não é igual a zero, transformar o tile atual em “parede”.* No caso da Figura 5.6, a regra é aplicada sem condições adicionais.

Figura 5.5 – Autômatos celulares, 12 iterações, áreas abertas.

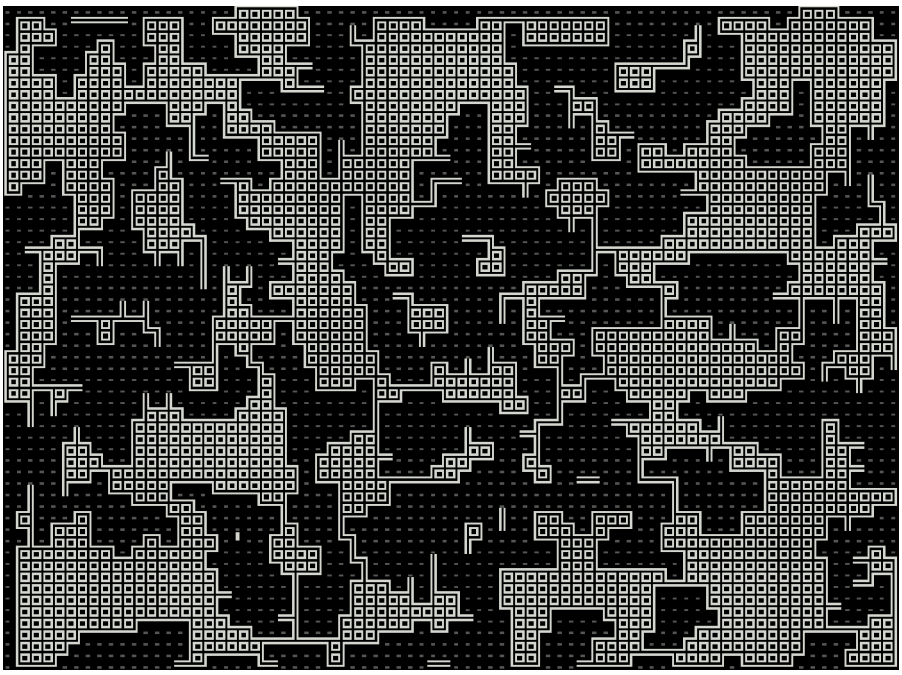
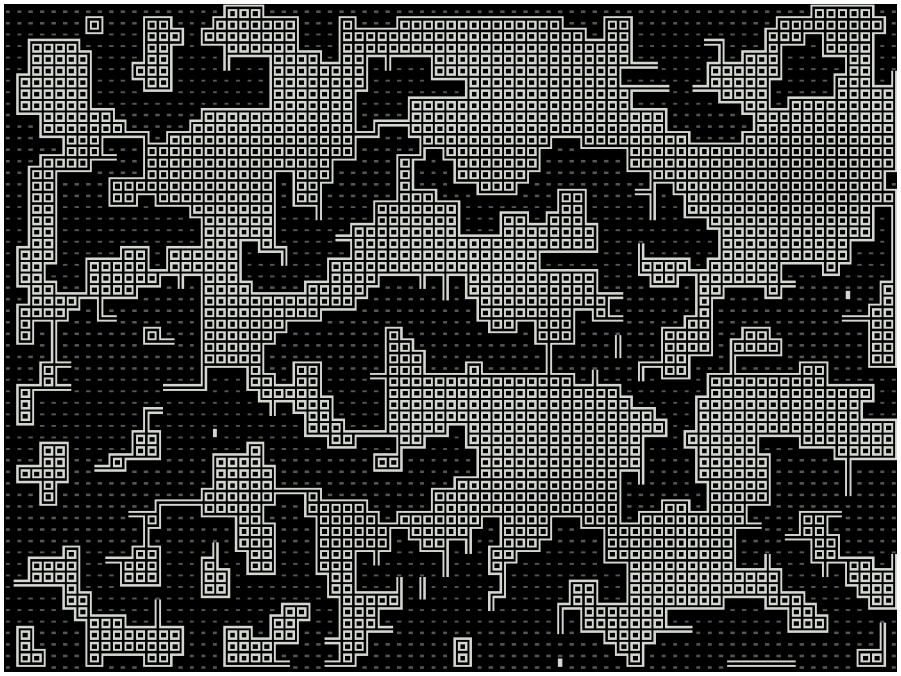


Figura 5.6 – Autômatos celulares, 12 iterações, áreas estreitas.



Como o algoritmo de passeio aleatório gera ambientes caóticos, ele pode ser naturalmente associado a autômatos celulares para suavizações adicionais. Como resultado, são gerados ambientes mais esteticamente agradáveis e diversos, como observados nas Figuras 5.7 e 5.8.

Em algoritmos de autômatos celulares, regras podem ser criadas e modificadas de acordo com a necessidade do programador. Neste trabalho, é adicionada uma importante regra adicional: *quando o número de vizinhos com tile do tipo "água rasa" for maior ou igual*

Figura 5.7 – Passeio aleatório ortogonal + 12 iterações de autômatos celulares.

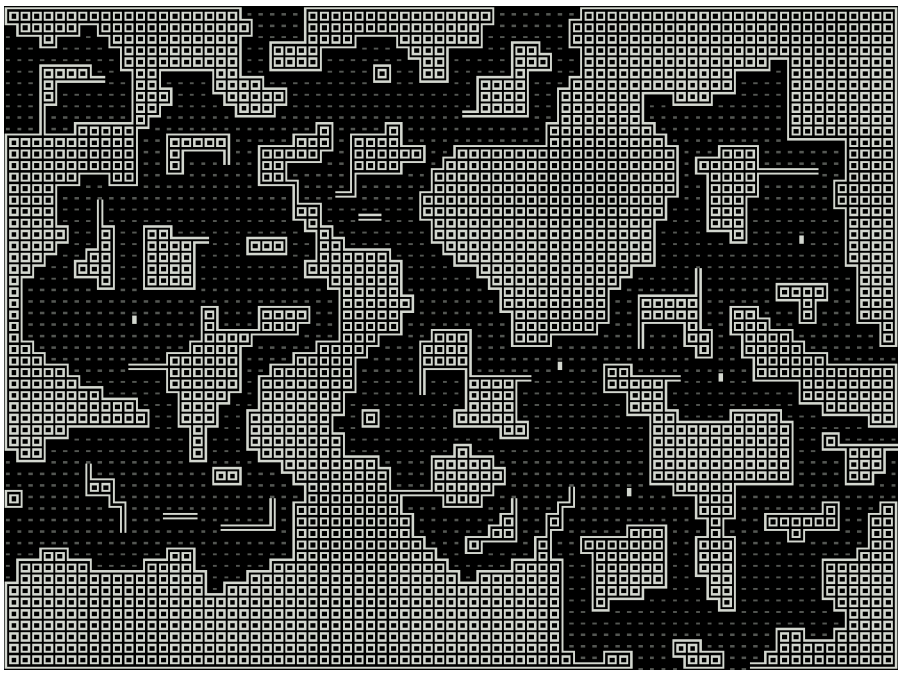
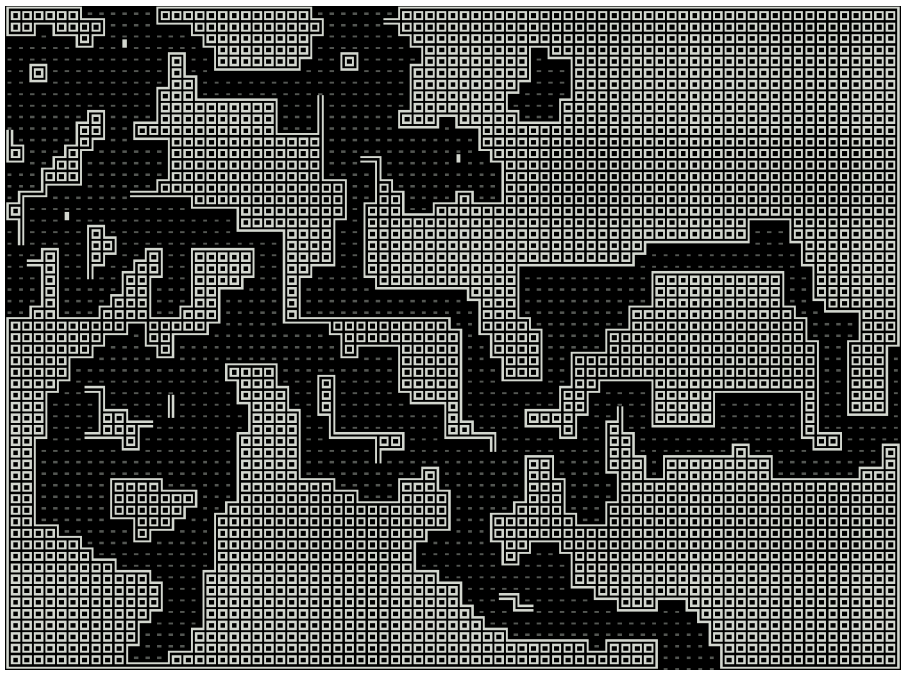


Figura 5.8 – Passeio aleatório diagonal + 12 iterações de autômatos celulares.



a 5, transformar o tile atual em "água profunda". Isso permite a diferenciação entre tipos de águas e provê uma dificuldade adicional ao jogador e seus inimigos, pois ao movimentar-se em águas profundas (azul escuro), pode haver penalidade de movimento. Um exemplo de *dungeon* com essa regra pode ser observado na Figura 5.9.

Autômatos celulares também podem ser utilizados para gerar florestas. Basta substituir *tiles* de "parede" por "árvores", e adicionar vegetação como etapa de pós-processamento, como pode ser observado na Figura 5.10.

Figura 5.9 – Autômatos celulares com lagos.

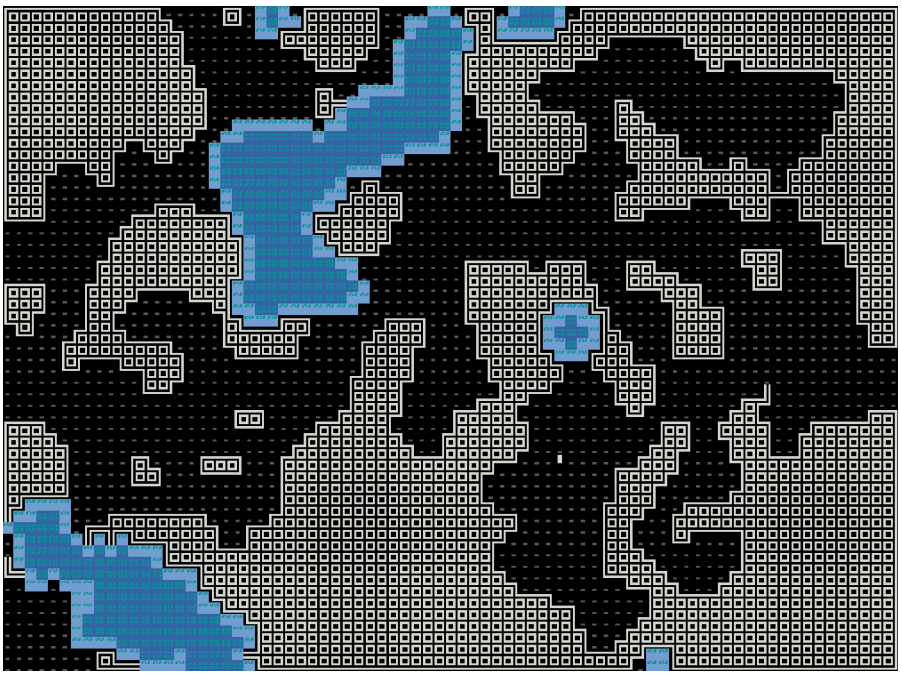
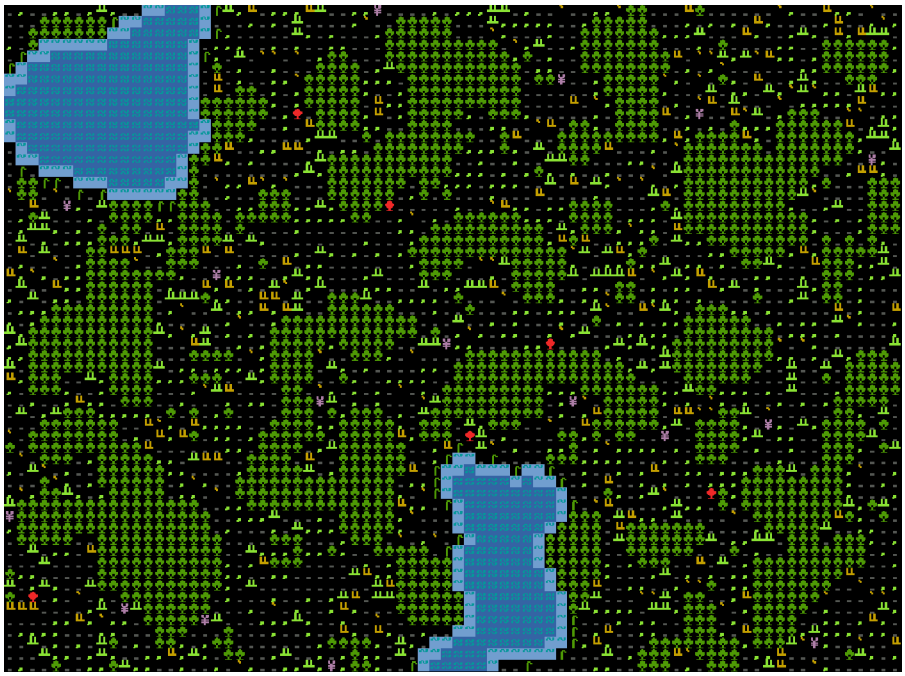


Figura 5.10 – Autômatos celulares gerando florestas.

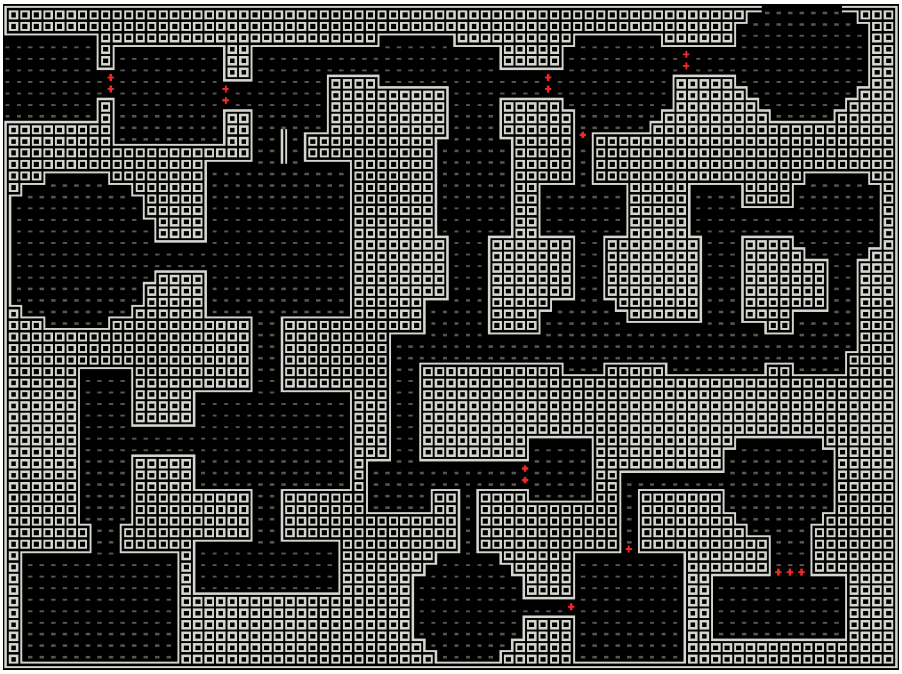


Na Seção 5.8, é explorada a capacidade sinérgica dos autômatos celulares ao combiná-los com outros algoritmos de geração procedural para gerar variados tipos de ambientes.

5.3 BSP DUNGEON

Ao contrário dos algoritmos de passeio aleatório e autômatos celulares, *o BSP Dungeon* gera ambientes visualmente artificiais: salas de formas diversas tipicamente conectadas por corredores, como pode ser observado na Figura 5.11.

Figura 5.11 – *BSP Dungeon* com salas conectadas aleatoriamente.



A conexão entre as salas é feita no último passo do algoritmo, atuando sobre a *lista* de salas resultantes. Inicialmente desordenada, essa lista pode ser ordenada de acordo com parâmetros arbitrários definidos pelo programador. Com a lista ordenada com base em um certo parâmetro, as conexões entre as salas podem seguir um formato específico (WOLVERSON, 2019), não precisando ser aleatórias como na Figura 5.11.

Por exemplo, na Figura 5.12 a lista de salas é ordenada com base na coordenada $x1$ (canto superior esquerdo de cada sala) de forma crescente; assim, as primeiras salas da lista são aquelas que estão localizadas mais à esquerda no mapa, enquanto as últimas salas da lista são as localizadas mais à direita. Como a conexão é feita sequencialmente a partir da lista, o resultado é uma *dungeon* que deve ser explorada pelo jogador de uma forma mais linear (caso o ponto de surgimento ou *spawn point* do jogador esteja localizado no extremo leste ou oeste). A *dungeon* da Figura 5.13 é gerada de modo similar, a diferença está no parâmetro considerado, que é a coordenada $y1$.

Nas figuras analisadas, os caracteres avermelhados '+' são definidos como portas, e não fazem parte do algoritmo *BSP Dungeon*. As portas são inseridas como uma etapa de pós-processamento.

Figura 5.12 – *BSP Dungeon* com salas conectadas da esquerda para a direita.

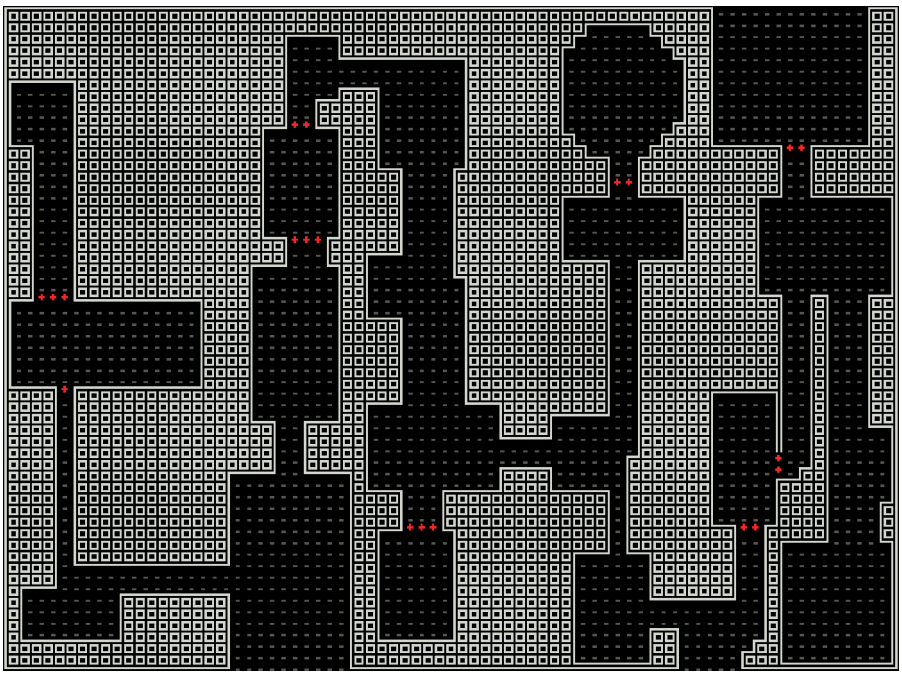
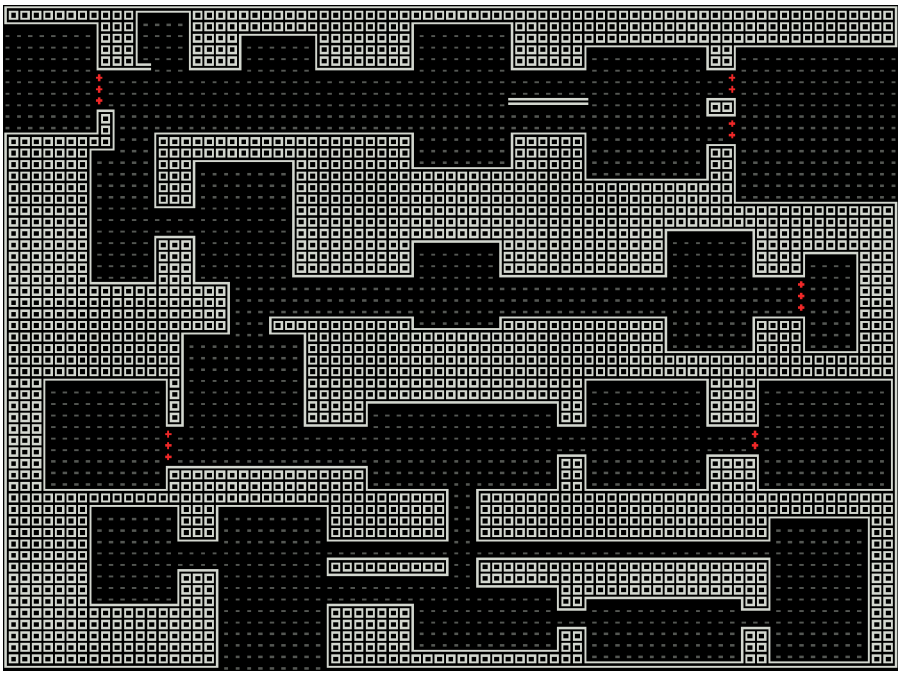


Figura 5.13 – *BSP Dungeon* com salas conectadas de cima para baixo.



5.4 DIGGER

Similarmente ao *BSP Dungeon*, o método *Digger* (ou *Tunneler*) também gera ambientes TDML. Em relação à implementação típica deste algoritmo[2], esta foi alterada para que, visualmente, as *dungeons* geradas pelo algoritmo sejam setorizadas, com salas menores conectando-se a salas maiores; assim, as *dungeons* geradas pelo *Digger* diferenciam-

[2] <http://www.roguebasin.com/index.php?title=Dungeon-Building_Algorithm>.

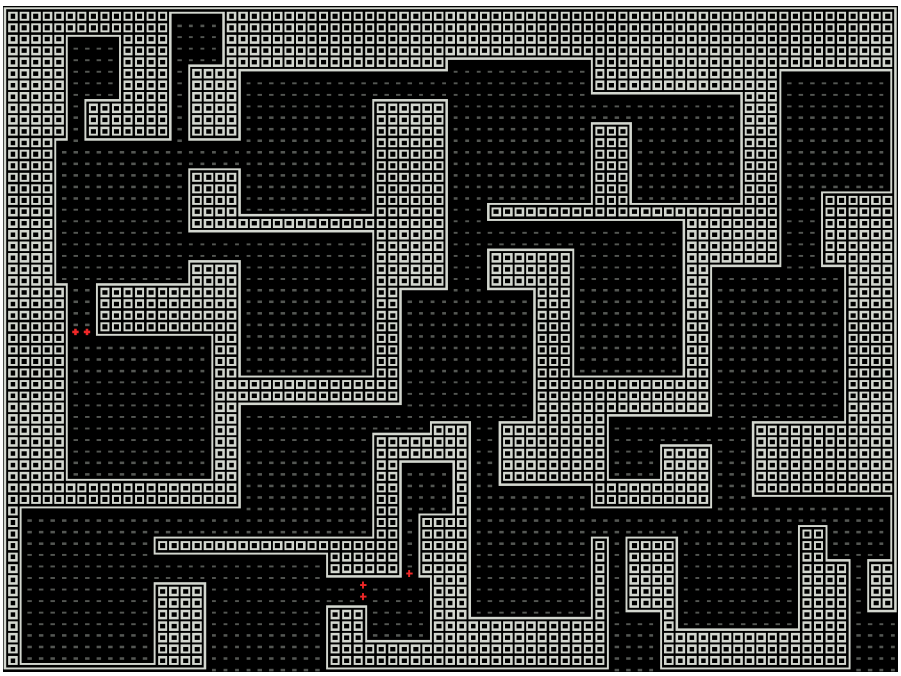
se mais daquelas geradas pelo método BSP. Isso se deve principalmente a duas modificações:

1. No *loop* de adição de salas, a próxima sala selecionada aleatoriamente a partir da qual será gerada e conectada outra sala *nunca* deve ser a sala selecionada na iteração anterior.

2. A adição de um *loop* extra de adição de salas periféricas menores após a inserção das salas principais.

Os tamanhos mínimos e máximos das salas principais são ajustados via passagem de parâmetros, e as salas periféricas terão metade dos tamanhos passados como parâmetro. Outro parâmetro relevante é o *número de features*: quanto maior, mais salas serão adicionadas no mapa *em ambos os loops*. Entretanto, por conta das limitações espaciais do mapa, o número de *features* não reflete o número exato de salas no mapa, pois o algoritmo para quando a inserção não é mais possível. Exemplos de saídas podem ser observados nas Figuras 5.14 e 5.15.

Figura 5.14 – *Digger*, tamanho mínimo 10, tamanho máximo 15, 30 *features*.



Utilizar tamanhos de salas menores gera ambientes quase labirínticos, mas ainda assim mantém-se o aspecto setorizado (Figura 5.16).

As salas geradas por ambos os algoritmos *BSP Dungeon* e *Digger* são vazias. Entretanto, arquitetura interna pode ser inserida de diversas maneiras, sendo uma delas utilizando WFC, como descrito nas Seções 5.6 e 5.8.

Figura 5.15 – *Digger*, tamanho mínimo 10, tamanho máximo 20, 30 *features*.

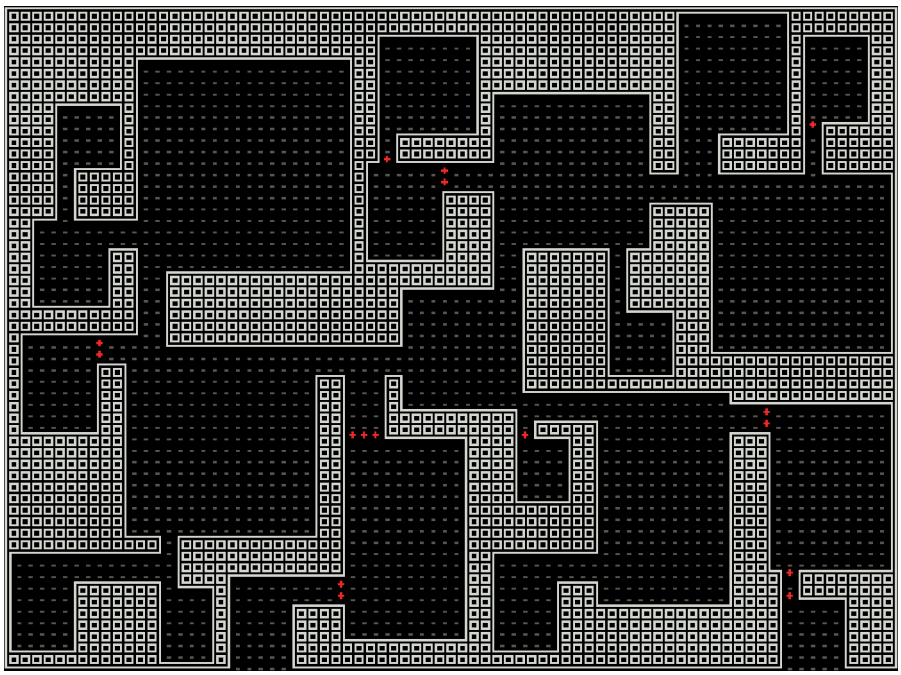
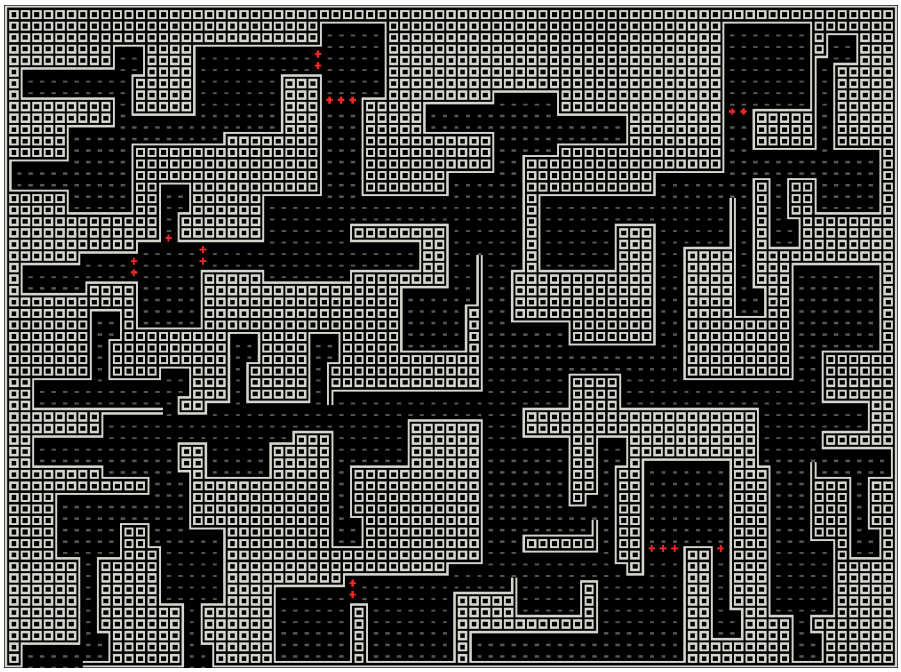


Figura 5.16 – *Digger*, tamanho mínimo 6, tamanho máximo 9, 30 *features*.



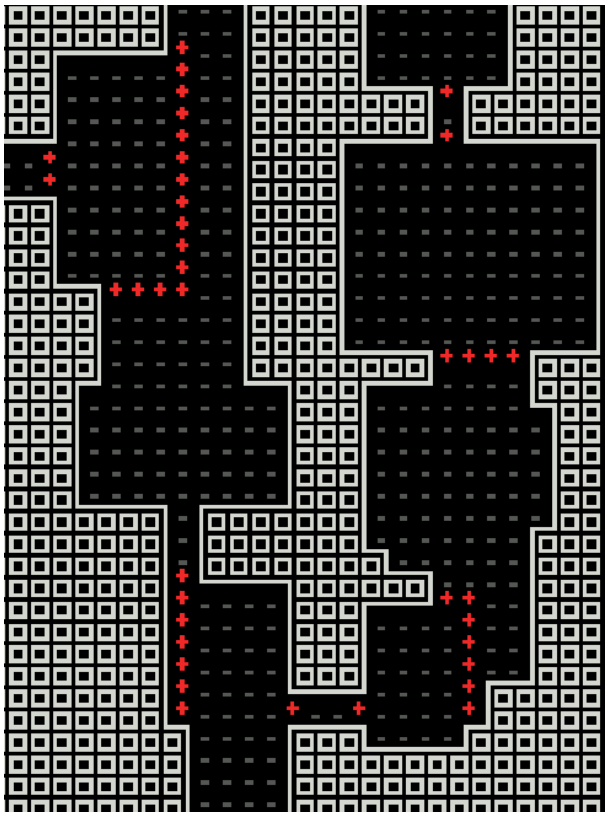
5.5 INSERÇÃO DE PORTAS

Tipicamente, *roguelikes* possuem a maior parte dos corredores com 1 *tile* de largura. Esses corredores são tipicamente utilizados por jogadores como *chokepoints*, nos quais inimigos são atraídos pelo jogador, permitindo que o jogador batalhe com eles um a um, anulando a possibilidade de cercamento em todos as direções. No jogo implementado neste trabalho, busca-se diminuir a quantidade de corredores com 1 *tile* de largura

de modo a mitigar essa estratégia e fomentar o combate à distância, pois áreas maiores propiciam a utilização de táticas mais cautelosas.

No entanto, enquanto é simples inserir portas em corredores com 1 *tile* (basta adicionar uma porta em cada abertura da sala conectada), inserir múltiplas portas para bloquear salas conectadas por corredores mais largos é um tanto mais complexo. Inicialmente, pode-se pensar que a solução para aberturas com 1 *tile* de largura funcionaria perfeitamente para aberturas maiores, mas isso pode acarretar comportamentos anômalos, como pode ser observado na Figura 5.17. Isso ocorre por conta do procedimento de conexão entre as salas: no processo de conexão de um sala com outra, *utilizar corredores maiores aumenta as chances de que o corredor "invada" a área de uma sala em seu caminho, deformando-a*. Como as coordenadas de cada sala são mantidas e a inserção de portas ocorre *após* a conexão, gera-se o efeito observado na Figura 5.17: as portas substituem as paredes "destruídas" pelos corredores.

Figura 5.17 – Comportamento anômalo na inserção de portas.



Para que o efeito anômalo não seja observado, é necessário que portas *não* sejam inseridas em salas onde isso provavelmente ocorreria. Portanto, a solução tem base probabilística, sendo baseada em restrições adicionais. Apesar de simples, a solução descrita na Figura 5.18 é única, não sendo previamente documentada em outros trabalhos.

Em resumo, o algoritmo simplesmente ignora a etapa de inserção de portas em salas com 5 ou mais aberturas. Caso a sala seja uma candidata para a inserção de portas, dois testes adicionais são realizados sequencialmente:

Figura 5.18 – Solução para a inserção de portas em corredores genéricos.

```
// Para cada conjunto de aberturas de cada sala...
for locs in locs_vec.iter() {
    // Se a sala tiver mais de cinco aberturas, ignore.
    if locs.len() <= 5 {
        // Para cada abertura da sala atual...
        for loc in locs.iter() {
            let pt = map.idx_pos(*loc);
            let door_count = // Vizinhos c/ tile de porta (Moore)
                count_neighbor_tile(map, pt, TileType::ClosedDoor, true);
            let wall_count = // Vizinhos c/ tile de parede (von Neumann)
                count_neighbor_tile(map, pt, TileType::Wall, false);
            if door_count >= 2 { continue; }
            if wall_count >= 3 && door_count < 2 { continue; }
            map.tiles[*loc] = Tile::closed_door();
        }
    }
}
```

1. Se a porta a ser inserida tiver 2 ou mais vizinhos do tipo “porta” nas 8 direções possíveis, não inserir a porta.

2. Se a porta a ser inserida tiver 3 ou mais vizinhos do tipo “parede” nas 4 direções possíveis, e menos de 2 vizinhos do tipo “porta” em 8 direções, não inserir a porta

5.6 WAVE FUNCTION COLLAPSE (WFC)

Como mencionado na Subseção 2.3.1.2, a versão do algoritmo WFC deste trabalho (Figuras 5.19 e 5.20) foi adaptada a partir da implementação de Sherratt (2019) do WFC; entretanto, ao invés de atuar sobre valores de pixel, atua-se diretamente sobre os tipos de *tile* – similarmente à implementação do WFC de Wolverson (2019) –, como blocos de parede, água, chão, entre outros. Isso foi feito como forma de contornar o problema de gerar um mapa baseado em *tiles* a partir de uma imagem BMP, o que necessitaria de uma dependência adicional ou de uma solução própria.

Na Figura 5.20, a entrada de tamanho 6x6 *tiles* é particionada por uma janela deslizante de tamanho 2x2, que coleta os padrões contidos na figura. Os padrões são rotacionados e refletidos como etapa de pré-processamento no WFC, e a saída é o resultado obtido a partir da interação entre essas partes, a partir das *regras de adjacência* – dois padrões são compatíveis numa direção x quando os *tiles* adjacentes nessa direção são idênticos.

Figura 5.19 – Exemplo de entrada/saída do WFC implementado (1).

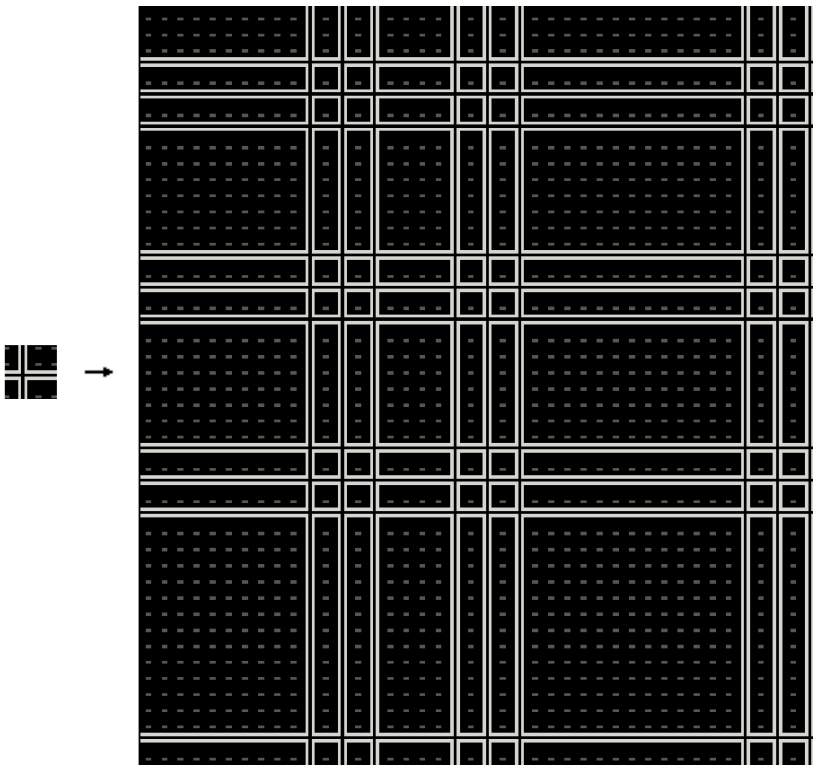
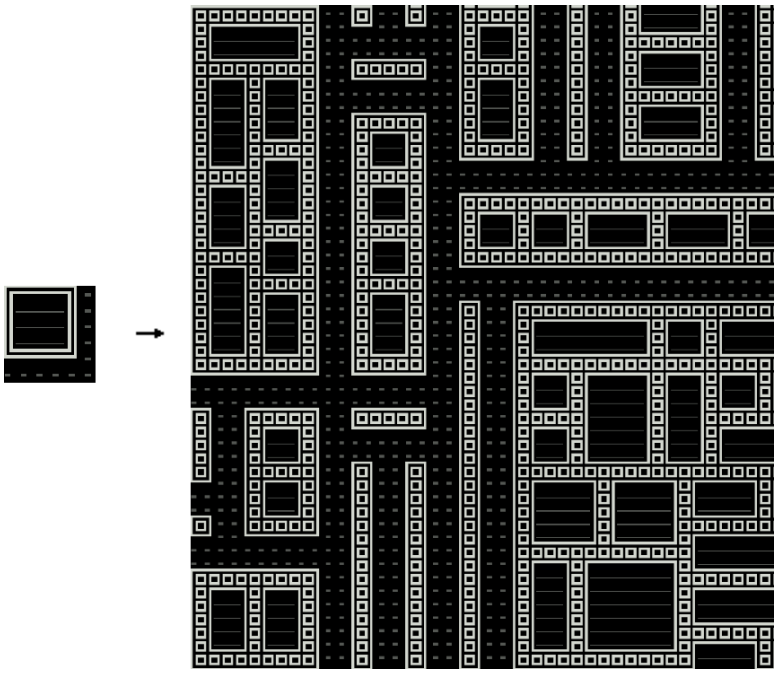


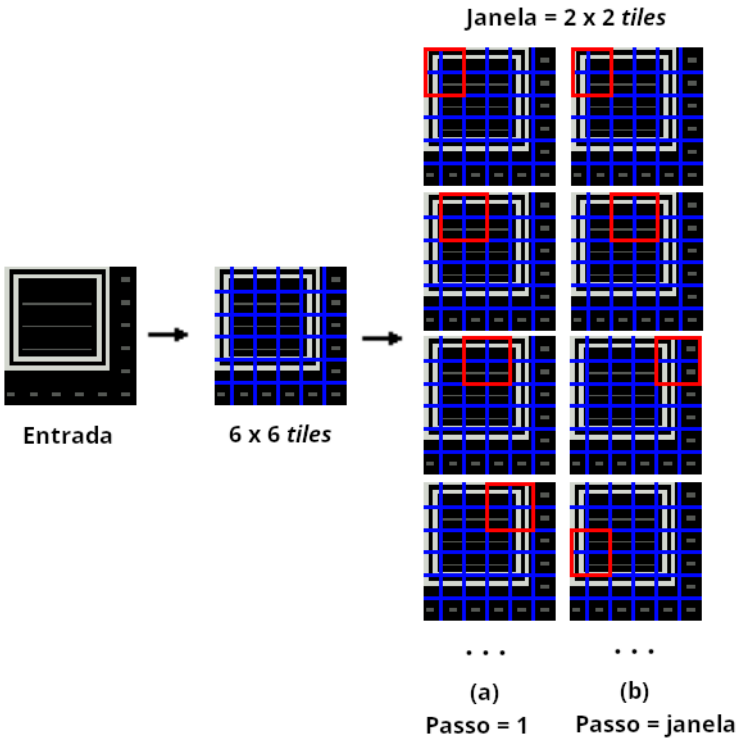
Figura 5.20 – Exemplo de entrada/saída do WFC implementado (2).



A Figura 5.21 descreve o funcionamento da janela deslizante – que percorre a entrada da esquerda para a direita e de cima para baixo – na etapa de coleta de padrões da entrada. Em (a) – modelo que representa o funcionamento padrão do WFC –, a janela avança 1 *tile* a cada iteração, até que toda a entrada seja observada. No modelo (b) – descrito na implementação alternativa do WFC de Wolverson (2019) –, a janela avança a

quantidade de *tiles* equivalente ao seu próprio tamanho a cada iteração. Em ambos os modelos, entrada é repetida em todas a direções (*wrapping*) para suportar casos em que a janela precisa ler os *tiles* das extremidades – nestes casos, a janela sairia dos confins da entrada.

Figura 5.21 – Dois modelos de funcionamento para a janela deslizante no WFC.



É importante notar que o WFC não é uma panaceia. Ao experimentar com o WFC no *pipeline* de geração de *Caves of Qud*, Bucklew (2019) encontrou algumas desvantagens:

1. Homogenia: não existe uma grande estrutura inerente (por exemplo, uma aldeia de casas retangulares pode não ter uma grande catedral).
   Solução: como etapa de pré-processamento, selecione uma grande região no mapa e execute o WFC dentro dela para gerar a arquitetura interna.

2. *Overfitting*: adicionar muitos detalhes frequentemente resulta em *overfitting* em pequenos detalhes, reduzindo a variabilidade da saída.
   Solução: inserir os pequenos detalhes (portas, móveis etc.) como etapas de pós-processamento ao invés de adicioná-los na entrada.

3. Falta de conectividade: não há como garantir a conectividade entre as regiões.
   Solução: executar um algoritmo de conexão como procedimento de pós-processamento.

Em comparação com este trabalho, todos os problemas acima foram mais proeminentes quando entradas pequenas e tamanhos de *tiles* menores foram escolhidos para

particionar a entrada. Ao utilizar entradas grandes (e.g. 20x20) e detalhadas para o WFC com o modelo (a) da janela deslizante, o algoritmo passa a falhar mais frequentemente (contradição) por conta da grande quantidade de padrões, ou, quando não há falha, a saída pode conter descontinuidades ou baixa similaridade com a entrada. Uma solução encontrada foi implementar o modelo (b) da janela deslizante, no qual o número de padrões é diminuído, mas requer uma janela de tamanho maior para manter a similaridade com a entrada.

Apesar da utilização limitada, o modelo (b) é uma maneira de contornar as limitações do WFC em relação a grandes entradas. O modelo (b) também oferece desempenho superior ao modelo (a), como descrito na Subseção 5.10.1. Uma comparação de saídas utilizando os dos modelos de janela deslizante podem ser observados na Figura 5.22; observe como em entradas maiores com grande número de detalhes o modelo (a) com tamanho de janela pequeno não oferece uma saída satisfatória. Na Figura 5.23 o tamanho da janela para o modelo (a) é aumentado; no entanto, ainda que a qualidade da saída seja melhorada, o modelo (b) oferece uma saída com uma quantidade menor de descontinuidades e, neste caso, menos padrões repetidos e boa similaridade com a entrada – como o desempenho do modelo (b) é superior, este é um caso de utilização ideal desse modelo.

Figura 5.22 – Saídas do WFC em uma mesma entrada utilizando os modelos (a) e (b) da janela deslizante (1).

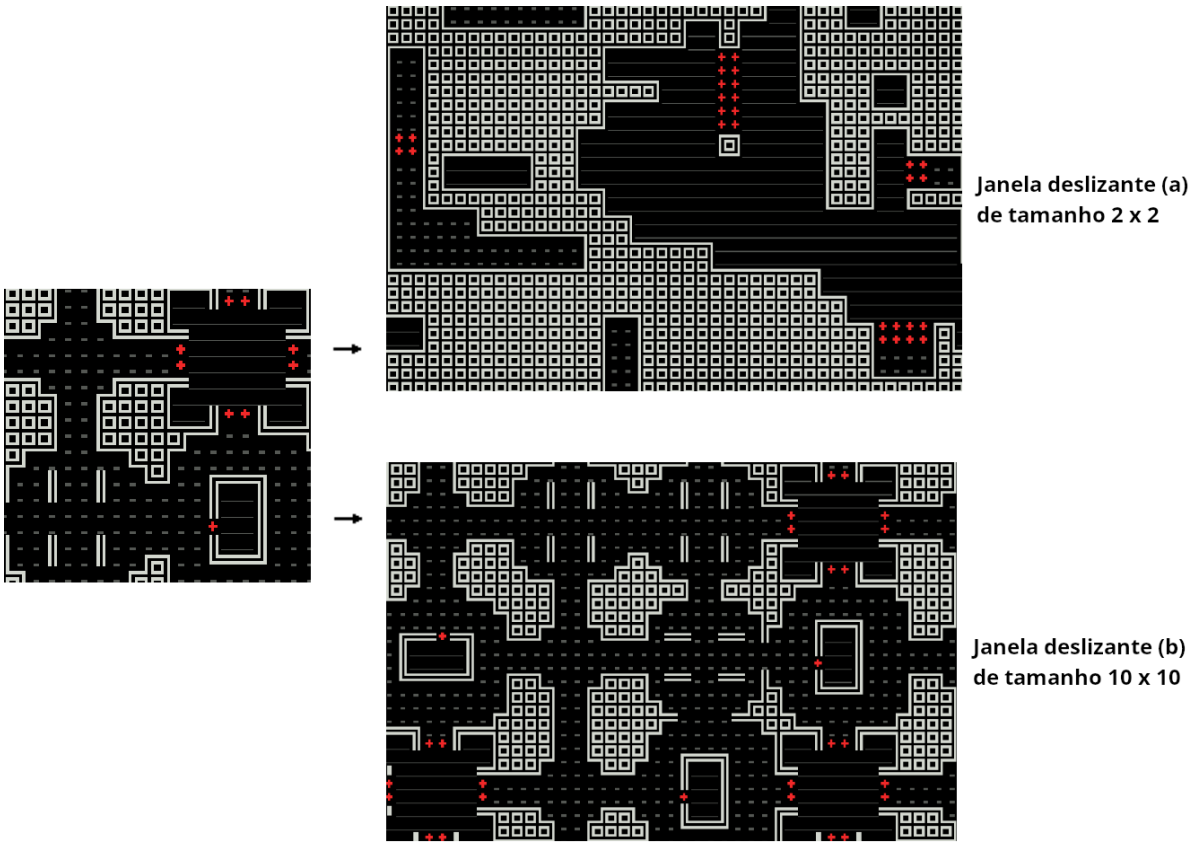
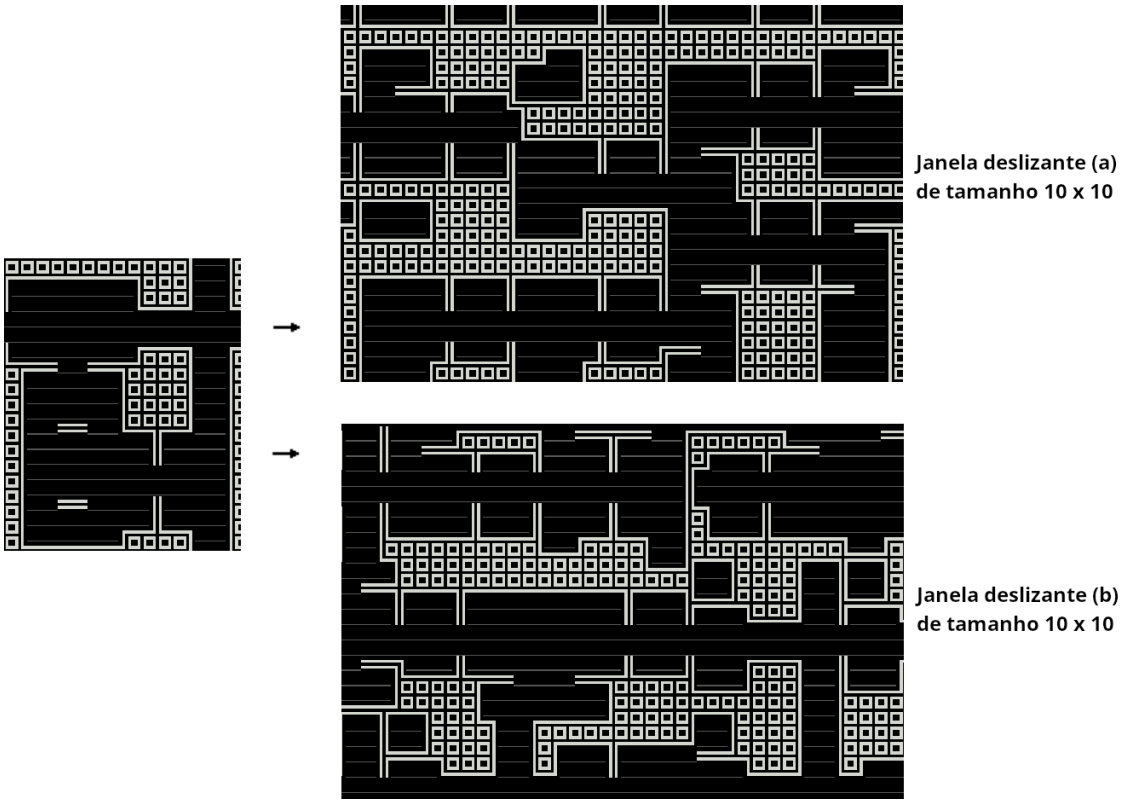
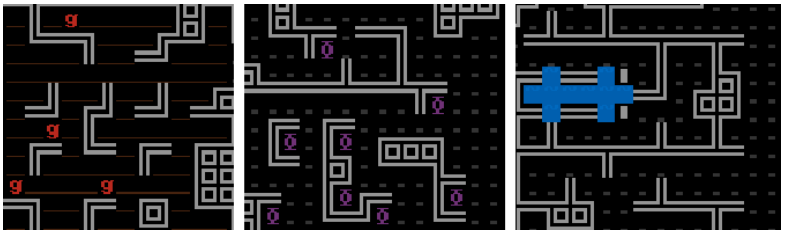


Figura 5.23 – Saídas do WFC em uma mesma entrada utilizando os modelos (a) e (b) da janela deslizante (2).



Por causa de tais problemas, o WFC (no contexto de jogos 2D com visão de cima para baixo) pode ser melhor utilizado como um algoritmo para gerar arquiteturas internas (Figura 5.24) ao invés de estruturas grandes e complexas. Isso reforça a noção de que o WFC é melhor utilizado quando combinado com outros algoritmos de PDG, ou seja, o WFC pode ser utilizado como apenas mais uma etapa no processo de geração. A escolha de utilização do modelo (a) ou (b) da janela deslizante depende unicamente da entrada; em alguns casos, o modelo (a) pode retornar resultados melhores, enquanto em outros, o modelo (b) pode ser uma melhor escolha.

Figura 5.24 – WFC convencional utilizado como arquitetura interna para salas vazias geradas previamente.

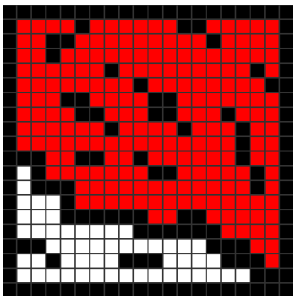


A contribuição do WFC no *pipeline* de geração de ambientes é notável: seria excepcionalmente complexo projetar algoritmos de geração procedural que produzissem saídas como as observadas nas Figuras 5.20, 5.22, 5.23 e 5.24.

5.7 CONECTANDO REGIÕES ISOLADAS

Muitos dos algoritmos utilizados (Quadro 2.1) não garantem conectividade total e, ao combinar diferentes métodos de geração, há um risco maior de criar regiões inacessíveis. O *problema de conectividade* sempre pode ser resolvido como um procedimento de pós-processamento; no caso deste trabalho, é utilizado o algoritmo *flood fill* (Figura 5.25) oito-conectado para detectar regiões isoladas no mapa. O *flood fill* atua como a ferramenta de "balde de tinta" em editores de imagens.

Figura 5.25 – Algoritmo de *flood fill*. A área em vermelho é totalmente isolada da área branca.



Fonte: <https://benasb.github.io/FloodFill-Visualization/index.html>.

Após coletar as diferentes regiões em uma lista, detectam-se os pontos fronteiriços – anteriores aos *tiles* de separação – de cada região, e são escolhidos os dois pontos com menor distância (em caso de empate, considera-se a primeira distância calculada) entre duas regiões; então, um túnel de conexão é esculpido no mapa entre esses dois pontos. Como os túneis podem parecer visualmente não naturais em ambientes de cavernas, é possível, ao fazer o túnel, esculpir fora dos limites do mesmo e, opcionalmente, aplicar uma única iteração de autômatos celulares para suavizá-lo. Nas Figuras 5.26 e 5.27, estão marcadas em vermelho as regiões onde foram esculpidos túneis de conexão entre regiões.

5.8 COMBINAÇÕES DE TÉCNICAS

Neste trabalho, uma estrutura *Map* representa todas as informações visuais contidas em uma *dungeon*, armazenando informações de *tiles* e entidades (jogadore, *mobs* e itens). Um mapa pode ser criado usando a estrutura *Map Generator*, que por sua vez pode invocar os vários *pipelines* de geração implementados.

Pipelines de base são usados para gerar ambientes "básicos" utilizando os vários algoritmos discutidos nesta seção. *Pipelines específicos* são funções que utilizam os *pipelines* de base para gerar ambientes mais complexos, provenientes da combinação dos algoritmos, como observado nas subseções a seguir.

Figura 5.26 – Conexões visualmente artificiais entre regiões isoladas.

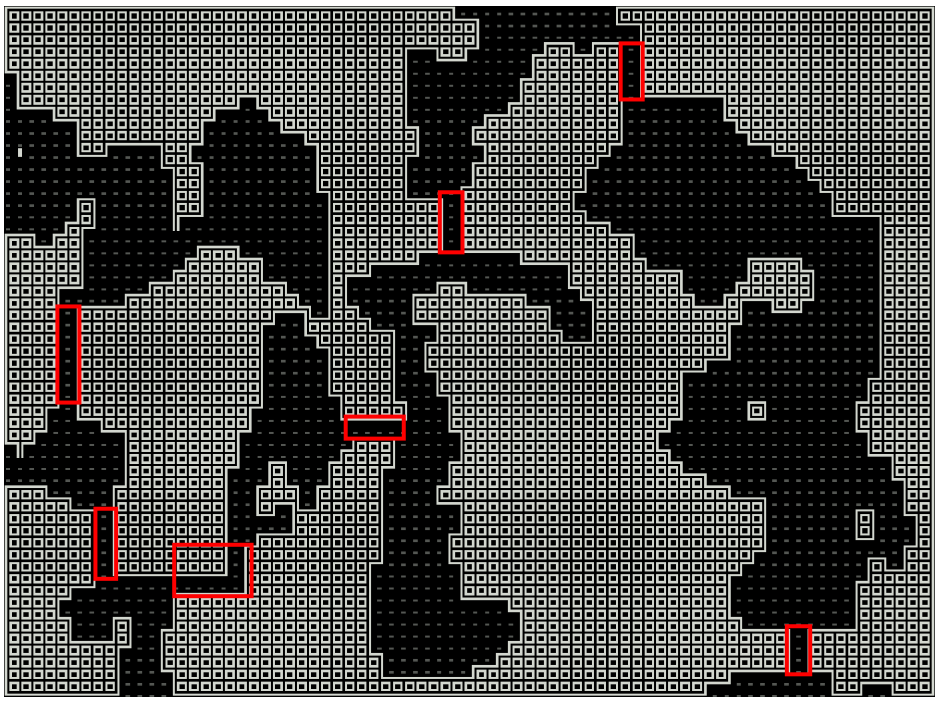
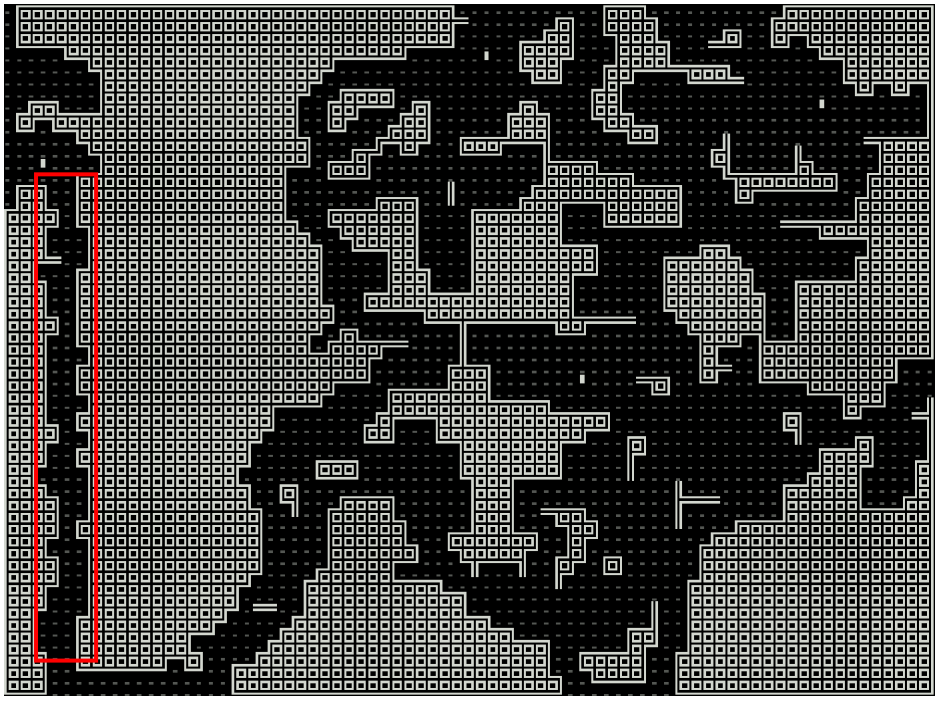


Figura 5.27 – Conexão visualmente natural entre duas regiões isoladas.



5.8.1 A sinergia dos autômatos celulares

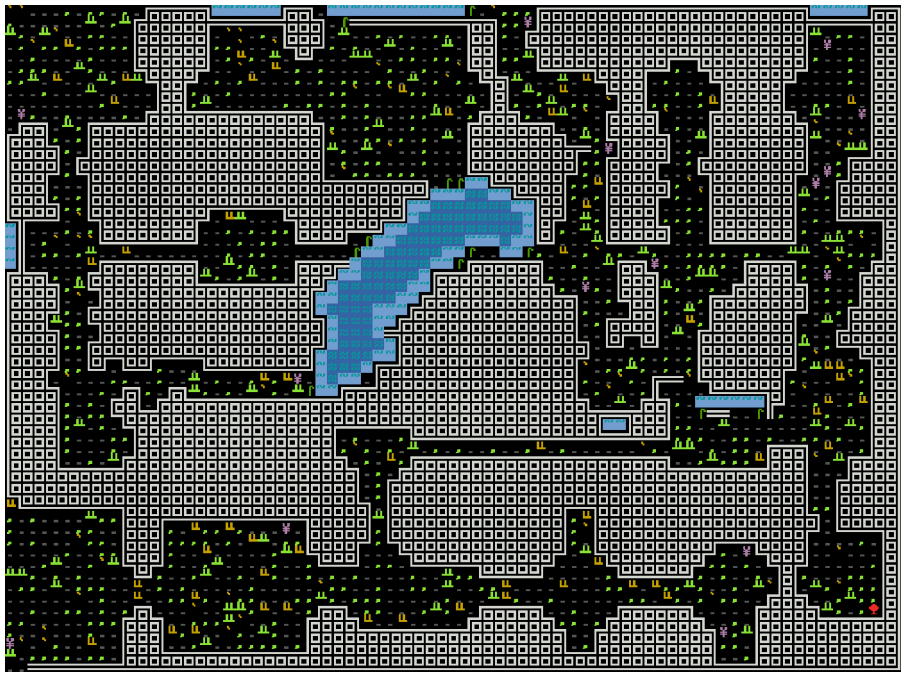
Na Seção 5.2, foi visto como autômatos celulares podem associar-se ao passeio aleatório para gerar sistemas de cavernas; no entanto, nesta seção é visto como ele pode ser combinado com essencialmente qualquer algoritmo de geração de ambientes para gerar resultados esteticamente singulares. Como uma das consequências do uso de autôma-

tos celulares consiste em gerar ambientes visualmente orgânicos, pode-se experimentar utilizá-lo em *dungeons* TDML geradas a partir dos métodos BSP e *Digger*. Até onde se sabe, as sinergias descritas nesta seção não foram previamente documentadas em outros trabalhos.

5.8.1.1 BSP Dungeon + autômatos celulares

Enquanto a combinação de um ambiente visualmente artificial e pouco caótico com autômatos celulares possa parecer inválida, pode-se gerar ambientes singulares que assemelham-se a construções em ruínas (Figuras 5.28 e 5.29). Como o jogo implementado neste trabalho possui, de fato, uma estética pós-apocalíptica onde ruínas são ambientes comuns, a combinação de *BSP Dungeon* com autômatos celulares gera ambientes proveitosos.

Figura 5.28 – Ruínas BSP (1).

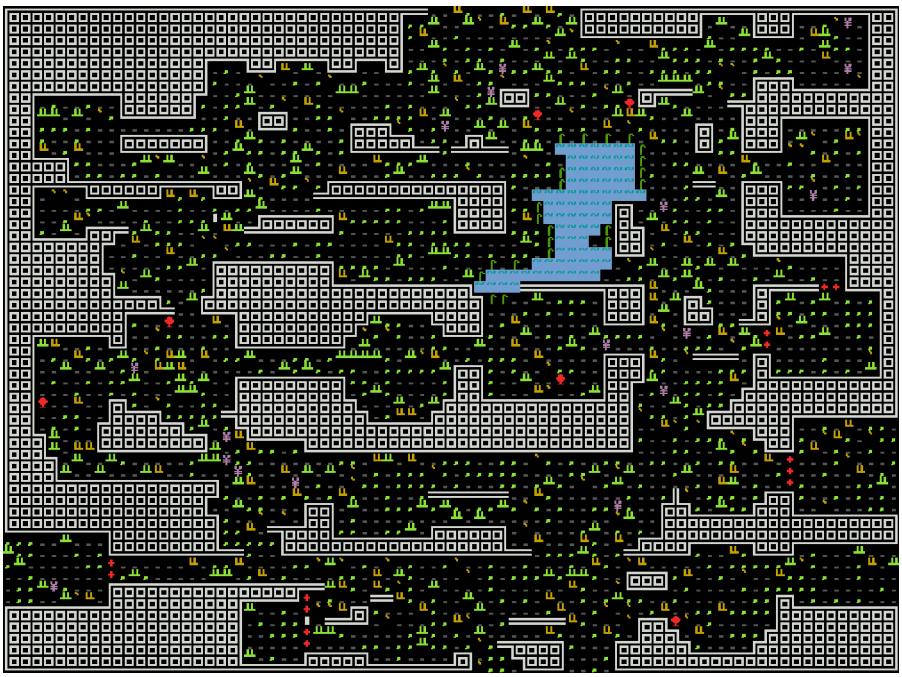


Ambientes similares ao apresentado na Figuras 5.28 podem ser gerados a partir do seguinte procedimento:

1. Gera-se uma *dungeon* BSP aleatória.
2. Cria-se uma quantidade arbitrária de lagos.
3. Aplicam-se apenas duas iterações de autômatos celulares.

Para gerar um ambiente similar ao da Figura 5.29, ocorre uma inversão:

Figura 5.29 – Ruínas BSP (2).



1. Gera-se uma caverna utilizando passeio aleatório juntamente com autômatos celulares (12 iterações).

2. Gera-se uma *dungeon* BSP aleatória.

3. Cria-se uma quantidade arbitrária de lagos.

Em ambos os casos, a vegetação é inserida no último passo; como detalhe especial, note também a inserção de vegetação ciliar ao redor dos lagos. Diferentes configurações específicas – informadas por parâmetro – de cada algoritmo podem ser experimentadas, e podem variar de implementação para implementação.

5.8.1.2 Digger + autômatos celulares

Combinando-se uma *dungeon* gerada a partir do método *Digger* com iterações especificamente configuradas de autômatos celulares, pode-se gerar *diggers invertidos*, como observado nas Figuras 5.30 e 5.31.

O procedimento utilizado para gerar ambos os ambientes é o mesmo.

1. Gera-se uma *dungeon Digger* aleatória.

2. Aplicam-se três iterações de autômatos celulares, com a seguinte regra específica: *transformar o tile atual em "parede" quando 7 ou mais vizinhos sejam do tipo "parede", ou quando não há vizinhos do tipo "parede"*.

Figura 5.30 – *Digger* invertido.

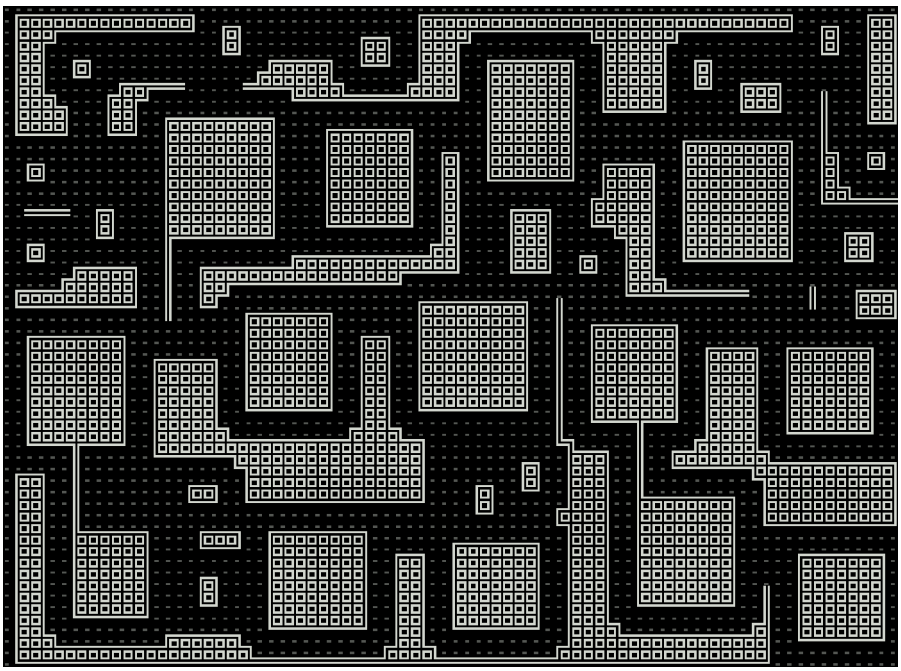
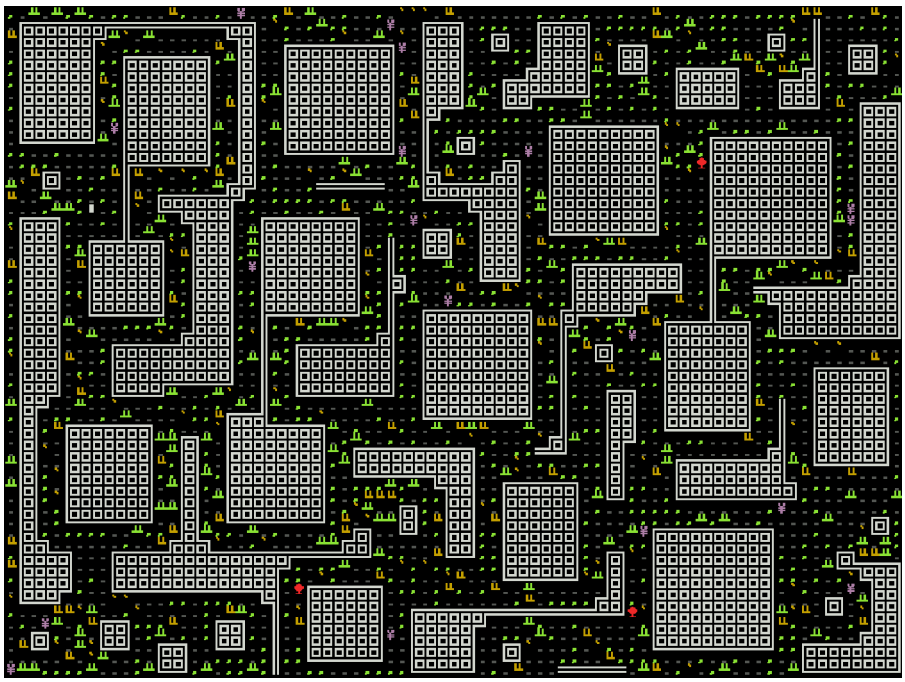


Figura 5.31 – *Digger* invertido com adição de vegetação.



Como consequência da regra específica em (2), grandes zonas vazias (salas) são preenchidas, o que gera o efeito de *digger invertido*. Quanto menor o número de vizinhos “parede” requeridos na regra, menor é o efeito de inversão.

5.8.2 Geração localizada

Enquanto é possível gerar *dungeons* estratégicas e esteticamente agradáveis utilizando um único *pipeline* de geração sobre o mapa inteiro, é possível gerar resultados mais diversos ao selecionar diferentes regiões de um mapa e aplicar procedimentos de geração distintos sobre cada uma delas, como pode ser observado na Figura 5.32 e nas outras figuras desta subseção.

Figura 5.32 – BSP (rosa) + Floresta (vermelho) + Ruínas BSP (amarelo).

Como mencionado na Seção 5.6, o WFC pode ser melhor utilizado em conjunto com outros algoritmos, seja como arquitetura interna ou externa. A Figura 5.33 contém um exemplo do WFC sendo utilizado como arquitetura interna. Ao utilizar "geração localizada", entretanto, o problema da conectividade é ainda mais aparente, e seria inviável caso não fosse apresentada uma solução viável como descrito na Seção 5.7; na mesma figura, é possível notar os túneis esculpidos entre as diferentes regiões – assegurando a conectividade total no mapa. Outras *dungeons* geradas a partir de geração localizada podem ser visualizadas nas Figuras 5.34, 5.35, 5.36 e 5.37.

A Figura 5.37 contém uma estrutura comumente chamada de *prefab* (pré-fabricado): uma seção do mapa criada manualmente pelo programador e inserida aleatoriamente no mapa. O *roguelike Dungeon Crawl Stone Soup*[3] faz extenso uso de centenas de seções pré-fabricadas durante o processo de geração de um mapa, tornando-o visualmente complexo. Entretanto, o jogo implementado neste trabalho visa minimizar o uso de pré-fabricados, utilizando, portanto, o WFC – neste caso, o modelo (b) com janelas maiores.

[3] Disponível online em <https://crawl.develz.org/>.

Figura 5.33 – Combinação utilizando WFC na região central (vermelho).

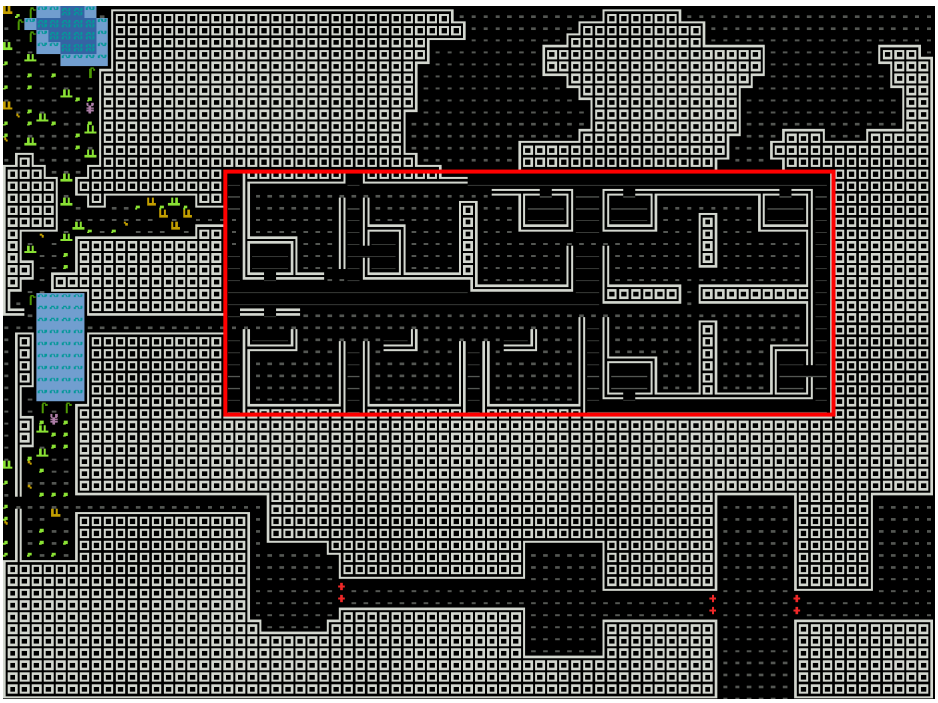


Figura 5.34 – Digger invertido (rosa) + WFC (vermelho) + Ruínas BSP (amarelo).

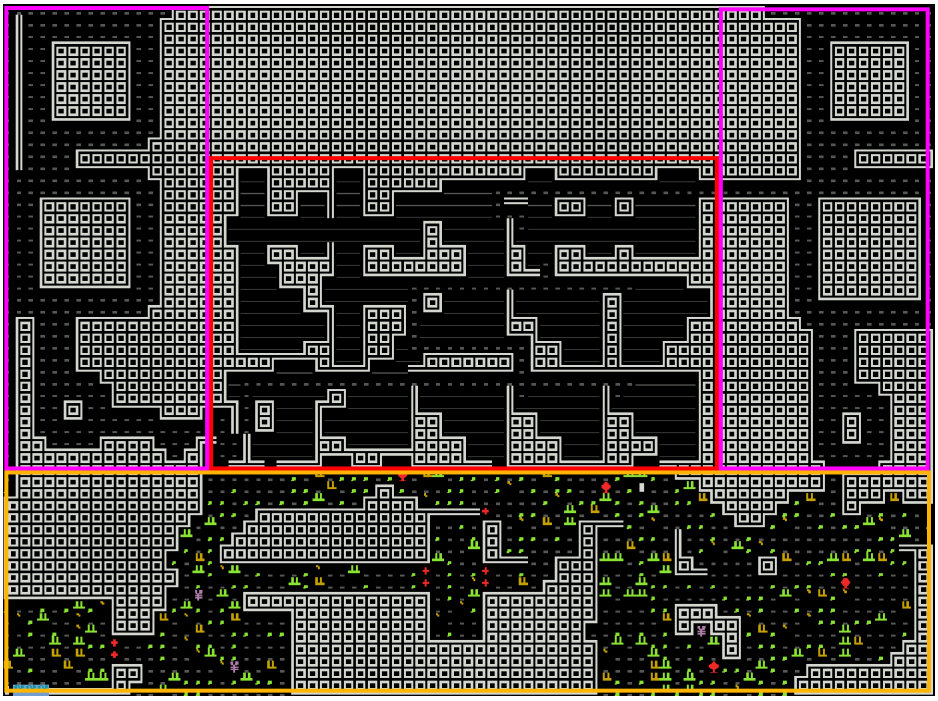


Figura 5.35 – Digger invertido (rosa) + BSP com arquitetura interna gerada via WFC (vermelho) + Ruínas BSP (amarelo) + Cavernas geradas via combinação de passeio aleatório com autômatos celulares (verde).

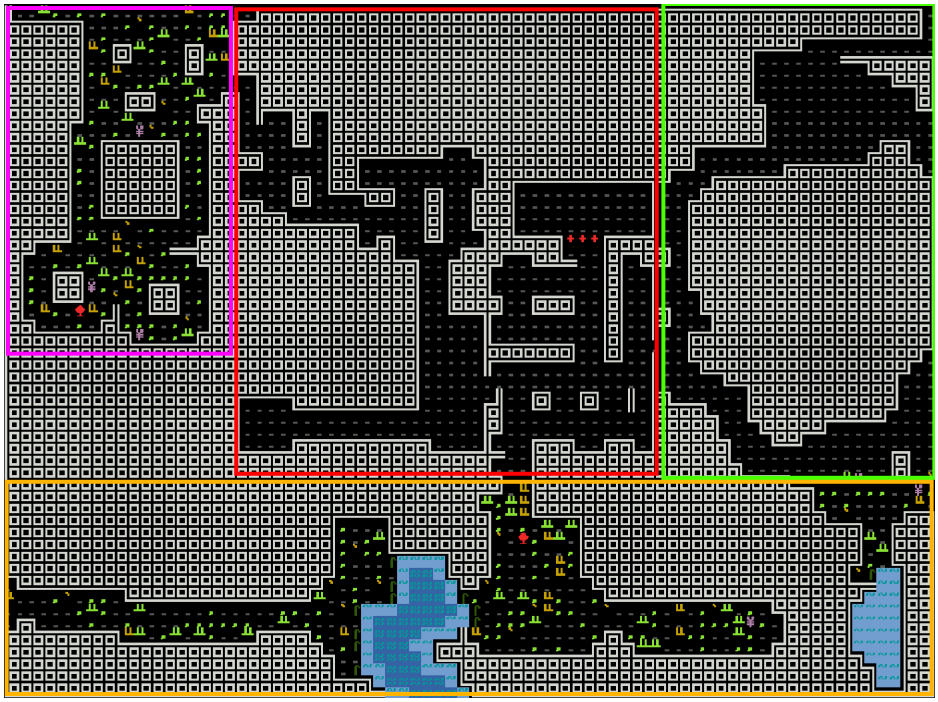


Figura 5.36 – Floresta gerada via autômatos celulares (amarelo) + Arquitetura externa gerada via WFC modelo (b) (vermelho) + BSP com arquitetura interna das salas gerada via WFC modelo (a) (rosa) + ruínas BSP (verde).

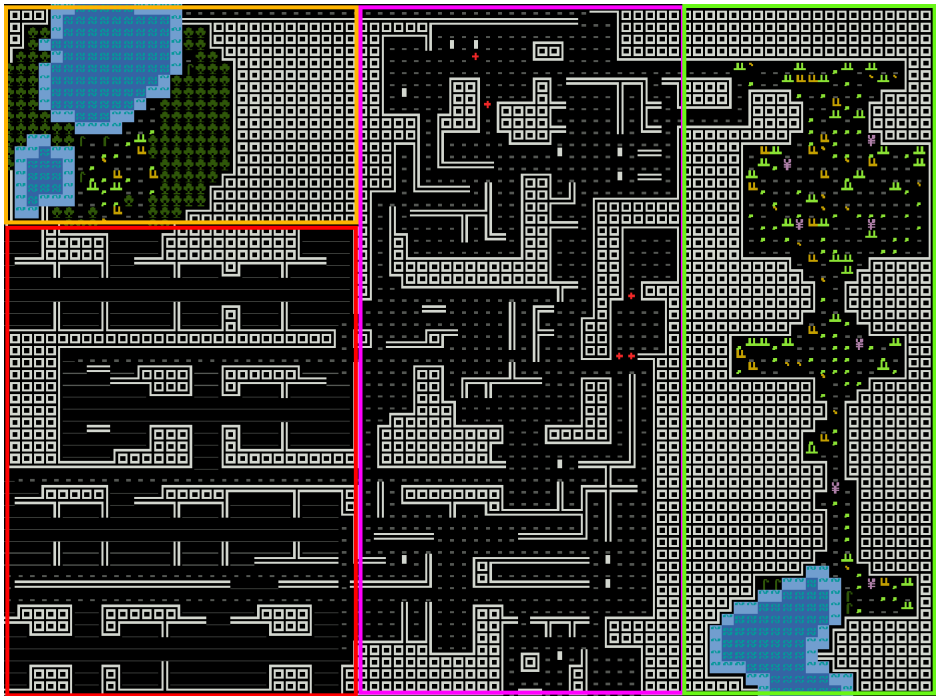
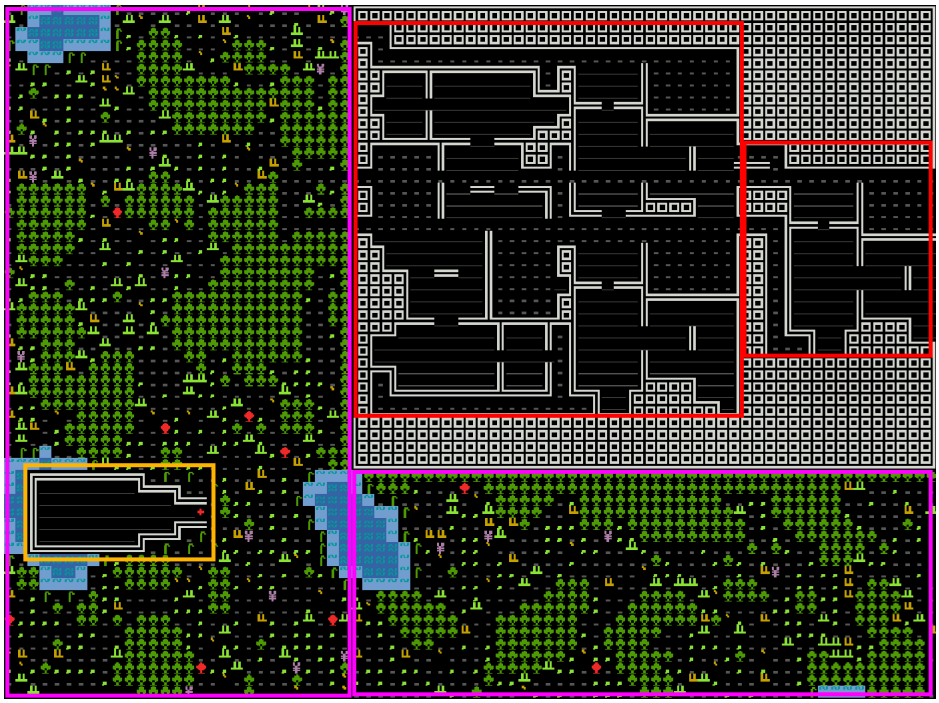


Figura 5.37 – Floresta gerada via autômatos celulares (rosa) + seção pré-fabricada (amarelo) + WFC (vermelho).



5.9 APLICAÇÕES DE CADA ALGORITMO

Nesta seção, são analisadas as aplicações de cada algoritmo de PDG no contexto do jogo implementado neste trabalho, como, por exemplo, para que tipo de ambiente cada um pode ser utilizado. Boa parte das conclusões aqui presentes são resultados provenientes dos experimentos descritos nas seções anteriores. Porém, devido à quantidade de possíveis aplicações e ao escopo deste trabalho, nem todas as possibilidades previstas de cada algoritmo foram diretamente aplicadas.

- **Passeio Aleatório**
  - Sistemas de cavernas com posterior suavização utilizando iterações de autômatos celulares.
  - Simulação de escavações.
  - Caso modificado para que uma direção seja priorizada em detrimento das outras, podem ser criados córregos e trilhas.
- **Autômatos Celulares**
  - Florestas e sistemas de cavernas.
  - Aplicações de regras adicionais para modificar regiões específicas do ambiente (e.g. regras para criar *tiles* de água profunda em lagos).

  - Suavização e simulação de erosão em ambientes. Por exemplo, a passagem do tempo pode transformar estruturas geradas a partir de *BSP* ou *Digger* em ruínas.
  - Alto fator sinérgico: autômatos celulares podem ser combinados com essencialmente todos os algoritmos, gerando resultados distintos e potencialmente inesperados.

- **BSP Dungeon**
  - Criação de ambientes artificiais – salas conectadas por corredores. Dependendo da ordem de conexão entre as salas, o nível de linearidade de um ambiente é alterado. Por exemplo, uma *dungeon* onde as salas são conectadas ordenadamente da esquerda para a direita gera um caminho muito mais linear e com menos ramificações do que um *dungeon* com salas conectadas aleatoriamente.
  - Pulando-se a etapa de conexão, pode ser utilizado para gerar pequenas aldeias com casas.
  - Pode ser facilmente utilizado em conjunto com outros algoritmos.

- **Digger**
  - Similarmente ao *BSP Dungeon*, gera salas conectadas por corredores.
  - Ramificações são feitas a partir de uma sala central (“tentáculos de polvo”) de tamanho arbitrário, na qual todos os caminhos convergem.
  - Ideal para mapas que possuem um *hub* central.
  - Em uma possível etapa de pós-processamento, os corredores podem ser removidos com o intuito de simular uma aldeia que possui uma grande (ou pequena) estrutura central, como uma catedral.

- **WaveFunctionCollapse**
  - Utilizado em maior grau para gerar arquitetura interna. Por exemplo, salas geradas por *BSP* e *Digger* são completamente vazias em termo de estruturas; o WFC pode atuar dentro dessas salas para criar padrões e preenchê-las.
  - Como arquitetura externa, pode substituir boa parte das estruturas pré-fabricadas inseridas no mapa, gerando maior variação.
  - Como o WFC aceita qualquer tipo de *tile*, mobs podem ser inseridos em posições diversas na entrada. Desse modo, o algoritmo pode atuar como sistema de *spawn* simples, porém eficaz (MININI; ASSUNÇÃO, 2020).

- Pode ser utilizado para fins estéticos no mapa (e.g. inserção de pequenos jardins).

Enquanto em muitos jogos – especialmente *roguelikes* – basta elaborar um único algoritmo para geração de *dungeons*, como, na maioria das vezes, algoritmos baseados em BSP e inserção de estruturas pré-fabricadas, é pouco convencional utilizar diversos algoritmos num mesmo mapa. Muitas sinergias podem ser encontradas no processo de criação de *pipelines* híbridos (utilizando diversos algoritmos em sequência) de geração de *dungeons*; infelizmente, tais sinergias não são normalmente documentadas como realizado neste trabalho.

5.10 DESEMPENHO

Esta seção apresenta resultados de desempenho referentes à maioria dos procedimentos de geração de *dungeons* apresentados nos capítulos anteriores. Todos os testes foram realizados em um computador com o sistema operacional Arch Linux x64, 16 GiB de memória principal e processador AMD Ryzen 5 3600, de 3.6 GHz; Cada *pipeline* de geração foi executado 30 vezes, e o programa foi executado utilizando otimizações provenientes do comando `$ cargo run -release`.

5.10.1 Algoritmos individuais

Na Tabela 5.1, encontram-se os tempos de geração para *dungeons* de tamanhos variados geradas a partir dos algoritmos individuais explorados anteriormente. Somam-se aos tempos – quando aplicáveis – os processos de inserção de portas e conexões entre regiões isoladas. Em ambos os modelos de janela deslizante do WFC, foi utilizada uma entrada de tamanho 20x20 e uma janela de tamanho 10x10 – o modelo (a) utiliza passo de tamanho 1 *tile* para a janela deslizante na etapa de coleta de padrões, enquanto o modelo (b) utiliza passo igual ao tamanho da janela em *tiles*.

Expandindo a Tabela 5.1, a Figura 5.38 contém um gráfico elaborado a partir dos dados presentes na tabela, referente ao crescimento em tempo de geração de cada algoritmo em relação ao tamanho do mapa a ser gerado.

A partir da análise do gráfico, percebe-se que o algoritmo WFC utilizando janela deslizante de passo 1 (a) é consistentemente o algoritmo com menor desempenho, diante da grande quantidade de padrões coletados da entrada. Como inicialmente cada célula do WFC pode aceitar todos os padrões possíveis, o tempo para o “colapso” de cada célula é substancialmente maior do que o observado no WFC (b), que possui uma menor

Tabela 5.1 – Tempos de geração de algoritmos individuais em tamanhos de mapa variados.

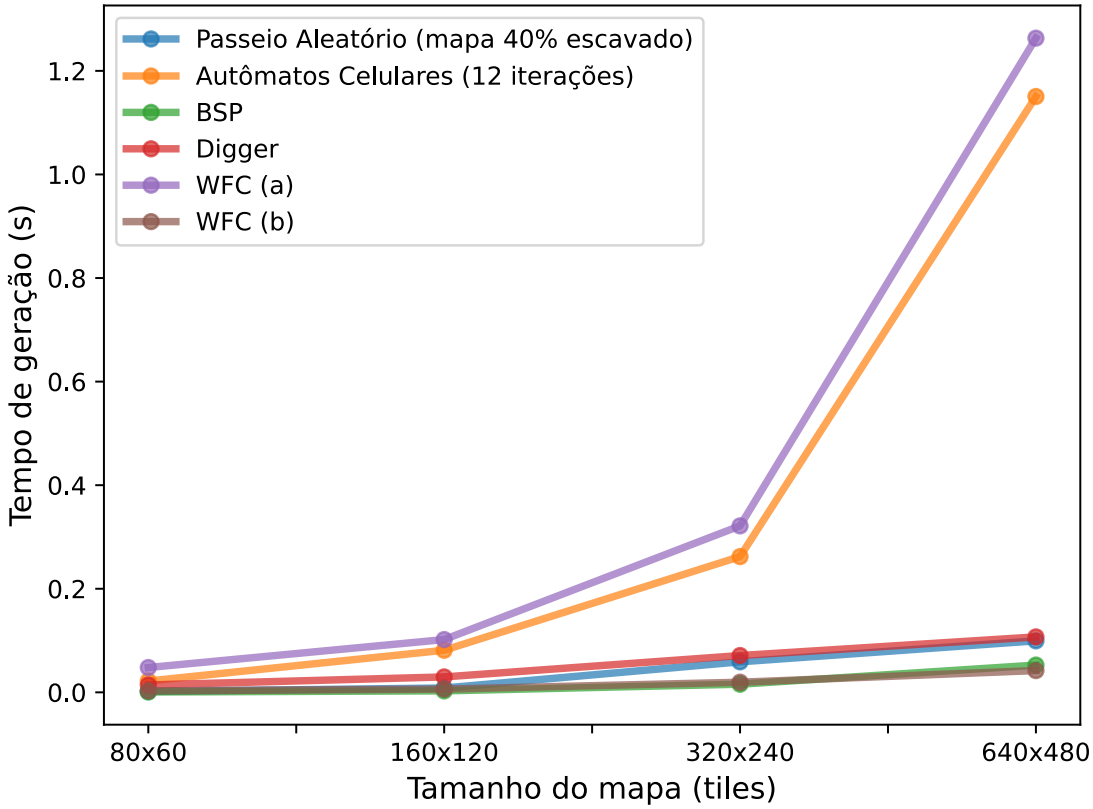
Tamanho	Algoritmo	Tempo
80x60	Passeio Aleatório (mapa 40% escavado)	1,71 ms
	Autômatos Celulares (12 iterações)	21,86 ms
	BSP	0,68 ms
	Digger	14,25 ms
	WFC (a)	**47,90 ms**
	WFC (b)	2,92 ms
160x120	Passeio Aleatório (mapa 40% escavado)	8,34 ms
	Autômatos Celulares (12 iterações)	81,26 ms
	BSP	2,83 ms
	Digger	29,76 ms
	WFC (a)	**101,89 ms**
	WFC (b)	6,19 ms
320x240	Passeio Aleatório (mapa 40% escavado)	58,91 ms
	Autômatos Celulares (12 iterações)	262,12 ms
	BSP	15,70 ms
	Digger	71,04 ms
	WFC (a)	**321,44 ms**
	WFC (b)	19,53 ms
640x480	Passeio Aleatório (mapa 40% escavado)	99,46 ms
	Autômatos Celulares (12 iterações)	1150,49 ms
	BSP	52,60 ms
	Digger	107,05 ms
	WFC (a)	**1263,13 ms**
	WFC (b)	42,02 ms

quantidade de padrões gerados, por conta da janela deslizante de maior passo.

Além disso, nota-se que um maior uso de autômatos celulares (CA) acarreta, geralmente, em maiores tempos de geração. Isso ocorre pois a cada iteração todos os vizinhos de cada *tile* do mapa e ele mesmo são verificados ($largura \times altura \times 9$). Em 12 iterações, utilizando vizinhança de Moore (8 direções) e considerando um mapa de dimensões 80x60, há $12 \times (80 \times 60 \times 9) = 518.400$ verificações de *tiles*.

Em contrapartida, os outros algoritmos têm seus tempos de geração relativamente estáveis à medida que o tamanho do mapa cresce. Espera-se, em *pipelines* de geração híbridos, que os tempos de geração relativos ao WFC (a) e autômatos celulares exerçam dominância sobre os outros algoritmos utilizados no *pipeline*, aumentando o tempo de geração.

Figura 5.38 – Relação entre tamanho do mapa e tempo de geração para algoritmos individuais.



5.10.2 *Pipelines* de geração híbridos

Na Tabela 5.2, encontram-se os resultados de desempenho para alguns *pipelines* híbridos de geração de *dungeons* utilizados no protótipo de jogo desenvolvido. Nessa tabela, os algoritmos analisados na Tabela 5.1 são combinados para gerar mapas de maior complexidade, como aqueles explorados na Seção 5.8.

Igualmente à subseção anterior, a Figura 5.39 expande sobre a Tabela 5.2, demonstrando a relação entre o tempo de geração de um mapa com o tamanho deste. Como esperado, *pipelines* que fazem maior uso do WFC (a) e de múltiplas iterações de autômatos celulares possuem desempenho inferior dado o crescimento da *dungeon*.

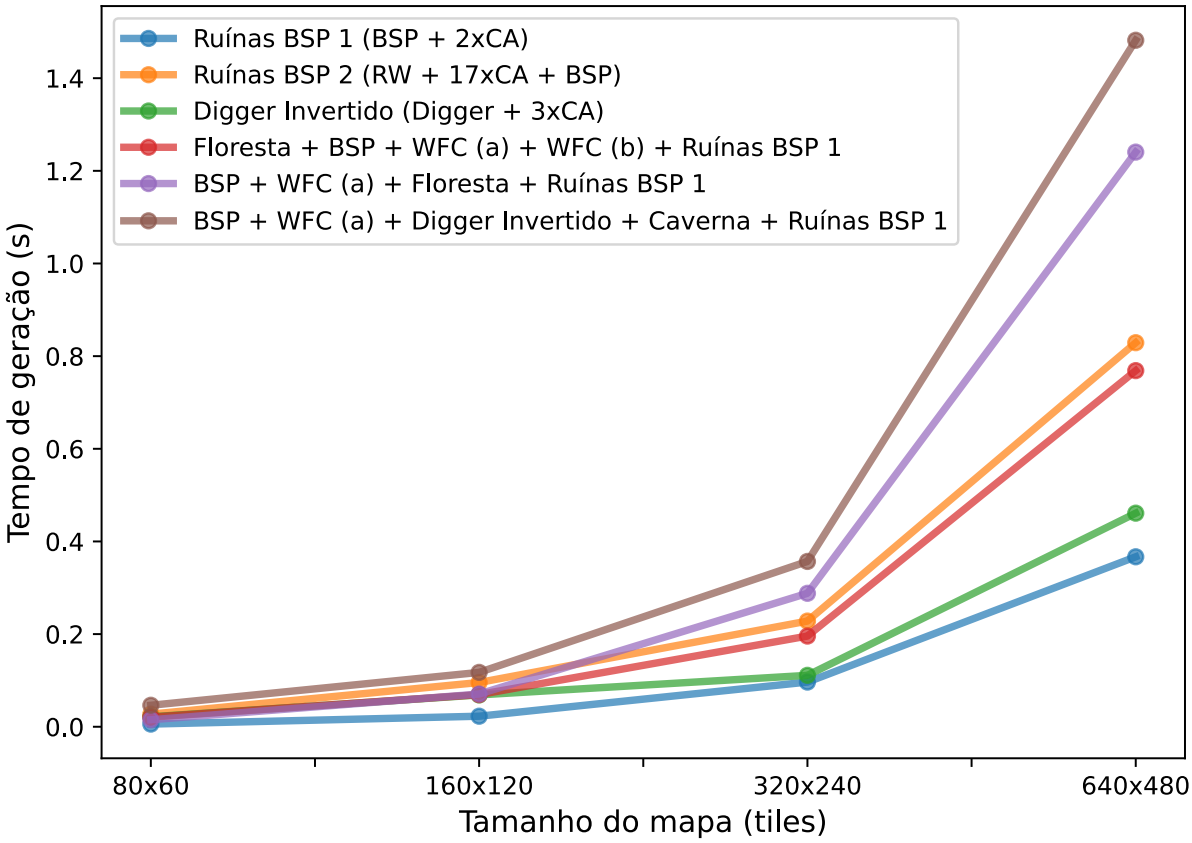
A maioria dos *roguelikes* tradicionais possui mapas de tamanho 80x60 *tiles*, raramente ultrapassando 200x200. Para tamanhos medianos, os *pipelines* de geração possuem um tempo de geração na faixa dos milissegundos, tornando o processo praticamente imperceptível ao jogador – o que comprova a eficiência dos métodos utilizados.

Tabela 5.2 – Tempos de geração de *pipelines* híbridos em tamanhos de mapa variados.

Tamanho	*Pipeline*	Tempo
80x60	Ruínas BSP 1 (BSP + 2xCA)	5,75 ms
	Ruínas BSP 2 (RW + 17xCA + BSP)	27,21 ms
	Digger Invertido (Digger + 3xCA)	21,41 ms
	Floresta CA + BSP + WFC (a) + WFC (b) + Ruínas BSP 1	20,62 ms
	BSP + WFC (a) + Floresta CA + Ruínas BSP 1	15,10 ms
	BSP + WFC (a) + Digger Invertido + Caverna + Ruínas BSP 1	**46,34 ms**
160x120	Ruínas BSP 1 (BSP + 2xCA)	22,75 ms
	Ruínas BSP 2 (RW + 17xCA + BSP)	95,51 ms
	Digger Invertido (Digger + 3xCA)	68,72 ms
	Floresta CA + BSP + WFC (a) + WFC (b) + Ruínas BSP 1	69,40 ms
	BSP + WFC (a) + Floresta CA + Ruínas BSP 1	70,62 ms
	BSP + WFC (a) + Digger Invertido + Caverna + Ruínas BSP 1	**117,34 ms**
320x240	Ruínas BSP 1 (BSP + 2xCA)	96,43 ms
	Ruínas BSP 2 (RW + 17xCA + BSP)	228,10 ms
	Digger Invertido (Digger + 3xCA)	110,96 ms
	Floresta CA + BSP + WFC (a) + WFC (b) + Ruínas BSP 1	195,75 ms
	BSP + WFC (a) + Floresta CA + Ruínas BSP 1	288,05 ms
	BSP + WFC (a) + Digger Invertido + Caverna + Ruínas BSP 1	**356,84 ms**
640x480	Ruínas BSP 1 (BSP + 2xCA)	366,90 ms
	Ruínas BSP 2 (RW + 17xCA + BSP)	829,04 ms
	Digger Invertido (Digger + 3xCA)	461,04 ms
	Floresta CA + BSP + WFC (a) + WFC (b) + Ruínas BSP 1	768,76 ms
	BSP + WFC (a) + Floresta CA + Ruínas BSP 1	1240,44 ms
	BSP + WFC (a) + Digger Invertido + Caverna + Ruínas BSP 1	**1481,86 ms**

Onde RW = Passeio Aleatório e CA = Autômatos Celulares.
“Floresta CA” equivale a 13 iterações de CA.
“Caverna” equivale à combinação de RW com 17 iterações de CA.

Figura 5.39 – Relação entre tamanho do mapa e tempo de geração para *pipelines* híbridos.



6 DATA-DRIVEN DESIGN

No protótipo de jogo implementado neste trabalho, todas as entidades, cores dos *tiles* e tabela de *spawn* são definidas separadamente do código-fonte, em arquivos de dados chamados de *raws*, utilizando-se o formato RON. A partir da leitura e interpretação desses arquivos de dados, o jogo pode ser facilmente modificado e ampliado em conteúdo sem a necessidade de reescrever código-fonte. Desse modo, o código-fonte é estendido apenas quando novos tipos de dados precisam ser interpretados. Essa prática é desejável em jogos com numeroso conteúdo.

6.1 DEFINIÇÃO DE ENTIDADES

Como exemplo, as Figuras 6.5 e 6.2 contêm a definição de duas entidades do jogo: uma espada curta japonesa (*Tantou*) e um guaxinim (*Raccoon*), respectivamente. Posteriormente à leitura dos *raws*, cada atributo definido para uma entidade será interpretado como um componente durante a interpretação do arquivo RON em código.

Figura 6.1 – Definição de itens e equipamentos em RON.

```
items: [
    (
        name: "Tantou",
        descr: "A guardless short sword.", // Descrição.
        tier: 2, // Qualidade do item.
        renderable: ( // Informações gráficas.
            glyph: '/',
            fg: "BrightCyan",
            bg: "Background",
            layer: 0,
        ),
        equipable: (
            slot: "weapon1", // Lugar onde o item é equipado.
        ),
        melee: (
            damage: "1d4", // Dano: 1 dado de 4 lados.
            class: "dagger" // Classe do item (adaga).
        ),
    ),
    // Continua...
],
```

Figura 6.2 – Definição de NPCs/mobs em RON.

```
mobs: [
    (
        name: "Raccoon",
        descr: "This furry creature carefully wanders the wild
        while carrying a nut with its small, dexterous hands.",
        mob_type: "Wildlife",
        renderable: (
            glyph: ‘r’,
            fg: "White",
            bg: "Background",
            layer: 1,
        ),
        fov_range: 20,
        blocker: true,
        stats: (
            hp: 3, // Vida atual.
            max_hp: 3, // Vida máxima.
            attack: "1d1", // Dano.
            defense: 0, // Defesa.
        ),
    ),
    // Continua...
],
```

Note a diferença de legibilidade ao comparar a definição de um mob na Figura 6.2 com o exemplo de definição de outro mob na Figura 2.10. Enquanto este trabalho utiliza um formato padrão para os *raws*, *Dwarf Fortress* utiliza um formato arbitrário.

6.2 DEFINIÇÃO DE CORES

O jogo permite a alternação dinâmica entre diferentes paletas de cores durante o *gameplay*, mudando drasticamente o visual do jogo quando o jogador desejar. Paletas de cores podem ser criadas e modificadas a partir de um arquivo de dados RON.

Na Figura 6.3 está a definição da paleta de cores *elemental*. Todas as paletas de cores criadas devem seguir o mesmo padrão, e são baseadas nas paletas de cores presentes no *website Extended ASCII Viewer*[1], que contém uma coleção de *tilesets* e paletas de cores em formato JSON utilizados em jogos do gênero *roguelike*. Para manter o visual consistente (ver Figura 6.4), o número de cores para cada paleta é limitado. As paletas foram convertidas de JSON para RON.

[1] <extended-ascii-viewer.herokuapp.com/>.

Figura 6.3 – Definição de paletas de cores em RON.

```
colorschemes: [
    (
        name: "elemental",
        colors: {
            "Black":"#3c3b30",
            "Shadow":"#3B3231",
            "Blue":"#497f7d",
            "BrightBlack":"#545444",
            "BrightBlue":"#78d8d8",
            "BrightCyan":"#58d598",
            "BrightGreen":"#60e06f",
            "BrightMagenta":"#cd7c53",
            "BrightRed":"#df502a",
            "BrightWhite":"#fff1e8",
            "BrightYellow":"#d69827",
            "Cyan":"#387f58",
            "Green":"#479942",
            "Magenta":"#7e4e2e",
            "Red":"#97280f",
            "White":"#807974",
            "Yellow":"#7f7110",
            "Background":"#21211c",
        },
    ),
    // Continua...
],
```

Figura 6.4 – Diferentes paletas de cores que podem ser alternadas a qualquer momento.

Fonte: Próprio autor.

6.3 DEFINIÇÃO DE CONTÊINERES

Contêineres de itens e equipamentos, como baús, também podem ser definidos facilmente. A partir destas informações, os contêineres são populados com itens no momento da geração de um mapa.

Figura 6.5 – Definição de contêineres (e.g. baús) em RON.

```
containers: [
    (
        name: "Chest",
        descr: "A plain wood chest.",
        blocker: true,
        renderable: (
            glyph: 'Æ',
            fg: "Magenta",
            bg: "Background",
            layer: 1,
        ),
        max_items: 5, // Capacidade máxima do contêiner.
        tiers: [1, 2], // Qualidade dos itens possíveis.
    ),
    // Continua...
],
```

6.4 DEFINIÇÃO DA TABELA DE *SPAWN*

A tabela de *spawn* (população) presente na Figura 6.6 define a possibilidade de inserção de *mobs* e itens no ambiente.

Figura 6.6 – Definição da tabela de *spawn* em RON.

```
spawn_table: [
    (
       name: "Med-Kit",
       spawn_weight: 1,
    ),
    (
        name: "Man-Ape",
        spawn_weight: 40,
        min_max_level: (1, 3),
        level_type: ["Forest", "Ruins", "Cave"],
    ),
    // Continua...
],
```

Quanto maior o `spawn_weight`, mais frequente é a observação da entidade no nível. `min_max_level` define o intervalo fechado de níveis nos quais a entidade pode existir, enquanto `level_type` restringe onde a entidade pode ser encontrada para certos tipos de níveis.

7 PROTÓTIPO DESENVOLVIDO

Para demonstrar o uso dos conceitos descritos nos Capítulos 4, 5 e 6, foi desenvolvido um protótipo de jogo que ilustra a sinergia entre as técnicas utilizadas. O estado atual pode ser observado na Figura 7.1. A tela representada na figura abaixo é onde ocorre a maior parte da jogabilidade: à esquerda está visível o estado do jogador (nome, pontos de vida, equipamentos), e à direita o ambiente de jogo, no qual o símbolo branco de "@" no centro representa o jogador; por fim, na região inferior da tela está o *log*, que registra as ações realizadas no jogo.

Figura 7.1 – Estado atual do jogo implementado neste trabalho.

Enquanto o jogo ainda está em fase de desenvolvimento, a sua base de funcionamento está completa e basta ser expandida para suportar novas interações e possibilidades; a partir do ECS aliado ao design *data-driven* e dos *pipelines* de geração procedural, essa expansão passa a exigir menos esforço em código; no entanto, questões referentes a *game design* ainda precisam ser ponderadas. O repositório do jogo está disponível publicamente no GitHub[1].

[1] <https://github.com/pprobst/tcc-ufsm-2020>.

Referente ao ECS desenvolvido no jogo, estão atualmente implementados (ao menos parcialmente) os seguintes sistemas:

- Sistema de inteligência artificial – no momento, apenas IA hostil, que consiste na perseguição ao jogador utilizando uma implementação eficiente do algoritmo de busca A*[2] (HART; NILSSON; RAPHAEL, 1968), já presente na *bracket-lib*.
- Sistema de itens consumíveis (e.g. itens para recuperar pontos de vida).
- Sistema de dano (Seção 4.1).
- Sistema de equipamentos (equipar ou desequipar itens).
- Sistema de campo de visão (*field of view*) do jogador e dos NPCs.
- Sistema de coleta de itens.
- Sistema de abandono de itens.
- Sistema de mapeamento (atualização de entidades no mapa).
- Sistema de combate corpo a corpo.
- Sistema de combate à distância.
- Sistema de carregamento de armas de fogo.

Grandes porções do código do jogo acessam ou escrevem informações referentes ao ECS, mas não formam necessariamente um sistema do ECS, por exemplo: renderização de componentes gráficos, todos os componentes da UI, população de NPCs no ambiente (*spawning*), algoritmos de geração procedural, remoção de entidades não utilizadas, ações específicas do jogador, entre outros. Quanto ao processo de geração procedural de *dungeons* no jogo, são implementados *pipelines* de geração diversos como exemplificados no Capítulo 5.

Além da tela principal do jogo, como observado na Figura 7.1, outras telas intermediárias também foram implementadas. Mais especificamente, a tela do menu principal, telas contendo descrições de entidades no mapa (Figura 7.2), telas de inventário (Figura 7.3) e telas de contêineres de itens (Figura 7.4). Com exceção do menu, todas as telas intermediárias surgem em regiões apropriadas na tela principal quando necessário, sem a sobrepor inteiramente, priorizando o fácil acesso a outras informações.

[2]Uma excelente explicação do algoritmo – incluindo visualizações interativas – está presente em <https://www.redblobgames.com/pathfinding/a-star/introduction.html>.

Figura 7.2 – Telas de descrições de entidades no mapa (*tooltips*).

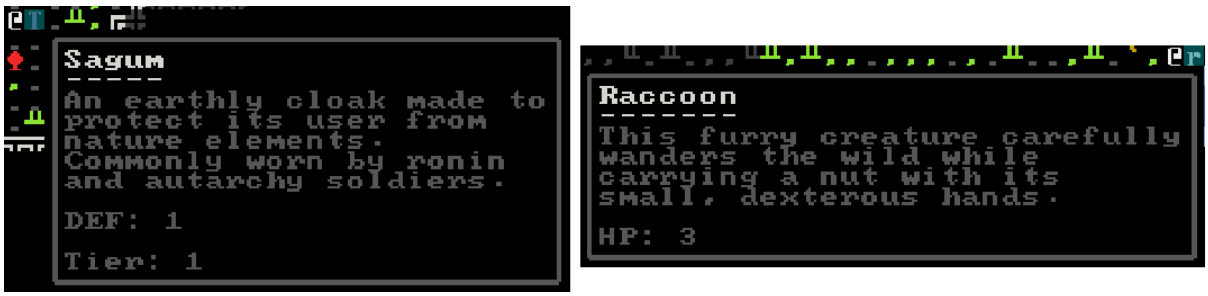


Figura 7.3 – Telas de inventário e equipamentos do jogador.

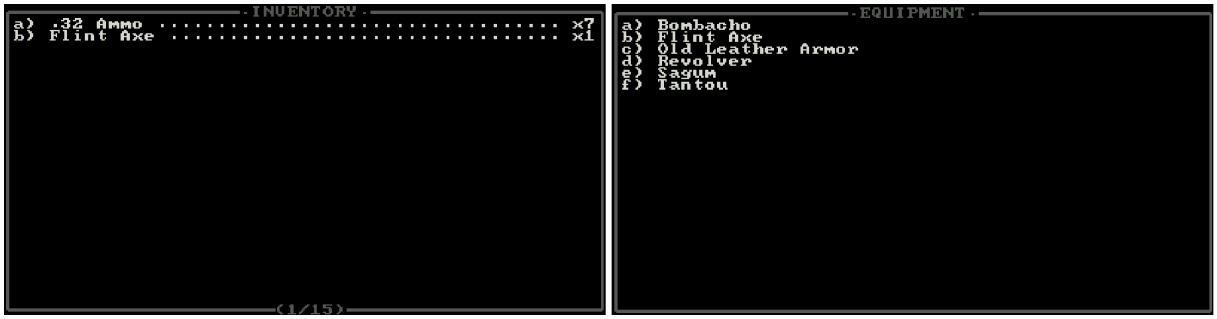
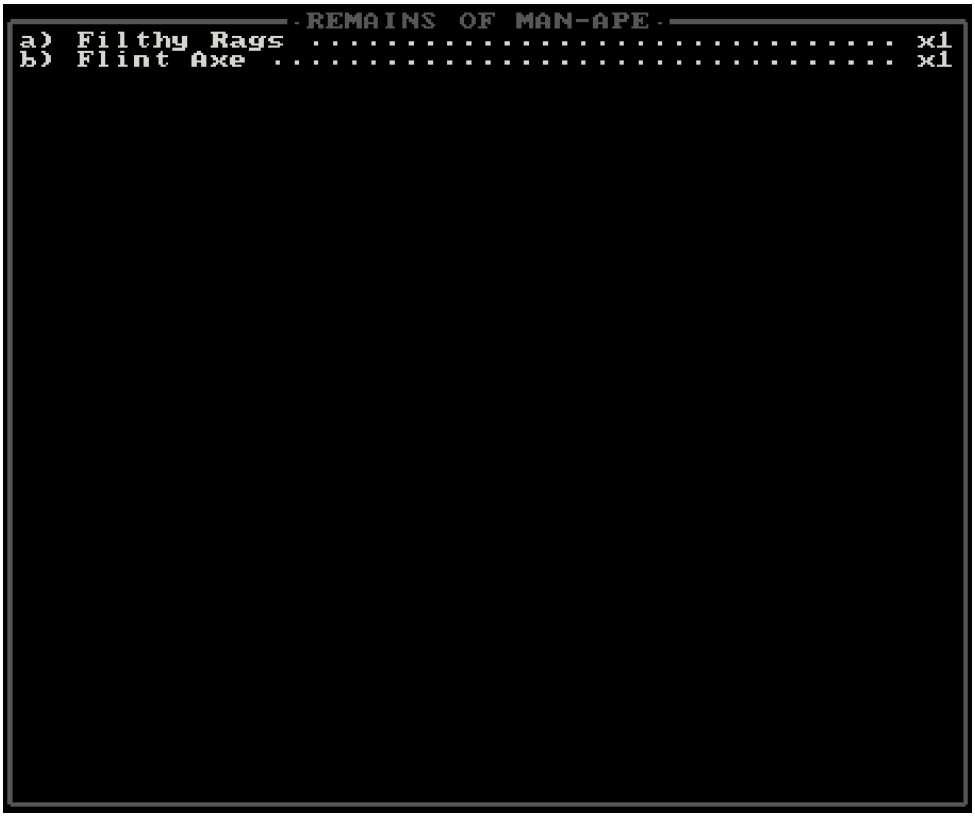


Figura 7.4 – Tela contendo os equipamentos deixados por um inimigo derrotado. Utilizado também em outros tipos de contêineres de itens, como baús.



8 CONCLUSÃO

Em suma, este trabalho explorou três áreas referentes ao desenvolvimento de jogos: *entity-component-system*, geração procedural de conteúdo e design *data-driven*.

Por ser uma arquitetura de software relativamente desconhecida, cada aspecto de um ECS foi explorado (excluindo o seu funcionamento interno, que pode fazer parte de outro trabalho), sendo demonstrado como ele pode ser utilizado no desenvolvimento de jogos mecanicamente complexos, como *roguelikes* ou jogos de estratégia em tempo real – casos onde um ECS pode ser muito bem utilizado, por conta de sua facilidade de manipulação e organização em código. Ao contrário de POO, não é preciso atentar-se com camadas complexas de herança; *uma entidade é apenas um contêiner de componentes*, e cada entidade pode ser manipulada em qualquer sistema, a partir de acessos ao ECS. Entretanto, é complexo implementar um *framework* para suportar ECS, e bibliotecas externas são tipicamente necessárias para que um ECS possa ser criado e utilizado, enquanto POO é suportado nativamente em diversas linguagens de programação.

O design *data-driven* foi explorado através da criação de arquivos de dados contendo a descrição de *mobs*, itens, equipamentos, entre outras informações. Em código, esses dados foram interpretados e inseridos no ECS. Enquanto o jogo ainda é escasso em conteúdo, note que os arquivos de dados podem ser estendidos de diversos modos, entre eles:

- Definição do comportamento (inteligência artificial) de cada *mob* e as interações entre diferentes espécies de criaturas do jogo.
- Definição dos métodos de geração procedural de ambientes utilizados em cada nível, facilitando a experimentação para programadores.
- Definição de efeitos (e.g. fogo, veneno, entre outros) possíveis em equipamentos que aparecem no jogo, aumentando a variedade de táticas possíveis ao jogador e seus inimigos.

Através de *pipelines* de geração relativamente simples, este trabalho demonstrou como diferentes métodos geração procedural podem ser combinados para criar ambientes 2D interessantes por meio de geração localizada com WFC. No entanto, enquanto os algoritmos construtivos utilizados são bastante versáteis e funcionais, podendo ser utilizados muitas vezes unicamente, o WFC – apesar de inovador – tem utilização limitada, sendo imprescindível sua combinação com outros algoritmos. Em especial, foi demonstrado como o WFC pode não ser interessante para arquitetura externa, sendo melhor utilizado para preencher pequenos detalhes, como arquitetura interna em salas geradas por outros algoritmos. Para jogos do gênero *roguelike*, pode ser proveitoso alterar internamente o WFC

para que sua utilização seja expandida, e isso pode fazer parte de outro trabalho. Em outros jogos – especialmente em 3D – a utilização do WFC é majoritariamente estética; por exemplo, no jogo *Rodina*, foi utilizado o WFC para a geração procedural de decorações em paredes[1].

Além disso, através de experimentos de geração procedural, foram descobertas acidentalmente diversas sinergias possíveis envolvendo a utilização de autômatos celulares juntamente com outros algoritmos, como *BSP Dungeon* e *Digger*. Portanto, a manipulação de *pipelines* de geração procedural utilizando autômatos celulares pode ser útil para a geração de ambientes diversos, não apenas cavernas e florestas como são normalmente utilizados. O processo de geração localizada – no qual vários algoritmos são utilizados em regiões diversas num mapa – foi igualmente importante, sendo raramente documentado por desenvolvedores de jogos.

Por ser um nicho de interesse, as próximas etapas do trabalho – além da conclusão do jogo – podem incluir a exploração de outros aspectos do PCG, como geração procedural de narrativa, cultura, sociedades e política. O *roguelike Ultima Ratio Regum*[2] é um excelente exemplo de exploração de tais tópicos, no qual todos os elementos gerados atuam entrelaçados por meio de "geração procedural qualitativa" (JOHNSON, 2016). Além disso, os algoritmos aqui apresentados também podem ser transformados para 3D, incluindo o WFC (STALBERG, 2018). O uso de inteligência artificial em PCG também está em uso crescente, uma vez que experimentos recentes com PCG via aprendizado por reforço (KHALIFA et al., 2020) e por meio de *generative playing networks* (BONTRAGER; TOGELIUS, 2020) mostram que é uma área de estudo promissora, podendo também fazer parte de uma possível continuação deste trabalho.

[1] <https://steamcommunity.com/games/314230/announcements/detail/3369147113795750369>.
[2] <https://www.markrjohnsongames.com/games/ultima-ratio-regum/>.

REFERÊNCIAS BIBLIOGRÁFICAS

BARRIGA, N. A. A short introduction to procedural content generation algorithms for videogames. **International Journal on Artificial Intelligence Tools**, World Scientific, v. 28, n. 02, p. 1930001, 2019.

BONTRAGER, P.; TOGELIUS, J. Fully differentiable procedural content generation through generative playing networks. **arXiv preprint arXiv:2002.05259**, 2020.

BUCKLEW, C. B. **Data-Driven Engines of Qud and Sproggiwood**. 2015. Acesso em: 26/08/2020. Disponível em: <https://youtu.be/U03XXzcThGU>.

____. **Dungeon Generation via Wave Function Collapse**. [S.l.]: Roguelike Celebration, 2019. <https://youtu.be/fnFj3dOKcIQ>. Acesso em: 09/12/2019.

FORD, T. **Overwatch Gameplay Architecture and Netcode**. 2017. Accesso em: 01/11/2020. Disponível em: <https://youtu.be/W3aieHjyNvw>.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1994. 20 p.

GRINBLAT, J.; BUCKLEW, C. B. Subverting historical cause & effect: generation of mythic biographies in caves of qud. In: **Proceedings of the 12th International Conference on the Foundations of Digital Games**. [S.l.: s.n.], 2017. p. 1–7.

HARKONEN, T. **Advantages And Implementation Of Entity-Component-Systems**. 2019. 28 p. Monografia (Trabalho de Conclusão de Curso) — Curso de Graduação em Engenharia de Software, Universidade de Tampere, Tampere, Finlândia, 2019.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE transactions on Systems Science and Cybernetics**, IEEE, v. 4, n. 2, p. 100–107, 1968.

JOHNSON, L.; YANNAKAKIS, G. N.; TOGELIUS, J. Cellular automata for real-time generation of infinite cave levels. In: **Proceedings of the 2010 Workshop on Procedural Content Generation in Games**. [S.l.: s.n.], 2010. p. 1–4.

JOHNSON, M. R. **Towards Qualitative Procedural Generation**. 2016. Accesso em: 24/06/2020. Disponível em: <https://youtu.be/Mk-TmpSUb54>.

KARTH, I.; SMITH, A. M. Wavefunctioncollapse is constraint solving in the wild. In: **Proceedings of the 12th International Conference on the Foundations of Digital Games**. [S.l.: s.n.], 2017. p. 1–10.

KHALIFA, A. et al. Pcgrl: Procedural content generation via reinforcement learning. **arXiv preprint arXiv:2001.09212**, 2020.

KIM, H. et al. Automatic generation of game content using a graph-based wave function collapse algorithm. In: IEEE. **2019 IEEE Conference on Games (CoG)**. [S.l.], 2019. p. 1–4.

LINDEN, R. V. D.; LOPES, R.; BIDARRA, R. Procedural generation of dungeons. **IEEE Transactions on Computational Intelligence and AI in Games**, IEEE, v. 6, n. 1, p. 78–89, 2013.

LLOPIS, N. **Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)**. 2009. Accesso em: 19/12/2020. Disponível em: <http://gamesfromwithin.com/data-oriented-design>.

MARTIN, A. **Entity Systems are the Future of MMOG Development – Part 2**. 2007. Acesso em: 21/08/2020. Disponível em: <http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/>.

MERTENS, S. **ECS: The beginning of the end for OOP?** 2018. Accesso em: 09/11/2020. Disponível em: <https://www.linkedin.com/pulse/ecs-beginning-end-oop-sander-mertens>.

MININI, P. P.; ASSUNÇÃO, J. V. Combining constructive procedural dungeon generation methods with wavefunctioncollapse in top-down 2d games. In: **2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)**. [S.l.: s.n.], 2020.

NYSTROM, R. Component. In: ____. **Game Programming Patterns**. [S.l.]: Genever Benning, 2014. cap. 14.

____. Data locality. In: ____. **Game Programming Patterns**. [S.l.]: Genever Benning, 2014. cap. 17.

PEREIRA, L. T. **Procedural Generation of Dungeon Maps, Missions and Rooms**. 2019. Dissertação (Mestrado) — Programa de Mestrado em Ciência da Computação, Universidade de São Paulo, São Carlos, Brasil, 2019.

SANDHU, A.; CHEN, Z.; MCCOY, J. Enhancing wave function collapse with design-level constraints. In: **Proceedings of the 14th International Conference on the Foundations of Digital Games - FDG '19**. ACM Press, 2019. Disponível em: <https://doi.org/10.1145%2F3337722.3337752>.

SCHUMACHER, R. et al. **Study for Applying Computer-generated Images to Visual Simulation**. Defense Technical Information Center, 1969. (AFHRL-TR). Disponível em: <https://books.google.com.br/books?id=fZOYtAEACAAJ>.

SCURTI, H.; VERBRUGGE, C. Generating paths with wfc. In: **Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference**. [S.l.: s.n.], 2018.

SHAKER, N. et al. Constructive generation methods for dungeons and levels. In: **Procedural Content Generation in Games**. [S.l.]: Springer, 2016. p. 31–55.

SHAKER, N.; TOGELIUS, J.; NELSON, M. J. **Procedural Content Generation in Games**. [S.l.]: Springer Publishing Company, Incorporated, 2016.

SHERRATT, S. **Procedural Generation with Wave Function Collapse**. 2019. <https://gridbugs.org/wave-function-collapse/>. Accesso em: 10/02/2020.

SILVA, C. G. P. da. **Analysis and Development of a Game of the Roguelike Genre**. 2015 — Curso de Graduação em Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil, 2015.

STALBERG, O. **Wave Function Collapse in Bad North**. 2018. Accesso em: 24/06/2020. Disponível em: <https://youtu.be/0bcZb-SsnrA>.

STRAUME, P. M. **Investigating Data-Oriented Design**. 2019. Dissertação (Mestrado) — Departamento de Ciência da Computação, Universidade Norueguesa de Ciência e Tecnologia, Gjøvik, Noruega, 2019.

TOGELIUS, J. et al. Search-based procedural content generation: A taxonomy and survey. **IEEE Transactions on Computational Intelligence and AI in Games**, v. 3, n. 3, p. 172–186, 2011.

TOY, M.; ARNOLD, K. **A Guide to the Dungeons of Doom**. [s.n.], 1980. 9 p. Disponível em: <https://books.google.com.br/books?id=ye9XGwAACAAJ>.

VAGEDES, J. A. et al. Ecs architecture for modern military simulators. In: THE STEERING COMMITTEE OF THE WORLD CONGRESS IN COMPUTER SCIENCE. **Proceedings of the International Conference on Scientific Computing (CSC)**. [S.l.], 2019. p. 118–122.

VELD, B. et al. Procedural generation of populations for storytelling. In: **Sixth FDG Workshop on Procedural Content Generation**. [s.n.], 2015. Acesso em: 26/08/2020. Disponível em: <https://graphics.tudelft.nl/Publications-new/2015/IKBC15/IKBC15.pdf>.

VIANA, B. M. F. **Geração Automática de Níveis de Masmorras com Barreiras para Jogos**. 2019 — Curso de Graduação em Ciência da Computação, Universidade Federal do Rio Grande do Norte, Natal, Brasil, 2019.

VIANA, B. M. F.; SANTOS, S. R. dos. A survey of procedural dungeon generation. In: **2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)**. IEEE, 2019. Disponível em: <https://doi.org/10.1109%2Fsbgames.2019.00015>.

WOLFRAM, S. Statistical mechanics of cellular automata. **Rev. Mod. Phys.**, American Physical Society, v. 55, p. 601–644, Jul 1983. Disponível em: <https://link.aps.org/doi/10.1103/RevModPhys.55.601>.

WOLVERSON, H. **Roguelike Tutorial - In Rust**. 2019. Acesso em: 01/03/2020. Disponível em: <https://bfnightly.bracketproductions.com/rustbook/>.

____. **Procedural Map Generation Techniques**. [S.l.]: Roguelike Celebration, 2020. <https://youtu.be/TlLIOgWYVpI>. Acesso em: 16/10/2020.