UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Classificação de algoritmos para geração procedural de conteúdo em jogos digitais

Nathan Oliveira

Itajubá, Novembro de 2016

UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Nathan Oliveira

Classificação de algoritmos para geração procedural de conteúdo em jogos digitais

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do título de Mestre em Ciência e Tecnologia da Computação

Área de Concentração: Matemática da Computação

Orientador: Prof. Dr. Rodrigo Duarte Seabra

Novembro de 2016
Itajubá — MG

Agradecimentos

À minha mãe, Raquel, e a minha avó, Auxiliadora, que me apoiaram durante esta jornada, e possibilitaram que eu pudesse me focar exclusivamente nesta pesquisa.

À minha irmã, Susan, meus tios, tias, primos e primas, os quais sempre me apoiaram.

Aos meus amigos e colegas, que enfrentaram este mestrado comigo, especialmente ao Demétrius, com o qual foram compartilhadas muitas das dificuldades desta jornada. Foi muito bom passar esse tempo com vocês.

A todos estes, muito obrigado também por ter me ajudado a lembrar que há mais à vida do que apenas pesquisas e estudos, o que muitas vezes eu esquecia.

Ao Professor Dr. Alexandre Carlos Brandão Ramos, agradeço pelas conversas que tivemos antes do início deste mestrado, e que me convenceram que valia a pena o esforço necessário para que esta pesquisa fosse completada. Embora não tenhamos trabalhado juntos nesta pesquisa, ela não teria sido sequer iniciada se não fosse pelos seus conselhos.

Ao meu orientador, Professor Dr. Rodrigo Duarte Seabra, agradeço, em primeiro lugar, por ter aceitado me orientar neste assunto, que é uma área a qual sempre tive bastante interesse. E, finalmente, também agradeço pela orientação neste trabalho, que foi muito importante para que este fosse concluído com sucesso e dentro dos prazos.

Reality is broken. Game designers can fix it.
— Jane McGonigal

Resumo

Atualmente, os jogos digitais tentam atingir um realismo cada vez maior. Com esse objetivo, a geração de conteúdo para os jogos tende a ser cada vez mais complexa e custosa. Neste cenário, a utilização de técnicas de geração procedural de conteúdo pode diminuir tanto os custos quanto o tempo de produção do conteúdo necessário aos jogos digitais. Esta dissertação propõe comparações entre técnicas responsáveis pela geração procedural de conteúdo para jogos digitais. Nesse sentido, os algoritmos das técnicas tratadas nesta pesquisa foram implementados nas linguagens C++ e, quando aplicável, GLSL. A partir dessas implementações, foram realizadas análises pautadas por algumas métricas, tais como complexidade algorítmica, tempo de execução, consumo de memória, qualidade dos resultados, tipo de conteúdo gerado, método de geração, momento da geração, corretude do conteúdo, reprodutibilidade dos resultados e condição de parada dos algoritmos. Essas análises foram realizadas por meio do estudo de cada algoritmo, no caso da complexidade algorítmica, consumo de memória, método e momento de geração, reprodutibilidade dos resultados e condição de parada; ou da análise do conteúdo gerado, no caso da qualidade dos resultados e tipo de conteúdo gerado; ou, ainda, instrumentação do código, no caso da medição do tempo de execução. As métricas complexidade algorítmica, tempo de execução, consumo de memória e condição de parada foram definidas para auxiliar o desenvolvedor a selecionar técnicas que sejam adequadas às quantidades de tempo e memória disponíveis para a execução do código; as métricas de qualidade e tipo do conteúdo gerado foram escolhidas visando à seleção de técnicas que gerem conteúdo necessários ao jogo e sejam similares ao estilo da arte usada; as métricas de momento de geração, corretude e reprodutibilidade dos resultados têm como objetivo guiar o desenvolvedor às melhores técnicas para cada necessidade, evitando que os resultados não sejam corretos e causem problemas que atrapalhem os jogadores; por fim, o método de geração também foi selecionado como métrica pois auxilia os desenvolvedores e pesquisadores a escolher técnicas que sejam algoritmicamente parecidas, o que pode diminuir o tempo gasto para se estudar e desenvolver novas técnicas. Foram selecionadas técnicas que possuem implementações abertas e disponíveis na Internet. Estas foram reimplementadas, com o objetivo de padronizar e otimizar o código disponível. Após isso, os resultados obtidos em cada técnica foram comparados com o objetivo de se obter uma base de dados que possa ser utilizada por pesquisadores e desenvolvedores, proporcionando a eles informações que auxiliem a escolha da melhor técnica para cada situação. Por fim, os códigos implementados foram disponibilizados em um repositório aberto, de livre acesso para consulta e uso das técnicas.

Palavras-chave: Jogos digitais; Conteúdo para jogos digitais; Geração procedural de conteúdo.

Abstract

Nowadays, digital games strive to achieve an ever growing realism. With this goal in mind, the generation of the contents needed for the games tends to be more complex and costly. In this scene, the use of procedural content generation techniques can lead to the reduction of both costs and production time of the contents necessary for digital games. This dissertation proposes comparisons between techniques for procedural content generation for digital games. In this sense, the researched techniques' algorithms were implemented in both C++ and, when applicable, GLSL. From these implementations, there were made analysis based on some metrics, such as algorithmic complexity, run time, memory consumption, and results' quality, type of the generated content, generation method, time of generation, correctness of the content, reproducibility of the results and stop condition of the algorithms. The analysis were made through algorithm analysis, in the case of algorithmic complexity, memory consumption, method and moment of the generation, reproducibility of the results and stop condition; or the analysis of the generated content, in the case of quality of results and type of content generated; or, lastly, code instrumentation, in the case of run time. The algorithmic complexity, run time, memory consumption and stop condition metrics were defined to help the developer select techniques that match its time and memory budget to run the code; the quality and type of content metrics were chosen to help select techniques that generate the needed content and match the game art style; the time of generation, correctness and reproducibility of results metrics tries to guide the developer to the best techniques to each need, preventing that the results be incorrect, creating problems to the player; lastly, the generation method was selected as a metric because it helps developers and researchers to choose techniques that are algorithmically similar, which can help to reduce the time needed to study and develop new techniques. There were implemented techniques that already had open implementations, available on the Internet. These were reimplemented, to padronize and optimize the available code. Then, the results obtained in each technique were compared as to obtain a database that can be used by researchers and developers, providing them with data that might help them in choosing the best technique for each situation. Lastly, the implemented code was made available in an open code repository, providing free access to read and use the techniques.

Keywords: Digital Games; Digital games' content; Procedural content generation.

Sumário

1 Introdução **1**
- 1.1 Justificativa . . . 2
- 1.2 Objetivos . . . 4
- 1.3 Organização da Dissertação . . . 5

2 Fundamentação Teórica **7**
- 2.1 Jogos Digitais . . . 7
- 2.2 Conteúdo de jogos digitais . . . 18
- 2.3 Geração Procedural de Conteúdo . . . 21
- 2.4 Algoritmos para Geração Procedural de Conteúdo . . . 24
  - 2.4.1 Algoritmos comumente utilizados nas técnicas de PCG . . . 25
  - 2.4.2 Algoritmos tratados nesta pesquisa . . . 28
    - 2.4.2.1 Ruídos . . . 28
    - 2.4.2.2 Nomes . . . 29
    - 2.4.2.3 Vegetação . . . 29
    - 2.4.2.4 Simulação de elementos naturais . . . 30
    - 2.4.2.5 Cavernas, labirintos e masmorras . . . 31
    - 2.4.2.6 Mapas de elevação . . . 32
    - 2.4.2.7 Cidades e ruas . . . 32
- 2.5 Trabalhos relacionados . . . 32

3 Método **35**
- 3.1 Metodologia utilizada . . . 35
  - 3.1.1 Análise de complexidade - big$O$ . . . 35
  - 3.1.2 Medição do tempo de execução . . . 36
  - 3.1.3 Análise do uso de memória . . . 38
- 3.2 Bits . . . 38
  - 3.2.1 Ruídos . . . 38
  - 3.2.2 Nomes . . . 39
  - 3.2.3 Vegetação . . . 39
  - 3.2.4 Simulação de elementos naturais . . . 40
- 3.3 Espaço . . . 40
  - 3.3.1 Cavernas, labirintos e masmorras . . . 40

3.3.2 Mapas de elevação 41
3.4 Sistemas 41
3.4.1 Cidades e ruas 41

4 Experimentos e Resultados **43**
4.1 Ruídos 43
4.1.1 Big O 43
4.1.2 Tempo de execução 44
4.1.3 Consumo de memória 45
4.1.4 Qualidade dos resultados 48
4.1.5 Tipo de algoritmo 48
4.2 Nomes 49
4.2.1 Big O 51
4.2.2 Tempo de execução 52
4.2.3 Consumo de memória 53
4.2.4 Qualidade dos resultados 53
4.2.5 Tipo de algoritmo 54
4.3 Vegetação 55
4.3.1 Big O 55
4.3.2 Tempo de execução 55
4.3.3 Consumo de memória 56
4.3.4 Qualidade dos resultados 56
4.3.5 Tipo de algoritmo 58
4.4 Simulação de elementos naturais 58
4.4.1 Big O 58
4.4.2 Tempo de execução 59
4.4.3 Consumo de memória 60
4.4.4 Qualidade dos resultados 60
4.4.5 Tipo de algoritmo 60
4.5 Cavernas, labirintos e masmorras 61
4.5.1 Big O 62
4.5.2 Tempo de execução 62
4.5.3 Consumo de memória 63
4.5.4 Qualidade dos resultados 64
4.5.5 Tipo de algoritmo 65
4.6 Mapas de elevação 66
4.6.1 Big O 66
4.6.2 Tempo de execução 66
4.6.3 Consumo de memória 66
4.6.4 Qualidade dos resultados 66
4.6.5 Tipo de algoritmo 68
4.7 Cidades e ruas 68
4.7.1 Big O 68

4.7.2 Tempo de execução 69
4.7.3 Consumo de memória 70
4.7.4 Qualidade dos resultados 70
4.7.5 Tipo de algoritmo 70

5 Conclusão **75**
5.1 Trabalhos futuros 77

Referências Bibliográficas **79**

Lista de Figuras

2.1 Jogo eletrônico de Goldsmith Jr., T. T. e Mann, E. R. 8
2.2 Reconstrução do computador *Nimrod* no *Computerspielemuseum Berlin*. 8
2.3 Reconstrução do jogo *Tennis for Two* para o *Museum of Electronic Games & Art*. 9
2.4 *Spacewar!* em execução no PDP-1 do *Computer History Museum*. 10
2.5 Jogos do início da década de 70: (a) *Pong*; (b) *Spasim*. 10
2.6 Jogos do fim da década de 70: (a) *Gun Fight*; (b) *Space Invaders*; (c) *Akalabeth*. 11
2.7 Jogos da década de 80: (a) *Rogue*; (b) *Mystery House*; (c) *Ultima I: The First Age of Darkness*; (d) *Prince of Persia* 13
2.8 Jogos da década de 90: (a) *Star Fox*; (b) *DOOM*; (c) *Diablo*; (d) *Shenmue*. 14
2.9 Editores de conteúdo dos motores de jogos: (a) *Blender Game Engine*; (b) *Aurora Engine*. 15
2.10 Jogos da geração atual: (a) *World of Warcraft: Mists of Pandaria*; (b) *The Elder Scrolls V: Skyrim*; (c) *Dwarf Fortress*; (d) *Grand Theft Auto V*. 17
2.11 Ferramentas para a geração de conteúdo procedural: (a) *CityEngine*; (b) *Grome*; (c) *SpeedTree*; (d) *Terragen*. 18
2.12 Exemplos de ***bits***: (a) textura aplicada a um cubo; (b) fogo e fumaça gerados de forma procedural. 19
2.13 Exemplos de **espaços** gerados de forma procedural: (a) Mapa interno em *Dungeons of Dredmor*; (b) Mapa externo em *Minecraft*. 20
2.14 Exemplo de **sistemas**: redes de estradas e ambiente urbano em *Cities: Skylines*. 21
2.15 Exemplo de **cenário**: Um *storyboard* que situa o jogador na história do jogo *Blocky Roads*. 22
2.16 Jogos procedurais da década de 80: (a) *Elite*; (b) *Rescue on Fractalus*; (c) *River Raid*; (d) *The Sentinel*. 23
2.17 Algoritmo diamante-quadrado: (a) grade com valores iniciais; (b, d) passos diamante, gerando os pontos centroides aos quadrados; (c, e) passos quadrado, gerando os pontos centroides aos diamantes. 25
2.18 Exemplo de autômato celular unidimensional. Utilizando a regra listada, cada linha é gerada aplicando-se a regra sobre a linha anterior, a partir de um caso base com apenas a célula central ativa. 26

2.19 Exemplo da *dragon curve* para as iterações: (a) uma iteração; (b) duas iterações; (c) três iterações. 28
2.20 Máquina de estados finitos para a geração de nomes. A máquina é iniciada a partir da escolha de um estado aleatório dentre {v, f, p, w}. A máquina produz uma saída de acordo com cada estado: vogal, se {v, v'}; consoante fricativa, se {f, f'}; consoante plosiva, se {p, p'}; e, "estranha", se {w}. . 30

3.1 Exemplo gráfico da notação big-*O*. 36
3.2 Tempo de execução por quadro vs quadros por segundo. 37

4.1 Tempo de execução, por tamanho de textura, dos métodos de ruído em C++. 46
4.2 Tempo de execução, por tamanho de textura, dos métodos de ruído em GLSL. 47
4.3 Tempo de execução, por pixel, dos métodos de ruído. 48
4.4 Consumo de memória dos métodos de ruído. 49
4.5 Ruídos: (a) valor; (b) gradiente; (c) convolução de grade; (d) convolução esparsa; (e) Perlin; (f) *OpenSimplex*; (g) Gabor. 50
4.6 Vegetação gerada pelas técnicas: (a) baseada em subdivisão aleatória dos ramos; (b) baseada em sistemas de funções iterativas; (c) baseada em L-*systems*. 57
4.7 Elementos gerados pelas técnicas: (a) fogo, baseado em ruído de fluxo; (b) fogo, baseado em partículas; (c) oceanos; (d) nuvens. 61
4.8 Tempo de execução, por tamanho de mapa, dos métodos de mapas internos em C++. 63
4.9 Partes de mapas internos gerados pelas técnicas: (a) cavernas com autômatos celulares; (b) cavernas com agregação por difusão limitada; (c) labirintos; (d) masmorras – primeira técnica; (e) masmorras – segunda técnica. As áreas verdes são passáveis, as brancas são portais e as outras são impassáveis. 64
4.10 Tempo de execução, por tamanho de mapa, dos métodos de mapas externos em C++. 67
4.11 Partes de mapas externos gerados pelas técnicas: (a) diamante-quadrado; (b) bisseção de hemisférios. 67
4.12 Resultados gerados pelas técnicas: (a) ruas e edifícios, baseado em agentes; (b) ruas, baseado em L-*systems* (áreas em verde são de alta densidade demográfica; em rosa é mostrado o mapa de elevação); (c) ruas e edifícios, baseado em L-*systems*. 71

Lista de Tabelas

2.1 Ferramentas para desenvolvimento de jogos digitais. 16

3.1 Tamanhos, em bytes, das variáveis básicas em C++. 38

4.1 Complexidade dos métodos de ruído. 45
4.2 Tempo de execução, por textura, dos métodos de ruído em C++. 45
4.3 Tempo de execução, por textura, dos métodos de ruído em GLSL. . . . 46
4.4 Complexidade algorítmica, tempo de execução, por pixel, e consumo de memória dos métodos de ruído. 47
4.5 Classificação dos métodos geradores de ruído. 49
4.6 Complexidade algorítmica e tempo de execução, por arquivo de entrada, dos métodos baseados em cadeias de Markov. 52
4.7 Complexidade algorítmica, tempo de geração, por nome, e consumo de memória dos métodos de geração de nomes. 52
4.8 Resultados dos geradores de nomes. 54
4.9 Classificação dos métodos geradores de nomes. 55
4.10 Complexidade algorítmica, tempo de geração e consumo de memória dos métodos de geração de vegetação. 56
4.11 Classificação dos métodos geradores de vegetação. 58
4.12 Tempo de inicialização dos métodos de simulação de elementos naturais. 59
4.13 Tempo de renderização dos métodos de simulação de elementos naturais na GPU para um quadro de tamanho 3840x2160 pixels. 59
4.14 Tempo de limpeza de recursos dos métodos de simulação de elementos naturais. 60
4.15 Classificação dos métodos simuladores de elementos naturais. 62
4.16 Complexidade algorítmica e tempo de execução, por mapa, dos métodos de mapas internos em C++. 63
4.17 Classificação dos métodos geradores de mapas internos. 65
4.18 Tempo de execução, por mapa, dos métodos de mapas externos em C++. 66
4.19 Classificação dos métodos geradores de mapas de elevação. 68
4.20 Complexidade algorítmica, tempo de execução e consumo de memória dos métodos de geração de cidades e ruas. 69
4.21 Classificação dos métodos geradores de cidades e ruas. 72

4.22 Resultados obtidos. 73
4.23 Resultados obtidos. 74

Lista de Listagens

3.1 Uso da biblioteca `chrono` para calcular o tempo de execução de um algoritmo em C++. 36
3.2 Uso da API do OpenGL para calcular o tempo de execução de um algoritmo em GLSL. 36

Lista de Abreviaturas e Siglas

API	*Application Programming Interface*
CPU	*Central Processing Unit*
GLSL	*OpenGL Shading Language*
GPU	*Graphics Processing Unit*
PCG	*Procedural Content Generation*
RAM	*Random Access Memory*
UHD	*Ultra High Definition*

Capítulo 1

Introdução

Jogos digitais são jogos que utilizam algum dispositivo capaz de realizar cálculos para a exibição de cenários interativos, nos quais um ou mais jogadores podem, por meio de dispositivos de entrada, realizar ações nesses ambientes com o objetivo de aprender, aprimorar habilidades, ou se divertir. Esses jogos podem ser classificados quanto ao seu objetivo principal: educacionais, recreativos, simuladores; quanto ao gênero: ação, aventura, estratégia, esportes, interpretação (*role-play*), simuladores, terror, tiro; dentre outras classificações.

Os dispositivos que reproduzem esses jogos requerem, para executar da forma esperada, certos tipos de dados que contêm as informações necessárias para a exibição dos cenários aos jogadores. Esses dados, no contexto de jogos digitais, são chamados de conteúdo.

O conteúdo para jogos digitais pode ser criado de variadas formas: manualmente, fazendo uso de ferramentas que permitam a manipulação dos dados – por exemplo, um terreno pode ser criado por meio da edição de curvas de nível pelo usuário; por conversão, usando ferramentas que gerem esses conteúdos a partir de um conjunto de dados de entrada – um modelo digital de elevação (*digital elevation model*) de um mapa topográfico pode ser utilizado para se gerar um terreno; ou, ainda, com o uso da geração procedural de conteúdo. Técnicas de geração procedural são empregadas em jogos digitais desde o fim da década de 70, com o objetivo de reduzir o consumo de memória ou produzir conteúdo de forma automatizada.

Nesse contexto, geração procedural consiste em gerar conteúdo de forma algorítmica ao invés de manualmente. Segundo Lagae et al. [77, p. 3], "o adjetivo procedural é usado em ciência da computação para distinguir entidades que são descritas por código de programa ao invés de estruturas de dados". Os algoritmos utilizados em técnicas de geração procedural de conteúdo (PCG, do inglês *Procedural Content Generation*) têm se tornado cada vez mais complexos devido à necessidade de se produzir conteúdo de maior qualidade, visando obter resultados mais realistas ou mais adequados ao estilo de cada jogador. Por esses motivos, atualmente existem variados tipos de algoritmos direcionados para situações específicas. Em sua pesquisa, Togelius et al. [137] propõem algumas distinções entre os tipos de algoritmos usados para a PCG, além de ressaltarem

a inexistência de livros-texto ou artigos que ofereçam uma taxonomia básica desses algoritmos.

Devido ao aumento da quantidade de algoritmos de PCG, causado principalmente pela especialização de algoritmos existentes, pesquisas recentes têm sido realizadas no sentido de classificar esses algoritmos quanto ao tipo de conteúdo gerado e técnicas exploradas em sua execução [29, 62, 137].

Algumas pesquisas a respeito dos algoritmos para PCG com foco em tipos específicos de conteúdo também têm sido realizadas. Entre elas estão as pesquisas de Kelly e McCabe [71], sobre técnicas para a geração de cidades, Lagae et al. [77], sobre técnicas para a geração de ruído, e Smelik et al. [120], sobre técnicas para a geração de terrenos. No entanto, não há registro de publicação de um estudo científico que classifique os algoritmos de PCG utilizando métricas que englobem tanto aspectos técnicos, como tempo de execução, consumo de memória, complexidade algorítmica, tipo de conteúdo, corretude dos resultados, condição de parada e método de geração, quanto aspectos de *design*, tais como qualidade dos resultados, momento da geração e reprodutibilidade.

Com isso, espera-se que a medição do tempo de execução e as análises de consumo de memória e complexidade algorítmica possam contribuir para um melhor entendimento dos desafios e soluções existentes na área de PCG.

1.1 Justificativa

Organizar e classificar os diferentes tipos de algoritmos, evidenciando seus pontos positivos e negativos, facilitará seu estudo. Considerando que a área de PCG passou a atrair o interesse dos estudiosos a partir dos últimos anos, faz-se necessária uma classificação que englobe diferentes aspectos do desenvolvimento, pesquisa e uso das técnicas de PCG. Além disso, aqueles que pretendem desenvolver algum tipo de software que use PCG, a partir deste trabalho, não necessitarão implementar diferentes algoritmos para fins de testes.

A proposta de uma possível comparação a ser realizada considerando métricas tais como complexidade algorítmica, tempo de execução, consumo de memória, qualidade dos resultados, tipo de conteúdo, método de geração, momento da geração, corretude dos resultados, reprodutibilidade e condição de parada pode auxiliar tanto desenvolvedores de jogos quanto de motores de jogos e pesquisadores da área de PCG e afins a encontrar o melhor algoritmo para cada caso, além de, possivelmente, resultar em novas técnicas e ferramentas [62].

No Brasil, os estudos na área de PCG ainda se iniciam, havendo pouca produção nacional. Dentre os poucos trabalhos, pode-se destacar as pesquisas de: Miranda, Cordeiro e Chaimowicz [95], que propõem uma técnica para geração de terrenos com o uso simultâneo da CPU e da GPU; Silva, França e Cabral [118], que propõem uma técnica baseada em sistemas *fuzzy* para a geração de trilhas sonoras para jogos; e Leite e Lima [80], que propõem uma técnica para a geração de mapas bidimensionais de cavernas, calabouços e ilhas. Além desses trabalhos, quatro dissertações de mestrado sobre geração procedural foram encontradas: Bevilacqua [16], com uma técnica para a

geração de terrenos costeiros em tempo real; Pereira [106], com uma técnica baseada em sistemas *fuzzy* para a geração de tarefas para um jogo educacional para o ensino de leitura e escrita; Carli [28], com uma técnica para a geração de mapas de cânions 3D; e Duarte [43], com uma técnica para a geração de cenários orientada a objetivos.

Esta pesquisa pretende analisar algumas das principais técnicas utilizadas em PCG, de forma a prover uma classificação que possa ser utilizada como base para estudos futuros. Esta proposta se faz necessária visto que, atualmente, não existe uma base de dados que contenha métodos procedurais já implementados, o que faz com que os pesquisadores ou tenham que implementar a técnica do zero, ou tenha que procurar alguma implementação disponível na Internet, sendo que as duas alternativas consomem bastante tempo.

Para esta pesquisa, foram selecionados dez critérios: complexidade algorítmica, tempo de execução, consumo de memória, qualidade dos resultados, tipo de conteúdo, método de geração, momento da geração, corretude dos resultados, reprodutibilidade e condição de parada.

A complexidade algorítmica pode ser uma boa referência de como o tempo de execução se altera ao se modificar os parâmetros de entrada, sendo importante em aplicações que necessitem executar em tempo limitado. O tempo de execução é importante para se selecionar as técnicas com base no limite de tempo disponível para a geração do conteúdo, especialmente quando o gerador for utilizado *online*. A análise do consumo de memória se faz necessária, principalmente, ao se utilizar as técnicas em dispositivos portáteis, que ainda tem memória bastante limitada. Esta também é importante ao se utilizar a PCG como forma de reduzir o consumo de memória ao custo de maior processamento, de forma a se determinar se realmente é interessante essa troca. A qualidade dos resultados, embora bastante subjetiva, dependendo do estilo de arte de cada jogo, representa uma métrica importante na área de PCG, visto que não basta que uma técnica gere uma saída, sendo necessário que esta tenha um nível de realismo adequado. A análise quanto ao tipo de conteúdo foi utilizada para se agrupar as diferentes técnicas pesquisadas em famílias, de acordo com o conteúdo gerado pelas técnicas, conforme proposto por Hendrikx et al. [62]. A classificação quanto ao método de geração pode facilitar a obtenção de técnicas relacionadas, o que pode auxiliar tanto pesquisadores, quanto desenvolvedores a encontrar as melhores técnicas para cada caso de uso, além de poder ser utilizada como ferramenta de ensino dos métodos de geração, direcionando professores e alunos a métodos tecnicamente similares. O momento da geração diz respeito à possibilidade de a técnica ser utilizada para gerar o conteúdo durante a criação do jogo ou durante sua execução. A corretude dos resultados é importante para geradores que serão utilizados para criar conteúdo que o jogador necessitará para o jogo, e, portanto, deve ser utilizável; geradores que não garantam a corretude podem ser utilizados para a geração de conteúdo opcional, com os quais o jogador não precisa interagir, como nuvens ou montanhas distantes. A reprodutibilidade de um conteúdo é útil para casos onde se deseje que o conteúdo gerado seja o mesmo para vários jogadores, ou como método de compressão de dados; no entanto, muitas técnicas são usadas especificamente para gerar resultados diferentes a cada execução, como simuladores de fogo e

fumaça. Por fim, a condição de parada de uma técnica diz respeito ao método utilizado para finalizar o algoritmo, sendo que, em sua maioria, as técnicas de PCG simplesmente geram um resultado e terminam, embora, atualmente, existam algumas técnicas que utilizam inteligência artificial como forma de garantir que o conteúdo gerado tenha as características desejadas, gerando e testando os resultados até a obtenção de um conteúdo adequado à situação.

1.2 Objetivos

O objetivo geral desta dissertação consiste em classificar alguns dos principais algoritmos para PCG, utilizados em jogos digitais, por meio dos critérios: complexidade algorítmica, tempo de execução, consumo de memória, qualidade dos resultados, tipo de conteúdo, método de geração, momento da geração, corretude dos resultados, reprodutibilidade dos resultados e condição de parada dos algoritmos.

O foco desta pesquisa é a geração do conteúdo, sendo que sua renderização é realizada apenas para a análise visual dos resultados. A discussão sobre métodos de renderização é uma proposta interessante, mas está fora do escopo deste trabalho. Isso se deve ao fato de que a maioria dos métodos de PCG pode ser utilizada tanto para jogos bidimensionais quanto para os tridimensionais, além de poder ser empregada dinâmica ou estaticamente na geração do conteúdo. Como essas diferentes situações exigem técnicas de renderização específicas, e de forma a não alienar nenhum desses casos, as técnicas de renderização não serão tratadas neste trabalho.

A partir do objetivo geral, os objetivos específicos da pesquisa são:

- Compilar uma lista com alguns dos principais algoritmos de PCG pertencentes a cada categoria;
- Classificar esses algoritmos quanto a:
  - complexidade, usando a notação *big O* [74];
  - tempo de execução, utilizando um computador com processador Intel Core i7-4790K, com 16GB de RAM e uma GPU Titan X;
  - consumo de memória;
  - qualidade dos resultados, comparando a semelhança entre o conteúdo produzido e a natureza de acordo com o tipo de conteúdo;
  - tipo de algoritmo, conforme proposto por Hendrikx et al. [62] e Togelius et al. [137], com base nas seguintes características: tipo de conteúdo gerado, método de geração, momento de geração, corretude do conteúdo, reprodução dos resultados e condição de parada.
- Comparar os principais algoritmos de PCG visando facilitar a seleção e a utilização dos melhores algoritmos para cada caso;
- Criar um repositório, aberto à comunidade científica, contendo os algoritmos estudados e suas implementações.

1.3 Organização da Dissertação

O texto segue a seguinte estrutura:

- **Capítulo 1** — contém a introdução, a justificativa e os objetivos do trabalho;
- **Capítulo 2** — apresenta a fundamentação teórica relacionada a jogos digitais, PCG e algoritmos que utilizam essas técnicas;
- **Capítulo 3** — detalha os algoritmos abordados por este trabalho, descrevendo o tipo de conteúdo gerado e as técnicas utilizadas por cada um deles;
- **Capítulo 4** — descreve os experimentos executados e os resultados obtidos a partir de cada algoritmo, além de comparar e tabular esses dados;
- **Capítulo 5** — contém as conclusões da pesquisa e propostas para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo é tratado o estado da PCG para jogos. A primeira seção lista um histórico dos jogos e da PCG para jogos, visando prover um entendimento da necessidade das técnicas de PCG, sobretudo, atualmente. Na segunda seção, são tratados os aspectos técnicos necessários à produção de conteúdo para jogos digitais. Em seguida, são abordados detalhes sobre a geração procedural de conteúdo, e, na seção seguinte, são listadas as principais técnicas utilizadas nos algoritmos de PCG, bem como os algoritmos tratados nesta pesquisa. Por fim, é realizado um levantamento do estado da arte em classificação de algoritmos para PCG.

2.1 Jogos Digitais

Jogos eletrônicos são jogos que utilizam sistemas eletrônicos para criar dispositivos interativos com os quais um ou mais jogadores podem se entreter. Um dos primeiros exemplos desses sistemas foi um jogo eletrônico implementado com o uso de um osciloscópio analógico [57]. O jogo consistia de um tubo de raios catódicos, uma resistência variável e um botão; o jogador, então, modificava a posição de um ponto luminoso no tubo com o uso da resistência variável e, após mover o ponto a um alvo posicionado no tubo, pressionava o botão, que causava a perda de foco do feixe de raios catódicos, simulando uma explosão no jogo. Uma fotografia desse dispositivo pode ser visualizada na Figura 2.1.

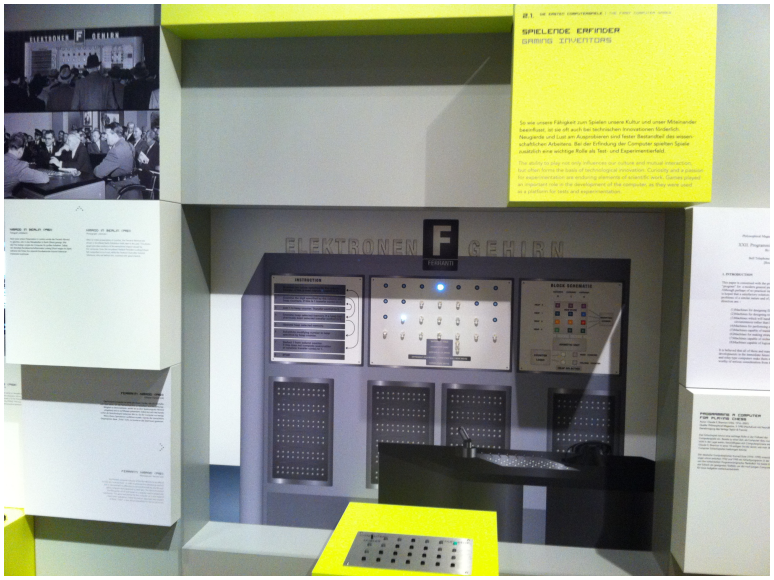
No início da década de 50, foram criados alguns jogos eletrônicos baseados em jogos físicos bastante conhecidos. Entre eles estavam dois simuladores de *jogo-da-velha*; *Bertie the Brain* [12], de 1950, que foi o primeiro computador a ser programado com a finalidade de ser um jogo de computador [119], e *OXO* [149], de 1952, que foi desenvolvido por Alexander S. Douglas, como parte de sua tese de doutorado sobre interação humano-computador. Outros dois jogos, ambos de 1951, também foram documentados; um, que simulava o jogo *Nim*, chamado *Nimrod* (Figura 2.2) [9]; e outro, que simulava o jogo de *damas*, que foi o primeiro programa a utilizar inteligência artificial com sucesso [47].

Em 1958, um jogo eletrônico conhecido como *Tennis for Two* foi projetado por Willian Higinbothan como uma atração para os visitantes do *Brookhaven National*

Figura 2.1: Jogo eletrônico de Goldsmith Jr., T. T. e Mann, E. R.

Fonte: Borgobello [23].

Figura 2.2: Reconstrução do computador *Nimrod* no *Computerspielemuseum Berlin*.

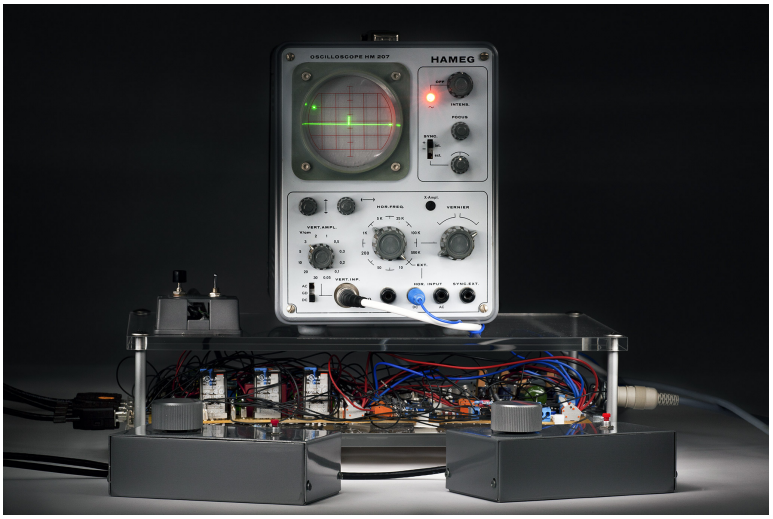


Fonte: Smith [121].

Laboratory [103]. Nesse jogo, dois jogadores controlavam um ponto na tela de um osciloscópio, que representava uma bola de tênis, por meio de um controle direcional e um botão para cada jogador. O jogo representava uma quadra de tênis vista pela

lateral (Figura 2.3) e os jogadores tinham como objetivo rebater a bola um para o outro. A trajetória da bola era simulada de acordo com a direção que o jogador a rebatia, utilizando a força da gravidade para alterar o percurso. Uma segunda versão, de 1959, permitia que a força da gravidade fosse alterada, simulando o jogo com características de outros planetas.

Figura 2.3: Reconstrução do jogo *Tennis for Two* para o *Museum of Electronic Games & Art*.



Fonte: Museum of Electronic Games & Art [98].

Durante a década de 60, surgem os primeiros jogos digitais, que são jogos implementados utilizando-se computadores digitais. Entre esses jogos estão *Spacewar!* [59], em 1962, no qual dois jogadores controlavam duas espaçonaves no campo gravitacional de uma estrela, tendo como objetivo atingir o adversário com um míssil, e, ao mesmo tempo, evitar uma colisão com a estrela (Figura 2.4); *The Sumer Game* [42], em 1968, um jogo baseado em texto, no qual o jogador controlava o estoque de grãos de uma civilização; *Lunar Landing Simulation*, em 1969, também baseado em texto, com o objetivo de controlar um módulo lunar e pousá-lo com segurança na superfície lunar; e *Space Travel* [53], em 1969, que simulava o sistema solar e permitia que o jogador explorasse esse espaço.

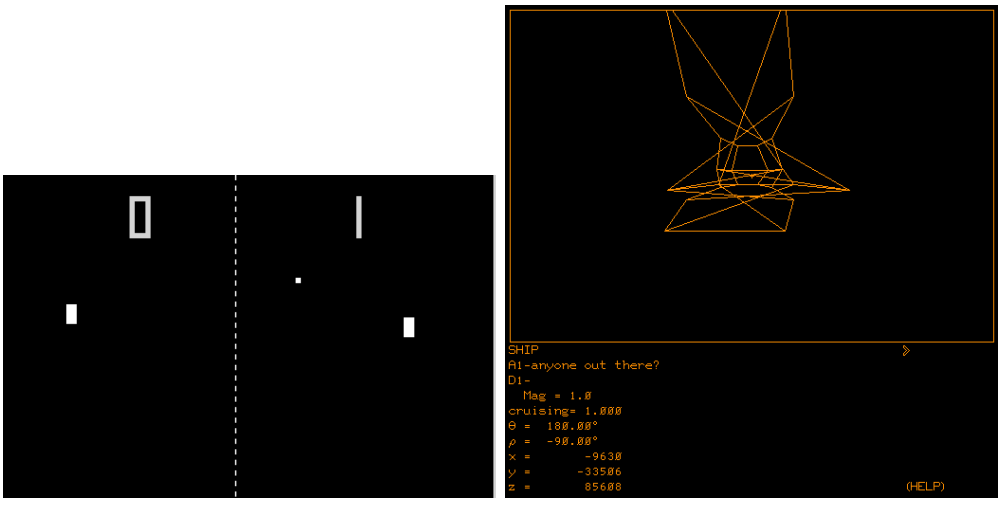
No início da década de 70, jogos digitais deixaram de ser limitados apenas aos laboratórios de pesquisa e passaram a ser comercializados. O primeiro jogo a ser comercializado foi *Computer Space* [122], em 1971, no qual o jogador controla uma espaçonave e deve desviar dos projéteis inimigos enquanto tenta atingir os adversários com seus projéteis. Outros jogos dessa época incluem *Star Trek* [17], em 1971, que é baseado em texto e usa o universo da série de televisão de mesmo nome como ambientação do jogo; *Pong* [34], em 1972, que foi um dos primeiros jogos digitais a atingir popularidade com o público geral (Figura 2.5a); e *Spasim* [26], em 1974, que foi o primeiro jogo de tiro em primeira pessoa, tridimensional e multi-jogador, utilizando redes de computadores,

Figura 2.4: *Spacewar!* em execução no PDP-1 do *Computer History Museum*.

Fonte: Ito [70].

que se tem documentação (Figura 2.5b).

Figura 2.5: Jogos do início da década de 70: (a) *Pong*; (b) *Spasim*.



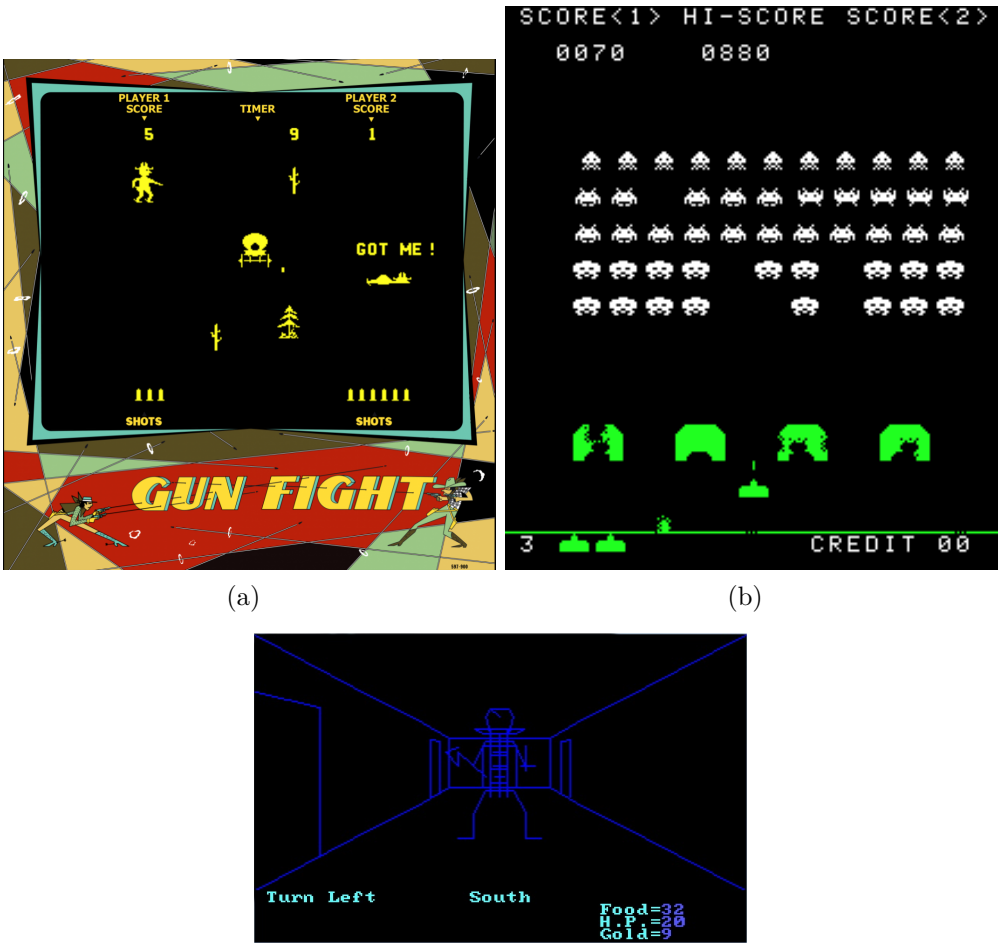
(a) (b)

Fontes: (a) International Arcade Museum [69]; (b) CBS Interactive Inc. [30].

Na segunda metade da década de 70, surgem os primeiros jogos a utilizarem microprocessadores. Essa evolução causou uma revolução no modo como os jogos digitais eram vistos, fazendo com que fosse possível a criação de video-games pequenos e baratos. Com isso, os jogos digitais deixaram de ser limitados apenas aos arcades e centros de

pesquisa. São exemplos de jogos dessa época, *Gun Fight*, em 1975, que foi o primeiro jogo a utilizar microprocessadores [72], e um dos primeiros a exibir figuras humanoides animadas na tela por meio do uso de *frame buffers* mapeados por *bit* [152] (Figura 2.6a); *Space Invaders* (Figura 2.6b), em 1978, que ajudou a popularizar os video-games domésticos, tendo vendido mais de dois milhões de cópias [64]; e *Akalabeth: World of Doom* (Figura 2.6c), em 1979, que foi um dos primeiros jogos a utilizar PCG para a construção dos mapas e das estatísticas do personagem [86].

Figura 2.6: Jogos do fim da década de 70: (a) *Gun Fight*; (b) *Space Invaders*; (c) *Akalabeth*.



(a) (b)

(c)

Fonte: (a) The Dot Eaters [132]; (b) Bowen [25]; (c) ChildOfTheMoon83 [32].

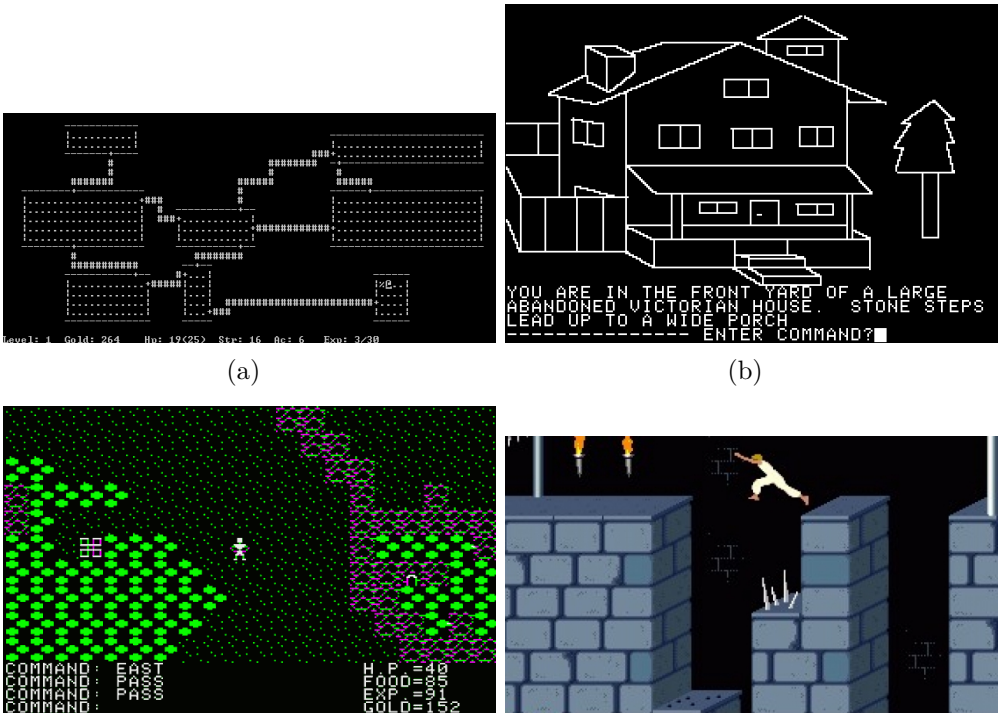
Durante a década de 80, os avanços nas técnicas utilizadas para a criação de jogos digitais fizeram com que eles se aproximassem da forma como são conhecidos atualmente. Dentre os jogos que introduziram novas técnicas estão *Rogue* [148], em 1980, que utiliza

PCG para gerar as masmorras e posicionar os itens que o jogador deve obter (Figura 2.7a); *Mystery House* [135], em 1980, que foi o primeiro jogo de aventura a utilizar gráficos para ambientar o jogador na história do jogo (Figura 2.7b); *Wizard and the Princess* [136], em 1980, o primeiro jogo de aventura a utilizar gráficos preenchidos à cores; *Frogger* [133], em 1981, que foi um dos primeiros jogos a utilizar mais de um processador; *Ultima I: The First Age of Darkness*, em 1981, um dos primeiros jogos a utilizar linguagem de montagem (*assembly language*) para aumentar o desempenho de seus gráficos (Figura 2.7c); *Zaxxon* [84], em 1982, que foi um dos primeiros jogos a exibir sombras para indicar a posição do jogador no cenário; *Telengard* [10], em 1982, que utilizava PCG para a geração das masmorras de forma a possibilitar sua execução em computadores com pouca memória RAM; *Moon Patrol* [55], em 1982, um dos primeiros jogos a utilizar técnicas de *parallax scrolling*, no qual o fundo da tela era dividido em várias camadas as quais se movimentavam com diferentes velocidades, induzindo a uma ilusão de profundidade; *3D Monster Maze* [63], em 1982, que utilizava PCG para a geração dos labirintos, sendo o primeiro jogo tridimensional para computadores domésticos; *Elite* [125], em 1984, que utilizava gráficos tridimensionais em *wireframe* com remoção de linhas ocultas de forma a aumentar o realismo [56], além de utilizar PCG para a geração das galáxias a serem exploradas pelo jogador devido a limitação dos computadores de 8 bits da época; *Starglider 2* [51], em 1988, um dos primeiros jogos a utilizar gráficos tridimensionais poligonais com sombreamento plano (*flat shaded*); e *Prince of Persia* [92], em 1989, que utilizava rotoscopia para modelar os personagens do jogo com base em atores, de forma a obter maior realismo nas animações dos personagens (Figura 2.7d).

Na década de 90, foram introduzidos ainda mais avanços nas técnicas utilizadas para os gráficos dos jogos, incluindo a criação de duas interfaces gráficas de programação de aplicativos (API, do inglês *Application Programming Interface*): *OpenGL*, em 1992, e *DirectX*, em 1995. Dentre os jogos criados nessa década, podem-se destacar: *Star Fox*, em 1993, que foi o primeiro jogo a utilizar um acelerador para gráficos tridimensionais [22] (Figura 2.8a); *DOOM*, em 1993, que popularizou os gráficos tridimensionais em primeira pessoa (Figura 2.8b); *Daytona USA*, em 1993, que foi um dos primeiros jogos a utilizar filtragem de textura (*texture filtering*) para aumentar o realismo das cenas [52]; *Diablo* [20], em 1996, que iniciou a série de jogos de mesmo nome e é um dos mais conhecidos a utilizar PCG para a geração de mapas (Figura 2.8c) com o objetivo de aumentar a rejogabilidade (*replay value*); *Final Fantasy VII*, em 1997, que foi o jogo mais caro da época, com custo de produção em torno de US$45000000 [88] devido, principalmente, ao alto custo de produção dos gráficos; e *Shenmue*, em 1999, que teve custo de produção de US$47000000 [116] devido tanto ao alto custo de produção dos gráficos quanto dos áudios, que incluíam uma orquestra e gravação de vozes nos idiomas inglês e japonês para os personagens do jogo (Figura 2.8d).

Na década de 90 e início dos anos 2000, com o aumento da complexidade de produção, equipes especializadas em cada área de desenvolvimento pertinente a jogos digitais começaram a surgir. Além disso, para diminuir os custos de desenvolvimento, os motores de jogos passaram a ser desenvolvidos de forma genérica, sendo reutilizados

Figura 2.7: Jogos da década de 80: (a) *Rogue*; (b) *Mystery House*; (c) *Ultima I: The First Age of Darkness*; (d) *Prince of Persia*



(a) (b)

(c) (d)

Fonte: (a) Artoftransformation [6]; (b) Bayo [14]; (c) PirateMink [110]; (d) Ford [54].

para outros jogos. Ademais, alguns desenvolvedores passaram a licenciar seus motores de jogos para outros desenvolvedores, criando uma indústria de desenvolvimento de ferramentas para a criação de jogos, simplificando a produção dos jogos digitais. Alguns dos motores de jogos e ferramentas mais conhecidos são: *Blender Game Engine* [18] (Figura 2.9a), *Chrome Engine* [129], *CryEngine* [38], *Euphoria* [100], *Infinity Engine*, *Aurora Engine* [117] (Figura 2.9b), *OGRE* [138], *RPG Maker* [49], *SFML* [58], *Source*, *SpeedTree* [66], *Unity* [142], e *Unreal Engine* [50].

As ferramentas listadas proveem APIs e interfaces para facilitar a criação de jogos e evitar a repetição de código em seus projetos. Na Tabela 2.1 são descritas algumas das características de cada ferramenta e exemplos de jogos que as utilizam.

Por fim, atualmente os desenvolvedores têm focado no aumento do realismo e da longevidade dos jogos digitais, sendo que esses dois objetivos têm em comum a produção de conteúdo de qualidade. Alguns exemplos de jogos modernos que utilizam grande quantidade de conteúdo para ampliar sua longevidade são: *World of Warcraft*, de 2004, para o qual foram desenvolvidos vários pacotes de expansão de conteúdo (Figura 2.10a); *Dwarf Fortress*, de 2006, que gera todo o conteúdo do jogo por meio de técnicas de

Figura 2.8: Jogos da década de 90: (a) *Star Fox*; (b) *DOOM*; (c) *Diablo*; (d) *Shenmue*.

(a) (b)

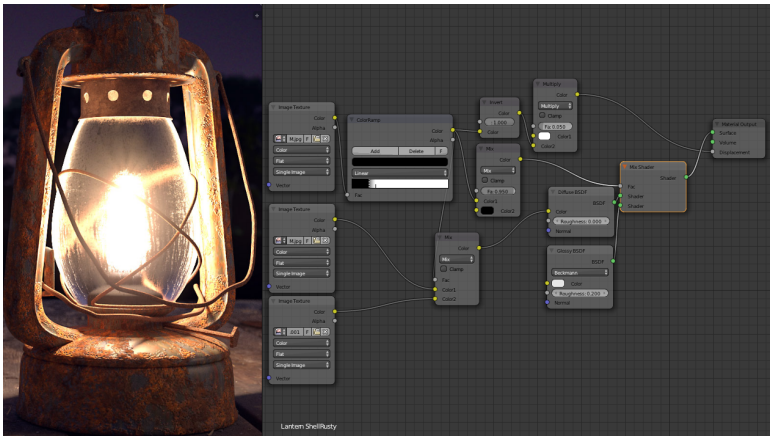
(c) (d)

Fonte: (a) Hahnchen [61]; (b) Sreejithk2000 [126]; (c) Grandpafootsoldier [60]; (d) Translucid2k4 [139].

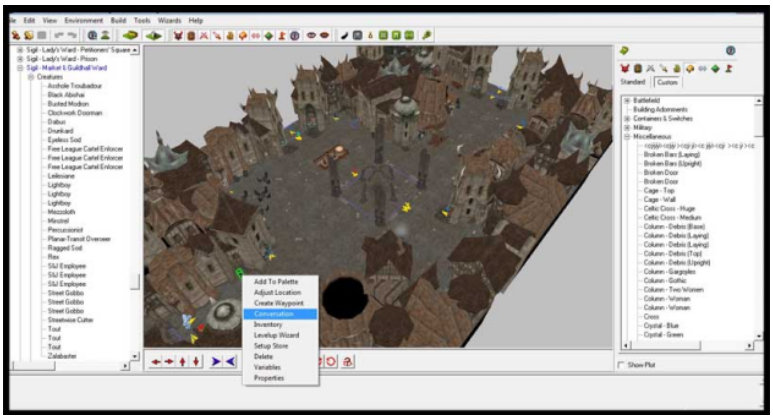
PCG (Figura 2.10c) e, segundo seus autores, ainda será desenvolvido por décadas [146]; e *Minecraft* [96], de 2009, que também gera os terrenos utilizando PCG. Exemplos de jogos que focam no aumento do realismo são as séries de jogos: *The Elder Scrolls* [15] (Figura 2.10b), *Far Cry* [141], e *Grand Theft Auto* [114] (Figura 2.10d). Esses jogos possuem grandes mapas a serem explorados e diversos objetivos a serem atingidos pelo jogador, garantindo uma longevidade de centenas a milhares de horas de jogo.

Na atualidade, a indústria de jogos digitais tem apresentado crescimento contínuo, com expectativa de atingir US$102.9 bilhões em 2017 [131]. No Reino Unido, os jogos digitais foram mais vendidos do que vídeos (DVDs, Blu-Rays e digitais) e música (vinil, CDs e digitais), representando 43% das vendas desses itens. Na Suécia, em 2014, estatísticas mostram que a indústria de jogos digitais cresceu 80% em relação a 2013, em parte, devido ao sucesso do jogo *Minecraft* [11], que tem como um dos principais

Figura 2.9: Editores de conteúdo dos motores de jogos: (a) *Blender Game Engine*; (b) *Aurora Engine*.



(a)



(b)

Fonte: (a) Blender Foundation [19]; (b) Watson [145].

atrativos o uso de técnicas de geração procedural de terrenos [109].

Com o avanço dos recursos de computação gráfica, na época atual, pode-se renderizar cenas e mundos virtuais cada vez mais realistas. Inicialmente, esses elementos eram criados de forma manual, por meio de editores 3D. No entanto, o processo de criação manual tem se tornado cada vez mais custoso, como exemplo, jogos como *Star Wars: The Old Republic* e *Grand Theft Auto V*, que tiveram custo de produção de US$200.000.000 e £170.000.000, respectivamente [143]. Ainda no caso do jogo *Grand Theft Auto V*, foram necessários mais de mil desenvolvedores, com relatos de que equipes de desenvolvimento chegavam a trabalhar doze horas por dia e seis dias por semana durante vários meses [124] devido aos curtos prazos necessários para a produção do jogo. Por esses motivos, técnicas de PCG têm sido utilizadas para facilitar e acelerar o

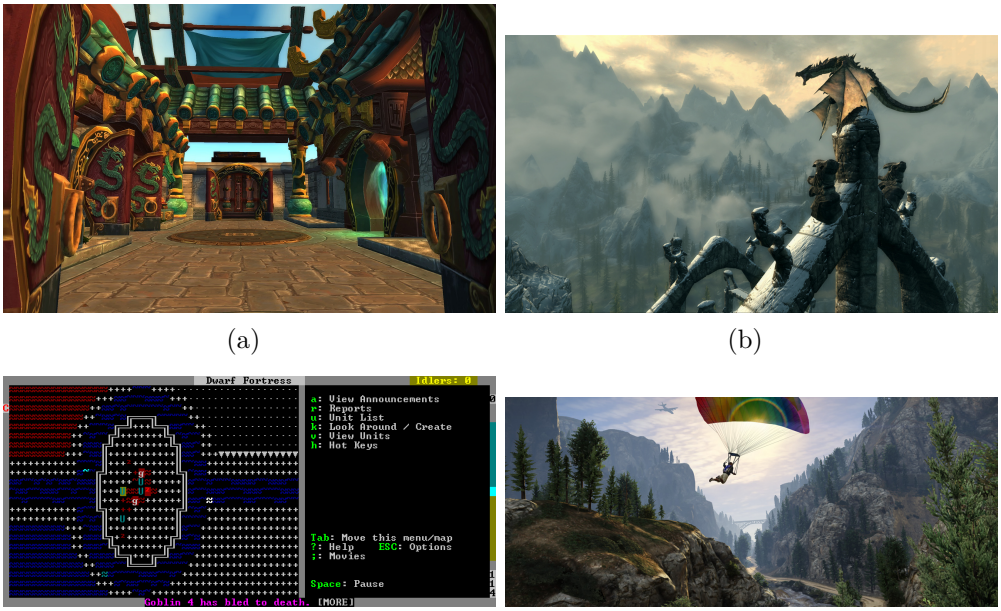
Ferramenta	Principais características	Jogos que utilizam a ferramenta
Blender Game Engine	Editor gráfico de comportamento interativo; detecção de colisão; simulação de dinâmica de veículos; *scripting* com *Python*; utiliza OpenGL e GLSL; suporte a áudio	*Yo Frankie!*; *Sintel The Game*
Chrome Engine	Terrenos baseados em mapa de elevação; vegetação procedural; fogo procedural; física de corpo rígido; suporte a áudio; suporte a redes; editor gráfico em tempo real	*Dying Light*; *Dead Island*; *Call of Juarez*
CryEngine	Editor gráfico em tempo real; vegetação procedural; motor de física *multi-thread*; suporte a áudio	*Far Cry*; *Crysis*; *Evolve*
Euphoria	Gerador procedural de animação de personagens 3D; capaz de simular interação física de músculos e nervos para obter resultados realistas	*Red Dead Redemption*; *Grand Theft Auto V*; *Max Payne 3*
Infinity Engine	Motor de jogo bidimensional; usa perspectiva isométrica para simular 3D; suporte a áudio	*Baldur's Gate*, *Planescape: Torment*; *Icewind Dale*
Aurora Engine	Motor de jogo sucessor ao *Infinity*; tridimensional; luzes e sombras em tempo real; suporte a conteúdo gerado pelo jogador para aumentar a longevidade do jogo	*Neverwinter Nights*
OGRE	Motor de renderização orientado a objetos; abstrai os recursos de renderização do computador de forma a facilitar o desenvolvimento multiplataforma	*Torchlight*; *Rigs of Rods*
RPG Maker	Motor de jogo bidimensional; editor gráfico; suporte a áudio	*Alpha Kimori*; *Aveyond*; *To the Moon*
SFML	Motor de jogo com suporte a áudio e rede	*KeeperRL*; *M.A.R.S.*; *Open Hexagon*
Source	Motor de jogo com suporte a áudio e rede	*Half Life*; *Portal*; *Left 4 Dead*
SpeedTree	Ferramenta para a geração procedural de vegetação	*Far Cry 4*; *The Order: 1886*; *Saints Row 4*; *The Witcher 3: Wild Hunt*
Unity	Motor de jogo com suporte a áudio e rede; editor gráfico	*Temple Run*; *Guns of Icarus Online*; *Cities in Motion 2*; *Hearthstone: Heroes of Warcraft*; *Kerbal Space Program*; *Cities: Skylines*
Unreal Engine	Motor de jogo com suporte a áudio e rede; editor gráfico	*Unreal*; *BioShock*; *Thief: Deadly Shadows*; *Antichamber*; *Mass Effect*; *Daylight*

Tabela 2.1: Ferramentas para desenvolvimento de jogos digitais.

processo de criação de conteúdo. Exemplos de editores 3D que utilizam essas técnicas incluem os softwares *Art of Illusion*[1], que é uma ferramenta para modelagem e renderização de objetos 3D que provê um editor gráfico para a criação de texturas de forma

[1] http://www.artofillusion.org/

Figura 2.10: Jogos da geração atual: (a) *World of Warcraft: Mists of Pandaria*; (b) *The Elder Scrolls V: Skyrim*; (c) *Dwarf Fortress*; (d) *Grand Theft Auto V*.



(a) (b)

(c) (d)

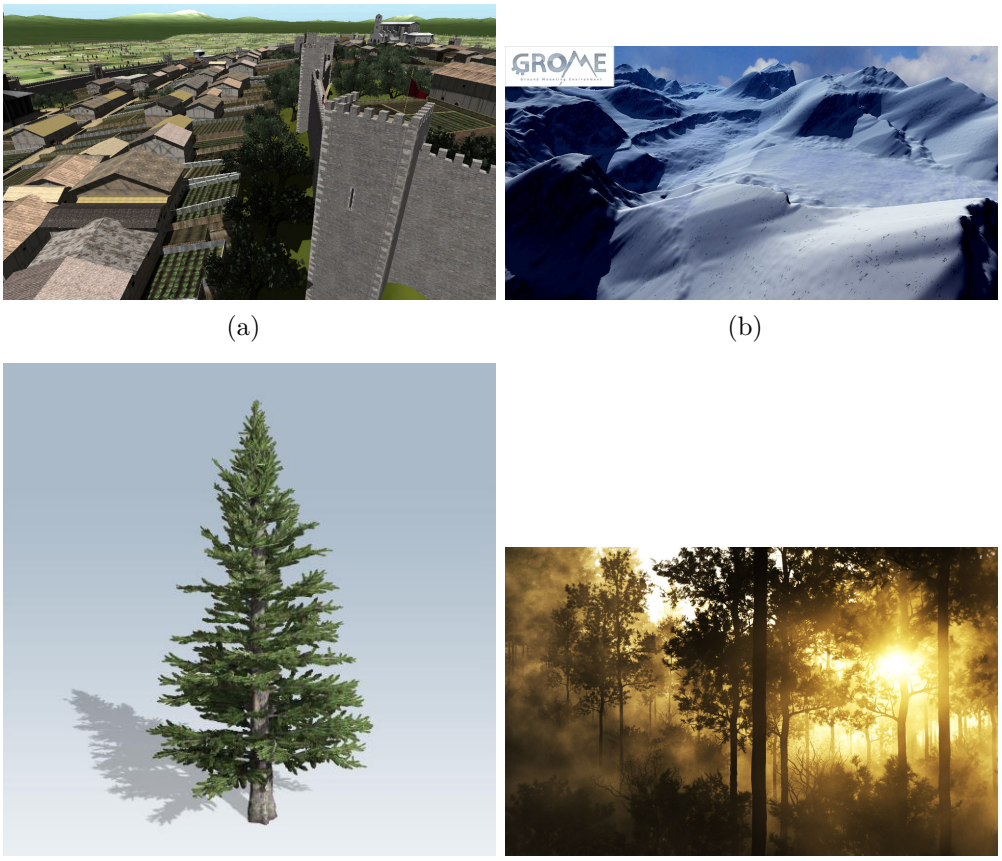
Fonte: (a) Blizzard Entertainment Inc. [21]; (b) NVIDIA Corporation [102]; (c) Bay 12 Games [13]; (d) Rockstar Games [115].

procedural; *CityEngine*[2], que é capaz de gerar e texturizar modelos 3D de construções a partir de dados bidimensionais (Figura 2.11a); *Grome*[3], que é capaz de gerar terrenos com o uso de PCG, além de gerar texturas também de forma procedural (Figura 2.11b); *SpeedTree*[4], que é capaz de gerar vegetação com o uso de PCG, além de simular efeitos de vento na vegetação (Figura 2.11c); *Terragen*[5], que é uma ferramenta de renderização e animação com suporte à geração procedural de terrenos (Figura 2.11d); e *Visual Nature Studio*[6], que é uma ferramenta para a criação de cenários com suporte à geração procedural de terrenos e de variação de objetos. Além disso, técnicas de PCG podem ser utilizadas para aumentar o desempenho de aplicações que sejam limitadas pela velocidade da memória, por meio da geração dinâmica do conteúdo, reduzindo a quantidade de dados lidos da memória. É importante notar que essa troca de memória por processamento deve ser analisada quanto à variação do desempenho, como qualquer outra forma de otimização.

No Brasil, a indústria de jogos ainda é pequena, sem a presença de grandes estúdios.

[2] http://www.esri.com/software/cityengine
[3] http://www.quadsoftware.com/index.php?m=section&sec=product&subsec=editor
[4] http://www.speedtree.com/
[5] http://planetside.co.uk/products/terragen3
[6] http://www.3dnworld.com/index.php/products/vns-3/

Figura 2.11: Ferramentas para a geração de conteúdo procedural: (a) *CityEngine*; (b) *Grome*; (c) *SpeedTree*; (d) *Terragen*.



(a) (b)

(c) (d)

Fonte: (a) Tecnalia [130]; (b) Quad Software [112]; (c) IDV Inc. [65]; (d) Archer [3].

Neste cenário, o uso de PCG pode proporcionar grande economia para os pequenos estúdios existentes por meio da geração automatizada de conteúdo.

2.2 Conteúdo de jogos digitais

Conteúdo de um jogo é todo tipo de elemento que é exibido ao jogador tanto de forma direta, como áudio e imagens, como de forma indireta, como o comportamento das entidades do jogo. Conteúdo de um jogo digital é um tipo de conteúdo representado de forma digital, sendo exibido ao jogador por meio de regras matemáticas que mapeiem os bits do conteúdo digital em informação que possa ser compreendida pelo jogador.

Dada a grande variedade de tipos diferentes de conteúdo necessários para a criação de jogos digitais, Hendrikx et al. [62] dividem esses tipos usando as seguintes categorias:

(i) *bits*; (ii) espaço; (iii) sistemas; (iv) cenários; (v) projeto; e (vi) conteúdo derivado.

A primeira categoria – ***bits*** – é composta por elementos simples, que necessitam ser combinados com outros elementos para que sejam criados objetos concretos no mundo do jogo, a saber: (a) texturas, que são imagens utilizadas para adicionar detalhes à geometria do mundo virtual (Figura 2.12a); (b) sons, que são utilizados para aumentar a imersão do jogador no mundo e para notificá-lo sobre alterações no ambiente virtual; (c) vegetação, que pode apresentar tanto objetivo estético, produzindo um ambiente mais realista, como de jogabilidade, fornecendo ao jogador novas táticas para serem exploradas durante o jogo; (d) construções, que podem ter objetivos estéticos ao exibir um ambiente urbano, além de serem utilizadas como elementos interativos. Como exemplo, o jogador pode acessar um edifício para utilizar algum recurso disponível em seu interior; (e) comportamentos, que determinam a forma como objetos interagem entre si, tal como o vento agitando a vegetação; (f) fogo (Figura 2.12b), água, pedras e nuvens que, assim como a vegetação, podem servir como decoração ou como elemento interativo, gerando novas táticas para o jogo.

Figura 2.12: Exemplos de ***bits***: (a) textura aplicada a um cubo; (b) fogo e fumaça gerados de forma procedural.

(a) (b)

Fonte: (a) Lighthouse3D [81]; (b) MCClelland [91].

A segunda categoria – **espaço** – representa o ambiente virtual, sendo composta por: (a) mapas internos, que representam ambientes compostos por salas, corredores, escadas etc., podendo servir como decoração ou elemento interativo. Alguns gêneros de jogos têm esses mapas como elemento central para a jogabilidade (Figura 2.13a); (b) mapas externos, que representam o mundo virtual, constituídos por florestas, montanhas e outros elementos que se façam necessários no jogo (Figura 2.13b); (c) corpos d'água como rios, lagos e oceanos, que podem ser usados como obstáculos para o jogador ou

como elementos interativos.

Figura 2.13: Exemplos de **espaços** gerados de forma procedural: (a) Mapa interno em *Dungeons of Dredmor*; (b) Mapa externo em *Minecraft*.

(a)

(b)

Fonte: (a) Captura de tela pelo autor; (b) Maggijanek [85].

A terceira categoria – **sistemas** – usa a teoria de sistemas complexos e modelagem para produzir elementos mais realistas, englobando: (a) ecossistemas, que controlam o posicionamento e a interação entre flora e fauna; (b) redes de estradas, que produzem uma estrutura básica de ruas e estradas entre pontos de interesse no mapa (Figura 2.14); (c) ambientes urbanos, que representam locais habitados por muitos indivíduos. São necessários em jogos que envolvam exploração, para que o jogador possa obter informações e suprimentos do ambiente ou dos habitantes de determinado local ou região; (d) comportamentos de entidades, que são usados para uma interação realista entre o jogador e personagens não-jogáveis, e também para a geração de padrões de movimento em grupos de personagens.

Figura 2.14: Exemplo de **sistemas**: redes de estradas e ambiente urbano em *Cities: Skylines*.

Fonte: notydino [101].

A quarta categoria – **cenários** – representa os eventos do jogo, sendo divididos em: (a) quebra-cabeças, que são problemas aos quais o jogador deve encontrar uma solução para prosseguir; (b) *storyboards*, que são utilizados para guiar o desenvolvedor do jogo ou o jogador (Figura 2.15); (c) narrativas, que exibem a lógica envolvida nas ações dos personagens do jogo e os objetivos que o jogador deve atingir; (d) níveis, que representam separadores entre sequências de áreas.

A quinta categoria – **projeto** – é composta por: (a) projeto de sistema, que representa as regras e objetivos do jogo; (b) projeto do mundo, que representa o estilo de arte usado, a narrativa e o tema do jogo.

A sexta categoria – **conteúdo derivado** – engloba o conteúdo gerado com base na experiência do jogador no mundo do jogo, incluindo: (a) notícias e transmissões, que podem ser utilizadas no ambiente do jogo para progredir a narrativa de acordo com as ações do jogador, ou externamente ao jogo por meio da publicação das sessões de jogos em sites ou programas de televisão; (b) *leaderboards*, que são uma forma de classificar o resultado de uma sessão de jogo, com o objetivo de criar competitividade entre jogadores.

2.3 Geração Procedural de Conteúdo

Geração procedural de conteúdo é um conjunto de técnicas de produção de conteúdo usado na geração de terrenos, mapas, personagens, sons, dentre outras características,

Figura 2.15: Exemplo de **cenário**: Um *storyboard* que situa o jogador na história do jogo *Blocky Roads*.

Fonte: Captura de tela pelo autor.

no contexto de jogos digitais. O estudo dessas técnicas surgiu na década de 70, com os trabalhos sobre gramáticas de formas [127] e fractais [87].

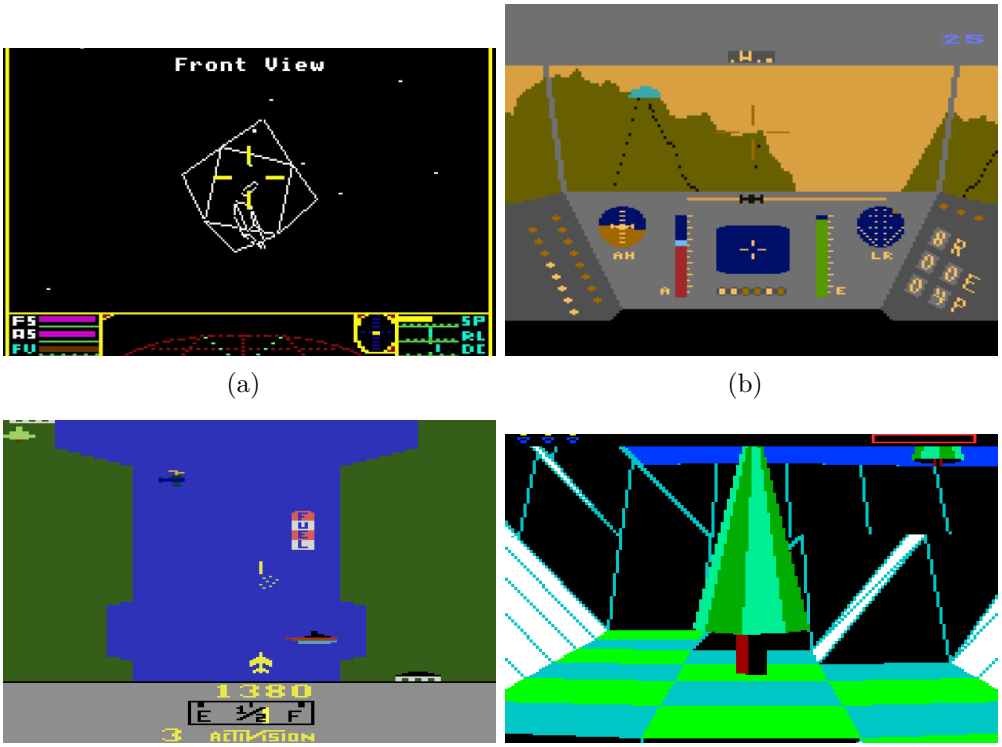
Gramáticas de formas são tipos de gramáticas formais, ou seja, conjuntos de regras de produção de cadeias em uma linguagem formal. Essas gramáticas utilizam cadeias que representam formas geométricas, sendo capazes de produzir formas n-dimensionais. Fractais são conjuntos matemáticos que exibem um padrão que se repete recursivamente ou em diversas escalas. Podem ser utilizados para descrever padrões que não sejam possíveis de serem representados com a geometria convencional.

Na década de 80, devido à pouca quantidade de memória computacional disponível, foram criados vários jogos eletrônicos utilizando técnicas de PCG como forma de contornar esse limite. Exemplos de jogos produzidos na época e que exploraram essas técnicas incluem *Akalabeth*, em 1980 [86], *Elite*, em 1984 (Figura 2.16a) [125], *Rescue on Fractalus*, em 1984 (Figura 2.16b) [82], *River Raid*, em 1982 (Figura 2.16c) [1] e *The Sentinel*, em 1986 (Figura 2.16d) [37]. Desde então, diversos algoritmos foram desenvolvidos para diferentes tipos de conteúdo e qualidade de resultados.

Hendrikx et al. [62] identificam seis grupos de métodos para PCG: (i) geradores de números pseudo-aleatórios; (ii) gramáticas geradoras; (iii) filtros de imagens; (iv) algoritmos espaciais; (v) modelagem e simulação de sistemas complexos; e (vi) inteligência artificial.

No primeiro grupo – **geradores de números pseudo-aleatórios** – são agrupados algoritmos que podem reproduzir a ilusão de aleatoriedade da natureza. Dentre os usos mais comuns, destaca-se o ruído de Perlin [107]. Este ruído é gerado com base em uma função n-dimensional e tem como resultado um *grid* de valores que representa uma textura capaz de simular efeitos naturais, como nuvens, de forma realista. Números pseudo-aleatórios também podem ser utilizados para simular a rolagem de dados e roletas.

Figura 2.16: Jogos procedurais da década de 80: (a) *Elite*; (b) *Rescue on Fractalus*; (c) *River Raid*; (d) *The Sentinel*.



(a) (b)

(c) (d)

Fonte: (a) Lusher [83]; (b) Tyan23 [140]; (c) Edwards [45]; (d) decoydoctorpus [41].

O segundo grupo – **gramáticas geradoras** – descreve conjuntos de regras usados para gerar elementos corretos a partir de elementos mais simples. São classificadas como: (a) sistemas de Lindenmayer [111], em que os símbolos das regras definem a estrutura ou o comportamento de um objeto; (b) *split grammars* [150], que geram novas formas a partir de um conjunto básico de formas e regras de conversão; (c) *wall grammars* [79], são gramáticas desenvolvidas para a criação de fachadas; (d) gramáticas de formas [97], são gramáticas sensíveis ao contexto e capazes de gerar formas mais complexas. As gramáticas geradoras podem ser utilizadas para gerar formas que apresentem padrões visíveis, como plantas e construções.

Filtros de imagem são algoritmos que têm como maior objetivo melhorar uma imagem com relação a alguma métrica, sendo compostos por: (a) morfologia binária, que visa obter informações sobre os elementos da imagem por meio do uso de uma imagem predefinida, sendo que os resultados são obtidos comparando as duas imagens em busca de pontos onde elas se encaixem; (b) filtros de convolução, que tem como objetivo alterar uma imagem de forma a se obter outra, para borrar (*blurring*), aguçar (*sharpening*), detectar bordas, dentre outras operações; essas operações são realizadas

pela multiplicação de uma matriz de convolução sobre cada pixel da imagem.

Dentro do quarto grupo – **algoritmos espaciais** – são agrupados algoritmos usados para manipular um espaço virtual, sendo compostos por: (a) *tiling* e *layering*, que decompõem um mapa em um *grid* e o recompõe quando necessário, reduzindo a quantidade de memória requerida para armazenar as texturas do mapa; (b) *grid subdivision* é uma técnica na qual apenas os elementos próximos ao jogador são melhor detalhados, economizando processamento dos elementos mais distantes; (c) fractais, figuras recursivas que podem ser geradas em vários níveis de detalhamento e são armazenadas como uma função recursiva, reduzindo a quantidade de memória necessária; (d) diagramas de Voronoi [7], que dividem um espaço em partes criando bordas que delimitam áreas próximas a pontos de interesse.

O quinto grupo – **modelagem e simulação de sistemas complexos** – descreve técnicas para simulação de fenômenos naturais, englobando: (a) autômatos celulares [33], que são modelos baseados em células de um *grid* e um conjunto de regras comportamentais; (b) campos de tensores [31], que podem ser utilizados para modelar a forma de mapas e objetos; (c) simulação baseada em agentes [39], na qual agentes simples são criados em um mapa e alteram sua forma.

Por fim, o sexto grupo – **inteligência artificial** – agrupa algoritmos que tentam reproduzir a inteligência encontrada na natureza utilizando técnicas como: (a) algoritmos genéticos, que tentam reproduzir a evolução biológica com o objetivo de encontrar as soluções mais adaptadas a determinado problema; (b) redes neurais artificiais, usadas para classificar e obter padrões; (c) satisfação de restrições e planejamento, que, a partir de um estado inicial, um conjunto de ações e um objetivo, tenta encontrar um caminho de ações que leve o estado inicial ao objetivo.

2.4 Algoritmos para Geração Procedural de Conteúdo

Atualmente, a PCG tem sido cada vez mais estudada em contexto acadêmico, principalmente devido à aplicação de técnicas de inteligência artificial usadas nos algoritmos de geração [113].

Segundo Togelius et al. [137], esses algoritmos podem ser distinguidos de acordo com os seguintes critérios: (a) *online* ou *offline*, que diz respeito ao tempo em que o conteúdo é gerado, se durante o design (*offline*) ou durante a execução do jogo (*online*); (b) conteúdo necessário ou opcional, que diz respeito à importância do conteúdo, onde o conteúdo necessário é aquele que o jogador precisa interagir, e, portanto, não pode apresentar erros em sua geração, enquanto que o conteúdo opcional, com o qual o jogador pode escolher interagir com ou não, pode apresentar erros sem impedir o progresso do jogo; (c) sementes aleatórias ou vetores de parâmetros, que definem se o algoritmo recebe um conjunto de parâmetros para gerar o conteúdo ou apenas uma semente para um gerador de números aleatórios (*random number generator*); (d) estocástico ou determinístico, que define se o algoritmo sempre terá a mesma saída quando for fornecida uma mesma entrada (determinístico) ou se o mesmo produz resultados diferentes para um mesmo conjunto de entrada (estocástico); (e) construtivo ou gerar-e-testar, que define se o

algoritmo executa uma vez e termina (construtivo) ou se após gerar o conteúdo, este é testado por meio de funções de aptidão (*fitness*), tendo seus parâmetros ajustados e o conteúdo novamente gerado até que se obtenha um bom resultado. Segundo Carli et al. [29], os algoritmos ainda podem ser classificados em assistidos, quando o usuário deve guiar o processo de geração do conteúdo, e não-assistido, onde o usuário apenas entra com os parâmetros iniciais do algoritmo.

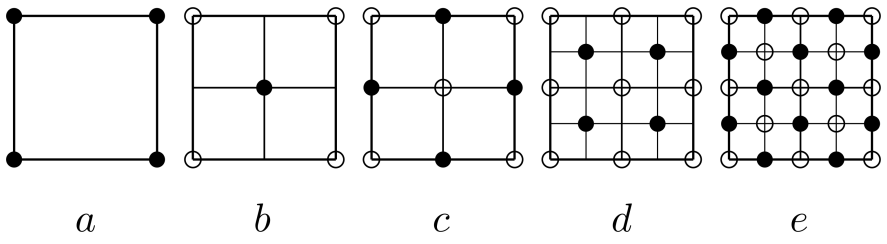
Kelly e McCabe [71] identificam, em sua pesquisa, os seguintes critérios para a análise de técnicas de PCG: (a) realismo – quão real o conteúdo gerado se parece; (b) escala – o conteúdo está em um tamanho equivalente ao real; (c) variação – a técnica recria o conteúdo com a mesma variação encontrada na natureza; (d) entrada – qual é o mínimo de dados requeridos para se obter uma saída básica e qual é a entrada necessária para a melhor saída; (e) eficiência – quanto tempo é necessário para se gerar os resultados, qual hardware é utilizado e quão eficiente é o algoritmo; (f) controle – qual o grau de controle que o usuário pode ter sobre os resultados, se esse controle é intuitivo, e se o usuário recebe um retorno imediato ao alterar algum parâmetro; (g) tempo real (*real time*) – os resultados podem ser exibidos em tempo real no hardware utilizado.

2.4.1 Algoritmos comumente utilizados nas técnicas de PCG

Alguns algoritmos clássicos são utilizados em diversas técnicas de PCG, tais como: algoritmo diamante-quadrado, autômatos celulares, cadeias de Markov, diagramas de Voronoi, fractais, geradores de números aleatórios e pseudo-aleatórios, sistemas de Lindermayer, e ruído de Perlin. Esses algoritmos são descritos a seguir, sendo dado um enfoque maior aos algoritmos utilizados nas implementações desta pesquisa.

O algoritmo diamante-quadrado é uma técnica para a geração de mapas de elevação para terrenos por meio do uso de um *grid* de valores, subdividido em *grid*s menores, cujos valores são calculados utilizando a média dos valores do *grid* anterior, acrescidos por uma componente aleatória. Os passos do algoritmo são ilustrados na Figura 2.17.

Figura 2.17: Algoritmo diamante-quadrado: (a) grade com valores iniciais; (b, d) passos diamante, gerando os pontos centroides aos quadrados; (c, e) passos quadrado, gerando os pontos centroides aos diamantes.

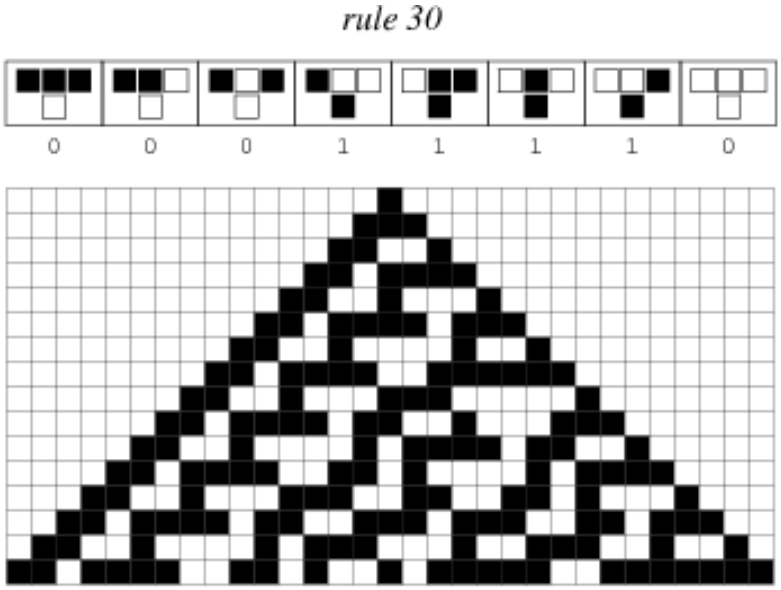


Fonte: Olsen [104]

Autômatos celulares são *grid*s de células, sendo que cada célula é um autômato e as transições de estado deles ocorrem ao mesmo tempo. Nesses autômatos, as regras de transição dependem dos estados das células vizinhas e da própria célula. Essa técnica

pode ser utilizada para a geração de padrões, tais como cavernas, ou para a simulação de interação entre animais de rebanhos, por exemplo. Um exemplo de autômato celular unidimensional é mostrado na Figura 2.18, onde cada linha é uma iteração sobre a linha anterior, utilizando a tabela de regras também mostrada na figura.

Figura 2.18: Exemplo de autômato celular unidimensional. Utilizando a regra listada, cada linha é gerada aplicando-se a regra sobre a linha anterior, a partir de um caso base com apenas a célula central ativa.



Fonte: Weisstein [147]

Cadeias de Markov podem ser vistas como máquinas de estado finitas nas quais a transição entre estados são realizadas por meio de uma função de probabilidade, onde as transições dependam apenas do estado atual. Essas cadeias podem ser utilizadas para gerar dados que tenham uma estrutura, ao invés de serem completamente aleatórias, como nomes de personagens ou sequências de notas musicais. Para a geração de bons resultados, são utilizados grandes volumes de dados de entrada que contenham estrutura parecida com a forma a ser gerada. Destes dados, são extraídos os valores de probabilidade de determinada parte de um dado ocorrer imediatamente após outra. É com base nessas probabilidades que os resultados posteriores são gerados.

Diagramas de Voronoi [7] são diagramas gerados a partir de um grupo de pontos de entrada e que mapeiam cada ponto no espaço para o ponto de entrada mais próximo do mesmo. São utilizados em algoritmos de PCG para gerar padrões em terrenos ou texturas com aparência de células.

Fractais [87] são conjuntos matemáticos utilizados para representar padrões que se repetem em diferentes níveis de detalhes. São utilizados em algoritmos de PCG para a geração de conteúdo que simule processos naturais como erosão, por exemplo, que

exibem padrões autossemelhantes.

Geradores de números aleatórios são algoritmos capazes de gerar números verdadeiramente aleatórios, geralmente com o uso de entropia obtida por meio de dispositivos de hardware específicos. Com isso, eles tendem a serem lentos, o que os torna não práticos para usos em que seja necessária a geração de muitos números em um curto espaço de tempo. Já os geradores de números pseudo-aleatórios geram, para determinado valor de entrada – a semente, sequências de números que parecem aleatórias, mas são idênticas, não se alterando em diferentes execuções com a mesma semente. Por não utilizarem qualquer forma de entropia para o cálculo da sequência, eles podem ser implementados de maneira eficiente, geralmente em tempo linear de execução. Dada a maneira determinística, os geradores pseudo-aleatórios não são seguros para aplicações como criptografia, mas na PCG esta característica faz com que os resultados possam ser reproduzidos de forma consistente. Desta forma, podem ser implementados algoritmos de PCG que também sejam determinísticos e permitam que jogadores possam compartilhar gerações interessantes entre si. Um dos geradores de números pseudo-aleatórios mais utilizados é o *Mersenne Twister*. Este algoritmo está disponível na biblioteca padrão (STL) do C++14, e, portanto, não será implementado neste trabalho.

Sistemas de Lindenmayer são gramáticas formais utilizadas para modelar o processo de crescimento de plantas [111]. O sistema é modelado a partir de um conjunto de regras de produção e uma cadeia de caracteres inicial, chamada de axioma. Com essa entrada, o algoritmo, então, expande recursivamente o axioma de acordo com as regras de produção. Um exemplo simples é a chamada *dragon curve*, que, com o axioma

$$FX$$

e as regras

$$X->X+YF+$$
$$Y->-FX-Y$$

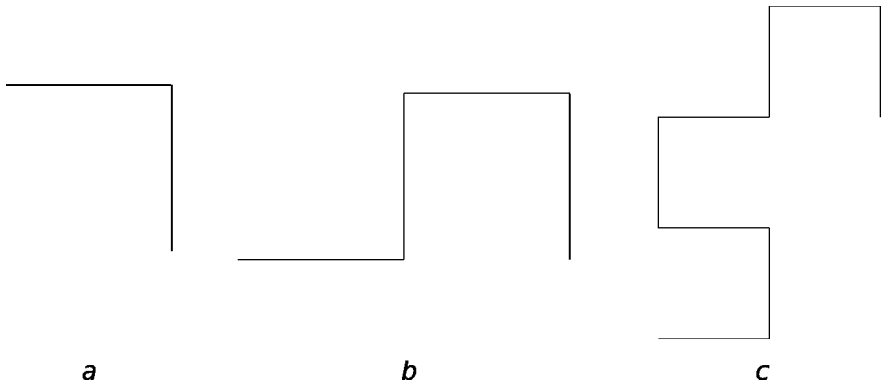
gera a sequência

$$FX+YF+$$
$$FX+YF++-FX-YF+$$
$$FX+YF++-FX-YF++-FX+YF+--FX-YF+$$
$$\vdots$$

que pode ser ilustrada pela Figura 2.19.

Perlin [107] propõe um método para a geração de texturas pseudo-aleatórias de forma rápida e paralelizável, sendo bastante utilizado para aumentar o grau de realismo de técnicas de PCG. Perlin também propõe um algoritmo chamado de ruído Simplex, com o objetivo de reduzir a complexidade do algoritmo original. O ruído de Perlin, juntamente com outros tipos de ruído, são tratados nessa pesquisa, em razão da ampla gama de aplicações em que esses ruídos são empregados.

Figura 2.19: Exemplo da *dragon curve* para as iterações: (a) uma iteração; (b) duas iterações; (c) três iterações.



Fonte: Autor.

2.4.2 Algoritmos tratados nesta pesquisa

Devido à grande quantidade de técnicas e algoritmos para PCG existentes, apenas alguns deles foram selecionados para serem analisados. O critério utilizado para a seleção dos algoritmos tratados foi a disponibilidade de uma implementação de referência da técnica. Isso se deu devido à limitação de tempo para se implementar técnicas que não tenham implementações completas e públicas. Os algoritmos implementados são descritos a seguir, agrupados por tipo de conteúdo gerado.

Dentre as implementações selecionadas estão tanto algumas de técnicas recentes quanto alguns dos mais clássicos algoritmos da área de PCG. Isso ocorreu devido ao fato de que esses algoritmos clássicos ainda são bastante utilizados em jogos modernos.

2.4.2.1 Ruídos

Ruídos são um modo de se organizar números gerados de forma aleatória visando criar padrões que sejam úteis para a geração de outros tipos de conteúdo. Por si só, os ruídos não apresentam algo com o qual o jogador possa interagir, mas podem ser utilizados para adicionar detalhes a outros tipos de conteúdo, de forma a aumentar o realismo dos gráficos.

Visto que muitos dos algoritmos para a geração de outros tipos de conteúdo comumente se utilizam dos algoritmos de geração de ruídos, estes serão tratados antes das outras categorias de algoritmos. Nesta subseção, são tratados os ruídos de (i) valor; (ii) gradiente; (iii) convolução de grade; (iv) convolução esparsa; (v) Perlin; (vi) OpenSimplex; e (vii) Gabor. O funcionamento de cada algoritmo será tratado no Capítulo 3. A seguir, são descritas as propriedades de cada ruído.

O ruído de **valor** (i) gera ruídos simples, garantindo que o ruído é limitado quanto à banda (*bandlimited*) [44]. O ruído de **gradiente** (ii) garante que o ruído seja limitado quanto à banda e que o valor do ruído seja 0 (zero) para todos os pontos que tenham coordenadas inteiras na grade [44]. O ruído de **convolução de grade** (iii)

gera resultados com menos artefatos alinhados aos eixos, e ainda permite certo grau de controle espectral por meio do uso de diferentes matrizes de convolução [44]. O ruído de **convolução esparsa** (iv) obtém resultados com menos alinhamento à grade por meio da convolução de um filtro sobre um conjunto de impulsos aleatórios, e, da mesma forma que o ruído de convolução de grade, permite o controle espectral por meio da escolha da matriz de convolução [44]. O ruído de **Perlin** (v) foi criado para o filme *Tron*, em 1982, com o objetivo de aumentar o realismo de texturas aplicadas a modelos virtuais, tendo sido o primeiro trabalho a propor uma função tridimensional, eliminando a necessidade do mapeamento da textura nas superfícies [107]. O ruído **OpenSimplex** (vi) foi criado por Kurt Spencer, baseado no ruído Simplex, de Ken Perlin. Assim como o original, este algoritmo tem como objetivo gerar ruídos com menos artefatos alinhados à grade e maior desempenho que o ruído de Perlin. Embora o desempenho do ruído OpenSimplex seja menor que o Simplex, foi selecionado o OpenSimplex neste trabalho devido à patente que limita o uso do ruído Simplex [123]. O ruído de **Gabor** (vii) se propõe a gerar ruídos com controle espectral preciso, fáceis de serem mapeados em superfícies e com filtragem anisotrópica de alta qualidade [76].

2.4.2.2 Nomes

Geradores de nomes são utilizados para criar cadeias de caracteres que se pareçam com nomes reais. Essas cadeias são utilizadas para que os jogadores se recordem de locais ou personagens importantes nos jogos. Nesta subseção, são tratados um gerador de nomes baseado em máquina de estados finitos e duas técnicas baseadas em Cadeias de Markov.

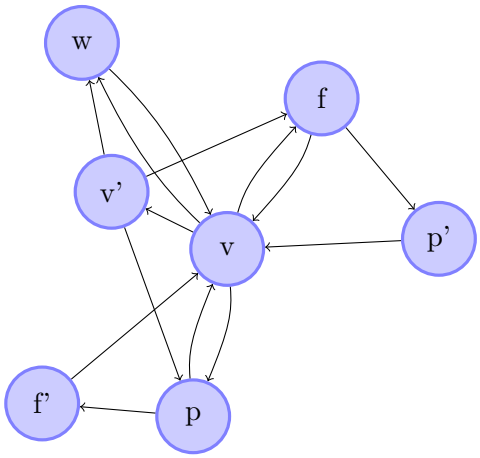
A técnica de geração de nomes baseada em máquina de estados finitos utiliza as regras da Figura 2.20 para gerar o resultado, não necessitando de qualquer pré-processamento e utilizando pouca memória RAM. As transições de estado ocorrem com base em uma escolha aleatória dentre os possíveis futuros estados. Essa técnica é uma simplificação das técnicas baseadas em Cadeias de Markov.

Duas técnicas baseadas em Cadeias de Markov foram tratadas nessa pesquisa. Uma delas utiliza conjuntos de estatísticas diferentes para letras que iniciam ou terminam o nome, em adição às estatísticas gerais, utilizando esses dados especiais quando se inicia ou termina uma palavra. A outra técnica utiliza Cadeias de Markov com ordem n, onde n é escolhido pelo usuário. Essa variação das Cadeias de Markov utiliza os n estados anteriores como base para a função de probabilidade da transição seguinte. Essas modificações tem o objetivo de obter resultados mais realistas.

2.4.2.3 Vegetação

Geradores de vegetação são utilizados para criar modelos de árvores, arbustos e outros tipos de flora. Esta é uma das categorias de conteúdo procedural que mais tem sucesso comercial, com geradores como *SpeedTree* IDV Inc. [66] sendo utilizado em centenas de jogos e filmes, e também em simuladores militares e softwares de arquitetura. Esses geradores tem como objetivo eliminar um dos processos mais lentos do *pipeline* de

Figura 2.20: Máquina de estados finitos para a geração de nomes. A máquina é iniciada a partir da escolha de um estado aleatório dentre {v, f, p, w}. A máquina produz uma saída de acordo com cada estado: vogal, se {v, v'}; consoante fricativa, se {f, f'}; consoante plosiva, se {p, p'}; e, "estranha", se {w}.



produção de conteúdo: a geração manual de modelos e texturas de vegetação. Essa geração manual é tediosa e não produz bons resultados, visto que não é possível gerar manualmente centenas de modelos diferentes. Nesta subseção, são tratados um algoritmo baseado em subdivisão aleatória de ramos, um algoritmo baseado em sistemas de funções iterativas aleatórias e um algoritmo baseado em L-*systems*.

A técnica baseada em subdivisão aleatória de ramos calcula, a partir do tronco da árvore, ângulos pseudo-aleatórios para os galhos a serem gerados, e, para cada galho, repete o procedimento de forma recursiva, gerando uma representação tridimensional da árvore produzida.

Já a técnica baseada em sistemas de funções iterativas aleatórias utiliza as fórmulas

$$x_{n+1} = ax_n + by_n + e$$

$$y_{n+1} = cx_n + dy_n + f$$

para calcular os pontos a serem selecionados, gerando uma imagem bidimensional que representa uma planta.

Por fim, a última técnica utiliza L-*systems* para a geração de um modelo tridimensional da vegetação produzida.

2.4.2.4 Simulação de elementos naturais

Os geradores de elementos naturais proveem resultados plausíveis para a exibição de fogo, fumaça, água, nuvens etc. sem que seja necessária a execução de algoritmos baseados em teorias físicas, que são bastante lentos. Esses geradores trabalham com a aproximação dos resultados de forma a se obter uma imagem que se pareça com o

elemento na natureza, mas que possa ser renderizada com processamento menor do que os modelos fisicamente corretos. Nesta subseção, são tratados um algoritmo para a renderização de fogo, um renderizador de fumaça, um de oceanos e um de nuvens.

O algoritmo para renderização de fogo utiliza ruído de fluxo para simular as chamas, gerando uma aproximação das chamas na forma de uma imagem bidimensional.

O renderizador de fumaça simula um conjunto de partículas, sendo que cada partícula tem uma posição e velocidade, permitindo um maior controle sobre o resultado final. Essas partículas são renderizadas como pontos texturizados.

O algoritmo para renderização de oceanos utiliza ruído fractal para simular a superfície do oceano, e gera uma imagem bidimensional da superfície, simulando, inclusive, o reflexo do sol na água.

O algoritmo para renderização de nuvens utiliza ruído Simplex para a geração de uma imagem bidimensional em escala de cinza. Esta imagem pode, então, ser aplicada à renderização do céu para simular a presença de nuvens.

2.4.2.5 Cavernas, labirintos e masmorras

Geradores de mapas internos estão entre os mais utilizados em jogos digitais. Isso se deve, principalmente, ao fato de que esses mapas são inerentemente limitados, o que previne que se tenha que implementar um limite de mapa. Muitos jogos utilizavam esses mapas devido às limitações das plataformas em que foram desenvolvidos, que impediam o uso de mapas externos mais complexos. Outros jogos utilizam esses mapas como elemento principal, sendo sua navegação um dos objetivos do jogador. Nesta subseção, são tratados dois geradores de cavernas, um de labirintos e dois de masmorras.

O primeiro gerador de cavernas tratado utiliza autômatos celulares. A técnica inicia com uma grade preenchida com um estado aleatório, onde as células podem ser passáveis ou não. A partir desse estado, o algoritmo executa uma ou mais iterações dos autômatos, de acordo com regras definidas pelo usuário.

O segundo gerador de cavernas se baseia em agregação por difusão limitada. Para isso, são utilizados *random walks* para a geração de grupos de células passáveis, partindo de uma grade com apenas uma célula passável. Essa técnica evita que sejam geradas áreas não acessíveis no mapa, o que pode ocorrer no gerador baseado em autômatos celulares.

O gerador de labirintos tratado nessa pesquisa é baseado em agentes para a geração do labirinto. Assim como a geração de cavernas por agregação por difusão limitada, o estado inicial é uma grade com apenas uma célula passável, de forma a evitar áreas inacessíveis. A partir desta célula, são criados agentes que "perfuram" as áreas impassáveis, criando caminhos que formam o labirinto.

Os dois geradores de masmorras utilizam o mesmo princípio de criação, selecionando uma célula aleatória na grade e definindo um tamanho para as salas. Eles se diferem na forma como geram conexões entre as salas e corredores. A primeira técnica parte de uma sala aleatória, gerando uma saída para ela e, a partir dessa saída, gera uma sala ou corredor; a segunda técnica cria as saídas no momento da geração da sala e utiliza

uma lista, com essas saídas, para criar, aleatoriamente, as salas e corredores que serão ligadas a estas saídas.

2.4.2.6 Mapas de elevação

Mapas de elevação são formas de se representar terrenos tridimensionais em que apenas a superfície seja necessária. Nestes mapas, um terreno é dividido em uma grade bidimensional e, para cada ponto dessa grade, é atribuído um valor de elevação. Esses mapas podem ser obtidos por meio da medição de terrenos reais, ou gerados com o uso de técnicas procedurais. Nesta pesquisa, foram analisadas duas técnicas para geração procedural, ambas baseadas em fractais.

Uma das técnicas é baseada no algoritmo diamante-quadrado, que subdivide uma grade, gerando valores de elevação com base na média dos pontos próximos acrescida de um valor aleatório, de forma a gerar o mapa de elevação.

A outra técnica é baseada na bisseção de um mapa esférico, elevando-se um dos lados com relação ao outro. Com a repetição desse processo, é obtido um terreno que simula as falhas geográficas causadas pelos movimentos de placas tectônicas.

2.4.2.7 Cidades e ruas

Mapas de cidades e redes de ruas são frequentemente utilizados em jogos nos quais o jogador atua em cenários ao ar livre. As cidades podem ser utilizadas como ambientação ao cenário, como locais onde o jogador pode utilizar serviços prestados por personagens não-jogáveis, ou ainda como ponto de encontro entre jogadores. Já as redes de ruas auxiliam o jogador a se orientar no espaço do jogo, sem que ele seja obrigado a consultar um mapa para chegar ao destino desejado. Elas também são utilizadas para aumentar o realismo das cidades, e, por isso, esses dois tipos de conteúdo geralmente são gerados em conjunto.

A primeira técnica utiliza agentes para a geração de ruas e edifícios. Esses agentes caminham de forma aleatória sobre o espaço livre do mapa, tentando estender e conectar ruas, além de encontrar espaços onde os edifícios possam ser inseridos.

A segunda técnica gera as ruas utilizando L-*system*, utilizando como entrada um mapa de elevação e um mapa de densidade populacional, de forma a gerar ruas mais parecidas com as de cidades reais.

A última técnica gera as ruas por meio de ramificações aleatórias, e, após essa geração, aplica um L-*system* para a geração de edifícios nos lotes delimitados pelas ruas.

2.5 Trabalhos relacionados

Dado o interesse científico, encontram-se na literatura da área algumas pesquisas realizadas com o objetivo de listar e classificar os diversos tipos de algoritmos de PCG. Exemplos incluem as pesquisas de Togelius et al. [137], que criam uma taxonomia para algoritmos baseados em busca; Kelly e McCabe [71], com uma pesquisa sobre técnicas

para geração de cidades; Carli et al. [29], com pesquisas sobre geração de terrenos, continentes, estradas, rios e cidades; Lagae et al. [77], com uma pesquisa sobre funções para geração de ruído procedural; Smelik et al. [120], com uma pesquisa sobre métodos procedurais para modelagem de terrenos; e Hendrikx et al. [62], que propõem uma taxonomia e classificam as principais técnicas de geração dos diversos tipos de conteúdo.

Carli et al. [29] apresentam uma pesquisa sobre técnicas para a geração procedural de terrenos, litorais, rios, estradas e cidades, e as classificam em assistidas ou não-assistidas. Os autores agrupam as técnicas estudadas de acordo com o tipo de conteúdo gerado, classificam essas técnicas com base na métrica proposta e descrevem cada uma das técnicas, ilustrando-as com imagens que exibem o resultado visual de cada técnica. Os autores ainda afirmam que as técnicas assistidas são melhores aplicadas em casos nos quais é necessário maior controle sobre o resultado obtido, embora não sejam utilizáveis em casos nos quais o conteúdo deva ser gerado *online*; enquanto as técnicas não-assistidas podem ser usadas *online*, mas são mais voltadas para conteúdo no qual o resultado não impacta diretamente a jogabilidade, visto que não se pode exercer muito controle sobre as mesmas.

Hendrikx et al. [62] apresentam uma pesquisa abrangente sobre PCG, na qual os autores propõe uma classificação para os diferentes tipos de conteúdos necessários para a criação de jogos digitais (Subseção 2.2) e outra classificação quanto ao modo como o conteúdo é gerado (Seção 2.3). Além da criação dessas duas categorias, os autores classificam diversos exemplos de PCG nessas categorias e listam, dentre as recomendações para pesquisas futuras, o estudo do balanço (*trade-off*) entre o nível de detalhamento dos resultados e o desempenho dos algoritmos para gerá-lo.

Kelly e McCabe [71] apresentam uma pesquisa sobre técnicas para a geração procedural de cidades, classificando essas técnicas de acordo com os critérios identificados pelos autores (conforme descrito em 2.4). Nessa pesquisa, são tratadas algumas técnicas para geração de conteúdo que são usadas como base para outros algoritmos mais complexos: fractais, que são formas matemáticas autossimilares; *L-Systems*, que são gramáticas formais utilizadas para descrever estruturas de plantas; ruído de Perlin, que é usado para gerar variações em texturas que seriam "planas", de forma a obter mais realismo; *tiling*, que é utilizado para criar grandes áreas por meio do uso de pequenas partes (*tiles*); e texturas baseadas em diagramas de *Voronoi*, que são utilizadas para a geração de texturas com padrões celulares. Os autores, após classificarem algumas técnicas de geração de cidades, identificam áreas de possível interesse para pesquisas futuras, destacando que técnicas para geração de cidades que venham a ser desenvolvidas devem levar em consideração os seguintes pontos: acessibilidade, onde as técnicas não devem necessitar da entrada de dados complexos, facilitando o uso por usuários leigos; interatividade, de forma que as técnicas tenham resultados controláveis, facilitando a obtenção de conteúdo adaptado para as necessidades do usuário; e execução em tempo real, de forma que o conteúdo possa ser renderizado em hardware comum e em tempo aceitável para o uso de forma interativa.

Lagae et al. [77] apresentam uma pesquisa sobre funções para geração procedural de

ruídos, classificando os ruídos resultantes quanto à densidade espectral e à amplitude de distribuição. As funções para geração de ruído tem como objetivo gerar variações que podem ser utilizadas para adicionar detalhes às imagens renderizadas, com o menor consumo de memória possível, devido aos limites da época em que essas técnicas começaram a ser desenvolvidas. Os autores citam o fato de a velocidade de processamento ter aumentado muito mais do que a velocidade de acesso à memória como justificativa para o uso dessas técnicas atualmente, sendo que, muitas vezes, é mais rápido gerar o conteúdo quando necessário do que armazenar e recuperar os resultados já calculados. Com essa pesquisa, os autores concluem que a escolha da melhor técnica de geração de ruído é dependente das necessidades de cada aplicação. Os autores ainda citam alguns objetivos para possíveis trabalhos futuros, principalmente no sentido de criar maior controle sobre os resultados obtidos, mas também afirmam que seria interessante a comparação do desempenho das diferentes técnicas, tanto em CPUs quanto em GPUs.

Smelik et al. [120] apresentam uma pesquisa sobre métodos procedurais para a modelagem de terrenos e os classificam de acordo com o realismo dos resultados, desempenho de geração e o nível de controle que o usuário tem sobre o resultado. Os autores dividem as técnicas de geração em grupos: mapas de elevação, corpos d'água, modelos de plantas e vegetação, redes de ruas e ambientes urbanos. As técnicas tratadas são descritas e, em alguns casos, o desempenho dos algoritmos é citado. Os autores afirmam que desses grupos de técnicas de geração, a que menos tem pesquisas é a de corpos d'água. Também foi destacado que essas técnicas têm se tornado cada vez mais interativas e seus desempenhos aumentados, também com o uso de GPUs.

Togelius et al. [137] apresentam uma taxonomia e uma pesquisa das técnicas de geração de conteúdo baseadas em busca, classificando essas técnicas de acordo com os critérios identificados pelos autores (conforme descrito em 2.4). No estudo, os autores diferenciam as técnicas de PCG convencionais das baseadas em busca, sendo que as últimas têm um componente evolutivo nos algoritmos que tentam gerar o conteúdo mais adequado para cada caso de forma automática, com base em parâmetros predefinidos. Com essa pesquisa, os autores tentam identificar os maiores desafios na PCG baseada em busca, sendo que alguns dos pontos levantados podem ser aplicados à PCG de forma geral: quais tipos de conteúdo são adequados a serem gerados; como evitar *falhas catastróficas*; como aumentar a velocidade de geração; como melhor representar o conteúdo para jogos; e como avaliar melhor a qualidade dos geradores de conteúdo.

Capítulo 3

Método

Os algoritmos selecionados (listados na Seção 2.4.2) foram implementados utilizando as linguagens C++ [128] e, quando aplicável, GLSL [134], com os resultados sendo exibidos em OpenGL [73]. Para essa implementação, os algoritmos foram obtidos já implementados, de forma a se ter a garantia de que funcionam corretamente. Após a implementação em C++, o código foi otimizado por meio do uso de *profilers*, focando os esforços de otimização nos *hot spots* do código. Esse processo de otimização foi realizado "*best effort*", não sendo garantida a melhor otimização possível.

Após a implementação, foram definidos os critérios de qualidade de resultados para cada tipo de conteúdo. Finalmente, os dados de tempo de execução e os resultados dos algoritmos foram coletados e analisados, gerando uma comparação entre os algoritmos de PCG que produzam conteúdos de tipos equivalentes.

3.1 Metodologia utilizada

Neste trabalho são aplicados os seguintes métodos para a obtenção de dados para a comparação das técnicas de PCG: análise de complexidade, medição do tempo de execução e análise do consumo de memória.

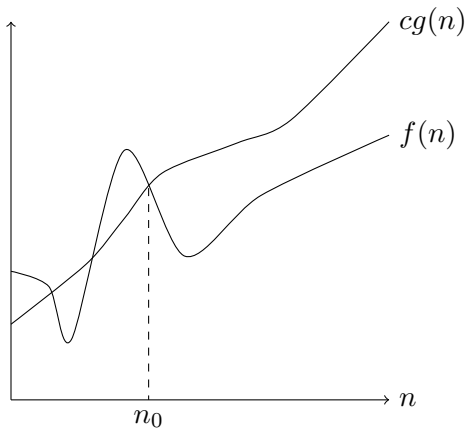
3.1.1 Análise de complexidade - big$O$

A notação big-O define o limite superior assintótico do tempo de execução de determinado algoritmo, provendo uma estimativa do comportamento deste algoritmo quanto à variação do conjunto de entrada. Big-O é usada para denotar o cenário de pior caso de um algoritmo [35]. A definição matemática da notação big-O é dada por:

Definição 1. *$f(n)$ é $O(g(n))$ se existem números positivos c e N tais que $f(n) \leq cg(n)$ para todo $n \geq N$.*

A Figura 3.1 demonstra a definição de forma gráfica.

Figura 3.1: Exemplo gráfico da notação big-O.



3.1.2 Medição do tempo de execução

As medições dos tempos de execução em C++ foram realizadas por meio do uso da biblioteca `chrono`, disponível na STL a partir da versão C++11. Para medir o tempo de execução de determinado algoritmo, é criada uma variável que armazena um ponto no tempo. Em seguida, o algoritmo é executado e um novo ponto no tempo é criado. Após isso, basta subtrair o tempo final do inicial e tem-se o tempo gasto, em nanosegundos. A Listagem 3.1 mostra como o código foi implementado.

Listagem 3.1: Uso da biblioteca `chrono` para calcular o tempo de execução de um algoritmo em C++.

```
using namespace std::chrono;
auto inicio = high_resolution_clock::now();
// Executar o algoritmo...
auto fim = high_resolution_clock::now();
auto duracao = duration_cast<nanoseconds>(fim - inicio).count();
```

As medições de tempo em GLSL foram realizadas com o uso de `GL_TIMESTAMP`, da API do OpenGL. O método é semelhante ao de C++ e é exibido na Listagem 3.2. Após a medição, subtraindo os *timestamps*, tem-se o tempo gasto em nanosegundos.

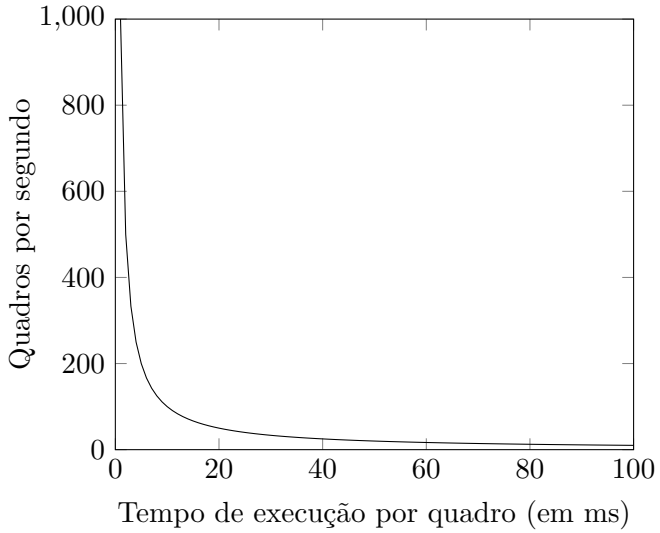
Listagem 3.2: Uso da API do OpenGL para calcular o tempo de execução de um algoritmo em GLSL.

```
GLuint consultas[2];
glGenQueries(2, consultas);
```

```
glQueryCounter(consultas[0], GL_TIMESTAMP);
// Executar o algoritmo...
glQueryCounter(consultas[1], GL_TIMESTAMP);
GLuint64 inicio;
glGetQueryObjectui64v(consultas[0], GL_QUERY_RESULT, &inicio);
GLuint64 fim;
glGetQueryObjectui64v(consultas[1], GL_QUERY_RESULT, &fim);
glDeleteQueries(2, consultas);
GLuint64 duracao = fim - inicio;
```

O tempo de execução por quadro foi escolhido em detrimento à quantidade de quadros por segundo devido a não-linearidade desta medida (Figura 3.2), que faz com que o tempo de execução seja a medida mais adequada para fins de otimização de algoritmos [4].

Figura 3.2: Tempo de execução por quadro vs quadros por segundo.



Os tempos de execução foram mensurados em um PC com processador Intel®Core™ i7-4790K, com 4 núcleos e *hyperthreading* com frequência de 4GHz quando executando algoritmos em vários núcleos em paralelo e 4.4GHz quando executando em apenas um núcleo; memória RAM DDR3 *dual channel* com 16GB e frequência de 1600MHz; e *chipset* Intel®H97. A GPU usada foi uma NVIDIA®GeForce GTX TITAN X, com 3072 núcleos com frequência base de 1000MHz, 12GB de memória RAM GDDR5, com frequência de 7Gbps e 384 bits de largura de banda, no padrão PCI Express 3.0 x16. Todos os testes que utilizam OpenGL foram executados em hardware (utilizando a GPU TITAN X) para obter o melhor desempenho possível.

O sistema operacional utilizado foi o Arch Linux, com *kernel* 4.7.2-1-ARCH. O

compilador utilizado foi o GCC, versão 6.2.1, com nível de otimização $-O3$ e o driver da GPU utilizado foi o proprietário da NVIDIA, com versão 370.28.

3.1.3 Análise do uso de memória

O uso de memória foi calculado a partir da análise dos códigos dos algoritmos, com o objetivo de totalizar toda a memória necessária para a execução do algoritmo, levando em consideração apenas as variáveis utilizadas para as estruturas de dados necessárias aos algoritmos, sem totalizar os custos relacionados às chamadas de métodos, *v-tables*, *stack*, dentre outros custos intrínsecos à linguagem C++. Os tamanhos, em bytes, dos tipos utilizados para o cálculo de uso de memória são mostrados na Tabela 3.1.

Tipo	Bytes
`char`	1
`unsigned char`	1
`short`	2
`unsigned short`	2
`int`	4
`unsigned int`	4
`long`	8
`unsigned long`	8
`float`	4
`double`	8

Tabela 3.1: Tamanhos, em bytes, das variáveis básicas em C++.

3.2 Bits

3.2.1 Ruídos

Para se obter a velocidade dos algoritmos na CPU, foram mensurados os tempos necessários para se gerar matrizes quadradas de pixels, utilizando todos os núcleos da CPU. A técnica utilizada para a medição é mostrada na Listagem 3.1. Para a velocidade na GPU, os algoritmos foram implementados utilizando a mesma estrutura da versão em CPU, sendo que a inicialização dos algoritmos e a geração dos números aleatórios são realizadas na CPU. A técnica de medição é mostrada na Listagem 3.2. Todas as medições foram realizadas apenas na geração dos valores de cada pixel, sendo descartado o tempo necessário para a inicialização dos vetores de dados dos geradores de ruído. Foram comparadas apenas as versões tridimensionais dos algoritmos, exceto no caso do ruído de Gabor, no qual a implementação disponível era bidimensional.

O ruído de valor é gerado por meio da criação de uma grade com valores aleatórios. Para cada ponto nessa grade, o valor do ruído é computado a partir da interpolação desses valores. Esses códigos foram baseados na implementação de Ebert et al. [44].

O ruído de gradiente também é gerado por meio da criação de uma grade, na qual cada ponto armazena um vetor gradiente. Esses vetores são, então, utilizados para a geração do valor do ruído. Esses códigos foram baseados na implementação de Ebert et al. [44].

O ruído de convolução de grade é um método que se propõe a corrigir artefatos alinhados a eixos que ocorrem nos ruídos baseados em grades. Para isso, utiliza-se um filtro de convolução, o que faz o algoritmo levar em consideração os pontos vizinhos ao ponto analisado. Esses códigos foram baseados na implementação de Ebert et al. [44].

O ruído de convolução esparsa é uma técnica que não utiliza a grade de valores aleatórios, empregando ao invés dessa um processo de convolução sobre impulsos aleatórios. Esses códigos foram baseados na implementação de Ebert et al. [44].

O ruído de Perlin é um tipo de ruído de gradiente desenvolvido para melhorar as imagens geradas por computador no filme *Tron*. Esses códigos foram baseados na implementação original de Perlin [108].

O ruído OpenSimplex é uma implementação criada para contornar limitações da patente do ruído simplex. O ruído simplex é uma evolução do ruído de Perlin e visa reduzir artefatos alinhados a eixos e aumentar o desempenho em alguns casos. Esses códigos foram baseados na implementação de Spencer [123].

O ruído de Gabor é um tipo de ruído de convolução esparsa que utiliza um *kernel* de Gabor para a convolução. Esses códigos foram baseados na implementação de Lagae et al. [76].

3.2.2 Nomes

Os tempos de execução dos geradores de nomes foram mensurados por meio da criação de 100000 nomes de forma sequencial. Também foram medidos os tempos de inicialização dos algoritmos que processam arquivos de entrada. Esses arquivos foram lidos de um SSD Kingston HyperX 3K de 120 GB com sistema de arquivos EXT4. Foram medidos os tempos com dois arquivos: um com nomes de lugares, contendo 10196 nomes e 99443 bytes e outro com nomes de pessoas, contendo 21986 nomes e 179353 bytes.

As três técnicas tratadas nessa pesquisa são: baseada em máquina de estados finitos, que foi implementada com base no código proposto por Lait [78]; baseada em cadeias de Markov, com estatísticas separadas para letras iniciais e finais, implementada com base no código proposto por Marczuk [89]; e, baseada em cadeias de Markov com ordem n, implementada com base no código proposto por Calabuig [27].

Devido à natureza dos algoritmos desta subseção, não é possível paralelizar os mesmos sem que haja perda de suas propriedades determinísticas.

3.2.3 Vegetação

Para a medição do tempo de execução, cada implementação foi executada cem vezes e foi extraída a média do tempo de execução. O conteúdo gerado por cada uma das técnicas foi o mesmo em cada uma das cem execuções, visto que o objetivo das repetições é garantir que fatores externos à geração sejam eliminados do cálculo.

O gerador baseado em subdivisão aleatória dos ramos foi implementado com base no código proposto por Komppa [75]. Para os testes, os parâmetros utilizados foram os parâmetros *default* do código original.

No gerador baseado em sistemas de funções iterativas, a implementação foi baseada no código de Bourke [24]. Os parâmetros utilizados nos testes são os mesmos que o autor do código original utilizou.

Já a técnica baseada em L-*systems* teve sua implementação baseada no código de Naderi-Afooshteh [99]. Os parâmetros utilizados para os testes são os definidos pelo autor original para gerar uma árvore.

3.2.4 Simulação de elementos naturais

Devido à natureza extremamente paralelizável destas simulações, as implementações foram realizadas em GLSL. Para o cálculo do tempo de execução por quadro, foi medida a quantidade de quadros gerados em dez minutos de execução para cada técnica.

O gerador de fogo baseado em ruído de fluxo foi implementado com base no código proposto por Mellblom [93]. Os parâmetros utilizados foram definidos empiricamente utilizando a interface gráfica da implementação original.

O gerador de fumaça baseado em partículas foi implementado com base no código disponibilizado por ARM MALI [5]. Os parâmetros utilizados para os testes foram os mesmos utilizados no código original.

Para o gerador de oceanos, a implementação foi baseada no código disponibilizado por Alekseev [2]. Os parâmetros utilizados nos testes são os mesmos que no algoritmo original.

Por fim, o gerador de nuvens foi implementado com base no algoritmo proposto por Elias [46]. Os parâmetros utilizados foram os mesmos propostos no algoritmo original.

3.3 Espaço

3.3.1 Cavernas, labirintos e masmorras

Para a obtenção dos tempos de execução, foi mensurada a geração de mapas quadrados com diferentes tamanhos. Dada a natureza dos algoritmos, com exceção da técnica de autômatos celulares, não é trivial paralelizar os mesmos sem que ocorra a perda do determinismo dos algoritmos.

O gerador de cavernas baseado em autômatos celulares foi implementado com base no código proposto por Babcock [8]. Para os testes, foram escolhidos como parâmetros além do tamanho dos mapas, 40% de probabilidade de preenchimento da grade inicial com terrenos não-passáveis; e, cinco iterações dos autômatos celulares, utilizando como regras: existe um terreno não-passável numa célula se o número de células vizinhas com terrenos não-passáveis for maior que ou igual a cinco (considerando as oito células adjacentes)ou menor que ou igual a 2 (considerando as vinte células mais próximas).

Para o gerador de cavernas baseado em agregação por difusão limitada, a implementação tomada como base foi proposta por Inman [68]. Além do tamanho do mapa, os

testes também utilizaram como parâmetros de entrada 50% de preenchimento do mapa com terrenos passáveis e o parâmetro falso para a opção de gerar corredores diagonais.

O gerador de labirintos foi implementado com base no código de Debski [40]. Esta técnica não utiliza qualquer parâmetro extra.

Os dois geradores de masmorras foram implementados com base nos códigos de Wallace e Mtvee [144] e MindControlDX, netherh e underww [94]. O primeiro recebeu como parâmetros adicionais os valores 110 para o controle de tamanho, 50% para o controle de produção de salas ou corredores e 128 para o máximo de salas geradas. O segundo gerador recebeu como parâmetro extra o valor 2000 como quantidade máxima de objetos a serem gerados.

3.3.2 Mapas de elevação

Similarmente aos algoritmos para a geração de cavernas, labirintos e masmorras, para a obtenção dos tempos de execução, foi mensurada a geração de mapas quadrados com diferentes tamanhos. Dada a natureza dos algoritmos, não é trivial paralelizar os mesmos sem que ocorra a perda do determinismo dos algoritmos.

O gerador de mapa de elevação baseado no algoritmo diamante-quadrado foi implementado com base no código proposto por Martz [90]. Além do tamanho, são usados como parâmetros um fator de escala, de 1.0, que determina quais são os valores mínimos e máximos do mapa de elevação, e um fator que determina o nível de rugosidade do mapa, de 0.5.

Já o gerador baseado em bisseção de mapas esféricos foi implementado com base no código de Olsson e drow [105]. Este algoritmo tem como parâmetros extras a quantidade de bisseções a serem realizadas, para o qual o valor usado foi 5000, a porcentagem de água do mapa de 50% e a porcentagem de gelo, que, para a qual, foi usado 12%.

3.4 Sistemas

3.4.1 Cidades e ruas

Para o gerador de ruas e edifícios baseado em agentes, foi realizada uma execução e medido o tempo, visto que a execução é bastante lenta. O código utilizado foi implementado com base no original por Wymer [151]. Os parâmetros de execução são os mesmos que o do código original.

O gerador de ruas baseado em L-*systems* foi implementado com base no código de Engilberge e Depraz [48]. Os parâmetros de entrada utilizados foram os mesmos disponibilizados pelos autores originais.

Por fim, o gerador de ruas e edifícios baseado em L-*systems* foi implementado com base no código original proposto por Craighead, Tanoi e Bryers [36]. Os parâmetros utilizados foram os listados pelos autores originais.

Capítulo 4

Experimentos e Resultados

Neste capítulo, são discutidos os conteúdos produzidos e as análises empregadas, de forma a se obter dados que possam ser utilizados para melhor compreensão das técnicas tratadas. Nenhuma das análises visa obter uma resposta definitiva sobre qual técnica de PCG deve-se utilizar, sendo que cada caso terá uma resposta específica de acordo com as necessidades de cada aplicação.

4.1 Ruídos

Os resultados obtidos a partir de cada técnica de geração de ruídos são mostrados a seguir.

4.1.1 Big O

Para a análise da complexidade dos algoritmos, foram considerados apenas os custos relativos à geração do ruído. Os custos de inicialização das tabelas utilizadas foram descartados, dado que os mesmos são necessários apenas uma vez, enquanto a geração do ruído é executada centenas de milhares de vezes – ou mais, dependendo da aplicação. As inicializações executam em tempo constante, podendo ser, inclusive, realizadas em tempo de compilação. Esta análise se aplica tanto à versão em C++ quanto à em GLSL.

Para a geração do **ruído de valor**[1], os métodos auxiliares `perm`, `index`, `vlattice` e `spline` executam em tempo constante $O(1)$. O método `noise` tem três `for` aninhados (linhas 56-58) que executam de $[-1; 2]$. Logo, o método executa em tempo constante $O(1)$.

No que tange ao **ruído de gradiente**[2], os métodos auxiliares `lerp`, `smoothStep`, `perm`, `index` e `glattice` executam em tempo constante $O(1)$, portanto, o método `noise` também executa em tempo constante $O(1)$.

[1] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/noise/value_noise.cpp

[2] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/noise/gradient_noise.cpp

O **ruído de convolução de grade**[3] tem como métodos auxiliares `catrom2`, `perm`, `index` e `vlattice`, que executam em tempo constante $O(1)$. O método `noise` tem três `for` aninhados (linhas 75, 78 e 81) que executam de $[-1; 2]$, executando em tempo constante $O(1)$.

Para o **ruído de convolução esparsa**[4], os métodos auxiliares `next`, `catrom2`, `index` e `perm` executam em tempo constante $O(1)$. O método `noise` tem quatro `for` aninhados (linhas 54-56 e 58), sendo que os três primeiros executam de $[-2; 2]$ e, o último, executa de $[3; 0)$. Neste caso, o método executa em tempo constante $O(1)$.

O **ruído de Perlin**[5] utiliza os métodos auxiliares `lerp`, `s_curve`, `setup`, `at` e `normalize3`, que executam em tempo constante $O(1)$. Neste caso, o método `noise` também executa em tempo constante $O(1)$.

Para gerar o **ruído OpenSimplex**[6], são utilizados os métodos auxiliares `fast_floor` e `extrapolate`, que executam em tempo constante $O(1)$. Novamente, o método `noise` também executa em tempo constante $O(1)$.

Por fim, para o **ruído de Gabor**[7] é empregado o método auxiliar `morton`, que executa em tempo constante $O(1)$; o método auxiliar `cell`, que tem um `for` (linha 82) que varia de acordo com os parâmetros iniciais escolhidos pelo usuário (na implementação original $[0; n)$, com $n \in [0; 13]$), e também utilizando o método de geração de números pseudo-aleatórios `poison`, executando em tempo linear $O(n)$. O método `noise` tem dois `for` aninhados (linhas 105-106) que executam de $[-1; 1]$, executando em tempo linear $O(n)$.

A análise é resumida na Tabela 4.1. A partir da análise, nota-se que, com a exceção do ruído de Gabor, os ruídos executam em tempo constante, independente dos parâmetros passados para sua criação. O tempo de execução do ruído de Gabor depende dos parâmetros passados de forma linear.

Como essas análises foram realizadas para apenas um ponto no ruído, é esperado que o aumento do tempo de execução dependa linearmente da quantidade de pontos calculados, exceto no caso do ruído de Gabor, cujo tempo varia de forma quadrática.

4.1.2 Tempo de execução

Os tempos de execução foram medidos durante a geração de texturas quadradas, com tamanhos de lado s tal que $\{s : 2^n, n \in \mathbb{N}, n \in [8; 14]\}$. As texturas foram geradas em C++ e GLSL.

Após o cálculo dos tempos para a geração de cada textura (Tabelas 4.2 e 4.3, Figuras 4.1 e 4.2), os tempos foram divididos pela quantidade de pixels gerados, de forma a se

[3] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/noise/lattice_convolution.cpp

[4] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/noise/sparse_convolution.cpp

[5] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/noise/perlin_noise.cpp

[6] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/noise/opensimplex.cpp

[7] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/noise/gabor.cpp

Ruído	Big-O
Valor	$O(1)$
Gradiente	$O(1)$
Convolução de grade	$O(1)$
Convolução esparsa	$O(1)$
Perlin	$O(1)$
OpenSimplex	$O(1)$
Gabor	$O(n)$

Tabela 4.1: Complexidade dos métodos de ruído.

obter o custo aproximado de tempo para se gerar um pixel, e, a partir desses valores, foram obtidas as médias exibidas na Tabela 4.4 e na Figura 4.3. É importante ressaltar que os tempos das tabelas e figuras estão em prefixos diferentes (ms para as primeiras e ns para a última) devido à variação da magnitude dos valores. Também é por este motivo que foi utilizada a escala logarítmica nas figuras.

Devido à natureza paralelizada da GLSL e ao fato de os tempos em C++ terem sido calculados utilizando vários núcleos do processador, esse tempo por pixel é menor do que o tempo real gasto para o cálculo do valor de um pixel em um núcleo. No entanto, foi adotado este critério de medição devido ao fato que quase sempre ruídos são utilizados para a geração de dezenas de milhares (ou mais) de pixels. O custo final de se gerar determinada textura é melhor aproximado utilizando o tempo mensurado nas versões paralelizadas, ao invés das versões não paralelas.

Ruído	Tempo (em ms) por tamanho de textura						
	256	512	1024	2048	4096	8192	16384
Valor	4.609	22.325	83.037	243.908	852.378	2852.638	11145.596
Gradiente	0.968	2.778	9.763	28.528	109.328	405.850	1573.440
Convolução de grade	4.648	11.712	44.588	181.071	664.610	2565.903	10050.759
Convolução esparsa	38.450	122.280	457.605	1762.663	6863.169	27053.967	107179.826
Perlin	0.752	2.594	6.157	16.886	72.563	260.997	1012.891
OpenSimplex	1.448	4.827	13.502	46.137	176.046	673.369	2660.113
Gabor	87.584	320.471	1260.530	4918.890	19223.873	76790.381	304809.111

Tabela 4.2: Tempo de execução, por textura, dos métodos de ruído em C++.

Com base nesses valores, pode-se concluir que é sempre mais rápido gerar os ruídos na GPU, mesmo que o resultado tenha que ser processado pela CPU após a geração. Isso se deve ao fato de que a cópia dos resultados da GPU para a CPU leva menos tempo do que a diferença de tempo de geração entre CPU e GPU.

4.1.3 Consumo de memória

Para a análise do consumo de memória, foram levados em consideração apenas os custos relativos às estruturas de dados, não levando em conta as variáveis dos métodos. O consumo é baseado nos modelos originais, embora as tabelas possam ser geradas com

Ruído	Tempo (em ms) por tamanho de textura						
	256	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
Valor	0.036	0.126	0.453	1.752	6.945	26.905	110.556
Gradiente	0.012	0.035	0.116	0.460	1.831	7.291	28.889
Convolução de grade	0.087	0.318	1.086	4.177	16.255	65.390	262.428
Convolução esparsa	0.445	1.786	6.320	24.139	96.621	399.279	1629.703
Perlin	0.011	0.029	0.096	0.381	1.519	6.051	24.012
OpenSimplex	0.027	0.084	0.288	1.106	4.379	16.929	66.879
Gabor	0.197	0.552	2.061	8.082	31.864	130.164	552.616

Tabela 4.3: Tempo de execução, por textura, dos métodos de ruído em GLSL.

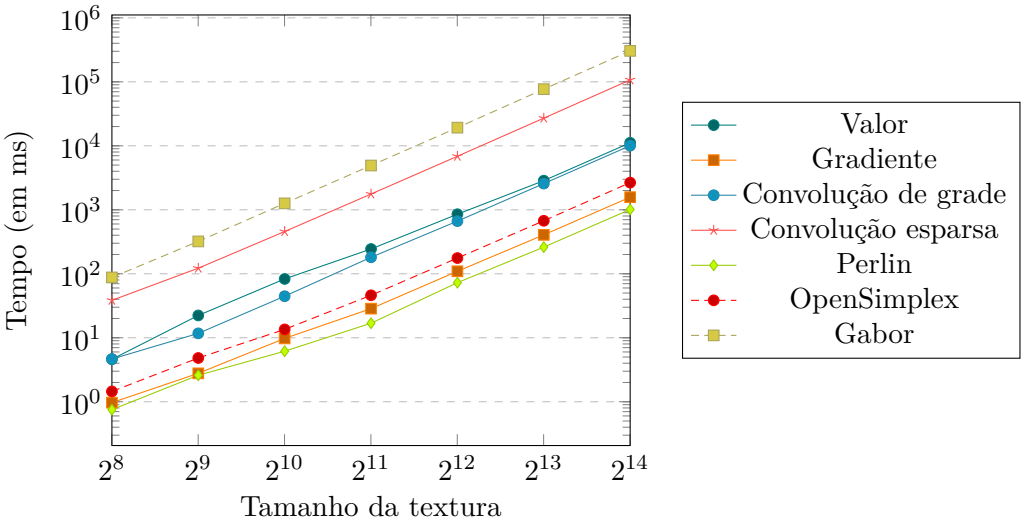


Figura 4.1: Tempo de execução, por tamanho de textura, dos métodos de ruído em C++.

tamanhos maiores, melhorando a qualidade final dos ruídos em troca de uma quantidade maior de memória utilizada. As tabelas poderiam ter uso de memória menor se fossem utilizados tipos com menor precisão, porém, os métodos estariam limitados aos modelos originais. Os valores mínimos são listados entre parênteses.

O **ruído de valor** utiliza uma tabela de índices com 256 `unsigned int`s (ou `unsigned char`) e uma tabela de valores com 256 `float`ss, totalizando 2048 (ou 1280) bytes.

Para a geração do **ruído de gradiente**, são utilizadas uma tabela de índices com 256 `unsigned int`s (ou `unsigned char`) e uma tabela de gradientes com 768 `float`s, totalizando 4096 (ou 3328) bytes.

O **ruído de convolução de grade** utiliza uma tabela de índices com 256 `unsigned int`s (ou `unsigned char`), uma tabela de valores com 256 `float`s e uma tabela de amostras com 401 `float`s, totalizando 3652 (ou 2884) bytes.

Para o **ruído de convolução esparsa**, são utilizadas uma tabela de índices com

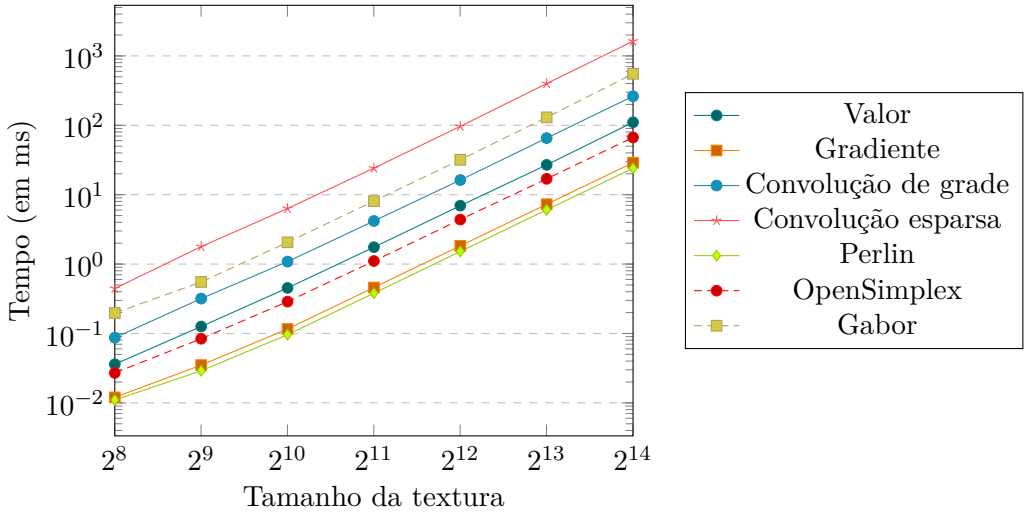


Figura 4.2: Tempo de execução, por tamanho de textura, dos métodos de ruído em GLSL.

Ruído	**Big-O**	**Tempo (em ns)**		**Memória (em bytes)**	
		C++	**GLSL**	**Mínimo**	**Usado**
Valor	$O(1)$	42.483	0.410	1280	2048
Gradiente	$O(1)$	5.953	0.108	3328	4096
Convolução de grade	$O(1)$	37.786	0.977	2884	3652
Convolução esparsa	$O(1)$	400.897	6.031	5956	6724
Perlin	$O(1)$	3.836	0.090	6682	8224
OpenSimplex	$O(1)$	9.990	0.251	328	1096
Gabor	$O(n)$	1138.362	2.027	28	28

Tabela 4.4: Complexidade algorítmica, tempo de execução, por pixel, e consumo de memória dos métodos de ruído.

256 `unsigned int`s (ou `unsigned char`), uma tabela de impulsos com 1024 `float`s e uma tabela de amostras com 401 `float`s, totalizando 6724 (ou 5956) bytes.

A técnica de **ruído de Perlin** emprega uma tabela de índices com 514 `unsigned int`s (ou `unsigned char`) e uma tabela de gradientes com 1542 `float`s, totalizando 8224 (ou 6682) bytes.

São necessárias duas tabelas para a geração do **ruído OpenSimplex**, uma de índices com 256 `unsigned int`s (ou `unsigned char`) e uma de gradientes com 72 `char`s, totalizando 1096 (ou 328) bytes.

Já o **ruído de Gabor** não utiliza qualquer tabela, sendo utilizadas apenas seis variáveis `float` e uma `unsigned int`, totalizando 28 bytes.

A análise é resumida na Tabela 4.4 e no gráfico da Figura 4.4. Para critério de comparação, uma textura quadrada, com 256 pixels de lado, gerada por meio de algum

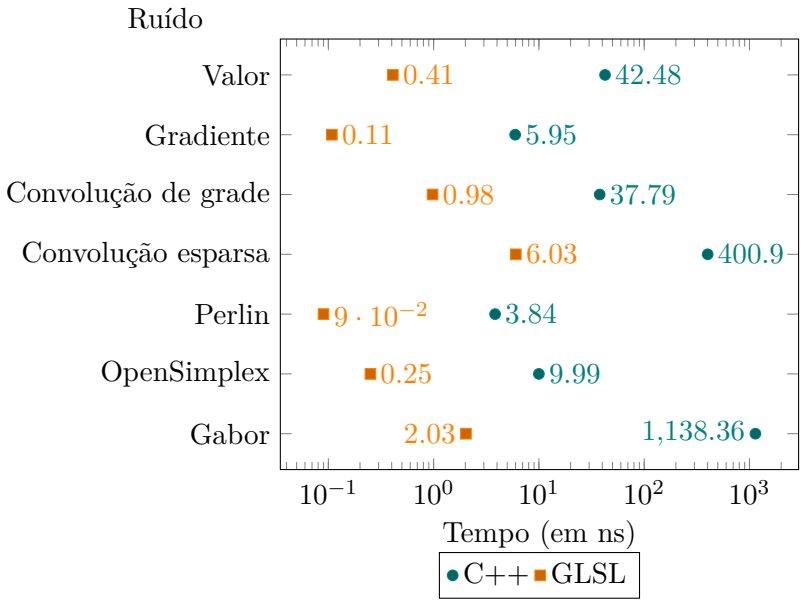


Figura 4.3: Tempo de execução, por pixel, dos métodos de ruído.

desses ruídos tem, no mínimo, 64 KiB[8]. Esse valor pode chegar a 256 MiB[9] no caso de texturas com 16384 pixels de lado. O uso dos métodos de geração de ruído podem eliminar esse custo elevado de memória, mas, em alguns casos, ao custo de maior tempo de execução.

4.1.4 Qualidade dos resultados

Os resultados obtidos pelos algoritmos são os ruídos apresentados na Figura 4.5.

Não faz sentido analisar as técnicas de geração de ruídos quanto ao realismo. Os ruídos são utilizados apenas como base para outras técnicas que geram o conteúdo final, sendo apenas um processo intermediário, com o objetivo de "organizar" números pseudo-aleatórios de alguma forma que facilite a geração desses conteúdos finais.

Um resultado que seja satisfatório para um tipo de aplicação pode não ser útil para outro tipo, e, portanto, a análise da qualidade deve ser feita com base em cada situação.

4.1.5 Tipo de algoritmo

Os ruídos são classificados como ***bits*** − **texturas** quanto ao tipo de conteúdo gerado.

Segundo a classificação proposta por Hendrikx et al. [62], ruídos são classificados como **geradores de números pseudo-aleatórios** quanto ao método de geração do conteúdo.

Os geradores de ruído podem ser gerados tanto *online* quanto *offline*.

[8] Kibibyte, ou seja, 2^{10} bytes[67].
[9] Mibibyte, ou seja, 2^{20} bytes[67].

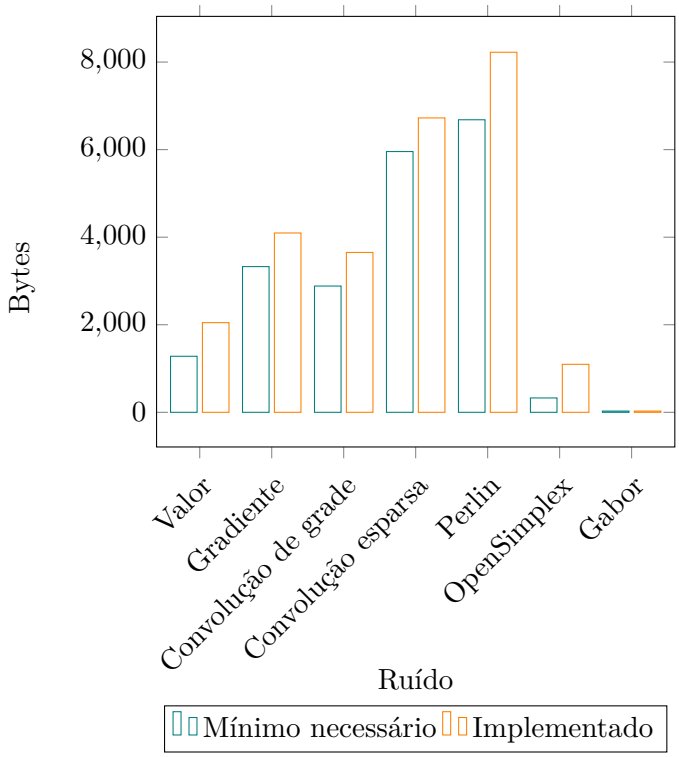


Figura 4.4: Consumo de memória dos métodos de ruído.

Os ruídos são usados para a geração de outros tipos de conteúdo, que podem ser tanto necessários quanto opcionais. Além disso, são determinísticos, o que permite que sejam gerados quando necessários, e não necessitam de um grande espaço de armazenamento.

Estas técnicas são construtivas, devido tanto ao fato de precisarem ser rápidas quanto à execução e também por que não existem objetivos não-matemáticos a serem atingidos pelos algoritmos.

Essa classificação é resumida na Tabela 4.5.

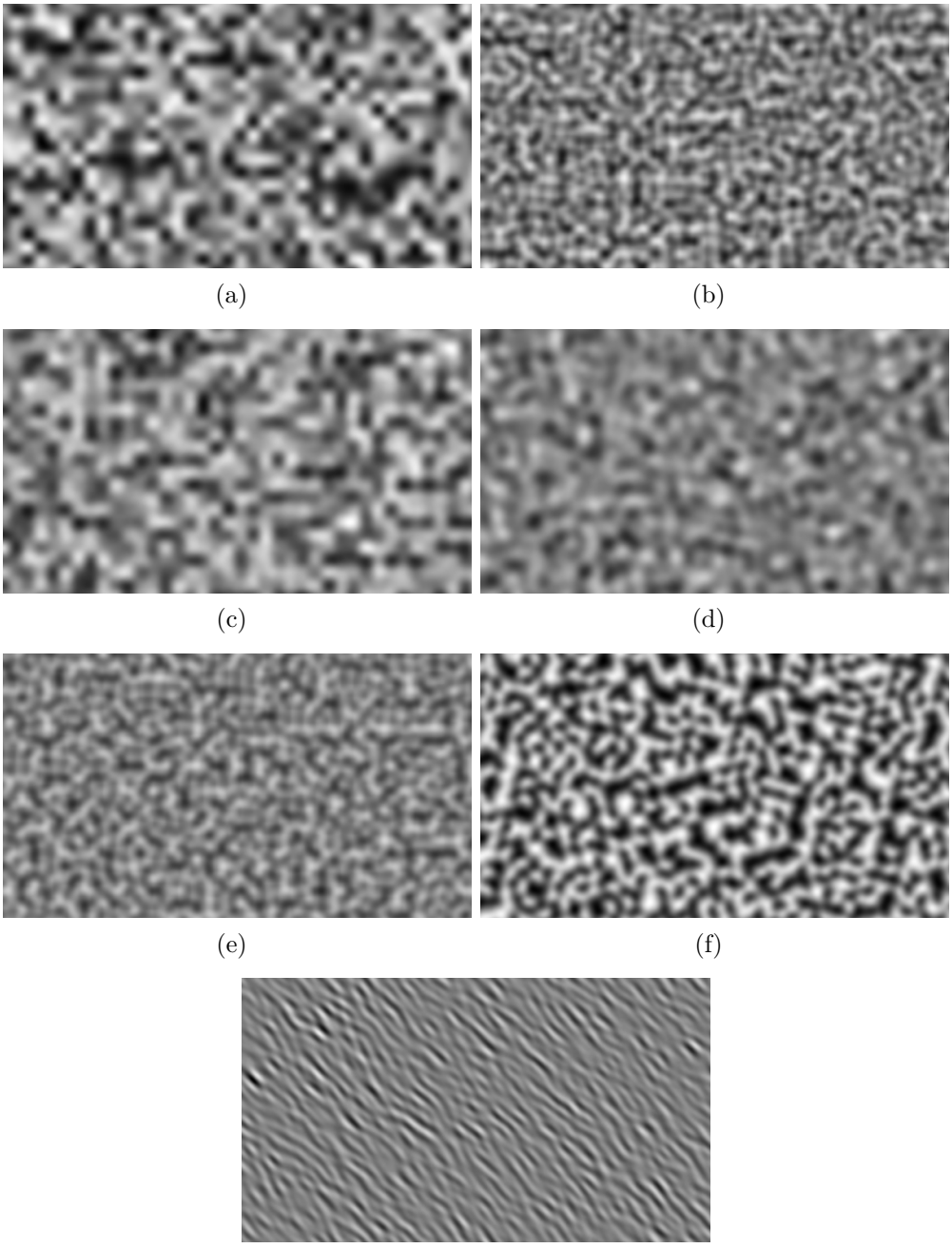
Método	**Tipo de conteúdo**	**Método de geração**	**Momento de geração**	**Corretude do conteúdo**	**Reprodução dos resultados**	**Condição de parada**
Valor	***bits* — texturas**	**geradores de números pseudo-aleatórios**	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Gradiente						
Convolução de grade						
Convolução esparsa						
Perlin						
OpenSimplex						
Gabor						

Tabela 4.5: Classificação dos métodos geradores de ruído.

4.2 Nomes

Os resultados das técnicas de geração de nomes são mostrados a seguir.

Figura 4.5: Ruídos: (a) valor; (b) gradiente; (c) convolução de grade; (d) convolução esparsa; (e) Perlin; (f) *OpenSimplex*; (g) Gabor.



(a) (b)

(c) (d)

(e) (f)

(g)

Fonte: Captura de tela pelo autor.

4.2.1 Big O

A geração de nomes por meio do uso de **máquinas de estados finitos**[10] tem apenas um laço `while` (linha 46) que executa de $[0; n]$, onde n é o tamanho máximo da palavra. Portanto, o método executa em tempo linear $O(n)$ e não tem um processo de inicialização.

O primeiro método baseado em cadeias de Markov[11], com tratamento especial para os caracteres iniciais e finais, lê um arquivo de palavras durante sua inicialização. Essa inicialização utiliza um laço `while` (linha 30) que executa para cada palavra p, e um laço `for` (linha 36), interno ao laço `while`, que executa de $[0; l-1]$, onde l é o tamanho da palavra. A inicialização ainda executa dois laços `for` (linhas 53 e 57) para cada um dos caracteres que iniciem ou ocorram em palavras, respectivamente. O tempo de execução depende tanto de p, quanto de l, tendo um limite superior de $O(n^2)$. Para a geração das palavras, o método utiliza um aninhamento de três `for`, sendo que o mais externo (linha 70) executa de $[1; m]$, onde m é um valor aleatório entre os tamanhos mínimo e máximo da palavra, o `for` intermediário (linha 72) executa de $[0; l]$, onde l é a quantidade de caracteres distintos que ocorreram no arquivo de entrada, e os três laços internos possíveis (linhas 76, 81 e 86), dos quais apenas um é executado por iteração do laço intermediário, e executam de $[0; p]$, onde p é a frequência dos caracteres lidos do arquivo de entrada. A geração dos nomes depende de m, l e p, mas como l e p são valores que não se alteram, a complexidade final é $O(n)$.

O segundo método baseado em Markov[12], com cadeias de ordem n, também utiliza uma inicialização com um laço `while` (linha 27), que executa uma vez para cada palavra p. Este laço, por sua vez, executa um segundo laço `for` (linha 39) que processa cada uma das palavras, utilizando, para isso, sub-cadeias das palavras. Esse laço interno executa de $[0; l-n]$, onde l é o comprimento da palavra e n, a ordem do processo. Dentro desse laço, são consultados um `std::map`, que tem complexidade $O(\log\ n)$. A inicialização deste método depende de p, l e das operações sobre o mapa, totalizando uma complexidade de ordem $O(n^2 \log\ n)$. Já a geração dos nomes utiliza um laço `do...while` (linha 58), que executa enquanto o tamanho do nome gerado for menor que o mínimo ou o nome gerado for um dos que foram lidos ou já tenha sido gerado. Essa garantia tem um custo $O(n)$, visto que é utilizado um `std::unordered_set` para armazenar e consultar os nomes gerados (e lidos) anteriormente. No entanto, é importante notar que no melhor caso, esse custo é $O(1)$. Dentro desse laço, é executada uma busca em um `std::map`, com custo $O(\log\ n)$, e executado um laço `while` (linha 67) que executa de $[0; l]$, onde l é o tamanho mínimo do nome a ser gerado. A geração é dependente de l e do custo do mapa. No entanto, este método, por garantir que sempre seja gerado um nome que ainda não tenha sido produzido, pode nunca finalizar, tendo "complexidade" máxima

[10] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/names/finite_state.cpp

[11] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/names/markov.cpp

[12] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/names/markov_n.cpp

de $O(\infty)$. Ainda assim, não ocorrendo geração repetida, considerando que o tamanho l da palavra e o tamanho do mapa não se alteram, o custo de geração passa a depender apenas dos tamanhos das palavras geradas, sendo, portanto, $O(n)$.

A complexidade algorítmica dos métodos está resumida nas Tabelas 4.6 e 4.7.

4.2.2 Tempo de execução

Os tempos de execução foram calculados para a inicialização dos geradores baseados em cadeias de Markov por meio de dois arquivos de textos contendo 10196 e 21986 nomes, respectivamente (Tabela 4.6). Note que o método baseado em máquina de estados finitos não tem custo de inicialização significativo. Também foram calculados, para os três métodos de geração de nomes, o tempo necessário para a geração de 100000 nomes (Tabela 4.7).

Método	**Big-O**	**Tempo (em ms) por tamanho do arquivo de entrada (em palavras).**	
		10196	**21986**
Estatísticas separadas para caracteres iniciais e finais	$O(n^2)$	2.084	3.280
Ordem n	$O(n^2 \log n)$	10.327	15.527

Tabela 4.6: Complexidade algorítmica e tempo de execução, por arquivo de entrada, dos métodos baseados em cadeias de Markov.

Método	**Big-O**	**Tempo (em ns) por nome.**	**Consumo de memória**
Máquina de estados finitos	$O(n)$	307.932	≈2.5KiB
Cadeias de Markov com estatísticas separadas para caracteres iniciais e finais	$O(n)$	10924.285	$O(n)$
Cadeias de Markov de ordem n	$O(n)$	1764.585	$O(n^2)$

Tabela 4.7: Complexidade algorítmica, tempo de geração, por nome, e consumo de memória dos métodos de geração de nomes.

Apesar de o método baseado em cadeias de Markov de ordem n apresentar tempo de execução menor que o método com estatísticas separadas, quando a quantidade de nomes aumenta muito, esse tempo de geração também aumenta infinitamente. O método baseado em estatísticas separadas não apresenta esse problema, mas pode gerar nomes repetidos. Os problemas oriundos dessa variação dependerão de cada aplicação.

4.2.3 Consumo de memória

O método de geração de nomes baseado em máquina de estados finitos utiliza como memória apenas um objeto `std::mt19937`, ou seja, 19937 bits (2.434KiB). Esse valor pode ser considerado como base para a análise de memória dos algoritmos de geração de nomes, visto que as três técnicas utilizam este objeto.

Para o gerador de nomes baseado em cadeias de Markov com estatísticas separadas para caracteres iniciais e finais, é utilizada uma classe `Word_frequency` com três `int`s e um `bool`, totalizando 13 bytes. Essa classe contém as frequências das letras e os objetos da classe são armazenados em uma matriz. Essa matriz tem tamanho 256x256, totalizando 832 KiB. São armazenados também dois vetores de caracteres de tamanho n, com n variando de acordo com a quantidade de caracteres únicos no arquivo de entrada, que não ultrapassa 255 elementos. A complexidade de memória do algoritmo é $O(n)$.

Já o gerador baseado em cadeias de Markov de ordem n utiliza um vetor e um `std::unordered_set` de cadeias de caracteres e um mapa de cadeias para vetores de caracteres. Esse mapa, portanto, tem custo de memória de $O(n^2)$, com n limitado pelo número de palavras, seus tamanhos e, também, de acordo com a ordem escolhida para o processo de Markov. O vetor de amostras é limitado pela quantidade de palavras do arquivo de entrada, tendo ordem $O(n)$. O `std::unordered_set` de palavras geradas aumenta de acordo com a quantidade de execuções do método de geração de palavras, também com complexidade $O(n)$. Embora o mapa tenha custo $O(n^2)$, após um número de execuções elevado, o custo $O(n)$ do `std::unordered_set` ultrapassa o custo do mapa.

O consumo de memória dos métodos está resumido nas Tabelas 4.6 e 4.7.

4.2.4 Qualidade dos resultados

Na Tabela 4.8 são apresentados os primeiros 20 resultados da execução de cada um dos métodos. Para os métodos baseados em cadeias de Markov, foi utilizado como entrada um arquivo com 21986 nomes de países anglófonos[13].

Embora os resultados a serem obtidos variem de acordo com a necessidade de cada jogo e linguagem[14], a qualidade dos resultados pode ser analisada de acordo com a facilidade de pronúncia dos nomes. Com essa métrica, apenas a técnica de cadeias de Markov de ordem n gera, consistentemente, nomes que não tenham agrupamentos de consoantes que prejudiquem suas pronúncias. Ainda sim, existem linguagens, tanto reais quanto fictícias, que tem palavras com muitas consoantes, e, portanto, essa métrica deve ser redefinida de acordo com a linguagem a ser gerada.

[13] Países que tem a língua inglesa como linguagem oficial.

[14] Aqui, *linguagem* se refere ao conjunto de palavras utilizadas por um grupo de personagens no jogo.

Máquina de estados finitos	Cadeias de Markov com estatísticas especiais	Cadeias de Markov de ordem n
God	Garazcil	Wilhell
Hace	Roemae	Bruith
Kag	Xinge	Deline
Uvemi	Jah	Coutl
Dofoj	Zagnnea	Alom
Wiq	Urrin	Lope
Ibavga	Ivl	Bento
Hyzcij	Diessterr	Ludline
Ivgev	Rosnger	Bartm
'ke	Fer	Ston
Jeq	Orcnwm	Wallin
Sto	Meawvyng	Fitt
Wer	Loan	Roly
Nb'spi	Moada	Honster
Ozgamte	Hatnti	Venport
Hde	Genlei	Sothieu
Yqapiw	Panrnanal	Lodove
Zgas	Jerltlola	Hamsd
Daspac	Netnl	Frode
Qezuhc	Gelstlct	Pherbs

Tabela 4.8: Resultados dos geradores de nomes.

4.2.5 Tipo de algoritmo

Nomes podem ser classificados como ***bits*** em jogos que os utilizem para nomear lugares ou personagens, por exemplo, ou como **espaço** em jogos que os utilizem para determinar lugares de forma abstrata.

De acordo com os métodos de geração empregados, as três técnicas podem ser classificadas como **geradores de números pseudo-aleatórios**. Essas técnicas são rápidas o suficiente para serem geradas *online*, mas também podem ser utilizadas *offline*.

Nomes podem ser classificados tanto como conteúdo necessário quanto opcional, dependendo do jogo e do uso dos resultados. Ainda assim, considerando que os nomes servem apenas para a identificação de algum elemento do jogo, é pouco provável que sejam gerados resultados que impeçam o progresso do jogador.

Os métodos são estocásticos, dado que utilizam geradores de números pseudo-aleatórios. Os métodos baseados em máquinas de estados finitos e cadeias de Markov com estatísticas especiais para letras iniciais e finais são construtivos. O método baseado em cadeias de Markov de ordem n testa se o nome já existe, e, caso afirmativo, reinicia

o processo de geração. Logo, esse método é do tipo gerar-e-testar.

Essa classificação é resumida na Tabela 4.9.

Método	Tipo de conteúdo	Método de geração	Momento de geração	Corretude do conteúdo	Reprodução dos resultados	Condição de parada
Máquina de estados finitos	***bits*** ou **espaço**	**geradores de números pseudo-aleatórios**	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Cadeias de Markov com estatísticas separadas para caracteres iniciais e finais						
Cadeias de Markov de ordem n						gerar-e-testar

Tabela 4.9: Classificação dos métodos geradores de nomes.

4.3 Vegetação

4.3.1 Big O

A técnica de geração baseada em subdivisão aleatória dos ramos[15] utiliza uma função recursiva `split` para gerar os ramos. Esta função tem custo linear e pode executar até duas recursões (linhas 177, 179 e 181) para cada ramo, e, portanto, tem complexidade máxima $O(2^n)$ sobre o número de níveis a serem gerados.

O gerador baseado em sistemas de funções iterativas[16] utiliza um `for` (linha 76) que realiza uma busca linear em um `vector` (linha 79). Com isso, a complexidade máxima do método é $O(n^2)$.

Já a técnica baseada em L-*systems*[17] utiliza uma função recursiva `reproduce` sobre o número de iterações (linha 84). Essa função utiliza uma função auxiliar `produce` (linha 88) que tem complexidade quadrática. Portanto, a complexidade máxima do gerador é $O(n^3)$.

A complexidade algorítmica das técnicas está resumida na Tabela 4.10.

4.3.2 Tempo de execução

Os tempos de execução foram medidos com a execução de cada técnica utilizando, como entrada, dados que os desenvolvedores das implementações originais propuseram. Os valores medidos são listados na Tabela 4.10.

Esses valores mostram que as técnicas têm tempo de execução baixo o suficiente para serem usadas *online*, principalmente devido ao fato de que os dados gerados têm que ser enviados para a GPU para serem renderizados, e, normalmente, isso é feito durante a etapa de inicialização da cena a ser renderizada, onde geralmente já se tem um tempo significativo de carregamento.

[15] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/vegetation/proctree.cpp

[16] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/vegetation/ifs.cpp

[17] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/vegetation/l_system.cpp

Método	Big-O	Tempo de geração (em ms).	Consumo de memória
Subdivisão aleatória de ramos	$O(2^n)$	0.583	$O(n^2)$
Sistemas de funções iterativas	$O(n^2)$	2.116	$O(n^2)$
L-*systems*	$O(n^3)$	4.342	$O(n^3)$

Tabela 4.10: Complexidade algorítmica, tempo de geração e consumo de memória dos métodos de geração de vegetação.

4.3.3 Consumo de memória

O gerador baseado em subdivisão aleatória dos ramos utiliza uma árvore binária com referência ao nó pai, onde cada nó utiliza cinco `float`s, um `int` e um `bool`, além de quatro `vector`s de `int`s. Esses quatro `vector`s armazenam os índices nos vetores de dados de vértices gerados. Esses vetores de dados de vértices são oito `vector`s, sendo seis de `float`s e dois de `int`s, compartilhados a todos os nós, e são usados para armazenar o modelo geométrico gerado. A técnica tem complexidade de memória de ordem $O(n^2)$ sobre o nível de iterações gerado.

A técnica baseada em sistemas de funções iterativas utiliza um `vector` de parâmetros em que cada elemento utiliza sete `double`s. Esse vetor é fornecido como parâmetro de entrada pelo usuário, e, geralmente, tem poucos elementos. O gerador produz uma imagem em escala de cinza contendo o resultado, o que ocupa espaço na memória na ordem de $O(n^2)$.

Já a técnica baseada em L-*systems* utiliza um `vector` de vértices, que contém nove `float`s e um `int` cada, e um `vector` de faces, que contém quatro `float`s, um `int`, um `bool` e um `vector` de ponteiros. No entanto, a quantidade de faces e vértices gerados depende das regras de entrada, sendo que, no caso teste utilizado, a complexidade de memória é $O(n^3)$ sobre o número de iterações realizadas.

O consumo de memória das técnicas está resumido na Tabela 4.10.

4.3.4 Qualidade dos resultados

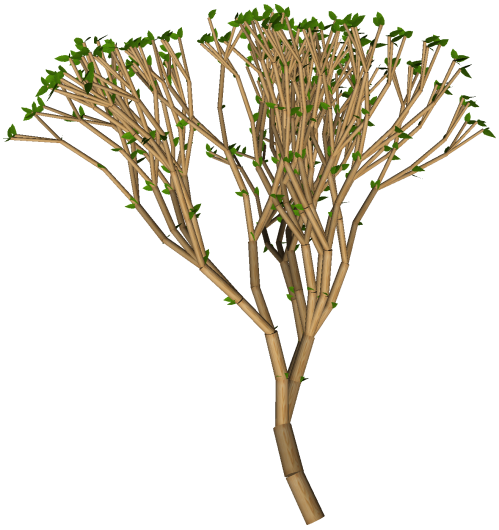
A Figura 4.6 mostra exemplos de resultados obtidos pelas técnicas. Com base nos resultados, pode-se observar que a qualidade da técnica baseada em subdivisão aleatória dos ramos é média. Embora o resultado não seja tão próximo de uma árvore real, ainda sim apresenta aparência próxima do real.

Quanto ao conteúdo gerado pelos geradores baseados em sistemas de funções iterativas e para o baseado em L-*systems*, a qualidade é baixa. No entanto, a qualidade das duas técnicas depende dos parâmetros de geração passados aos geradores.

Figura 4.6: Vegetação gerada pelas técnicas: (a) baseada em subdivisão aleatória dos ramos; (b) baseada em sistemas de funções iterativas; (c) baseada em L-*systems*.

(a)

(b)



(c)

Fonte: (a, c) Captura de tela pelo autor; (b) Gerado pela implementação.

4.3.5 Tipo de algoritmo

Quanto ao tipo de conteúdo gerado, as três técnicas são classificadas como ***bits*** – **vegetação**.

O gerador baseado em subdivisão aleatória dos ramos utiliza **geradores de números pseudo-aleatórios** na construção dos resultados. O gerador baseado em sistemas de funções iterativas é classificado, quanto ao método de geração, como **fractal**, e a técnica baseada em L-*systems* como **gramáticas geradoras – sistemas de *Lindenmayer***.

A técnica baseada em subdivisão aleatória dos ramos pode ser utilizada *online*. As outras duas técnicas têm sua utilização *online* limitada a entradas simples, pois podem consumir muito tempo para produzir resultados para regras mais complexas.

Apesar de a qualidade do conteúdo gerado não ser alta, este pode ser utilizado como conteúdo necessário, visto que os resultados são corretos. No entanto, a garantia de estarem corretos depende da seleção de parâmetros de entrada bem definidos, sendo que, em casos extremos, o conteúdo pode ser gerado de forma incorreta. Isso exige que o desenvolvedor limite os parâmetros de entrada à faixas aceitáveis.

As três técnicas são estocásticas, sendo que utilizam geradores de números pseudo-aleatórios para a geração do conteúdo. Além disso, todas as técnicas são construtivas, não se preocupando em testar os resultados quanto a nenhuma métrica.

Essa classificação é resumida na Tabela 4.11.

Método	Tipo de conteúdo	Método de geração	Momento de geração	Corretude do conteúdo	Reprodução dos resultados	Condição de parada
Subdivisão aleatória de ramos	***bits*** – **vegetação**	**geradores de números pseudo-aleatórios**	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Sistemas de funções iterativas		**fractal**				
L-*systems*		**gramáticas geradoras – sistemas de *Lindenmayer***				

Tabela 4.11: Classificação dos métodos geradores de vegetação.

4.4 Simulação de elementos naturais

4.4.1 Big O

Com exceção da técnica baseada em partículas[18], as outras três técnicas têm complexidade de inicialização $O(1)$. Já a técnica baseada em partículas, tem complexidade de inicialização $O(n)$ sobre a quantidade de partículas (linha 123).

[18] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/elements/particles.cpp

Quanto à complexidade de execução, todos os algoritmos têm complexidade $O(n)$ sobre a quantia de pixels gerados. Vale notar que a técnica baseada em partículas utiliza um laço `for` para atualizar dados sobre cada partícula, com complexidade $O(n)$ sobre a quantidade de partículas (linha 135).

4.4.2 Tempo de execução

Para os simuladores de elementos naturais foram realizadas medições em três pontos críticos da execução: inicialização (Tabela 4.12), renderização de um quadro de 3840x2160 pixels (Tabela 4.13), e limpeza dos recursos (Tabela 4.14). Esses valores se mostram importantes nestas técnicas pois, ao menos no caso dos simuladores de fogo, os locais em chamas nos jogos podem se alterar, com elementos se ignizando ou extinguindo em uma mesma cena.

Método	**Tempo de inicialização (em ms).**
Fogo baseado em ruído de fluxo	0.858
Fumaça baseada em partículas	1.096
Oceanos	1.662
Nuvens	2.306

Tabela 4.12: Tempo de inicialização dos métodos de simulação de elementos naturais.

Método	**Tempo de renderização (em ms).**
Fogo baseado em ruído de fluxo	0.671
Fumaça baseada em partículas	0.595[a]
Oceanos	18.557
Nuvens	3.765

[a] Para 2800 partículas.

Tabela 4.13: Tempo de renderização dos métodos de simulação de elementos naturais na GPU para um quadro de tamanho 3840x2160 pixels.

Com esses resultados, pode-se concluir que todas as técnicas podem ser executadas *online*. Nota-se, no entanto, que a técnica de geração de oceanos pode rapidamente esgotar o tempo disponível para a renderização do quadro e causar lentidão no jogo se não usada com cuidado, sendo que não é possível utilizá-la para renderizar em UHD a 60 quadros por segundo.

Método	Tempo de limpeza (em μs).
Fogo baseado em ruído de fluxo	3.640
Fumaça baseada em partículas	13.646
Oceanos	4.859
Nuvens	3.309

Tabela 4.14: Tempo de limpeza de recursos dos métodos de simulação de elementos naturais.

4.4.3 Consumo de memória

Os geradores de fogo baseado em ruído de fluxo, oceanos e nuvens utilizam apenas alguns poucos `float`s e/ou `int`s cada, sem uso de memória permanente para os cálculos. Com isso, eles apresentam complexidade $O(1)$ quanto à memória utilizada.

Já o gerador de fumaça baseado em partículas utiliza um `vector` de 2800 partículas, com cada uma utilizando nove `float`s. Estes vetores totalizam 100800 bytes. Ainda é utilizada uma textura de 128x128 pixels, com 16384 bytes. Estes dados totalizam 114.44 KiB. Com isso, a complexidade quanto à memória é $O(1)$.

4.4.4 Qualidade dos resultados

Os resultados obtidos pelas técnicas são exibidos na Figura 4.7.

Os geradores de fogo apresentam bons resultados, com a versão baseada em ruído de fluxo apresentando chamas que se agrupam, formando um aglomerado de labaredas. Já a versão baseada em partículas apresenta um aspecto fragmentado, mas oferece mais controle sobre o resultado.

Tanto o gerador de oceanos quanto o de nuvens apresentam resultados excelentes, gerando resultados com alto nível de realismo.

4.4.5 Tipo de algoritmo

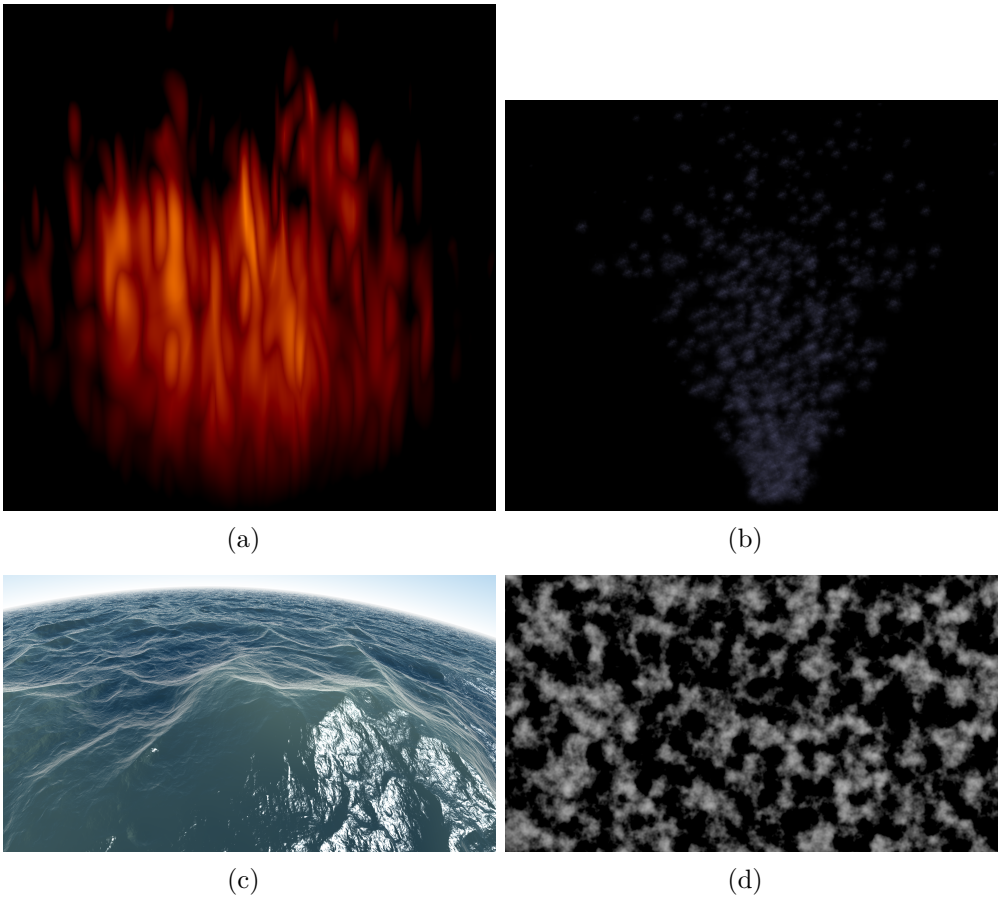
As quatro técnicas são classificadas quanto ao tipo de conteúdo como ***bits* – fogo, água, pedras e nuvens**.

Já quanto ao método de geração, as quatro técnicas utilizam ruídos como base para seu funcionamento, e, portanto, são classificadas como **geradores de números pseudo-aleatórios**.

Todas as técnicas podem ser utilizadas *online*, embora a técnica de geração oceanos seja bastante lenta para ser utilizada em cenas complexas. É importante notar que estas técnicas raramente são utilizadas *offline*, visto que este tipo de conteúdo é aleatório, e a reutilização de conteúdo pré-renderizado se torna repetitivo para o jogador.

As quatro técnicas podem ser utilizadas como conteúdo necessário. Todas elas geram apenas renderizações do conteúdo, e, portanto, não apresentam problemas à lógica

Figura 4.7: Elementos gerados pelas técnicas: (a) fogo, baseado em ruído de fluxo; (b) fogo, baseado em partículas; (c) oceanos; (d) nuvens.



(a) (b)

(c) (d)

do jogo. São, ainda, estocásticas, sendo baseadas em geradores de números pseudo-aleatórios.

Dado que as técnicas geram conteúdo renderizável em tempo real, é necessário que elas sejam construtivas, visto que não há tempo para se testar o conteúdo.

Essa classificação é resumida na Tabela 4.15.

4.5 Cavernas, labirintos e masmorras

Os resultados das técnicas de geração de mapas internos são descritos a seguir.

Método	Tipo de conteúdo	Método de geração	Momento de geração	Corretude do conteúdo	Reprodução dos resultados	Condição de parada
Fogo baseado em ruído de fluxo	***bits*** **– fogo, água, pedras e nuvens**	**geradores de números pseudo-aleatórios**	*online*	necessário ou opcional	estocástico	construtivo
Fumaça baseada em partículas						
Oceanos						
Nuvens						

Tabela 4.15: Classificação dos métodos simuladores de elementos naturais.

4.5.1 Big O

A técnica de geração de cavernas baseada em autômatos celulares[19] utiliza dois laços `for` (linhas 43 e 44) que variam de acordo com as dimensões da grade a ser gerada. A técnica também depende da configuração de repetições das iterações do algoritmo de autômatos celulares, apresentando complexidade máxima $O(n^3)$. Já a técnica para geração de cavernas baseada em agregação por difusão limitada[20] utiliza apenas um laço `while` (linha 39), que varia de acordo com a quantidade de células na grade, dada pelas dimensões desta grade. A complexidade desta técnica é $O(n^2)$.

Para a técnica de geração de labirintos[21] são utilizados dois laços `while` (linhas 30 e 34). Esses laços variam de acordo com a quantidade de células na grade, portanto, a técnica tem complexidade $O(n^2)$.

A primeira técnica para geração de masmorras[22] utiliza um laço `while` (linha 40) que varia de acordo com a quantidade de salas a serem geradas, além de uma constante que limita a quantidade de tentativas de colocação de salas na grade. Logo, essa técnica tem complexidade $O(n)$. A segunda técnica[23] utiliza um laço `for` (linha 61) que varia de acordo com o número de elementos escolhido pelo usuário, apresentando complexidade $O(n)$.

A complexidade algorítmica das técnicas é resumida na Tabela 4.16.

4.5.2 Tempo de execução

Os tempos de execução foram medidos com a geração de mapas quadrados de lado s tal que $s : 2^n, n \in \mathbb{N}, n \in [8; 12]$. Os valores medidos são listados na Tabela 4.16 e exibidos na Figura 4.8. Apenas a técnica baseada em autômatos celulares foi paralelizada devido ao fato de as outras técnicas não serem trivialmente paralelizáveis, sem perda do determinismo das técnicas.

[19] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/indoor_maps/ca_cave.cpp

[20] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/indoor_maps/dla_cave.cpp

[21] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/indoor_maps/maze.cpp

[22] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/indoor_maps/dungeon_a.cpp

[23] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/indoor_maps/dungeon_b.cpp

Técnica	Big-O	Tempo (em ms) por tamanho de mapa				
		256	512	1024	2048	4096
Cavernas com autômatos celulares – não-paralelo	$O(n^3)$	8.272	32.608	131.654	530.735	2232.396
Cavernas com autômatos celulares – paralelo	$O(n^3)$	5.265	12.900	49.105	181.898	667.646
Cavernas com agregação por difusão limitada	$O(n^2)$	2.215	9.571	39.423	372.392	1668.109
Labirintos	$O(n^2)$	2.504	9.409	38.466	184.167	940.077
Masmorras – primeira técnica	$O(n)$	5.801	5.211	5.266	7.118	16.425
Masmorras – segunda técnica	$O(n)$	1.902	1.867	2.070	2.699	6.832

Tabela 4.16: Complexidade algorítmica e tempo de execução, por mapa, dos métodos de mapas internos em C++.

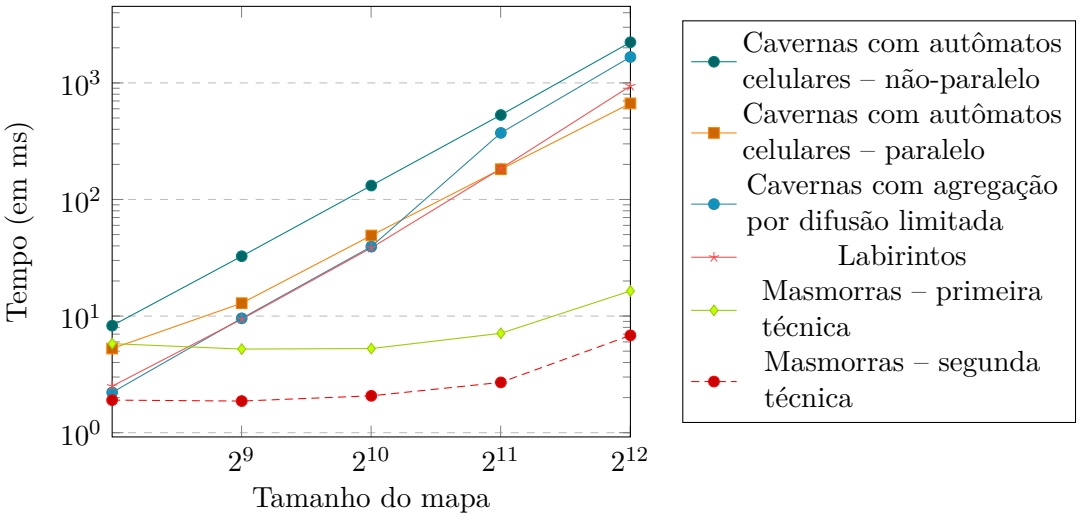


Figura 4.8: Tempo de execução, por tamanho de mapa, dos métodos de mapas internos em C++.

A partir desses dados, nota-se que todas as técnicas executam em tempo baixo o suficiente para o uso *online*.

4.5.3 Consumo de memória

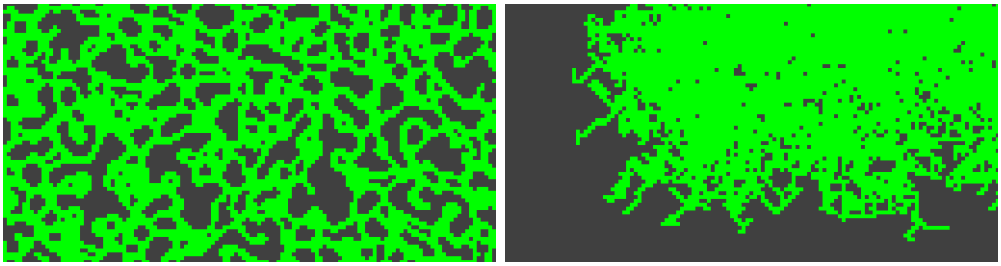
As cinco técnicas de geração de mapas internos seguem a mesma complexidade de memória quadrática ($O(n^2)$) devido ao uso de memória da grade gerada. Algumas técnicas utilizam mais espaço, e esses gastos serão destacados a seguir.

A técnica de geração de cavernas com uso de autômatos celulares utiliza uma grade auxiliar para as iterações do algoritmo, e, portanto, consome aproximadamente duas vezes mais memória que as outras técnicas. Já as técnicas de geração de masmorras utilizam dois vetores com informações sobre as salas geradas. O consumo de memória desses vetores é, no entanto, linear, e muito menor que o consumo da grade, não alterando a complexidade dos algoritmos.

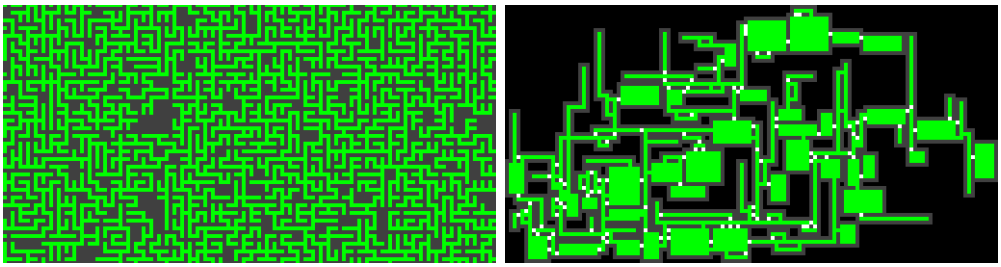
4.5.4 Qualidade dos resultados

Os resultados obtidos pelos algoritmos são exibidos na Figura 4.9. As imagens foram recortadas de forma a ficarem mais visíveis. Os cortes foram realizados de forma a destacar partes "interessantes" das imagens. Nas imagens, as áreas representadas em verde são passáveis, as brancas são portais, as cinzas são paredes geradas, e as pretas são áreas onde nenhum tipo de terreno foi gerado. Foi escolhida essa representação pelo fato de ser mais compacta que a representação em caracteres ASCII utilizada pelos algoritmos originais.

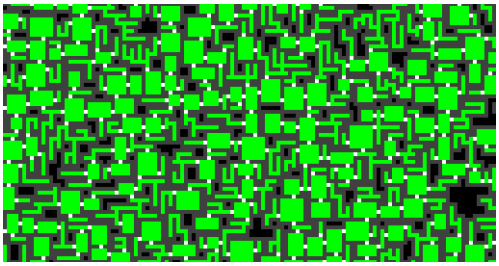
Figura 4.9: Partes de mapas internos gerados pelas técnicas: (a) cavernas com autômatos celulares; (b) cavernas com agregação por difusão limitada; (c) labirintos; (d) masmorras – primeira técnica; (e) masmorras – segunda técnica. As áreas verdes são passáveis, as brancas são portais e as outras são impassáveis.



(a) (b)



(c) (d)



(e)

Quanto à qualidade dos resultados de cavernas, em mapas pequenos, as duas técnicas apresentam resultados aceitáveis. Porém, para mapas com dimensões maiores, a técnica

baseada em autômatos celulares é claramente superior, visto que a outra técnica gera áreas muito grandes, que não parecem muito realistas.

A técnica de geração de labirintos gera resultados visivelmente complexos, com muitos *deadends*[24], que tornam o labirinto bastante complexo de ser resolvido para um jogador que tenha visão interna ao labirinto, mas, ao mesmo tempo, facilita a obtenção da solução por um jogador que tenha uma visão geral do labirinto.

Por fim, quanto às técnicas de geração de masmorras, nota-se que a primeira delas tende a gerar longos corredores e salas largas, com grandes áreas impassáveis entre elas, enquanto a segunda gera salas e corredores menores, mas utiliza melhor a área dos mapas. A primeira técnica ainda tende a gerar portas em locais que não fazem sentido, como ao lado de uma passagem aberta.

4.5.5 Tipo de algoritmo

Segundo o tipo de conteúdo, as cinco técnicas são classificadas como **espaço – mapas internos**.

Quanto à geração, o gerador de cavernas com autômatos celulares, o de cavernas com agregação por difusão limitada e o de labirintos são classificados como **modelagem e simulação de sistemas complexos**. Já os geradores de masmorras utilizam técnicas de **geradores de números pseudo-aleatórios**.

Essas técnicas podem ser utilizadas tanto de forma *online* quanto *offline*.

Das cinco técnicas, a de geração de cavernas com autômatos celulares pode gerar áreas não atingíveis, e a de geração de cavernas com agregação por difusão limitada pode não gerar a caverna, dependendo dos parâmetros de entrada, e, portanto, não são apropriadas ao uso como conteúdo necessário, pois podem impedir o progresso do jogador. As outras técnicas podem gerar os dois tipos de conteúdo.

As técnicas são estocásticas, utilizando geradores de números pseudo-aleatórios para a geração dos resultados, além de serem construtivas.

Essa classificação é resumida na Tabela 4.17.

<table>
<tr><th>Método</th><th>Tipo de conteúdo</th><th>Método de geração</th><th>Momento de geração</th><th>Corretude do conteúdo</th><th>Reprodução dos resultados</th><th>Condição de parada</th></tr>
<tr><td>Cavernas com autômatos celulares</td><td rowspan="5">espaços – mapas internos</td><td rowspan="2">simulação de sistemas complexos</td><td rowspan="5">online ou offline</td><td>opcional</td><td rowspan="5">estocástico</td><td rowspan="5">construtivo</td></tr>
<tr><td>Cavernas com agregação por difusão limitada</td></tr>
<tr><td>Labirintos</td><td rowspan="3">geradores de números pseudo-aleatórios</td><td rowspan="3">necessário ou opcional</td></tr>
<tr><td>Masmorras – primeira técnica</td></tr>
<tr><td>Masmorras – segunda técnica</td></tr>
</table>

Tabela 4.17: Classificação dos métodos geradores de mapas internos.

[24]Corredores que não levam a nenhum outro, terminando abruptamente.

4.6 Mapas de elevação

Os resultados das técnicas de geração de mapas de elevação são descritos a seguir.

4.6.1 Big O

O método baseado no algoritmo diamante-quadrado[25] utiliza um laço `while` (linha 37) que executa os passos quadrado e diamante com o uso de dois `for` (linhas 38 e 39, e, 46 e 48) aninhados para cada passo, que executam com passos p, tal que $p_{i+1} = p_i/2$ e $p > 0$, com $p_0 = tamanho/2$. Essa técnica tem custo $O(n^2)$.

Já o método baseado em bisseção de hemisférios[26] utiliza um laço `for` (linha 63) e outro laço `for` (linha 192), interno ao primeiro, logo, o método tem complexidade $O(n^2)$.

4.6.2 Tempo de execução

Os tempos de execução foram medidos com a geração de mapas quadrados de lado s tal que $s : 2^n, n \in \mathbb{N}, n \in [8; 12]$. Os valores medidos são listados na Tabela 4.18 e exibidos na Figura 4.10.

Técnica	Tempo (em ms) por tamanho de mapa				
	256	512	1024	2048	4096
Diamante-quadrado	0.397	1.585	6.230	24.425	103.576
Bisseção de hemisférios	9.388	20.888	52.389	165.485	542.705

Tabela 4.18: Tempo de execução, por mapa, dos métodos de mapas externos em C++.

Com base nesses dados, nota-se que as duas técnicas podem ser utilizadas *online*, sendo que a técnica que utiliza o algoritmo diamante-quadrado pode ser utilizada *online* para mapas cuja área renderizada não ultrapasse um quadrado de 256 pontos de lado.

4.6.3 Consumo de memória

O custo de memória de ambas as técnicas é o tamanho da grade de valores utilizada para representar o resultado final do algoritmo. O método baseado em bisseção de hemisférios utiliza dois vetores auxiliares, mas o custo destes é pequeno com relação ao da grade. Portanto, os dois métodos têm complexidade de memória $O(n^2)$.

4.6.4 Qualidade dos resultados

Os resultados obtidos pelos algoritmos são exibidos na Figura 4.11.

As duas técnicas geram bons resultados de mapas de elevação, sendo que a técnica de bisseção de hemisférios tende a gerar mapas mais próximos dos encontrados na

[25] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/height_maps/diamond_square.cpp

[26] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/height_maps/bissection.cpp

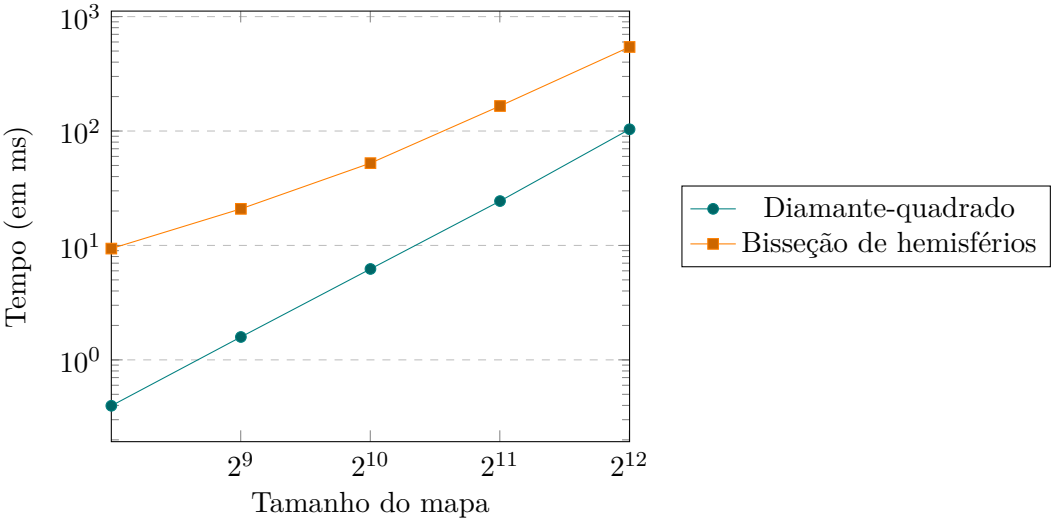
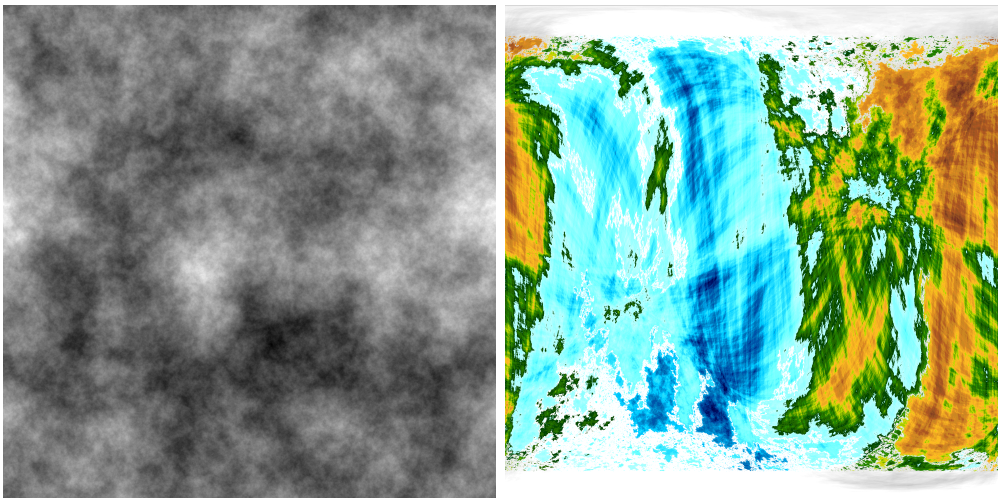


Figura 4.10: Tempo de execução, por tamanho de mapa, dos métodos de mapas externos em C++.

Figura 4.11: Partes de mapas externos gerados pelas técnicas: (a) diamante-quadrado; (b) bisseção de hemisférios.



(a) (b)

natureza, com longas cadeias de montanhas. Esta técnica é mais apropriada para a geração de planetas inteiros. Já a técnica baseada no algoritmo diamante-quadrado gera resultados que se parecem mais com pequenas áreas de mapas, onde a existência de, por exemplo, cadeias de montanhas, possa atrapalhar a experiência do jogador.

4.6.5 Tipo de algoritmo

Segundo o tipo de conteúdo, as duas técnicas são classificadas em **espaço – mapas externos**. Quanto ao método de geração, as técnicas são classificadas como **fractais**.

Essas técnicas podem ser utilizadas tanto *online* quanto *offline*. Isso se deve ao fato de que os resultados são gerados de forma rápida e não necessitam de tratamento após a geração.

A possibilidade de utilizar os resultados dessas técnicas como conteúdo necessário depende do tipo de jogo. Um simulador de voo pode utilizar mapas que não sejam atravessáveis por um jogador à pé, por exemplo. Os resultados podem, no entanto, ser utilizados como conteúdo opcional em qualquer tipo de jogos que necessitem de mapas de elevação.

As técnicas são estocásticas, utilizando geradores de números pseudo aleatórios para a geração. Ambas as técnicas são construtivas, principalmente devido a serem fractais.

Essa classificação é resumida na Tabela 4.19.

Método	Tipo de conteúdo	Método de geração	Momento de geração	Corretude do conteúdo	Reprodução dos resultados	Condição de parada
Diamante-quadrado Bisseção de hemisférios	**espaço – mapas externos**	**fractais**	*online* ou *offline*	necessário ou opcional	estocástico	construtivo

Tabela 4.19: Classificação dos métodos geradores de mapas de elevação.

4.7 Cidades e ruas

4.7.1 Big O

O gerador de ruas e edifícios baseado em agentes[27] executa um `for` sobre o número de iterações (linha 101), e este executa cada um dos agentes, que tem complexidade $O(n)$ (linhas 161, 218 e 273). A técnica tem complexidade máxima $O(n^2)$.

Para a técnica de geração de ruas baseada em L-*systems*[28], é utilizada uma função com um `for` (linha 97), que executa uma função auxiliar `generate_partial_network` que utiliza três `for` aninhados (linhas 159, 471 e 472), e, portanto, tem complexidade máxima $O(n^4)$.

Por fim, o inicializador do gerador de ruas e edifícios baseado em L-*systems*[29] utiliza, em um dos pontos mais complexos do código, um `for` (linha 146) que executa uma função auxiliar `generate_buildings_from_sections` com um `for` (linha 1842), que executa outra função auxiliar `generate_residential_building`, com um `for` (linha 2021), que executa uma terceira função auxiliar `render_windows` com dois laços `for` aninhados (linhas 2187 e 2491). Com isso, o gerador tem complexidade $O(n^5)$.

[27] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/towns/agents.cpp

[28] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/towns/roads.cpp

[29] https://bitbucket.org/oliveiranathan/pcg_masters/src/master/src/pcg/towns/buildings.cpp

É importante notar que para alguns valores como parâmetros de entrada o gerador nunca termina. Já a renderização é realizada por meio de dois laços `for` independentes (linhas 39 e 676), cada um executando funções de complexidade linear, e, portanto, apresentando complexidade $O(n^2)$.

A complexidade algorítmica dos métodos é resumida na Tabela 4.20.

4.7.2 Tempo de execução

Para o gerador de ruas e edifícios baseado em agentes, e para o gerador de ruas baseado em L-*systems*, foram realizadas dez execuções e obtida uma média do tempo necessário. Já o gerador de ruas e edifícios baseado em L-*systems* utiliza OpenGL para renderizar os resultados e, portanto, teve os tempos de inicialização, renderização e limpeza de recursos mensurados. Os resultados são listados na Tabela 4.20.

Método	**Big-O**	**Tempo de execução (em ms).**	**Consumo de memória**
Ruas e edifícios baseados em agentes	$O(n^2)$	17714.216[a]	$O(n^2)$
Ruas baseado em L-*systems*	$O(n^4)$	101.353	$O(n^2)$
Ruas e edifícios baseado em L-*systems* - inicialização	$O(n^5)$	632.329	$O(n^3)$
Ruas e edifícios baseado em L-*systems* - renderização	$O(n^2)$	6.012[b]	
Ruas e edifícios baseado em L-*systems* - limpeza dos recursos	$O(n)$	0.650	

[a] Para 5000 iterações dos agentes.
[b] Para um quadro de 3840x2160 pixels.

Tabela 4.20: Complexidade algorítmica, tempo de execução e consumo de memória dos métodos de geração de cidades e ruas.

Com isso, nota-se que, com exceção do gerador de ruas baseado em agentes, as outras técnicas são apropriadas para uso *online*.

4.7.3 Consumo de memória

O gerador de ruas e edifícios baseado em agentes utiliza uma matriz de `char`s que armazena o resultado da geração e dois `vector`s de `vector`s de pares de `int`s utilizados durante a geração. Esses pares de `int`s são coordenadas que referenciam pontos na matriz de `char`s. O limite máximo da complexidade de memória é $O(n^2)$.

Quanto ao gerador de ruas baseado em L-*systems*, são utilizados: uma matriz com os dados de entrada, com dois `double`s e dois `bool`s por elemento; um `vector` de nós com três `int`s por elemento; um `vector` de arestas com cinco `int`s por elemento; e, um `vector` de proposições com um `float` e ao menos dois `char`s por elemento. O vetor de proposições é fornecido pelo usuário, e os vetores de arestas e nós são referências a pontos na matriz. Assim, a complexidade máxima é da ordem de $O(n^2)$.

Por fim, o gerador de ruas e edifícios baseado em L-*systems* utiliza alguns vetores com os dados necessários para a geração da geometria a ser renderizada, mas é esta que tem maior consumo de memória. A complexidade máxima de memória é de $O(n^3)$, variando sobre o tamanho do mapa bidimensional de ruas e a altura dos edifícios gerados.

O consumo de memória dos métodos é resumido na Tabela 4.20.

4.7.4 Qualidade dos resultados

Os resultados das técnicas são exibidos na Figura 4.12.

O gerador de ruas e edifícios baseado em agentes apresenta resultado com média qualidade, exibindo um padrão gerado próximo aos encontrados em jogos com cenários medievais.

O gerador de ruas baseado em L-*systems* é capaz de utilizar mapas de elevação e densidade demográfica para gerar redes de ruas mais próximas às reais, apresentando resultados de alta qualidade.

Por fim, o gerador de ruas e edifícios baseado em L-*systems* produz resultados semelhantes a cidades modernas, apresentando resultados de alta qualidade.

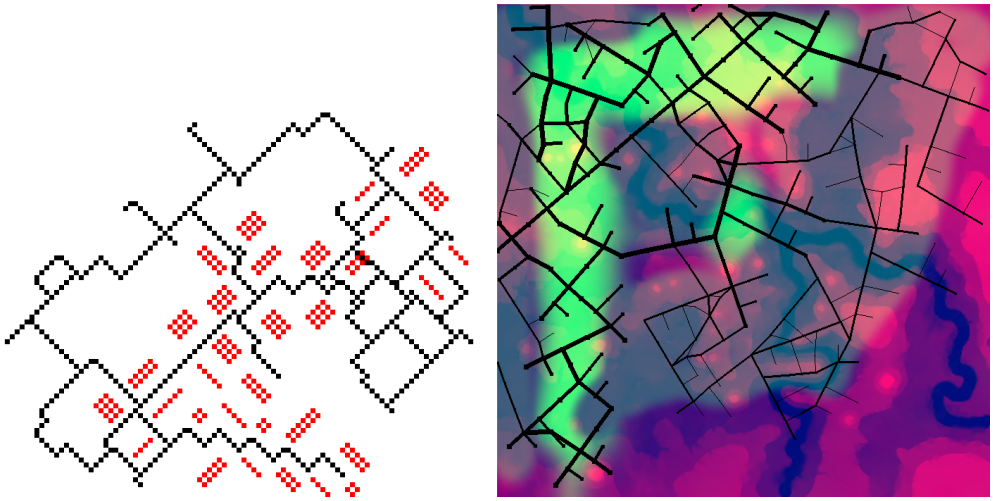
4.7.5 Tipo de algoritmo

Quanto ao tipo de conteúdo gerado, os três métodos podem ser classificados como **sistemas – redes de ruas**. As duas técnicas que também geram edifícios também podem ser classificadas como **sistemas – ambientes urbanos**.

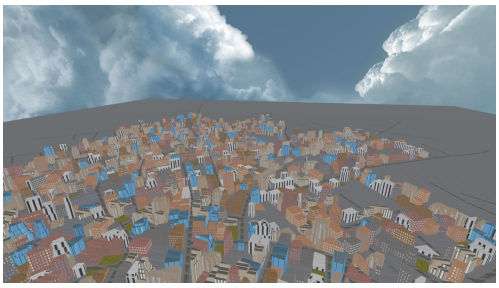
Quanto ao método de geração, o gerador de ruas e edifícios baseado em agentes é classificado como **modelagem e simulação de sistemas complexos**, enquanto as outras técnicas são classificadas como **gramáticas geradoras – sistemas de Lindenmayer**.

Os geradores de ruas baseados em L-*systems* e de ruas e edifícios baseados em L-*systems* podem ser utilizados de forma *online*. O método baseado em agentes é bastante lento e só deve ser usado *offline*.

Figura 4.12: Resultados gerados pelas técnicas: (a) ruas e edifícios, baseado em agentes; (b) ruas, baseado em L-*systems* (áreas em verde são de alta densidade demográfica; em rosa é mostrado o mapa de elevação); (c) ruas e edifícios, baseado em L-*systems*.



(a) (b)



(c)

O gerador de ruas e edifícios baseado em agentes pode produzir ruas que cruzam edifícios, e, portanto, pode não ser correto o suficiente para ser utilizado como conteúdo necessário em certas aplicações. Ainda sim, os resultados podem ser utilizados como conteúdo necessário em algumas situações, como exemplo, os edifícios interceptados por ruas podem ser considerados como uma garagem. Já os outros dois geradores produzem conteúdo correto, podendo ser utilizado como conteúdo necessário.

As três técnicas são estocásticas, fazendo uso de geradores de números pseudo-aleatórios em seus processos de geração. Além disso, são construtivas, não verificando o resultado quanto a nenhuma métrica.

Essa classificação é resumida na Tabela 4.21.

As Tabelas 4.22 e 4.23 resumem os resultados obtidos neste capítulo.

Método	Tipo de conteúdo	Método de geração	Momento de geração	Corretude do conteúdo	Reprodução dos resultados	Condição de parada
Ruas e edifícios baseados em agentes	**sistemas – redes de ruas**	**simulação de sistemas**	*offline*	necessário ou opcional	estocástico	construtivo
Ruas baseado em L-*systems*		**sistemas de Lindenmayer**	*online* ou *offline*			
Ruas e edifícios baseado em L-*systems*						

Tabela 4.21: Classificação dos métodos geradores de cidades e ruas.

Técnica	**Complexidade**	**Tempo de execução**	**Consumo de memória**	**Qualidade dos resultados**	Tipo de algoritmo					
					Tipo de conteúdo gerado	Método de geração	Momento de geração	Corretude do conteúdo	Reprodução dos resultados	Condição de parada
Ruído de valor	$O(1)$	0.036ms[a]	$O(1)$	média	*bits* – texturas	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Ruído de gradiente	$O(1)$	0.012ms[a]	$O(1)$	média	*bits* – texturas	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Ruído de convolução de grade	$O(1)$	0.087ms[a]	$O(1)$	alta	*bits* – texturas	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Ruído de convolução esparsa	$O(1)$	0.445ms[a]	$O(1)$	alta	*bits* – texturas	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Ruído de Perlin	$O(1)$	0.011ms[a]	$O(1)$	alta	*bits* – texturas	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Ruído OpenSimplex	$O(1)$	0.027ms[a]	$O(1)$	alta	*bits* – texturas	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Ruído de Gabor	$O(n)$	0.197ms[a]	$O(1)$	alta	*bits* – texturas	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	determinístico	construtivo
Nomes por máquina de estados finitos	$O(n)$	307.932ns[b]	$O(1)$	média	*bits* – nomes	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Nomes por cadeias de Markov com estatísticas especiais	$O(n)$	10924.2ns[b]	$O(n)$	baixa	*bits* – nomes	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Nomes por cadeias de Markov de ordem n	$O(n)$	1764.58ns[b]	$O(n)$	alta	*bits* – nomes	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	estocástico	gerar-e-testar
Vegetação – subdivisão aleatória de ramos	$O(2^n)$	0.583ms	$O(n^2)$	média	*bits* – vegetação	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário[c] ou opcional	estocástico	construtivo
Vegetação – sistemas de funções iterativas	$O(n^2)$	2.116ms	$O(n^2)$	baixa	*bits* – vegetação	fractal	*online*[d] ou *offline*	necessário[c] ou opcional	estocástico	construtivo
Vegetação – L-*systems*	$O(n^3)$	4.342ms	$O(n^3)$	baixa	*bits* – vegetação	gramáticas geradoras – sistemas de *Lindenmayer*	*online*[d] ou *offline*	necessário[c] ou opcional	estocástico	construtivo

[a] Tempo de execução da implementação em GLSL para uma textura de 256x256 pixels.
[b] Tempo de execução para a geração de um nome.
[c] Ver texto.
[d] Apenas para entradas simples.

Tabela 4.22: Resultados obtidos.

Técnica	**Complexidade**	**Tempo de execução**	**Consumo de memória**	**Qualidade dos resultados**	Tipo de algoritmo					
					Tipo de conteúdo gerado	Método de geração	Momento de geração	Corretude do conteúdo	Reprodução dos resultados	Condição de parada
Fogo – baseado em ruído de fluxo	$O(n)$	0.671ms[a]	$O(1)$	boa	*bits* – fogo, água, pedras e nuvens	geradores de números pseudo-aleatórios	*online*[b]	necessário ou opcional	estocástico	construtivo
Fumaça – baseado em partículas	$O(n)$	0.595ms[a]	$O(1)$	boa	*bits* – fogo, água, pedras e nuvens	geradores de números pseudo-aleatórios	*online*[b]	necessário ou opcional	estocástico	construtivo
Oceanos	$O(n)$	18.557ms[a]	$O(1)$	excelente	*bits* – fogo, água, pedras e nuvens	geradores de números pseudo-aleatórios	*online*[b]	necessário ou opcional	estocástico	construtivo
Nuvens	$O(n)$	3.765ms[a]	$O(1)$	excelente	*bits* – fogo, água, pedras e nuvens	geradores de números pseudo-aleatórios	*online*[b]	necessário ou opcional	estocástico	construtivo
Cavernas com autômatos celulares	$O(n^3)$	8.272ms[c]	$O(n^2)$	alta	espaço – mapas internos	modelagem e simulação de sistemas complexos	*online* ou *offline*	opcional	estocástico	construtivo
Cavernas com agregação por difusão limitada	$O(n^2)$	2.215ms[c]	$O(n^2)$	baixa	espaço – mapas internos	modelagem e simulação de sistemas complexos	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Labirintos	$O(n^2)$	2.504ms[c]	$O(n^2)$	alta	espaço – mapas internos	modelagem e simulação de sistemas complexos	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Masmorras – primeira técnica	$O(n)$	5.801ms[c]	$O(n^2)$	média	espaço – mapas internos	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Masmorras – segunda técnica	$O(n)$	1.902ms[c]	$O(n^2)$	alta	espaço – mapas internos	geradores de números pseudo-aleatórios	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Mapas de elevação por diamante-quadrado	$O(n^2)$	0.397ms[c]	$O(n^2)$	alta	espaço – mapas externos	fractais	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Mapas de elevação por bisseção de hemisférios	$O(n^2)$	9.388ms[c]	$O(n^2)$	alta	espaço – mapas externos	fractais	*online* ou *offline*	necessário[b] ou opcional	estocástico	construtivo
Ruas e edifícios baseados em agentes	$O(n^2)$	17.714s	$O(n^2)$	média	sistemas – ambientes urbanos	modelagem e simulação de sistemas complexos	*offline*	necessário[b] ou opcional	estocástico	construtivo
Ruas baseado em L-*systems*	$O(n^4)$	101.353ms	$O(n^2)$	alta	sistemas – redes de ruas	gramáticas geradoras – sistemas de *Lindenmayer*	*online* ou *offline*	necessário ou opcional	estocástico	construtivo
Ruas e edifícios baseados em L-*systems*	$O(n^2)$	6.012ms[a]	$O(n^3)$	alta	sistemas – ambientes urbanos	gramáticas geradoras – sistemas de *Lindenmayer*	*online* ou *offline*	necessário ou opcional	estocástico	construtivo

[a] Tempo de execução por quadro de 3840x2160 pixels.
[b] Ver texto.
[c] Tempo de execução para uma grade de 256x256 células.

Tabela 4.23: Resultados obtidos.

Capítulo 5

Conclusão

Esta pesquisa teve como objetivo a classificação de técnicas comummente utilizadas para a geração procedural de conteúdo. Esta classificação se faz necessária dado o crescente interesse nessas técnicas tanto por desenvolvedores de jogos digitais, como por pesquisadores da área. Embora esforços tenham sido realizados na direção de classificar os algoritmos de PCG, notavelmente por Hendrikx et al. [62], Kelly e McCabe [71] e Togelius et al. [137], ainda não há registros de trabalhos científicos focados em classificar estas técnicas quanto à complexidade algorítmica, tempo de execução e consumo de memória. Estas métricas têm, no entanto, grande utilidade para o desenvolvimento de algoritmos para PCG em jogos, visto que estes necessitam executar com desempenho e consumo de memória aceitáveis.

O estudo ainda classifica estas técnicas quanto a métricas definidas por Hendrikx et al. [62] e Togelius et al. [137], tais como tipo de conteúdo gerado, método de geração, momento da geração, corretude do conteúdo, reprodutibilidade dos resultados e condição de parada. Estas métricas têm como objetivo agrupar técnicas que sejam semelhantes entre si, de forma a auxiliar seu estudo. Por fim, também foi analisada a qualidade dos resultados das técnicas, visto que, atualmente, um dos principais focos de jogos com alto custo de produção é o realismo. A análise da qualidade dos resultados pode fomentar a criação de técnicas com alta qualidade, o que, por sua vez, pode reduzir os custos e o tempo de produção destes jogos.

Nesta pesquisa, foram utilizadas implementações abertas disponíveis na Internet, principalmente devido ao tempo limitado disponível para sua realização. Isso impediu que técnicas interessantes e com boa qualidade de resultados, mas sem código disponível, fossem selecionadas. No entanto, esse critério foi adotado para que um volume maior de técnicas pudesse ser estudado e melhor otimizado, de forma a se obter tempos de execução próximos do que as técnicas apresentariam em jogos comerciais. Essa otimização foi realizada por meio da localização dos *hot spots* das implementações, com o uso da ferramenta de *profiling* `perf`, disponível no *kernel* do Linux.

Após isso, foram realizadas as medições de tempo de execução através do uso de instrumentação do código com as ferramentas de medição de tempo disponíveis nas próprias linguagens utilizadas, a saber: a biblioteca `chrono` em C++ e a consulta de

`TIMESTAMP` em GLSL. As duas ferramentas foram utilizadas para obter os tempos de execução com precisão de nanosegundos, que posteriormente foram convertidos para suas devidas escalas de tempo, de acordo com os valores obtidos para cada tipo de algoritmo.

Então, foram realizadas as análises de consumo de memória, complexidade algorítmica, método de geração, reprodutibilidade dos resultados e condição de parada por meio da análise da implementação, de forma a se identificar os pontos-chave que caracterizam cada uma destas métricas no código. Por fim, os resultados obtidos pelas técnicas foram analisados para caracterizá-las quanto às métricas tipo de conteúdo gerado, momento de geração, corretude do conteúdo e qualidade dos resultados obtidos.

De posse dessas características, e, determinadas as classes as quais cada técnica pertence, foram, então, conduzidas análises sobre cada tipo de conteúdo gerado, de forma a se obter recomendações de melhores técnicas para os casos de uso comuns a jogos digitais. No entanto, ao analisar as técnicas de geração de conteúdo procedural, foi constatado que nem sempre o resultado mais rápido é o adequado para todos os casos de uso. Com base nestas considerações, nesta dissertação foram realizadas recomendações genéricas, que devem ser analisadas para cada caso de uso específico.

No caso dos ruídos, esses podem ser utilizados como forma de adicionar detalhes a polígonos, de forma a não renderizar cenas com cores planas e artificiais. Neste caso, a escolha do ruído que tenha o menor tempo de execução, como o ruído de Perlin, é uma boa opção, visto que o aspecto visual do ruído não faz diferença. No entanto, quando o ruído for utilizado para a geração de texturas específicas, pode ser necessário um controle maior dos parâmetros de geração, justificando um tempo de execução elevado, como no caso do ruído de Gabor.

Quanto aos geradores de nomes, embora a técnica baseada em máquina de estados finitos seja muito mais rápida que as demais, ajustar os resultados para a geração de nomes em diferentes linguagens requer conhecimento estatístico da linguagem a ser gerada. Portanto, a escolha de uma das técnicas baseadas em cadeias de Markov pode se fazer necessária nos casos em que seja importante gerar nomes em diferentes linguagens, visto que estas podem ser treinadas para gerar diferentes tipos de nomes, necessitando apenas de um arquivo com nomes de exemplo.

Em se tratando dos geradores de vegetação, embora a técnica baseada em subdivisão aleatória seja a mais rápida delas, e possa ser utilizada com bons resultados, quando é necessário mais controle sobre o conteúdo gerado, o uso das outras técnicas se faz necessário. O gerador baseado em sistemas de funções iterativas é melhor adaptado para a geração de folhas e vegetais que apresentem padrões fractais. Já o gerador baseado em L-*systems* provê maior controle sobre os resultados obtidos, ao custo de uma maior complexidade para a modelagem das regras de geração do conteúdo.

Já quanto aos simuladores de elementos naturais, quando é necessário simular apenas fogo ou fumaça, os geradores específicos de cada elemento tem alta qualidade e baixo custo. No entanto, quando for necessária a simulação de fogo e fumaça, o simulador baseado em partículas pode ser configurado para alterar a cor das partículas de acordo com o tempo de vida delas, e, com isso, simular tanto fogo quanto fumaça. Embora

os resultados não sejam tão bons para a geração de fogo, geralmente esses elementos não necessitam de muita qualidade para que o resultado seja bom. Apesar disso, em um momento em que o jogo desperte a atenção do jogador para uma lareira, por exemplo, o gerador de fogo baseado em ruído de fluxo, embora apresente maior custo de processamento, produz resultados com qualidade muito maior do que o baseado em partículas.

Os geradores de cavernas apresentam qualidade inversamente proporcional ao tempo de execução dos algoritmos, sendo que a técnica com autômatos celulares apresenta grande similaridade com cavernas reais. A técnica com agregação por difusão limitada é mais rápida, mas só exibe bons resultados para mapas pequenos e, ainda assim, gera cavernas pouco similares às naturais.

Para os geradores de masmorras, a segunda técnica, além de ter menor custo computacional, apresenta resultados bem mais próximos dos utilizados por jogos *roguelike*, apresentando salas e corredores que preenchem quase toda a área disponível no mapa. No entanto, a primeira técnica apresenta resultados mais naturais no caso de o jogo se situar em uma masmorra construída no subsolo.

Quanto aos geradores de mapas de elevação, o gerador por diamante-quadrado é muito mais rápido do que o por bisseção de hemisférios. No entanto, quando utilizados para a geração de planetas inteiros, o gerador baseado em bisseção apresenta resultados muito mais realistas. Já a técnica por diamante-quadrado gera conteúdo mais realista para pequenas áreas de mapas.

Por fim, os geradores de ruas e cidades, embora apresentem grande diferença nos tempos de execução, também devem ser escolhidos com base no tipo de cidade a ser criada. O gerador baseado em agentes gera mapas que são bastante parecidos com vilarejos medievais. O gerador de ruas baseado em L-*systems*, que trata de valores de densidade populacional e relevo, gera redes de ruas bastante próximas das encontradas em grandes cidades. Já o gerador de cidades baseado em L-*systems* é capaz de gerar resultados muito próximos dos encontrados em grandes cidades. No entanto, o gerador baseado em agentes não deve ser utilizado *online*, visto que o tempo de execução desta técnica é elevado.

Esta pesquisa também resultou em um repositório aberto, contendo os códigos-fonte das implementações das técnicas tratadas. Este repositório está disponível em https://www.bitbucket.org/oliveiranathan/pcg_masters.

5.1 Trabalhos futuros

O trabalho não se apresenta como definitivo, pelo contrário, levanta novas questões de investigação, como:

- analisar as dezenas (ou até mesmo centenas) de outras técnicas disponíveis, que não puderam ser abordadas nesta oportunidade;
- mensurar os tempos de execução em diferentes CPUs e GPUs;

- efetuar a medição dos tempos de execução em plataformas móveis, como Android e iOS, que atualmente são cada vez mais utilizadas como plataformas de jogos. Essas medições não foram realizadas nesta pesquisa devido à diferença das plataformas móveis com relação aos PCs, visto que a abordagem utilizada nesta pesquisa obteria, nas plataformas móveis, resultados não só incorretos, como ainda, com pouca aplicabilidade.

Referências Bibliográficas

[1] Activision. *River Raid*. [Fita cassete], [Disquete]. 1982.

[2] Alexander Alekseev. *Seascape*. 2014. URL: https://www.shadertoy.com/view/Ms2SD1 (acesso em 02/07/2016).

[3] Ryan Archer. *Terragen Image Gallery*. 2013. URL: http://planetside.co.uk/galleries/terragen-gallery (acesso em 28/07/2015).

[4] ARM. *Use frame time instead of FPS for comparisons*. 2013. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0555a/BEIGDEGC.html (acesso em 22/06/2016).

[5] ARM MALI. *Mali OpenGL ES SDK Documentation*. 2013. URL: http://malideveloper.arm.com/downloads/deved/tutorial/SDK/linux/2.4.4/html/index.html (acesso em 16/09/2016).

[6] Artoftransformation. *Rogue Unix Screenshot CAR*. Maio de 2008. URL: https://en.wikipedia.org/wiki/File:Rogue_Unix_Screenshot_CAR.PNG (acesso em 26/07/2015).

[7] Franz Aurenhammer. "Voronoi Diagrams&Mdash;a Survey of a Fundamental Geometric Data Structure". Em: *ACM Comput. Surv.* 23.3 (set. de 1991), pp. 345–405. ISSN: 0360-0300. DOI: 10.1145/116873.116880. URL: http://doi.acm.org/10.1145/116873.116880.

[8] Jim Babcock. *Cellular Automata Method for Generating Random Cave-Like Levels*. 2005. URL: http://www.jimrandomh.org/misc/caves.html (acesso em 20/02/2016).

[9] Chris Baker. *Nimrod, the World's First Gaming Computer*. Fev. de 2010. URL: http://www.wired.com/2010/06/replay/ (acesso em 19/05/2015).

[10] Matt Barton. *Interview with Daniel M. Lawrence, CRPG Pioneer and Author of Telengard*. Jun. de 2007. URL: http://armchairarcade.com/neo/node/1366 (acesso em 24/05/2015).

[11] James Batchelor. *There are now 170 studios across the nation, and Mojang's a major revenue driver*. 2014. URL: http://www.develop-online.net/news/swedish-games-industry-grew-by-80-this-year/0197509 (acesso em 21/01/2015).

[12] Chris Bateman. *Meet Bertie the Brain, the world's first arcade game, built in Toronto*. Ago. de 2014. URL: http://spacing.ca/toronto/2014/08/13/meet-bertie-brain-worlds-first-arcade-game-built-toronto/ (acesso em 19/05/2015).

[13] Bay 12 Games. *Dwarf Fortress*. 2015. URL: http://www.bay12games.com/dwarves/screens/arena2.html (acesso em 27/07/2015).

[14] Bayo. *Mystery House - Apple II - 2*. Dez. de 2005. URL: https://commons.wikimedia.org/wiki/File:Mystery_House_-_Apple_II_-_2.png (acesso em 26/07/2015).

[15] Bethesda Softworks LLC. *The Elder Scrolls*. 2015. URL: http://www.elderscrolls.com/ (acesso em 30/05/2015).

[16] Fernando Bevilacqua. "Ferramenta para geração em tempo real de bordas de mapas virtuais pseudo-infinitos para jogos 3D". Universidade Federal de Santa Maria, 2008.

[17] Michael Birken. *Star Trek 1971 Text Game*. Jul. de 2008. URL: http://www.codeproject.com/Articles/28228/Star-Trek-1971-Text-Game (acesso em 20/05/2015).

[18] Blender Foundation. *Blender 2.74: Features*. 2015. URL: http://www.blender.org/features/ (acesso em 29/05/2015).

[19] Blender Foundation. *Features*. 2015. URL: https://www.blender.org/features/ (acesso em 27/07/2015).

[20] Blizzard Entertainment. *Classic Games*. 2015. URL: http://us.blizzard.com/en-us/games/legacy/ (acesso em 28/05/2015).

[21] Blizzard Entertainment Inc. *Mists of Pandaria*. Out. de 2011. URL: http://us.battle.net/wow/en/media/screenshots/mop?view=wowx4-screenshot-14 (acesso em 27/07/2015).

[22] Syd Bolton. *The "Other"Lord British: Interview with Jez San, OBE*. 201? URL: http://www.armchairempire.com/Interviews/jez-san-interview.htm (acesso em 27/05/2015).

[23] Bridget Borgobello. *The 'CRT Amusement Device' that spawned a multi-million dollar industry*. Maio de 2011. URL: http://www.gizmag.com/first-video-game/18695/ (acesso em 25/07/2015).

[24] Paul Bourke. *How to create the IFS Fern*. 1988. URL: http://paulbourke.net/fractals/ifs_fern_a/ (acesso em 02/07/2016).

[25] Kevin Bowen. *Space Invaders*. 2004. URL: https://web.archive.org/web/20040902162808/http://archive.gamespy.com/legacy/halloffame/spaceinvaders.shtm (acesso em 26/07/2015).

[26] Jim Bowery. *Spasim (1974) The First First-Person-Shooter 3D Multiplayer Networked Game*. Abr. de 2001. URL: http://web.archive.org/web/20010410145350/http://www.geocities.com/jim_bowery/spasim.html (acesso em 21/05/2015).

[27] Martín Candela Calabuig. *MapGenerator*. 2013. URL: https://github.com/Rellikiox/MapGenerator/blob/master/MarkovNames/MarkovNames.cpp (acesso em 20/02/2016).

[28] Daniel Michelon De Carli. "Geração Procedural de Cenários 3D de Cânions com Foco em Jogos Digitais". Universidade Federal de Santa Maria, 2012.

[29] Daniel Michelon De Carli et al. "A Survey of Procedural Content Generation Techniques Suitable to Game Development." Em: *SBGames*. 2011, pp. 26–35.

[30] CBS Interactive Inc. *Spasim*. Mar. de 2013. URL: http://www.giantbomb.com/spasim/3030-23131/ (acesso em 25/07/2015).

[31] Guoning Chen et al. "Interactive Procedural Street Modeling". Em: *ACM SIGGRAPH 2008 Papers*. SIGGRAPH '08. Los Angeles, California: ACM, 2008, 103:1–103:10. ISBN: 978-1-4503-0112-1. DOI: 10.1145/1399504.1360702. URL: http://doi.acm.org/10.1145/1399504.1360702.

[32] ChildOfTheMoon83. *Akalabeth: World of Doom*. Jun. de 2011. URL: https://en.wikipedia.org/wiki/File:Akalabeth.jpg (acesso em 26/07/2015).

[33] B. Chopard e M. Droz. *Cellular Automata Modeling of Physical Systems*. Alea Saclay. Cambridge University Press, 2005. URL: http://books.google.com/books?id=jfcxaoNZuMUC.

[34] Computer History Museum. *This Day in History. November 29, 1972*. 2015. URL: http://www.computerhistory.org/tdih/November/29/ (acesso em 21/05/2015).

[35] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. The MIT Press, 2009. ISBN: 0262033844.

[36] Hannah Craighead, Tana Tanoi e Cameron Bryers. *Procedural city generation*. 2015. URL: https://github.com/TanaTanoi/new-londis-city (acesso em 02/07/2016).

[37] Geoff Crammond. *The Sentinel*. [Fita cassete], [Disquete]. 1986.

[38] Crytek GmbH. *CryEngine*. 2015. URL: http://cryengine.com/ (acesso em 29/05/2015).

[39] Paul Davidsson. "Multi Agent Based Simulation: Beyond Social Simulation". English. Em: *Multi-Agent-Based Simulation*. Ed. por Scott Moss e Paul Davidsson. Vol. 1979. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 97–107. ISBN: 978-3-540-41522-0. DOI: 10.1007/3-540-44561-7_7. URL: http://dx.doi.org/10.1007/3-540-44561-7_7.

[40] Jakub Debski. *Mazes*. 2006. URL: http://rec.games.roguelike.development.narkive.com/9nTSReHd/mazes (acesso em 20/02/2016).

[41] decoydoctorpus. *procedurally generated offense*. Ago. de 2008. URL: https://doctorpus.wordpress.com/2008/08/11/ (acesso em 31/07/2015).

[42] Digital Equipment Computer Users Society. *DECUS Program Library Catalog for PDP-8, FOCAL8*. Jul. de 1973. URL: http://pdp-8.org/scans/highgate/decus/decus_lib_73b.pdf (acesso em 20/05/2015).

[43] Philip Michel Duarte. "Geração Procedural de Cenários Orientada a Objetivos Philip Michel Duarte Geração Procedural de Cenários Orientada a Objetivos". Universidade Federal do Rio Grande do Norte, 2012.

[44] David S. Ebert et al. *Texturing and Modeling, Third Edition: A Procedural Approach (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, 2002. ISBN: 1558608486.

[45] Benj Edwards. *VC&G Interview: Carol Shaw, The First Female Video Game Developer*. Out. de 2011. URL: http://www.vintagecomputing.com/index.php/archives/800 (acesso em 31/07/2015).

[46] Hugo Elias. *Cloud Cover*. 20?? URL: http://freespace.virgin.net/hugo.elias/models/m_clouds.htm (acesso em 03/04/2016).

[47] Encyclopædia Britannica, Inc. *Christopher Strachey*. 2015. URL: http://global.britannica.com/EBchecked/topic/752661/Christopher-Strachey (acesso em 19/05/2015).

[48] Martin Engilberge e Florian Depraz. *Procedural road network generation*. 2014. URL: https://github.com/pokitoz/Procedural-road-network-generation (acesso em 02/07/2016).

[49] Enterbrain Inc. *MAKE YOUR OWN GAME WITH THE RPG MAKER SERIES!* 2015. URL: http://www.rpgmakerweb.com/ (acesso em 29/05/2015).

[50] Epic Games. *What is Unreal Engine 4?* 2015. URL: https://www.unrealengine.com/what-is-unreal-engine-4 (acesso em 29/05/2015).

[51] Frank Eva. "Starglider II". Em: *Computer Gaming World* (dez. de 1988): *Entertaining The Troops*. URL: http://www.cgwmuseum.org/galleries/issues/cgw_54.pdf (acesso em 24/05/2015).

[52] Travis Fahs. *IGN Presents the History of SEGA*. Abr. de 2009. URL: http://uk.ign.com/articles/2009/04/21/ign-presents-the-history-of-sega?page=8 (acesso em 27/05/2015).

[53] David Fiedler. "The Unix Tutorial. Part 1: An Introduction to Features and Facilities". Em: *Byte Magazine* (ago. de 1983): *The C Language*. URL: https://archive.org/stream/byte-magazine-1983-08/1983_08_BYTE_08-08_The_C_Language (acesso em 20/05/2015).

[54] Lily Ford. *Prince of Persia (1989)*. Dez. de 2014. URL: http://princeofpersia.wikia.com/wiki/Prince_of_Persia_(1989) (acesso em 26/07/2015).

[55] GamesRadar US. *Gaming's most important evolutions*. Out. de 2010. URL: http://www.gamesradar.com/gamings-most-important-evolutions/?page=3 (acesso em 24/05/2015).

[56] Alison Gazzard. "The Platform and the Player: exploring the (hi)stories of Elite". Em: *Game Studies* (dez. de 2013). URL: http://gamestudies.org/1302/articles/agazzard (acesso em 24/05/2015).

[57] J.T.T. Goldsmith e M.E. Ray. *Cathode-ray tube amusement device*. US Patent 2,455,992. Dez. de 1948. URL: http://www.google.com/patents/US2455992.

[58] Laurent Gomila. *Simple and Fast Multimedia Library*. 2015. URL: http://www.sfml-dev.org/ (acesso em 29/05/2015).

[59] J. M. Graetz. *The origin of Spacewar*. Ago. de 1981. URL: http://www.wheels.org/spacewar/creative/SpacewarOrigin.html (acesso em 20/05/2015).

[60] Grandpafootsoldier. *Diablo screen*. Dez. de 2006. URL: https://en.wikipedia.org/wiki/File:Diabloscreen.jpg (acesso em 27/07/2015).

[61] Hahnchen. *Star Fox - Gameplay*. Set. de 2012. URL: https://en.wikipedia.org/wiki/File:Star_Fox_-_Gameplay.png (acesso em 27/07/2015).

[62] Mark Hendrikx et al. "Procedural Content Generation for Games: A Survey". Em: *ACM Trans. Multimedia Comput. Commun. Appl.* 9.1 (fev. de 2013), 1:1–1:22. ISSN: 1551-6857. DOI: 10.1145/2422956.2422957. URL: http://doi.acm.org/10.1145/2422956.2422957.

[63] Hoovooloo. *The Wonderful Computers of Clive Sinclair*. Nov. de 2004. URL: http://h2g2.com/edited_entry/A821648 (acesso em 24/05/2015).

[64] Stephen Hutcheon. *The video games boom has yet to come*. Ed. por The Age. Jun. de 1983. URL: https://news.google.com/newspapers?id=fC5VAAAAIBAJ&sjid=npQDAAAAIBAJ&pg=4131,3188851&hl=en (acesso em 22/05/2015).

[65] IDV Inc. *Norway Spruce*. 2015. URL: https://store.speedtree.com/product/norway-spruce/ (acesso em 28/07/2015).

[66] IDV Inc. *SpeedTree*. 2015. URL: http://www.speedtree.com/ (acesso em 29/05/2015).

[67] *IEEE Standard for Prefixes for Binary Multiples*. DOI: 10.1109/ieeestd.2009.5254933. URL: http://dx.doi.org/10.1109/IEEESTD.2009.5254933.

[68] Nathaniel Inman. *Diffusion-limited aggregation*. 2012. URL: http://www.roguebasin.com/index.php?title=Diffusion-limited_aggregation (acesso em 20/02/2016).

[69] International Arcade Museum. *Pong*. 2015. URL: http://www.arcade-museum.com/game_detail.php?game_id=9074 (acesso em 25/07/2015).

[70] Joi Ito. *Spacewar! running on the Computer History Museum's PDP-1*. Maio de 2007. URL: https://commons.wikimedia.org/wiki/File:Spacewar!-PDP-1-20070512.jpg (acesso em 25/07/2015).

[71] George Kelly e Hugh McCabe. "A Survey of Procedural Techniques for City Generation". Em: *ITB Journal* (2006), pp. 87–130.

[72] Steven Kent. *The Ultimate History of Video Games: From Pong to Pokemon–The Story Behind the Craze That Touched Our Lives and Changed the World*. Three Rivers Press, 2001.

[73] KHRONOS Group. *OpenGL*. 2015. URL: https://www.opengl.org/ (acesso em 25/01/2015).

[74] Donald E. Knuth. "Big Omicron and Big Omega and Big Theta". Em: *SIGACT News* 8.2 (abr. de 1976), pp. 18–24. ISSN: 0163-5700. DOI: 10.1145/1008328.1008329. URL: http://doi.acm.org/10.1145/1008328.1008329.

[75] Jari Komppa. *Procedural tree mesh generator (and editor)*. 2015. URL: https://github.com/jarikomppa/proctree (acesso em 02/07/2016).

[76] Ares Lagae et al. "Procedural noise using sparse Gabor convolution". Em: *TOG* 28.3 (jul. de 2009), p. 1. DOI: 10.1145/1531326.1531360. URL: http://dx.doi.org/10.1145/1531326.1531360.

[77] A. Lagae et al. "A Survey of Procedural Noise Functions". Em: *Computer Graphics Forum* 29.8 (out. de 2010), pp. 2579–2600. DOI: 10.1111/j.1467-8659.2010.01827.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01827.x.

[78] Jeff Lait. *Random Name Generator*. 2008. URL: https://groups.google.com/forum/#!msg/rec.games.roguelike.development/cdgIgD3Plds/MG2Y6LeQXBgJ (acesso em 20/02/2016).

[79] Mathieu Larive e Veronique Gaildrat. "Wall Grammar for Building Generation". Em: *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*. GRAPHITE '06. Kuala Lumpur, Malaysia: ACM, 2006, pp. 429–437. ISBN: 1-59593-564-9. DOI: 10.1145/1174429.1174501. URL: http://doi.acm.org/10.1145/1174429.1174501.

[80] Gabriel de Oliveira Barbosa Leite e Edirlei Soares De Lima. "Geração Procedural de Mapas para Jogos 2D". Em: *Proceedings of SBGames 2015* (2015), pp. 244–247.

[81] Lighthouse3D. *GLSL Tutorial - Texturing with Images*. 201? URL: http://www.lighthouse3d.com/tutorials/glsl-tutorial/texturing-with-images/ (acesso em 30/07/2015).

[82] Lucasfilm Games. *Rescue on Fractalus!* [Fita cassete], [Disquete]. Mar. de 1984.

[83] Adam Lusher. *Elite: the game that changed the world*. Ago. de 2014. URL: http://www.telegraph.co.uk/games/11051122/Elite-the-game-that-changed-the-world.html (acesso em 31/07/2015).

[84] Tony Lyon. *Zaxxon*. Set. de 2013. URL: http://retrovideogamesystems.com/zaxxon/ (acesso em 23/05/2015).

[85] Maggijanek. *Nice Landscape Background*. Maio de 2013. URL: https://www.reddit.com/r/Minecraft/comments/1e92yw/nice_landscape_background/ (acesso em 30/07/2015).

[86] Jimmy Maher. *Akalabeth*. 2011. URL: http://www.filfre.net/2011/12/akalabeth/ (acesso em 10/01/2015).

[87] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman e Company, 1982. ISBN: 0716711869.

[88] Nich Maragos. *The Old Order Passeth*. 201? URL: http://www.1up.com/features/essential-50-final-fantasy-vii (acesso em 28/05/2015).

[89] Dominik Marczuk. *Markov chains-based name generation*. 2010. URL: http://www.roguebasin.com/index.php?title=Markov_chains-based_name_generation (acesso em 20/02/2016).

[90] Paul Martz. *Generating Random Fractal Terrain*. 1997. URL: http://www.gameprogrammer.com/fractal.html (acesso em 20/02/2016).

[91] Mike MCClelland. *Make a Particle Explosion Effect*. Out. de 2009. URL: http://www.gamedev.net/page/resources/_/creative/visual-arts/make-a-particle-explosion-effect-r2701 (acesso em 30/07/2015).

[92] Jordan Mechner. *The Making of Prince of Persia: Journals 1985 - 1993*. CreateSpace Independent Publishing Platform, 2011.

[93] Linnéa Mellblom. *WebGL procedural fire*. 2015. URL: https://github.com/lmellblom/fire-webGL (acesso em 02/07/2016).

[94] MindControlDX, netherh e underww. *C++ Example of Dungeon-Building Algorithm*. 2015. URL: http://www.roguebasin.com/index.php?title=C%2B%2B_Example_of_Dungeon-Building_Algorithm (acesso em 20/02/2016).

[95] Fábio Markus Miranda, Carlúcio S. Cordeiro e Luiz Chaimowicz. "Um Sistema para Geração Procedural de Terrenos Pseudo-Infinitos em Tempo-Real Utilizando GPU e CPU". Em: *VIII Brazilian Symposium on Games and Digital Entertainment* (2009), pp. 113–116.

[96] Mojang. *Minecraft*. 2015. URL: https://minecraft.net/ (acesso em 30/05/2015).

[97] Pascal Müller et al. "Procedural Modeling of Buildings". Em: *ACM SIGGRAPH 2006 Papers*. SIGGRAPH '06. Boston, Massachusetts: ACM, 2006, pp. 614–623. DOI: 10.1145/1179352.1141931. URL: http://doi.acm.org/10.1145/1179352.1141931.

[98] Museum of Electronic Games & Art. *T42 - Tennis for Two*. Maio de 2011. URL: http://www.m-e-g-a.org/research-education/research/t42-tennis-for-two/ (acesso em 25/07/2015).

[99] Abbas Naderi-Afooshteh. *3D L-System for generating 3D trees and bushes from simple rules*. 2014. URL: https://github.com/abiusx/L3D (acesso em 02/07/2016).

[100] NaturalMotion. *Morpheme with Euphoria*. 2015. URL: http://www.naturalmotion.com/middleware/euphoria/ (acesso em 29/05/2015).

[101] notydino. *High Train Traffic Solutions*. Mar. de 2015. URL: http://steamcommunity.com/sharedfiles/filedetails/?id=408643569&insideModal=1 (acesso em 30/07/2015).

[102] NVIDIA Corporation. *The Elder Scrolls V: Skyrim*. Nov. de 2011. URL: http://www.geforce.com/games-applications/pc-games/elder-scrolls-v-skyrim (acesso em 27/07/2015).

[103] Office of Scientific and Technical Information. *Video Games – Did They Begin at Brookhaven?* Jan. de 2011. URL: http://www.osti.gov/accomplishments/videogame.html (acesso em 20/05/2015).

[104] Jacob Olsen. *Realtime procedural terrain generation - realtime synthesis of eroded fractal terrain for use in computer games*. 2004.

[105] John Olsson e drow. *Fractal World Generator*. 1999. URL: http://donjon.bin.sh/code/world/ (acesso em 20/02/2016).

[106] Adalberto Bosco Castro Pereira. "Um sistema fuzzy para geração de tarefas de ensino de leitura e escrita em um jogo digital". Universidade Federal do Pará, 2012.

[107] Ken Perlin. "An Image Synthesizer". Em: *SIGGRAPH Comput. Graph.* 19.3 (jul. de 1985), pp. 287–296. ISSN: 0097-8930. DOI: 10.1145/325165.325247. URL: http://doi.acm.org/10.1145/325165.325247.

[108] Ken Perlin. *Noise and Turbulence*. 1999. URL: http://mrl.nyu.edu/~perlin/doc/oscar.html#noise (acesso em 17/01/2016).

[109] Markus Persson. *Terrain generation, Part 1*. 2011. URL: http://notch.tumblr.com/post/3746989361/terrain-generation-part-1 (acesso em 21/01/2015).

[110] PirateMink. *Ultima1-2*. Jan. de 2008. URL: https://en.wikipedia.org/wiki/File:Ultima1-3.gif (acesso em 26/07/2015).

[111] P. Prusinkiewicz et al. *The Algorithmic Beauty of Plants*. The Virtual Laboratory. Springer New York, 1996. ISBN: 9780387946764.

[112] Quad Software. *Application and Scenes Screenshots*. 201? URL: http://www.quadsoftware.com/index.php?m=section&sec=product&subsec=editor&target=editor_screenshots (acesso em 28/07/2015).

[113] Jonathan Roberts e Ke Chen. "Learning-Based Procedural Content Generation". Em: *CoRR* abs/1308.6415 (2013). URL: http://arxiv.org/abs/1308.6415.

[114] Rockstar Games. *Grand Theft Auto*. 2015. URL: http://www.rockstargames.com/grandtheftauto/ (acesso em 30/05/2015).

[115] Rockstar Games. *The Great Outdoors*. 2015. URL: http://www.rockstargames.com/V/lsbc/the-great-outdoors (acesso em 27/07/2015).

[116] Brandon Sheffield. *GDC 2011: Yu Suzuki: 'Sega Will Probably Let Me Make Shenmue 3'*. Mar. de 2011. URL: http://www.gamasutra.com/view/news/33329/GDC_2011_Yu_Suzuki_Sega_Will_Probably_Let_Me_Make_Shenmue_3.php (acesso em 28/05/2015).

[117] Jo Shields. *Review: Neverwinter Night [PC]*. Ago. de 2002. URL: http://hexus.net/gaming/reviews/pc/411-neverwinter-night-pc/ (acesso em 29/05/2015).

[118] Marcelo Cardoso Silva, Felipe M G França e Giordano Ribeiro Eulalio Cabral. "Construindo Trilhas Sonoras Dinâmicas Em Jogos Utilizando Sistemas Fuzzy". Em: *Proceedings of the SBGames 2014* (2014), pp. 974–977.

[119] Marlene Simmons. *Bertie the Brain programmer heads science council*. Ed. por Ottawa Citizen. Out. de 1975. URL: https://news.google.com/newspapers?id=rKYyAAAAIBAJ&sjid=pe0FAAAAIBAJ&pg=916,3790974&dq=josef-kates&hl=en (acesso em 19/05/2015).

[120] Ruben Smelik et al. "Integrating procedural generation and manual editing of virtual worlds". Em: (2010). DOI: 10.1145/1814256.1814258. URL: http://dx.doi.org/10.1145/1814256.1814258.

[121] Chuck Smith. *Picture of the Nimrod exhibit at the Computerspielemuseum in Berlin*. Mar. de 2011. URL: https://commons.wikimedia.org/wiki/File:Nimrod_in_Computerspielemuseum.jpg (acesso em 25/07/2015).

[122] Keith Smith. *Nimrod, the World's First Gaming Computer*. Maio de 2014. URL: http://allincolorforaquarter.blogspot.co.uk/2014/05/what-were-first-ten-coin-op-video-games.html (acesso em 20/05/2015).

[123] Kurt Spencer. *Visually axis-decorrelated coherent noise algorithm based on the Simplectic honeycomb*. Out. de 2014. URL: https://gist.github.com/KdotJPG/b1270127455a94ac5d19 (acesso em 10/01/2016).

[124] Rockstar Spouses. *Wives of Rockstar San Diego employees have collected themselves*. Jul. de 2010. URL: http://www.gamasutra.com/blogs/RockstarSpouse/20100107/4032/Wives_of_Rockstar_San_Diego_employees_have_collected_themselves.php (acesso em 30/05/2015).

[125] Francis Spufford. *Masters of their universe*. 2003. URL: http://www.theguardian.com/books/2003/oct/18/features.weekend (acesso em 10/01/2015).

[126] Sreejithk2000. *Doom gibs*. Jul. de 2010. URL: https://en.wikipedia.org/wiki/File:Doom_gibs.png (acesso em 27/07/2015).

[127] George Stiny e James Gips. "Shape Grammars and the Generative Specification of Painting and Sculpture". Em: *Proceedings of IFIP Congress 1971*. 1971.

[128] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional, 2013. ISBN: 0321563840.

[129] Techland. *Chrome Engine*. 2013. URL: http://technology.techland.pl/ (acesso em 29/05/2015).

[130] Tecnalia. *Reconstruction of Vitoria-Gasteiz*. 201? URL: http://www.esri.com/software/cityengine/industries/reconstruction-of-vitoria-gasteiz (acesso em 28/07/2015).

[131] The Association for UK Interactive Entertainment. *UK VIDEO GAMES FACT SHEET*. 2015. URL: http://ukie.org.uk/sites/default/files/cms/UK%20Games%20Industry%20Fact%20Sheet%209%20January%202015.pdf (acesso em 21/01/2015).

[132] The Dot Eaters. *gun-fight*. Abr. de 2013. URL: http://thedoteaters.com/?attachment_id=1908 (acesso em 26/07/2015).

[133] The International Arcade Museum. *Frogger*. 2015. URL: http://www.arcade-museum.com/game_detail.php?game_id=7857 (acesso em 23/05/2015).

[134] The Khronos Group. *OpenGL Shading Language*. 2015. URL: https://www.opengl.org/documentation/glsl/ (acesso em 10/01/2016).

[135] The Sierra Help Pages. *Mystery House Help*. 2006. URL: http://www.sierrahelp.com/Games/MysteryHouseHelp.html (acesso em 23/05/2015).

[136] The Sierra Help Pages. *Wizard and the Princess Help*. 2006. URL: http://www.sierrahelp.com/Games/WizardAndThePrincessHelp.html (acesso em 23/05/2015).

[137] Julian Togelius et al. "Search-Based Procedural Content Generation: A Taxonomy and Survey". Em: *IEEE Trans. Comput. Intell. AI Games* 3.3 (set. de 2011), pp. 172–186. DOI: 10.1109/tciaig.2011.2148116. URL: http://dx.doi.org/10.1109/TCIAIG.2011.2148116.

[138] Torus Knot Software. *About OGRE*. 2015. URL: http://www.ogre3d.org/about (acesso em 29/05/2015).

[139] Translucid2k4. *Shenmue - Quicktime event*. Jul. de 2006. URL: https://en.wikipedia.org/wiki/File:Shenmue_quicktimeevent.jpg (acesso em 27/07/2015).

[140] Tyan23. *File:A5200 Rescue on Fractalus.png*. Out. de 2007. URL: https://en.wikipedia.org/wiki/File:A5200_Rescue_On_Fractalus.png (acesso em 31/07/2015).

[141] Ubisoft Entertainment. *Far Cry 4*. 2014. URL: http://far-cry.ubi.com/en-GB/home/index.aspx (acesso em 30/05/2015).

[142] Unity Technologies. *Unity*. 2015. URL: http://unity3d.com/ (acesso em 29/05/2015).

[143] Luke Villapaz. *'GTA 5' Costs $265 Million To Develop And Market, Making It The Most Expensive Video Game Ever Produced: Report*. 2013. URL: http://www.ibtimes.com/gta-5-costs-265-million-develop-market-making-it-most-expensive-video-game-ever-produced-report (acesso em 10/01/2015).

[144] Steve Wallace e Mtvee. *Dungeon builder written in Python*. 2014. URL: http://www.roguebasin.com/index.php?title=Dungeon_builder_written_in_Python (acesso em 20/02/2016).

[145] Joe Watson. *NWN Level-Design*. Jun. de 2011. URL: http://multimasky.com/showcase/2-nwn-level-design/ (acesso em 27/07/2015).

[146] Jonah Weiner. *Where Do Dwarf-Eating Carp Come From?* Jul. de 2011. URL: http://www.nytimes.com/2011/07/24/magazine/the-brilliance-of-dwarf-fortress.html?pagewanted=all&_r=0 (acesso em 30/05/2015).

[147] Eric W. Weisstein. *Cellular Automaton*. 201? URL: http://mathworld.wolfram.com/CellularAutomaton.html (acesso em 01/07/2016).

[148] Glenn R. Wichman. *A Brief History of "Rogue"*. 1997. URL: http://www.wichman.org/roguehistory.html (acesso em 23/05/2015).

[149] David Winter. *Noughts And Crosses - The oldest graphical computer game*. 2013. URL: http://www.pong-story.com/1952.htm (acesso em 19/05/2015).

[150] Peter Wonka et al. "Instant Architecture". Em: *ACM Transaction on Graphics* 22.3 (jul. de 2003). Proceedings ACM SIGGRAPH 2003, pp. 669–677. ISSN: 0730-0301. URL: http://www.cg.tuwien.ac.at/research/publications/2003/Wonka-2003-Ins/.

[151] Tyler Wymer. *Simple town and road procedural content algorithm for a class project*. 2012. URL: https://github.com/twymer/town-gen (acesso em 02/07/2016).

[152] Marshall C. Yovits. *Advances in Computers, Vol. 23*. Academic Press, 1984.