Geração procedural de ambientes virtuais para jogos de mesa

Lucas Margato Ladeira

JUIZ DE FORA

JULHO, 2019

Geração procedural de ambientes virtuais para jogos de mesa

Lucas Margato Ladeira

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Marcelo Caniato Renhe

JUIZ DE FORA

JULHO, 2019

GERAÇÃO PROCEDURAL DE AMBIENTES VIRTUAIS PARA JOGOS DE MESA

Lucas Margato Ladeira

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Marcelo Caniato Renhe
Doutor em Engenharia de Sistemas e Computação

Igor de Oliveira Knop
Doutor em Modelagem Computacional

Rodrigo Luis de Souza da Silva
Doutor em Engenharia Civil

JUIZ DE FORA

09 DE JULHO, 2019

Aos familiares e amigos, que tiveram participação direta ou indireta nesse momento.

Em especial aos meus pais Maurício e Adriana, por todo o amor, carinho e compreensão, e ao meu orientador Marcelo por toda a dedicação e presteza.

Resumo

A criação de jogos digitais envolve diversas áreas da computação, e os recursos visuais e sonoros utilizados são em grande parte responsáveis pela imersividade destes jogos. Porém, no caso dos jogos de mesa, os recursos para gerar imersão são mais limitados. Assim, a proposta deste trabalho é utilizar a geração procedural de conteúdo, um recurso clássico dos jogos digitais, como suporte a jogos não-digitais, com o objetivo de atingir um grau de imersão maior. São apresentados e comparados algoritmos de geração procedural de conteúdo que se adaptem bem ao problema em questão, apontando quais os pontos positivos e negativos de cada um. O foco da aplicação é a geração de ambientes virtuais para serem usados como complemento de jogos de tabuleiro e RPGs de mesa. Um conjunto de parâmetros para customização destes ambientes é proposto e, ao final deste trabalho, são apresentados e discutidos os resultados obtidos com a aplicação.

Palavras-chave: Geração de ambientes virtuais, jogos de mesa, Geração Procedural de Conteúdo, Diagrama de Voronoi.

Abstract

The development of digital games involves several topics in the computing field. The visual and audio resources used in those games are the main responsible for immersion. However, in the case of tabletop games, the means to produce immersion are much more limited. Therefore, this work proposes to use procedural content generation, a classic feature in digital games, in non-digital games, aiming to achieve a higher level of immersion. We selected and compared procedural content generation algorithms that adequately fit the problem, pointing out which are the advantages and problems of each one. The application is focused on the generation of virtual environments for board games and tabletop RPGs. A set of parameters is proposed for customizing these environments. Finally, we present and discuss the obtained results.

Keywords: Virtual environment generation, tabletop games, Procedural Content Generation, Voronoi Diagram.

Conteúdo

Lista de Figuras

Lista de Abreviações

DCC	Departamento de Ciência da Computação
UFJF	Universidade Federal de Juiz de Fora
RPG	Role-playing Game
GPU	Graphics Processing Unit
GPC	Geração Procedural de Conteúdo
GM	Game Master

1 Introdução

A computação gráfica costuma ser uma das primeiras áreas que surge associada com jogos na visão do usuário, dado o apelo visual que esta proporciona. Entretanto, apesar de sua real importância e de sua evolução continuarem sendo relevantes para o mercado de jogos, existem outros alicerces para o sucesso de um jogo. O desenvolvimento de estruturas de dados eficientes, a escolha e implementação adequada de técnicas de inteligência artificial, e um bom trabalho de *design* são todos muito importantes nesse quesito. Dentre os elementos envolvidos no processo de *design*, podemos citar, por exemplo, a interface, a jogabilidade e a produção de conteúdo.

Dentro das diversas formas de se produzir conteúdo, esse trabalho irá se basear em uma forma automatizada de criação, chamada geração procedural de conteúdo (ou GPC). Alguns dos exemplos mais simples e diretos em que a GPC pode ser utilizada em jogos são a criação de níveis completos, como ocorre na franquia de jogos *Diablo* (Diablo, 1996), ou diferentes topografias em um ambiente virtual. Sua utilização pode trazer inúmeras vantagens, que variam do volume de conteúdo produzido à economia de armazenamento de dados em memória secundária. Tais vantagens e seus contrapontos são explicados na Seção 2.1.

Existem algumas propostas de programas projetados para serem extensões para jogos não-virtuais. Jogos de tabuleiro e RPGs de mesa (ou *tabletop role-playing games*) são grandes exemplos de entretenimento que podem se beneficiar amplamente desse tipo de aplicação. O *TARBoard* e o *Tankwar* (Lee et al., 2005; Nilsen et al., 2005), que mostraremos mais adiante, são exemplos que seguem a proposta de serem extensões para jogos. A ideia deste trabalho é também seguir essa proposta, aplicando-a sobre as formas de entretenimento citadas, com ênfase no RPG de mesa.

Sendo a boa capacidade de imersão um dos aspectos mais desejados para um jogo, os jogos virtuais se destacam nesse aspecto, por poderem usar recursos visuais e auditivos que os não-virtuais não dispõem. Dessa forma, se pudermos incrementar jogos de mesa com alguns desses recursos, podemos atingir um nível maior de imersão. Os jogos de mesa

algumas vezes utilizam mapas de terreno para a ambientação dos jogadores. Por serem produzidos de forma manual, são limitados. Dessa forma, a utilização de um ambiente virtual gerado de forma procedural, composto de um terreno e objetos dispostos sobre ele torna-se útil por ser possível criar facilmente diversos ambientes diferentes.

1.1 Problema

A geração procedural em jogos digitais, de uma forma geral, é bastante explorada. O propósito deste trabalho é a utilização do mesmo princípio em jogos não-digitais, como o RPG de mesa, podendo o jogador descrever um ambiente presente em seu jogo através de parâmetros, para que ele seja gerado visualmente de modo procedural em um ambiente virtual. Assim, pretende-se responder à seguinte questão: **o uso de geração procedural pode aumentar a imersão em jogos de mesa?** É preciso levar também em consideração que o produto final deverá ser eficiente, simples de usar e que o resultado final dos mapas seja bem ambientado, para ser apreciado pelos usuários.

1.2 Objetivos

1.2.1 Objetivo Geral

Comparar métodos de geração procedural que sejam capazes de gerar um terreno e distribuir conteúdo sobre ele. A meta final é oferecer um meio de proporcionar melhor imersão para jogadores no que diz respeito à ambientação dos jogos de RPG de mesa e tabuleiro, além de definir através desse estudo quais as melhores formas de implementar a GPC para a aplicação escolhida.

1.2.2 Objetivos Específicos

- Explorar técnicas de geração procedural de conteúdo presentes na literatura;
- Implementar um protótipo para a geração de ambientes virtuais a partir de um conjunto de parâmetros que possam ser mapeados em um algoritmo de distribuição de conteúdo sobre um terreno previamente gerado;

- Discutir os resultados obtidos com variações na combinação dos parâmetros propostos.

2 Fundamentação Teórica

O RPG é um gênero de jogos de estratégia no qual os jogadores assumem um ou mais papéis de personagens em uma narrativa, podendo o jogo ser eletrônico ou não. No caso do eletrônico, há diversas outras características que podem interferir na experiência do usuário, como a jogabilidade ou a parte gráfica. Porém, no RPG de mesa, o principal fator é a narrativa em si, com todos os pormenores que a constituem, sendo a grande maioria dos agentes de imersão criados na imaginação de cada jogador. Geralmente são jogados de forma cooperativa, com os jogadores fazendo parte de um mundo. Suas decisões influenciam e são influenciadas pelo ambiente à sua volta.

Já a geração procedural consiste basicamente na utilização de um processo ou função preestabelecido para a geração de conteúdo com intervenção pequena ou indireta do usuário, ao invés de criá-lo e adicioná-lo de forma manual (Shaker et al., 2016). Repare que isso pode ocorrer em tempo de execução, de forma automatizada. Em alguns casos, a criação do conteúdo é realizada em uma etapa de carregamento, toda vez que o jogo é executado, mas antes do jogador assumir o controle. Um exemplo são jogos que geram o mapa de um estágio e o posicionamento dos seus objetos enquanto mostram uma tela de carregamento da fase. Em outras situações, é necessário que a criação seja feita em tempo real, vide, por exemplo, a geração de objetos no jogo *No Man's Sky*, já citado, ou jogos de "progressão infinita", em que o estágio é continuamente expandido, de forma transparente, enquanto o jogador o percorre.

2.1 Vantagens do uso da geração procedural de conteúdo

Algumas das vantagens da criação de conteúdo dessa forma são: economia de espaço de armazenamento; volume de conteúdo gerado; e conteúdo dinâmico. Tais vantagens estão explicadas a seguir.

2.1.1 Economia de espaço de armazenamento

Com a utilização da GPC, a necessidade de armazenarmos o conteúdo é reduzida, uma vez que a função de geração pode criá-lo novamente a qualquer momento. Isso possibilita uma economia de espaço em disco bastante significativa, fator imprescindível para a viabilidade de jogos com mundos de grandes proporções, como é o caso do *No Man's Sky* com seus 18 quintilhões de planetas. Se a geração é feita em tempo real, pode-se também economizar espaço na memória principal, embora em menor escala. A contrapartida de toda essa economia de espaço é, naturalmente, um custo maior de processamento.

2.1.2 Volume de conteúdo gerado

Outro bom motivo para se recorrer à GPC é a necessidade de disponibilizar para o jogador um volume de conteúdo suficientemente grande para que a criação manual seja considerada trabalhosa demais ou até mesmo impraticável. Dessa forma, há uma grande economia de trabalho e tempo humanos, tendo como consequência a redução de custos no desenvolvimento do jogo. Geralmente, quando uma abordagem procedural é utilizada com essa finalidade, a geração do conteúdo é feita em uma etapa de carregamento dos dados, antes da execução. Caso contrário, o algoritmo em questão deverá ser extremamente eficiente para conseguir produzir conteúdo em larga escala em tempo real.

2.1.3 Conteúdo dinâmico

O fato do conteúdo ser produzido algoritmicamente facilita a introdução de algum elemento de aleatoriedade. Assim, a experiência pode tornar-se diferente a cada execução do jogo (visto que o conteúdo muda), e se contorna a inviabilidade de se criar um ambiente suficientemente grande e diversificado que possa ser considerado limitado apenas pelos recursos disponíveis para a plataforma de destino. Normalmente, esses elementos são controlados por configurações específicas para a situação, pois algo criado de forma completamente aleatória dificilmente fará sentido dentro do contexto aplicado.

Vale lembrar que na computação em geral se utiliza geradores de números pseudo-aleatórios. Isso significa que, com as mesmas entradas, uma função com o objetivo de

retornar um valor randômico traria sempre os mesmos valores. Porém, essas funções recebem um valor de entrada, a chamada semente de geração, que influencia no retorno. O mais comum é que essas funções já venham implementadas diretamente na linguagem de programação de forma a utilizar uma semente diferente para cada uso, gerando valores praticamente aleatórios. Sendo assim, quando o termo "aleatório" for utilizado neste trabalho, é como referência à essa forma de geração. Em certos casos, salvar uma semente utilizada pode ser útil, pois dessa forma é possível reproduzir o resultado anterior.

A vantagem que mais foi explorada neste trabalho é a de dinamismo de conteúdo, visto que a proposta é criar ambientes distintos sempre que o aplicativo for executado. Para que o conteúdo a ser gerado seja condizente com a necessidade do usuário, foram utilizados alguns parâmetros de controle. Embora a geração procedural possa ser aplicada nos jogos de diversas maneiras, este trabalho se restringe a duas finalidades específicas: a geração do terreno e a distribuição do conteúdo sobre ele. Nas seções a seguir, serão apresentadas as principais técnicas utilizadas neste trabalho.

2.2 Geração do terreno

O objetivo nesse momento é utilizar a geração procedural para obter a representação de um terreno. Para isso, explicaremos o conceito de mapa de altura, que é vital nessa etapa, e diferentes formas de se obtê-lo.

2.2.1 Mapa de altura

Uma estrutura de dados de um terreno precisa armazenar suas características relevantes, como altura e tipo de solo. Geralmente, as informações são guardadas para cara coordenada (x, y) dentro do plano do terreno. Para o contexto deste trabalho, o conceito mais promissor é o de *Digital Terrain Model* (ou DTM) (Li et al, 2004). Ele nos permite generalizar o modelo de terreno como sendo um conjunto específico de funções que mapeiam uma distribuição espacial de várias informações sobre o terreno. Dessa forma, podemos definir:

$$A_p = f(x_p, y_p),$$

onde A_p representa o valor de um certo atributo A no ponto p ao aplicar a função f às coordenadas (x, y) do ponto p.

Nota-se que a definição é favorável à forma de operação da geração procedural, visto que nos dá a possibilidade de obter valores para os atributos de interesse através de uma função matemática. Elementos de aletoriedade também se tornam mais simples de serem adicionados dessa forma.

Um dos principais conceitos utilizados neste trabalho é o de **mapa de altura**. Basicamente, trata-se de uma matriz de duas dimensões, cujos índices representam as coordenadas (x, y), e o valor atribuído representa a altura do terreno nas respectivas coordenadas.

Porém, essa não é a única maneira de se representar terrenos. Uma outra abordagem bastante conhecida é a utilização de voxels. Nessa representação, ao invés de termos como base a matriz bidimensional de alturas, o terreno é composto por voxels. Um bom exemplo dessa forma de representação é o jogo Minecraft (2011), apesar de seus voxels terem um tamanho bastante exagerado se comparado ao que se costuma utilizar. Uma das vantagens deste tipo de representação é a possibilidade de criação de terrenos não completamente convexos, sendo viável a concepção de cavernas, por exemplo, o que não é possível com mapas de altura.

2.2.2 Valores Aleatórios

Uma primeira tentativa ingênua seria gerar valores de altura completamente aleatórios para cada coordenada (x, y). Porém, o resultado obtido dessa forma é ruim, pois não leva em consideração a altura de locais próximos, ocasionando grandes diferenças de altura entre coordenadas próximas. A Figura 2.1 ilustra este cenário. Ainda assim é interessante comentar sobre essa forma, pois os métodos que geram resultados melhores também se utilizam de alturas obtidas de forma aleatória, mas fazem isso de forma mais utilizável, tratando os resultados. Mostraremos adiante como.

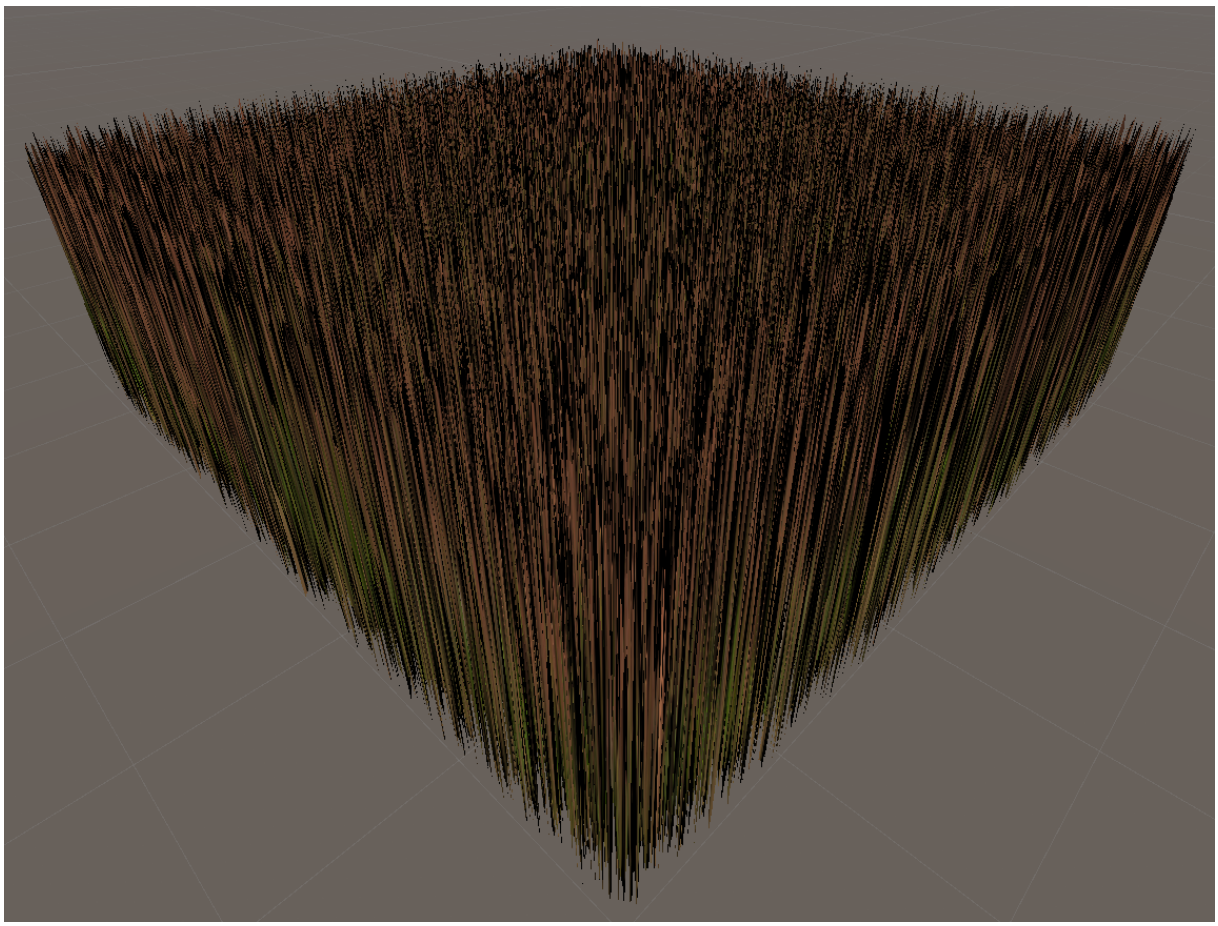


Figura 2.1: Terreno gerado com valores completamente aleatórios

2.2.3 Interpolação

A ideia dos métodos de interpolação é sortear valores apenas para alguns dos pontos do terreno e calcular as alturas intermediárias entre esses pontos, de modo que não haja variações de altura discrepantes entre um ponto e outro. A partir de agora, serão apresentados métodos que geram terrenos contínuos, ou seja, a altura de um ponto é influenciada pela altura dos pontos próximos, evitando a introdução de descontinuidades no terreno.

Bilinear

Uma interpolação linear entre dois valores é basicamente uma média ponderada entre eles, na qual o peso é o inverso da distância normalizada em que o ponto se encontra para cada coordenada (Shaker et al., 2016). O termo bilinear implica que a interpolação linear será realizada nas duas direções, primeiro em x e depois em y (ou vice-versa).

Podemos definir a interpolação em x como

$$I_x(x, y) = \left(1 - \frac{x - x_e}{x_d - x_e}\right) * A(x_e, y_e) + \left(\frac{x - x_e}{x_d - x_e}\right) * A(x_d, y_d),$$

sendo (x_e, y_e) as coordenadas do ponto mais à esquerda a ser interpolado, (x_d, y_d) as coordenadas do ponto mais à direita e A o valor de altura atribuído ao ponto.

Agora, chamando de (x_s, y_s) e (x_i, y_i) as coordenadas dos dois pontos superiores e inferiores em relação ao ponto que desejamos obter a altura, respectivamente, podemos obter suas interpolações em x. Vamos chamá-las de I_s e I_i. Assim, podemos finalmente definir a altura em um ponto (x, y) como:

$$A_c(x, y) = (1 - \frac{y - y_i}{y_s - y_i}) * I_i + (\frac{y - y_i}{y_s - y_i}) * I_s$$

Por exemplo, digamos que os valores nos pontos $(10, 10)$, $(10, 20)$, $(20, 10)$ e $(20, 20)$ foram obtidos de forma aleatória. Para se calcular o valor intermediário em $(12, 15)$, fazemos uma interpolação em x e obtemos os valores em

$$(12, 10) = 0,8 * (10, 10) + 0,2 * (10, 20)$$

e em

$$(12, 20) = 0,8 * (10, 20) + 0,2 * (20, 20),$$

e depois calculamos o valor final desejado fazendo a interpolação em y

$$(12, 15) = 0,5 * (12, 10) + 0,5 * (12, 20).$$

Bicúbica

Embora a interpolação anterior já apresente resultados melhores, é observado na Figura 2.3a que as transições entre as alturas ainda são muito bruscas. O método de interpolação bicúbica funciona da mesma forma que o bilinear, tendo como única diferença a função que define o peso de cada ponto na hora de interpolar. Como o objetivo dessa melhoria é obter resultados mais suaves em relação aos adquiridos com a

bilinear, é necessário utilizar uma função que se aproxime melhor de um relevo natural, apresentando um formato em S típico de encostas. Devido à sua simplicidade, uma função comumente utilizada é a definida por

$$s(x) = -2x^3 + 3x^2,$$

cujo gráfico da Figura 2.2 mostra sua avaliação para x entre 0 e 1 (Shaker et al., 2016).

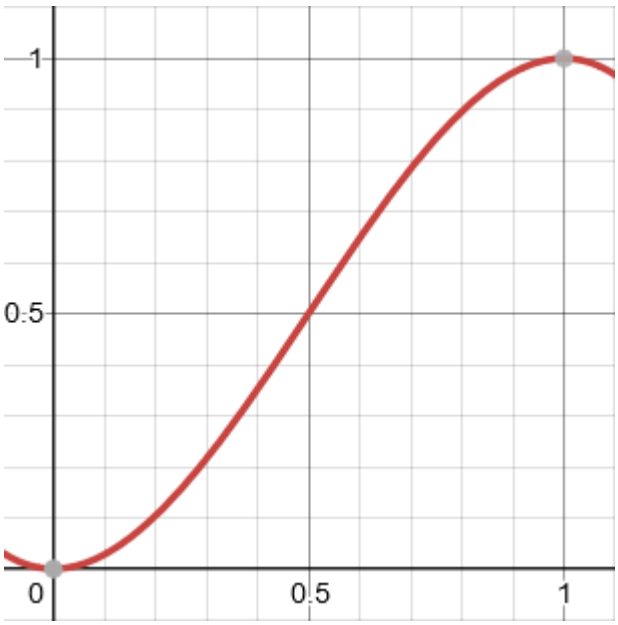
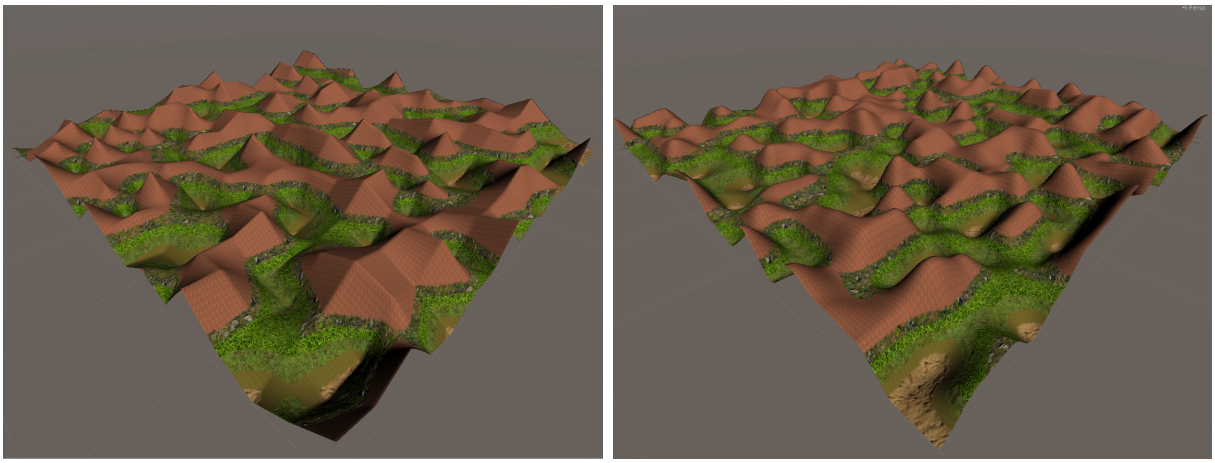


Figura 2.2: Gráfico da avaliação da função $s(x) = -2x^3 + 3x^2$, com x entre 0 e 1



(a) Terreno gerado com interpolação bilinear (b) Terreno gerado com interpolação bicúbica

Figura 2.3: Terrenos gerados utilizando os dois métodos de interpolação

2.2.4 *Diamond-Square*

O *Diamond-Square* é um algoritmo que gera um terreno utilizando o conceito de fractais, que de forma resumida é a repetição de padrões ou variações, mas em contextos e escalas cada vez menores. O algoritmo começa preenchendo os valores dos quatro cantos do terreno. Os valores podem ser aleatórios ou não. Depois disso, alterna-se entre os dois seguintes passos:

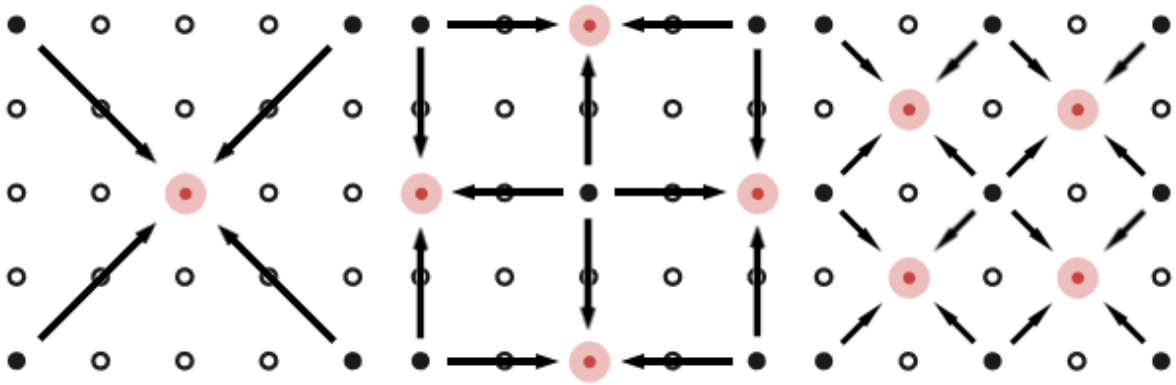
Passo *Diamond*

No centro do quadrado formado pelos quatro valores inicializados, aplica-se o valor calculado pela média aritmética desses quatro cantos e soma-se um valor aleatório, com variação entre $-\alpha$ e α, sendo α definido como parâmetro.

Passo *Square*

Os pontos médios entre os quatro cantos são obtidos, e seus valores são dados pelas médias entre os dois respectivos pontos que os definiram e o ponto central calculado no passo anterior, somando novamente um valor entre $-\alpha$ e α. Depois, α é dividido por dois.

Após a aplicação desses dois passos, note que quatro novos quadrados com os quatro cantos preenchidos se formarão, assim como no início. Os dois passos são executados recursivamente nos novos quadrados até que todo o mapa tenha sido preenchido com seus valores de altura. Repare que ao dividir α por dois, os valores absolutos aleatórios que estão sendo adicionados vão sendo reduzidos (menor escala) e aplicados em quadrados menores (menor contexto) (Shaker et al., 2016).



(a) Passo *Diamond* (b) Passo *Square* (c) Segundo passo *Diamond*

Figura 2.4: Funcionamento do algoritmo *Diamond-Square*

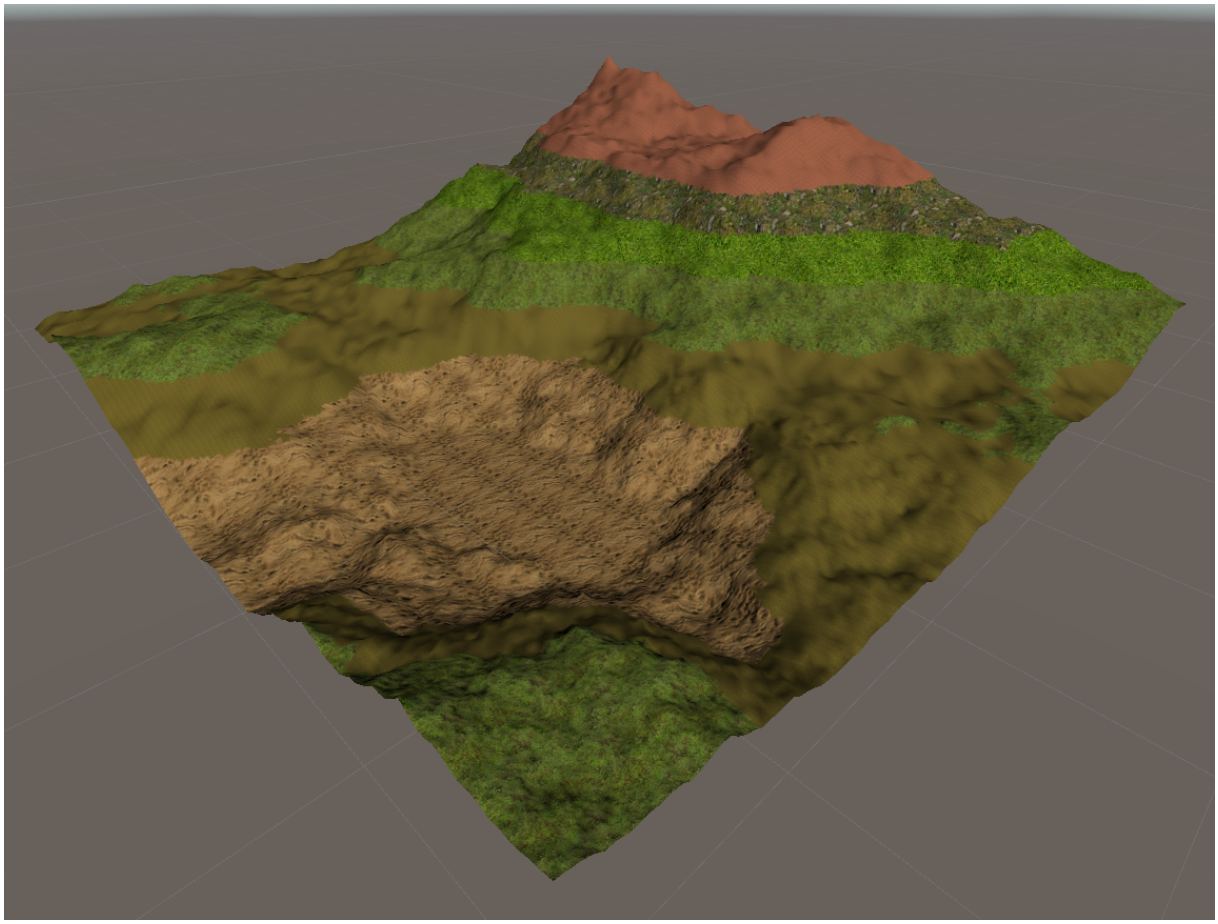


Figura 2.5: Terreno gerado usando o algoritmo *Diamond-Square*

2.2.5 Ruído de Perlin

Introduzido por Perlin (1985) e aprimorado por ele mesmo anos mais tarde (Perlin, 2002), o ruído de Perlin é uma função que utiliza vetores de gradientes pseudo-aleatórios para gerar um valor de influência (que no caso do nosso trabalho será a altura)

em cada ponto do plano. Como os valores são decorrentes de uma função pseudo-aleatória preestabelecida, o retorno será sempre igual se receber como parâmetro os mesmos valores de x e y do terreno. Por isso é interessante adicionarmos um deslocamento aleatório, pegando locais diferentes no plano de Perlin sempre que formos utilizar esse método.

Um conceito frequentemente associado a ruídos é o de frequência. Utilizando-se um multiplicador (a frequência) nos valores de x e y passados para a função de Perlin, é possível acelerar ou atrasar as transições entre os valores. Para se gerar um ruído coerente costuma-se misturar diferentes frequências, as chamadas **oitavas** (termo advindo de frequências sonoras na música). Os resultados de vários retornos da função do ruído de Perlin para cada ponto são então somados, a cada passo dobrando a frequência, mas diminuindo seu peso no valor final, conforme mostrado na Figura 2.6. Repare que apesar do ruído de Perlin não ser uma forma de geração fractal por si só, o método de soma de suas oitavas simula o comportamento de estratégias que seguem esse paradigma. O ruído de Perlin é utilizado em diversas outras aplicações além de terreno. Como exemplo, podemos citar geração de nuvens, texturas e animação. Aplicando a função de ruído em um terreno, obtemos resultados como o mostrado na Figura 2.7.

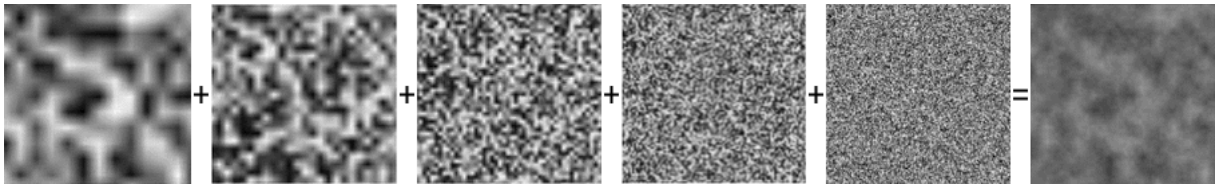


Figura 2.6: Soma de algumas oitavas do ruído de Perlin (Li et al., 2016)

2.3 Distribuição do conteúdo

Depois que o terreno já teve seu mapa de altura gerado e aplicado, precisamos começar a distribuição do conteúdo. Entretanto, essa etapa ainda engloba a modificação da textura no terreno, para refletir visualmente os caminhos e locais onde construções poderão ou não ser colocadas. Nas subseções a seguir, será apresentado o conceito de diagrama de Voronoi e os algoritmos usados para a sua geração. Estes conceitos serão utilizados como base para a aplicação de textura e a distribuição do conteúdo no protótipo

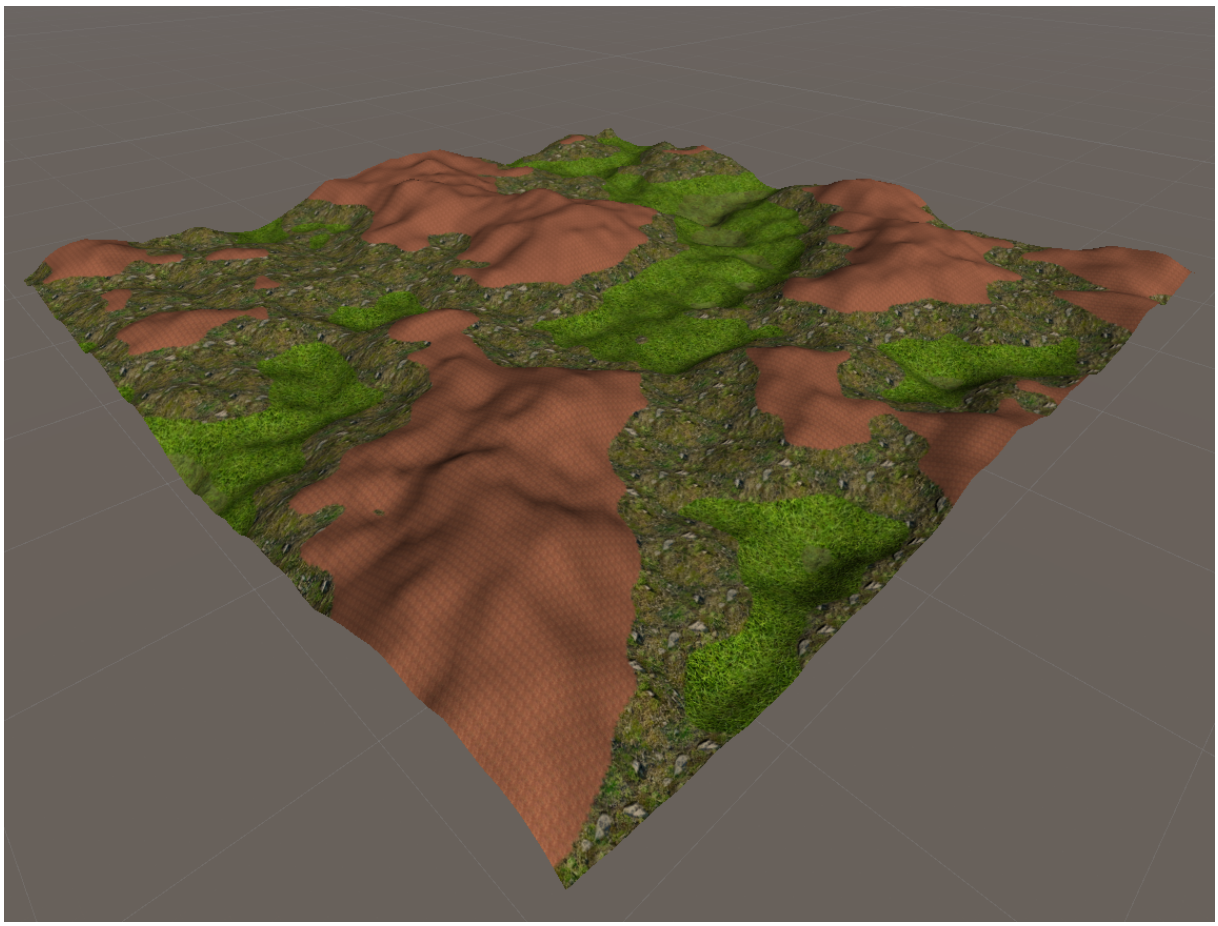


Figura 2.7: Terreno gerado com a soma de quatro oitavas do ruído de Perlin e frequência-base 3

que será apresentado no Capítulo 4.

2.3.1 Diagrama de Voronoi

O diagrama ou partição de Voronoi é uma forma de divisão de um espaço em regiões. Um conjunto de pontos é escolhido em um plano e cada um desses pontos recebe o nome de sítio. Delimita-se então uma região (chamada de célula) ao redor de cada sítio, na qual todos os pontos internos tem o referido sítio como o mais próximo, resultando em algo como na Figura 2.8a. Suas aplicações são bastante variadas, podendo ser usado, por exemplo, na criação de padrões de textura. Também é possível notar seu padrão espalhado na natureza, como nas escamas de répteis, nos pelos da girafa, em colmeias, terrenos ressecados, agrupamento de células, entre muitos outros (de Berg et al., 1997).

2.3.2 Algoritmo ingênuo de Voronoi

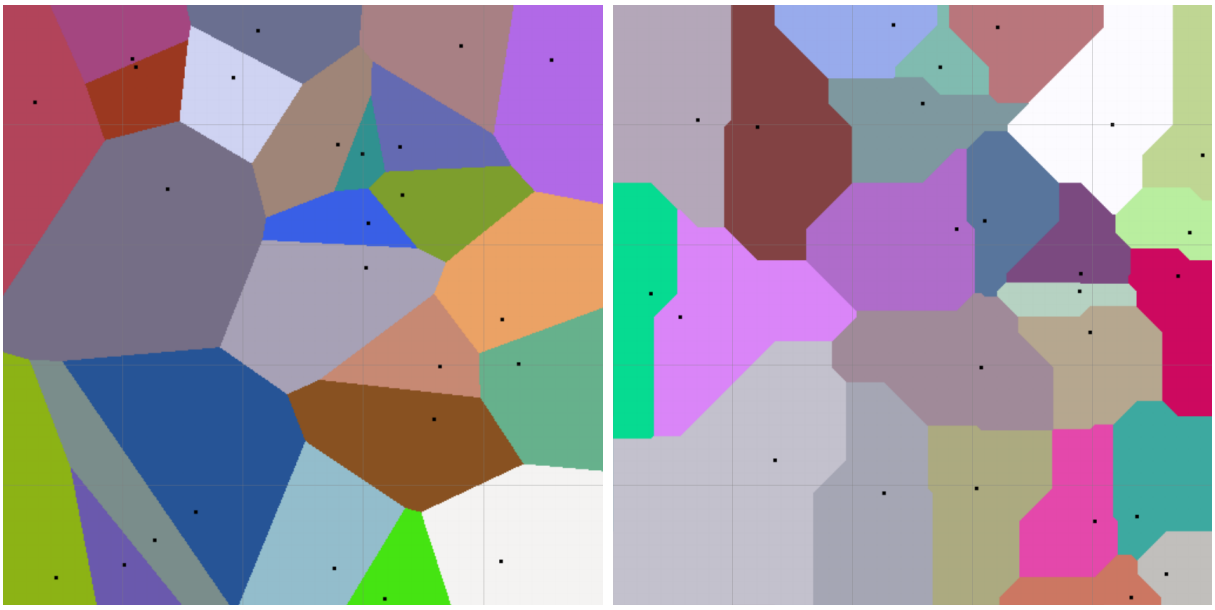
Nessa abordagem, primeiro escolhemos os sítios, o que geralmente é feito de forma aleatória. Depois disso, percorremos o terreno e, para cada coordenada (x, y), calculamos a distância para cada um dos sítios, adicionando o ponto na célula do sítio mais próximo. Apesar de ser bem simples sua compreensão e implementação, a eficiência desse algoritmo é extremamente ruim, e o aumento do número de sítios tem um impacto muito grande no tempo de execução.

Utilizando a distância de Manhattan

Embora a implementação ingênua do diagrama de Voronoi seja pouco eficiente, conseguimos alterá-la facilmente para obter resultados diferenciados. Um bom exemplo é a utilização da distância de Manhattan ao invés da distância euclidiana. Por definição, ela é a soma das diferenças entre as coordenadas x e y de dois pontos, ou seja:

$$d_M = |x_1 - x_2| + |y_1 - y_2|$$

O resultado é um mapeamento menos orgânico, com o ângulo das divisões entre os sítios sendo sempre um multiplo de 45 graus (Figura 2.8b).



(a) Exemplo de diagrama de Voronoi gerado utilizando 25 sítios e distância euclidiana

(b) Exemplo de diagrama de Voronoi gerado utilizando 25 sítios e distância de Manhattan

Figura 2.8: Diferença entre diagramas gerados com distância euclidiana e de Manhattan

2.3.3 Algoritmo de Fortune

A proposta de Fortune (1987) para a geração de um diagrama de Voronoi utiliza uma abordagem por *"sweep-line"*, ou linha de varredura. Essa linha passa pelo plano, de cima para baixo, adicionando os sítios sobre os quais ela passa, e gerando parábolas definidas pela posição do sítio (foco) e a posição atual da linha de varredura (diretriz). As interseções entre as parábolas, à medida que a linha de varredura se move, formam retas, que constituirão as arestas que dividem as células de cada sítio. Quando uma célula é completamente fechada por arestas, não há mais modificações em seus limites, e por isso sua parábola não precisa mais ser verificada por interseções com outras. As parábolas e as arestas que ainda precisam ser verificadas fazem parte de um conjunto que chamamos de *beachline* (Fortune, 1987).

Na prática, não realizamos uma varredura completa em y, e sim marcamos eventos que ocorrerão e que precisam ser tratados. Tais eventos precisam ser colocados em uma lista ordenada decrescentemente pelo y da linha de varredura, de forma a tratarmos sempre o evento com maior y, simulando de maneira precisa o que ocorreria em uma varredura real. Existem dois tipos diferentes de eventos que precisam ser tratados: eventos de sítio e eventos de círculo. A seguir descreveremos cada um deles.

Evento de sítio

Este evento ocorre quando a linha de varredura encontra um novo sítio. Sua parábola deve ser então adicionada à *beachline*, a parábola que já existia naquele ponto deve ser dividida em duas partes (uma parte fica posicionada antes e uma depois da nova parábola), e duas novas arestas são iniciadas no ponto de interseção, em direções opostas, entre as parábolas, conforme a Figura 2.9. Note que no momento em que vamos adicionar essa nova parábola, as coordenadas y do foco e da diretriz são as mesmas. Isso faz com que tenhamos uma linha reta vertical, sendo fácil identificarmos a posição em que essa nova parábola deve ser adicionada, e qual parábola ela intersecta inicialmente. Conferimos então se as duas novas arestas irão em algum momento se encontrar com as arestas vizinhas. Caso essa interseção seja prevista, deve-se adicionar à lista de eventos um novo evento de círculo para cada interseção possível.

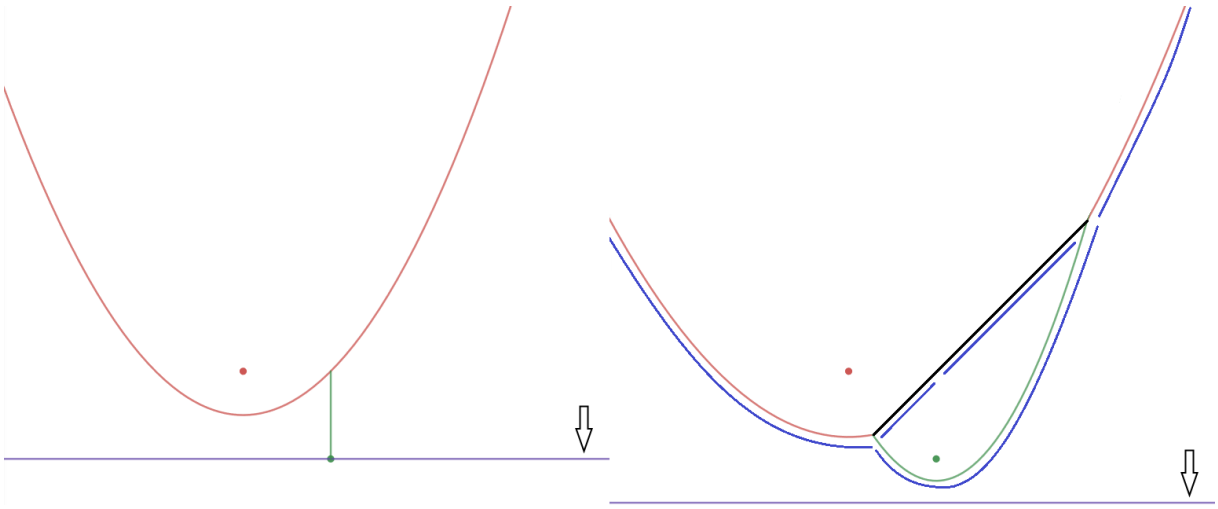


Figura 2.9: Evento de sítio. À esquerda, o momento em que um evento de sítio, em verde, é acionado. A parábola em vermelho foi intersectada, enquanto a verde é a nova a ser adicionada à *beachline*. À direita, a representação dos elementos que devem ser adicionados à *beachline*. O arco vermelho é dividido em duas partes e, entre elas, adicionamos a aresta que cresce para a esquerda, o arco novo e a aresta que cresce para a direita, nessa ordem.

Evento de círculo

O evento de círculo ocorre quando duas arestas que ainda estão na *beachline* se encontram, marcando que ambas foram fechadas e não irão mais crescer. Deve-se então retirá-las da *beachline*, junto com a parábola que existia entre elas, e colocar uma nova aresta começando no ponto de interseção. As arestas retiradas entram no conjunto de arestas da solução como arestas completas, com ponto de início e de fim. Deve-se então verificar se a nova aresta colocada na *beachline* irá em algum momento intersectar as arestas vizinhas. Em caso positivo, um novo evento de círculo deve ser adicionado. O nome círculo vem do fato de que o ponto de interseção entre essas três arestas (as duas fechadas e a nova que foi iniciada) é o circuncentro que passa pelos três sítios que essas arestas separam.

Depois que a lista de eventos é totalmente percorrida, é necessário fechar as arestas restantes na *beachline*. O procedimento de fechamento dessas arestas varia de acordo com a aplicação. Após essa etapa, o resultado está pronto no formato de uma lista de arestas que dividem as células de Voronoi.

Enquanto o algoritmo ingênuo que foi mostrado possui complexidade quadrática de tempo em todos os casos, o algoritmo de Fortune apresenta complexidade de tempo $O(n \log n)$ no pior caso, o que o torna bem mais eficiente. No capítulo seguinte, será

mostrado como o diagrama de Voronoi foi aplicado à geração procedural de conteúdo neste trabalho.

No caso desse trabalho, as arestas restantes são fechadas em suas interseções com os limites do terreno.

2.3.4 *Forest Fire* (*Flood-Fill*)

O algortimo de *Flood-Fill* tem como objetivo visitar todos os pontos dentro de uma célula de uma matriz multidimensional, delimitada por algum valor escolhido. A implementação por vezes conhecida como *Forest Fire* tem esse mesmo objetivo, mas utiliza uma fila de eventos ao invés da recursão presente no *Flood-Fill*, com o objetivo de evitar problemas com a pilha de recursão (Torbert, 2016).

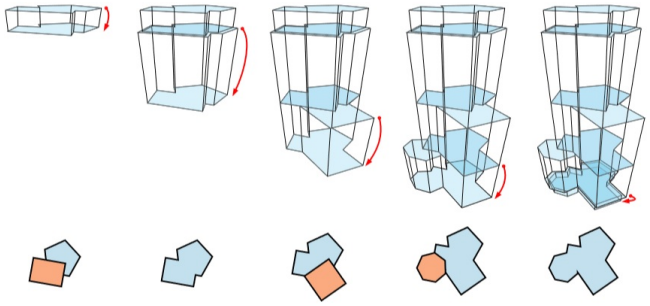
3 Revisão Bibliográfica

Neste capítulo alguns dos principais trabalhos relacionados ao tema do trabalho são analisados. A primeira seção trata de trabalhos que empregam geração procedural de conteúdo. Embora muitos destes trabalhos sejam aplicados na área de jogos, existem exceções. Além disso, na Seção 3.2, são também apresentados e analisados trabalhos que propoem extensões e melhorias para jogos não virtuais, porém sem necessariamente utilizar a geração procedural de conteúdo.

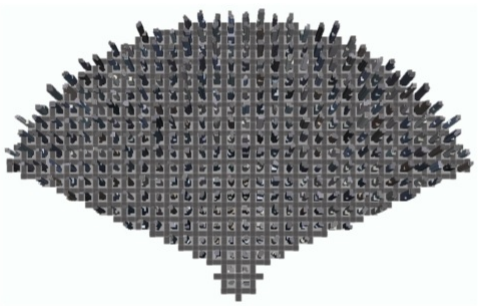
3.1 Geração procedural de conteúdo

Como exemplo de trabalhos desvinculados dos jogos, podemos citar Greuter et al. (2003) com o desenvolvimento de um gerador em tempo real de uma cidade "pseudo infinita" (denominação do próprio autor). Tal cidade pode ser usada em simuladores, animações, treinamento de algoritmos evolutivos ou jogos. Cidades virtuais também são excelentes candidatas para serem usadas em testes de técnicas em computação gráfica, como por exemplo no caso de descarte por oclusão. Em seu trabalho, os prédios também são gerados de forma procedural. Cada um deles possui um posicionamento no eixo cartesiano, representado por x e y, e sua função de geração é alimentada com esses valores para se criar o modelo visual de cada prédio: em cada parte deles, novas formas geométricas são adicionadas ao formato anterior já produzido do topo até a base do prédio, como mostra a Figura 3.1a.

Outro trabalho foi o publicado por Groenewegen et al. (2009), que desenvolveu uma forma procedural de criar o leiaute de cidades. Usando modelos de uso das terras urbanas, o algoritmo recebe informações fornecidas pelo usuário (tamanho da cidade, continente de localização, contexto histórico, número de rodovias) e gera um modelo separando os tipos de distrito da cidade, como por exemplo distritos industriais, comerciais, residenciais, centro histórico, etc. O método proposto pelos autores não necessita de informações externas, além de ser computacionalmente eficiente e simples de usar. Dessa



(a) Geração dos prédios



(b) Visao superior

(c) Resultado final

Figura 3.1: Geração procedural de cidade “pseudo infinita” por Greuter et al. (2003)

forma, seu trabalho pode ser usado por pessoas sem conhecimento prévio em urbanismo e suas aplicações.

Diversos outros trabalhos podem ser citados nesse contexto. Galin et al. (2010) apresentou uma forma de gerar estradas, enquanto Emilien et al. (2012) focou na geração de vilas, ambos gerados de forma procedural e em terrenos arbitrários e complexos. Mustafa et al. (2018) realizaram um trabalho que cria, de forma procedural, o leiaute de uma cidade e, a partir do modelo criado, realiza modificações, que podem ser mais ou menos agressivas através de parâmetros, com o intuito de diminuir a gravidade de possíveis alagamentos na região. O trabalho de Summerville et al. (2018), por sua vez, aborda a geração procedural de conteúdo através da utilização de técnicas de aprendizado de máquina.

3.1.1 Geração procedural em jogos

A geração automática de conteúdo para jogos começou no início dos anos 1980. Na época, o maior problema era a limitação de hardware, principalmente no que diz respeito a armazenamento (Cardamone et al., 2011; Freiknecht et al., 2017). Hoje em

dia os dispositivos de armazenamento têm uma capacidade muito maior, mas a demanda por conteúdo também aumentou consideravelmente. Há o interesse em se gerar ambientes realistas e detalhados, o que implica em maiores custos e tempo dedicado na criação do conteúdo. A geração procedural de conteúdo, além de tentar resolver ou pelo menos amenizar esses problemas, faz com que o jogo ofereça experiências diferentes em cada vez que é jogado, visto que, embora sua mecânica de jogo não mude, seu conteúdo é potencialmente diferente. Dessa forma, há um grande número de trabalhos voltados para este tema.

Um exemplo conhecido e recente de jogo que utiliza a GPC em boa parte do seu conteúdo é o No Man's Sky (2016), desenvolvido pelo estúdio independente Hello Games. A proposta do jogo desde o início de sua criação é ter uma grande quantidade de conteúdo, por isso seus criadores recorreram a métodos procedurais. Seu universo é populado por 18 quintilhões (10^{18}) de planetas gerados de forma procedural, com tamanhos, cores e biomas diferentes. Além dos planetas, também são gerados proceduralmente plantas, animais, naves, entre outros. Na Figura 3.2 vemos alguns dos animais gerados no jogo.

Figura 3.2: Exemplo de animais gerados de forma procedural para o jogo No Man's Sky (**?**)

Cardamone et al. (2011) foram responsáveis pelo desenvolvimento de uma ferramenta que cria pistas para jogos de corrida, utilizando um algoritmo de geração procedural baseado em busca. Cada pista gerada é então classificada de acordo com a sua adequação

pelo usuário, e então as soluções são melhoradas de forma evolutiva e interativa, ou seja, usuários realizam avaliações dos resultados, que por sua vez realimentam o sistema para gerar resultados mais próximos das boas avaliações.

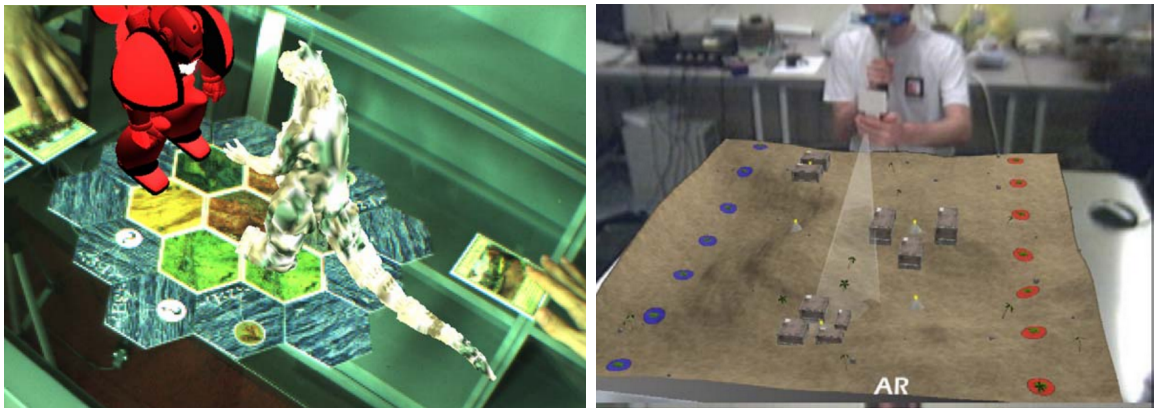
Frank et al. (2017) propôs um método para realizar a geração procedural de cidades através do ruído de Perlin, método que também foi explorado neste trabalho, mas para a criação do relevo do terreno. Os autores também trazem a discussão da viabilidade da solução que criaram, realizando uma pesquisa com 27 pessoas, na qual fizeram perguntas sobre a percepção de naturalidade das cidades apresentadas.

O trabalho de Beyer (2017), por sua vez, utiliza a geração procedural aplicada em RPGs para construir uma linha narrativa para o jogo, definindo locais, encontros, objetivos, etc. A partir disso, os locais são criados levando em consideração todas as características que foram definidas para eles na etapa de geração da linha narrativa. Por exemplo, se foi definido que local x terá conexão com o local y, na geração de ambos os locais haverá uma saída de um para o outro. Se foi definido que haverá um encontro no local z, o personagem com quem o encontro vai ocorrer é colocado nesse local.

3.2 Extensões para jogos não virtuais

A computação, como mencionado anteriormente, já começou há alguns anos a dar suporte a jogos não virtuais, criando facilitações e extensões para os mesmos. Em 2005, foi publicado um estudo sobre como melhorar jogos de tabuleiro com o uso de eletrônica, no qual são levantadas as principais tarefas que precisam ser executadas por alguma implementação de melhoria, como por exemplo “pegar objeto de uma posição” ou “ordenar jogador a executar tarefa” (Eriksson et al., 2005).

Lee et al. (2005) propuseram o que chamaram de *TARBoard*, um sistema para adicionar realidade aumentada em jogos de mesa. Enquanto esse trabalho tem como foco jogos de carta (ou *card games*), o apresentado por Nilsen et al. (2005), de nome *Tankwar*, foca em jogos de tabuleiro de guerra. A Figura 3.3 mostra o resultado de ambos os projetos. Também relacionado a realidade aumentada, o artigo de Tan et al. (2015) apresenta uma estruturação para jogos de tabuleiro que pretendem aplicá-la usando RFID.



(a) TARBoard (Lee et al., 2005) (b) Tankwar (Nilsen et al., 2005)

Figura 3.3: Resultado dos projetos de realidade aumentada apresentados

O *Undercurrents* (Bergström et al., 2010) compartilha do mesmo foco de aplicação que esse trabalho, o RPG de mesa. Porém, a proposta é voltada para a melhoria da jogabilidade, servindo de uma ferramenta para troca de mensagens entre os jogadores e o GM (*Game master*, espécie de mediador e juiz do jogo), suporte e gerência de documentação, controlador de mídia, anotações, e várias outras funcionalidades com esse propósito. O software ainda implementa diversos recursos para facilitar e agilizar esses procedimentos. Outro software que tem o objetivo de complementar o RPG de mesa é o *Companion* (Stickert et al., 2018), que apresenta uma ferramenta para a unificação das mídias auditiva e visual, ofececendo a utilização de imagens de mapas junto com trilha sonora e sons ambientes. As configurações e a disposição dos objetos podem ser salvas, permitindo ao GM que prepare tudo que será usado no jogo.

Além de todos os trabalhos citados, existem algumas pesquisas que procuram agrupar o que já foi explorado na literatura sobre geração procedural. Algumas delas são as apresentadas por Smith (2015) e Freiknecht et al. (2017). Enquanto a primeira tem como tema uma abordagem histórica de como a GPC foi aplicada em jogos analógicos, a segunda apresenta uma abordagem mais técnica, mostrando os métodos utilizados para geração procedural de vários tipos de conteúdo, como terreno, vegetação, estradas, construções, histórias, entre outros.

Diferentemente dos trabalhos mostrados, a proposta do presente trabalho é a criação de ambientes genéricos e completos, a partir de algumas características iniciais estabelecidas via parâmetros, para que sejam usados de diferentes formas durante o jogo.

A geração do ambiente não depende de uma narrativa, como acontece no trabalho de Beyer (2017), por exemplo, mas ela pode ser criada posteriormente, de acordo com o resultado obtido.

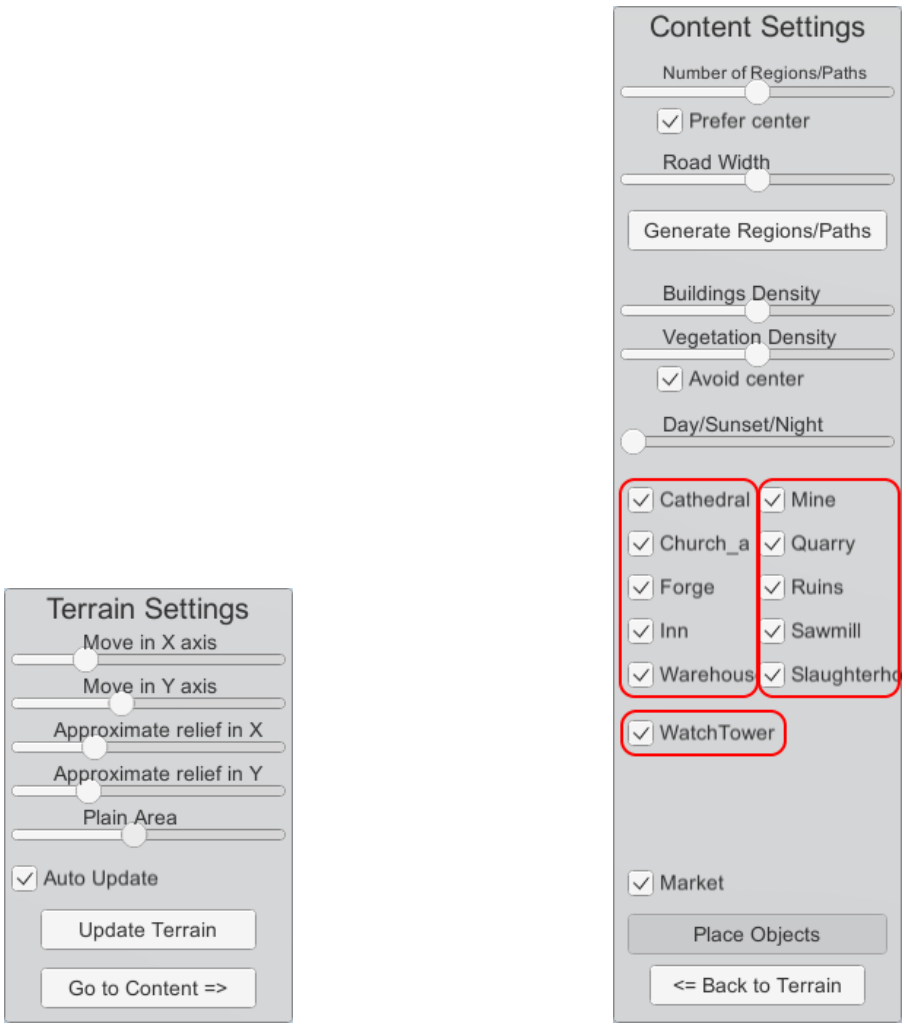
4 Desenvolvimento

Neste capítulo, apresentaremos como os conceitos apresentados no Capítulo 2 foram aplicados para a construção de um protótipo que possa ser usado para gerar ambientes virtuais para jogadores de RPG. Serão detalhados a forma de implementação dos recursos utilizados e cada um dos parâmetros que foram confeccionados para controlar o processo de geração. A ferramenta foi desenvolvida no motor de jogos Unity versão 2019.3.0a3.

4.1 Visão geral do protótipo

O protótipo apresenta um terreno que tem seu mapa de altura alterado utilizando o ruído de Perlin. O usuário pode visualizar as alterações em tempo real nessa etapa, na medida em que altera as opções da interface. Já no momento em que os caminhos são definidos e os objetos alocados utilizando um diagrama de Voronoi, é necessário alterar as opções de interface e depois acionar a geração novamente caso o usuário queira realizar mudanças, já que um novo diagrama precisa ser criado e todos os objetos alocados novamente.

A interface da aplicação ficou dividida em duas partes, conforme mostrado na Figura 4.1, e reflete os parâmetros que foram selecionados para compor a aplicação, mostrados mais adiante nesse mesmo capítulo. Os itens circulados em vermelho na figura são para indicar que esses itens são colocados na interface de forma dinâmica, de acordo com os objetos que compõem cada um dos tipos de espalhamento. Isso significa que para mudar a classificação de qualquer um dos objetos ou adicionar novos, basta colocá-los nas devidas pastas dentro da Unity. Automaticamente a modificação será visível na interface, e marcar ou desmarcar a opção que aparecerá com o nome do objeto em questão definirá se ele é colocado na cena ou não.



(a) Configurações do terreno

(b) Configurações do conteúdo, incluindo a geração das regiões e o espalhamento de objetos

Figura 4.1: Opções da inteface de usuário

4.2 Terreno: ruído de Perlin

Para representar o terreno, manteremos duas matrizes, uma de textura, que será tratada na Seção 4.3, e a outra de altura, que trataremos a seguir. Na Unity existe um tipo de objeto específico para terrenos. Preenchendo matrizes de altura e textura, é possível carregar as informações diretamente no terreno, facilitando o processo. Para a matriz de altura, precisamos armazenar os valores normalizados, ou seja, entre 0 e 1 (com o valor 0 representando a altura mínima e 1 a máxima).

Apesar de todos os métodos de geração de terreno citados no capítulo anterior terem sido implementados, optamos pela utilização do ruído de Perlin para compor a ferramenta final. O método *Diamond-Square* também é capaz de gerar resultados interessantes, mas a adoção do ruído de Perlin facilita a parametrização adotada neste trabalho.

4.2.1 Implementação

Considere uma função *PerlinNoise* que retorna o valor do ruído de Perlin em determinada coordenada (x, y). A partir desta função, foi implementado o seguinte algoritmo:

Algoritmo 1: Gerando matriz com ruído de Perlin para se aplicar ao terreno

Resultado: Matriz com valores entre 0 e 1

Inicializar uma matriz altura[largura, profundidade];

Definir o fator de redistribuição r;

for $y \leftarrow 0$ **to** profundidade **do**

  for $x \leftarrow 0$ **to** largura **do**

    $val = 0.533 * \texttt{PerlinNoise}(x \textit{ normalizado, } y \textit{ normalizado})$;

    $val+ = 0.266 * \texttt{PerlinNoise}(2\textit{*(x normalizado), } 2\textit{*(y normalizado)})$;

    $val+ = 0.133 * \texttt{PerlinNoise}(4\textit{*(x normalizado), } 4\textit{*(y normalizado)})$;

    $val+ = 0.066 * \texttt{PerlinNoise}(8\textit{*(x normalizado), } 8\textit{*(y normalizado)})$;

    altura [y,x] $= val^r$;

  end for

end for

Retornar altura

Vale notar que foram utilizadas quatro oitavas, e por isso percebe-se que o valor do ruído de Perlin foi calculado quatro vezes, multiplicando a frequência e dividindo o peso por dois, a cada passo. A utilização do fator de redistribuição será explicada mais à frente.

4.2.2 Parâmetros

Mover no eixo x e Mover no eixo y

Conforme já exposto anteriormente na Subseção 2.2.5, é útil que, ao utilizar o ruído de Perlin, se adicione um valor aleatório às informações passadas para a função de geração, de forma a obtermos terrenos diferentes a cada execução. Portanto, esses dois parâmetros terão essa finalidade: são inicializados com um valor aleatório e somados ao

valor passado para a função de ruído. Ambos os parâmetros podem ser alterados para mover o local inicial de aplicação da função de ruído no plano em tempo de execução, tendo o efeito visual de mover o padrão de relevo em cada eixo.

Aproximar relevo em x e Aproximar relevo em y

O algoritmo apresentado utiliza como frequência-base o valor 1. Porém, é possível alterar essa base utilizando um multiplicador no valor passado para a função de ruído. A situação é parecida com o que é feito com as oitavas, mas nesse caso é a frequência-base que é alterada (consequentemente afetando cada uma das oitavas). Esses dois parâmetros multiplicadores começam com valores aleatórios e podem ser alterados. Visualmente, quanto maior o valor desses parâmetros, mais próximas as regiões de alto relevo ficam, cada um controlando esse fator em um dos eixos.

Área plana

Pelo algoritmo apresentado, nota-se que o valor adquirido pelo ruído de Perlin não é diretamente colocado na matriz de alturas. Antes disso, ele é elevado a um fator de redistribuição. O efeito que esse processo provoca é o de reduzir ainda mais o valor da altura quanto mais próximo de zero ele já for inicialmente (lembre-se que estamos lidando com valores entre 0 e 1). Foi criado então o parâmetro **Área plana**, que controla esse fator, com a intenção de gerar mais locais de área plana, propícios para se alocar as contruções e outros objetos.

4.3 Regiões/Caminhos: Voronoi

Depois da geração do mapa de altura ser completada, é necessário definir regiões e caminhos antes de se começar a posicionar objetos no ambiente. Para tanto, foi utilizado o diagrama de Voronoi, apresentado na Seção 2.3.1, na tentativa de criar caminhos aleatórios de aparência natural. As arestas resultantes do algoritmo de Fortune (escolhido por sua eficiência) são utilizadas para a representação dos caminhos, preenchendo os valores da matriz de textura com o resultado. O valor dentro da matriz deve ser um vetor de valores também normalizados, representando o peso de cada textura preestabelecida no terreno.

No caso desse trabalho, os caminhos são apresentados por uma textura de terra, que se mistura gradativamente com a textura de grama nos locais onde não há caminhos.

4.3.1 Estrutura

Para a implementação do algoritmo de Fortune, algumas importantes estruturas de dados precisaram ser contruídas para manter sua eficiência. A seguir, são descritas cada uma delas.

Evento

O algoritmo de Fortune se baseia em uma fila de prioridade de eventos para agir. O evento precisa de algumas informações para que possamos identificar como proceder.

- Tipo de evento: Se é evento de sítio ou evento de círculo;
- Coordenadas (x, y) do evento: No caso de evento de sítio, são as coordenadas do próprio sítio, e no evento de círculo são as coordenadas onde a interseção entre as duas arestas vai ocorrer;
- y da varredura: Coordenada y em que a linha de varredura estará no momento em que o evento for ocorrer. No caso de evento de sítio, é sua própria coordenada y, mas no caso de evento de círculo, deve-se calcular onde a linha de varredura deve estar para que a interseção ocorra. Esse é o atributo que vai guiar a ordenação da lista;
- Arestas em crescimento: Relevante apenas para evento de círculo, diz quais arestas vão se encontrar quando o evento ocorrer.

Aresta finalizada

A solução apresentada pelo algoritmo é uma lista dessa estrutura, representando arestas que já foram retiradas da *beachline* e não vão mais ser alteradas. Esta estrutura é composta pelos seguintes campos:

- Coordenadas (x, y) dos pontos inicial e final;

- Os dois sítios que a aresta divide: Essa informação não é estritamente necessária, mas é importante para nossa implementação para definirmos quais arestas um sítio possui.

Objeto da *beachline*

A *beachline* contém elementos de dois tipos: Parábulas (ou arcos) e arestas em contrução. Mas sua estrutura é um pouco mais elaborada que as anteriores. Para tratar da *beachline*, foi implementada uma árvore binária de busca, cuja ordenação reflete o posicionamento, da esquerda para a direita, dos componentes presentes na mesma, em que as folhas são os arcos, e os nós internos são as arestas em crescimento. Com isso, podemos definir as duas operações a serem realizadas na árvore:

Operação de sítio: Ao definirmos onde o novo arco será adicionado, o arco anterior que estava naquela posição deve ser dividido em duas partes e, entre elas, o novo arco será inserido, envolto pelas duas novas arestas iniciadas no ponto de interseção. Vale lembrar que as arestas adicionadas, apesar de compartilharem o ponto inicial, possuem direções opostas. Na prática, o arco anterior é apenas copiado para duas posições diferentes. Esse procedimento está ilustrado na Figura 4.2.

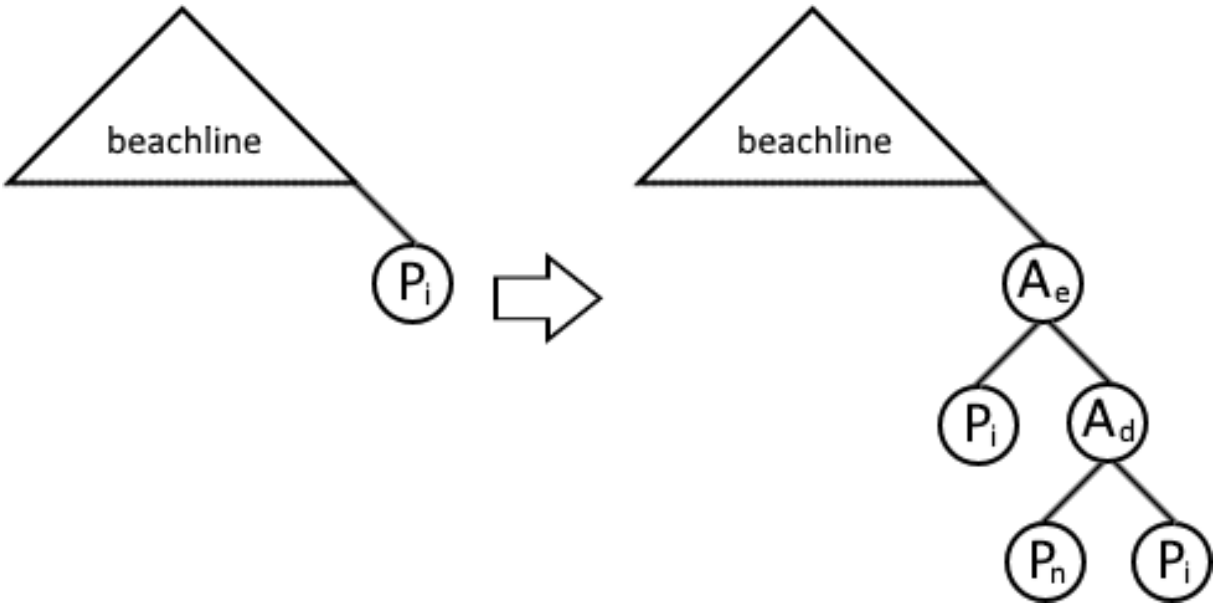


Figura 4.2: Modificação sofrida na árvore de *beachline* decorrida de um evento de sítio. A identificação de cada nó está descrita no Algoritmo 3.

Operação de círculo: As duas arestas que se encontraram, juntamente com a

parábola que existia entre elas, são retiradas da árvore. Essas duas arestas são adicionadas à solução e uma nova é iniciada, na posição onde a aresta de menor profundidade dessas duas estava. Esse procedimento está ilustrado na Figura 4.3.

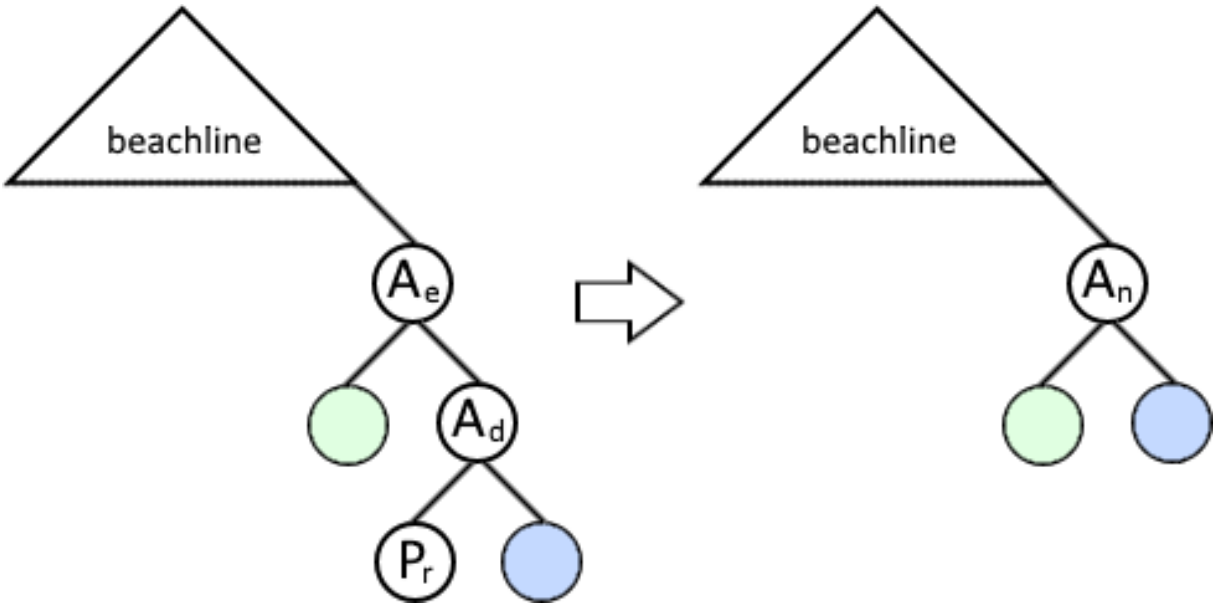


Figura 4.3: Modificação sofrida na árvore de *beachline* decorrida de um evento de círculo. A identificação de cada nó está descrita no Algoritmo 4

Tanto o arco quanto a aresta em crescimento são estruturas que possuem ponteiros para os filhos da esquerda e da direita, e um ponteiro para seu pai. Quanto aos dados que precisamos guardar, a parábola precisa apenas das coordenadas do sítio que a gerou (o foco), já que sua equação só depende desse ponto e da posição y da linha de varredura (a diretriz). Já a aresta em contrução precisa de um ponto inicial e uma direção, que é muito importante na hora de cálcular as interseções, pois as arestas são apenas semi-retas e podem não encontrar com outras mesmo que suas equações tenha um ponto de interseção.

4.3.2 Implementação

Com as estruturas e os procedimentos principais definidos, o funcionamento do método de Fortune pode ser simplificado através dos Algoritmos 2, 3 e 4.

Algoritmo 2: Algoritmo de Fortune simplificado

Resultado: Conjunto de arestas que dividem as células de Voronoi

Inicializar a lista eventos;

Inicializar a lista arestas;

Inicializar a árvore beachline;

foreach *sítio* **do**
| *Adicionar evento de sítio em* eventos

end foreach

while eventos *não está vazio* **do**

| **if** *primeiro item e de* eventos *é do tipo sítio* **then**
| | *Realiza operação de sítio em e*
| **else**
| | *Realiza operação de círculo em e*
| **end if**

end while

Retornar arestas

Algoritmo 3: Operação de Sítio

P_i = *Parábola da beachline que sofreu a interseção*;

P_n = *Parábola nova a ser adicionada*;

A_e = *Aresta nova à esquerda de* P_n;

A_d = *Aresta nova à direita de* P_n;

Substituir P_i *pela sequência (*P_i, A_e, P_n, A_d, P_i*)*;

if *as arestas entre a cópia de* P_i *da esquerda forem se encontrar* **then**
| *Adicionar evento de círculo entre suas arestas em* eventos

end if

if *as arestas entre a cópia de* P_i *da direita forem se encontrar* **then**
| *Adicionar evento de círculo entre suas arestas em* eventos

end if

Algoritmo 4: Operação de Círculo

P_r = Parábola da beachline que será removida pois suas arestas se encontraram;

A_e = Aresta à esquerda de P_r;

A_d = Aresta à direita de P_r;

A_n = Aresta nova a ser adicionada;

Substituir a sequência (A_e, P_r, A_d) por A_n;

if *A_n for se encontrar com sua vizinha à esquerda* **then**
| *Adicionar evento de círculo entre A_n e aresta à sua esquerda em* eventos

end if

if *A_n for se encontrar com sua vizinha à direita* **then**
| *Adicionar evento de círculo entre A_n e aresta à sua direita em* eventos

end if

4.3.3 Parâmetros

Dois parâmetros foram definidos, um deles para a etapa de execução do algoritmo de Fortune e outro para ser utilizado, posteriormente, no preechimento da matriz de textura a partir do resultado retornado pelo algoritmo. São eles:

Número de regiões e caminhos

Esse parâmetro dita quantos sítios serão utilizados durante a geração do diagrama e, portanto, quantas regiões existirão no terreno.

Regiões - preferir região central

O parâmetro faz com que o usuário possa escolher qual distribuição de probabilidade a escolha dos sítios de Voronoi vai seguir. Se essa opção for marcada, será usada a distribuição gaussiana. Caso contrário será totalmente aleatória.

Largura do caminho

No momento em que a matriz de textura é preenchida, o valor deste parâmetro influencia na largura que se estende para fora das arestas, definindo onde a textura utilizada

para os caminhos será aplicada.

4.4 Espalhamento de objetos

O último passo do desenvolvimento é o espalhamento de objetos sobre o terreno. Uma terceira matriz foi utilizada para este propósito, visando representar o que podemos colocar em cada posição do terreno. Esta **matriz de permissões**, na prática, recebe códigos preestabelecidos, identificando o que pode ou não ser colocado em cada posição. Como não haverá construções nos caminhos, ela é iniciada com permissão negativa para construções nesses locais. Isto também é útil pois serve como um delimitador de regiões para o algoritmo *Forest Fire*, que é executado para realizar o preenchimendo da matriz de permissões com os identificadores dos sítios. Segue abaixo o pseudo-código implementado:

Algoritmo 5: Flood Fill (Forest Fire)

Inicializar fila *de coordenadas* (x, y) ;
Enfileira as coordenadas (x, y) *do sítio em* fila;
while fila *não está vazia* **do**
    p = *desenfileira* fila;
    if x *e* y *está dentro dos limites e* matrizPermissao *[x, y] ainda não foi preenchido* **then**
        matrizPermissao *[x, y] = código do sítio*;
        Enfileira as coordenadas $(x, y-1)$ *em* fila;
        Enfileira as coordenadas $(x, y+1)$ *em* fila;
        Enfileira as coordenadas $(x-1, y)$ *em* fila;
        Enfileira as coordenadas $(x+1, y)$ *em* fila;
    end if
end while

4.4.1 Tipos de espalhamento

Com a matriz de permissões inicializada e as células definidas, os objetos já podem ser colocados no terreno. Dependendo do tipo de objeto, ele seguirá regras diferentes. Por isso, foi necessário dividir a distribuição em tipos diferentes, que serão explicados e

exemplificados a seguir, em ordem de execução. Todos os tipos de objetos evitam áreas do terreno com inclinação grande. A divisão dos objetos em cada tipo é feito através de uma hierarquia de pastas. Portanto, para adicionar um novo objeto ou alterar o tipo de espalhamento que ele segue, basta movê-lo para a pasta desejada.

Sítios especiais

Antes de começar com as construções localizadas, os sítios especiais reservam a célula inteira com algum propósito. Assim, um sítio é sorteado, e objetos específicos são espalhados por sua célula. Nesse caso, cada tipo de sítio especial terá um padrão de espalhamento diferente. Implementamos como exemplo um "mercado", no qual modelos diferentes de barracas são colocadas espalhadas no sítio em posições aleatórias.

Construções especiais

As construções especiais são alocadas antes das demais, e apenas uma vez cada. Inicialmente, um sítio é escolhido de forma aleatória. Depois, o objeto é posicionado em algum local próximo a uma das arestas daquele sítio. Finalmente o objeto é rotacionado de forma a ficar de frente para a aresta. Vale relembrar que no modelo implementado, as arestas estão representando os caminhos e ruas, e por isso realizamos o posicionamento e rotação das construções vinculados a elas. Como exemplo desse tipo de espalhamento, foram colocados **catedral**, **igreja**, **forja**, **estalagem** e **galpão**. Foi criada também como experimento as construções isoladas, com a **torre de vigia** como seu único objeto. Esse grupo é semelhante ao das construções especiais, tendo como única diferença um raio de distância em que outras construções e árvores não podem ser alocadas. Outras características como essa podem ser adicionadas para diferenciar de alguma forma a alocação dos objetos.

Construções

As contruções comuns funcionam exatamente como as especiais em termos de alocação e rotação. Porém, são criadas várias instâncias de um mesmo modelo, que é escolhido aleatoriamente de um conjunto armazenado em um vetor. Além disso, criamos

diferentes vetores com modelos diferentes, e cada sítio só apresenta modelos de um mesmo vetor escolhido também aleatoriamente. Exemplificando, separamos as casas em dois grupos, **casas grandes** e **casas pequenas**.

Beirada do caminho

Pequenos objetos, como **carroças**, **caixas**, **barris**, **feno**, podem ser alocados na beirada das estradas. Para isso, o algoritmo implementado percorre as arestas e pode, com uma probabilidade fixa de 2%, sortear um dos objetos para alocar. Em caso positivo de alocação de um objeto, o algoritmo pula as posições que poderiam criar objetos sobrepostos e continua as tentativas de alocação.

Extras

O conceito de extras são para contruções que não seguem um padrão de alocação, nem de rotação. Assim como as contruções especiais, aparecem apenas uma vez, mas em qualquer local do terreno, sem estarem vinculadas a um sítio ou aresta. Os exemplos implementados foram **mina**, **pedreira**, **ruínas**, **serraria** e **matadouro**.

Grama e pedras

Elementos como grama e pequenas pedras são adicionados por todo o terreno, de forma aleatória, evitando apenas os caminhos e os locais com construção.

Árvores

As árvores funcionam de maneira similar à grama e às pedras, mas ocorrem em menor quantidade. Além disso, algumas construções podem restringir a alocação de árvores próximas, que é o caso já comentado da torre de vigia.

4.4.2 Atualizando a matriz de permissões

Sempre que um objeto é alocado no terreno, é preciso atualizar a matriz de permissões para que outro não seja colocado em locais próximos, evitando assim uma sobreposição parcial entre os objetos. Para realizar essa tarefa, foi criado um raio que

desce perpendicular ao plano do terreno, e que é movimentado pelo mapa com o intuito de checar em quais posições ele colide com o objeto adicionado. Nos casos positivos, a coordenada é marcada na matriz de permissões com o devido código.

O mesmo procedimento é realizado para verificar se um objeto pode ser colocado na posição sorteada para ele. Se o raio colidir com o objeto e o ponto no mapa estiver marcado para não permitir novo objeto, uma nova posição deve ser sorteada.

O número máximo de tentativas de alocação é predefinido em 100 para **extras** e **contruções especiais**. Se as tentativas se esgotarem, o objeto não é alocado. Os **sítios especiais** foram projetados para ter comportamentos específicos, mas no caso do exemplo do **mercado**, o valor máximo de tentativas foi predefinido em 1000, e como seus modelos são alocados mais de uma vez, o algoritmo esgota todas as tentativas antes de parar. Para as **contruções**, **árvores** e outros elementos como **grama e pedra**, o número de tentativas é definido através de parâmetros que serão explicados mais à frente nessa seção. No caso dos objetos na **beirada do caminho**, não há um número de tentativas. Todas as posições possíveis são visitadas, e a alocação ou não do objeto depende da probabilidade definida.

4.4.3 Distribuições de probabilidade

Em alguns dos casos apresentados anteriormente, uma escolha de posicionamento completamente aleatória pode não trazer resultados tão naturais. É o caso das árvores, e dos sítios de Voronoi, por exemplo. Pensando no contexto de uma cidade, é razoável pensar que as regiões habitadas estarão mais agrupadas em um centro, enquanto as árvores estarão mais distantes dos locais com construção. Foi então criada uma forma de obter valores aleatórios seguindo uma distribuição gaussiana para o sorteio das regiões, e o inverso dela para a alocação das árvores. Para isso, utilizamos um algoritmo conhecido como **Método polar de Marsaglia** (Marsaglia, 1964). Apesar da probabilidade de números próximos de zero serem escolhidos ser grande, na teoria a distribuição gaussiana pode retornar qualquer valor. Por isso, utilizaremos a **regra dos três sigma** (Upton et al., 2008). De acordo com essa regra e considerando que σ é o desvio padrão da distribuição divido por 2, aproximadamente $99,7\%$ das vezes os resultados obtidos utilizando uma

distribuição normal ficam entre -3σ e 3σ. Dessa forma, sempre que um número for obtido fora dessa faixa, ele é descartado e um novo número é gerado. Também precisamos transpor a distribuição para os nossos valores de coordenadas, mapeando 0 como -3σ e o valor máximo das coordenadas do terreno como 3σ. Unificando todas essas questões, obtemos o algoritmo a seguir:

Algoritmo 6: Distribuição Gaussiana pelo método polar de Marsaglia

do

  do

    a1 = número aleatório entre 0 e 1;

    a2 = número aleatório entre 0 e 1;

    $u = 2 * (v1) - 1$;

    $v = 2 * (v2) - 1$;

    $S = u^2 + v^2$;

  while $S >= 1$;

  $fac = \sqrt{-2 * \log S/S}$;

  $media = (\mathsf{valorMinimo} + \mathsf{valorMaximo})/2$;

  $sigma = (\mathsf{valorMaximo} - media) * \mathsf{desvioPadrao}/3$;

  $\mathsf{rdm} = u * fac * sigma + media$;

while rdm $<$ valorMinimo *ou* rdm $>$ valorMaximo;

Retornar rdm

4.4.4 Parâmetros

Densidade de construções

O parâmetro de densidade de construções é responsável pela quantidade de tentativas que são feitas em cada sítio para se alocar construções. Como a alocação é feita por tentativas, regiões menores geralmente ficarão com maior densidade de contruções naturalmente, e a utilização da distribuição gaussiana para gerar o diagrama de Voronoi tem impacto direto nas alocações.

Densidade de vegetação

Funciona da mesma forma que a densidade de construções, mas para a alocação de árvores, grama e pedras. No Capítulo 5, serão apresentados resultados obtidos utilizando a alocação completamente aleatória e também com a distribuição gaussiana invertida.

Vegetação - Evitar região central

Esse parâmetro tem como objetivo deixar o usuário escolher qual distribuição de probabilidade a alocação das árvores vai seguir. Se marcada, será usada a gaussiana invertida. Caso contrário será totalmente aleatória.

***Checkboxes* para cada especial ou extra**

Cada uma das opções de sítios especiais, construções especiais ou extras podem ser marcados ou desmarcados individualmente na interface de usuário, que será apresentada no Capítulo 5, de forma a alocar ou não tais objetos no terreno.

Período do dia

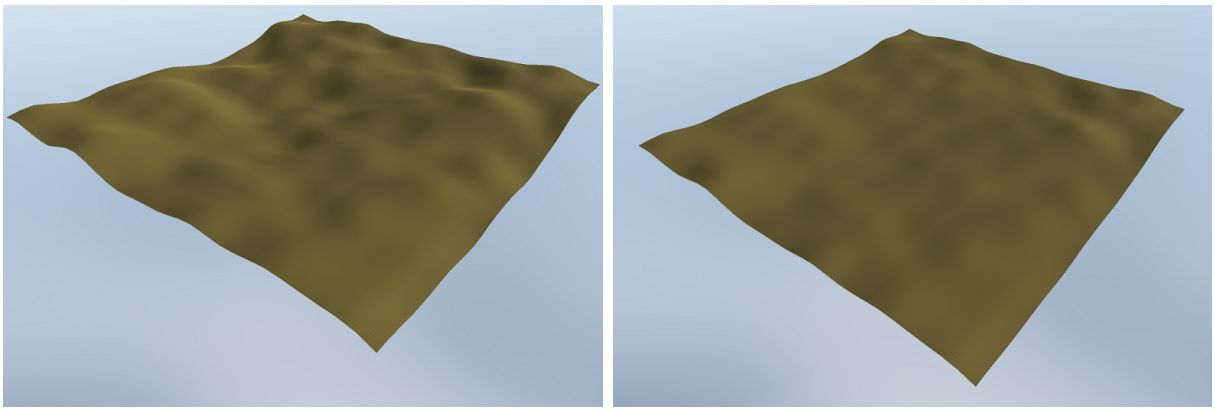
É possível, através desse parâmetro, alterar o período do dia em que a visualização acontece. Ao modificá-lo, duas mudanças são feitas: as imagens que compõem a *skybox* são alteradas, assim como as configurações de iluminação na cena. As alterações na iluminação envolvem mudar sua cor, onde foram utilizados tons amarelados durante o dia, alaranjados no pôr do sol e azulados durante a noite, bem como sua intensidade, na qual foram usados os valores 1 para o dia e o pôr do sol e 0.3 para a noite. A rotação da luz também é alterada, fazendo com que as sombras formadas no pôr do sol fiquem mais alongadas, visto que os raios de luz atingem o terreno e os objetos mais lateralmente, formando ângulos menores com o plano do terreno.

5 Resultados e Experimentos

Neste capítulo apresentaremos os resultados obtidos com as implementações, bem como as modificações e testes importantes realizados durante o processo.

5.1 Terreno

Utilizando apenas a soma de quatro oitavas de Perlin, obtivemos terrenos como o da Figura 5.1a. Observa-se que, apesar de ser um bom resultado, ainda temos pouca área plana para as construções. Por isso foi necessário criar o fator de redistribuição. Dessa forma, fazemos com que as regiões com baixa elevação não destoem muito entre si. As diferenças ficam claras na Figura 5.1b.



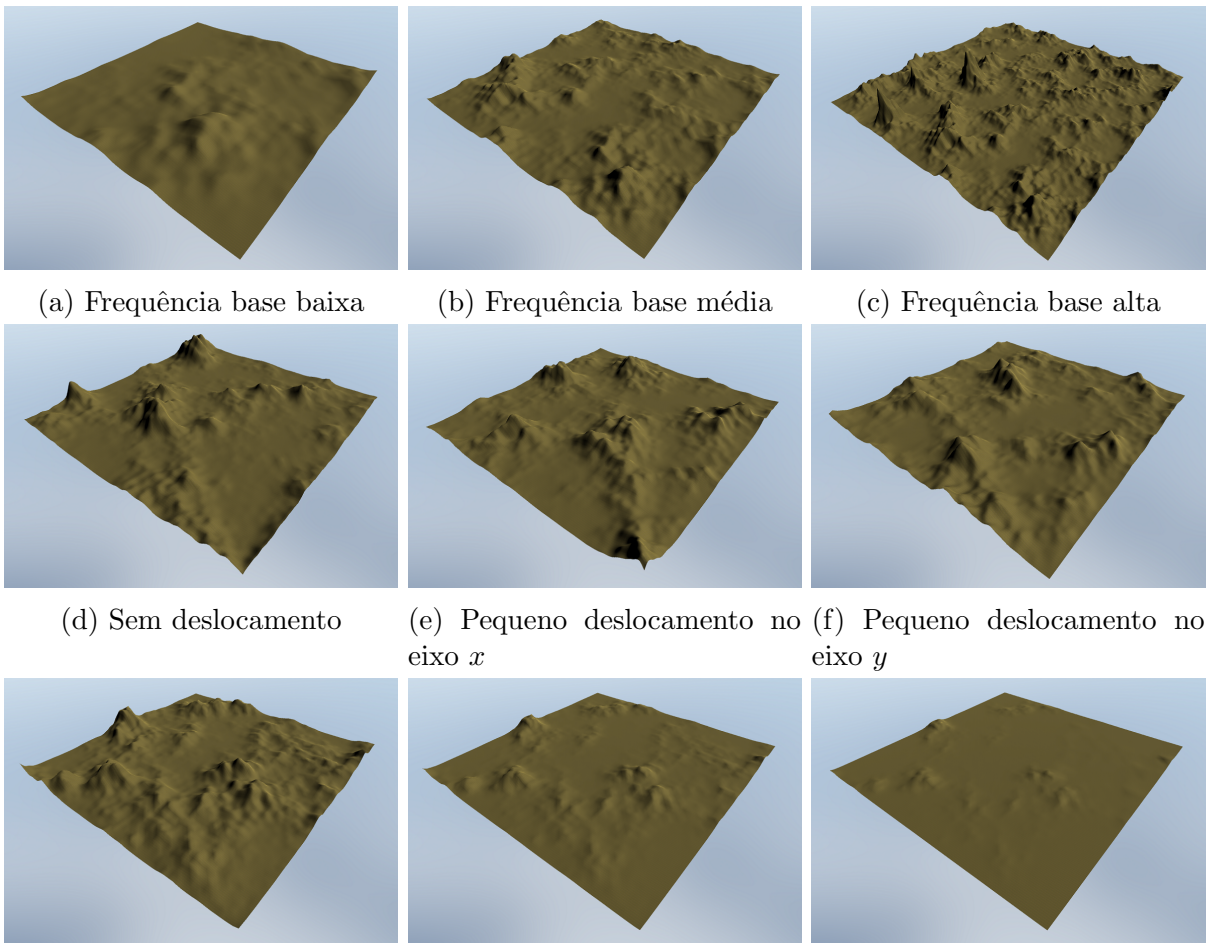
(a) Antes de aplicar o fator de redistribuição (b) Após de aplicar o fator de redistribuição

Figura 5.1: Terrenos gerados usando frequência-base 1, mostrando o efeito da utilização de um fator de redistribuição

Porém, ambas as figuras citadas estão utilizando frequência-base 1, e não possuem deslocamento nenhum no plano do ruído de Perlin. Alterando os parâmetros, obtemos resultados como os mostrados na Figura 5.2.

5.2 Regiões e caminhos

Com a escolha completamente aleatória dos sítios de Voronoi, era comum ocorrerem casos em que as regiões ficassem muito espalhadas. Isso não retrata bem a realidade,



(a) Frequência base baixa (b) Frequência base média (c) Frequência base alta

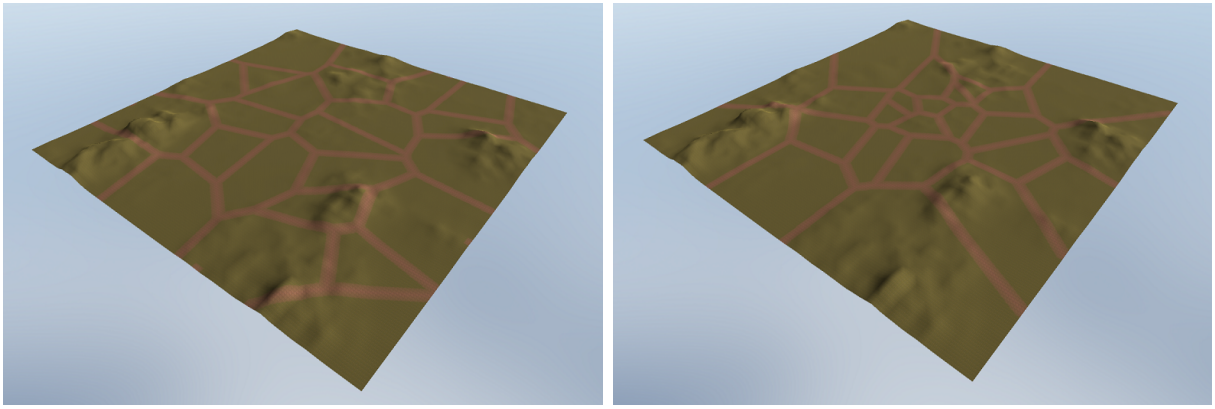
(d) Sem deslocamento (e) Pequeno deslocamento no eixo x (f) Pequeno deslocamento no eixo y

(g) Fator de redistribuição baixo (h) Fator de redistribuição médio (i) Fator de redistribuição alto

Figura 5.2: Demostração dos efeitos dos parâmetros sobre o terreno gerado com ruído de Perlin

em que construções tendem a se concentrar em núcleos menores, e diminuirem de densidade à medida que se afastam desse núcleo. Dessa discrepância, surgiu a ideia de uma distribuição de probabilidade diferente. A distribuição escolhida foi a gaussiana, por ter o comportamento bem condizente a necessidade. Aplicando-a nos dois eixos x e y, obtemos uma distribuição mais interessante das regiões. A diferença dos resultados estão exemplificados na Figura 5.3, e o usuário pode marcar qual tipo de distribuição ele deseja usar.

O número de sítios (ou regiões pelo nome dado ao parâmetro) também é um fator que oferece grande impacto no terreno gerado. Quanto mais sítios, mais arestas. E quanto mais arestas (por mais que elas sejam geralmente menores), mais área tomada pelos caminhos, consequentemente reduzindo posições possíveis para as construções. Naturalmente, a largura definida para as estradas também afeta a ocupação do espaço. Por esse motivo,



(a) Diagrama de Voronoi com sítios totalmente aleatórios aplicado sobre o terreno

(b) Diagrama de Voronoi com sítios escolhidos por distribuição gaussiana aplicado sobre o terreno

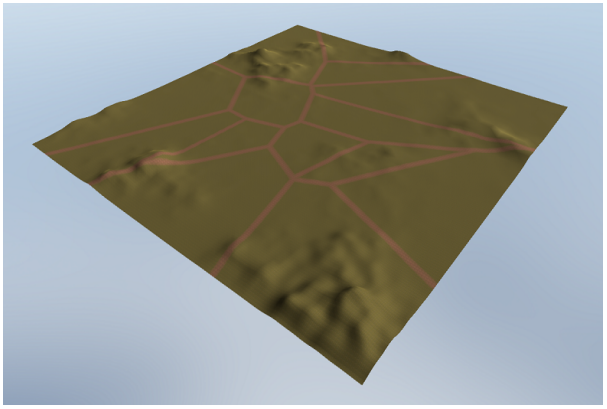
Figura 5.3: Diferença entre escolha aleatória e gaussiana de sítios para o diagrama de Voronoi

um número muito grande de sítios for pareado com uma largura grande para as estradas, os resultados não são tão bons, visto que acabam criando regiões pequenas, comprimidas ainda mais por bordas largas. Todas essas questões são mostradas na Figura 5.4.

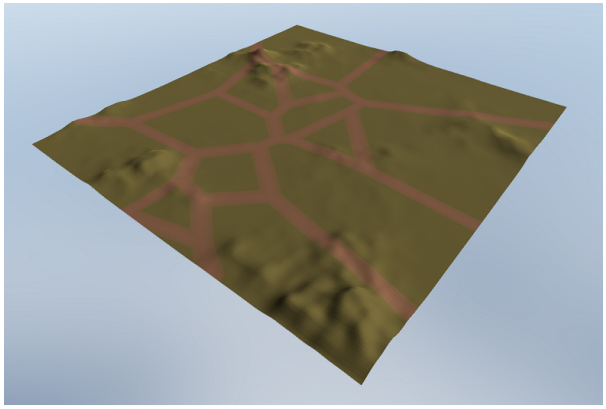
5.3 Espalhamento de objetos

Os parâmetros de densidade de construções e densidade de vegetação tem grande influência nos resultados dessa etapa. A Figura 5.5 ilustra essa relação. Mais adiante, na Figura 5.6, retratamos a diferença entre a utilização do espalhamento completamente aleatório e do espalhamento baseado na distribuição gaussiana invertida, agora para as árvores. Na imagem, atribuímos zero para o parâmetro de densidade de construções, e o máximo para o de vegetação, utilizando apenas três sítios de Voronoi para melhor visualizar o resultado das distribuições.

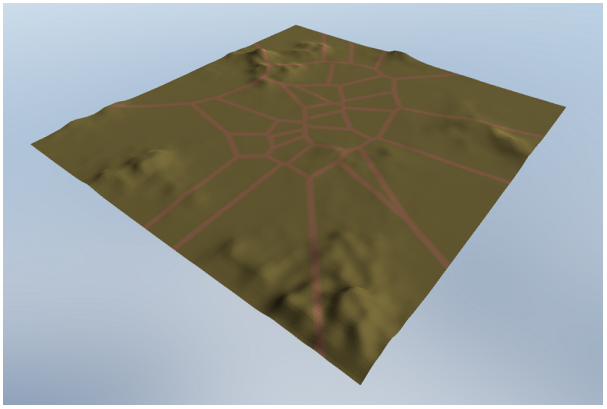
Na Figura 5.7, podemos observar como os diferentes tipos de espalhamentos funcionam. Construções especiais são alocadas como qualquer construção, dependendo de um sítio e um caminho (incluindo a torre de vigia que está classificada como contrução isolada). Os extras são colocados em qualquer lugar do mapa, mas, seguindo a probabilidade da gaussiana invertida, sendo mais provável de aparecerem perto das bordas. Já os sítios especiais são reservados, ou seja, dentro deles apenas os modelos selecionados em sua pasta poderão ser alocados. Outras opções além do mercado teriam formas diferentes



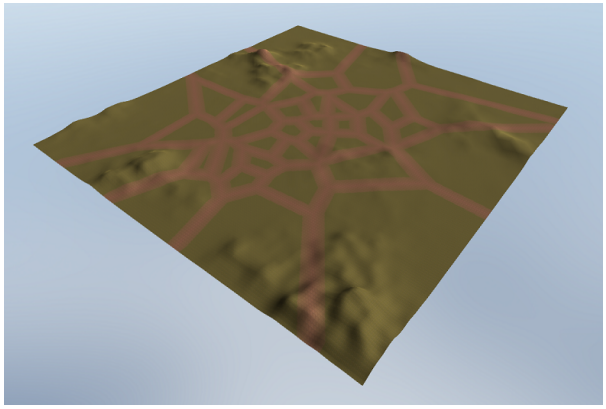
(a) Diagrama de Voronoi aplicado com poucos sítios e estradas estreitas



(b) Diagrama de Voronoi aplicado com poucos sítios e estradas largas



(c) Diagrama de Voronoi aplicado com muitos sítios e estradas estreitas



(d) Diagrama de Voronoi aplicado com muitos sítios e estradas largas

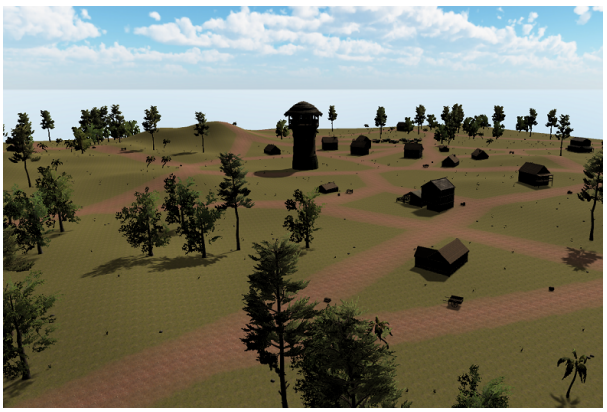
Figura 5.4: Demonstração dos efeitos dos parâmetros para números de regiões e largura dos caminhos

de serem colocadas no mapa.

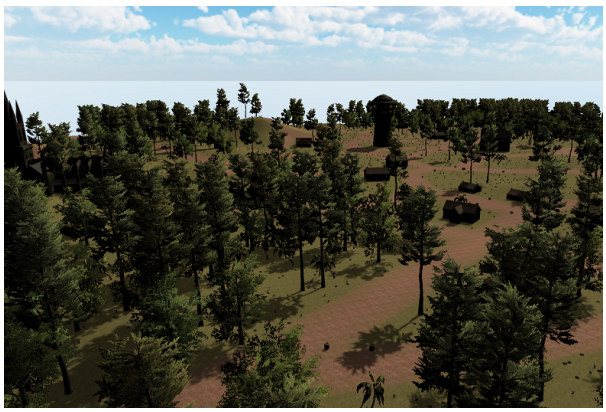
5.4 Outros fatores

Como explicado na Subseção 4.4.4, existe um parâmetro para a escolha do período do dia em que a visualização será apresentada. O reflexo da mudança desse parâmetro é mostrada na Figura 5.8.

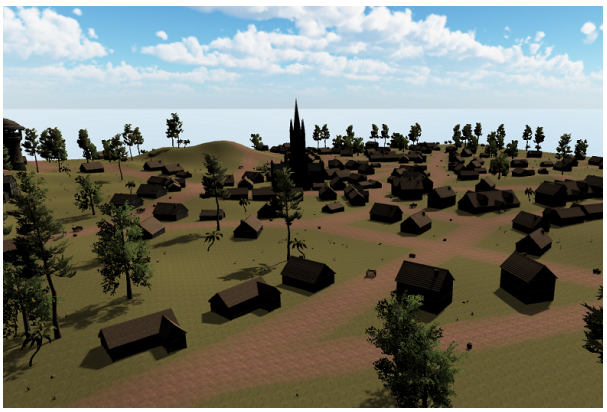
Foi também implementado um modo de jogo, em que a câmera fica dentro do terreno e com a rotação travada de forma a mostrar sempre os objetos de cima para baixo (Figura 5.9). Existe também a opção de alterar o modo de exibição da câmera de perspectiva para ortogonal, recurso demonstrado na mesma imagem. Em casos onde construções muito altas como a torre de vigia ou a catedral estão presentes na câmera de jogo, usar a opção de câmera ortogonal se torna mais viável, pois tais construções acabam



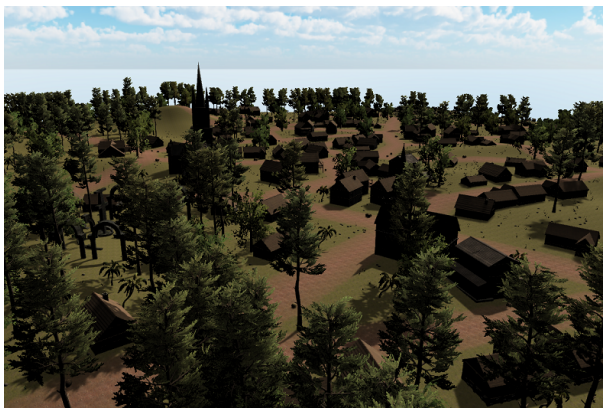
(a) Conteúdo gerado com baixa densidade de construções e baixa densidade de vegetação



(b) Conteúdo gerado com baixa densidade de construções e alta densidade de vegetação

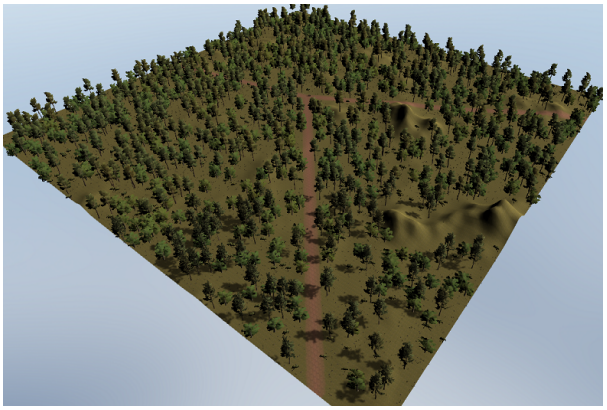


(c) Conteúdo gerado com alta densidade de construções e baixa densidade de vegetação

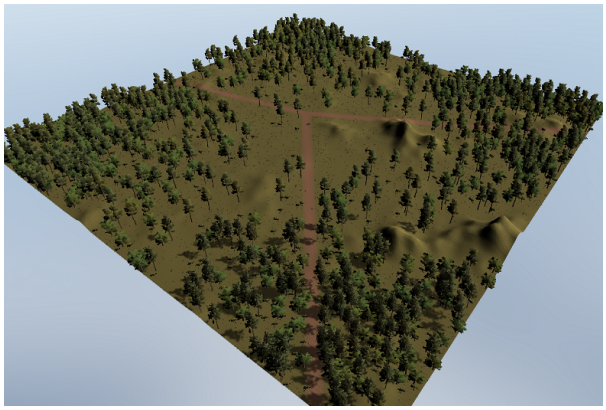


(d) Conteúdo gerado com alta densidade de construções e alta densidade de vegetação

Figura 5.5: Demonstração dos efeitos dos parâmetros de densidade de construções e vegetação



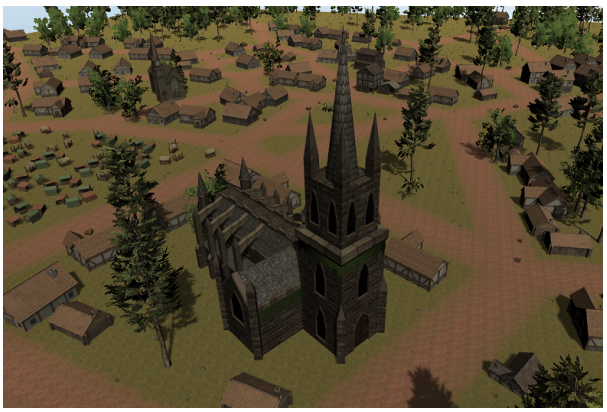
(a) Vegetação alocada de forma totalmente aleatória



(b) Vegetação alocada usando distribuição gaussiana invertida

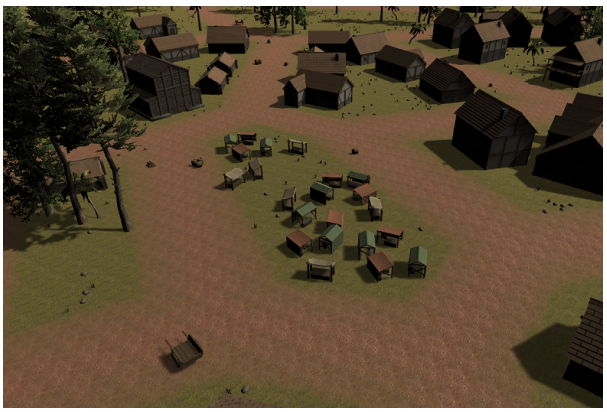
Figura 5.6: Diferenças na distribuição de probabilidade para vegetação

tomando grande parte da visão no caso da câmera em perspectiva.



(a) Exemplo de alocação de contruções Eepeciais: Catedral

(b) Exemplo de alocação de extras: Ruínas



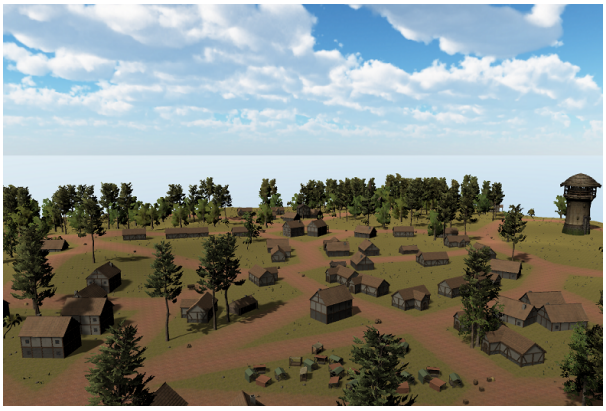
(c) Exemplo de alocação de sítios especiais: Mercado

(d) Exemplo de alocação de construções especiais isoladas: Torre de vigia

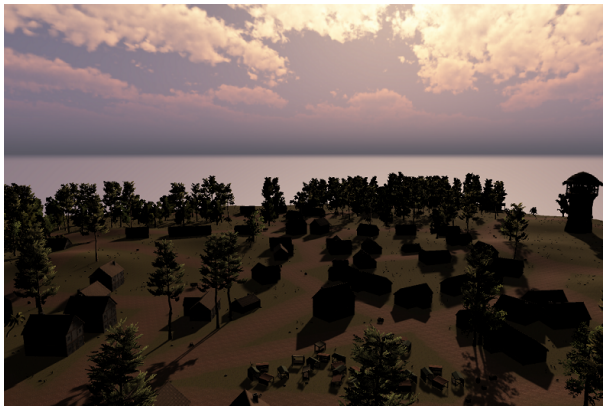
Figura 5.7: Exemplos de tipos de alocações de objetos

5.5 Cenários de uso

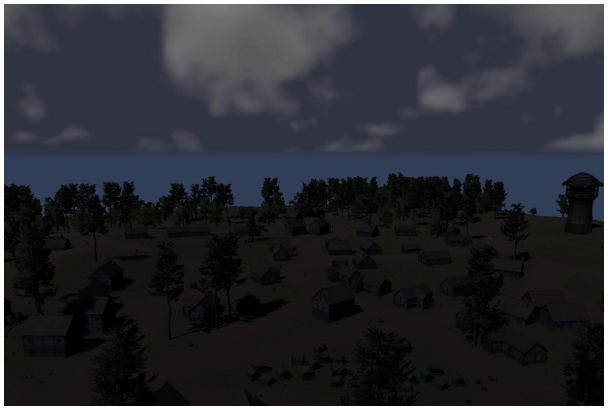
Os resultados obtidos podem ser utilizados de diferentes formas, de acordo com a necessidade dos usuários. Cenas como as da Figura 5.5 podem servir de inspiração para o GM descrever o local para os jogadores, situando melhor a localização de cada construção, ou até mesmo mostrá-los parte do que foi gerado. Outro formato que costuma ser bem utilizado é o de visão superior, como apresentado no modo de jogo. Gerando resultados como o da imagem 5.9, é possível tanto a impressão para uso sobre uma superfície quanto a utilização de uma TV ou outro dispositivo do gênero, virado para cima, de forma que miniaturas ou outro tipo de marcador representando os personagens possam ser colocados sobre a tela do dispositivo. A Figura 5.10 ilustra esse tipo de uso com uma mesa digital. As imagens presentes na mesa poderiam ser substituídas pelas geradas com o modo de jogo deste trabalho.



(a) Período: Dia

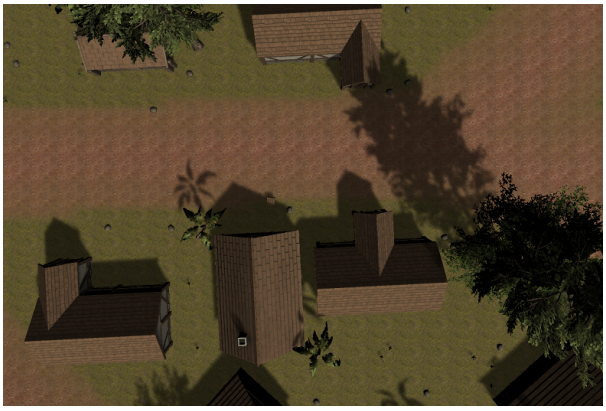


(b) Período: Pôr do sol

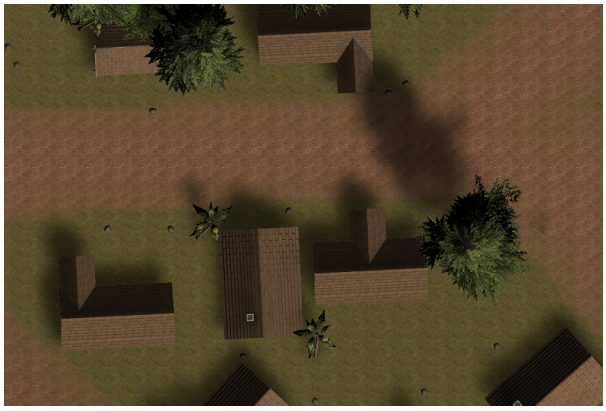


(c) Período: Noite

Figura 5.8: Exemplos de mesmo terreno e objetos alocados em diferentes períodos do dia

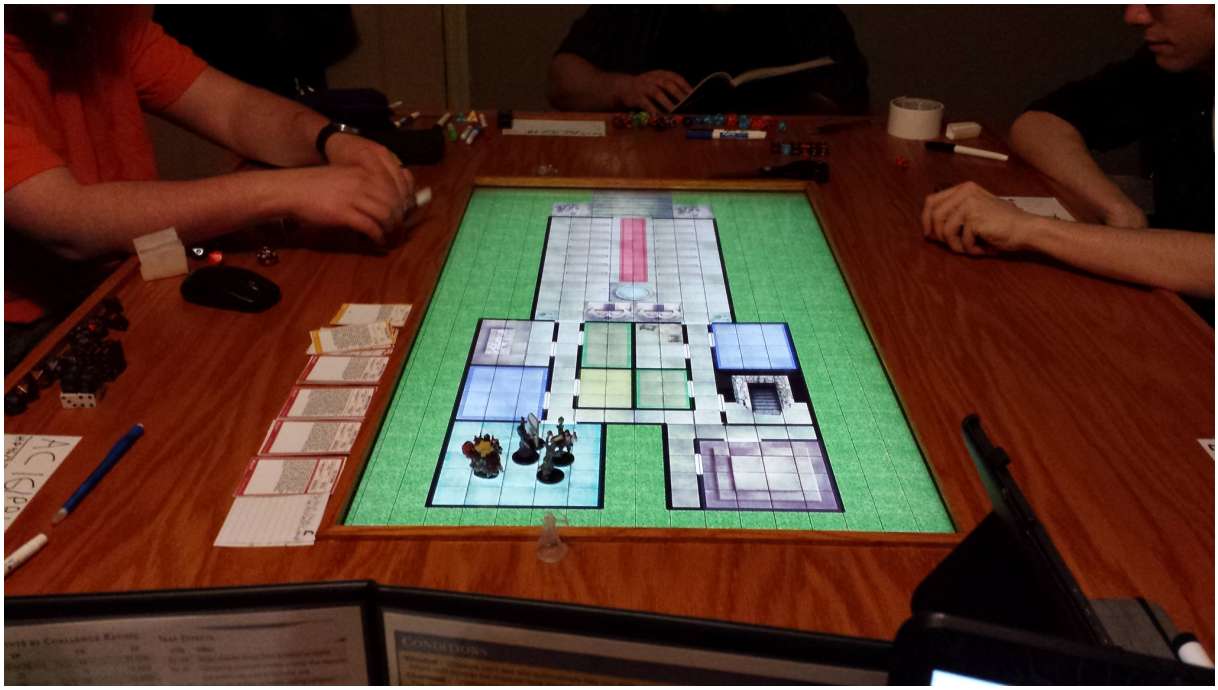


(a) Câmera de visão em perspectiva dentro do modo de jogo

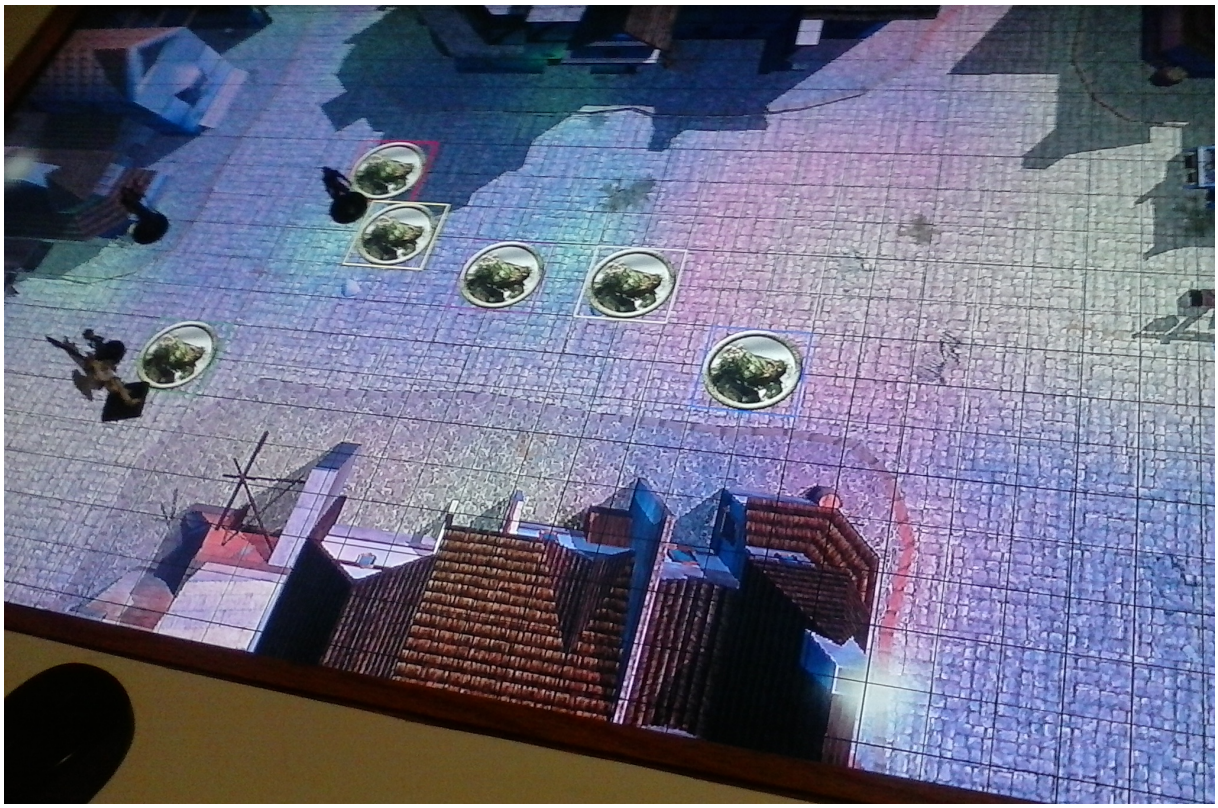


(b) Câmera de visão ortogonal dentro do modo de jogo

Figura 5.9: Câmeras no modo de jogo



(a)



(b)

Figura 5.10: Exemplo de utilização de imagens vistas de cima em uma mesa digital (Enworld, 2016)

6 Conclusões e trabalhos futuros

O desenvolvimento desse trabalho teve como objetivo obter uma aplicação capaz de produzir uma ambientação visual para jogos de tabuleiro e RPG de mesa. Todo o processo foi realizado dentro do motor de jogos Unity, e, para atingir os propósitos da aplicação, tivemos como base a utilização de geração procedural, que foi por sua vez dividida em duas etapas: a geração do terreno e a distribuição de conteúdo.

A geração do terreno foi realizada utilizando a soma de quatro oitavas do ruído de Perlin, pareado com um fator de redistribuição para que os resultados ficassem mais condizentes com as necessidades das etapas seguintes. Depois de construído o mapa de altura do terreno, foi proposto a utilização de um diagrama de Voronoi para separar o terreno em regiões definindo caminhos entre elas. Para a implementação do diagrama, foi utilizado o algoritmo de Fortune, que oferece uma eficiência bastante superior ao algoritmo ingênuo. Depois das regiões definidas, alocamos os objetos de acordo com suas características predefinidas.

A principal limitação da solução que criamos é em relação à inclinação do terreno. Ângulos grandes gerados no mapa de altura impossibilitam o alocamento da maioria dos objetos, pois parte deles ficaria por dentro do terreno. Para as construções, isso daria a impressão de estarem soterradas, principalmente quando as portas de entrada ficassem na parte de maior altura. Uma possibilidade de melhoria nesse aspecto seria o desenvolvimento de uma forma de alterar o próprio terreno, de acordo com as construções que são alocadas. Além disso, alguns dos objetos utilizados precisaram ser tratados dentro da Unity, de forma a terem uma geometria de colisão associada. No trabalho, esse componente é usado na hora de verificar quais as posições devem ser marcadas para não alocarmos construções sobrepostas. Assim, seria interessante implementar uma forma de atribuir essa geometria de colisão automaticamente, visando permitir ao usuário adicionar os seus próprios modelos diretamente na aplicação.

Como trabalhos futuros, seria interessante a implementação de múltiplos terrenos conectados entre si por alguma das arestas de seus diagramas de Voronoi. Dessa forma,

poderíamos ter, por exemplo, pequenas vilas conectadas por uma floresta. Múltiplos terrenos também poderiam ser usados para separar áreas que seriam planas e áreas que seriam montanhosas. Dessa forma, poderíamos evitar o uso de um fator de redistribuição para criarmos um região plana. Uma outra ideia interessante seria a opção de adicionar outros elementos, como um rio que corta o terreno, por exemplo. Além dos elementos de terreno, há a possibilidade de se incluir modelos de pessoas e animais se movimentando pelo mapa, aplicando algum método de simulação de multidões.

Para além da proposta inicial, uma melhoria interessante seria a possibilidade dos jogadores carregarem os modelos e informações dos seus personagens para serem mostradas no ambiente, durante o jogo. Por fim, a adição de áudios de ambiente também traria grande acréscimo ao trabalho, visto que aumentaria ainda mais a imersão dos jogadores.

Bibliografia

Bergström, K.; Jonsson, S. ; Björk, S. **Undercurrents - a computer-based gameplay tool to support tabletop roleplaying**. In: Nordic DiGRA 2010, 2010.

Beyer, T. **Story guided procedural generation of complex connected worlds and levels for role play games**. 2017. Dissertação de Mestrado - Technical University of Munich, Germany.

Cardamone, L.; Loiacono, D. ; Lanzi, P. L. **Interactive evolution for the procedural generation of tracks in a high-end racing game**. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation, p. 395–402. ACM, 2011.

Diablo. Blizzard North, 1996.

Emilien, A.; Bernhardt, A.; Peytavie, A.; Cani, M.-P. ; Galin, E. Procedural generation of villages on arbitrary terrains. **The Visual Computer**, v.28, n.6-8, p. 809–818, 2012.

Digital gaming tables. Enworld: Morrus' Unofficial Tabletop RPG News, 2016. Disponível em:<http://www.enworld.org/forum/showthread.php?477324-Digital-Gaming-Tables>. Acesso em: 20 jun. 2019.

Eriksson, D.; Peitz, J. ; Björk, S. **Enhancing board games with electronics**. In: Proceedings of the 2nd International Workshop on Pervasive Games-PerGames, 2005.

Fortune, S. A sweepline algorithm for voronoi diagrams. **Algorithmica**, v.2, n.1-4, p. 153, 1987.

Frank, E.; Olsson, N. **Procedural city generation using perlin noise**, 2017.

Freiknecht, J.; Effelsberg, W. A survey on the procedural generation of virtual worlds. **Multimodal Technologies and Interaction**, v.1, n.4, p. 27, 2017.

Galin, E.; Peytavie, A.; Maréchal, N. ; Guérin, E. **Procedural generation of roads**. In: Computer Graphics Forum, volume 29, p. 429–438. Wiley Online Library, 2010.

Greuter, S.; Parker, J.; Stewart, N. ; Leach, G. **Real-time procedural generation of "pseudo infinite" cities**. In: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia, p. 87–94. ACM, 2003.

Groenewegen, S.; Smelik, R. M.; de Kraker, K. J. ; Bidarra, R. **Procedural city layout generation based on urban land use models.** In: Eurographics (Short Papers), p. 45–48, 2009.

Lee, W.; Woo, W. ; Lee, J. **Tarboard: Tangible augmented reality system for table-top game environment**. In: 2nd International Workshop on Pervasive Gaming Applications, PerGames, volume 5, p. 57–61, 2005.

Li, Z.; Zhu, C. ; Gold, C. **Digital terrain modeling: principles and methodology**. CRC press, 2004.

Li, H.; Yang, H.; Xu, C. ; Cao, Y. **Water surface simulation based on perlin noise and secondary distorted textures**. In: Advances in Computer Science and Ubiquitous Computing, p. 236–245. Springer, 2016.

Marsaglia, G.; Bray, T. A. A convenient method for generating normal variables. **SIAM review**, v.6, n.3, p. 260–264, 1964.

Minecraft. Mojang, 2011.

Mustafa, A.; Wei Zhang, X.; Aliaga, D. G.; Bruwier, M.; Nishida, G.; Dewals, B.; Erpicum, S.; Archambeau, P.; Pirotton, M. ; Teller, J. Procedural generation of flood-sensitive urban layouts. **Environment and Planning B: Urban Analytics and City Science**, 2018.

Nilsen, T.; Looser, J. **Tankwar - tabletop war gaming in augmented reality**. In: 2nd International Workshop on Pervasive Gaming Applications, PerGames, volume 5, p. 86–91. Citeseer, 2005.

No man's sky. Hello Games, 2016.

Perlin, K. An image synthesizer. **ACM Siggraph Computer Graphics**, v.19, n.3, p. 287–296, 1985.

Perlin, K. **Improving noise**. In: ACM transactions on graphics (TOG), volume 21, p. 681–682. ACM, 2002.

Shaker, N.; Togelius, J. ; Nelson, M. J. **Procedural content generation in games**. Springer, 2016.

Smith, G. **An analog history of procedural content generation**. In: FDG, 2015.

Stickert, S.; Hiller, H. ; Echtler, F. **Companion - a software toolkit for digitally aided pen-and-paper tabletop roleplaying**. In: The 31st Annual ACM Symposium on User Interface Software and Technology Adjunct Proceedings, p. 48–50. ACM, 2018.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A. ; Togelius, J. Procedural content generation via machine learning (pcgml). **IEEE Transactions on Games**, v.10, n.3, p. 257–270, 2018.

Tan, J.; Rau, P.-L. P. A design of augmented tabletop game based on rfid technology. **Procedia Manufacturing**, v.3, p. 2142–2148, 2015.

Torbert, S. **Applied computer science**. Springer, 2016.

Upton, G.; Cook, I. **A Dictionary of Statistics**. Oxford University Press, 2008.

De Berg, M.; Van Kreveld, M.; Overmars, M. ; Cheong, O. **Computational Geometry Algorithms and Applications**. Springer, 1997.