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

GERAÇÃO DE CAMINHOS EM CENÁRIOS 3D A PARTIR DE MAPAS DE CUSTO UTILIZANDO MEAN SHIFT

TRABALHO DE GRADUAÇÃO

Bernardo Henz

Santa Maria, RS, Brasil

2011

GERAÇÃO DE CAMINHOS EM CENÁRIOS 3D A PARTIR DE MAPAS DE CUSTO UTILIZANDO MEAN SHIFT

por

Bernardo Henz

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

Orientador: Prof. Dr. Cesar Tadeu Pozzer

Trabalho de Graduação N. 322

Santa Maria, RS, Brasil

2011

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

GERAÇÃO DE CAMINHOS EM CENÁRIOS 3D A PARTIR DE MAPAS DE CUSTO UTILIZANDO MEAN SHIFT

elaborado por
Bernardo Henz

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Cesar Tadeu Pozzer
(Presidente/Orientador)

Prof. Dr. Marcos Cordeiro d'Ornellas (UFSM)

Profª. Dra. Juliana Kaizer Vizzotto (UFSM)

Santa Maria, 16 de Dezembro de 2011.

“The beginning of wisdom is the statement ’I do not know.’ The person who cannot make that statement is one who will never learn anything. And I have prided myself on my ability to learn.”

THRALL (WARCRAFT)

AGRADECIMENTOS

Agradeço primeiramente a minha família: pai, mãe e irmão , por sempre estarem comigo durante os melhores e mais desafiadores momentos, apoiando e me dando força sempre que precisei.

Agradeço a todos os amigos que fizeram parte da vida e que de alguma forma contribuíram para eu chegar aonde estou. Em especial aos grandes amigos do InfCoolGuys: André, Cícero, Cristiano, Frederico e Daniel; grupo que sempre esteve unido durante a graduação.

Agradeço meu orientador Cesar Tadeu Pozzer, que além de me aconselhar, corrigir e cobrar foi um grande amigo. Agradecimentos a todo o pessoal do LaCA (Dudu, Lennom e muitos outros) por terem me aguentado junto no laboratório por tanto tempo. A todos os amigos que fiz no PET, grandes parceiros que não serão esquecidos. Ao pessoal do DAINF (Fred, Bass, Lunardi, Kreutz) que sempre estiveram dispostos a se meterem em furada para melhorar o Curso. Agradeço aos veteranos que durante o curso e mesmo depois de formados continuaram sendo grandes amigos: Ceretta, Gottin, Rosca, PL, Magoo e Celito.

Agradeço a todos os professores que contribuíram para eu crescer como pessoa. Agradeço a Janice e Benhur por sempre estarem dispostos a me ajudar quando necessário. Agradeço a todos os mestres que me ensinaram e cobraram para eu poder crescer.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

GERAÇÃO DE CAMINHOS EM CENÁRIOS 3D A PARTIR DE MAPAS DE CUSTO UTILIZANDO MEAN SHIFT
Autor: Bernardo Henz
Orientador: Prof. Dr. Cesar Tadeu Pozzer
Local e data da defesa: Santa Maria, 16 de Dezembro de 2011.

Na área de desenvolvimento de jogos, grande parte do tempo é gasto na criação de conteúdo (terrenos, vegetação, caminhos, cidades). Dessa forma, várias ferramentas de geração automática de conteúdo são desenvolvidas para facilitar e agilizar o processo de criação de jogos. O presente trabalho propõe um método automático de geração de caminhos em cenários 3D. Dado um conjunto de imagens de entrada, representando características do cenário, regiões são determinadas para a construção de um mapa de custo final, que é utilizado para construir o caminho de menor custo conectando dois pontos do cenário. Apresenta-se um sistema capaz de gerar mapas de custo utilizando o ruído de Perlin, classificar as imagens de entrada usando Mean shift e encontrar o caminho de menor custo utilizando o A*. O caminho gerado pode ser exportado, podendo ser utilizado para enriquecer cenários ou para movimentar personagens.

Palavras-chave: Geração procedural; Perlin Noise; Mean shift; pathfinding; A*; computação gráfica.

ABSTRACT

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

GENERATION OF PATHS IN 3D ENVIRONMENTS FROM COST MAPS USING MEAN SHIFT
Author: Bernardo Henz
Advisor: Prof. Dr. Cesar Tadeu Pozzer

In game development area, most of the time is spent on content creation (terrain, vegetation, paths, cities). In this way, many methods for automatic generation of content are designed to facilitate and accelerate the process of creating a game. The present paper proposes an automatic method of path generation in 3D scenario. Given a set of input images, which represents the scenario's features, clusters are determined to build a final cost map, which is used to build the lowest cost path connecting two points in the scenario. This paper presents a system capable of generating cost maps using Perlin noise, classify the input images using Mean shift and find the lowest cost path using A*. The path generated can be exported and can be uses to enrich scenarios or to move characters.

Keywords: procedural generation, Perlin Noise, Mean shift, pathfinding, A*.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

LaCA	Laboratório de Computação Aplicada
GUI	Graphical User Interface
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision
OGRE	Object-oriented Graphics Rendering Engine
API	Application Programming Interface
IDE	Integrated Development Environment
SCV	Simple Component for Visual

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos tem sido possível perceber a ascensão da indústria de jogos, a qual já se mostra em uma posição importante no mercado atual. Apesar da indústria de jogos ser nova (se comparada com a indústria cinematográfica e fonográfica), esta apresenta um crescimento exponencial: em 2010 movimentou mais de US$ 60,4 bilhões, o dobro do valor movimentado pelos filmes produzidos em Hollywood (LANDIM, 2011). Entretanto, o preço para desenvolver um jogo é muito alto, o que faz com que sejam explorados meios de diminuir isso.

Devido à necessidade da criação de jogos cada vez mais inovadores, realistas e desafiadores, a indústria de jogos investe pesadamente na área de desenvolvimento. Uma das principais preocupações na área de desenvolvimento de jogos é a geração de conteúdo, onde *game designers* passam meses desenhando terrenos, populando cidades, modelando estradas e vegetações em cenários, entre outros. Com o objetivo de diminuir o tempo gasto na criação de conteúdo para o jogo, várias ferramentas foram desenvolvidas para facilitar e agilizar o trabalho de criação de jogos. Dentre estas, a geração automática de conteúdo a partir de técnicas procedurais é uma alternativa que vem ganhando espaço no mercado de jogos.

A geração procedural visa à utilização de algoritmos procedurais (fractais, ruídos pseudorandômicos) para geração de conteúdo. Muitos jogos já fazem uso destes algoritmos para geração do conteúdo: o jogo Minecraft (PERSSON, 2009) utiliza algoritmos procedurais para construir o terreno inicial (Figura 1.1), o jogo Dwarf Fortress (Bay 12 Games, 2006) utiliza geração procedural para construir o mundo, gerando história e características dos personagens (Figura 1.2), o jogo Spore (MAXIS, 2006) utiliza estas técnicas para criação dos personagens (a Figura 1.3 mostra a tela de criação de personagens, onde a geração procedural possibilita ao jogador uma quantidade inimaginável de combinações

para criar o personagem). Pode-se observar, dessa forma, que além da geração procedural diminuir o trabalho do *game designer* de produzir manualmente o conteúdo do jogo, ela também é capaz de acrescentar um dinamismo ao jogo, visto que pode gerar conteúdo em tempo-real, fazendo com que o mesmo não seja tão repetitivo e monótono.

Figura 1.1: Tela do jogo Minecraft, terreno gerado a partir de técnicas de geração procedural. Fonte: http:botchweed.comwp-contentuploads201109minecraft_adventure_update_terrain.jpg

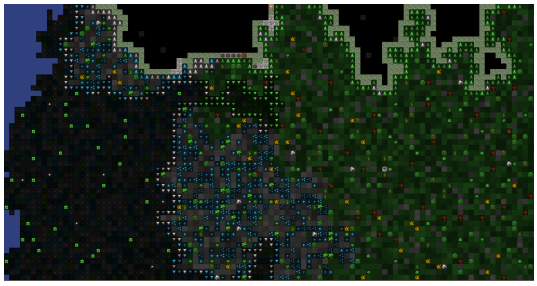


Figura 1.2: Tela do jogo Dwarf Fortress.

Figura 1.3: Tela de criação de personagem no jogo Spore. Fonte: http://www.aeropause.com/2008/06/play-spore-creature-creator-now/

Da mesma forma que existem vários algoritmos para automatizar a geração de terrenos, histórias e personagens, vários trabalhos já foram desenvolvidos visando agilizar o processo de geração de caminhos; tanto no que diz respeito a estradas geradas em cenários, quanto à geração de caminhos a serem seguidos por personagens. É importante

ressaltar que a geração de caminhos possui suas particularidades, como preocupar-se em não criar caminhos impossíveis de serem percorridos pelo personagem. Logo, uma geração utilizando algoritmos pseudorandômicos (como os utilizados para geração de terrenos, por exemplo) não geram resultados satisfatórios, visto que não é possível manter um controle do caminho gerado.

Devido a essas particularidades, muitas ferramentas (Six Times Nothing, 2011) e métodos foram propostos para criação de um rascunho de uma estrada (BRUNETON; NEYRET, 2008) ou edição de uma já criada (MCCRAE; SINGH, 2009). Entretanto, estes métodos ainda fazem com que seja necessário um esforço do *game designer* para desenhar e projetar o caminho. Um método proposto por Galin (2010) utiliza o algoritmo de busca em grafos A* para encontrar o caminho de menor custo, dado um conjunto de mapas de custo (GALIN et al., 2010). Este último método se mostrou muito interessante na medida em que é possível gerar diferentes caminhos a partir de diferentes mapas de custo, onde são atribuídos diferentes pesos a cada mapa. Entretanto, o método proposto por Galin limita a criação de caminhos, pois o peso de um mapa é uniforme.

Este trabalho tem como proposta um algoritmo que leve em conta o conjunto de características no momento de atribuir um custo a determinado ponto. Assim, dado um conjunto de entrada de mapas de custos (os quais podem ser gerados por técnicas de geração procedural), será utilizado um algoritmo de segmentação que agrupará regiões com características semelhantes. Dessa forma poderá ser atribuído o peso de cada mapa em cada uma das regiões. Com isso, será possível construir um mapa de custo final, que será utilizado como entrada para o algoritmo A*, com o qual se encontrará o caminho de menor custo.

1.1 Objetivos

O objetivo geral deste trabalho é propor uma técnica de geração de caminhos que leve em conta características do terreno definidas pelo usuário. As características serão representadas por mapas de custo, os quais podem ser criados a partir do Ruído de Perlin. Serão atribuídos pesos a estes mapas, de forma com que seja construído um mapa final a ser avaliado na função de custo utilizada pelo algoritmo A*, o qual encontrará o caminho de menor custo estre dois pontos arbitrários. Visa-se um controle do peso de cada característica (mapa) na avaliação da função de custo. Entretanto, diferente de modelos atuais

(onde é determinado um peso uniforme e constante a uma característica), pretende-se fazer com que o peso da característica possa variar de acordo com o conjunto em que esta está incluída.

Dessa forma será utilizado um algoritmo de segmentação (Mean shift) para agrupar regiões de características semelhantes, de forma que estas possam ser avaliadas e pesadas conforme a ocasião. Este trabalho tem como objetivos específicos:

- Geração de *heightmaps* a partir do ruído de Perlin;
- Utilização do Mean shift para segmentação de mapas de entrada;
- Utilização do OpenCL para otimização do algoritmo do Mean shift;
- Utilização do algoritmo A* para encontrar o caminho de menor custo em um mapa de custos final;
- Utilização do OpenCV para manipulação de imagens utilizadas/geradas.

1.2 Estrutura do Trabalho

De modo a estruturar o texto, o Capítulo 2 apresenta uma revisão dos fundamentos teóricos envolvidos neste trabalho. Assim, são explicados os conceitos de geração procedural, segmentação de imagens em *clusters* utilizando o Mean shift, algoritmo de *pathfinding* A* e ferramentas utilizadas no trabalho: OpenCL e OpenCV. No Capítulo 3 são tratadas as proposições do trabalho e justificada a escolha dos algoritmos utilizados. O Capítulo 4 esclarece a implementação do trabalho proposto e como as técnicas foram utilizadas. O Capítulo 5 apresenta um cenário de teste utilizando o sistema proposto, mostrando os resultados obtidos. Considerações e trabalhos futuros encontram-se no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem por objetivo a apresentação das tecnologias utilizadas no projeto. São expostos a técnica de geração procedural (ruído de Perlin), o algoritmo de segmentação de imagens Mean shift e o algoritmo de pathfinding A*. Também serão apresentadas bibliotecas utilizadas no trabalho: OpenCL, OpenCV e SCV.

2.1 Geração procedural

A geração procedural é uma técnica largamente utilizada para geração de mídia para jogos (EBERT et al., 2002). Esta ferramenta de desenvolvimento surgiu inicialmente com o intuito de contornar problemas de memória que existiam nos primeiros consoles e computadores, onde projetistas utilizavam algoritmos procedurais para construção do conteúdo desejado em tempo de execução. Entretanto, estes algoritmos não podiam ser randômicos, o que resultaria na geração de um conteúdo diferente a cada execução do jogo. Devido a isso, eram utilizados algoritmos com funções determinísticas, as quais para a mesma entrada geram a mesma saída, dessa forma era possível reconstruir o mesmo conteúdo em cada execução do jogo. Ademais, estes algoritmos de geração costumavam ser pseudo-randômicos, evitando a geração de padrões e repetições (que normalmente não são desejados em jogos).

Com a evolução do hardware de consoles e computadores, a memória utilizada para armazenamento das informações não é mais uma limitação e o propósito dos algoritmos de geração procedural acabou mudando de foco. Atualmente algoritmos de geração procedural são utilizados para automatizar e agilizar o processo do *designer* na criação de conteúdo, diminuindo o tempo gasto modelando e projetando conteúdos manualmente. Dessa forma, são utilizadas funções parametrizadas que gerem conteúdo de forma automática, fazendo com que o *designer* só necessite realizar pequenas mudanças no conteúdo

gerado.

Dentre os diversos algoritmos para geração procedural, tais como geração de fractais e aplicação de filtros em texturas (LEWIS, 1984), um dos mais utilizados é a geração de ruído através de funções iterativas. O Ruído de Perlin (PERLIN, 1985) é uma das técnicas mais utilizadas para geração de ruído e será detalhada a seguir.

2.1.1 Ruído de Perlin

O Ruído de Perlin (do inglês *Perlin Noise*) é uma técnica de geração procedural que foi criada inicialmente para geração de texturas. Sua principal característica é que não é necessária uma imagem/textura de origem para gerar um resultado, visto que o ruído de Perlin faz uso de expressões matemáticas para construção da textura. Esta construção é feita por uma soma finita de funções ruído (como pode ser visto na Figura 2.1) de diferentes frequências e amplitudes.

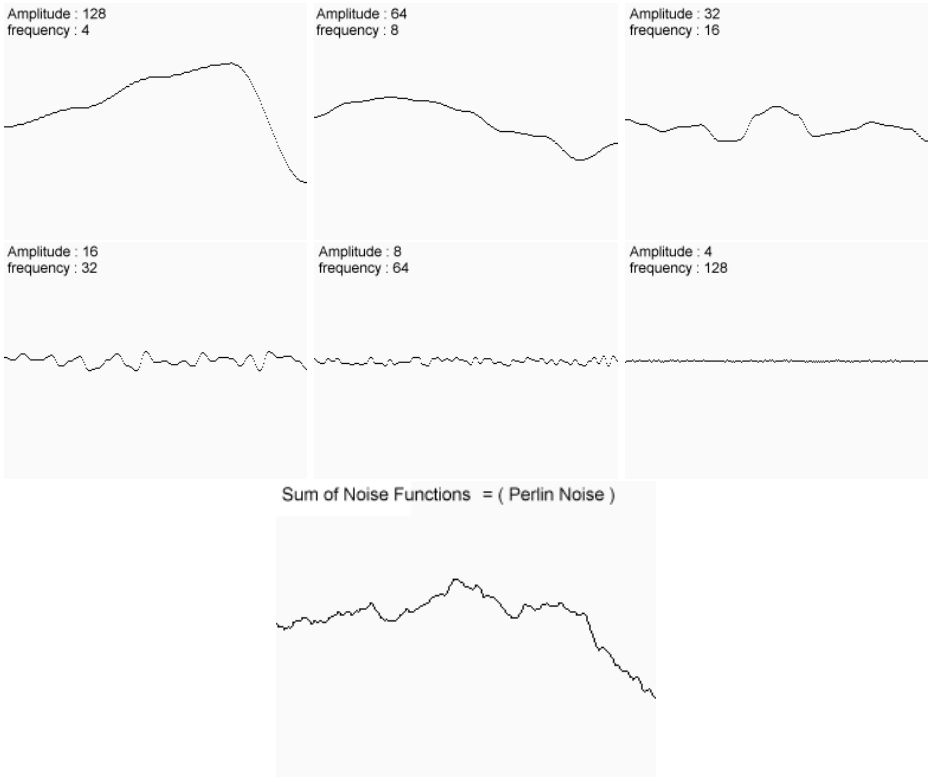


Figura 2.1: Exemplo 1D da composição do Ruído de Perlin a partir de 6 funções de frequências e amplitudes diferentes. Fonte: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Assim, o ruído de Perlin funciona recebendo parâmetros de entrada para gerar uma textura final. Todo o ruído de Perlin é construído tendo como base uma única função que,

utilizando um *grid* e e gradientes pseudorandômicos, retorna o valor de ruído do ponto de entrada (para maiores informações, o Anexo A apresenta a implementação original do ruído de Perlin). A partir deste ponto, o resultado final do ruído é consequência de vários parêmetros que, utilizando os valores obtidos por esta função, geram o ruído resultante. Os parâmetros que influenciam na geração do ruído de Perlin serão tratados no decorrer da seção.

Primeiramente, deve-se ressaltar que a criação do ruído é realizada por N iterações (onde N será definido pelo número de oitavas). Cada iteração irá gerar uma parte do ruído (semelhante a Figura 2.1) com frequência e amplitude específicas. A cada iteração, os valores de frequência e amplitude mudam de acordo com os parâmetros de entrada: lacunaridade e persistência.

Um dos parâmetros que influencia no resultado gerado é a persistência, que representa o valor que multiplicará a amplitude para cada iteração, de forma que modificando a persistência é possível controlar se funções com frequências mais altas irão ter maior ou menor influência no ruído gerado. Outro parâmetro que também influencia no ruído gerado é a lacunaridade: valor que representa o quão rápido a frequência irá aumentar para cada iteração sucessiva. Dessa forma, considerando a i-ésima iteração, pode-se calcular sua frequência pela Equação 2.1 e sua amplitude pela Equação 2.2.

$$frequencia = frequencia * lacunaridade^{i} \tag{2.1}$$

$$amplitude = amplitude * persistencia^{i} \tag{2.2}$$

Outro parâmetro tomado pelo ruído de Perlin é o número de oitavas que será utilizado. O número de oitavas representa a quantidade de vezes que a frequência será multiplicada por 2, ou seja, a quantidade de iterações do algoritmo. Na Figura 2.1 foram utilizadas 6 frequências distintas, o que representa que o número de oitavas e o número de iterações do algoritmo foram 6. Ao aumentar o número de oitavas é possível gerar texturas e ruídos com mais detalhes, ao custo de maior processamento. Pode-se perceber a diferença do ruído gerado utilizando 4 oitavas na Figura 2.2 e outro ruído gerado pela mesma função com 8 oitavas na Figura 2.3.

Dessa forma, tais conceitos de geração são extendidos para funções 2D, o que possibilita a geração de texturas pseudorandômicas (Figura 2.4). Como o ruído de Perlin é determinístico, a textura gerada uma vez pode ser reconstruída pela passagem dos mes-

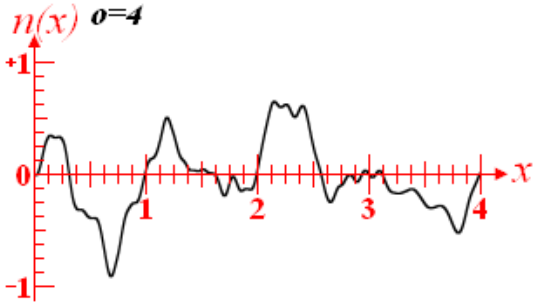


Figura 2.2: Ruído gerado somando 4 oitavas de uma função. Fonte: http://libnoise.sourceforge.net/glossary/index.html

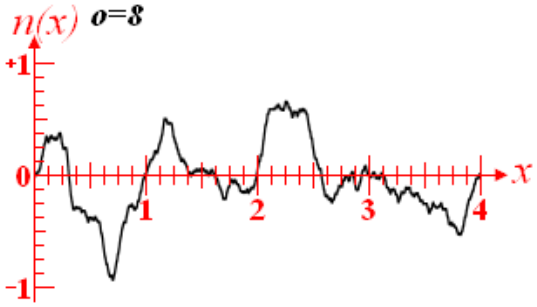


Figura 2.3: Ruído gerado somando 8 oitavas de uma função. Fonte: http://libnoise.sourceforge.net/glossary/index.html

mos parâmetros e, como o ruído de Perlin é contínuo em todo o espaço, pode-se gerar uma infinidade de texturas diferentes pela mudança de seus parâmetros.

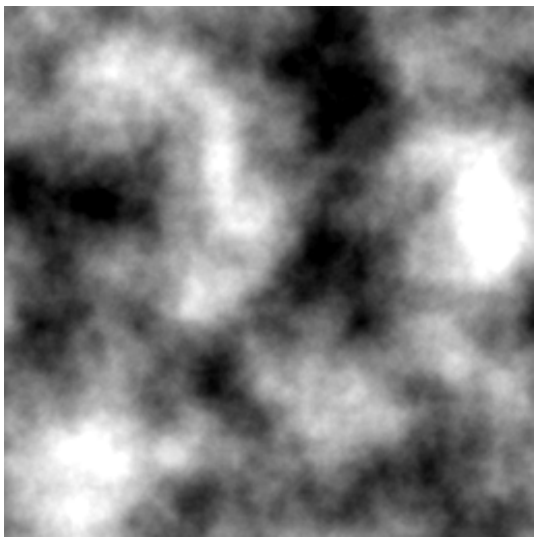


Figura 2.4: Textura 2D gerada pelo ruído de Perlin.

2.2 Mean shift

O *Mean shift* é uma técnica que estima o gradiente de uma função densidade de probabilidade através de um algoritmo iterativo não-paramétrico (FUKUNAGA; HOSTE-

TLER, 1975). Para melhor compreensão do funcionamento do algoritmo são necessários alguns conhecimentos relacionados a área de estatística e probabilidade, os quais são apresentados nas seções seguintes.

2.2.1 Função de Probabilidade

Na área da estatística, a função de probabilidade é uma função que associa cada ocorrência de uma variável aleatória discreta a sua respectiva probabilidade (JOHNSON; KOTZ; KEMP, 1993). Tome por exemplo uma variável aleatória discreta como sendo o "resultado de um dado", onde as possíveis ocorrências são 1, 2, 3, 4, 5 e 6. Considerando um dado não viciado, a função de probabilidade irá associar a cada uma destas ocorrências a probabilidade igual a 1/6 (Figura 2.5).

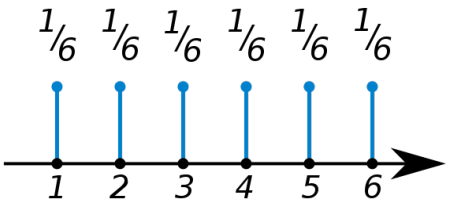


Figura 2.5: Função probabilística do resultado de um dado.

A função de probabilidade sempre possui valores não negativos e a soma total dos valores sempre é igual a 1, ou seja, a soma das probabilidades de todas as ocorrências deve resultar 100%. A Figura 2.6 representa uma função probabilística qualquer, onde é possível perceber que todos os valores são não-negativos e a soma total destes é 1.

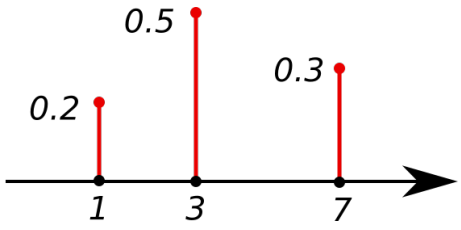


Figura 2.6: Função probabilística.

2.2.2 Função Densidade de Probabilidade

O conceito função densidade de probabilidade é análogo ao de função de probabilidade (Seção 2.2.1), exceto que enquanto a função de probabilidade associa a ocorrência de uma variável aleatória discreta, a função densidade de probabilidade associa uma variável aleatória contínua. Uma variável aleatória contínua, ao contrátio de uma discreta,

tem um número infinito de ocorrências, o que impossibilita o uso de uma função de probabilidade para associar uma probabilidade a cada ocorrência; dessa forma, a probabilidade de uma variável aleatória contínua não opera com valores pontuais, e sim com intervalos infinitesimais, como pode ser visto na Figura 2.7. De forma semelhante, a probabilidade de todo o espaço amostral de uma função densidade de probabilidade $f(x)$ é 1, conforme a Equação 2.3.

$$\int_{-\infty}^{\infty} f(x)dx = 1 \tag{2.3}$$

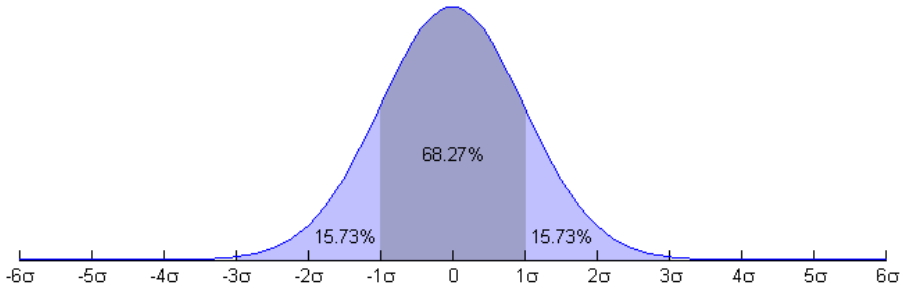


Figura 2.7: Função densidade de probabilidade de uma distribuição normal.

2.2.3 Gradiente da Função Densidade de Probabilidade

Em problemas de reconhecimento de padrões, pouca informação está disponível na forma da função densidade de probabilidade. Devido a esta falta de informação muitos algoritmos não-paramétricos foram desenvolvidos para estimar funções densidade de probabilidades, entre estes está o Mean shift (FUKUNAGA; HOSTETLER, 1975), o qual é uma técnica não-paramétrica para obter uma estimativa do gradiente de uma função densidade de probabilidade. No cálculo vetorial, o gradiente é a alteração de um valor por unidade de espaço, a definição do vetor gradiente é dada pela Equação 2.4.

$$grad\ f = < \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n} > \tag{2.4}$$

Ou seja, o Mean shift é utilizado para estimar a variação de uma função densidade de probabilidade.

2.2.4 Moda de uma Função

Neste mesmo trabalho (FUKUNAGA; HOSTETLER, 1975), Fukunaga apresentou o uso deste gradiente para encontrar as modas de uma determinada função. Da estatística, moda é o máximo valor local de uma função e é muito utilizado na captura de informações

sobre variáveis/populações randômicas. A moda não é necessariamente única, visto que dependendo da função escolhida esta pode possuir vários máximos locais. A Figura 2.8 ilustra como o gradiente calculado pelo Mean shift foi utilizado para encontrar as modas da função $p(x)$. Note que para qualquer valor de x é possível utilizar o gradiente para encontrar a moda de sua região.

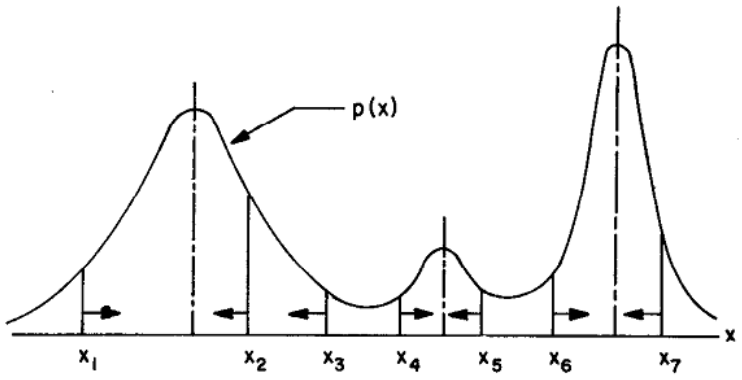


Figura 2.8: Utilização do gradiente para encontrar modas de uma função

Cada valor de x é associado a sua moda e valores que possuem a mesma moda pertencem a mesma região, ou seja, pertencem ao mesmo *cluster*. Dessa forma é possível segmentar a função em *clusters* que podem ser avaliados posteriormente.

2.2.5 Mean shift em processamento de imagens

Da mesma maneira que o Mean shift segmenta funções bidimensionais (Figura 2.8), este pode ser utilizado na segmentação de funções 3D, o que motivou seu uso na área de processamento de imagens em imagens multiespectrais. Segundo Comaniciu e Meer (1999), a filtragem com preservação de arestas em alta qualidade e segmentação de imagens podem ser obtidas aplicando o Mean shift. O método desenvolvido é baseado na mesma ideia base do Mean shift, fazendo com que uma janela fixa se desloque iterativamente para a região de maior concentração de pontos) - esta janela é representada por uma circunferência de raio h centrada no ponto em análise, onde o raio da circunferência é visto como um parâmetro do algoritmo: *bandwidth*. Esta técnica faz com que cada ponto no espaço seja processado pela função iterativa que irá levá-lo a máxima densidade local (seu centróide), que posteriormente é utilizada para segmentar a imagem em *clusters*.

Pela definição formal: seja $\{x_i\}_{i=1..n}$ um conjunto arbitrário de n pontos em um espaço Euclidiano d-dimensional R^d; a estimativa da densidade multivariada do kernel ob-

tida com o kernel $K(x)$ e com uma janela de raio h, computada no ponto x é definido pela Equação 2.5.

$$\hat{f}(x) = \frac{1}{nh^d} \sum_{i=1}^{n} K(\frac{x - x_i}{h}) \tag{2.5}$$

Dessa forma, o Mean shift itera sobre cada pixel calculando a área de maior concentração e desloca a janela a este ponto. Este processo é repetido iterativamente até não ocorrer alteração da área de concentração. A Figura 2.9 mostra como ocorre o processo de encontrar o centro de concentração de dois pixels diferentes. Assim, cada pixel estará ligado a um centróide, que define a qual *cluster* este pertence, ou seja, pixels que possuírem o mesmo centróide pertencerão ao mesmo *cluster*, o que significa que possuem características semelhantes.

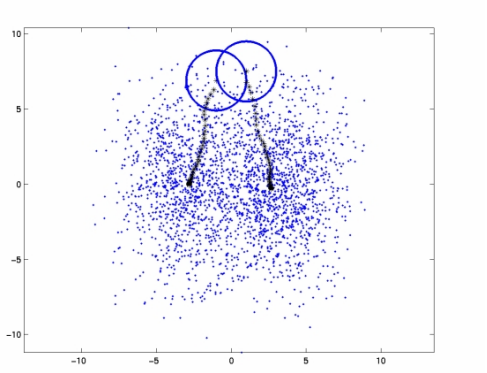


Figura 2.9: Funcionamento do Mean shift. Comparação entre dois pontos que iterativamente procuram seu centróide.

Logo, utilizando o Mean shift é possível segmentar imagens em *clusters* que contenham cores semelhantes. Portanto, caso as imagens representem diferentes características (como peso, altura e idade), é possível utilizar o Mean shift para agrupar conjuntos com características semelhantes (no caso de um *cluster* representar pessoas altas, magras e velhas e outro *cluster* que representa pessoas baixas, magras e jovens).

Deste modo, pode-se utilizar o Mean shift não apenas para segmentar uma única imagem, mas um conjunto de N imagens que representem diferentes características (como relevo de um terreno, densidade da vegetação, perigo associado a área). Entretanto o resultado gerado pelo Mean shift sempre será uma única imagem contendo a segmentação das imagens de entrada como um todo. A Figura 2.11 mostra a segmentação de duas imagens (presentes na Figura 2.10).

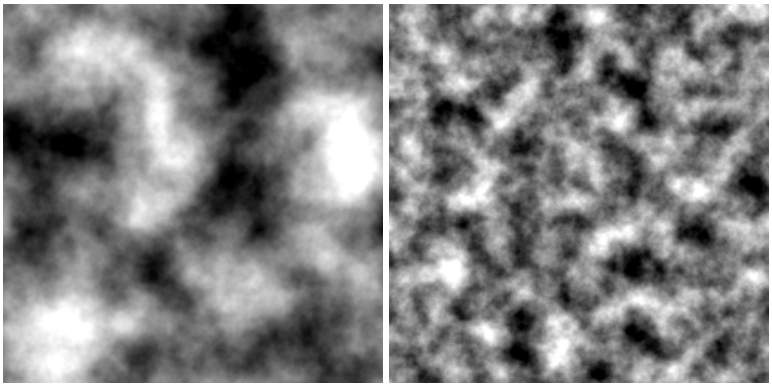


Figura 2.10: Imagens construídas utilizando diferentes parâmetros do ruído de Perlin.

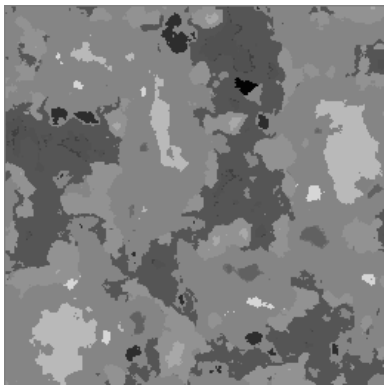


Figura 2.11: Resultado da segmentação utilizando o Mean shift nas imagens da Figura 2.10.

2.3 Algoritmo de Pathfinding

Algoritmos de *pathfinding* são utilizados para descobrir a rota de menor custo entre dois pontos (BOURG; SEEMANN, 2004). Esta área é baseada principalmente no algoritmo de Dijkstra (DIJKSTRA, 1959), o qual utiliza a abordagem gulosa para encontrar o caminho de menor custo entre dois nós de um grafo. Este algoritmo consiste em, dado um nó inicial, sempre analisar o nó de menor custo (o qual pode representar, por exemplo, a distância do nó em questão para com o nó inicial). Devido a este comportamento, o algoritmo de Dijkstra é conhecido como algoritmo guloso. Entretanto, o algoritmo de Dijkstra não é muito utilizado em jogos, pois este acaba expandindo uma quantidade de nós muito grande, o que faz com que este algoritmo não tenha um desempenho desejável para jogos que necessitam encontrar caminhos em tempo real.

2.3.1 Algoritmo de busca A*

O algoritmo A* (HART; NILSSON; RAPHAEL, 1968) é uma variação do algoritmo de Dijkstra onde a escolha do nó a ser analisado não é dada somente pelo custo deste ao

nó inicial, mas também é levada em conta uma estimativa de quão distante este nó está do objetivo. Esta estimativa faz com que o algoritmo A* seja capaz de dar uma direção e indicar quais nós devem ser avaliados, evitando que muitos nós que não levam ao objetivo sejam avaliados. Considerando que se deseja encontrar o menor caminho do nó A ao nó B de um grafo, o algoritmo A* seguirá os seguintes passos para realizar a busca:

1. Adiciona o nó de início A na **open list**. A **open list** representa a lista de nós que são candidatos a compor o caminho e que devem ser analisados. Esta lista é ordenada crescentemente a partir da função F. Esta função F é usada para definir a ordem que os nós serão avaliados; tradicionalmente o valor de F é dado pela Equação 2.6, onde G é o custo do nó inicial A ao nó em questão (calculado a partir do caminho gerado até este nó) e H é a estimativa do custo do nó em questão até o nó B (objetivo).

$$F = G + H \tag{2.6}$$

2. Considerando N o primeiro nó da **open list**, retira N da **open list** e adiciona N na **closed list** (lista que representa quais nós já foram visitados/analisados).

3. Verifica se N é o objetivo. Se sim, encerra algoritmo e o caminho é reconstruído a partir das informações dos nós pai.

4. Considerando o conjunto de nós que podem ser alcançados a partir de N, para cada nó T deste conjunto:

   Se T está na **closed list**, ignorá-lo.

   Se T está na **open list**, comparar se o novo caminho para o nó T, por N, é menor que o caminho já estabelecido para T na **open list**, caso o novo caminho tenha menor custo, remover o antigo nó da **open list** e adiciona o nó T com o novo caminho na **open list** com o nó N sendo o nó pai.

   Caso nenhuma das situações anteriores ocorra, adiciona o nó T à **open list** com o nó N sendo o nó pai.

5. Se **open list** está vazia, encerra algoritmo pois não existe caminho entre A e B; senão volte para o item 2.

O desempenho do algoritmo do A* é dependente de qual função é utilizada para estimar o custo entre dois nós (DECHTER; PEARL, 1985). Esta estimativa é calculada

utilizando uma função heurística, a qual não pode superestimar o custo ao objetivo, ou seja, o custo estimado pela heurística não pode ser maior que o real. Do contrário o algoritmo A* não garante um resultado ótimo, que é quando o A* realmente encontra o menor caminho. Dessa forma, a função heurística não somente direciona e determina o quão rápido o A* chegará a uma solução, mas também garante que o A* encontrará a melhor resposta se a função não for superestimada.

2.4 OpenCL e paralelização

O OpenCL(*Open Computing Language*) é uma arquitetura utilizada para escrever programas que funcionem em diferentes plataformas (CPUs, GPUs e outros processadores)(Apple Inc., 2008). O OpenCL permite a criação de programas paralelos, de forma a aumentar o desempenho de algoritmos com alto custo computacional. A ideia principal do OpenCL é poder, através de uma única linguagem, programar para diferentes plataformas de diferentes fornecedores (Intel, Nvidia, ATI, ARM).

Esta linguagem assemelha-se muito com CUDA (NVIDIA Corporation, 2007), onde o programa é escrito em um *kernel* (função que é executada no *device* do OpenCL). O Código 2.1 mostra um exemplo de função *kernel*. O OpenCL utiliza o modelo *host-device* para programar, onde o código do *host* é o executado normalmente e o *device* é o dispositivo (GPU por exemplo) que irá executar o *kernel* programado. Para a utilização do OpenCL, deve-se preocupar com a transferência de dados entre as memórias do *host* e *device*, visto que estes representam dispositivos diferentes.

```
__kernel void euclidean_distance(__global float *output,
    __global const float *dataset, const int numOfPoints, const
    int numOfDimensions, __global const float *currentKMean)
{
   int index = get_global_id(0);
   float sum = 0.f;
   int dimension;
   for(dimension = 0; dimension < numOfDimensions; dimension++){
      sum += pow((float)currentKMean[dimension] -
          (float)dataset[dimension * numOfPoints + index], 2);
   }
   output[index] = (float)sqrt((float)sum);
}
```

Código 2.1: *Kernel* de OpenCL que realiza o cálculo da distância euclidiana.

Neste trabalho, foi utilizado o OpenCL para paralelizar o algoritmo de segmentação

de imagens Mean shift, o qual fez com que o Mean shift tivesse um ganho de desempenho considerável (mostrando a versão paralela com OpenCL sendo 16 vezes mais rápida que a versão *serial*). A escolha do OpenCL justifica-se pela possibilidade de paralelizar o programa em GPU, fazendo com que a paralelização possa ser realizada sem limitações em GPUs de diferentes fabricantes.

2.5 OpenCV e processamento de imagens

O OpenCV(*Open Source Computer Vision Library*) é uma biblioteca focada no processamento de imagens (Intel Corporation, 1999). Esta biblioteca contém vários algoritmos clássicos de processamento de imagens, bem como possui um bom suporte para programas que necessitam carregar, armazenar e manipular imagens. Esta biblioteca oferece um conjunto de classes e funções que podem, por exemplo, tratar imagens como matrizes, facilitando a manipulação das mesmas no programa.

Na internet, é possível obter vários tutoriais e guias para utilização do OpenCV, o que facilita o aprendizado da biblioteca. Apesar de oferecer várias funções de alto nível para manipular imagens (como segmentação de imagens, detecção de faces, reconhecimento de padrões), a biblioteca foi utilizada neste trabalho com o intuito de facilitar a manipulação de imagens. A abstração oferecida pelo OpenCV para carregar e salvar imagens, bem como obter valores de *pixels* e aplicar máscaras facilitaram a implementação deste trabalho.

3 PROPOSIÇÃO DO TRABALHO

Este trabalho propõe uma técnica que possa gerar caminhos a partir de um conjunto de mapas de custo. Diferente de trabalhos já desenvolvidos (GALIN et al., 2010), onde cada mapa de custo recebe um peso referente a sua importância no custo da geração do caminho, este trabalho visa desenvolver uma técnica que leve em consideração o conjunto de características de cada ponto para o cálculo de seu custo.

Dessa forma, esta técnica proporcionará ao *game designer* um modo flexível de gerar caminhos, onde o cenário será dividido em regiões para melhor ajuste de pesos. Utilizando um algoritmo de segmentação, o conjunto de entrada será classificado em regiões com características semelhantes, tornando possível um ajuste de pesos mais detalhados do que somente um peso por mapa de entrada. Dessa forma, será possível identificar regiões específicas que devam representar um maior custo para o caminho, e ajustar o peso das mesmas sem modificar pesos de outras regiões. A seguir será mostrada uma visão mais técnica do programa proposto, mostrando como se pretende alcançar os resultados almejados.

A partir de mapas de custo de entrada é possível definir um espaço multiespectral, onde as características de cada ponto são representadas por um vetor N-dimensional. Isto é válido na medida em que as coordenadas cartesianas dos pontos em cada mapa correspondem à mesma posição espacial, considerando mapas de mesmo tamanho. Para segmentar estes mapas em regiões, um algoritmo de clusterização é utilizado, fazendo com que pontos com características semelhantes (onde cada mapa de custo representa uma característica) sejam agrupados em um mesmo *cluster*. Com as regiões segmentadas, é possível obter informações sobre cada região (como as médias da região) e mostrá-las ao usuário, para que este possa identificar quais características definem cada grupo, para então poder ajustar os pesos como desejar.

Uma vez que as regiões estão definidas e seus pesos determinados, é gerado um mapa de custo final, o qual representará o custo agregado de cada ponto. Este mapa possui apenas uma dimensão espectral, independente da quantidade de mapas de entrada. Estas etapas podem ser processadas em *offline* para um mesmo mapa e conjunto de custos. Desta forma, é possível utilizar um algoritmo de *pathfinding* para encontrar o caminho de menor custo entre dois pontos quaisquer em tempo real.

Um exemplo prático de problema em que esta técnica pode ser utilizada é: deseja-se fazer com que um personagem se mova do ponto A ao ponto B de um cenário, entretanto este personagem, por algum motivo, não pode passar por pântanos. Ao receber os mapas representando características (tais como relevo, vegetação e umidade, por exemplo), a técnica proposta utilizará o algoritmo de clusterização para agrupar áreas de características semelhantes. Ao saber as áreas segmentadas, é possível então avaliar quais destas se assemelham com um pântano (que pode ser determinada analisando as características como relevo e vegetação do ponto). Assim é atribuído um peso maior a estas regiões, devido ao fato deste tipo de região ser muito custosa para o personagem. Logo, o mapa de custo final gerado irá atribuir custos muito elevados em áreas indesejadas, fazendo com que ao rodar o A*, o caminho encontrado evite percorrer estas regiões. A Figura 3.1 ilustra o funcionamento proposto para criação do caminho.

As próximas seções explicam a proposta de geração das imagens de entrada (mapas de custo), a justificativa e importância do Mean shift para segmentação das imagens e a justificativa da utilização do algoritmo do A* para encontrar o caminho de menor custo.

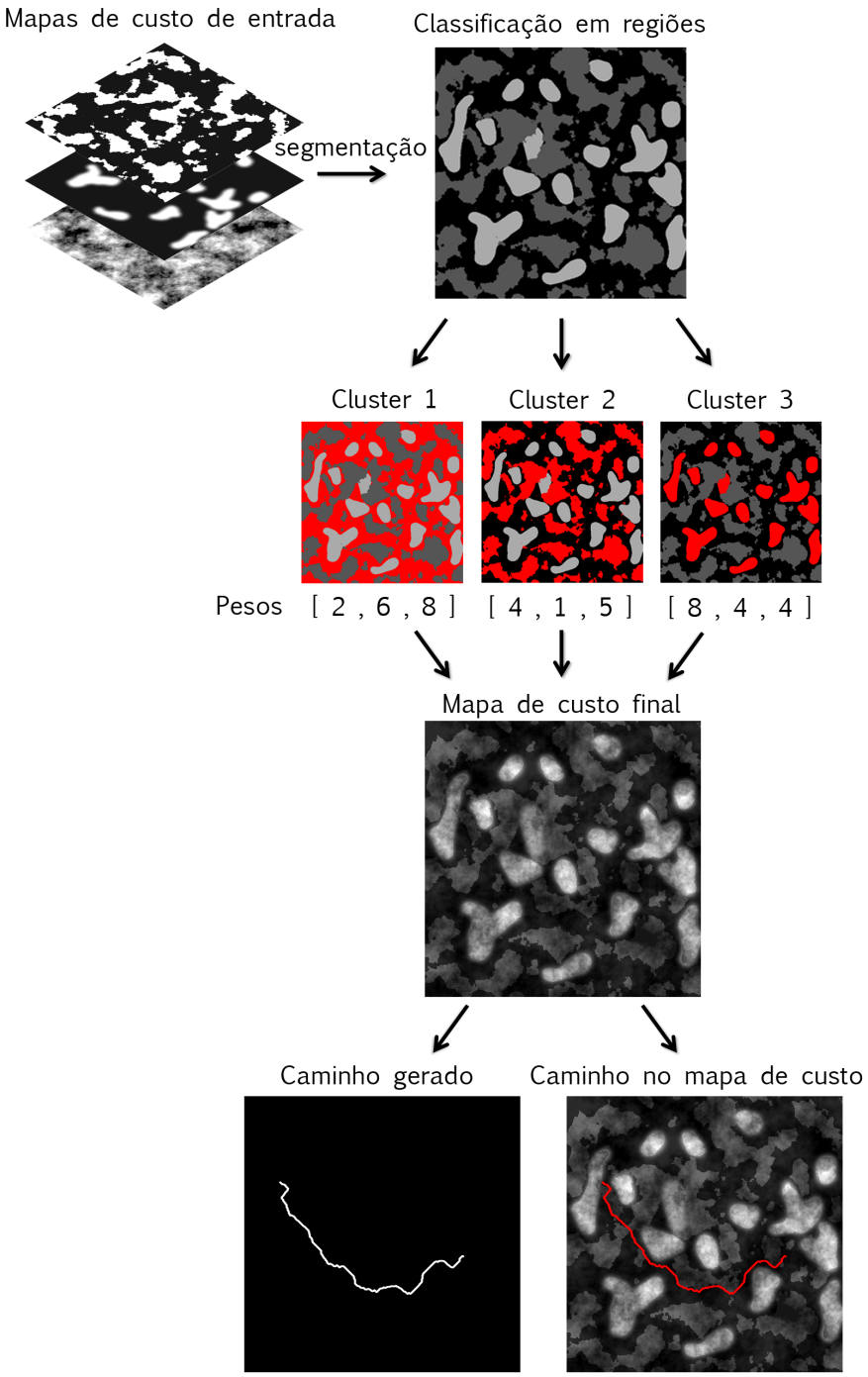


Figura 3.1: Funcionamento do sistema proposto.

3.1 Ruído de Perlin

O ruído de Perlin é o algoritmo de geração procedural escolhido para gerar os mapas de custo utilizados como entrada. Estes mapas de custo serão armazenados na forma de *heightmaps*, que são figuras em tons de cinza, as quais são utilizadas para representar diferentes características (como altura de um terreno, densidade de vegetação, nível

de umidade). Assim, o ruído de Perlin (que inicialmente foi criado para gerar texturas pseudorandômicas) é capaz de gerar diferentes *heightmaps* dependendo dos parâmetros de entrada.

O uso do ruído de Perlin é justificado pelo fato deste ser contínuo em todo o espaço, ou seja, esse nunca produzirá mapas com falhas bruscas (o que não condiz com a natureza caso estes mapas forem representar o relevo de um terreno, ou a densidade de vegetação). Além disso, o ruído de Perlin é pseudorandômico e isso faz com que, diferentemente de outros métodos de geração como fractais, os resultados gerados não contenham padrões de repetições (o que empobrece o conteúdo gerado).

Outra característica fundamental para a escolha do ruído de Perlin é que este é determinístico, fazendo com que seja possível gerar o mesmo mapa quando desejado, contanto que sejam utilizados os mesmos parâmetros de entrada. Uma vez gerados os mapas de custo, o trabalho propõe que estes *heightmaps* sejam a entrada de um algoritmo de segmentação, o qual agrupará pontos com características semelhantes em um mesmo *cluster*.

3.2 Mean shift

Para realizar a segmentação dos *heightmaps* de entrada em *clusters*, necessita-se de uma técnica de *Clustering*, a qual classifica grande quantidade de dados em diferentes categorias de forma não-supervisionada (JAIN; MURTY; FLYNN, 1999). Dentre as duas técnicas de *clustering* mais conhecidas estão o **k-means** e o **Mean shift**.

A principal diferença entre os dois algoritmos está na forma com que é realizada a segmentação (BLASIAK, 2007). Para segmentar uma imagem utilizando o k-means é escolhido (de maneira randômica ou com alguma heurística) um conjunto de pontos que são considerados segmentos. A partir destes pontos o algoritmo itera para agrupar as regiões semelhantes. O principal problema deste algoritmo é que o k-means necessita determinar o número de segmentos para iniciar o algoritmo, o que não é uma tarefa simples, já que o número de *clusters* varia de um conjunto para outro; ademais, dependendo da escolha dos pontos para se iniciar o algoritmo, o k-means pode obter resultados diferentes para o mesmo conjunto de dados.

O Mean shift, por outro lado, não tem necessidade de determinar o número de *clusters* e não necessita selecionar um conjunto de pixels para iniciar o algoritmo (visto que o mesmo calcula a moda para cada ponto em particular). Ou seja, para cada ponto será cal-

culado seu centróide (o qual define a qual *cluster* este pertence), de forma que o número de *clusters* dependerá da distribuição de pontos do conjunto. Por esta razão optou-se por utilizar o algoritmo Mean shift, visto que este atende melhor às necessidades de segmentação para a ideia proposta.

Com a definição do algoritmo de segmentação a ser utilizado, é possível a geração de um mapa de custo final. Assim, utilizando um algoritmo de *pathfinding* neste mapa de custo final, é possível encontrar o caminho de menor custo entre dois pontos.

3.3 Algoritmo A*

Dentre os algoritmos de *pathfinding* atuais, o A* é o mais utilizado em jogos, devido ao seu desempenho em encontrar o caminho de menor custo. O algoritmo do A* acaba tendo melhor desempenho que os demais algoritmos pelo fato de utilizar uma heurística para decidir quais nós devem ser analisados. Dessa forma, essa heurística é de fundamental importância no funcionamento do A*, visto que ela controla o quão rápido o algoritmo encontra a solução, além de garantir que a solução encontrada é a solução ótima (caso seja usada uma função heurística que não superestime o custo ao objetivo).

A heurística a ser utilizada neste trabalho é a distância euclidiana (FABBRI et al., 2008). A distância euclidiana é a distância absoluta entre dois pontos no espaço euclidiano, e foi escolhida por fornecer uma boa estimativa da distância entre dois pontos em um mapa de custo. Esta heurística é admissível, ou seja, nunca superestima o custo para chegar ao objetivo, o que garante uma solução ótima ao A*.

4 DESENVOLVIMENTO

Neste capítulo é tratado o desenvolvimento do sistema proposto pelo trabalho. A implementação foi dividida em três módulos para melhor controlar o desenvolvimento do projeto. A primeira preocupação foi a de gerar os *heightmaps* (mapas de custo), que são utilizados como entrada para criação do caminho. Para tanto, o primeiro módulo implementado consiste em utilizar o ruído de Perlin para gerar os *heightmaps* (usados para representar características do cenário 3D). O segundo módulo engloba a implementação do algoritmo de segmentação Mean shift, que foi paralelizado utilizando o OpenCL para paralelização. O terceiro módulo é referente ao algoritmo A*, responsável por encontrar o caminho de menor custo entre dois pontos dado um mapa de custo. Este A* foi modificado, com o uso de matrizes auxiliares, para otimizar o desempenho e para facilitar o uso posterior no momento de unir os módulos.

4.1 Geração de heightmaps

Este módulo foca na geração de *heightmaps* a partir do ruído de Perlin. Este ruído é a base para a criação de *heightmaps* que representem características (as quais serão utilizadas para geração do mapa de custo final).

Para a implementação do ruído de Perlin foi utilizada uma biblioteca de geração de ruído: libnoise (DAVIES; BEVINS, 2004). A libnoise é uma biblioteca C++ *opensource* de geração de ruído coerente (*coherent noise*). A libnoise pode gerar ruído de Perlin, ruído multifractal e outros; os geradores de ruídos são encapsulados em classes chamadas de *noise modules*. A libnoise disponibiliza diferentes tipos de módulos de ruído (*noise modules*), os quais podem ser combinados e modificados para gerar ruídos complexos.

A libnoise somente gera valores de ruído coerente, não sendo capaz de gerar imagens. A libnoise possui algoritmos de interpolação, visto que muitos ruídos só existem para

valores inteiros ou de maneira discreta. Para poder gerar imagens a partir da libnoise foi utilizada uma outra biblioteca *opensource*: noiseutils. Esta biblioteca possui diversas funções que fazem uso da libnoise para geração de imagens e *heightmaps* de maneira eficaz.

Dessa forma, utilizando a libnoise para obter valores do ruído de Perlin e a noiseutils para geração das imagens, foi possível implementar um módulo genérico que pudesse gerar *heightmaps* de acordo com os parâmetros de entrada. O Código 4.1 mostra um exempo de código, onde é criado um *noise module* que representa o ruído que é utilizado e seus parâmetros são determinados a partir de uma interface gráfica; posteriormente são utilizadas funções da noiseutils para geração da imagem a partir do ruído criado. A Figura 4.1 é o diagrama das principais classes implementadas e utilizadas neste módulo.

```
module::Perlin myModule;
myModule.SetFrequency( mInterface->getFrequency() );
myModule.SetLacunarity( mInterface->getLacunarity() );
myModule.SetOctaveCount( mInterface->getOctaves() );
myModule.SetPersistence( mInterface->getPersistence() );
myModule.SetSeed( mInterface->getSeed() );

utils::NoiseMap heightMap;
utils::NoiseMapBuilderPlane heightMapBuilder;
heightMapBuilder.SetSourceModule(myModule);
heightMapBuilder.SetDestNoiseMap(heightMap);
heightMapBuilder.setDestSize(IMAGE_WIDTH, IMAGE_HEIGHT);
heightMapBuilder.SetBounds( mInterface->getminX(),
   mInterface->getmaxX(),
               mInterface->getminY(), mInterface->getmaxY());
heightMapBuilder.Build();
```

Código 4.1: Exemplo de código utilizando a libnoise e noiseutils.

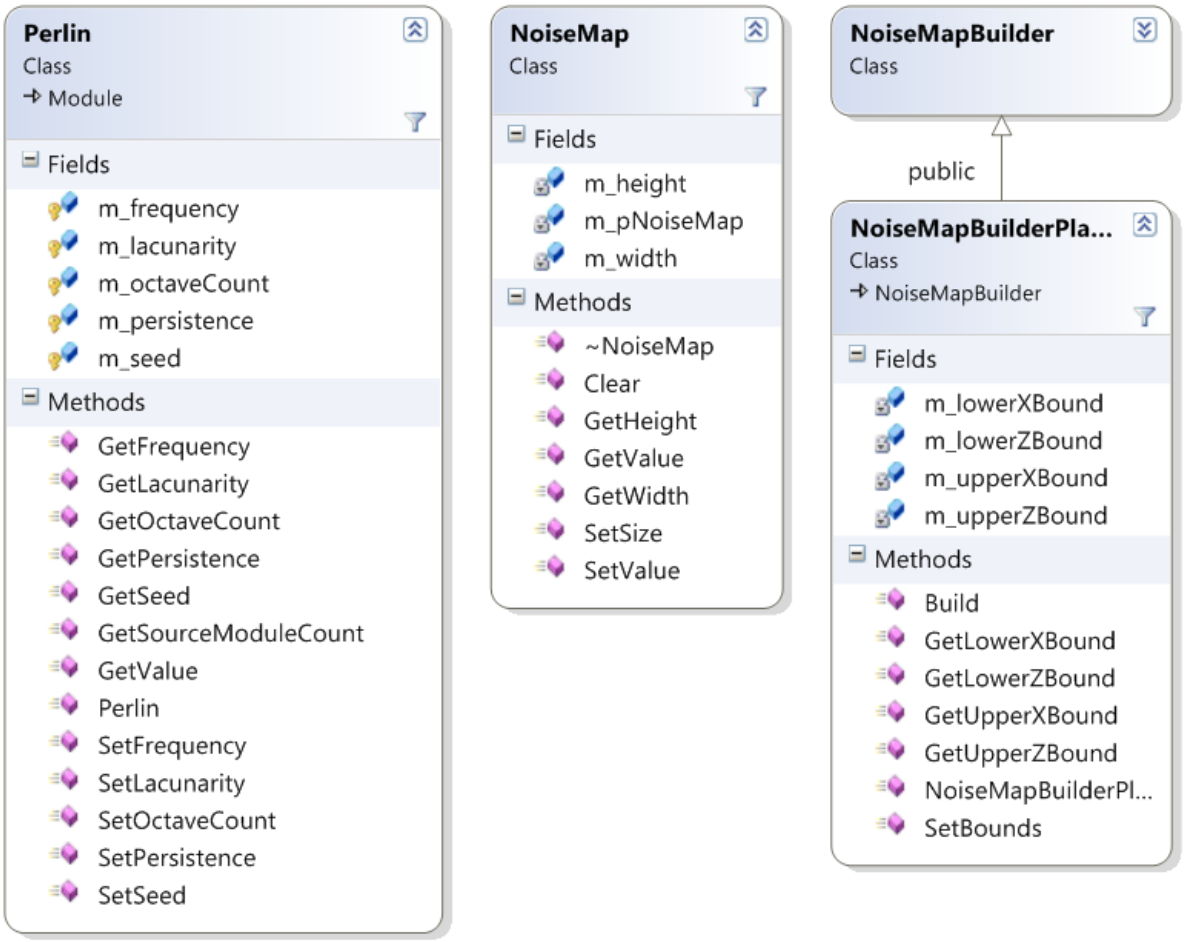


Figura 4.1: Diagrama de classes do módulo de geração de *heightmaps*.

4.2 Classificação de regiões

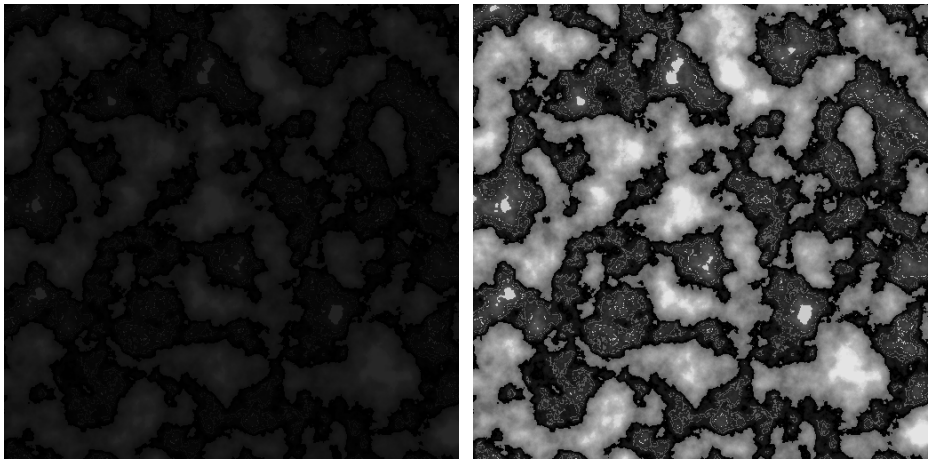
O segundo módulo implementado consiste em utilizar o algoritmo do Mean shift para segmentar o conjunto de imagens de entrada em regiões com características semelhantes. Dessa forma, a partir de uma implementação base do Mean shift desenvolvida por alunos do LaCA, foi possível a reformulação desta para utilizar neste trabalho. O algoritmo do Mean shift foi modificado, fazendo com que aceitasse um conjunto variado de imagens de entrada; pequenos erros na implementação base foram corrigidos e o algoritmo foi otimizado.

Uma otimização para o Mean shift foi implementada. Esta modificação consiste em, ao invés de calcular e buscar pelo centróide de cada *pixel* da imagem, a função utilizada para encontrar o centróide de um *pixel* também determina o centróide de outros. Esta otimização pode ser pensada da seguinte forma: quando o algoritmo de Mean shift executa a função para achar o centróide do ponto A, este acaba passando por vários pontos

(devido ao fato deste algoritmo ser iterativo) antes de chegar ao resultado; pelo conceito do Mean shift, estes pontos analisados no processo de descobrir o centróide do ponto A irão pertencer ao mesmo *cluster* do ponto A, visto que encontrarão o mesmo centróide. Logo, utilizando-se desta ideia, os centróides destes pontos são atualizados com o mesmo centróide relativo ao ponto A, fazendo com que não seja necessário calcular o centróide destes pontos novamente.

Testes realizados mostraram que o algoritmo Mean shift tornava-se muito lento na medida em que o conjunto de entrada aumenta. Assim, em um trabalho conjunto realizado no LaCA, o qual acabou se transformando em um trabalho de graduação (SCHARDONG, 2011), foi utilizado o OpenCL para paralelizar uma pequena parte de sua implementação. A parte escolhida para ser paralelizada foi a do cálculo da distância euclidiana. Este cálculo é usado no Mean shift para analisar a distância de cada ponto da população para o ponto que está em análise. É possível perceber que, conforme o conjunto de entrada aumenta, esta etapa específica da implementação acaba se tornando mais pesada; com o OpenCL, este cálculo foi paralelizado e obteve-se ganhos consideráveis de desempenho (testes realizados pelo trabalho de Schardong (SCHARDONG, 2011) mostram situações de um ganho que diminuiu em 16 vezes o tempo para execução do Mean shift).

Também foi desenvolvida uma classe interface para facilitar o uso do algoritmo de Mean shift, fazendo com que o programador possa facilmente adicionar imagens ao conjunto de entrada, rodar o Mean shift e obter a imagem resultante que representa a segmentação do conjunto de entrada. Esta classe efetua automaticamente as operações de inicialização do OpenCL, fazendo com que o uso da técnica do Mean shift consista basicamente de carregar as imagens desejadas, rodar o Mean shift e obter a resposta. Esta interface também calcula a média de cada *cluster*, de modo que para cada *cluster* é armazenado um vetor com a média de cada uma das imagens de entrada. Também são armazenadas duas imagens pelo sistema, uma onde cada pixel representa exatamente o valor do *cluster* a que pertence e outra que utiliza pseudocores para facilitar a visualização dos *clusters* (no caso de resultado com 3 *clusters*, a primeira imagem irá gerar uma imagem com valores de 0-2 enquanto que a segunda utilizará pseudocores como 0,127 e 255 para representar os *clusters*). A Figura 4.2 mostra a diferença das duas imagens armazenadas. A Figura 4.3 é o diagrama das classes implementadas para realizar a segmentação das imagens.



(a) *Clusters* indexados (b) Pseudocores para cada *cluster*

Figura 4.2: Comparação de uma segmentação em 32 *clusters*, a primeira imagem com cada *cluster* representando seu índice e outra utilizando pseudocores.

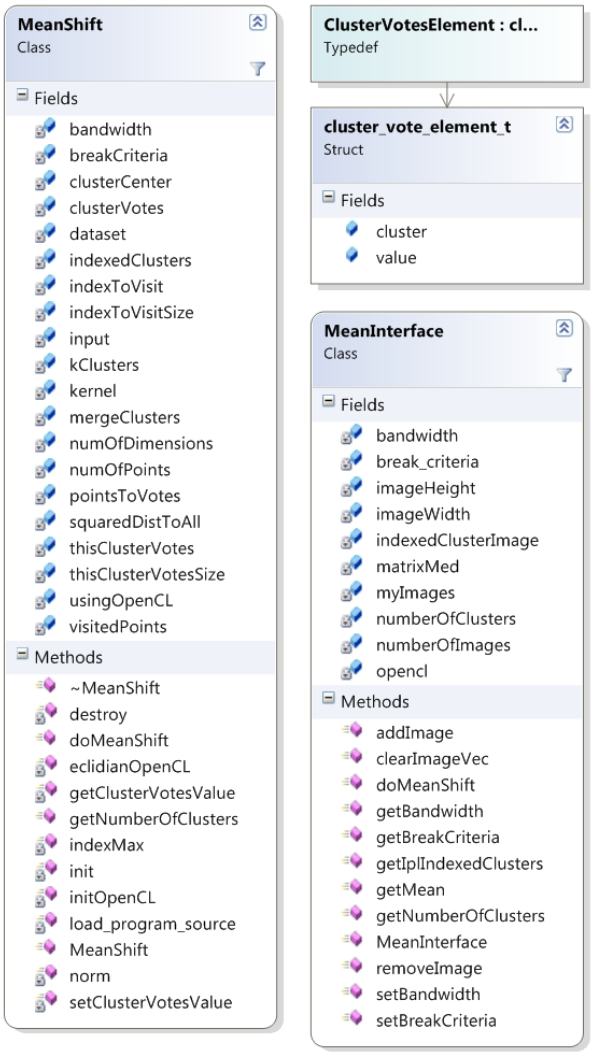


Figura 4.3: Diagrama de classes do módulo de classificação das regiões.

4.3 Algoritmo de pathfinding

A implementação do algoritmo A* partiu do presuposto de que o algoritmo seria utilizado em imagens (ou estruturas) que pudessem ser representadas em matrizes. A decisão de implementar o A* desta forma é o de poder otimizar seu desempenho, de forma que, apesar de demandar mais memória para alocar várias matrizes auxiliares na implementação, o acesso a dados fosse direto, poupando cálculos de indexação ou tempo de busca em listas e vetores. Assim, o A* implementado leva em consideração que estará percorrendo um plano, onde irá sempre avaliar os 8 vizinhos ao seu redor.

O algoritmo A* foi implementado de maneira que a função que atribui custos para cada ponto possa ser genérica. Desta forma, o custo de cada ponto no plano é determinado por um objeto que contém o cálculo do custo. Esta classe pode ser extendida para a criação de um novo objeto, fazendo com que o cálculo do custo possa ser modificado de acordo com o desejado. O resultado do algoritmo de A* é um vetor dos pontos no plano que representa o caminho entre os pontos de início e fim. A Figura 4.4 é o diagrama das classes implementadas para gerar o caminho de menor custo.

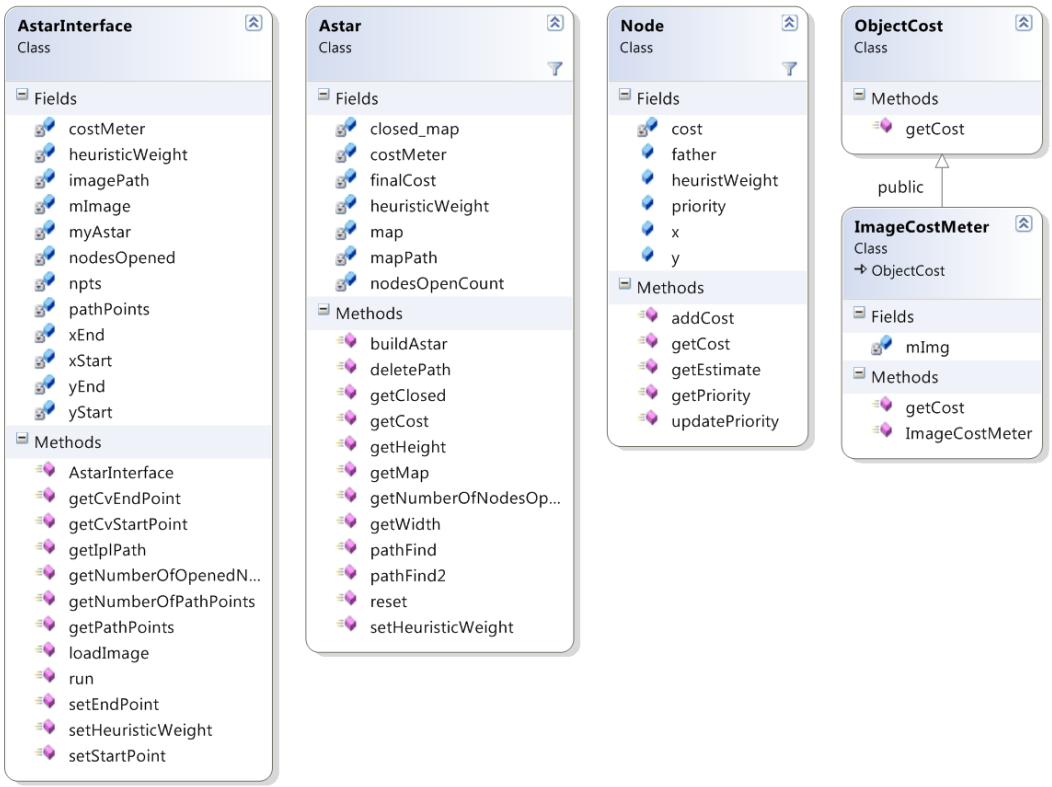


Figura 4.4: Diagrama de classes do módulo de geração do caminho.

4.4 Integração dos módulos

Com os módulos individuais desenvolvidos e devidamente testados, o próximo passo tornou-se a integração destes. Como já explicado no Capítulo 3, os módulos são organizados de maneira que o primeiro módulo (Seção 4.1) possibilitará a geração de *heightmaps* a partir de um conjunto de parâmetros de entrada, o segundo módulo (Seção 4.2) classifica o conjunto de imagens de entrada em regiões de características semelhantes, e o terceiro módulo (Seção 4.3) gera o caminho a partir do mapa de custo final. A geração deste mapa de custo final é detalhada nas seções seguintes, juntamente com a interface gráfica.

Assim, a etapa de integração dos módulos foi realizada sem grandes problemas, visto que os módulos eram genéricos e robustos. Durante a integração, o uso do OpenCV para armazenar e manipular imagens facilitou a interligação dos módulos, visto que todos estes trabalham diretamente com imagens. Para completar a comunicação entre os módulos foi construída uma interface gráfica, que é detalhada a seguir.

4.5 Construção de uma GUI (*Graphical User Interface*)

A criação de uma interface gráfica se fez necessária tanto para facilitar a entrada de dados no programa, quanto para mostrar os resultados gerados em tempo real. Para a construção da mesma foi utilizado uma API para construção de GUIs: o SCV. O SCV (*Simple Components for Visual*) é uma API desenvolvida sobre o OpenGL que fornece uma grande quantidade de componentes gráficos para construção de uma interface gráfica (HENZ et al., 2010) (LARA PAHINS et al., 2011). Esta API possibilitou a construção de uma interface intuitiva, onde cada etapa do funcionamento do programa está sequencialmente organizada em uma aba.

Utilizando os componentes do SCV e a funcionalidade de carregar imagens na interface gráfica, foi possível fazer uma aplicação que funcionasse em tempo real, de forma que a mudança dos parâmetros e dos dados de entrada atualizem o resultado e mostrem-no ao usuário. A interface gráfica também facilitou determinar o peso de cada imagem de entrada em cada um dos *clusters*, os quais são usados para a criação do mapa de custo final.

4.6 Geração do mapa de custo final

O mapa de custo final é construído a partir dos *heightmaps* de entrada e dos valores de pesos determinados. A ideia é que, após a classificação da entrada em regiões de características semelhantes, cada região possuirá um vetor de pesos, onde cada posição do vetor corresponde a uma imagem de entrada. Deste modo, o mapa de custo final é construído analisando cada *pixel* da imagem, verificando a qual *cluster* o *pixel* pertence e, utilizando o vetor de pesos respectivo, realizando o somatório do valor do *pixel* de cada *heightmap* multiplicado pelo seu respectivo peso.

O resultado é armazenado em uma imagem com valores do tipo *float 32bits*, o que possibilita armazenar uma gama maior de valores se comparado a imagens no formato *8bits*. Inicialmente foi pensado em normalizar os valores, entretanto isso acarretaria em perda de precisão no momento de construção do caminho. Apesar deste tipo de imagem não ter sido testado no algoritmo de *pathfinding* implementado, extendendo a classe que possui a função de custo de cada ponto, foi possível adaptar sem dificuldades este formato. A normalização dos valores entre 0-255 foi realizada para facilitar a visualização do mapa de custo na interface gráfica, o que gerou uma imagem com pseudocores que expressam uma ideia do mapa de custo final para quem esteja utilizando o programa.

4.7 Funcionamento do sistema

As seções anteriores mostraram a implementação do sistema, detalhando os módulos criados e as ferramentas e ideias utilizadas. Esta seção tem por objetivo detalhar o funcionamento do sistema como um todo, mostrando as funcionalidades oferecidas, bem como os passos para gerar o resultado. O funcionamento será explicado a partir das funcionalidades oferecidas através da interface gráfica construída.

A Figura 4.5 mostra a parte da interface responsável pela geração procedural de *heightmaps* e importação destes e outras imagens para o programa. Pode-se perceber que a interface possibilita uma fácil modificação dos parâmetros e a visualização da imagem gerada pelo ruído de Perlin. A imagem gerada pode ser ser salva para ser posteriormente utilizada na construção do caminho. Também é possível importar imagens geradas pelo usuário (esboço de imagens criadas manualmente; ou imagens geradas a partir de algoritmos de processamento de imagem), de maneira a proporcionar uma maior flexibilidade ao usuário na entrada de imagens para o programa.

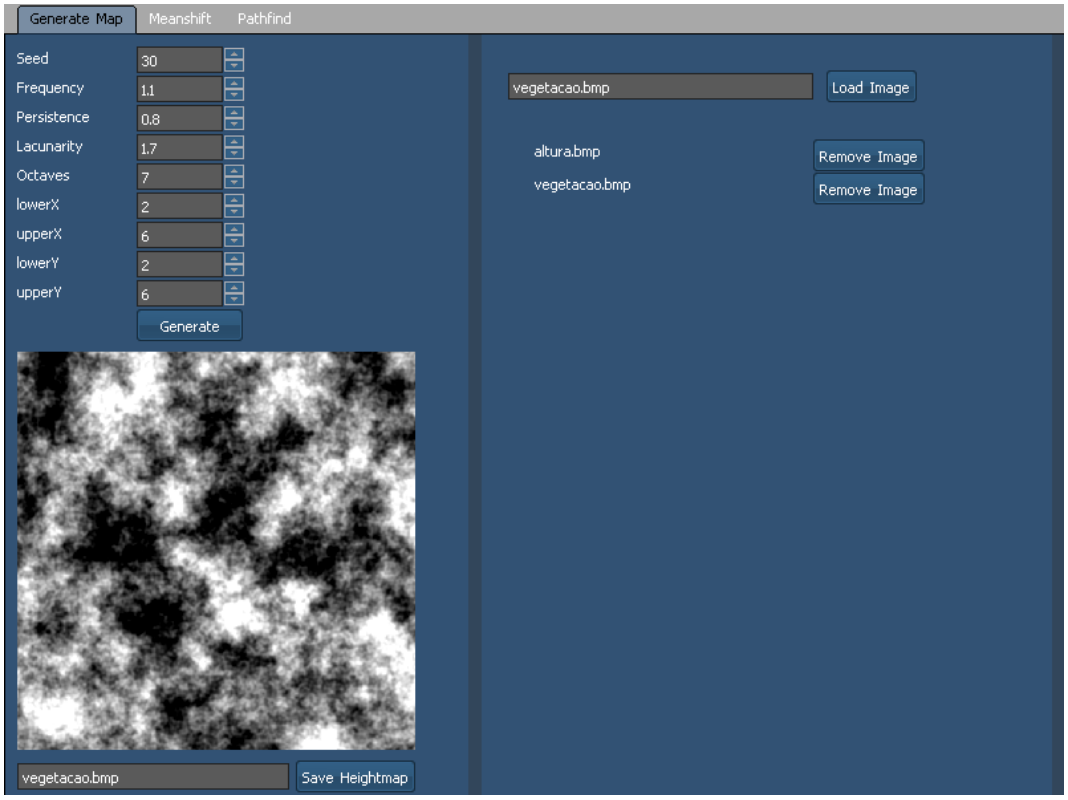


Figura 4.5: Interface do programa responsável pela criação de *heightmaps* com o ruído de Perlin e carregar imagens de entrada.

Após carregar as imagens desejadas, o programa apresenta uma interface gráfica para classificação das regiões de características semelhantes. A Figura 4.6 representa a interface gráfica oferecida para mudança de parâmetros do algoritmo de segmentação Mean shift (*bandwidth* e *break criteria*) e para executar a classificação do conjunto de entrada. Ao segmentar o conjunto em regiões de características semelhantes, é possível modificar os pesos de cada imagem de entrada para o *cluster* selecionado. O programa mostra em vermelho qual o *cluster* selecionado para indicar ao usuário a região que este representa, bem como mostra a média dos valores de cada imagem em cada *cluster*.

Nesta mesma etapa é construído o mapa final de custo, o qual é criado a partir dos pesos atribuídos pelo usuário. A interface mostra uma imagem (com pseudocores) que representa o mapa final de custo criado, de forma a fazer com que o usuário possa interativamente visualizar o mapa final gerado e determinar os melhores pesos. É importante ressaltar que o real mapa de custo final está armazenado em uma matriz e que a imagem mostrada na interface é uma normalização do mapa de custo final original.

Finalmente é realizada a construção do caminho a partir do mapa de custo final. Determinando dois pontos, o programa exibe o caminho de menor custo encontrado, o qual

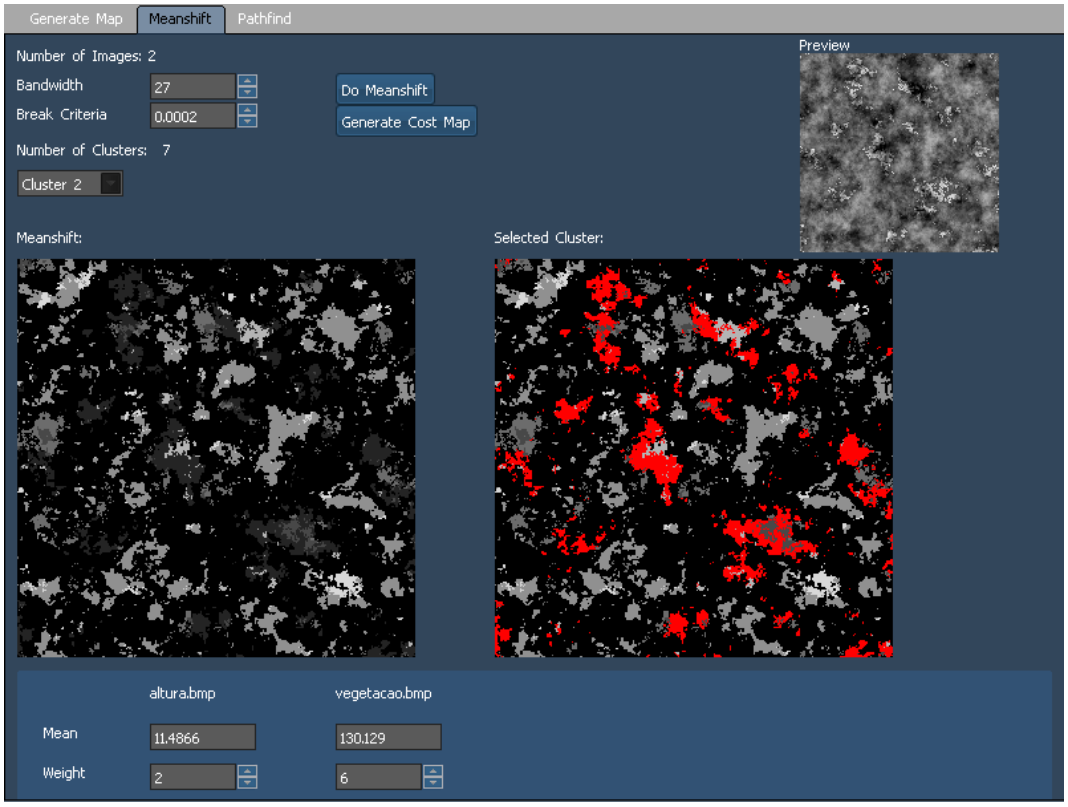


Figura 4.6: Interface responsável pela classificação de regiões e construção do mapa de custo final a partir dos pesos especificados.

pode ser exportado para ser utilizado posteriormente pelo usuário (Figura 4.7). A GUI traz a possibilidade de mudar o peso da heurística do A*, que pode ser utilizada para aumentar o desempenho do A* em imagens muito grandes (entretanto arriscando não obter o resultado ótimo).

A Seção 5 expõe um cenário de teste onde o sistema é utilizado. Nesta pode-se visualizar um exemplo real e prático de como utilizar a interface oferecida pelo programa e como os resultados gerados podem ser utilizados.

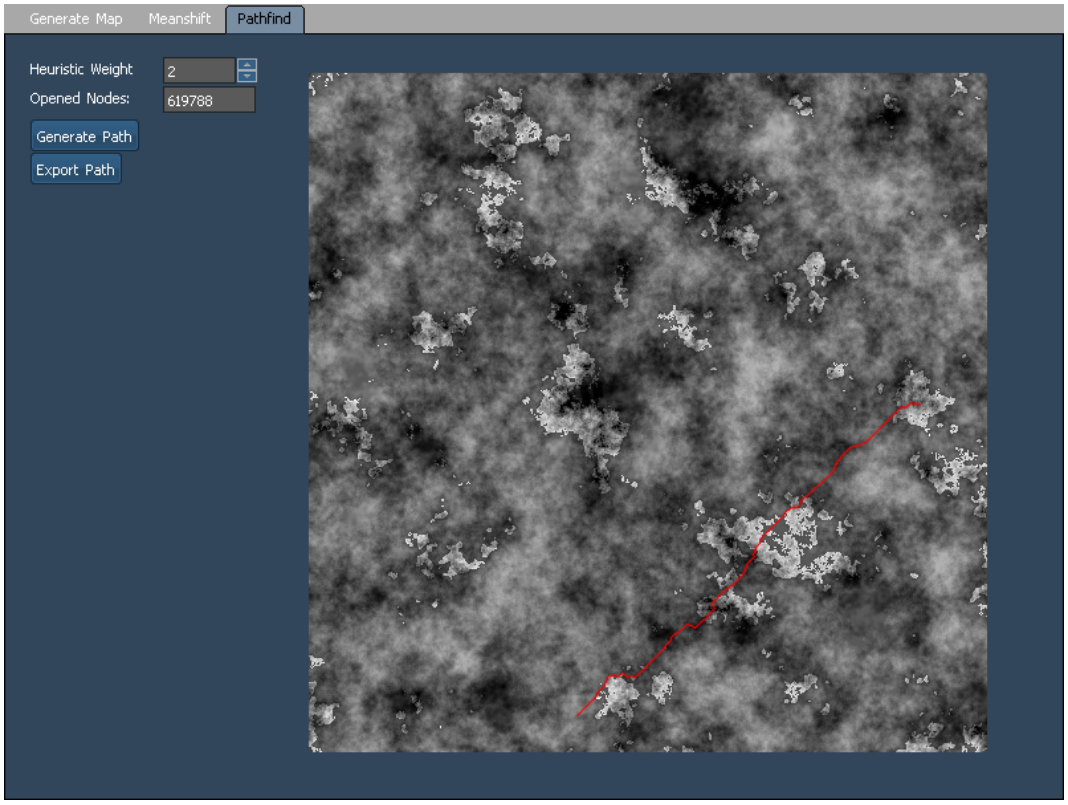


Figura 4.7: Interface responsável pela construção do caminho a partir dos pontos determinados.

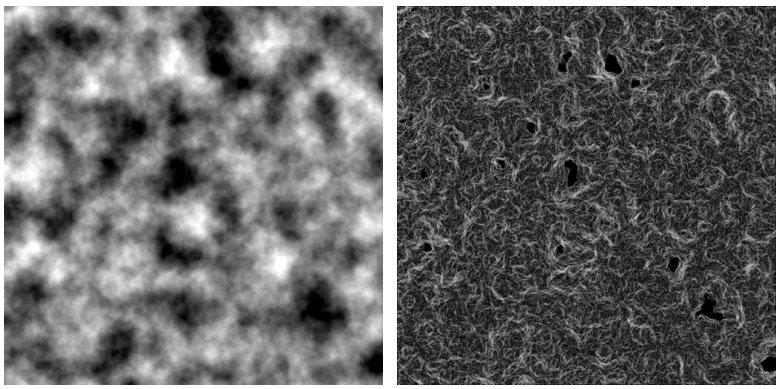
5 CENÁRIO DE TESTE

Esta seção tem por objetivo mostrar como o sistema pode ser utilizado na prática e quais funcionalidades são oferecidas. Considerando que se deseja criar um cenário 3D com as seguintes características:

- a altura do terreno do cenário é representada por meio de uma imagem;
- a densidade de vegetação do cenário é definida por um *heightmap*;
- é definido uma altura para ser considerado o nível da água do cenário.

Desejando que o caminho gerado procure percorrer a menor inclinação do terreno, e que este não passe por baixo d'água, será explicado como as funcionalidades do sistema gerado podem ser usadas para alcançar este resultado.

Utilizando a funcionalidade do sistema de criar imagens proceduralmente, foi possível a criação de um *heightmap* que represente a altura do terreno em questão (Figura 5.1a). Para que a inclinação do terreno possa ser incluída na construção do mapa de custo final, o gradiente da imagem de altura foi calculado e salvo em uma imagem (Figura 5.1b).



(a) Mapa de altura (b) Mapa de inclinação

Figura 5.1: Imagens de entrada representando altura e inclinação do terreno.

Para evitar que o caminho que se deseja gerar passe por áreas com água, é necessário criar uma imagem que possa representar as áreas que estão abaixo da altura determinada como sendo o nível de água. Para tanto, usando algoritmos de processamento de imagem (*thresholding*), foi possível isolar estas áreas em uma imagem binária. A Figura 5.2 mostra a imagem resultante, onde partes em branco representam regiões com água e as áreas em preto, regiões com altura acima do nível da água.

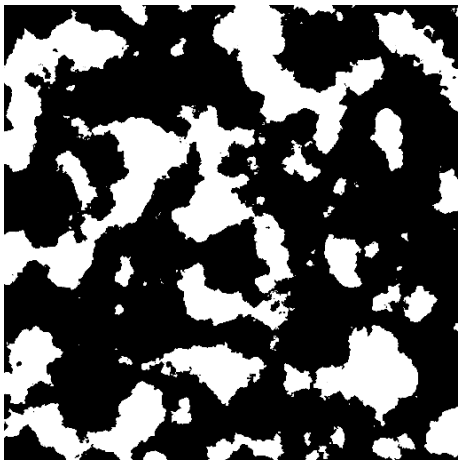


Figura 5.2: Imagem binária representando a presença de água no cenário.

O *heightmap* que representa a densidade de vegetação do cenário foi criado manualmente (Figura 5.3). Isto mostra que tanto imagens criadas pelo sistema desenvolvido (através de ruído de Perlin), quanto imagens criadas pelo usuário (manualmente ou através da manipulação de imagens) podem ser utilizadas igualmente. Também é importante notar que algumas imagens criadas (mapa de inclinação e mapa de água) não representam elementos em um cenário, mas são necessárias para construir o mapa de custo final que será utilizado para geração do caminho.

Após preparar as imagens de entrada, deve-se carregá-las para o programa poder classificá-las em regiões de características semelhantes. A mudança dos parâmetros do Mean shift na interface influencia diretamente na quantidade de *clusters* gerados, podendo controlar o quão específica deve ser esta classificação. A Figura 5.4 mostra o uso de um *bandwidth* de valor 60, o que resultou na classificação das imagens de entrada em 4 regiões distintas.

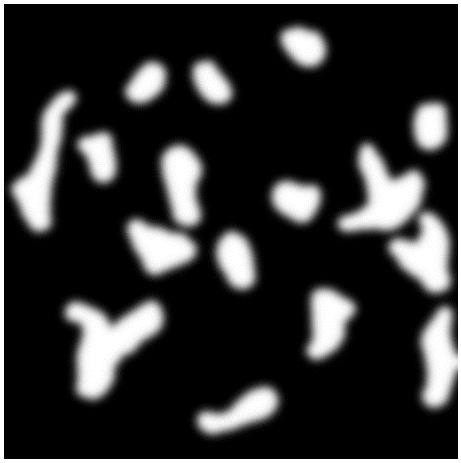


Figura 5.3: Imagem gerada manualmente representando a densidade de vegetação do cenário.

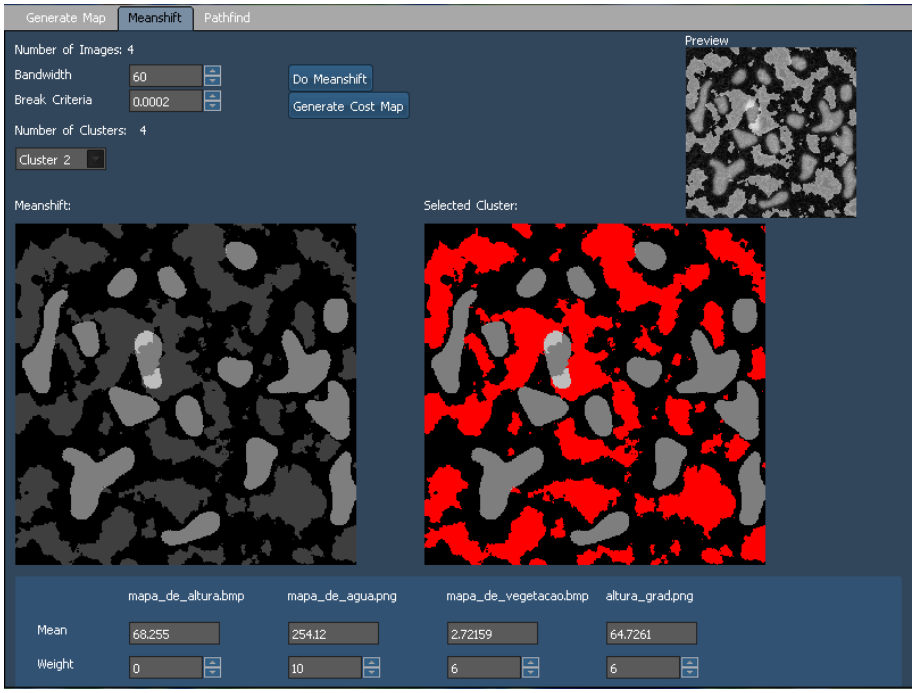


Figura 5.4: Classificação da entrada em 4 regiões distintas.

Realizada a classificação, pode-se, através da interface, ajustar os pesos que cada imagem terá em cada um dos *cluster*. A Figura 5.4 exemplifica um ajuste de pesos do *cluster* 2 (uma região que olhando os valores médios percebe-se representar uma região com baixa altura, baixa inclinação, baixa vegetação e que apresenta água). Como deseja-se que o caminho não percorra regiões com água, deve-se atribuir um alto custo ao mapa de água. O fato da média da imagem representando água ser alta não significa que todos os pontos desta região estão debaixo d'água, portanto foram atribuídos pesos para a vegetação e inclinação do terreno (a altura do terreno foi zerada, pois não se deseja utilizá-la no

cálculo do mapa de custo final).

Com os pesos ajustados, o mapa de custo final é construído para ser utilizado como entrada do algoritmo A*. Definindo o ponto de início e o ponto final desejado, o sistema gera o caminho de menor custo baseando-se no mapa de custo final. O caminho pode então ser exportado (Figura 5.5) em uma imagem para ser usado posteriormente.

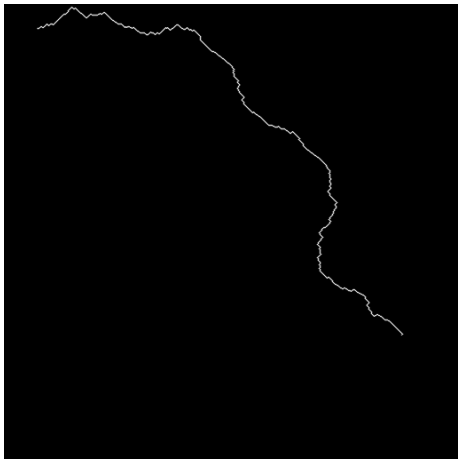


Figura 5.5: Imagem da exportação do caminho gerado.

Utilizando o Ogitor Scene Builder (MOEN, 2009), um plugin para a *engine* gráfica OGRE (Ogre Team, 2002), foi possível fazer a representação 3D dos *heightmaps* criados neste cenário. O plugin possibilitou carregar o mapa de altura para definir o terreno e determinar um valor como nível da água do cenário. A Figura 5.6 mostra o cenário criado pelo Ogitor, onde a textura de grama representa a vegetação do cenário (conforme o *heightmap* da Figura 5.3) e o caminho gerado (Figura 5.5) foi destacado com uma textura vermelha para facilitar a visualização.

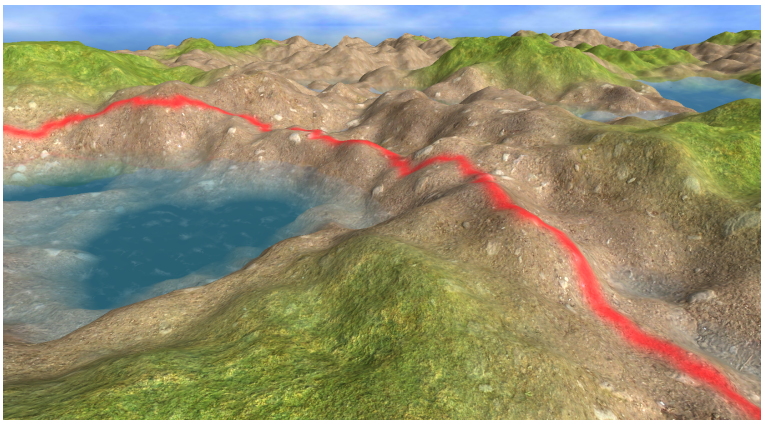


Figura 5.6: Parte do cenário criado com o plugin Ogitor.

Percebe-se que o caminho gerado não passa pela água, resultado dos altos pesos que foram atribuídos para a construção do mapa de custo final. Pode-se notar também que a construção do caminho levou em conta a inclinação do terreno, de forma que o caminho não passa por elevações acentuadas. A Figura 5.7 mostra grande parte do cenário e dessa forma é possível notar a relação dos *heightmaps* criados, bem como as imagens utilizadas para geração do caminho.

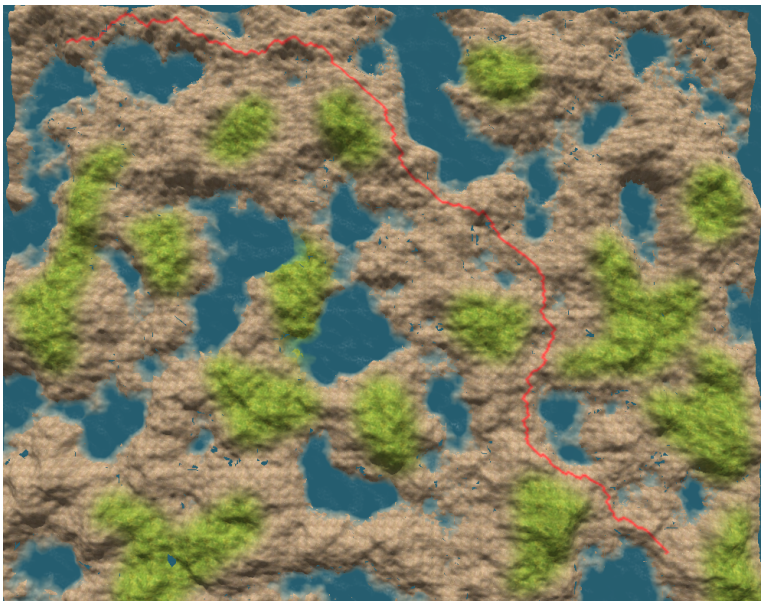


Figura 5.7: Cenário criado com o plugin Ogitor.

Esta seção mostrou como o caminho gerado pelo sistema pode ser utilizado. Este foi um exemplo muito simples de como se pode utilizar o caminho gerado para enriquecer cenários, controlando através dos pesos o caminho gerado. Como já mencionado, o sistema preocupa-se unicamente com a criação do caminho a partir do mapa de custo final, gerando uma imagem que indica os pontos do caminho; o uso de texturas, plugins e tudo relacionado à arte do cenário 3D deve ser tratado por quem estiver desenvolvendo o cenário.

6 CONCLUSÃO

O objetivo deste trabalho foi a elaboração de um sistema que fosse capaz de gerar caminhos de maneira automática através da entrada de *heightmaps* que representam características de um cenário. Diferente de trabalhos já propostos, este sistema segmenta o cenário em regiões com características semelhantes, proporcionando uma maneira flexível de analisar cada região para melhor controle do caminho gerado.

Mesmo que a geração do caminho ainda tenha de ser supervisionada (pela necessidade de controlar regiões classificadas e ajuste de pesos), o objetivo de elaborar um sistema para facilitar a geração de caminhos foi alcançado. Os caminhos gerados pelo sistema podem ser utilizados para enriquecer cenários (como foi mostrado na Seção 5) ou para movimentar personagens em jogos.

A escolha do ruído de Perlin para geração de imagens de entrada do sistema possibilitou a criação de *heightmaps* com características naturais. A utilização do algoritmo de Mean shift para classificação mostrou-se uma solução adequada para o problema de classificar regiões com características semelhantes. Um problema deste algoritmo de segmentação é sua complexidade computacional: o algoritmo acaba se tornando muito lento na medida em que o número e tamanho de imagens de entrada aumentam.

Os resultados obtidos se mostraram satisfatórios na medida em que, dado um conjunto de imagens de entrada, o sistema classifica o cenário em regiões, para serem pesadas e então gerar o caminho desejado. Este trabalho traz consigo uma novo método de construção de caminhos, possibilitando um controle muito mais detalhado e minuscioso do caminho a ser gerado.

6.1 Trabalhos futuros

Com o sistema desenvolvido, é necessária a preocupação com questões de desempenho. Como mencionado, o Mean shift é um método de segmentação muito complexo do ponto de vista computacional, o que acaba comprometendo o desempenho do programa em situações com alto volume de características a ser analisado. Pretende-se procurar meios de otimizar ainda mais o algoritmo Mean shift para que este não comprometa o desempenho do sistema.

Está sendo estudada a ideia de fazer com que o sistema proporcione a funcionalidade de utilizar um grupo de imagens para a classificação das regiões e um grupo distinto de imagens para a geração do mapa de custo final. Isto traria mais liberdade e flexibilidade, visto que seria possível especificar quais características que seriam utilizadas para classificar as regiões, diferenciando-as das características que seriam utilizadas no cálculo de custo do caminho gerado.

Outra ideia é a utilização de pontos pré-definidos pelo usuário para a construção do mapa de custo final. Ou seja, ao invés do usuário ter que ajustar os pesos para cada uma das regiões encontradas pelo Mean shift, o usuário define pesos para diferentes combinações de regiões. Como exemplo, será determinado um vetor de pesos X_a para a região A e outro vetor de pesos X_b para a região B; ao classificar as regiões do cenário, os pesos de cada região serão determinados pela similaridade que cada uma possui com as combinações previamente estabelecidas (região A ou região B). Esta ideia ainda está sendo amadurecida para possibilitar uma maneira diferente de ajustar os pesos das regiões.

REFERÊNCIAS

Apple Inc., K. **Open Computing Language - OpenCL**. 2008.

Bay 12 Games. **Dwarf Fortress**. 2006.

BLASIAK, A. **A Comparison of Image Segmentation Methods**. 2007.

BOURG, D. M.; SEEMANN, G. **AI for Game Developers**. [S.l.]: O'Reilly Media, Inc., 2004.

BRUNETON, E.; NEYRET, F. Real-time rendering and editing of vector-based terrains. In: EUROGRAPHICS, 2008. **Anais...** [S.l.: s.n.], 2008.

COMANICIU, D.; MEER, P. Mean Shift Analysis and Applications. In: INTERNATIONAL CONFERENCE ON COMPUTER VISION-VOLUME 2 - VOLUME 2, 1999, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1999. (ICCV '99).

DAVIES, K.; BEVINS, J. **libnoise**. 2004.

DECHTER, R.; PEARL, J. Generalized best-first search strategies and the optimality of A*. **J. ACM**, New York, NY, USA, v.32, p.505–536, July 1985.

DIJKSTRA, E. A note on two problems in connexion with graphs. **Numerische mathematik**, [S.l.], v.1, n.1, p.269–271, 1959.

EBERT, D. S.; MUSGRAVE, F. K.; PEACHEY, D.; PERLIN, K.; WORLEY, S. **Texturing and Modeling**: a procedural approach. 3rd.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

FABBRI, R.; COSTA, L. D. F.; TORELLI, J. C.; BRUNO, O. M. 2D Euclidean distance transform algorithms: a comparative survey. **ACM Comput. Surv.**, New York, NY, USA, v.40, p.2:1–2:44, February 2008.

FUKUNAGA, K.; HOSTETLER, L. The estimation of the gradient of a density function, with applications in pattern recognition. **Information Theory, IEEE Transactions on**, [S.l.], v.21, n.1, p.32 – 40, jan 1975.

GALIN, E.; PEYTAVIE, A.; MARéCHAL, N.; GUéRIN, E. Procedural Generation of Roads. **Computer Graphics Forum (Proceedings of Eurographics)**, [S.l.], v.29, n.2, p.429–438, 2010.

HART, P.; NILSSON, N.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. **Systems Science and Cybernetics, IEEE Transactions on**, [S.l.], v.4, n.2, p.100 –107, july 1968.

HENZ, B.; GOTTIN, V. M.; LIMBERGER, F. A.; SPERONI, E. A.; LARA PAHINS, C. A. de; POZZER, C. T. Uma API Free Software para Composição de GUI em Aplicativos Gráficos. **XI Workshop sobre Software Livre, FISL**, [S.l.], 2010.

Intel Corporation. **Open Source Computer Vision Library - OpenCV**. 1999.

JAIN, A. K.; MURTY, M. N.; FLYNN, P. J. Data clustering: a review. **ACM Comput. Surv.**, New York, NY, USA, v.31, p.264–323, September 1999.

JOHNSON, N. L.; KOTZ, S.; KEMP, A. W. **Univariate Discrete Distributions (Wiley Series in Probability and Statistics)**. [S.l.]: Wiley-Interscience, 1993.

LANDIM, W. **O tamanho da indústria dos vídeo games**. [S.l.]: TecMundo, 2011.

LARA PAHINS, C. A. de; LIMBERGER, F. A.; CAMPAGNOLO, L. Q.; SCHLESNER, Y. K.; HENZ, B. **SCV - Simple Components for Visual**. 2011.

LEWIS, J.-P. Texture synthesis for digital painting. **SIGGRAPH Comput. Graph.**, New York, NY, USA, v.18, p.245–252, January 1984.

MAXIS. **Spore**. 2006.

MCCRAE, J.; SINGH, K. K.: sketch-based path design. In: IN PROC. GRAPHICS INTERFACE (2009), 2009. **Anais...** [S.l.: s.n.], 2009. p.95–102.

MOEN, J. **Ogitor Scene Builder**. 2009.

NVIDIA Corporation. **NVIDIA CUDA Compute Unified Device Architecture Programming Guide**. [S.l.]: NVIDIA Corporation, 2007.

Ogre Team. **Open Source 3D Graphics Engine**. 2002.

PERLIN, K. An image synthesizer. In: SIGGRAPH, 1985. **Anais...** [S.l.: s.n.], 1985. p.287–296.

PERSSON, M. **Minecraft**. 2009.

SCHARDONG, G. G. **Acelerando Meanshift com OpenCL**. Trabalho de Graduação do Curso de Ciência da Computação. 2011.

Six Times Nothing. **Unity Tool**: road/path tool. 2011.

APÊNDICE A IMPLEMENTAÇÃO ORIGINAL DO RUÍDO DE PERLIN

Fonte: http://cs.nyu.edu/~perlin/doc/oscar.html.

```
/* coherent noise function over 1, 2 or 3 dimensions */
/* (copyright Ken Perlin) */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define B 0x100
#define BM 0xff

#define N 0x1000
#define NP 12 /* 2^N */
#define NM 0xfff

static p[B + B + 2];
static float g3[B + B + 2][3];
static float g2[B + B + 2][2];
static float g1[B + B + 2];
static start = 1;

static void init(void);

#define s_curve(t) ( t * t * (3. - 2. * t) )

#define lerp(t, a, b) ( a + t * (b - a) )

#define setup(i,b0,b1,r0,r1)\
  t = vec[i] + N;\
  b0 = ((int)t) & BM;\
  b1 = (b0+1) & BM;\
  r0 = t - (int)t;\
  r1 = r0 - 1.;

double noise1(double arg)
{
  int bx0, bx1;
  float rx0, rx1, sx, t, u, v, vec[1];

  vec[0] = arg;
  if (start) {
    start = 0;
    init();
  }

  setup(0, bx0,bx1, rx0,rx1);

  sx = s_curve(rx0);

  u = rx0 * g1[ p[ bx0 ] ];
  v = rx1 * g1[ p[ bx1 ] ];
```

```
  return lerp(sx, u, v);
}

float noise2(float vec[2])
{
  int bx0, bx1, by0, by1, b00, b10, b01, b11;
  float rx0, rx1, ry0, ry1, *q, sx, sy, a, b, t, u, v;
  register i, j;

  if (start) {
    start = 0;
    init();
  }

  setup(0, bx0,bx1, rx0,rx1);
  setup(1, by0,by1, ry0,ry1);

  i = p[ bx0 ];
  j = p[ bx1 ];

  b00 = p[ i + by0 ];
  b10 = p[ j + by0 ];
  b01 = p[ i + by1 ];
  b11 = p[ j + by1 ];

  sx = s_curve(rx0);
  sy = s_curve(ry0);

#define at2(rx,ry) ( rx * q[0] + ry * q[1] )

  q = g2[ b00 ] ; u = at2(rx0,ry0);
  q = g2[ b10 ] ; v = at2(rx1,ry0);
  a = lerp(sx, u, v);

  q = g2[ b01 ] ; u = at2(rx0,ry1);
  q = g2[ b11 ] ; v = at2(rx1,ry1);
  b = lerp(sx, u, v);

  return lerp(sy, a, b);
}

float noise3(float vec[3])
{
  int bx0, bx1, by0, by1, bz0, bz1, b00, b10, b01, b11;
  float rx0, rx1, ry0, ry1, rz0, rz1, *q, sy, sz, a, b, c, d, t, u, v;
  register i, j;

  if (start) {
    start = 0;
    init();
  }

  setup(0, bx0,bx1, rx0,rx1);
  setup(1, by0,by1, ry0,ry1);
  setup(2, bz0,bz1, rz0,rz1);

  i = p[ bx0 ];
  j = p[ bx1 ];

  b00 = p[ i + by0 ];
  b10 = p[ j + by0 ];
  b01 = p[ i + by1 ];
  b11 = p[ j + by1 ];

  t  = s_curve(rx0);
  sy = s_curve(ry0);
  sz = s_curve(rz0);

#define at3(rx,ry,rz) ( rx * q[0] + ry * q[1] + rz * q[2] )

  q = g3[ b00 + bz0 ] ; u = at3(rx0,ry0,rz0);
  q = g3[ b10 + bz0 ] ; v = at3(rx1,ry0,rz0);
  a = lerp(t, u, v);
```

```
  q = g3[ b01 + bz0 ] ; u = at3(rx0,ry1,rz0);
  q = g3[ b11 + bz0 ] ; v = at3(rx1,ry1,rz0);
  b = lerp(t, u, v);

  c = lerp(sy, a, b);

  q = g3[ b00 + bz1 ] ; u = at3(rx0,ry0,rz1);
  q = g3[ b10 + bz1 ] ; v = at3(rx1,ry0,rz1);
  a = lerp(t, u, v);

  q = g3[ b01 + bz1 ] ; u = at3(rx0,ry1,rz1);
  q = g3[ b11 + bz1 ] ; v = at3(rx1,ry1,rz1);
  b = lerp(t, u, v);

  d = lerp(sy, a, b);

  return lerp(sz, c, d);
}

static void normalize2(float v[2])
{
  float s;

  s = sqrt(v[0] * v[0] + v[1] * v[1]);
  v[0] = v[0] / s;
  v[1] = v[1] / s;
}

static void normalize3(float v[3])
{
  float s;

  s = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
  v[0] = v[0] / s;
  v[1] = v[1] / s;
  v[2] = v[2] / s;
}

static void init(void)
{
  int i, j, k;

  for (i = 0 ; i < B ; i++) {
    p[i] = i;

    g1[i] = (float)((random() % (B + B)) - B) / B;

    for (j = 0 ; j < 2 ; j++)
      g2[i][j] = (float)((random() % (B + B)) - B) / B;
    normalize2(g2[i]);

    for (j = 0 ; j < 3 ; j++)
      g3[i][j] = (float)((random() % (B + B)) - B) / B;
    normalize3(g3[i]);
  }

  while (--i) {
    k = p[i];
    p[i] = p[j = random() % B];
    p[j] = k;
  }

  for (i = 0 ; i < B + 2 ; i++) {
    p[B + i] = p[i];
    g1[B + i] = g1[i];
    for (j = 0 ; j < 2 ; j++)
      g2[B + i][j] = g2[i][j];
    for (j = 0 ; j < 3 ; j++)
      g3[B + i][j] = g3[i][j];
  }
}
```