UNIVERSIDADE FEDERAL DE
SANTA CATARINA

COMPARAÇÃO PRATICA ENTRE SIMPLEX NOISE E PERLIN NOISE PARA GERAÇÃO DE TERRENOS PROCEDURAIS

RAFAEL GUEDES MARTINS

TRABALHO DE CONCLUSÃO DO CURSO DE SISTEMAS DE INFORMAÇÃO

Orientador: José Eduardo De Lucca

Florianópolis, 08 de julho de 2012.

RAFAEL GUEDES MARTINS

COMPARAÇÃO PRATICA ENTRE SIMPLEX NOISE E PERLIN NOISE PARA GERAÇÃO DE TERRENO PROCEDURAL

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Bacharel do curso de Sistemas de Informação da Universidade Federal de Santa Catarina.

Prof. orientador: José Eduardo De Lucca

Florianópolis, 2012

Autoria: Rafael Guedes Martins
Título: Comparação pratica entre Simplex Noise e Perlin Noise para geração de terreno procedural

Trabalho de conclusão de curso apresentado como parte das atividades para obtenção do título de Bacharel do curso de Sistemas de Informação da Universidade Federal de Santa Catarina.

Banca examinadora:

_____

Prof. M.Sc. José Eduardo De Lucca
Orientador

_____

Prof. Dr. João Candido Dovicchi
Membro da banca avaliadora

_____

Dennis Kerr Coelho
Membro da banca avaliadora

Resumo

Funções de ruido procedural são usadas vastamente em Computação Gráfica, desde a renderização de filmes a videogames. A habilidade de adicionar detalhes complexos e complicados a baixos custos de memoria é uma de suas principais características. Este Trabalho de Conclusão de Curso é motivado pela importância do ruido procedural na computação e o uso abrangente na industria. O objetivo é demonstrar as diferenças de implementação e desempenho entre dois dos algorítimos de geração de ruido famosos, O Perlin Noise e o Simplex Noise. Ambos feitos pelo mesmo autor Ken Perlin. Neste Trabalho será explicado breviamente o que são funções de ruido, como normalmente funcionam, e quais vantagens estas costumam ter, dando enfase a Perlin Noise e Simplex Noise. Será explicado a metodologia para a realização dos experimentos e da implementação, quais ferramentas serão usadas e porque. Finalmente será apresentado os resultados dos experimentos.

Palavras-chave: Ken Perlin, Perlin Noise, Simplex Noise, Noise, Ruido, Geração Procedural.

Abstract

Procedural Noise Functions are widely used in computer graphics, be that on film rendering or game making. The ability to quickly and efficiently add rich and interesting details at low prices is noise's greatest characteristic. This work is motivated by the importance of Noise in the Computer Graphics industry. The goal is to demonstrate the differences in implementation and performance between two of the most famous Noise Functions, the Perlin Noise and the Simplex Noise, Both made by Ken Perlin. In this work there will be a brief explanation on how noise work and which advantages it usually brings with it. Focusing mostly on Perlin Noise and Simplex Noise. It will be explained how the work will be done and which frameworks or tolls will be used, and why. And finally, the results from the work will be shown.

Keywords: Ken Perlin, Perlin Noise, Simplex Noise, Noise, Procedural Generation.

Lista de abreviaturas e siglas

NPC – Non Player Character (Personagem não controlado por jogador).

MMORPG – Massively Multiplayer Online Role Playing Game (Jogo de Interpretação de Personagem Online Massivamente Multijogador).

PRNG – Pseudo Random Number Generator (Gerador de Números Pseudo Aleatórios).

IDE – Integrated Development Environment (Ambiente de Desenvolvimento Integrado).

Lista de Figuras

Lista de Tabelas

Sumario

1 Introdução

Constantes evoluções na capacidade de processar e visualizar aplicações 3D causaram um aumento na exigência do publico em geral quanto a qualidade visual em diversas áreas, em especial filmes e jogos.

Adicionar detalhes ricos e interessantes sempre foi uma tarefa árdua para a área de computação gráfica, seja para filmes ou jogos. O aumento da exigência de qualidade aumentou a procura por ferramentas ou meios de criar ou adicionar tais detalhes. Uma das alternativas disponíveis foi a Geração Procedural.

A Geração Procedural pode criar medias rapidamente e com grande riqueza de detalhes, mas requer um algorítimo especializado para cada tipo de media, e fazer estes algorítimos pode levar uma grande quantidade de tempo e esforço. As medias criadas proceduralmente normalmente tem uma qualidade inferior a uma media criada por um humano, portanto, usualmente quando se trabalha com medias procedurais, ou estas são muito simples, ou existe um trabalho sobre estas feito por humanos para assegurar qualidade, ou ainda, não há requisitos de alta qualidade.

Mais especificamente, as funções de ruido tem uma grande participação na geração procedural, especialmente o Perlin Noise feito por Ken Perlin. Desde a primeira imagem do vaso de mármore apresentada por K. Perlin, o Perlin Noise tem se popularizado vastamente na indústria (A. Lagae et al, 2010, pg 1). Algum tempo depois Ken Perlin criou o Simplex Noise, que é uma versão aprimorada do antigo Perlin Noise, para resolver alguns dos problemas que o seu algoritmo original tinha.

1.1 Pergunta da Pesquisa

A motivação deste trabalho é explorar o porquê as pessoas ainda preferem utilizar Perlin Noise em vez do Simplex Noise, que deveria ser melhor. Por exemplo, Minecraft e Minetest usam Perlin Noise (M Persson, 2011 e P Ahola, 2011) e não foi encontrado exemplo do uso de simplex noise na prática.

1.2 Objetivos Gerais

Descobrir o porquê Perlin Noise é tão vastamente usado quando o Simplex Noise, que é supostamente melhor em tudo, é praticamente ignorado.

1.3 Objetivos Específicos

Realizar experimentos de performance sobre ambos algorítimos para geração de terreno procedural e comparar os resultados junto a métricas de complexidade de algoritmo.

1.4 Metodologia e Resultados esperados

A execução do trabalho vai seguir a seguinte ordem:

1. Implementação da função Perlin Noise e Simplex Noise.
2. Testes de desempenho sobre ambos os algorítimos em duas para geração de terreno procedural, espera-se que o Simplex Noise seja mais rápido em todos os casos. A geração e visualização dos resultados será feita usando Unity. Todo o código referente aos algorítimos Simplex Noise e Perlin Noise será escrito em Java, o resto será escrito em C# ou Java Script.
3. Com os dados obtidos será feita uma análise comparativa de desempenho, medindo o tempo levado para criar um terreno de mesmo tamanho em milissegundos e também baseado no código será comparado o esforço, tempo estimado para programar e numero estimado de bugs.

2 Revisão Bibliográfica

2.1 Geração Procedural

Geração Procedural é o processo de criar conteúdo seguindo um procedimento ou função. Fractais são um exemplo de Geração Procedural visto que estes são produzidos a partir de formulas matemáticas.

Geração Procedural pode ser usada para criar Texturas, Malhas Poligonais, Musica, Síntese de Voz entre outros.

2.1.1 Geração Procedural em Filmes

Geração Procedural é usado em filmes para criação rápida de ambientes completos, exemplos disso são AVATAR (James Cameron, 2009) usando SpeedTree e TRON (Steven Lisbergerpted, 1982) usando Perlin Noise, em ambos os casos a Geração Procedural foi usada para criar ambientes extensos e completos rapidamente.

2.1.2 Geração Procedural em Jogos

Esta seção foi feita levando em consideração as afirmações feitas por M. Hendrikx. et al (2011).

Jogos eletrônicos se passam usualmente em mundos imensos e diversos, e a medida que os computadores ficam mais potentes, os jogadores começam a exigir um nível maior de detalhes nesses mundos, o que leva ao aumento do custo para produzir mundos grandes e detalhados.

Conteúdo é uma parte importante do jogo para manter o jogador entretido, sejam leveis, itens, moveis, armas, ambientes, monstros, NPCs. Tudo isso consome tempo e recurso para ser produzido e, a medida que o tempo passa, os consumidores exigem mais e mais conteúdo, e até conteúdo personalizado para cada jogador, sendo que o processo de criação de tal conteúdo já é árduo e caro e não escalável.

Ao contrario do processo manual, a geração procedural de conteúdo usa o poder de processamento do computador para geral vários tipos diferentes de conteúdo usando formulas e procedimentos específicos para cada tipo de conteúdo.

Mas Geração Procedural de Conteúdo para Jogos não requer somente poder de processamento, também é necessário a capacidade de julgar se o produto gerado é próprio

para a cultura/idade alvo ou não. Não é surpreendente, que mesmo após anos de pesquisa e desenvolvimento na área de Geração Procedural de Conteúdo para Jogos que ainda não exista um gerador de proposito universal.

Os jogos mais populares tem ficado maiores, mais bonitos, mais imersivos e mais detalhados a cada geração. Por exemplo, o MMORPG World of Warcraft apresenta aos seus jogadores um mundo de fantasia de grande complexidade. Cada um dos dois continentes (Atualmente três continentes, um pedaço de planeta e uma tartaruga gigante) tem aproximadamente 1/4 da área de Florianópolis e inclui uma grande diversificação de ambientes, ecossistemas, cidades, rede de estradas. O jogo possui vários tipos de conteúdo, como Sons, texturas, terrenos, construções, cidades, comportamento de objetos e cenários. No total, em 2008 o jogo era composto de mais de 1400 localizações geográficas, 30.000 itens, 5.300 criaturas, 7.600 missões, e 2.000.000 palavras de texto. Isso tudo foi criado em aproximadamente 5 anos.

Nos jogos atuais, a produção de conteúdo de alta qualidade requer algumas centenas de pessoas, entre artistas, designers, programadores, engenheiros de áudio. Como consequência, o preço de produção e tempo necessários de desenvolvimento cresceu a ponto de virar um gargalo.

Técnicas procedurais são uma alternativa para criar mundos virtuais complexos em um tempo curto sem sobrecarregar os designers. A ideia principal da geração procedural, é o conteúdo ser gerado não por humanos, mas por computadores executando procedimentos bem definidos. Idealmente, tais procedimentos possuem vários parâmetros de configuração, para que os designers não percam o controle do resultado final.

Em 3 décadas de pesquisas foram criados vários procedimentos para gerar diversos tipos de conteúdo. Um dos mais simples foi o uso de números pseudoaleatórios (PRNG). O jogo espacial Elite de 1980 usou esta estratégia para gerar universos imensos. Ou gerar texturas para objetos como feito por Perlin em 1985.

Apesar da abundância de procedimentos para gerar conteúdo, sua aplicação comercial não é mainstream. Métodos procedurais de geração de conteúdo foram aplicados para geração de diversos tipos diferentes de conteúdo com sucesso, mas ainda não existe uma solução de propósito genérico. A literatura de geração procedural para jogos está espalhada em diversas áreas do conhecimento, como computação gráfica, processamento de imagem, inteligência artificial, psicologia, linguística, ciências sociais, etc..)

2.1.2.1 Terreno Procedural

Os mundos virtuais tem, nas últimas duas décadas, se distanciando de simples retas e plataformas para ambientes de alta complexidade, entretanto as técnicas de modelagem de terreno não tem se aperfeiçoado no mesmo ritmo.

Terreno em computação (especialmente em jogos) é usualmente representado em *heightmaps*, que são grades regulares que representam a altura do terreno. Pode-se encontrar terrenos representados por *voxels*, ou até mesmo *meshes* conectados, mas estas duas representações são menos comuns, sendo a segunda normalmente usada somente para ambientes internos como cavernas, naves ou calabouços e são usualmente chamados de *level* ou *stage* em vez de terreno.

2.1.2.1.1 Mapa de Altura

Na área da computação gráfica um mapa de altura (em inglês: *heightmap* ou *heightfield*) é uma grade bidimensional utilizada para armazenar valores relativos à elevação de uma superfície, para a exibição como gráficos tridimensionais.

Como explicado por L. Szirmay-Kalos e T. Umenhoffer (2008), um *Heightmap* é usualmente visualizado como uma imagem que possui apenas um canal o qual é interpretado como a distância de um determinado ponto até o chão. Geralmente apresentam-se como imagens em escala de cinza, onde o preto representa a menor e o branco a maior altitude possível.

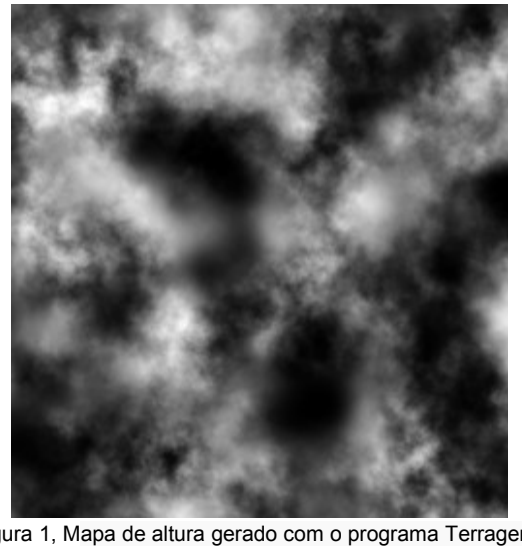

Figura 1, Mapa de altura gerado com o programa Terragen, Fonte: http://upload.wikimedia.org/wikipedia/commons/5/57/Heightmap.png

Mapas de altura são amplamente utilizados em sistemas de informação geográfica, na renderização de terrenos em geral e nos vídeo games modernos. Eles são a maneira ideal de armazenar um terreno de acordo com a sua elevação. E se comparado às comuns malhas (em

inglês: *Polygon Mesh* ou *Mesh*) poligonais o espaço requerido para armazenar tal informação geralmente é inferior.

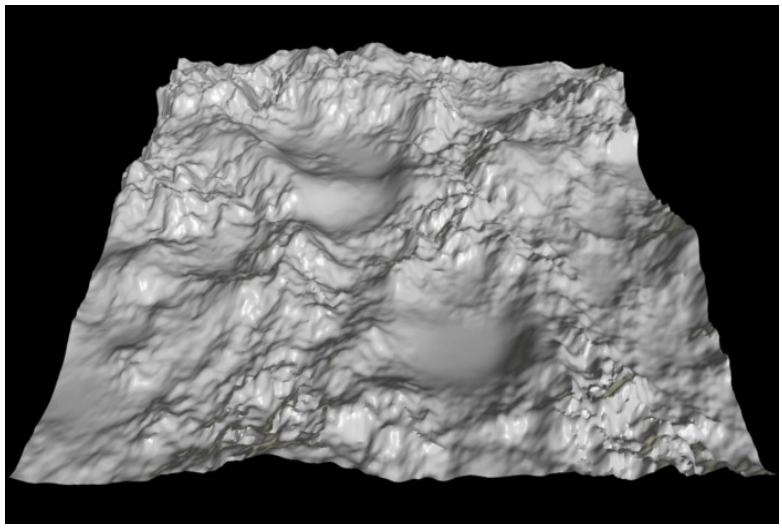

Figura 2, O mesmo mapa de altura convertido em uma malha poligonal e renderizada com o programa Anim8or, Fonte: http://upload.wikimedia.org/wikipedia/commons/2/2f/Heightmap_rendered.png

Os algoritmos mais antigos para geração de Mapas de Altura são os baseados em subdivisão. Um mapa de altura plano é subdividido diversas vezes, e a cada passo, é adicionado um pouco de aleatoriedade controlada com o objetivo de acrescer detalhe. Atualmente, é mais comum usar algoritmos baseados em ruído fractal como Perlin Noise.

Como dito por R. M. Smelik et al (2009) Mapas de Altura podem ser trabalhados adicionando efeitos de tratamento de imagem como suavizar ou simular erosão. Erosão Térmica, pode ser aplicada para simular o efeito da erosão, diminuindo as pontas afiadas, que iterativamente remove ‘material’ das partes altas e ‘deposita’ nas partes baixas até o angulo de estabilidade (varia de material para material). Embora estes algoritmos de erosão criam estruturas mais realistas, eles são custosos e lentos.

Geradores de Mapa de Altura baseados em Ruido tem a capacidade de criar mapas de altura com um bom grau de aleatoriedade, entretanto só dando ao usuário um controle global via parâmetros pouco instrutivos. Varias soluções foram desenvolvidas para contornar este problema, como utilizar mapas de altura base.

Uma das desvantagens de se usar mapas de altura é que eles nativamente não conseguem representar cavernas e outras estruturas de ‘mais de um andar’, ou até mesmo ângulos de 90º.

2.1.2.1.2 Voxel

Voxel (*Pixel* Volumétrico ou Elemento de Figura Volumétrica) é um elemento de volume em uma grade regular 3D, sendo análogo a um pixel em uma imagem Bitmap 2D. Assim como *pixels*, *voxels* não armazenam sua posição explicitamente, mas esta pode ser obtida implicitamente pela posição do mesmo na grade 3D.

Voxels são usados na maior parte das vezes para visualizações de radiografias, ultrassom, etc... Mas vários jogos usaram *Voxels* para renderização de naves/itens/terreno.

Voxels costumam também conter informações como cor, densidade, fluxo, etc, da mesma maneira que *pixels* contem cor/transparência, especialmente para aplicações voltadas a medicina.

2.1.2.1.3 Arvores/Distribuição da Vegetação

Quanto a vegetação, autores desenvolveram diversos procedimentos para geração de arvores e plantas procedurais e métodos para automaticamente adicionar tais modelos no terreno.

Plantas procedurais crescem, iniciando da raiz, adicionando galhos cada vez menores e terminando em folhas.

2.1.2.1.4 Rios/Oceanos/Lagos

Vários autores propuseram algoritmos para tanto durante quanto depois da geração do mapa de altura. R. M. Smelik apud Kelley et al (1988) propôs usar uma rede de rios como base do mapa de altura, usando um único rio reto que é subdividido recursivamente, resultado em uma rede de rios, e então usado como base para fazer o mapa de altura. Sendo que o tipo de solo pode influenciar no formato do da rede de rios.

R. M. Smelik apud Prusinkiewicz e Hammel (1993) combinam a geração de um rio curvo com o esquema de subdivisão do mapa de altura. Do triangulo inicial, um lado é marcado como inicio e outro como fim. A cada passo de subdivisão, cada triangulo é subdividido em 4 triângulos, agora existem diversos caminhos por onde o rio pode passar e ainda entrar e sair pelos mesmos lados. Uma das desvantagens do método é que o rio tem que ser plano e acaba cavando buracos estranhos em montanhas.

Com exceção dos rios, corpos d'água procedurais, como oceanos, lagos e suas conexões, deltas e cachoeiras, tem recebido pouquíssima atenção. Tipicamente os lagos são completamente desconsiderados. Oceanos são usualmente gerados inundando tudo abaixo da altitude 0 ou com algoritmos de inundação iniciando das regiões mais baixas.

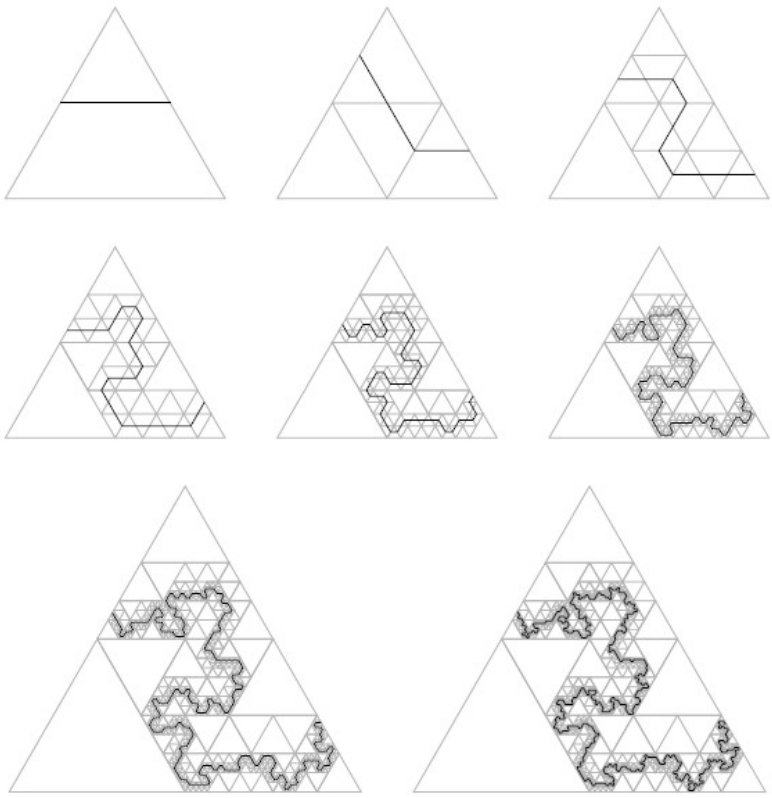

Imagem 3, Exemplo de execução do algorítimo de Prusinkiewicz e Hammel (1993), Fonte: Prusinkiewicz, P. and Hammel, M. (1993). A Fractal Model of Mountains with Rivers. Em Pro-ceeding of Graphics Interface '93, paginas 174-180.

2.1.2.3 Texturas/Imagens Procedurais

De acordo com D. S. Ebert et al (2003) texturas procedurais em geral tem certas vantagens sobre os métodos tradicionais de texturas:

- São mais leves, usualmente na casa de *kbytes*.
- Não tem resolução pré definida, podendo teoricamente ser ampliada infinitamente.
- Não tem largura ou altura pré definida, podendo cobrir áreas teoricamente infinitas sem repetir o padrão.
- Podem ser parametrizada, assim permitindo um numero grande de texturas parecidas, mas diferentes.

É importante ressaltar que estas vantagens não são garantidas, a geração procedural da as ferramentas para se assegurar tais vantagens. Um algorítimo mal escrito ou mal planejado poderia perder todas essas vantagens.

Mas texturas procedurais também tem suas desvantagens, tais como:

- É difícil de programar e debugar, a construção de padrões fica extremamente complicada em situações não triviais.
- Calcular uma textura procedural pode demorar bem mais que simplesmente usar uma do disco.
- Antialias com texturas procedurais é extremamente complicado e as vezes simplesmente não funciona

2.1.2.3.1 Imagem Vetorial

Talvez a forma mais comum de imagem procedural. A imagem vetorial (*Vector Graphics*) é uma imagem composta de diversas definições de formas geométricas como linhas, curvas, triângulos, gradientes, elipses, etc.

Por ser basicamente uma coleção de funções, as imagens vetoriais ocupam pouca memoria, entretanto são mais lentas que as imagens *raster* tradicionais.

2.1.2.3.2 Texturas Procedurais

Usualmente texturas procedurais são usadas para criar representações realistas de elementos da natureza que tem uma aparência fractal, como a textura de pedras de mármore, granito, madeira, metais, etc..

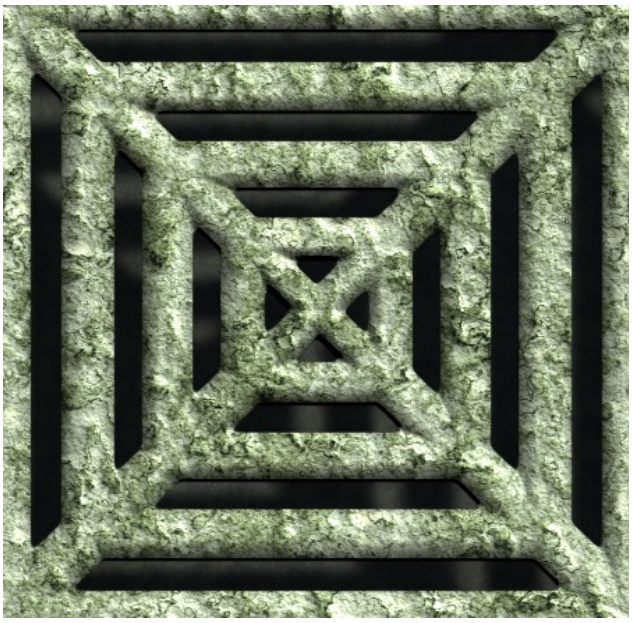

Imagem 4, Uma textura de grade de chão gerada proceduralmente usando o editor Genetica, Fonte: http://en.wikipedia.org/wiki/File:Procedural_Texture.jpg

2.1.2.3.3 Skybox/Skydome

Uma skybox é um cubo, onde as faces internas do cubo são preenchidas com uma imagem de fundo que representam o ambiente do level. Seja o espaço, um céu estrelado, um céu de dia ou interior de uma caverna, o objetivo da skybox é poder fazer ambientes 'abertos'.

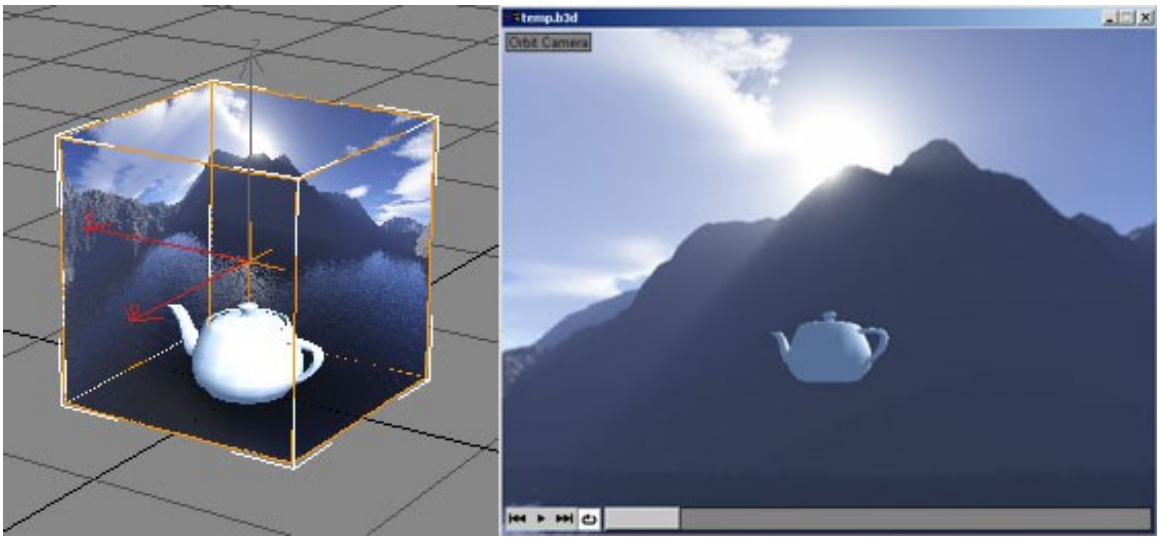

Figura 5, Exemplo de Skybox, Fonte: http://www.onigirl.com/pipeline/Images/Sample_SkyBox.jpg

Um Skydome é a mesma coisa que um skybox, mas usa uma esfera ou semisfera em vez de um cubo.

Uma imagem de skybox pode ser gerada proceduralmente[9].

2.2 Funções de Ruído

Ruido é usualmente criado somando faixas de valores. É como tocar um instrumento musical, se você tocar varias notas ao mesmo tempo, vai resultar em um ruido, ao mudar as notas, muda o ruido, mas as notas não tem necessariamente o mesmo peso sobre o ruido que é ouvido. O mesmo ocorre em varias Funções de Ruido, cria-se faixas e então as soma para obter o resultado final. Outro detalhe interessante é que usualmente as Funções de Ruido tendem a gaussianidade, seja por somar vários números independentes ou não.

2.2.1 Como funciona o Perlin Noise

Esta seção foi feita levando em consideração as afirmações feitas por S. Gustavson (2005).

Perlin Noise funciona sobre uma grade N dimensional de reais, tendo os números inteiros como pontos da grade.

Para produzir um valor com perlin noise para um ponto P(x, y, n... | x, y, n... E R) deve ser possível produzir/resgatar um vetor N dimensional para cada posição P'(X, Y, N....| X, Y, N... E I) Usualmente usa-se duas tabelas, uma tabela de permutação e uma tabela de vetores.

O primeiro passo é gerar um numero para cada posição inteira, desconsiderando a parte fracionaria do numero, o qual sempre deve ser o mesmo para a mesma posição. A maneira mais comum é:

1. Usa-se a tabela de permutação que contem de 0 a 255 em uma ordem aleatória (perm[0] = 2, perm[1] = 15, perm[2] = 235, perm[4] = 0, …)
2. Pegue um valor da tabela usando X (X%256 por exemplo)
3. Some Y ao valor adquirido e usa-o para pegar um valor da tabela novamente ((Y+perm[X%256])%256)
4. Repete 3 (trocando Y por n...) até que não sobre mais dimensões.

O objetivo é poder resgatar um vetor n dimensional usando o valor de 0 a 255.

Sendo o valor de 0 a 255 sempre o mesmo para o mesmo ponto, o vetor é sempre o mesmo para o mesmo ponto.

Deve-se então descobrir pontos(P1, Pn...) que formam um n-cubo[1] unidade que contenha o ponto P. A maneira convencional é arredondar x, y, n... para baixo, assim obtendo P1(X, Y2), P2(X+1,Y), P3(X,Y+1), P4(X+1,Y+1) (para duas dimensões).

Para cada ponto Pn deve-se resgatar seu vetor referente Vn, e criar um vetor VPn do ponto Pn até o ponto P. Deve-se então calcular o produto escalar En entre Vn e VPn.

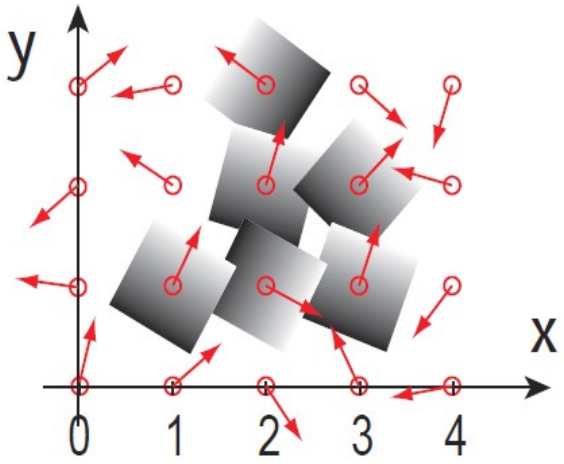

Figura 6, Vetores referentes aos pontos na grade, Fonte: Stefan Gustavson(2005) Simplex noise demystified, pagina 2.

Para cada En deve-se finalmente calcular a influencia In. Se o ponto P está próximo ao P1, e longe de P2. A influencia de E1 é maior que de E2. Idealmente, se a distancia entre Pn e P(x,y) for 1 ou mais, a influencia deve ser 0, se for 0,5 a influencia deve ser 0,5 e se a

1 Quadrado(2D), Cubo(3D), 4D-Hipercubo(4D), etc

distancia for 0 a influencia deve ser 1. Então faz-se uma media ponderada entre os produtos escalares En usando a influencia como peso. Entretanto, somente isso cria um resultado não natural, é necessário também aplicar uma formula que suaviza a transição. As duas formuas mais comuns são: 3In^2-2In^3 e 6In^5-15In^4+10In^3, suas respectivas curvas estão representadas no gráfico abaixo

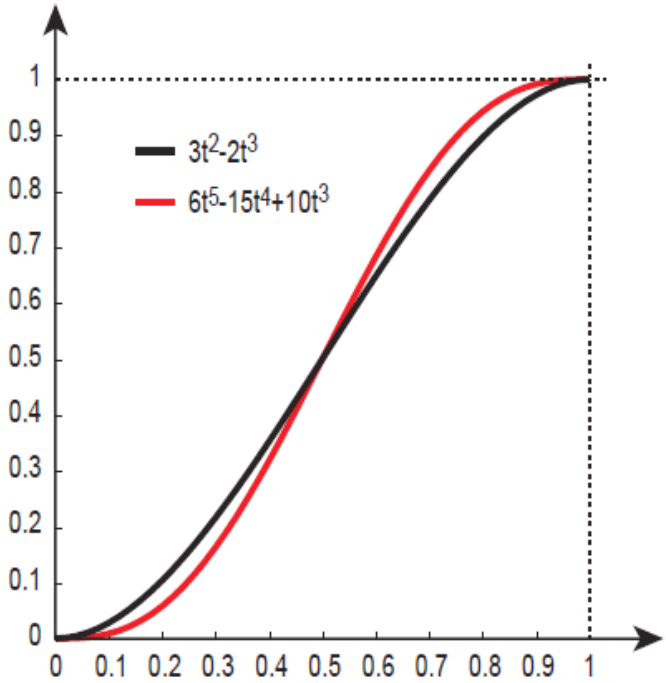

Figura 7, Curva de suavidade, Fonte: Stefan Gustavson(2005) Simplex noise demystified, pagina 2.

Observações:

- O valor de pontos inteiros (1,1;1,2;0,1;0,0) é sempre zero, pois a influencia dos demais pontos é zero e o produto escalar de um vetor V e o vetor (0,0) é 0.
- Trocando a tabela de permutação você pode trocar o resultado final do algorítimo. Ou seja, a tabela de permutação é a 'seed' do algorítimo.
- Os vetores Vn determinam a aparência do produto final.
- O método usado para calcular a influencia In muda o 'formato' do resultado de uma maneira semelhante a trocar os vetores Vn.

2.2.2 Como o Simplex Noise funciona

Esta seção foi feita levando em consideração as afirmações feitas por S. Gustavson (2005).

Simplex noise funciona sobre uma grade de simplex[1], pode ser visualizado como uma tesselação do espaço usando o polígono mais simples. Para duas dimensões o triangulo equilátero é o simplex usado, para 3D é usado um tetraedro.

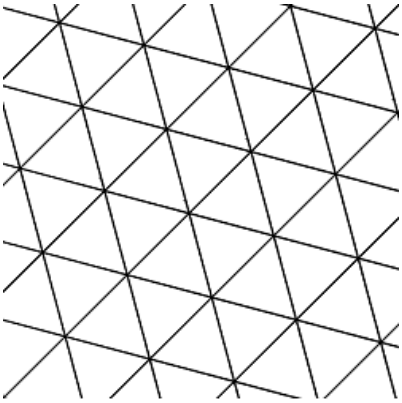

Figura 8, Grade simplex 2D

O primeiro passo é descobrir em qual simplex(P1,P2,Pn..) o ponto P se encontra. Para isso se distorce a coordenada do ponto para que esta seja equivalente a mesma coordenada em uma grade simples[2].

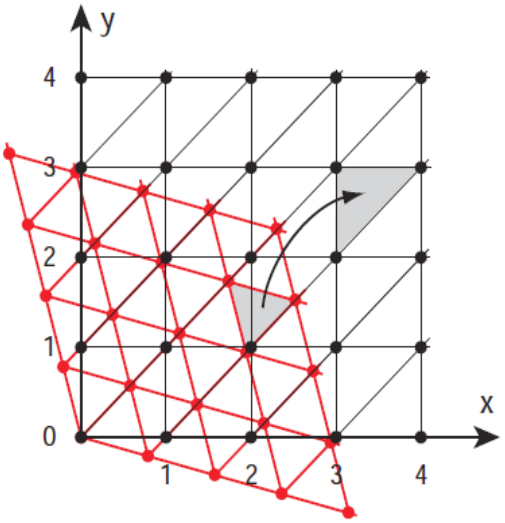

Figura 9, Dobramento da grade simplex, Fonte: Stefan Gustavson(2005) Simplex noise demystified, pagina 6.

Para tal, soma-se a coordenada x, y, n... e multiplica os valores por um coeficiente de dobramento. A formula para o coeficiente é: $(\sqrt{(n+1)} - 1) / n$ sendo n o numero de dimensões. Então após descobrir em qual cubo o ponto está, confere se a parte decimal da coordenada X é maior que a parte decimal da coordenada Y para saber se o ponto está no simplex de cima ou no de baixo. O fator de dobramento é a soma das coordenadas multiplicado pelo coeficiente de dobramento. Para descobrir as coordenadas do ponto equivalentes caso este estivesse em

1 Simplex é o polígono que tem menos vértices/lados de sua dimensão (triangulo, tetraedro, pentachoron)

2 Grade de quadrados(2D), de cubos(3D), etc

uma grade simples Xs, Ys, Ns... basta adicionar o fator de dobramento as coordenadas x, y, n....

Depois deve-se retirar a parte inteira de Xs, Ys, Ns... para determinar em qual simplex que forma o n-cubo o ponto P está. E para fazer o fator de desdobramento. Assim como o fator de dobramento, o fator de desdobramento é composto pela soma de Xs, Ys, Ns... multiplicado pelo coeficiente de desdobramento, cuja formula é (n+1- √(n+1)) / n * (n+1). A distância entre o ponto P e o ponto mais próximo da origem do n-cubo que contem P pode ser calculada por Dn = n – fator de desdobramento para toda coordenada n que forma o ponto P.

A distancia Dn é usada para determinar em qual simplex que forma o n-cubo que contem P como mostra nas figuras abaixo e qual a distancia entre o ponto P e os pontos P1, P2, Pn que formam o simplex que contem P.

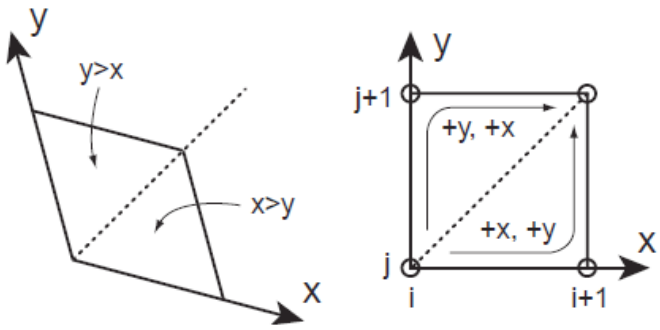

Figura 10, Calculando o simplex em um quadrado, Fonte: Stefan Gustavson(2005) Simplex noise demystified, pagina 6.

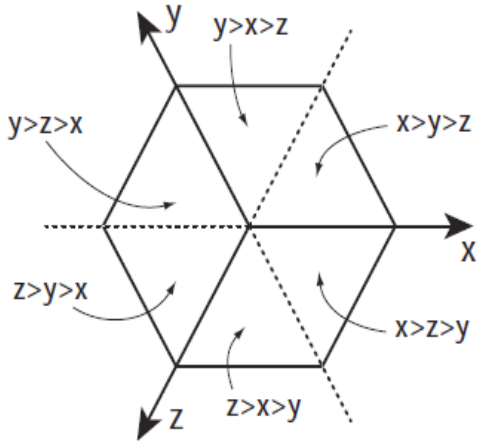

Figura 11, Calculando o simplex em um cubo, Fonte: Stefan Gustavson(2005) Simplex noise demystified, pagina 6.

Tendo os pontos P1, P2, Pn... e suas distancias Dpnx, Dpny, Dpnn... com P, calcula-se qual é o vetor Vn para cada ponto que forma o Simplex da mesma maneira que é feito no Perlin Noise para calcular os produto escalar entre P e Vx, Vy, Vn...

Os produtos escalares PS1, PS2, Psn... são então somados levando em consideração sua distancia com P usando a seguinte formula:

contribuição(n) = (0.5 – Dpnx^2 – Dpny^2 – Dpnn...^2)^4 * PSn.

2.3 Métricas de Halstead

Para Halstead, um programa de computador pode ser visto como uma sequencia de operandos e operadores. Todas as métricas de Halstead são funções das soma de operandos e operadores como parâmetros. As quatro principais medidas são:

- n_1 = numero de operadores únicos.
- n_2 = numero de operandos únicos.
- N_1 = numero total de operadores.
- N_2 = numero total de operandos.

Usando essas medidas é possível calcular as métricas:

- Comprimento do Programa $N = N_1 + N_2$ que representa o numero de palavras do programa.
- Vocabulario do Programa $n = n_1 + n_2$ que representa o numero de palavras diferentes do programa.
- Volume $V = N \times \log_2 n$ que representa quanta informação o leitor deve absorver para compreender o codigo.
- Dificuldade $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$. O numero de operadores únicos tem um grande impacto na dificuldade do programa. Comparado a outras formas de texto, como um livro, quanto mais um numero pequeno de palavras é repetido, mais facil é a leitura, mas em um programa, quanto mais um operando é usado, mais dificil é a compreenção do mesmo pois este é modificado ou lido em mais lugares.

Halstead também criou uma serie de métricas derivadas:

- Esforço $E = D \times V$ representa o quanto esforço é necessario para escrever o codigo, vale resaltar que isso não inclui o esforço de interpretar as especificações do programa nem o esforço para realizar os testes.
- Tempo estimado para programar $T = \frac{E}{18}$ segundos. O numero 18 foi escolhido baseado nos estudos do psicologista John Stroud concluio que um humano pode detectar entre 5 e 20 eventos ou momentos por segundo. De acordo com halstead, considerando que um programador costuma programar constantemente, este deve

portanto, estar habituado a este tipo de informação e treinado a um numero bem alto na escala.

- Numero de bugs estimado $B = \frac{V}{3000}$

2.4 Motores de Jogo

Um Motores de Jogo é um conjunto de ferramentas integradas que visam facilitar a criação de jogos eletrônicos. As ferramentas disponíveis variam de motor para motor, mas algumas estão quase sempre presentes, tais como:

- Simulação de Física: Detecção de Colisão entre dois objetos, aplicar força em determinada direção, etc.
- Edição e Criação de Cenas: Cenas são pedaços do jogo, sendo um level, sala, ou até mesmo um menu.
- Programação e Scripting: Scripts são pedaços de código, usualmente monolíticos, que realizam uma ação simples baseado em um gatilho. Por exemplo uma armadilha ativando quando um personagem pisa em um painel de pressão.
- Depuração: Ser capaz de rodar um código com erro sem travar, e ao invés avisar ao desenvolvedor onde ocorreu o erro e manter o maior numero de variáveis intactas para facilitar a identificação do erro. Ou até mesmo permitir editar o código durante a execução.

2.4.1 Unity 3D

Unity3D é um motor de jogos comercial que possui uma licença gratuita. A licença gratuita possui diversas limitações como não poder gerar informação de navegação automaticamente, sombras em tempo real ou trocar a imagem de abertura do jogo. Unity exporta o jogo para Windows, Mac, Android, IOS, PS3, Xbox, WiiU, Navegadores de Internet e adicionado recentemente Linux e Adobe Flash. Unity também conta com a *Asset Store*, uma loja virtual onde se pode vender e comprar pedaços de software, como um sistema de *pathfind* ou um sistema de interface gráfica.

A estrutura do Unity é similar a outros ambientes de desenvolvimento, sendo baseada em projetos. Cada projeto corresponde a uma aplicação ou jogo. Uma aplicação é organizada em cenas, cada cena contendo um conjunto de objetos, scripts, sons, etc.. responsáveis por fazer a aplicação funcionar.

Unity vem com o MonoDevelop, um IDE para programação de scripts. Os scripts podem ser escritos em C#, Javascript e Boo. Scripts implementam a interface MonoBehaviour e são atribuídos a objetos de cena. Quando a cena é iniciada, todos os scripts associados a algum objeto da cena são ativados via o método start(). Outros métodos como awake() ou Update() podem ser usados para implementar as funcionalidades do script.(Unity 3D – Game Engine)

2.5 Maquina Virtual Java

Maquinas virtuais são uma implementação em software de uma maquina que executa programas como uma maquina física. A Maquina Virtual Java é uma maquina virtual cujo proposito é executar Java bytecode, existem diversas implementações para diferentes sistemas

2.5.1 IKVM

IKVM é uma implementação Java para Mono e Microsoft .NET Framework. IKVM contem os seguintes componentes:

- Maquina Virtual Java implementada em .NET
- Classes das bibliotecas Java implementadas em .NET
- Conjunto de ferramentas que viabilizam a interoperabilidade entre Java e .NET (IKVM.NET Home Page)

2.6 Microsoft .NET Framework

O Microsoft .NET Framework tem o objetivo de prover as seguintes funcionalidades:

- Programação orientada a objeto consistente independente do código estar sendo executado localmente, localmente mas com conteúdo distribuído pela internet ou remotamente.
- Um ambiente de execução de código que minimiza os conflitos de versionamento, permite criação de código seguro, independentemente de ter sido criado por um desenvolvedor desconhecido ou não e elimina os problemas de performance de ambientes interpretados ou baseados em script.
- Permitir uma experiencia de desenvolvimento consistente para tanto desenvolvimento de aplicações Windows ou baseadas na web.
- Constrói toda a comunicação baseada nos padrões da industria para permitir que .NET se integre com qualquer tipo de código.

O .NET consiste de uma linguagem comum de runtime e a biblioteca .NET. (Overview of the .NET Framework)

3 Resultados obtidos

Foram criadas 3 classes java durante o desenvolvimento do trabalho, uma responsável por gerar o ruido de Perlin (PerlinNoise.java) outra responsável por gerar o ruido simplex(SimplexNoise.java) e uma terceira responsável por gerenciar e criar os vetores e o cálculo de produto vetorial usado igualmente em ambas as classes que geram ruído(VectorFactory.java).

Para visualizar os resultados em duas dimensões, foi criado uma classe java que tanto realiza um teste de eficiência quanto mostra os resultados de ou Perlin Noise ou Simplex

Para visualizar os resultados em três dimensões foi utilizado o Unity, foi criado uma cena, um script, um terreno, plano e três texturas. Foi utilizado IKVM para exportar os códigos java para .NET dll para que o script em C# no Unity pudesse utilizá-los. O terreno foi usado para visualizar os resultados do noise, o plano representa a altura zero, a textura do plano é um ponto preto com transparência e as outras duas foram aplicadas ao terreno para representar a grade referente ao algorítimo sendo usado.

Um teste simples de desempenho foi realizado para comparar ambos algoritmos. Cada um dos algoritmos é usado para gerar 1000000 resultados em uma, duas, três e quatro dimensões.

As Métricas de Halstead foram calculadas usando JHawk 5.

3.1 Perlin Noise

Teste de desempenho:

Dimensões	Tempo	Intervalo
1	0.49 segundos	[-0.5, 0.5]
2	1.14 segundos	[-0.49, 0.49]
3	2.68 segundos	[-0.42, 0.42]
4	6.65 segundos	[-0.37, 0.37]

Metricas de Halstead:

Metrica	Valor
Esforço	147727
Tempo estimado	2h 16m 47s
Numero esperado de bugs	1.26

Representação do resultado do Perlin Noise em 3D

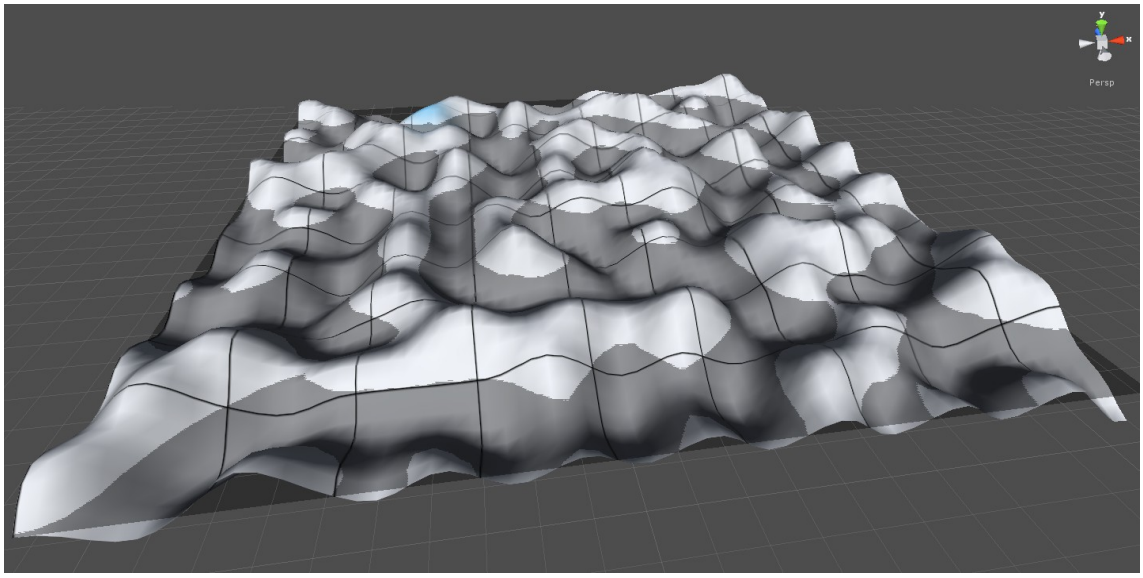

Figura 12, Perlin Noise de duas dimensões visualizado em 3D

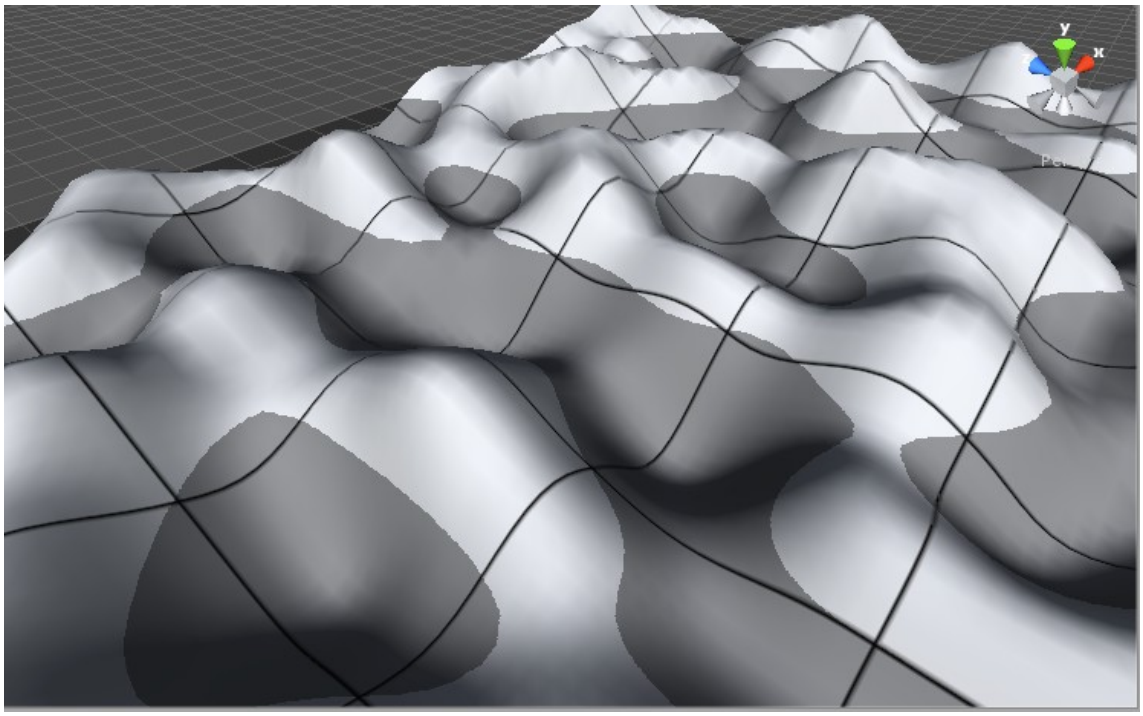

Figura 13, Perlin Noise de duas dimensões visualizado em 3D com enfase nos pontos zero

3.2 Simplex Noise

Teste de desempenho:

Dimensões	Tempo	Intervalo
1	0.46 segundos	[-1.4, 1.4]
2	0.82 segundos	[-1, 1]
3	0.541 segundos	[-1.17, 1.17]

4	0.746 segundos	[-1.5, 1.5]

Métricas de Halstead:

Métrica	Valor
Esforço	311147
Tempo estimado	4h 48m 6s
Numero esperado de bugs	1.60

Representação do resultado do Perlin Noise em 3D

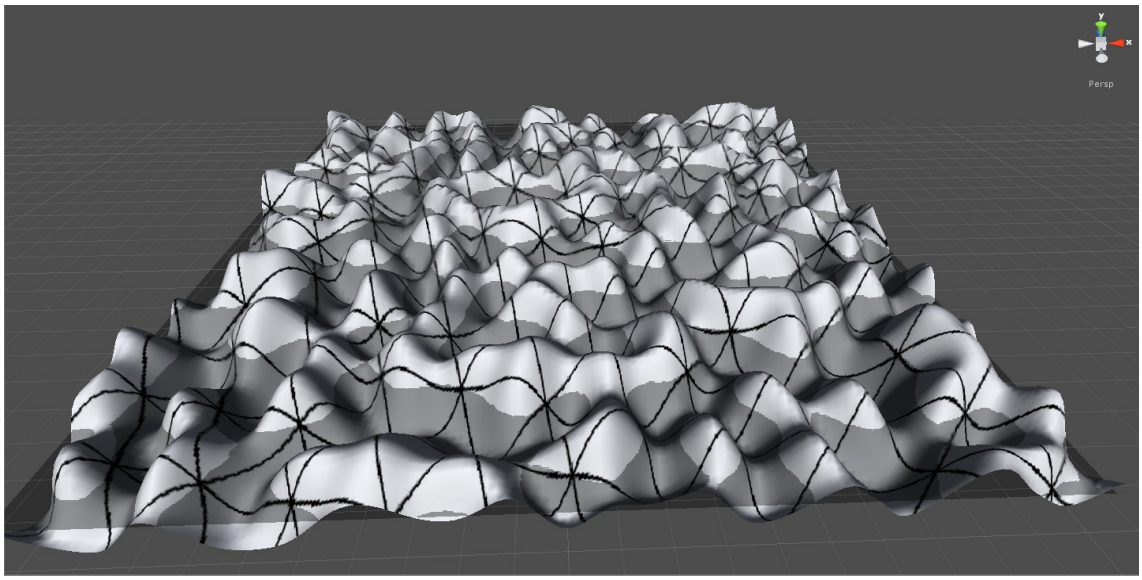

Figura 14, Simplex Noise de duas dimensões visualizado em 3D

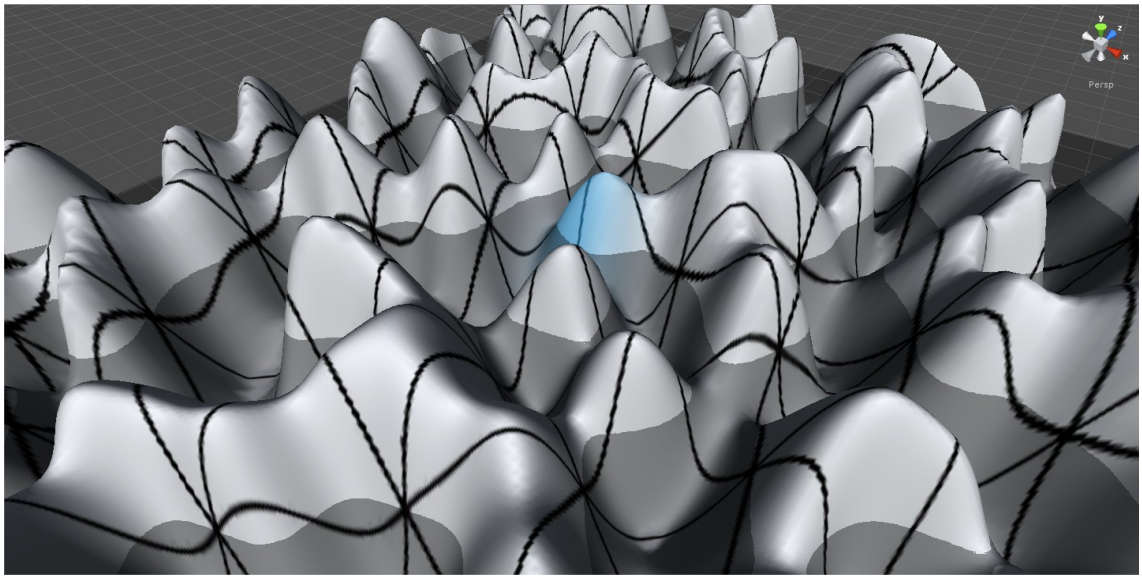

Figura 15, Simplex Noise de duas dimensões visualizado em 3D com enfase nos pontos zero

3.3 Geração de Terreno Procedural

Geração de terreno procedural é um problema mais complexo que geração de ruído procedural. O requisitos de um terreno procedural podem mudar drasticamente dependendo do objetivo, mas em geral estes tem que ter um certo nível de realismo.

O terreno gerado para este experimento será uma aproximação simples do relevo da terra usando quatro valores diferentes criados a partir das funções de ruido Simplex Noise ou Perlin Noise. O código usado para gerar o terreno estará disponível como anexo.

3.3.1 Perlin Noise

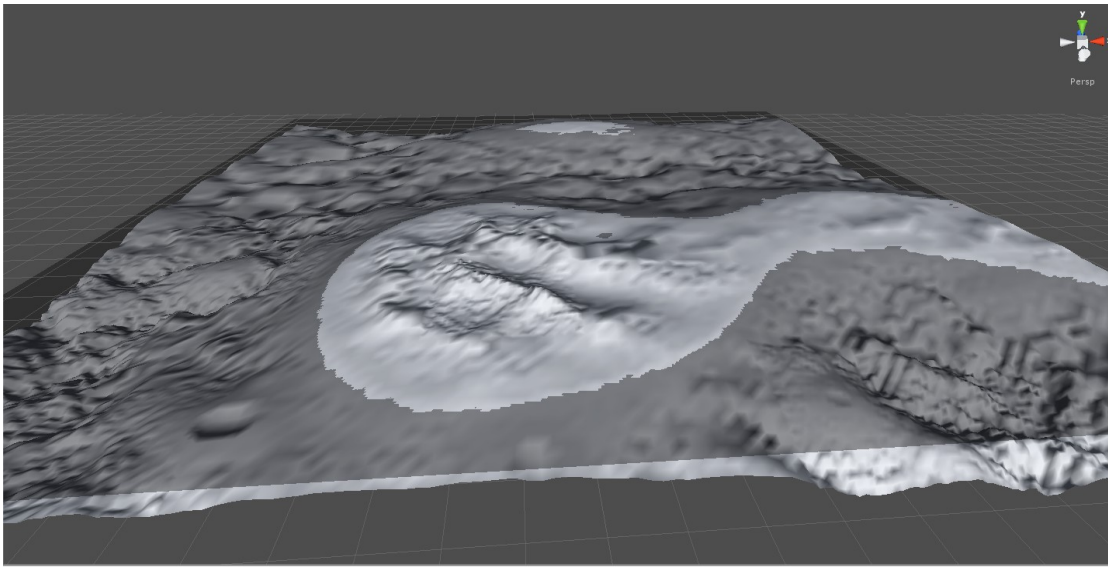

Figura 16, Geração de terreno procedural com Perlin Noise

3.3.1 Simplex Noise

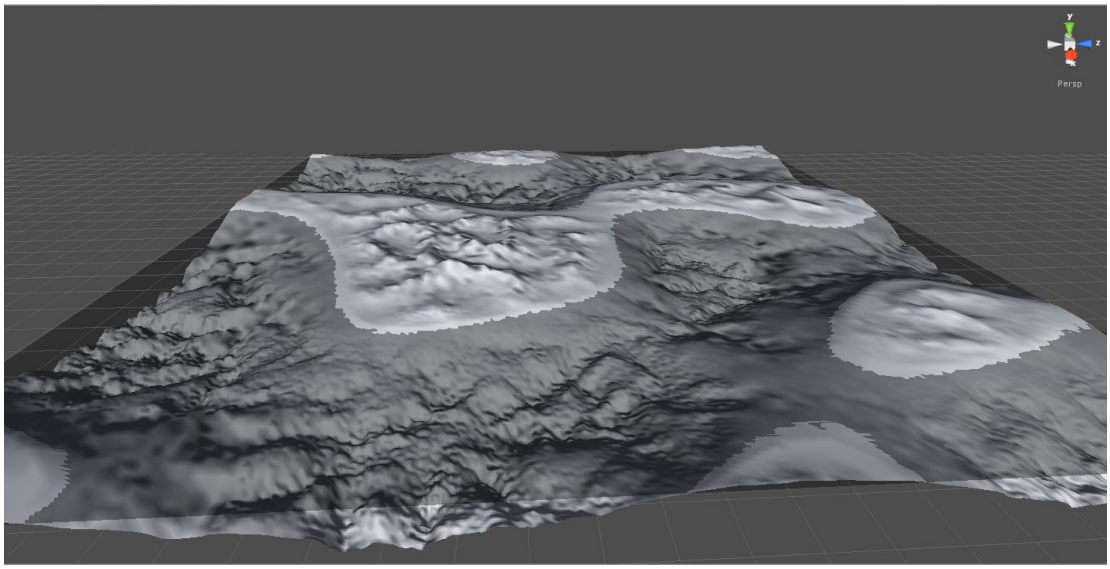

Figura 17, Geração de terreno procedural com Simplex Noise

4 Conclusões

O Terreno gerado proceduralmente usando ambas as soluções tem praticamente a mesma qualidade, por mais que o Simplex Noise tenha demorado menos para gerar a mesma quantidade de informação, o tempo necessário para gerar os 1052676 valores de ruido 2D foi aproximadamente 1 segundo em ambos os casos.

O Simplex Noise é exponencialmente mais eficiente para gerar o ruido, seus pontos zero seguem uma grade de triângulos regulares alinhada a diagonal dos eixos X e Y e possui um nível de gauseanidade exponencialmente inferior ao do Perlin Noise em dimensões maiores que 3.

O Perlin Noise é extremamente mais fácil e rápido de implementar, não requer conhecimento de simplex e seus pontos zero são bem mais previsíveis pois seguem uma grade simples alinhada ao eixo X e Y.

Qual algorítimo é o melhor vai depender muito da aplicação onde este vai ser usado. Em geração de terreno procedural a gauseanidade e previsibilidade dos pontos zero são irrelevantes na maioria dos casos. Como geração procedural de terreno utiliza somente duas dimensões, o fato do Perlin Noise ter complexidade $O(n^2)$ comparado ao Simplex Noise tendo complexidade $O(2^n)$ é menos relevante. Sendo assim os pontos determinantes entre qual algoritmo é melhor para geração de terreno procedural na maioria dos casos são:

Disponibilidade do algoritmo, caso os dois já estejam implementados e prontos para serem usados, o ideal é o Simplex Noise.

Quanto tempo é gasto com a geração do terreno? Dependendo de como ou para qual proposito, o tempo que demora para gerar o terreno pode ser menos de 10 segundos a cada hora, ou pode ser mais de 15 segundos por minuto, ou até mesmo o terreno pode ser gerado somente uma vez. Se nenhum dos dois algoritmos estiver disponível, e o tempo gasto com geração de terreno for baixo, o ideal é o Perlin Noise.

5 Trabalhos Futuros

Estudo de desempenho e qualidade dos algoritmos para outros usos, como geração de texturas.

Adicionar outros algoritmos aos estudos, como o Value Noise, Wavelet Noise (R. L. Cook 2005) ou Worley noise (S. Worley 1996).

Referências

A. Abran, R. E. Al Qutaish, Halstead's Metrics: Analysis of Their Designs, in Software Metrics and Software Metrology, John Wiley & Sons, Inc., Hoboken, NJ, USA. doi: 10.1002/9780470606834.ch7, 2010 Disponivel em: <http://profs.etsmtl.ca/aabran/Accueil/AlQutaish-Abran%20IWSM2005.pdf > Acesso em: 18 Nov. 2012

A. D. Kelley, M. C. Malin, and G. M. Nielson. Terrain Simulation Using a Model of Stream Erosion. In SIGGRAPH '88: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pages 263-268, New York, NY, USA. ACM, 1998, Disponivel em: <http://www.cs.duke.edu/courses/fall02/cps124/resources/p263-kelley.pdf > Acesso em: 08 Jul. 2012

A. Lagae et al, State of the Art in Procedural Noise Functions, 2010, Disponível em: <http://graphics.cs.kuleuven.be/publications/LLCDDELPZ10STARPNF/LLCDDELPZ10STARPNF.pdf>.Acesso em: 11 Dez. 2011.

D. S. Ebert et al. Texturing & Modeling: A Procedural Approach. 3 Edição. Morgan Kaufmann, 2003.

IKVM.NET Home Page, Disponivel em: <http://www.ikvm.net/> Acesso em 22 Nov 2012

L. Szirmay-Kalos, and T. Umenhoffer. Displacement mapping on the GPU - State of the Art, 2008, Disponivel em: <http://sirkan.iit.bme.hu/~szirmay/egdisfinal3.pdf> Acesso em: 27 Mai. 2012

M. Hendrikx. et al. Procedural Game Content Generation: A Survey. ACM Trans. Multimedia Comput. Commun. Appl. -, -, Artigo 1 (2011), 24 paginas. Disponivel em: <http://www.st.ewi.tudelft.nl/~iosup/pcg-g-survey11tomccap_rev_sub.pdf> Acesso em: 08 Jul. 2012

M. Person, Terrain generation, Part 1, 2011 Disponível em: <http://notch.tumblr.com/post/3746989361/terrain-generation-part-1> Acesso em: 11 Dez. 2011.

Overview of the .NET Framework, Disponivel em: <http://msdn.microsoft.com/en-us/library/zw4w595w.aspx> Acesso em 22 Nov 2012

P. Ahola, Generating terrain from 3D noise, 2011 Disponível em: <http://celeron.55.lt/~celeron55/random/2011-07/noisestuff/> Acesso em: 11 Dez. 2011.

P. Prusinkiewicz and M. Hammel. *A Fractal Model of Mountains with Rivers*. Em Proceeding of Graphics Interface '93, paginas 174-180, 1993, Disponivel em: <http://algorithmicbotany.org/papers/mountains.gi93.pdf > Acesso em: 09 Jul. 2012

R. M. Smelik et al. A Survey of Procedural Methods for Terrain Modelling, 2009, Disponivel em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.186.420&rep=rep1&type=pdf> Acesso em: 08 Jul. 2012

ROCHA, José Antonio Meira da. **Modelo de monografia e Trabalho de Conclusão de Curso (TCC).** Documento digital do programa Openoffice.org versão 2, disponível em: <http://meiradarocha.jor.br/news/tcc/files/2009/06/modelo_tcc-2011-11-23a.ott>. Acesso em: 11 Dez. 2011.

R. L. Cook, T. DeRose, Wavelet Noise, 2005, Disponivel em: < http://graphics.pixar.com/library/WaveletNoise/paper.pdf> Acesso em 23 Nov 2012.

S. Gustavson, Simplex noise demystified, 2005, Disponivel em: <http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf > Acesso em 22 Nov 2012

Speed Tree. Disponível em: <http://www.speedtree.com/> Acesso em: 09 Mar. 2012.

S. Wittber, Procedural Skybox for Unity3D. 2011, Disponivel em: <http://entitycrisis.blogspot.com/2011/01/procedural-space-skybox-for-unity3d.html> Acesso em: 09 Mar. 2012

S. Worley, A Cellular Texture Basis Function, 1996, Disponivel em: <http://www.rhythmiccanvas.com/research/papers/worley.pdf> Acesso em 23 Nov 2012.

Unity – Game Engine, Disponivel em: <http://unity3d.com> Acesso em 22 Nov 2012

Anexos

Código fonte Perlin Noise(Java)

```
package control.main.noise;

import java.util.LinkedList;
import java.util.Random;
import control.VectorFactory;

public class PerlinNoise {
	private static final int DEFAULT_PERM_SIZE = 256;
	private VectorFactory factory = new VectorFactory();
	private int[] permutationTable;
	private int permSize;
	public PerlinNoise() {
		this((long) (Math.random() * Long.MAX_VALUE));
	}
	public PerlinNoise(long seed) {
		this(seed, DEFAULT_PERM_SIZE);
	}
	public PerlinNoise(long seed, int permSize) {
		this.permSize = permSize;
		permutationTable = new int[permSize];
		LinkedList<Integer> perm = new LinkedList<Integer>();
		for (int b = 0; b < permSize; b++) {
			perm.add(b);
		}
		Random r = new Random(seed);
		int i = 0;
		while (!perm.isEmpty()) {
			permutationTable[i++] = perm.remove((int) (r.nextDouble() *
perm.size()));
		}
	}
	private static int fastfloor(double x) {
		return x > 0 ? (int) x : (int) x - 1;
	}
	public double get(double... values) {
		int dim = values.length;
		double cube[] = new double[dim];
		double offs[] = new double[dim];
		for (int i = 0; i < dim; i++) {
			cube[i] = fastfloor(values[i]);
			offs[i] = values[i] - cube[i];
		}
		int total = 1;
		for (int i = 0; i < dim; i++) {
			total *= 2;
		}
		double[] gridPoints = new double[total * dim];// points on the grid
		for (int add = 0; add < total; add++) {
			for (int dimm = 0; dimm < dim; dimm++) {
				gridPoints[add * dim + dimm] = cube[dimm] + ((add >>
dimm) & 1);
			}
		}
		int[] vectors = new int[total];
		double[] gridVectors = new double[total * dim];
```

```
			for (int i = 0; i < total; i++) {
				for (int j = 0; j < dim; j++) {
					gridVectors[i * dim + j] = cube[j] - gridPoints[i * dim
+ j] + offs[j];
				}
				vectors[i] = getPerm(gridPoints, i, dim);
			}
			factory.checkDimension(dim);
			double[][] ds = factory.vectors[dim];
			double sum = 0;
			int t = 1;
			for (int i = 0; i < dim; i++) {
				t *= 2;
			}
			for (int i = 0; i < t; i++) {
				double dotprod = factory.dotProd(gridVectors, ds[vectors[i] %
ds.length], i * dim);
				double distance = getVectorMagnitude(gridVectors, i * dim,
dim);
				double dist = 0;
				if (distance < 1) {
					dist = 1 - distance;
				} else {
					continue;
				}
				double ease = getEase(fastabs(dist));
				sum += dotprod * ease;
			}
			return sum;
		}
		private double fastabs(double dist) {
			if (dist < 0) {
				return -dist;
			}
			return dist;
		}
		public double getEase(double dist) {
			return 6 * dist * dist * dist * dist * dist - 15 * dist * dist * dist
* dist + (10 * dist * dist * dist);
		}
		public double getVectorMagnitude(double[] vt, int j, int dim) {
			double total = 0;
			for (int i = j; i < j + dim; i++) {
				total += Math.pow(vt[i], 2);
			}
			total = Math.pow(total, .5);
			return total;
		}
		private int getPerm(double[] ds, int ind, int dimen) {
			int which = 0;
			for (int i = 0; i < dimen; i++) {
				which = (permutationTable[(int) ((which + Math.abs(ds[ind *
dimen + i]) * 31) % permSize)]);
			}
			return (int) which;
		}
}
```

Codigo fonte Simplex Noise(Java)

```java
package control.main.noise;

import java.util.Arrays;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.Random;
import control.VectorFactory;

public class SimplexNoise {
	private static final int DEFAULT_PERM_SIZE = 256;

	private VectorFactory factory = new VectorFactory();

	private int[] perm;
	public SimplexNoise() {
		this((long) (Math.random() * Long.MAX_VALUE));
	}
	public SimplexNoise(long seed) {
		this(seed, DEFAULT_PERM_SIZE);
	}
	public SimplexNoise(long seed, int permSize) {
		perm = new int[permSize * 2];
		LinkedList<Integer> p2 = new LinkedList<Integer>();
		int[] p = new int[permSize];
		for (int b = 0; b < permSize; b++) {
			p2.add(b);
		}
		Random r = new Random(seed);
		int i = 0;
		while (!p2.isEmpty()) {
			p[i++] = p2.remove((int) (r.nextDouble() * p2.size()));
		}
		for (i = 0; i < 512; i++) {
			perm[i] = p[i & 255];
		}
	}
	private double skewFactor(int dim) {
		int ver = dim + 1;
		return (Math.sqrt(ver) - 1d) / dim;
	}
	private double unskewFactor(int dim) {
		int ver = dim + 1;
		return (ver - Math.sqrt(ver)) / (dim * ver);
	}
	private double dimSum(double[] coords) {
		double total = 0;
		for (Number i : coords) {
			total += i.doubleValue();
		}
		return total;
	}
	private static int fastfloor(double x) {
		return x >= 0 ? (int) x : (int) x - 1;
	}

	public double get(double... coords) {
		int dim = coords.length;
		if (dim == 0) {
			return 0;
```

```
		}
		double skew = dimSum(coords) * skewFactor(dim);
		double[] unityCube = new double[dim];
		for (int i = 0; i < dim; i++) {
			unityCube[i] = fastfloor(coords[i] + skew);
		}
		double G = unskewFactor(dim);
		double t = dimSum(unityCube) * G;
		final double[] distanceFromCorner = new double[dim];
		for (int i = 0; i < dim; i++) {
			double D0 = unityCube[i] - t;
			distanceFromCorner[i] = coords[i] - D0;
		}
		int[] vectors = new int[(dim - 1) * dim];
		Integer[] unskewIndex = new Integer[dim];
		for (int i = 0; i < dim; i++) {
			unskewIndex[i] = i;
		}
		Arrays.sort(unskewIndex, new Comparator<Integer>() {
			@Override
			public int compare(Integer g1, Integer g2) {
				if (distanceFromCorner[g1] > distanceFromCorner[g2]) {
					return -1;
				} else if (distanceFromCorner[g1] <
distanceFromCorner[g2]) {
					return 1;
				} else {
					return 0;
				}
			}
		});
		int depth = 1;
		for (int i = 0; i < dim - 1; i++) {
			for (int j = 0; j < depth; j++) {
				vectors[i * (dim - 1) + unskewIndex[j]] = 1;
			}
			depth++;
		}
		double[] distancesFromSimplex = new double[(dim + 1) * dim];
		for (int i = 0; i < dim; i++) {
			distancesFromSimplex[i] = distanceFromCorner[i];
		}
		for (int i = 1; i < dim + 1; i++) {
			for (int j = 0; j < dim; j++) {
				distancesFromSimplex[i * dim + j] =
distancesFromSimplex[j] - (i != dim ? vectors[(i - 1) * (dim - 1) + j] : 1) + i *
G;
			}
		}
		int[] gradient = new int[dim + 1];
		int[] cube255 = new int[dim];
		for (int i = 0; i < dim; i++) {
			cube255[i] = (int) (unityCube[i] % 255);
		}
		for (int i = 0; i < dim + 1; i++) {
			int value = 0;
			for (int j = 0; j < dim; j++) {
				value = perm[value + cube255[(dim - 1) - j] + (i == 0 ?
0 : (i == dim ? 1 : vectors[(i) * (dim - 1) - j]))];
```

```
                        }
                        gradient[i] = (int) value;
                }
                double contr = 0;
                factory.checkDimension(dim);
                for (int i = 0; i < dim + 1; i++) {
                        double t0 = 0.5;// - x0*x0 - y0*y0 - z0*z0;
                        for (int j = 0; j < dim; j++) {
                                double d = distancesFromSimplex[i * dim + j];
                                t0 -= d * d;
                        }
                        if (t0 >= 0) {
                                t0 *= t0;
                                double dot = factory.dotProduct(distancesFromSimplex,
gradient[i], dim, i * dim);
                                contr += t0 * t0 * dot;
                        }
                }
                return contr * 100;
        }
}
```

Codigo fonte Vector Factory(Java)

```
package control;

import java.util.Random;

public class VectorFactory {
        public double[][][] vectors;
        public void checkDimension(int dim) {
                if (!isDimensionReady(dim)) {
                        prepareDimension(dim);
                }
        }

        public double dotProd(double[] a, double[] b, int j) {
                double total = 0;
                for (int i = 0; i < b.length; i++) {
                        total += a[j + i] * b[i];
                }
                return total;
        }

        public void prepareDimension(int dim) {
                prepareVectors(dim);
                if (dim == 1) {
                        vectors[dim] = new double[9][dim];
                        for (double i = 0; i < 9; i++) {
                                vectors[dim][(int) i][0] = (8d - i * 2) / 8d;
                        }
                } else if (dim == 2) {
                        vectors[dim] = new double[16][dim];
                        for (int i = 0; i < 16; i++) {
                                double rad = Math.toRadians(360 / 16 * i);
                                vectors[dim][i][0] = Math.sin(rad);
                                vectors[dim][i][1] = Math.cos(rad);
                        }
                } else {
                        moreThanTwoDimensions(dim);
```

```
			}
		}
		private void moreThanTwoDimensions(int dim) {
			int magic = (int) Math.pow(2, dim - 1);
			vectors[dim] = new double[dim * magic][dim];
			int zeroPos = 0;
			int whereIsMinus = 0;
			while (zeroPos < dim) {
				for (int i = 0; i < dim; i++) {
					vectors[dim][zeroPos * magic + whereIsMinus][i] = (i ==
zeroPos ? 0 : ((whereIsMinus >> ((dim - 2) - i + (i > zeroPos ? 1 : 0)) & 1) ==
1 ? -1 : 1));
				}
				double mag = getVectorMagnitude(vectors[dim][zeroPos * magic +
whereIsMinus]);
				for (int i = 0; i < dim; i++) {
					vectors[dim][zeroPos * magic + whereIsMinus][i] /= mag;
				}
				mag = getVectorMagnitude(vectors[dim][zeroPos * magic +
whereIsMinus]);
				whereIsMinus++;
				if (whereIsMinus == magic) {
					whereIsMinus = 0;
					zeroPos++;
				}
			}
		}
		private double getVectorMagnitude(double[] ds) {
			double total = 0;
			for (int i = 0; i < ds.length; i++) {
				total += Math.pow(ds[i], 2);
			}
			total = Math.pow(total, .5);
			return total;
		}

		private void prepareVectors(int dim) {
			if (vectors == null) {
				vectors = new double[dim + 1][][];
			} else {
				if (vectors.length <= dim) {
					double[][][] oldv = vectors;
					vectors = new double[dim + 1][][];
					for (int i = 0; i < oldv.length; i++) {
						vectors[i] = oldv[i];
					}
				}
			}
		}
		protected boolean isDimensionReady(int dim) {
			return !(vectors == null) && vectors.length > dim && vectors[dim] !=
null;
		}
		public double dotProduct(double[] vec, int gradient, int dim, int from) {
			return dotProd(vec, this.vectors[dim][gradient %
this.vectors[dim].length], from);
		}
		public double getVectorMagnitude(double[] vt, int j, int dim) {
			double total = 0;
```

```
			for (int i = j; i < j + dim; i++) {
				total += Math.pow(vt[i], 2);
			}
			total = Math.pow(total, .5);
			return total;
		}
		public double getEase(double dist) {
			return 6 * dist * dist * dist * dist * dist - 15 * dist * dist * dist
* dist + (10 * dist * dist * dist);
		}
}
```

Codigo fonte SetHeight(C#)

```
using UnityEngine;
using System.Collections;
using control.main.noise;

public class setHeight : MonoBehaviour {

	public Terrain t;

	//double[] param = new double[2];
	void Start () {
		PerlinNoise p = new PerlinNoise();
		double modfix = 2;
		//long time = System.DateTime.Now.Ticks;
		int i = 0;
		while(i++ < 100){
			//param[0]=Random.Range(0, 100);
			//param[1]=Random.Range(0, 100); 22590725
			//print(""+);
		}
		float[,] values = new float[513,513];
		for(int x = 0; x < 513; x++){
			for(int y = 0; y < 513; y++){
				double v1 = p.get(x/253d+1400, y/253d+1400)*modfix-.3;
				double v2 = p.get(x/67d+400, y/67d+400)*modfix;
				v2 = .9-(v2 > 0? v2 : -v2);
				double v3 = p.get(x/23d+60, y/23d+60)*modfix/3;
				if(v3 < 0){
					v3 = 0;
				}
				double v4 =p.get(x/7d, y/7d)*modfix/7;
				v4 = .5-(v4 > 0? v4 : -v4);
				if(v4 < 0){
					v4 = 0;
				}
				double absv1 = v1 < 0 ? -v1 : v1;
				double absv2 = v2 < 0 ? -v2 : v2;
				v3 *= absv1*absv2;
				v4 *= v1*v2;
				if(v1 < 0){
					v1 /= 2;
				}
				if(v1 > 0){
					v1 /= 130;
				}if(v2 > 0){
					v2 /= 60;
				}
```

```
                        float gotten = (float)(1f+v1+v2+v3+v4+.02)/5;
                        values[x,y] = gotten;
                    }
                }
                t.terrainData.SetHeights(0, 0, values);
            }
        }
```