UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENAÇÃO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

LAYS HELENA LOPES VELOSO

DESENVOLVIMENTO DE UM CENÁRIO COM TERRENO PROCEDURAL BASEADO NO RUÍDO PERLIN

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2012

LAYS HELENA LOPES VELOSO

DESENVOLVIMENTO DE UM CENÁRIO COM TERRENO PROCEDURAL BASEADO NO RUÍDO PERLIN

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, da Coordenação de Análise e Desenvolvimento de Sistemas - COADS, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. André Koscianski

PONTA GROSSA

2012

TERMO DE APROVAÇÃO

DESENVOLVIMENTO DE UM CENÁRIO COM TERRENO PROCEDURAL BASEADO NO RUÍDO PERLIN

por

LAYS HELENA LOPES VELOSO

Este Trabalho de Conclusão de Curso foi apresentado em 13 de novembro de 2012, como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. A candidata foi arguida pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
André Koscianski
Prof. Orientador

_____
Thalita Scharr Rodrigues
Membro

_____
Wellton Costa de Oliveira
Membro

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

Dedicado à minha família, ao meu namorado e às pessoas que direta e indiretamente ajudaram no desenvolvimento deste trabalho pelo simples fato de manterem-se ao meu lado.

AGRADECIMENTOS

Agradeço a Deus em primeiro lugar por ter me mantido de pé, pelas oportunidades e pelas pessoas que Ele colocou em meu caminho.

Aos meus pais José Jorge Veloso e Helena Maria Lopes Veloso pela minha existência e pelo investimento em meu futuro e educação.

A minha irmã Thays Veloso, minha fonte de inspiração e exemplo de como lidar com as dificuldades.

Aos professores que me acompanharam em minha vida acadêmica em especial ao Prof. Dr. André Koscianski e Profa. Mônica Hoeldtke Pietruchinski que tornaram possível a realização do trabalho, pela confiança, paciência e dispensa de tempo dedicado a minha orientação.

Ao meu namorado Rogério Ranthum pelo carinho e apoio constantes, incentivo a crescer, vencer minhas fraquezas, a descobrir e explorar meu potencial.

A simplicidade é o último grau de sofisticação.

Aprender é a única coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende. (DA VINCI, Leonardo)

RESUMO

VELOSO, Lays H. **Desenvolvimento de um cenário com terreno procedural baseado no ruído Perlin**. 2012. 37 páginas. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2012.

Este trabalho é motivado pela importância da utilização de técnicas procedurais no desenvolvimento de jogos eletrônicos. Essa necessidade é justificada pela capacidade de adicionar detalhes complexos que não podem ser produzidos manualmente e sem utilizar muita memória. Neste trabalho é implementado o ruído Perlin para a geração procedural de um terreno e é explicado de que forma ele é aplicado à superfície de forma a proporcionar uma qualidade adequada.

Palavras-chave: desenvolvimento de jogos. terreno procedural. ruído Perlin.

ABSTRACT

VELOSO, Lays H. **Development of a scene with procedural terrain based on Perlin noise**. 2012. 37 pages. Final Paper (Systems Analysis and Development Technology) - Federal Technology University of Paraná. Ponta Grossa, 2012.

This work is motivated by the importance of using procedural techniques in development of eletronic games. This need is justified by the ability to add complex detail using low memory. This work consists of an implementation of Perlin noise to create a procedural terrain and an explanation of how it's applied to the surface to provide adequate quality.

Keywords: game development. procedural terrain. Perlin noise.

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de jogos computacionais é uma área em constante expansão. Há cada vez mais mercado e envolve diversas áreas como Computação Gráfica, Física e Matemática. São divididos em vários gêneros como, por exemplo, Ação, Aventura, Estratégia, Simulação. Seja seu objetivo final o de educar ou entreter os jogos tentam representar a realidade e a preocupação primária do designer de jogos é o jogador.

O problema que motivou a realização deste trabalho foi a presença de barreiras em jogos eletrônicos, que delimitam o espaço em que o jogador se locomove e tornam seus mundos finitos. Essas barreiras são comuns em jogos do gênero *RPG* (Jogo de Interpretação). As formas mais utilizadas para delimitar a área do jogo são a presença de mares extensos em volta de uma área de terra, obstáculos físicos como paredes invisíveis bloqueando a passagem ou até mesmo uma superfície que termina e o personagem cai no vazio, afundando para sempre. Estas técnicas tornam a experiência do jogador desagradável, porém são utilizadas principalmente para minimizar a necessidade de recursos gráficos e o uso de memória.

Um jogo muito grande, inteiro carregado na memória, gasta muito espaço em disco, o seu instalador pode ficar muito grande e irá demorar a ser carregado.

Outra saída utilizada por *designers* de jogos são mundos circulares: quando o terreno acaba do lado esquerdo, o personagem reaparece do lado direito. Pode atender as necessidades de jogos mais simples, porém não é uma solução muito realista, sobretudo para jogadores mais atentos. Um exemplo de jogo que utiliza essa solução é o conhecido PACMAN e essa situação pode ser observada na Figura 1. No jogo Snake apresentado na Figura 2 o personagem não pode esbarrar nas paredes que delimitam seu espaço de locomoção.

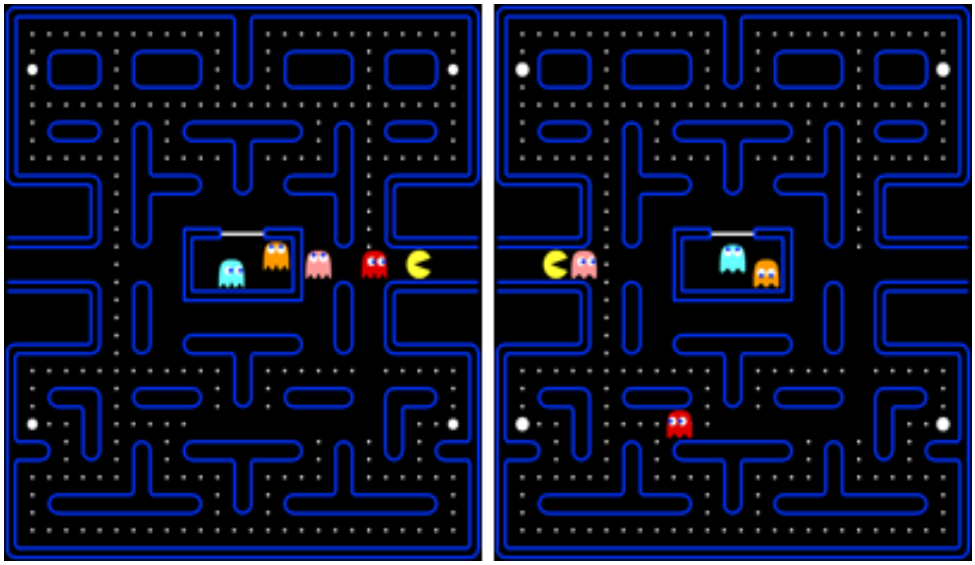

Figura 1 – Capturas do jogo Pacman (Exemplo de mundo circular)
Fonte: www.clickjogos.uol.com.br (2012)

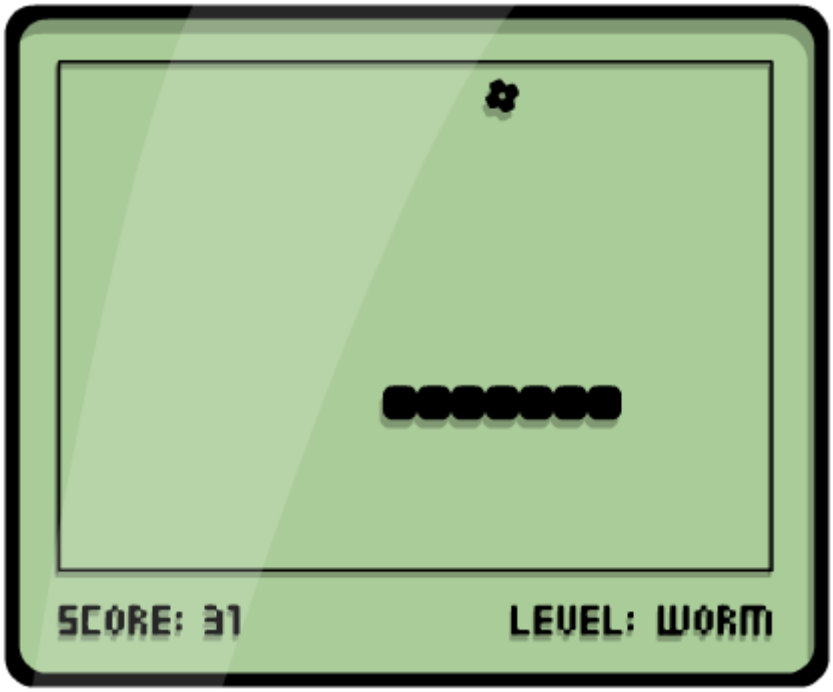

Figura 2 – Captura do jogo Snake (Delimitação do espaço de locomoção)
Fonte: www.snakegame.net (2012)

1.1 OBJETIVOS GERAIS

Com o desenvolvimento deste trabalho propõe-se a criação de um cenário de jogo 3D infinito que proporcione ao jogador a experiência ilimitada de andar livremente sobre o terreno do jogo sem que ele termine inesperadamente.

Para a solução ao problema abordado será desenvolvido um jogo com terreno criado em tempo de execução por meio da utilização de técnicas procedurais.

1.2 OBJETIVOS ESPECÍFICOS

Para a criação de Scripts, objetos e componentes do cenário será utilizado o motor de jogo Unity que possui um conjunto de recursos que abstraem o desenvolvimento de jogos 3D (UNITY TECHNOLOGIES, 2012).

Para que seja possível a geração procedural de um terreno com aparência aleatória, realista e em tempo de execução, será utilizado o algoritmo *Perlin Noise*, desenvolvido por PERLIN (1985) que utiliza técnicas de ruído.

1.3 ESTRUTURA DO TRABALHO

O presente trabalho encontra-se dividido em seis capítulos. No Capítulo 1 é dada uma introdução ao tema de desenvolvimento de jogos, uma abordagem dos problemas comuns encontrados neste universo e algumas vantagens e desvantagens das soluções comumente utilizadas chegando ao problema a ser resolvido com o trabalho. No Capítulo 2 são encontrados conceitos que se encaixam no escopo do trabalho e autores das áreas de conhecimento estudadas. No Capítulo 3 são descritas as etapas do desenvolvimento do trabalho. No Capítulo 4 são apresentados e discutidos os resultados obtidos com esse trabalho. No Capítulo 5 as considerações finais do trabalho são apresentadas, juntamente com sugestões para trabalhos futuros. No Capítulo 6 estão concentradas as referências bibliográficas utilizadas.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta conceitos e terminologias da área de Computação Gráfica e desenvolvimento de jogos estudados para a realização deste trabalho.

2.1 MALHAS POLIGONAIS

Segundo Faxin (2010), uma malha poligonal em Computação Gráfica é uma coleção de vértices, arestas e faces semelhante a uma grade desestruturada que define a forma de um objeto tridimensional. As faces são geralmente constituídas de triângulos.

Malhas poligonais estão sempre presentes em gráficos e modelagem tridimensionais. Seus polígonos são utilizados para modificar a geometria dos objetos tridimensionais e obter a superfície aproximada de objetos de contexto natural com formas complexas (FLETCHER, 2002). É possível aumentar o número de polígonos da malha para melhor manipular as curvas do objeto de forma a se obter uma aparência suavizada e se aproximar da complexidade da forma desejada, o que resulta em baixa taxa de *frame*. Deve-se encontrar um equilíbrio entre a aparência visual e taxa de quadro aceitável.

Os objetos tridimensionais em Computação Gráfica são normalmente representados através de uma malha poligonal, e devem ser gerados, seja numa etapa inicial ou durante a execução do jogo, sendo armazenados e distribuídos de algum modo.

2.2 TÉCNICAS PROCEDURAIS

Dizer em computação que algo é gerado proceduralmente é equivalente a dizer que é gerado através da programação de um código.

Técnicas procedurais são utilizadas em Computação Gráfica para especificar características de um modelo ou efeito (EBERT et al., 2003). Foram inicialmente introduzidas por PERLIN (1985) para produzir texturas para objetos tridimensionais. Por exemplo, uma textura procedural para uma superfície de mármore, usa algoritmos e funções matemáticas para determinar sua cor.

Uma dessas técnicas é a Modelagem Procedural, onde a geometria do objeto 3D é criada em tempo real pelo computador (FLETCHER, 2002). A criação de terrenos é um exemplo de modelagem procedural onde se obtém bons resultados.

O poder de processamento é maior que o poder de modelagem humano. Modelos produzidos proceduralmente possuem um nível de detalhe que não podem ser produzidos a mão (EBERT et al., 2003).

O uso de técnicas procedurais é útil quando se deseja um resultado com aspecto aleatório como paisagens. Personagens virtuais são normalmente modelados manualmente.

Com uma semente de números randômicos junto a uma função geradora, é possível a criação e reprodução de um terreno completamente procedural que tenha a consistência visual de ambientes virtuais 3D sem que seja necessário armazenar informações como altura e escala em uma estrutura de dados. O objetivo da função geradora é fornecer estas informações para cada coordenada da cena 3D.

2.3 RUÍDO

Ruído em Computação Gráfica é uma função pseudo-aleatória comumente utilizada para gerar texturas irregulares procedurais (EBERT et al., 2003), não se limitando a imagens estáticas, sendo usada para a animação e simulação de fenômenos naturais e para adicionar detalhes à geometria de objetos tridimensionais.

As funções de ruído adicionam a aleatoriedade existente no mundo real (BEVILACQUA, 2009).

Um exemplo de textura irregular gerada por uma função de ruído é um Mapa de Altitude, que é uma textura em escala de cinza utilizada para armazenar valores tais como dados de elevação de uma superfície. O tom de cinza que um pixel tem determina a altura da superfície sendo branco o ponto mais alto e o preto o mais baixo da superfície. Ferramentas de modelagem 3D utilizam Mapas de Altitude para representar um terreno.

Usar texturas procedurais para formar as curvas da malha permite a criação de geometria detalhada em pouco tempo (VAUGHAN, 2011).

2.3.1 Perlin Noise

Desenvolvida por Ken Perlin no início de 1980, a função *Perlin Noise* produz uma sequência ordenada de números randômicos (SHIFFMAN, 2008). O algoritmo *Perlin Noise* implementa uma função de ruído com um, dois e/ou três parâmetros, uni, bi ou tridimensional referindo-se ao espaço. Seus valores de saída variam entre -1.0 e 1.0. Para se obter um valor apropriado, se multiplica as saídas por uma escala. Considerando o valor 200 para a escala, o resultado da multiplicação varia de 0 a 200, como ilustrado na Tabela 1.

Tabela 1 – Resultado escalar para saídas do Perlin Noise

Ruído	Escala	Resultado
0	200	0
0.12	200	24
0.57	200	114
0.89	200	178
1	200	200

Fonte: Shiffman (2008)

Perlin Noise permite criar ruídos suaves utilizando interpolações lineares para encontrar valores entre os números randômicos (KOZOVITS, 2003).

A qualidade do resultado de uma função de ruído depende da interpolação utilizada (EBERT et al., 2003).

A função de ruído pode ser usada para criar superfícies com características desejadas em diferentes escalas visuais (PERLIN, 1985).

As propriedades de uma função de ruído ideal são (EBERT et al., 2003):

- É uma função repetitiva pseudo-aleatória em função de suas entradas;
- Tem um intervalo de valores definido, entre -1.0 e 1.0;
- Não apresenta padrões regulares;
- É invariante ao fator de escala e rotação.

Em resumo, o algoritmo gera funções de ruído aleatório com várias frequências e amplitudes, realiza sua soma e suaviza o resultado[1].

A função de ruído Perlin elimina a necessidade de qualquer armazenamento prévio. Dadas as coordenadas da malha a ser modificada, pode-se obter a altura do ponto.

O objeto plano que constitui o terreno por si só é uma malha com aspecto liso. Para dar um aspecto montanhoso ao terreno é necessária a modificação de sua geometria trabalhando suas propriedades altitude e escala.

Uma técnica utilizada para evitar este trabalho é aplicar uma textura no objeto a fim de dar a aparência de um detalhe que não é realmente presente em sua superfície. O objetivo do trabalho é justamente eliminar os disfarces comuns em jogos que fazem com que percam seu realismo. No exemplo citado o personagem não iria de fato colidir com as deformações da textura não presentes no terreno.

2.3.2 Interpolação Linear

Interpolação é uma média para interpolar entre pontos. A Interpolação linear é invariante a transformações, mantidas entre 0 e 1 (CLAUDINO, 2007).

2.4 MOTOR DE JOGO

Motores de jogo (*game engines*) são ferramentas que abstraem o desenvolvimento de jogos eletrônicos a partir do estado da arte até a matemática de tomada de decisões (GOLDSTONE, 2009).

É responsável pela renderização de objetos na tela, simulação de física e detecção de colisão.

---

[1] Detalhes matemáticos do algoritmo Perlin Noise são encontrados na página http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html.

Motores de jogo também fornecem ao desenvolvedor a abstração de hardware, descartando a necessidade de conhecer a arquitetura da plataforma para a qual o jogo será desenvolvido.

Com a distribuição digital e a disponibilidade de ferramentas de baixo custo (ou gratuitas) de desenvolvimento de jogos como a Unity 3D, a democratização do desenvolvimento do jogo está bem encaminhada (CREIGHTON, 2010).

2.4.1 O Motor de Jogo Unity

Unity é um motor de jogo que permite criar jogos e implantá-los para uma série de diferentes dispositivos, incluindo (no momento da escrita) a web, *PCs*, plataformas *iOS* e *WiiWare*, com módulos para *Android* e *Xbox Live Arcade*.

Tendo apelado para muitos desenvolvedores de jogos, a Unity tem preenchido uma lacuna no mercado de desenvolvimento de jogos que poucas outras podem atender inteiramente. Tendo a capacidade de produzir jogos de padrão profissional, publicar em 3D para Mac e PC, bem como ter seu próprio Web Player, Unity é um dos motores de jogos que mais cresce em seu setor. (GOLDSTONE, 2009)

2.4.2 Área de Trabalho

A área de trabalho da Unity é composta por janelas chamadas *Views*. Abaixo são descritas suas funcionalidades.

- Barra de Ferramentas: Contém ferramentas para a seleção dos objetos da cena;
- Editor de Cena: Manipulação gráfica dos objetos que compõe a cena com o mouse (girar, posicionar, esticar);
- Modo Jogo: Visualização prévia do comportamento da aplicação e dos elementos que a compõe;
- Projeto: Manipulação e organização dos arquivos que compõe o projeto: Scripts, Prefabs, texturas;
- Hierarquia: Contém os elementos da cena que está sendo editada: Câmera, objetos, luz;
- Inspetor: No Inspetor é possível ajustar os atributos do objeto selecionado na cena.

A Figura 3 é uma captura da área de trabalho da Unity 3D.

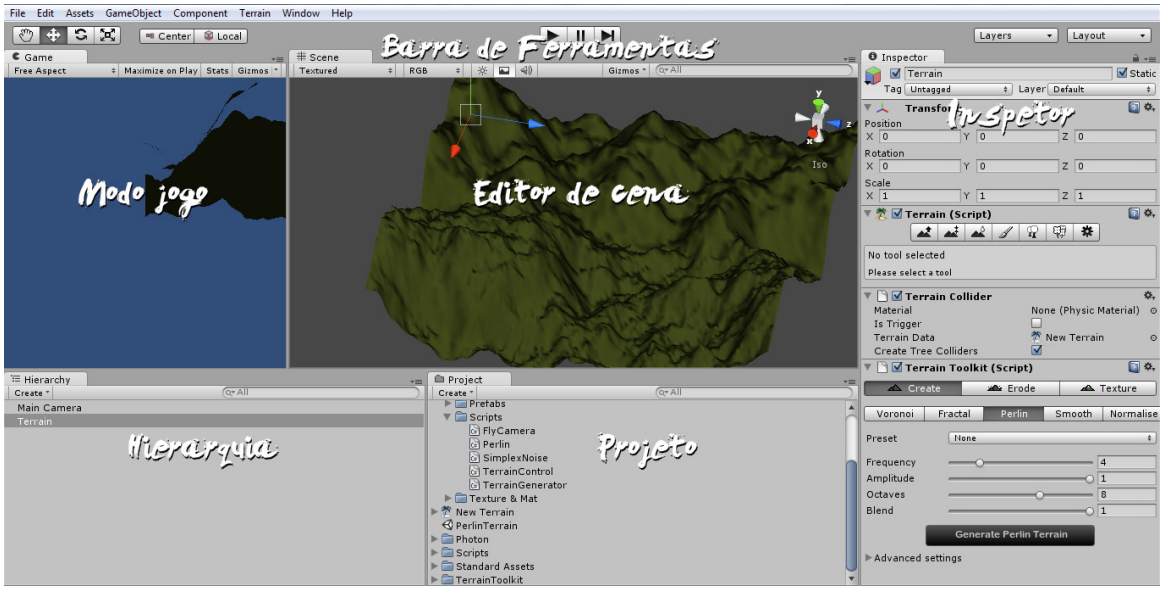

Figura 3 – Captura da área de trabalho da Unity 3D
Fonte: Extraída da ferramenta Unity e edição Própria (2012)

2.4.3 Especificações Técnicas

A Unity fornece uma forte combinação de gráficos, áudio e física. Funciona bem com a maioria dos pacotes de modelagem 3D (incluindo o mais popular, Blender, totalmente gratuito). Sua execução é rápida, e permite construir para plataforma Windows, Mac, Web e vários dispositivos portáteis.

Para os cálculos de colisões e todos os demais cálculos físicos do jogo, Unity utiliza o motor físico PhysX (NVIDIA CORPORATION, 2011). Para que um objeto esteja sob o controle do motor de física da Unity é necessário anexar ao objeto o componente *Rigid Body*. Assim o objeto passará a possuir as seguintes propriedades: massa, gravidade, velocidade e fricção.

2.4.4 Linguagem de Programação

Scripts da Unity são escritos em três linguagens: *JavaScript*, *C#*, e *Boo*. A escrita de *Scripts* dentro da Unity consiste em anexá-los a objetos do jogo atribuindo-os comportamentos personalizados. Diferentes funções dentro dos objetos com *Scripts* são chamados em certos eventos. Sendo os mais utilizados os seguintes:

1. *Update*: Esta função é chamada antes de exibir um quadro. Este é o lugar aonde estarão a maioria dos códigos de comportamento, exceto os códigos de física.
2. *FixedUpdate*: Esta função é chamada uma vez a cada passo de tempo de física. Este é o lugar para fazer um comportamento de jogo baseado em física. Cálculos de física, como detecção de colisão e movimento de corpos rígidos são realizados em discretos passos de tempo que não dependem de taxa de quadros. Isso torna a simulação mais consistente entre computadores diferentes ou quando ocorrerem alterações na taxa de quadros. Por exemplo, a taxa de quadros pode cair devido a uma aparência de muito jogo na tela, ou porque o usuário executou outro aplicativo em segundo plano.
3. Codificação fora de qualquer função: Código escrito fora de funções é executado quando o objeto é carregado. Pode ser usado para inicializar o estado do *Script* supondo que a linguagem utilizada é *JavaScript*. O que não for colocado em função *Javascript,* coloca-se dentro da função *Awake* ou *Start* em *C#* ou *Boo*.

A versão de *JavaScript* da Unity é um pouco diferente do utilizado para desenvolvimento web, pois trata de funcionalidades relacionadas à ferramenta e também por sua execução ser mais rápida. A Unity contém um módulo de referência de linguagem semelhante a um dicionário que contém cada palavra reservada. Sua linguagem é *case sensitive*, o que significa que uma palavra com letra maiúscula é tratada de forma diferente que a mesma palavra em letra minúscula. Utiliza o paradigma orientado a objetos permitindo a criação de classes, objetos e instâncias.

2.4.5 Interface Gráfica do Usuário

Qualquer jogo 3D provavelmente vai exigir uma quantidade razoável de programação em 2D, por exemplo, lojas, telas de inventário, telas de seleção de nível, etc. Interfaces gráficas do usuário incluem todos os botões, barras de rolagem, menus, setas e textos que auxiliam o jogador a compreender e a percorrer o jogo. (CREIGHTON, 2010)

A Unity usa o que é chamado de GUI de modo imediato. O termo é emprestado da programação gráfica, e isso requer uma programação um pouco

diferente. A estrutura condicional no código (1) significa “Se o botão foi clicado”. O método *GUI.Button* precisa de dois argumentos, um retângulo que define o canto superior esquerdo e tamanho do botão, e um rótulo para o botão. *Rect* leva quatro entradas: posição x, posição y, largura e altura.

[1] if(GUI.Button(Rect(0,0,100,50),"Play Game"))

A função *OnGUI* é chamada repetidamente durante a execução do jogo. A linha do código é chamada duas vezes por quadro, sendo uma vez para criar o botão e uma vez para verificar se o botão foi clicado. Em cada quadro, sua interface toda é recriada a partir do zero com base no código em sua função *OnGUI*.

2.4.6 Recursos

A Unity fornece uma série de recursos para simplificar o desenvolvimento de jogos. Os principais são:

- GameObjects: São os objetos que estão sendo usados na cena do jogo.
- Componentes: Um componente na Unity é um Script que pode ser anexado a um objeto do jogo para definir sua aparência, comportamento, torná-los visíveis, adicionar física, etc.
- Objetos Pré-fabricados: Um objeto pré-fabricado pode ser reutilizado em um projeto sem que este esteja presente na cena do jogo. Pode ser armazenado no diretório do Projeto da Unity juntamente com os componentes anexos a ele e suas configurações. São principalmente utilizados quando se deseja duplicá-los em tempo de execução. Podem ser inseridos em qualquer número de cenas e múltiplas vezes por cena. Quando se adiciona um objeto pré-fabricado em uma cena, se cria uma instância deste. Todas as instâncias do pré-fabricado são linkadas ao original e são clones deste.

2.4.7 Licença

Uma grande vantagem da Unity é o preço. Com nenhum custo, qualquer um pode fazer download da Unity com licença que permite a distribuição comercial de qualquer produto criada com ela. Embora existam algumas características disponíveis apenas na versão profissional, a versão gratuita é suficiente para introduzir alguém ao desenvolvimento de jogos.

3 DESENVOLVIMENTO

Neste capítulo são descritas as fases e etapas do desenvolvimento do projeto.

3.1 IMPLEMENTAÇÃO DO PERLIN NOISE EM C

A primeira implementação do algoritmo ruído Perlin foi feita na linguagem de programação C para um melhor entendimento de seu funcionamento antes da sua utilização no motor de jogo Unity.

O algoritmo possui três funções: *findnoise*, *interpolate* e *noise*.

A função *findnoise* foi testada com uma e, posteriormente, duas dimensões.

3.1.1 Com uma dimensão

São apresentados os resultados da implementação mais simples da função, com uma dimensão.

```
double findnoise(int n)
{
    int x = (int) (n<<13) ^ n;
    return (double) ( 1.0 - ( (x * (x * x * 15731 + 789221) + 1376312589) & 0x7fffffff) /
    1073741824.0);
}
```

O valor retornado por ela sempre será o mesmo e entre -1.0 e 1.0 para uma dada coordenada. Supondo a chamada da função como abaixo:

```
printf("\n%f",findnoise(0));
printf("\n%f",findnoise(1));
printf("\n%f",findnoise(1));
printf("\n%f",findnoise(2));
```

Dada a coordenada 1 mais que uma vez, o resultado obtido é o mesmo. Os valores retornados pelas chamadas a função são os seguintes:

-0.281791
-0.226373
-0.226373
0.293633

3.1.2 Com duas dimensões

Para a função com duas dimensões se tem *findnoise*(x, y) = z, ou seja, dadas as coordenadas x e y do espaço obtém-se a coordenada z.

É importante que para as mesmas coordenadas seja devolvido sempre o mesmo valor.

Se essa função for invocada para as coordenadas do terreno tridimensional a profundidade para cada ponto será igual a cada chamada da função garantindo a mesma aparência quando o jogador retorne ao ponto.

A função *noise* possui chamadas as funções *findnoise* e *interpolate* e seu retorno é o ruído suavizado. Para isso realiza os seguintes passos:

1. Encontra dois pares de números:

s=findnoise2(x, y);
t=findnoise2(x+1, y);
u=findnoise2(x, y+1);
v=findnoise2(x+1, y+1);

2. Encontra a média do primeiro par de números:

float int1=interpolate(s,t,x-x);

3. Encontra a média do segundo par de números:

float int2=interpolate(u,v,x-x);

4. Faz a média desses dois novos números juntos:

float int3=interpolate(int1,int2,y-y);

3.1.3 Geração do ruído

A geração de um ruído começa com um ou mais números pseudo-aleatórios uniformemente distribuídos em cada ponto no espaço cujas coordenadas são números inteiros. Então é realizada uma interpolação suave entre os números pseudo-aleatórios.

Nas Figuras 4 e 5 é possível visualizar uma permutação entre valores randômicos gerados por uma função randômica comum e pela função de ruído

pseudo-aleatório do algoritmo *Perlin Noise* respectivamente em grafos gerados por algoritmos de Shiffman (2008)[2].

No grafo de números puramente randômicos o ruído não é coerente, se usados na geometria de uma superfície esta teria uma aparência em desacordo com a de um terreno real. No gráfico de números do *Perlin Noise* o ruído é coerente, interpola suavemente entre os valores para que não haja descontinuidade dando uma aparência semelhante a uma curva.

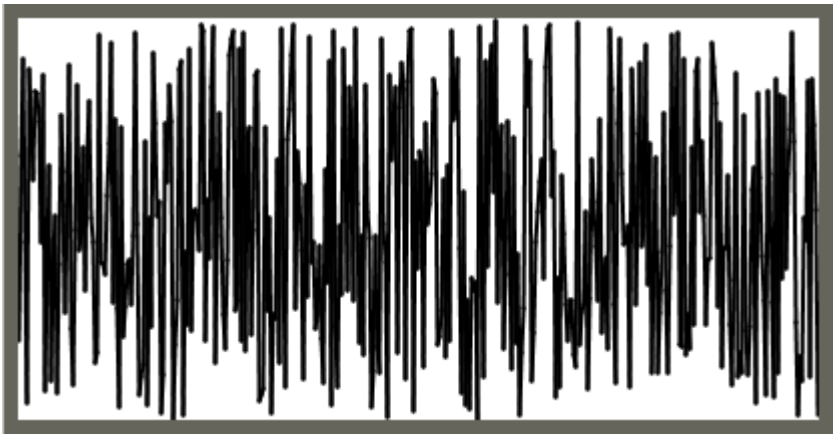

Figura 4 – Ruído não coerente
Fonte: www.learningprocessing.com (2012)

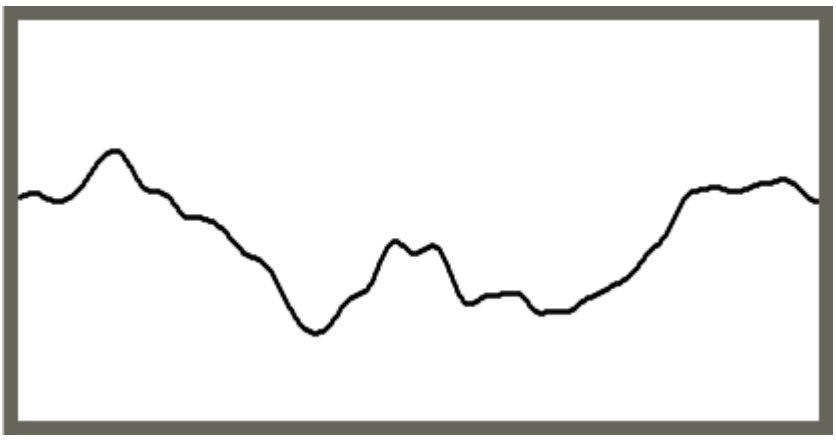

Figura 5 – Ruído Perlin coerente
Fonte: www.learningprocessing.com (2012)

[2] Encontrado no endereço http://www.learningprocessing.com/.

3.2 IMPLEMENTAÇÃO NO MOTOR UNITY

A implementação do cenário foi feita com o motor de Jogo *Unity 3D*[3]. Com ele é possível a criação de malhas proceduralmente. Para isso é necessário entender como funcionam as técnicas utilizadas pela ferramenta para o tratamento de malhas.

Como visto na seção 2.2 criar algo proceduralmente é criá-lo via programação.

Uma malha na Unity consiste de uma matriz de vértices e índices: Os vértices são um Vetor de três posições e os índices de faces triangulares são números inteiros.

As etapas necessárias no processo de implementação no motor Unity foram: a criação de uma malha 3D, um algoritmo que, dadas as coordenadas da malha, se obtém a altura da superfície e um algoritmo para a manipulação desta malha.

3.2.1 Criação da malha 3D

Uma malha poligonal no motor Unity é composta de triângulos como a presente no objeto plano apresentado na Figura 6. Entender a manipulação da malha no Unity é essencial no desenvolvimento do trabalho, pois será realizada proceduralmente.

[3] Documentação disponível em http://unity3d.com/support/documentation.

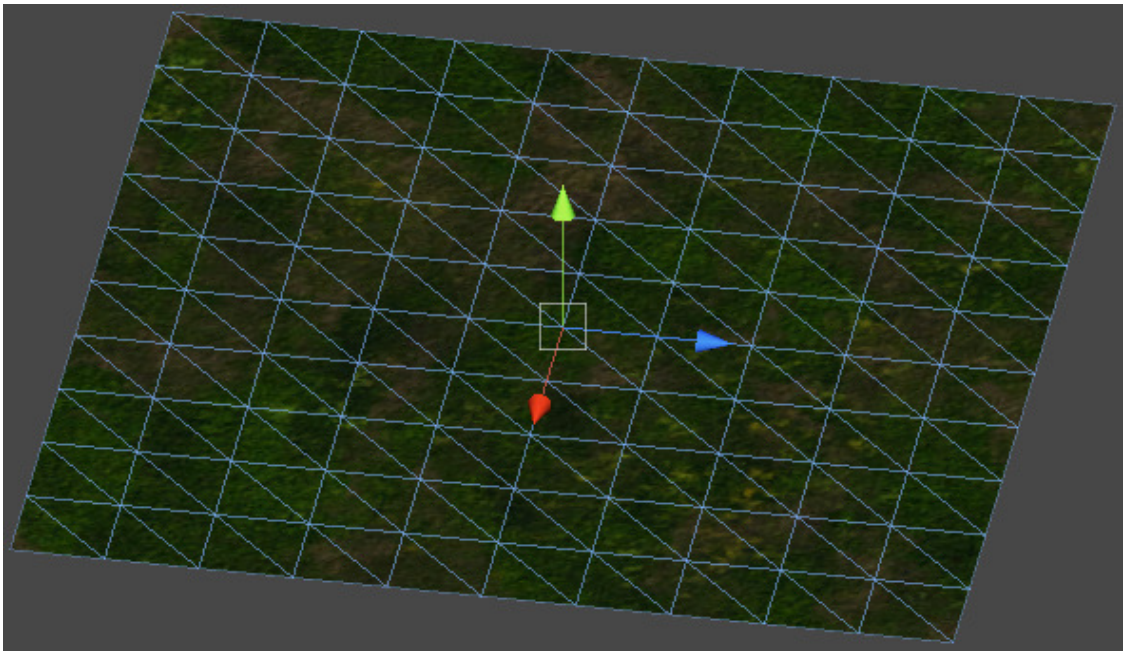

Figura 6 – Exemplo de malha triangular
Fonte: Autoria Própria (2012)

O *GameObject* criado com a ferramenta por si só não é uma malha e não é possível manipular sua forma. O único componente que ele possuirá é o *Transform*. Para que o objeto se torne uma malha é necessária a adição de dois componentes: *MeshFilter* e *MeshRenderer*. O primeiro irá anexar uma malha 3D ao objeto e o segundo é responsável por torná-la visível na cena.

A Figura 7 mostra todos os componentes necessários anexos ao objeto plano nomeado *Terrain Plane* na *view Inspector* da ferramenta.

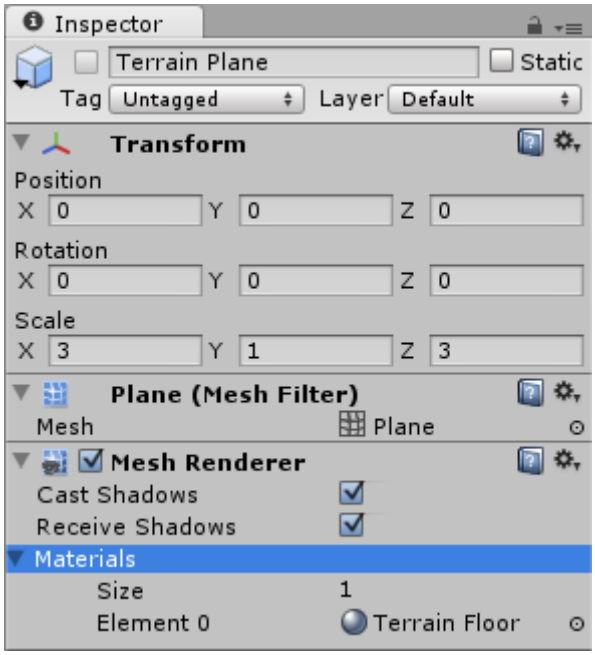

Figura 7 – Inspetor do *GameObject Terrain Plane*
Fonte: Extraída da ferramenta Unity 3D e edição Própria (2012)

3.2.2 Classe Perlin

A Classe Perlin utilizada no jogo foi obtida na página oficial da ferramenta Unity em um pacote com vários exemplos procedurais envolvendo a manipulação de malhas[4]. Ela é a classe para a geração de ruído Perlin.

Como visto na implementação do *Perlin Noise* em C na seção 3.1, o valor retornado sempre será o mesmo dado o ponto do terreno, ou seja, suas coordenadas. Passando X e Y se obtêm Z. A largura do terreno é o tamanho dele com relação ao eixo X do espaço, enquanto que a profundidade é o tamanho do mesmo com relação ao eixo Z.

Esta função será invocada cada vez que o jogador passar por um ponto no terreno e garante que a mesma aparência observada anteriormente esteja presente neste ponto quando ele passar por ele novamente sem que haja um armazenamento prévio.

3.2.3 Algoritmo Gerador do terreno

O terreno do jogo desenvolvido é constituído de várias cópias de um objeto pré-fabricado do tipo plano, criadas e removidas em tempo de execução.

O Script *TerrainGenerator* é o responsável por gerar o terreno em tempo de execução. Sua função é a manipulação da malha. Foi originalmente desenvolvido com base no algoritmo *Simplex Noise*[5].

A seguir são descritas as variáveis do algoritmo e suas respectivas funções.

private Perlin noise = new Perlin();

Instância do objeto da classe *Perlin* que contém o método de ruído.

public Transform target;

O alvo a partir do qual serão adicionados novos pedaços de terreno. O *Transform* pode ser qualquer componente da cena do jogo. Desta forma podemos

---

[4] Pode ser obtido através do link http://unity3d.com/support/resources/example-projects/procedural-examples.
[5] Página do desenvolvedor: http://www.quickfingers.net/.

obter suas posições na cena. Para que o terreno seja adicionado e removido conforme o movimento da câmera principal do jogo, ela será o alvo. Através do *Transform* podemos capturar suas coordenadas atuais de tempo em tempo.

Para acompanhar a posição da câmera na cena, deve-se seguir o exemplo abaixo em *JavaScript*:

```
var target: Transform;
function Update () {
    guiText.text = "x= " + target.position.x + " y= " + target.position.y + " z= " +
    target.position.z;
}
```

Tudo o que se encontra na função *Update*() é executado a cada *frame*. Conforme a mudança de posição do *target*, um componente *GUI Text* é atualizado com essas informações. Deve-se adicionar um componente deste tipo na tela com o *Script* anexo a ele.

```
public Object terrainPlane;
```

O *terrainPlane* é o objeto pré-fabricado do tipo plano do motor Unity, a malha que será replicada várias vezes na cena do jogo conforme a câmera se movimenta constituindo o terreno infinito.

```
public int buffer;
```

Delimita a quantidade de objetos planos adicionados na cena de cada vez.

```
public float detailScale;
```

Define o tamanho do ruído. É a distância entre as áreas integrais. Valores maiores que 0.1 criam um ruído com aspecto pontiagudo. Valores menores que 0.1 dão um aspecto ondulado semelhante a colinas.

Quando seu valor é menor o resultado obtido são menos montanhas com altura menor. Quando seu valor é alto são criadas várias montanhas de altura maior.

public float heightScale;

Controla a aparência do ruído, já que a função de ruído sempre retorna valores entre -1.0 e 1.0, para que a mesma seja coerente e realista. Se a variável *heightScale* assume o valor 0 o terreno terá aparência plana como na Figura 8. Se assumir o valor 40 o terreno terá pequenas montanhas como na Figura 9.

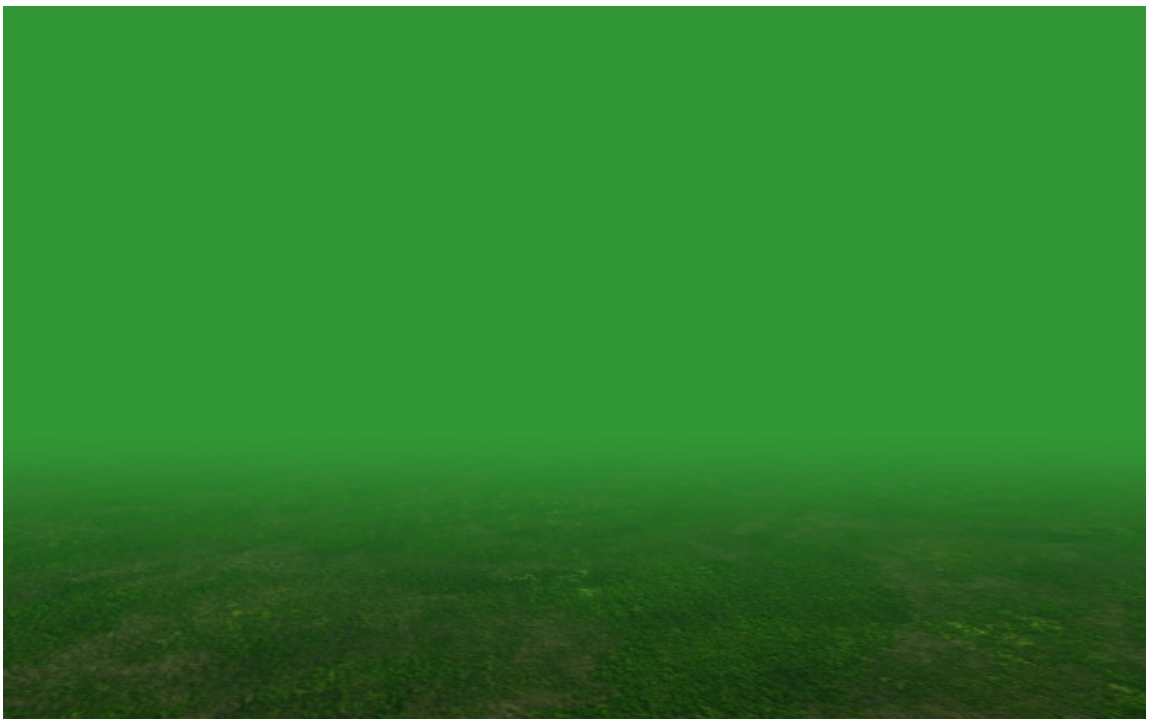

Figura 8 – Terreno plano gerado pelo algoritmo
Fonte: Autoria Própria (2012)

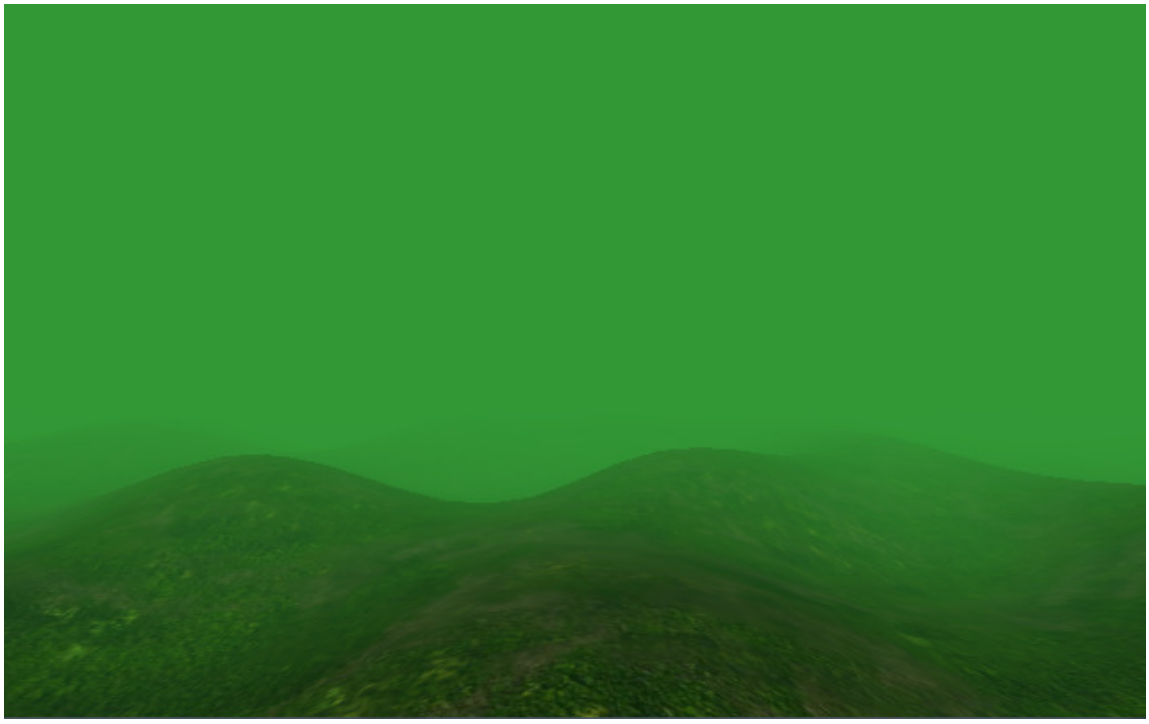

Figura 9 – Terreno montanhoso gerado pelo algoritmo
Fonte: Autoria Própria (2012)

public float planeSize = 60f;

As dimensões dos objetos planos.

private int planeCount;

Delimita a quantidade de planos para cada linha.

private int tileX; private int tileZ;

Armazenam a posição atual do objeto plano.

private Tile[,] terrainTiles;

Vetor de planos atualmente na cena.

Na função *Start* é atribuido o valor da quantidade de planos e das posições X e Z partindo da posição da câmera dividida pelo tamanho do plano.

```
void Start() {
        planeCount = buffer * 2 + 1;
        tileX = Mathf.RoundToInt(target.position.x / planeSize);
        tileZ = Mathf.RoundToInt(target.position.z / planeSize);
        Generate();
}
```

Para a remoção dos planos, é verificada a existência de objetos no vetor que contém os planos da cena, então é percorrido sendo utilizada a função *Destroy* que remove estes objetos e todos os seus componentes.

```
if (terrainTiles != null) {
         foreach (Tile t in terrainTiles){
                Destroy(t.gameObject);
         }
}
```

Para a adição de planos, é armazenado no vetor de planos cada objeto gerado pela função *GenerateTile*.

```
for (int x = 0; x < planeCount; x++) {
         for (int z = 0; z < planeCount; z++) {
                terrainTiles[x, z] = GenerateTile(tileX - buffer + x, tileZ - buffer + z);
         }
}
```

O trecho de código abaixo demonstra a chamada da função de ruído Perlin para cada vértice da malha, ou seja, cada ponto comum entre os lados da malha, gerando sua altura.

```
for (int v = 0; v < vertices.Length; v++) {
        Vector3 vertexPosition = plane.transform.position + vertices[v] * planeSize / 10f;
        float height = noise.Noise(vertexPosition.x * detailScale, vertexPosition.z *
        detailScale);
        vertices[v].y = height * 60;
}
```

Como especificado na seção 3.3.1 o objeto plano possui um componente chamado *MeshFilter* o que o torna uma malha. Sendo assim, os vértices da malha são obtidos com a função de acesso *GetComponent* que permite acessar os componentes de um objeto:

```
Mesh mesh = plane.GetComponent<MeshFilter>().mesh;
Vector3[] vertices = mesh.vertices;
```

Os vértices são identificados por uma posição tridimensional, por isso é utilizado o *Vector3[ ]* para armazená-los.

Em uma malha de triângulos, cada triângulo possui três vértices. Supondo que as dimensões da malha sejam 1x1, tem-se:

vertices[0]: canto superior esquerdo do triângulo;

vertices[1]: canto superior direito;

vertices[2]: canto inferior esquerdo.

4 RESULTADOS

O trabalho resultou em duas implementações do algoritmo Perlin Noise. A primeira na linguagem C com o objetivo de analisar o comportamento da função de ruído a partir dos valores de saída. A segunda no desenvolvimento de um cenário 3D utilizando alguns recursos do motor de jogo Unity para a sintetização de um terreno infinito.

Para analisar a eficiência da ferramenta proposta foi realizada a avaliação da mesma em dois ambientes operacionais diferentes. Foi levada em consideração a análise do FPS resultante do processamento da aplicação em cada um dos computadores que estão apresentadas na Tabela 2.

Tabela 2 – Comparação de ambientes operacionais

Computador 01	**Computador 02**
Processador: Core 2 Duo	**Processador:** Core 2 Duo
1.66 GHz 1.67 GHz	2.20 GHz 2.20 GHz
Memória: 2,00 GB	**Memória:** 3,00 GB
FPS: 381.35	**FPS:** 377.22

Fonte: Autoria Própria (2012)

Percebe-se que não houve nenhuma diferença significativa na execução dos cenários, desta forma entendemos que a ferramenta poderá ser aplicada em diversos ambientes operacionais sem que haja piora no desempenho geral.

Em comparação com o motor de jogo GameMaker (YOYO GAMES, 2007) que trabalha com codificação de Scripts semelhante ao Unity observou-se que o seu objetivo não é a criação de mundos 3D. Embora possua funcionalidades limitadas para gráficos 3D, seu foco é no desenvolvimento de jogos 2D.

O GameMaker trabalha com uma linguagem próprio chamada *GML* (*The Game Maker Language*) já o Unity trabalha com a linguagem C# que torna o desenvolvimento menos complexo por ser uma linguagem de alto nível.

5 CONCLUSÕES

Perlin Noise se trata de um algoritmo simples e útil para a produção de vários efeitos que imitam fenômenos naturais e se mostrou eficiente em atender o objetivo proposto com o desenvolvimento deste trabalho.

O terreno de um jogo pode ser criado utilizando técnicas não-procedurais: Por meio de ferramentas de modelagem em 3D, e técnicas procedurais como as abordadas aqui. A utilização desta técnica tem como principais vantagens a economia de espaço em disco, visto que a criação do cenário do jogo pode ser gerada inteiramente em tempo de execução e economia de memória, não sendo necessário o armazenamento de valores em estruturas de dados. Um ponto negativo são as inúmeras chamadas à função para cada ponto do terreno enquanto este é percorrido. Para atingir efeitos mais sofisticados maior é a complexidade do cálculo da função e isso pode consumir muito processamento.

Foi criado um terreno infinito combinando funções específicas do motor Unity para a replicação de objetos na cena conforme o movimento da câmera com a função de ruído da classe Perlin para modificar os aspectos de altitude e escala dos objetos.

O trabalho foi enriquecedor visto que as áreas de estudo não são exploradas nas disciplinas curriculares do curso de Tecnologia em Análise e Desenvolvimento de Sistemas e técnicas procedurais é uma área de pesquisa ativa em computação gráfica.

Como trabalho futuro, fica a sugestão de uma exploração mais profunda de outros efeitos que podem ser sintetizados com a função de ruído, na criação de um jogo com aparência inteiramente procedural, onde o tempo real do usuário seja empregado dentro do jogo e que alguns fenômenos naturais dos dois mundos estivessem em sincronia: O dia, noite, pôr do sol e outros.

6 REFERÊNCIAS

Unity Technologies. Unity, versão 3.4.2f3. Unity Technologies, 2012. Disponível em: <http://unity3d.com/unity/download/>. Acesso em: 14 de novembro de 2012.

FAXIN, Yu; ZHEMING, Lu. **Three-Dimensional Model Analysis and Processing.** Advanced Topics in Science and Technology in China, 2010.

FLETCHER, Dunn; PARBERRY, Ian. **3D Math Primer for Graphics and Game Development.** 2002.

EBERT, David S. **Texturing & Modeling:** A Procedural Approach. 3rd Ed. 2003.

BEVILACQUA, Fernando. **Ferramenta para geração em tempo real de mundos virtuais pseudo-infinitos para jogos 3D.** 2009.

VAUGHAN, William. **Digital Modeling.** 2011.

KOZOVITS, Lauro E. **Um estudo para visualização de objetos com geometria dinâmica em jogos multi-jogador.** PUC-RioInf.MCC34/03, 2003.

PERLIN, Ken. **An Image Synthesizer. In Computer Graphics.** Courant Institute of Mathematical Sciences New York University, 1985.

CLAUDINO, André. **Funções para interpolação de pontos aplicado à Modelagem Geométrica**. 2007. Disponível em: <http://lvelho.impa.br/i3d07/demos/claudino/>. Acesso em: 17 de maio de 2012.

FOLEY, James D. **Computer Graphics:** Principles and Practice. 2nd Ed. 1996.

ROGERS, David F. **Procedural Elements for Computer Graphics.** 1998.

SHIFFMAN, Daniel. **Learning Processing:** A Beginner's Guide to Programming Images, Animation, and Interaction. 2008.

EBERLY, David H. **3D Game Engine Design:** A Practical Approach to Real-Time Computer Graphics.

GOLDSTONE, Will. **Unity Game Development Essentials.** Published by Packt Publishing Ltd.**,** 2009.

CREIGHTON, Ryan H. **Unity 3D Game Development by Example: Beginner's Guide.** Published by Packt Publishing Ltd., 2010.

YoYo Games. GameMaker. YoYo Games, 2007. Disponível em: <http://www.yoyogames.com/gamemaker/studio/>. Acesso em: 18 de novembro de 2012.