Geração Procedural de Terrenos em GPU

Felipe Gomes Sampaio
Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação
Orientadora: Jesuliana Nascimento Ulysses

Agenda

- Introdução
- GPU
- Terrenos Procedurais
- Implementação
- Resultados
- Conclusão

Introdução

- Geração Procedural
  - Modelos gerados a partir de funções ou algoritmos
- Modelar a natureza
- Terrenos
  - Relevo (montanhas, rios)
  - Materiais (rocha, água, areia)
- GPU

GPU

- Por quê?
  - Crescimento do poder de processamento
  - O *hardware* é relativamente barato
  - Como?

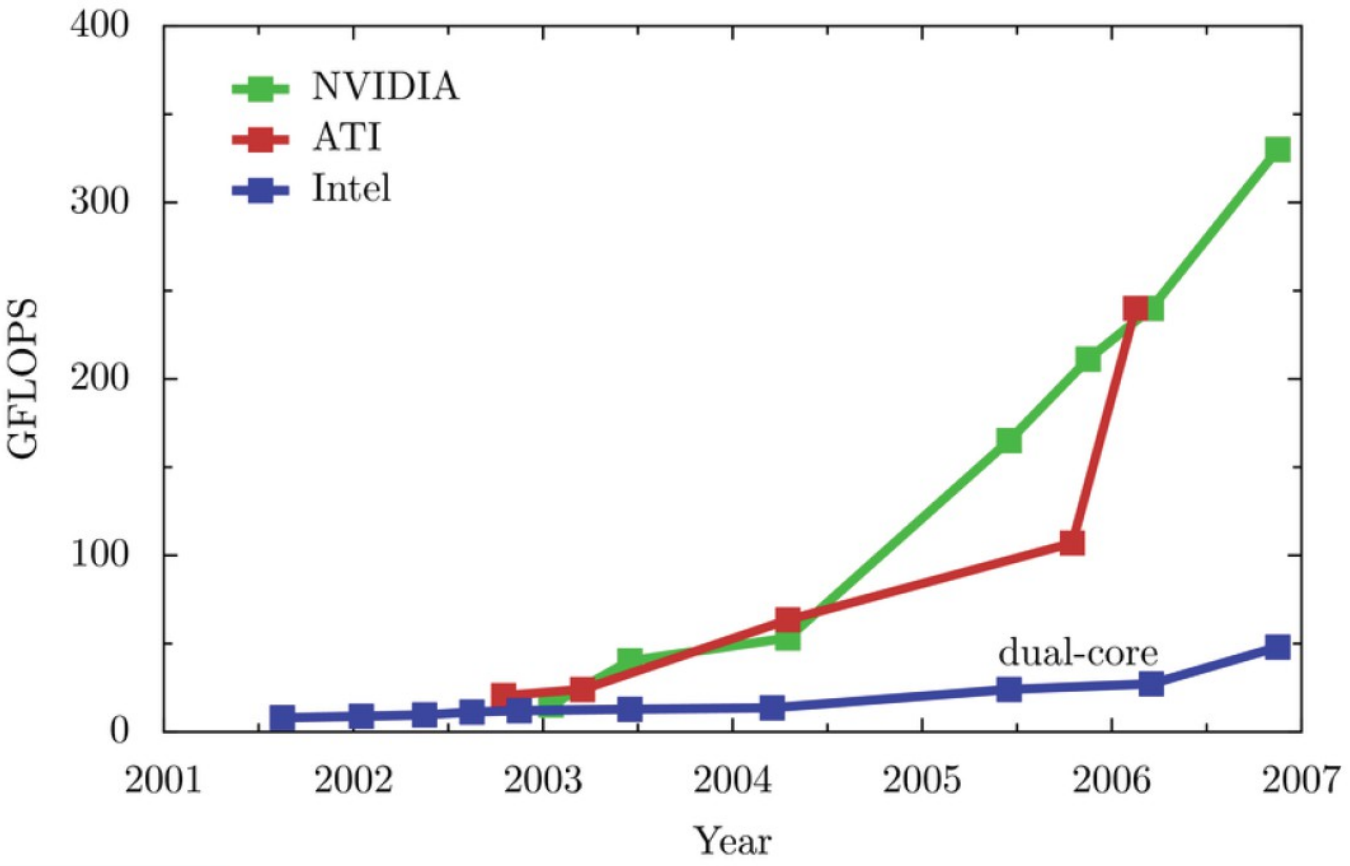
GPU

- JOGOS !
- Industria exige cada vez mais poder de processamento
- Gera bilhões de dólares por ano
- Incentiva cada vez mais o desenvolvimento do *hardware*
- E acaba por baixar o custo

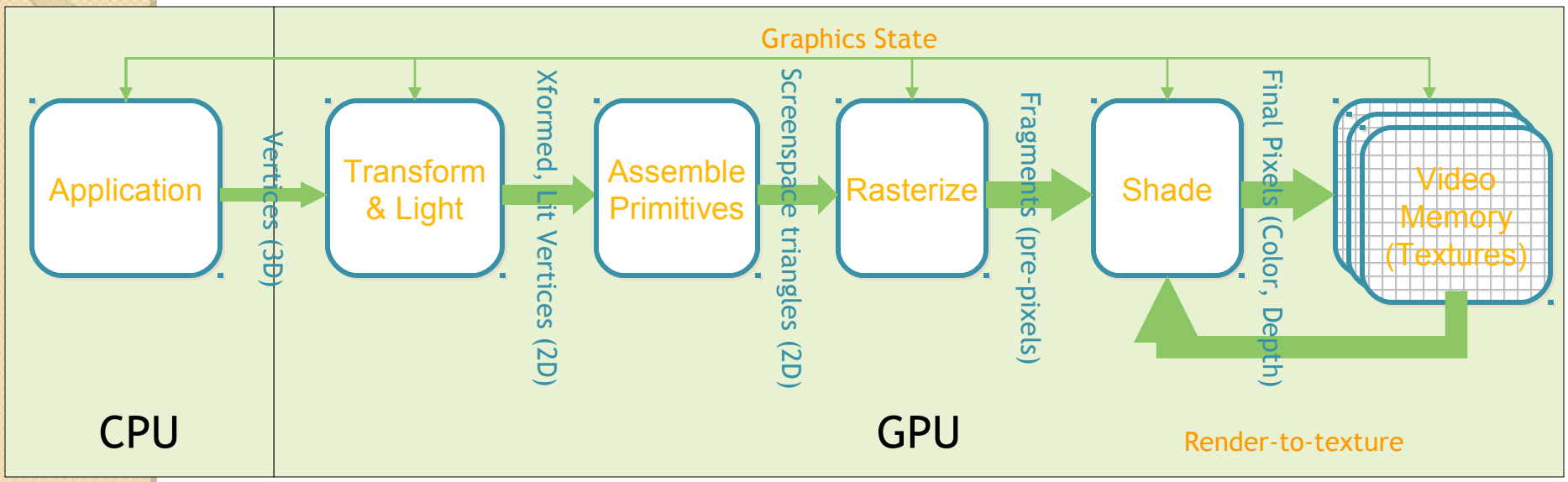
GPU

- GPUs são rápidas...
  - High-end quad core CPU
    - Unidades de processamento: 16
    - Poder: 96 GFLOPS
  - ATI Radeon HD 4870:
    - Unidades de processamento: 800
    - Poder: 1200 GFLOPS

GPU

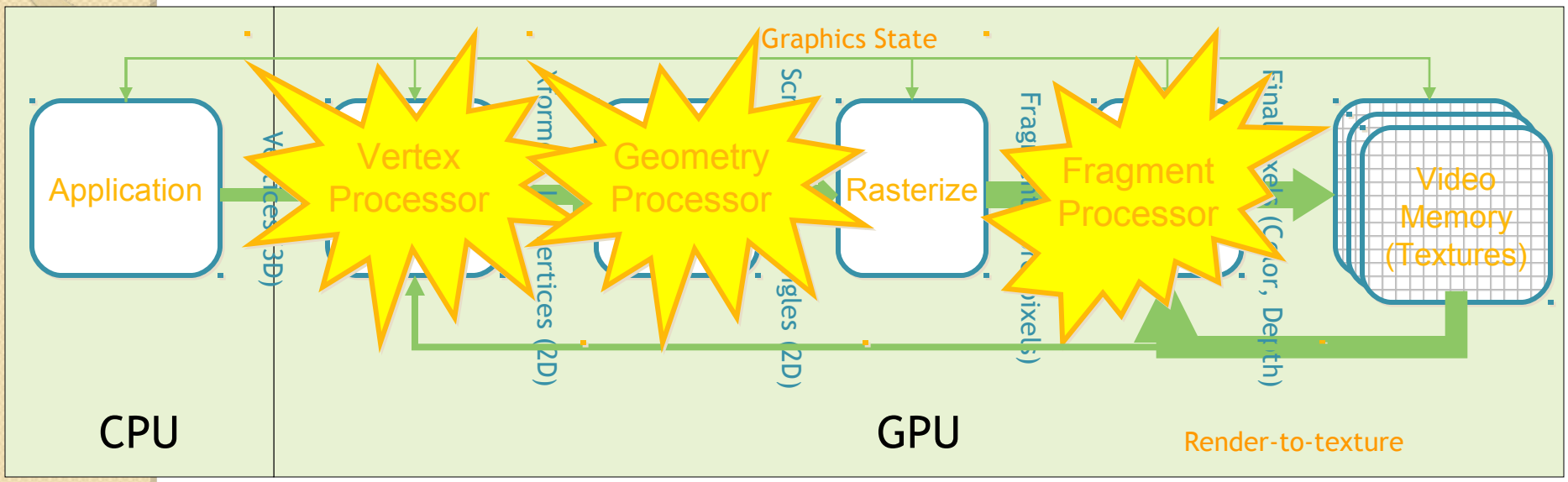

400
300
200
100
0
GFLOPS
NVIDIA
ATI
Intel
dual-core
2001
2002
2003
2004
2005
2006
2007
Year

GPU



- Versão simplificada da pipeline

GPU



- Processador de Vértices Programável
- Processador de Geometria Programável
- Processador de Fragmentos Programável

GPU

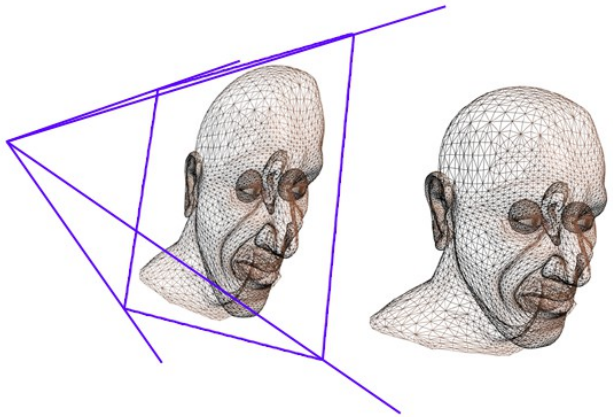
- Shaders
  - Shaders são programas que executam em determinadas etapas da pipeline
  - Não são aplicações *stand-alone*
  - Necessitam de uma aplicação que utilize um API (OpenGL ou Direct3D)

GPU

- Shaders
  - Vertex shader – Vertex Processor
  - Fragment shader – Fragment Processor
  - Geometry shader – Geometry Processor

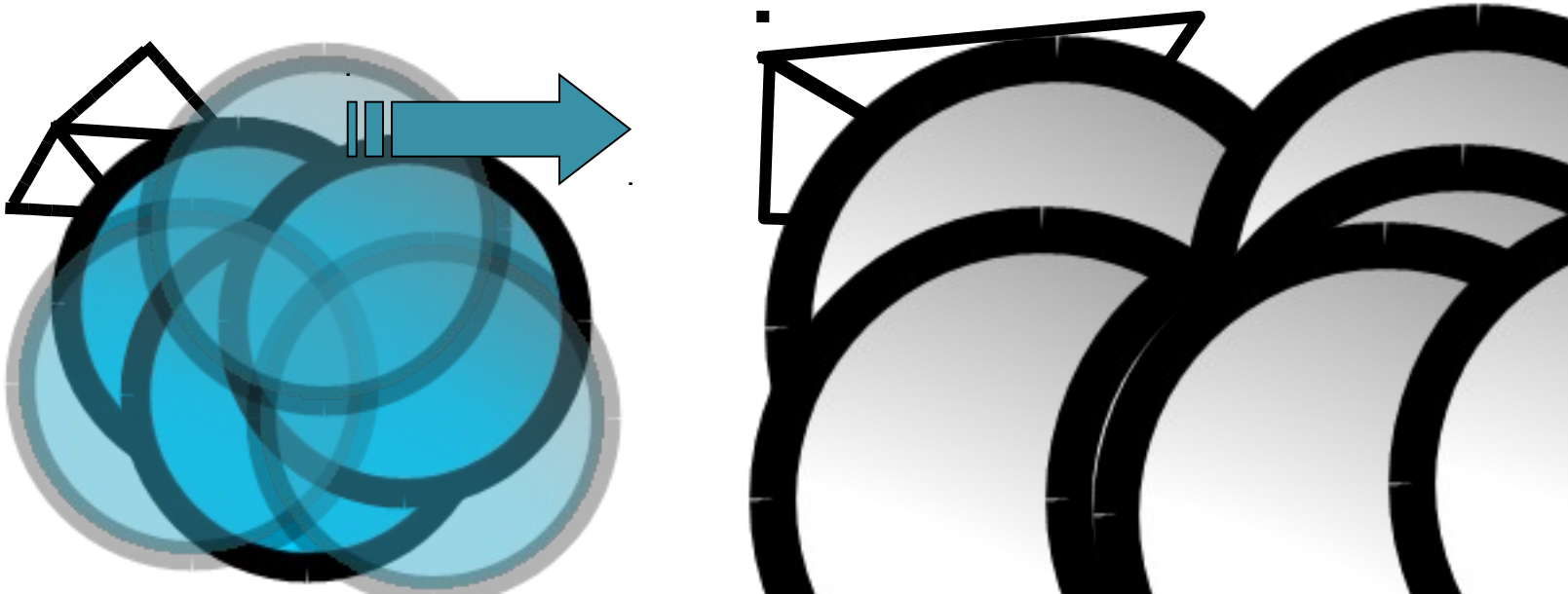
GPU

- Vertex Processor
  - Transforma do espaço de mundo para o espaço de tela
  - Calcula iluminação per-vertex



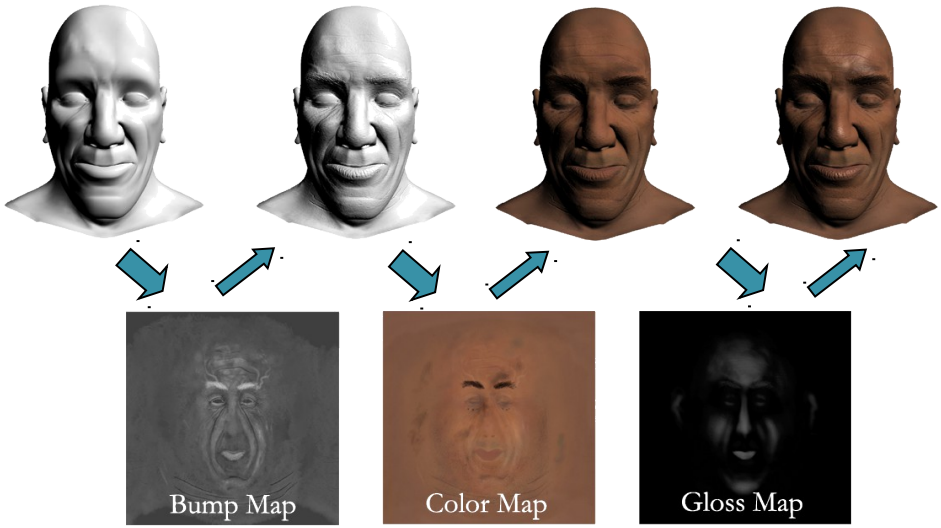
GPU

- Geometry Processor
  - Como os vértices se conectam para formar a geometria
  - Operações por primitiva



GPU

- Fragment Processor
  - Calcula a cor de cada pixel
  - Obtém cores de texturas



GPU

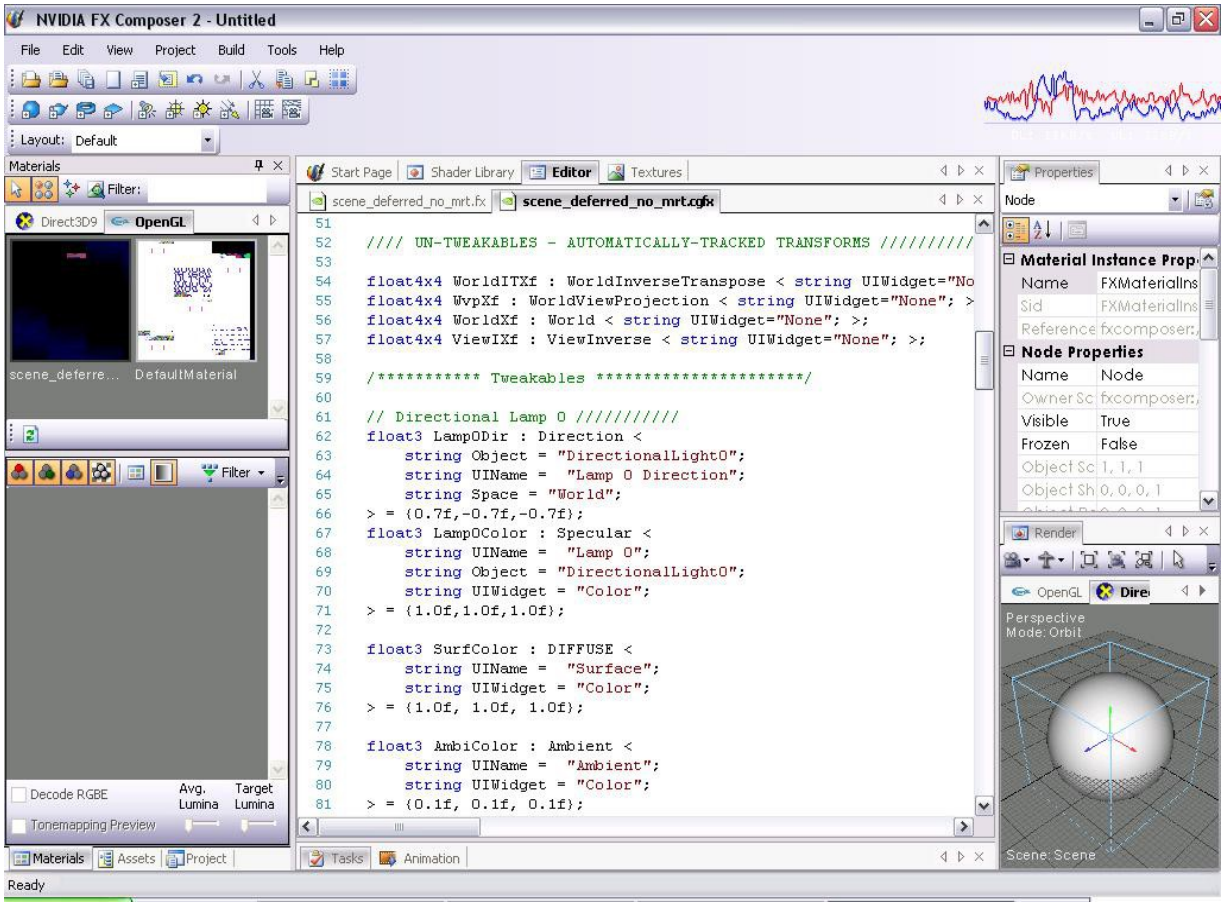
- Hoje temos que a programação de shaders é feita em linguagens de alto nível
- No estilo de c/c++
- As principais que temos hoje:
  - GLSL – OpenGL Shader Language
  - HLSL – High Level Shader Language
  - Cg – C for Graphics

GPU

- Temos algumas ferramentas que servem para criação e edição de shaders:
  - NVIDIA FX Composer
  - ATI RenderMonkey

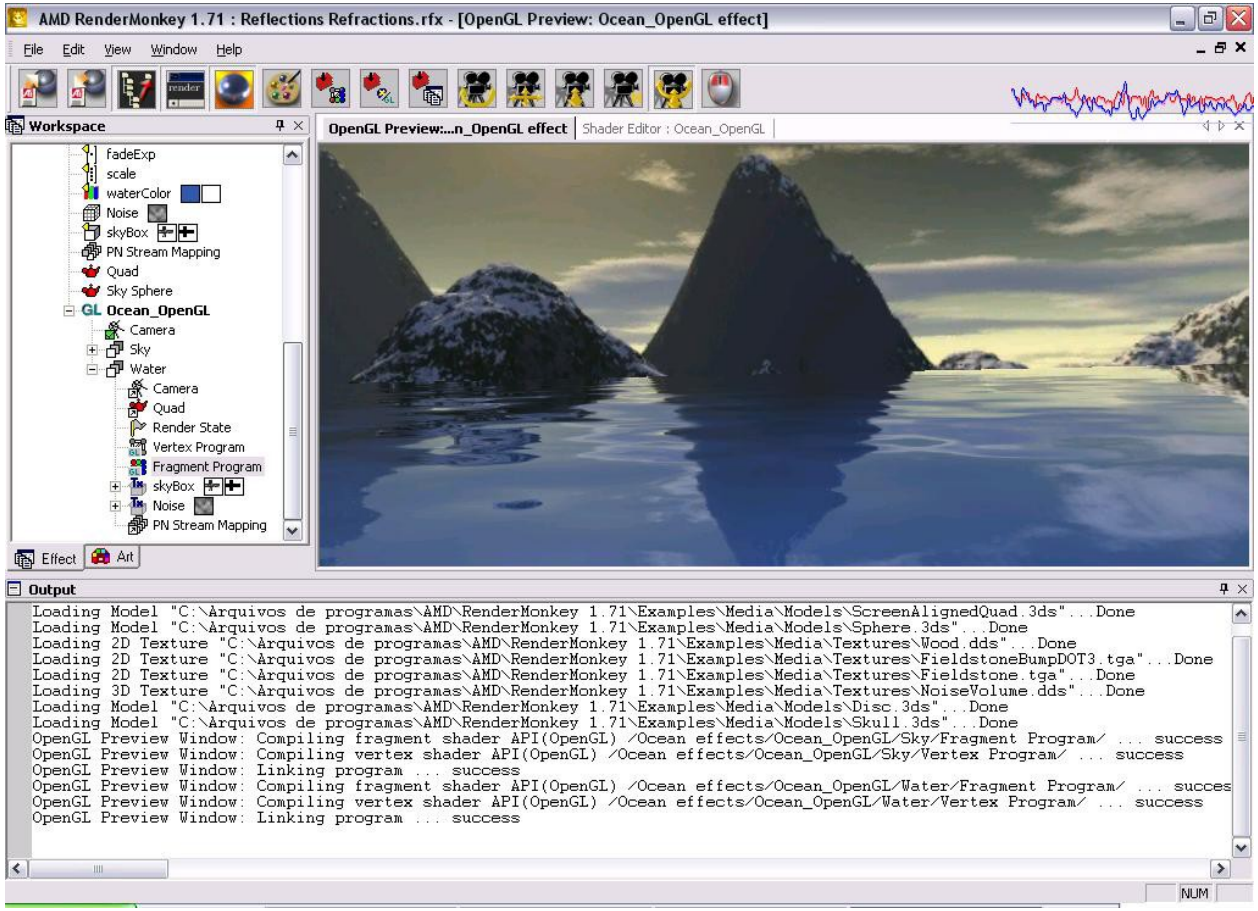
GPU

- NVIDIA FX Composer



GPU

- ATI RenderMonkey



Terrenos Procedurais

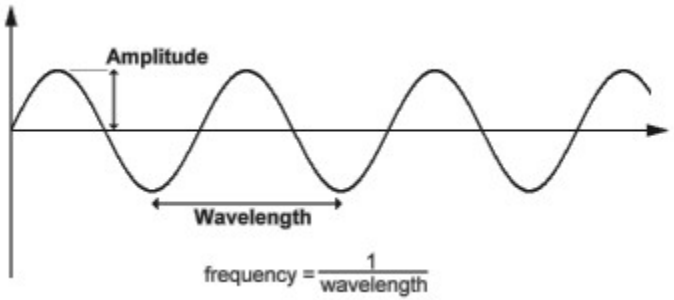
- Geração Procedural
  - Relevo (montanhas, rios)
  - Materiais (rocha, água, areia)
- Funções procedurais
  - Funções básicas
  - Funções fractais

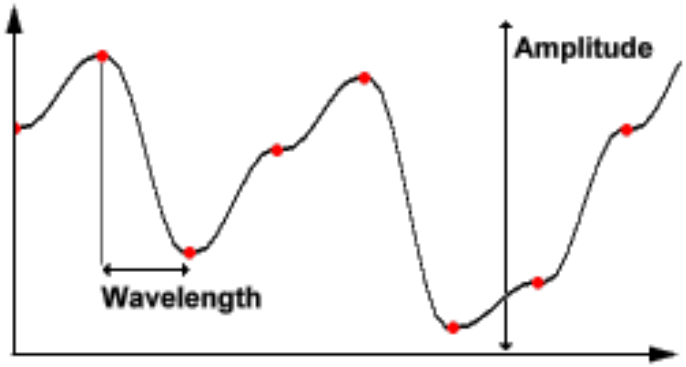
Terrenos Procedurais

- Funções básicas
  - Pseudo-aleatoriedade
  - A não ocorrência de periodicidade de padrões
  - Devem ser invariantes a rotação e translação

Terrenos Procedurais

- Função Noise
  - Criada por Ken Perlin em 1985
  - Como o nome diz, é uma função que produz ruído
  - 2 estágios





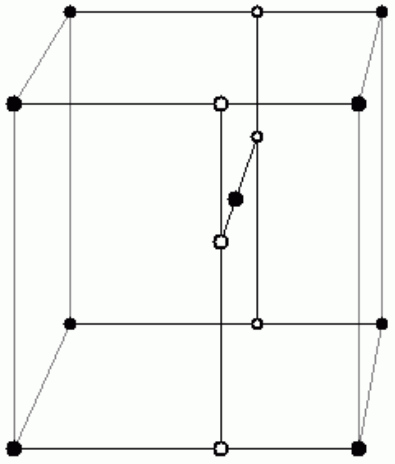
Terrenos Procedurais

- 1º estágio:
  - Gera um valor pseudo-aleatório para cada posição no espaço 3D
  - Utiliza coordenadas X, Y, Z para consultar uma tabela de permutação

```
static int permutation[] = { 151,160,137,91,90,15,
131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
};
```

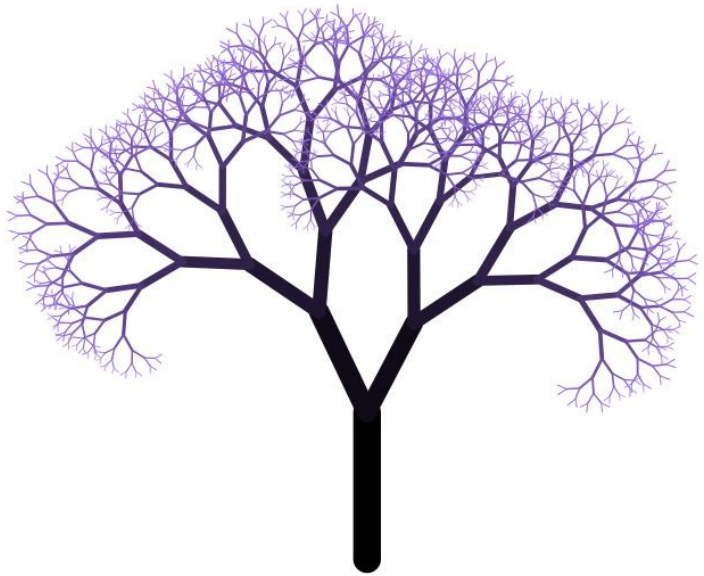
Terrenos Procedurais

- 2º estágio:
  - Valor obtido no 1º estágio usado para consultar uma tabela de gradientes
  - 8 consultas são realizadas e interpoladas para gerar o valor final da Noise



Terrenos Procedurais

- Fractais
  - “Objeto geometricamente complexo, complexidade esta que surge pela repetição de uma fórmula para variações de escala”, Kenton Musgrave



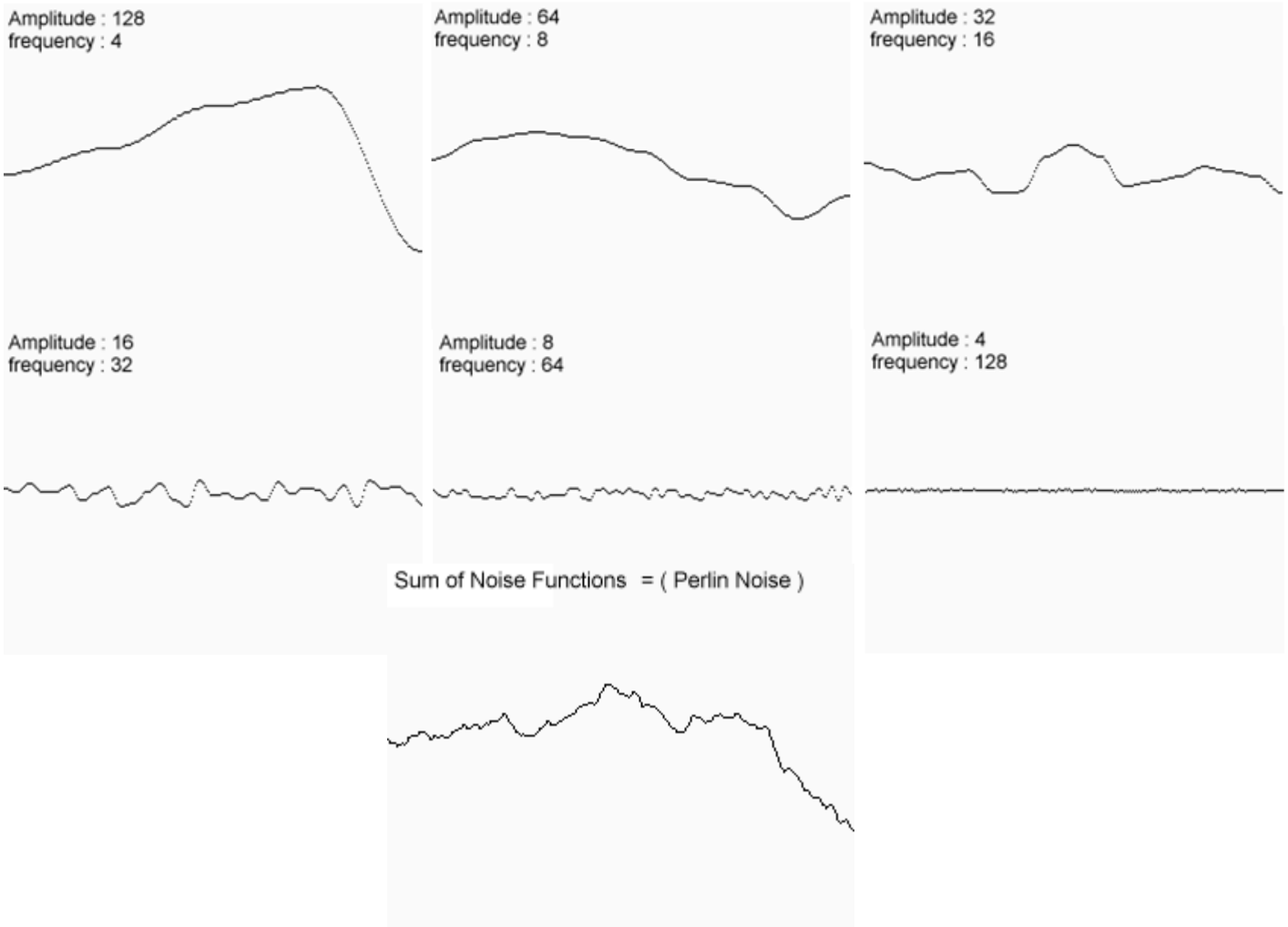
Terrenos Procedurais

- Turbulência
  - Criada por Ken Perlin em 1985
  - Utiliza a função noise em suas iterações
  - Simples implementação

```
Resultado = 0;
frequência = Freq_Inicial;
Enquanto (frequência < Freq_Máxima)
    Resultado += Absoluto ( noise (P * frequência) * Amplitude (frequência) );
    frequência *= 2;
```

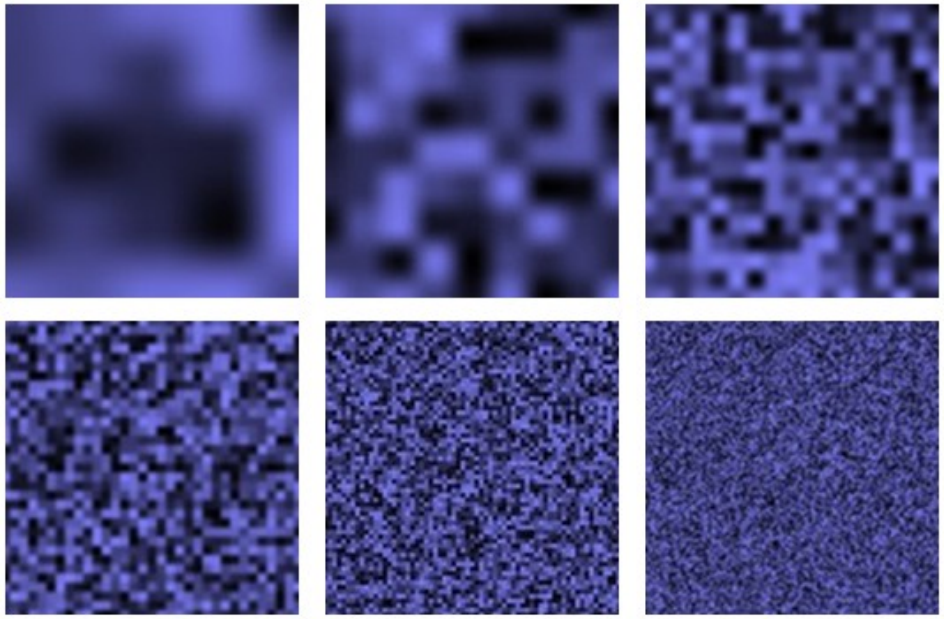
Terrenos Procedurais

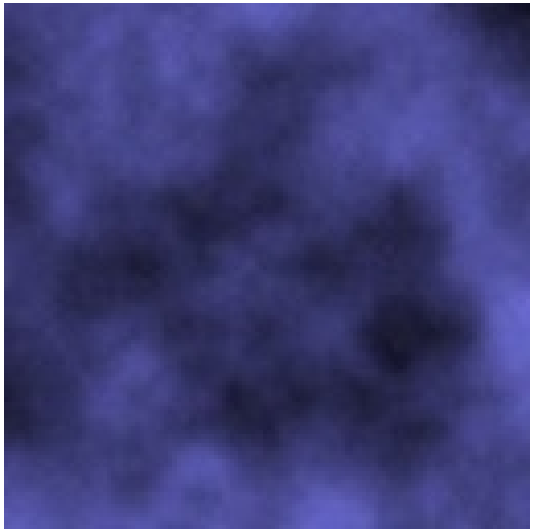
- Turbulência 1D



Terrenos Procedurais

- Turbulência 2D



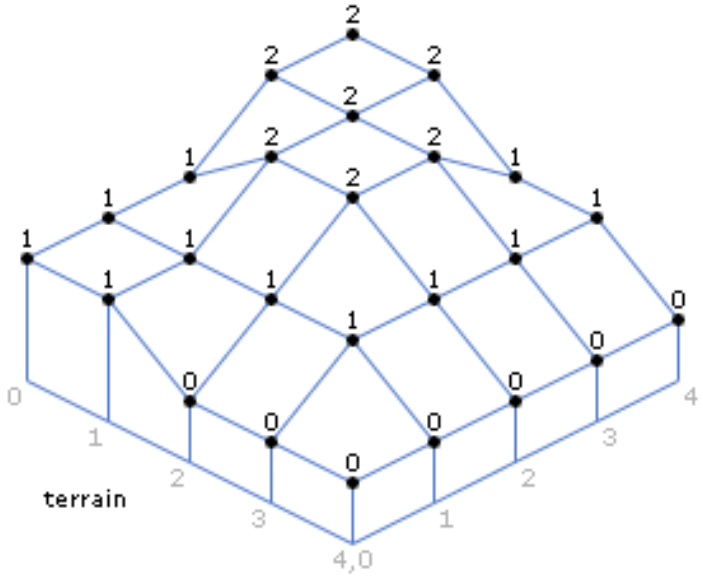


Terrenos Proceduráis

- Mapas de altura
  - Representação do terreno na forma de matriz

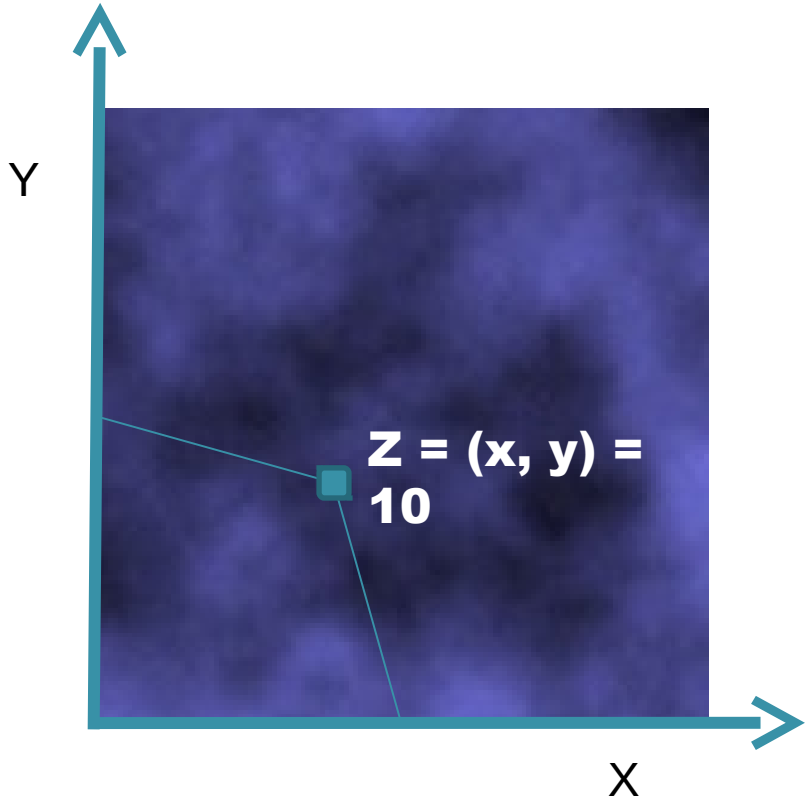
	0	1	2	3	4
0	1	1	1	2	2
1	1	1	2	2	2
2	0	1	2	2	1
3	0	1	1	1	1
4	0	0	0	0	0

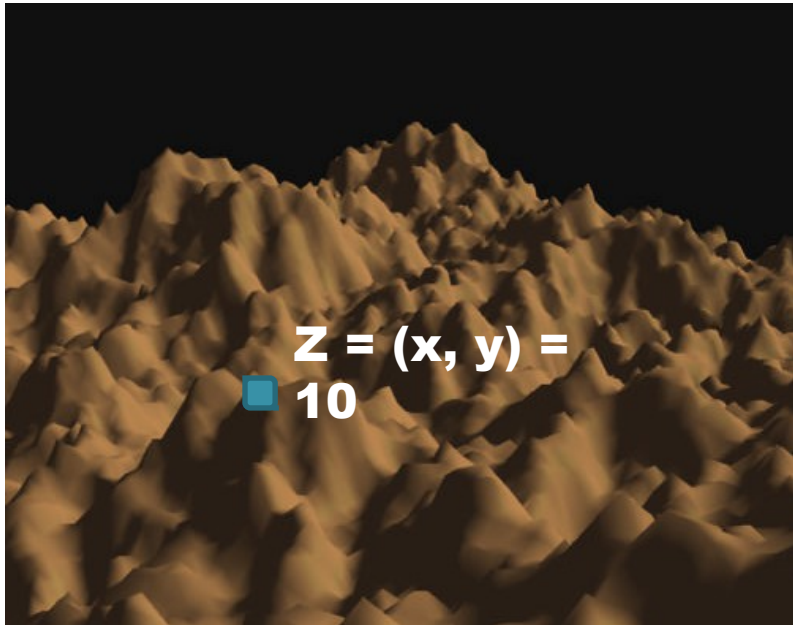
heightmap



Terrenos Procedurais

- Geração do terreno



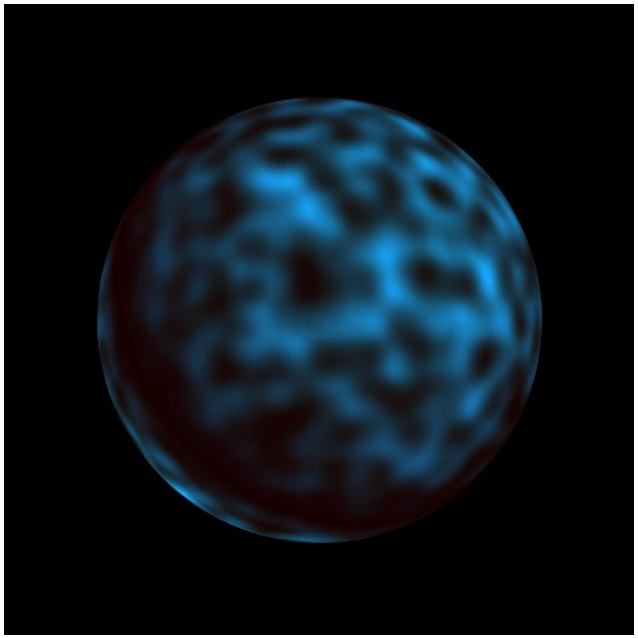


Terrenos Procedurais

- Texturas procedurais
  - Representam os materiais (rocha, areia, água)
  - Também podem ser geradas a partir da função noise/turbulência
  - Não apresentam problemas de *aliasing*
  - Não precisam ser armazenadas
  - Não precisam ser mapeadas sobre a superfície
  - Exemplos

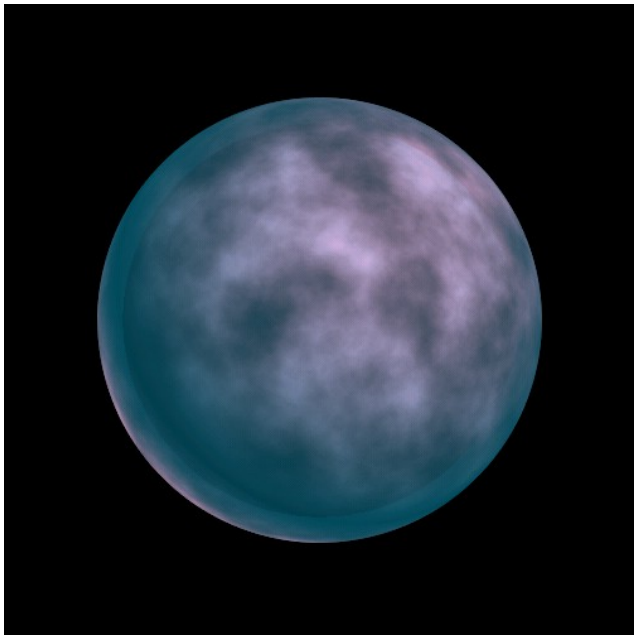
Terrenos Procedurais

- Exemplo 1
  - noise
  - Usada em conjunto com outras texturas para dar um aspecto de sujeira



Terrenos Procedurais

- Exemplo 2
  - noise(**p**) + ½ noise(2**p**) + ¼ noise(4**p**) ...
  - Geração procedural de nuvens

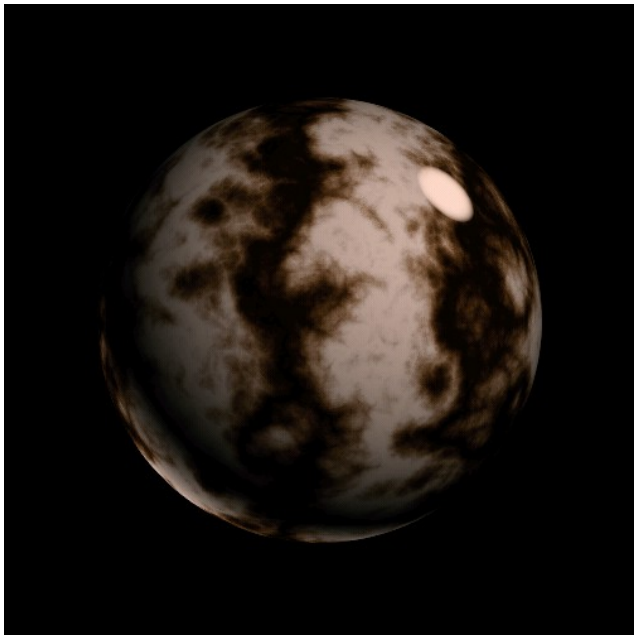


Terrenos Procedurais

- Exemplo 3
  - **|**noise(**p**)**|** + ½ **|**noise(2**p**)**|** + ¼ **|** noise(4**p**)**|** ...
  - Turbulência
  - Aparênci

Terrenos Procedurais

- Exemplo 4
  - sin(x + |noise(**p**)| + ½ |noise(2**p**)| + ...)
  - Textura de mármore



Implementação

- Terrenos gerados com Noise e Turbulência
- Utilização de OpenGL + GLSL
- Shaders implementados no RenderMonkey
- Toda a geração realizada no Pixel Shader
- Tabela de permutação e vetor de gradiente convertidos em texturas

Implementação

- Vetor de gradientes

```
static float3 g[] = {

  1,1,0,     -1,1,0,     1,-1,0,     -1,-1,0,

  1,0,1,     -1,0,1,     1,0,-1,     -1,0,-1,

  0,1,1,     0,-1,1,     0,1,-1,     0,-1,-1,

  1,1,0,     0,-1,1,     -1,1,0,     0,-1,-1,

};
```

Implementação

- Código em GLSL da função Noise

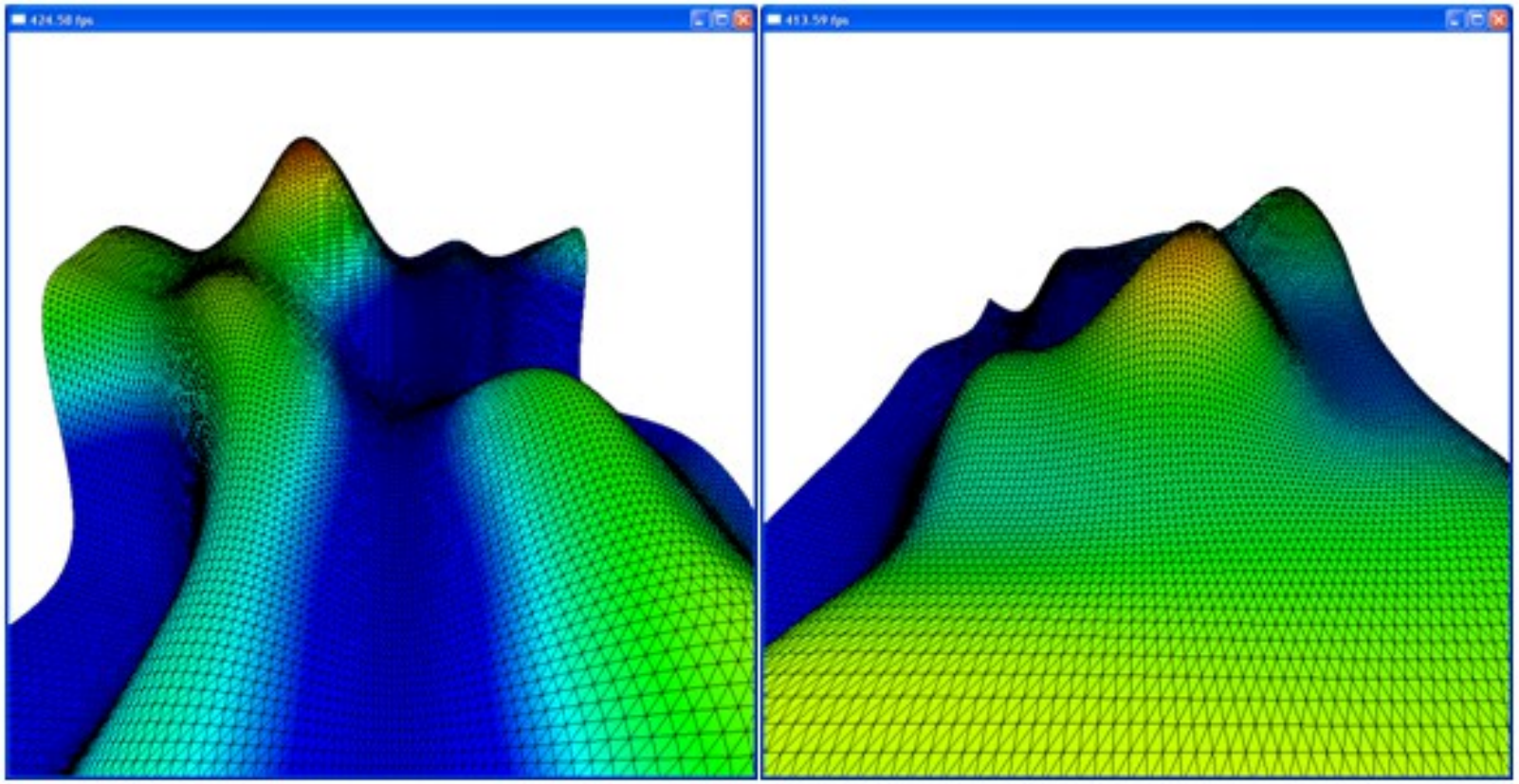
```
float inoise(vec3 p)
{
   vec3 P = mod(floor(p), 256.0);    // CUBO UNITARIO QUE CONTEM O PONTO
   p -= floor(p);                    // POSICAO X, Y, Z RELATIVA DO PONTO NO CUBO

   P = P / 256.0;                   // FAZ COM QUE P FIQUE ENTRE 0.0 E 1.0
   const float one = 1.0 / 256.0;   // PARA CONSULTA EM TEXTURA

    // CONSULTA A TABELA DE PERMUTACAO
   float A = perm(P.x) + P.y;
   vec4 AA;
   AA.x = perm(A) + P.z;
   AA.y = perm(A + one) + P.z;
   float B =  perm(P.x + one) + P.y;
   AA.z = perm(B) + P.z;
   AA.w = perm(B + one) + P.z;

   // INTERPOLA O RESULTADO DOS 8 CANTOS DO CUBO
   return
   mix(
   mix(
   mix(grad(perm(AA.x), p), grad(perm(AA.z), p + vec3(-1, 0, 0)), p.x),
   mix(grad(perm(AA.y), p + vec3(0, -1, 0)), grad(perm(AA.w), p + vec3(-1, -1, 0)), p.x),
   p.y),
   mix(
   mix(grad(perm(AA.x + one), p + vec3(0, 0, -1)), grad(perm(AA.z + one), p + vec3(-1, 0, -1)), p.x),
   mix(grad(perm(AA.y + one), p + vec3(0, -1, -1)), grad(perm(AA.w + one), p + vec3(-1, -1, -1)), p.x),
   p.y),
   p.z);
}
```

Implementação

- Frame Buffer Object (FBO)
  - Buffer para se realizar renderizações
  - Saída do Pixel Shader renderizada no FBO
  - Mapa de alturas
- Vertex Buffer Object (VBO)
  - Buffer utilizado pelo OpenGL como uma lista de vértices
  - Dados copiados do FBO
  - Pronto para renderização do terreno

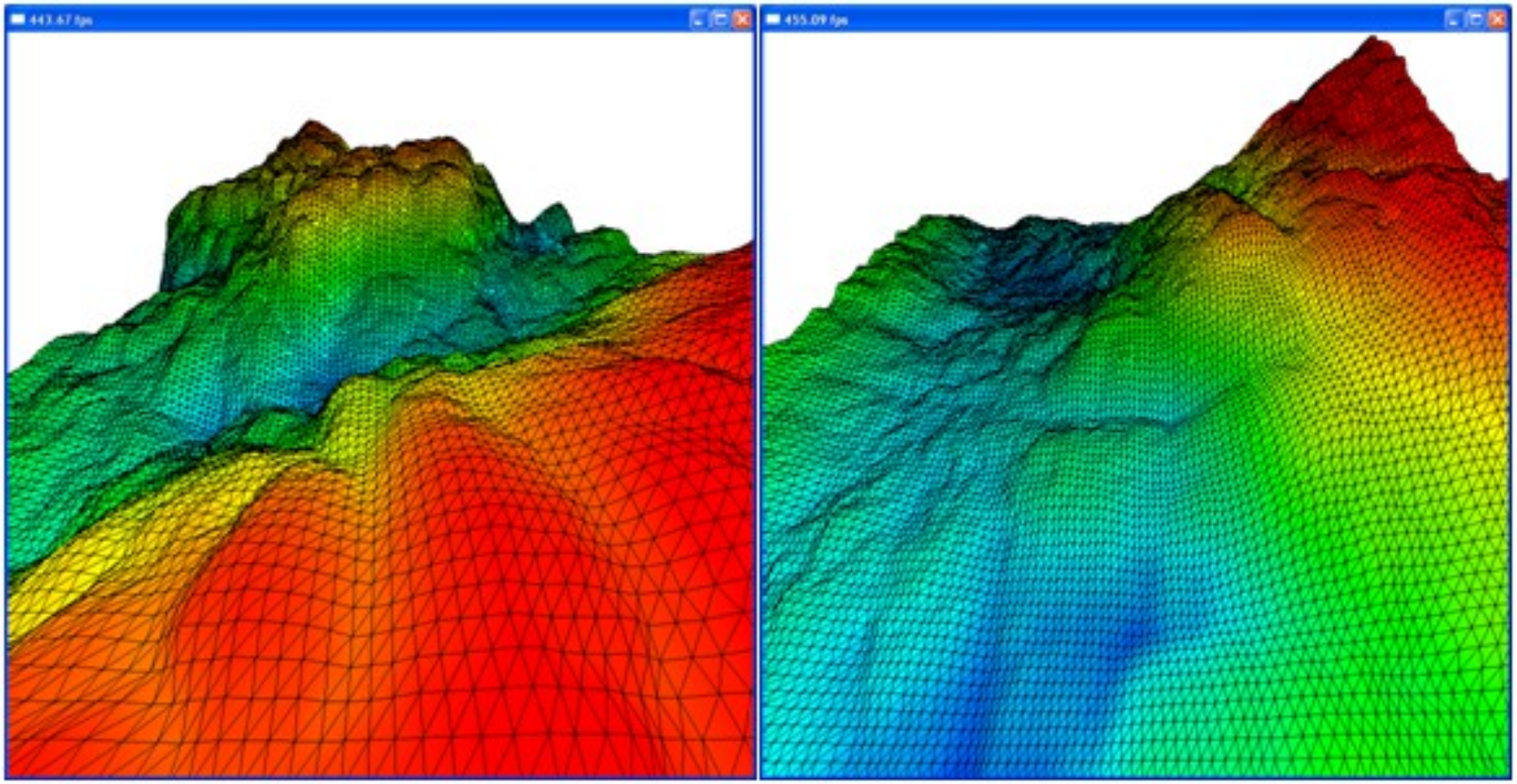
Resultados

- Perlin Noise



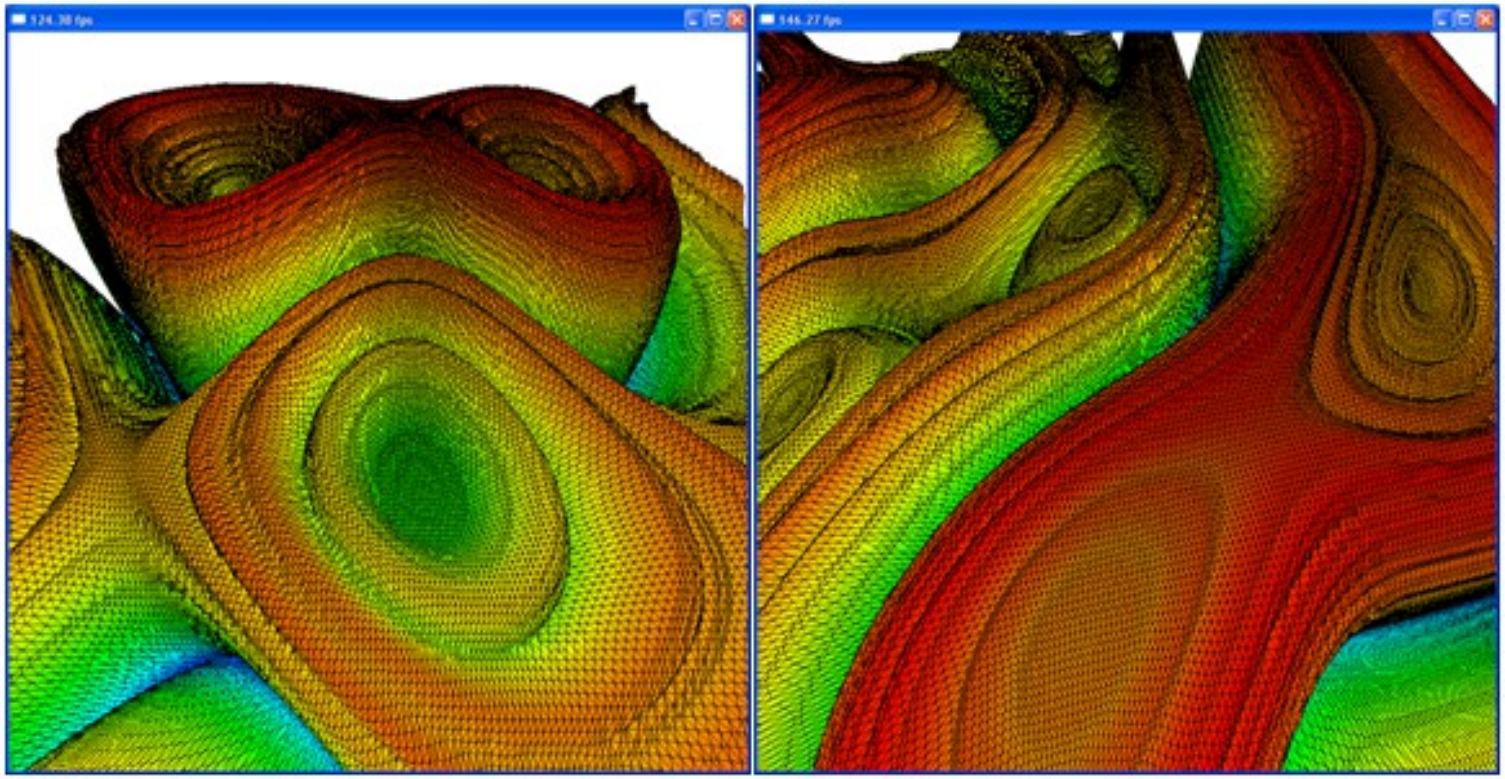
Resultados

- Turbulência



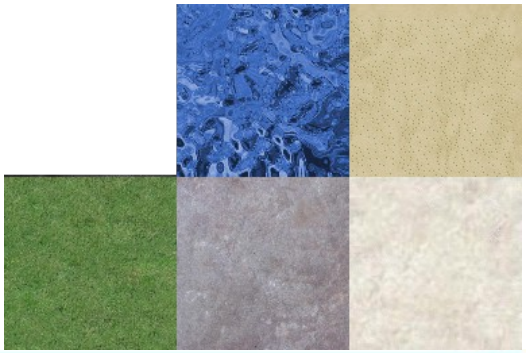
Resultados

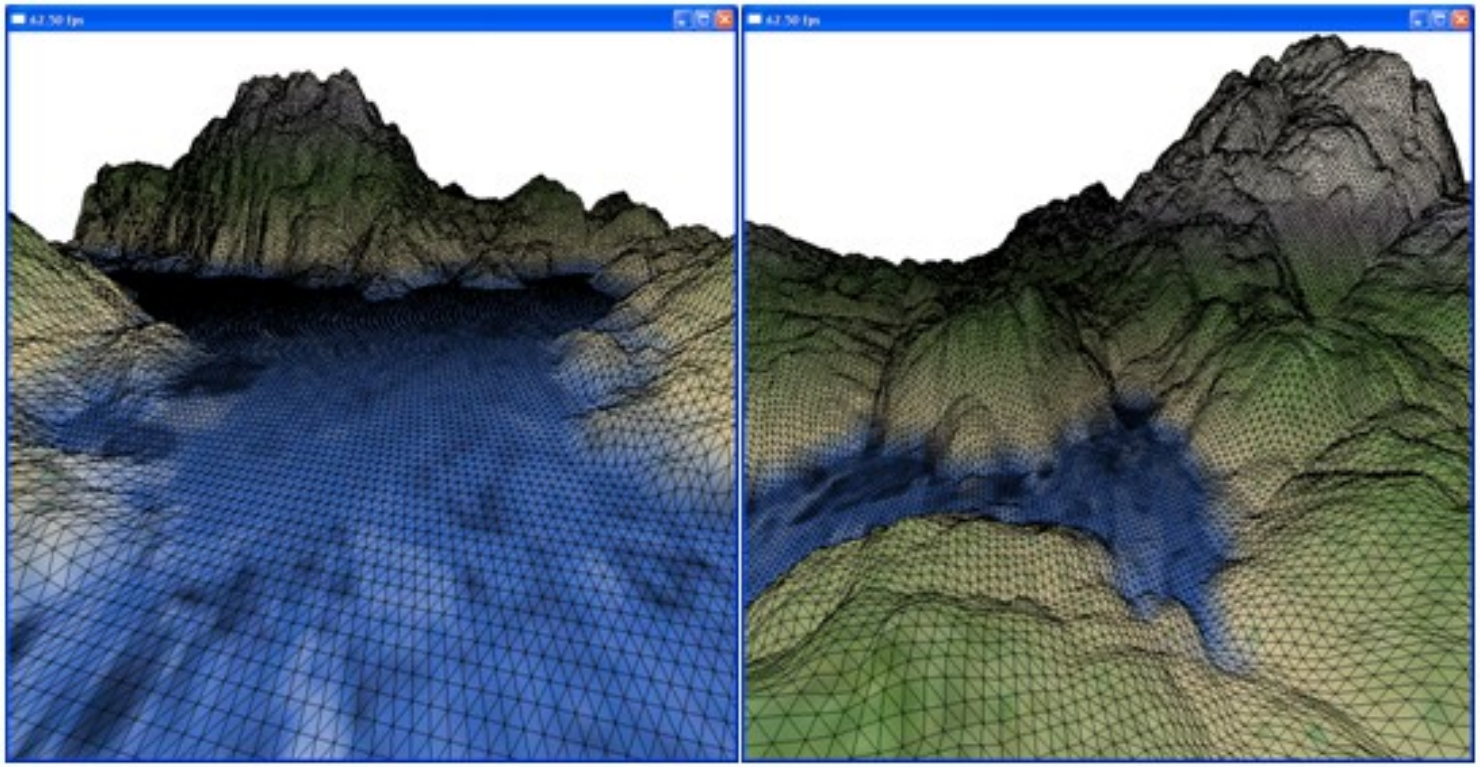
- Warping
  - turbulencia(noise(X, Y, Z))



Resultados

- Turbulência texturizada





Conclusão

- A Turbulência apresentou um bom resultado
- Paralelismo torna a GPU uma boa opção
- Programação em GPU é simples