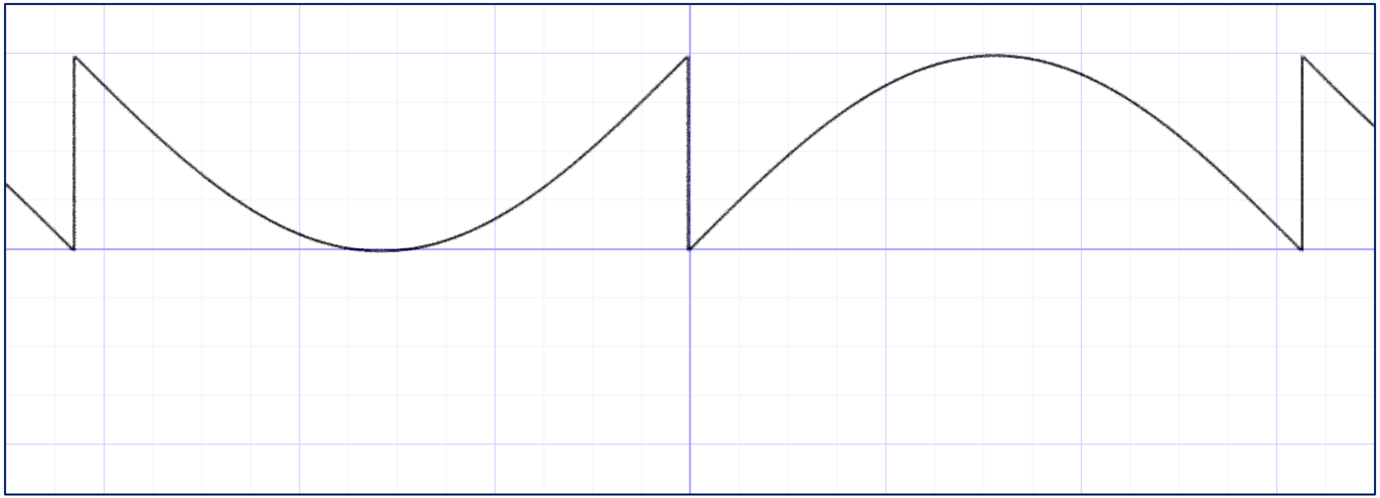
Computação Gráfica

Aleatoriedade e Ruídos

Aleatoriedade

Computadores convencionais não geram números realmente aleatórios, porém podemos criar situações que simulam isso.



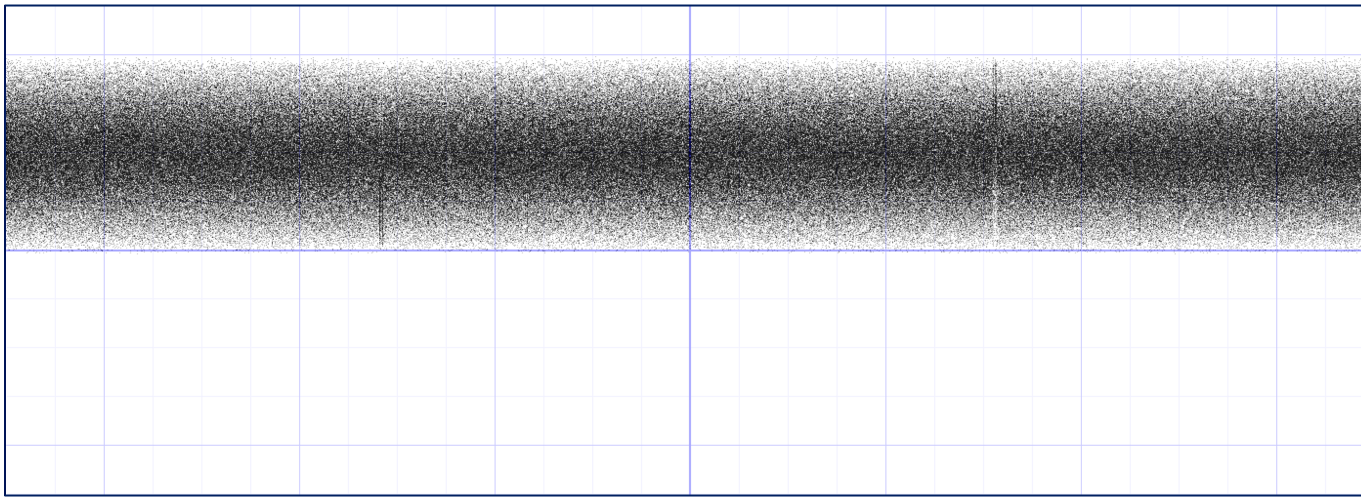
```
y = fract(sin(x)*1.0);
```


Referência: The Book of Shaders https://thebookofshaders.com/10/

Insper

Aleatoriedade

Porém se repetirmos muito o padrão. O que você acha que acontecerá com a imagem?



```
y = fract(sin(x)*100000.0);
```

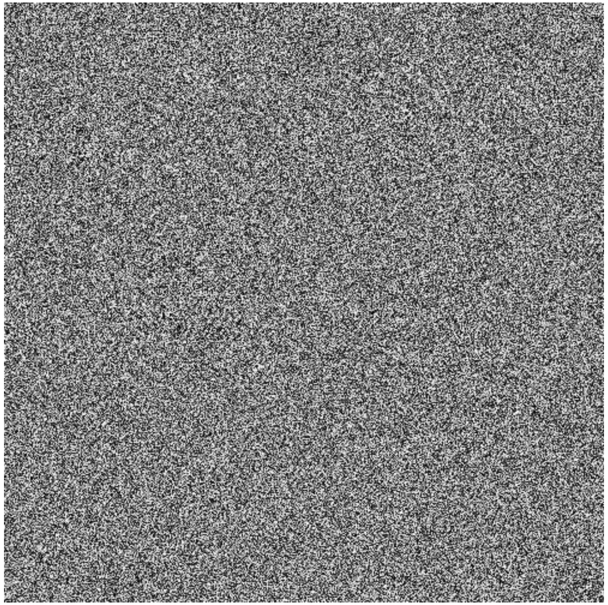
Referência: The Book of Shaders https://thebookofshaders.com/10/

Insper

Aleatoriedade

Brincando um pouco com o padrão conseguimos gerar algo como:

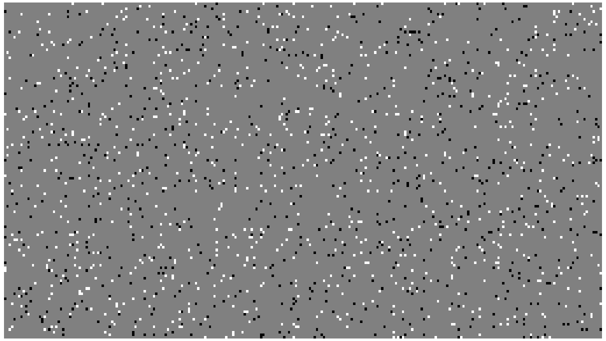
```
float rand(vec2 st) {
  float val = dot(st.xy,vec2(12.9898,78.233));
  return fract(sin(val)*43758.5453123);
}


void mainImage(out vec4 fragColor, in vec2 fragCoord){
  vec2 st = fragCoord.xy/iResolution.xy;
  float rnd = rand(st);
  fragColor = vec4(vec3(rnd),1.0);
}
```



Referência: The Book of Shaders https://thebookofshaders.com/10/

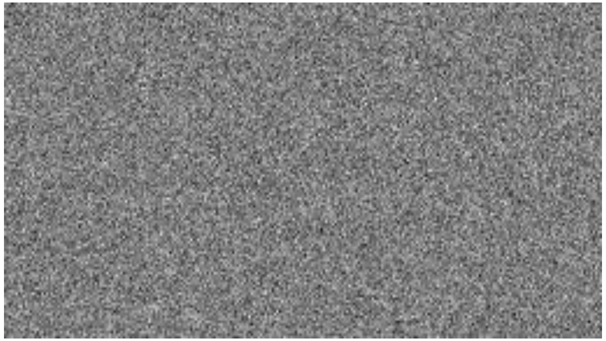
Ruídos

salt pepper noise

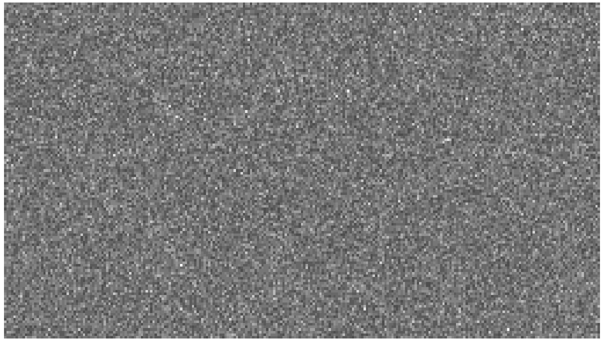


coherent noise

gaussian noise



Poisson noise ou shot noise

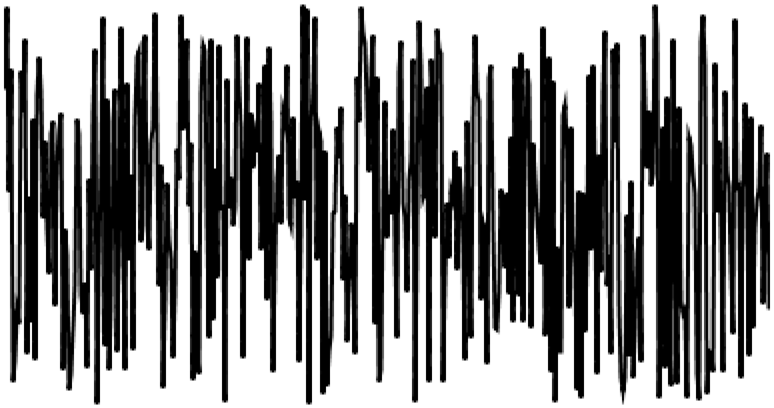


Quais as características desses ruídos?

Fonte: https://commons.wikimedia.org/wiki/User:Alejo2083/Pictures_engineering

Ruídos em 1D

Totalmente aleatório:



suavemente aleatório:

fonte: https://pt.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise

Terrenos

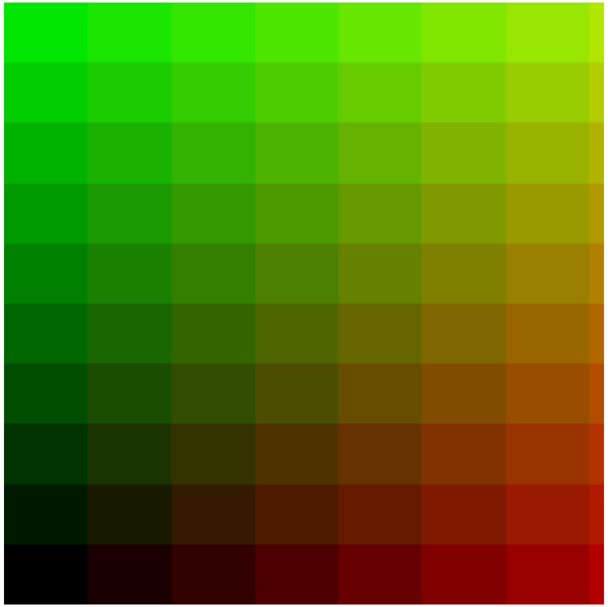
Por exemplo no horizonte dessa imagem, vemos que as alturas da montanha não mudam de forma totalmente aleatória.

fonte: https://www.youtube.com/watch?v=H6FhG9VKhJg

Pergunta

Como ficará a tela nesse caso?

```
float rand(vec2 st) {
  float val = dot(st.xy,vec2(12.9898,78.233));
  return fract(sin(val)*43758.5453123);
}


void mainImage(out vec4 fragColor, in vec2 fragCoord){
  vec2 st = fragCoord.xy/iResolution.xy;
  st *= 10.0;
  vec2 ipos = floor(st);
  fragColor = vec4(ipos/10.0, 0.0 ,1.0);
}
```

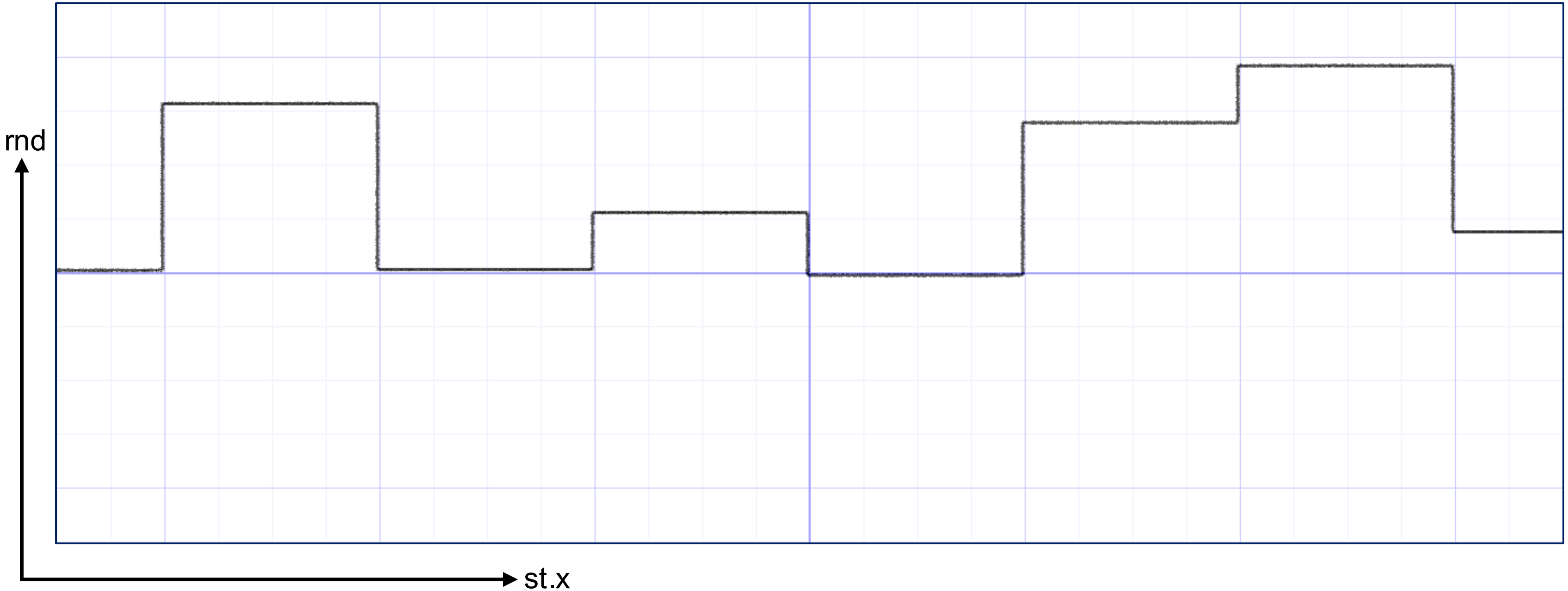


Perceba que agora posso colocar o **ipos** como entrada da função **rand()**

Referência: The Book of Shaders https://thebookofshaders.com/10/

Funções aleatórias

Vamos verificar a função recém criada rand() em GLSL em 1D




```
vec2 ipos = floor(st); // inteiro
float rnd = rand(ipos);
```

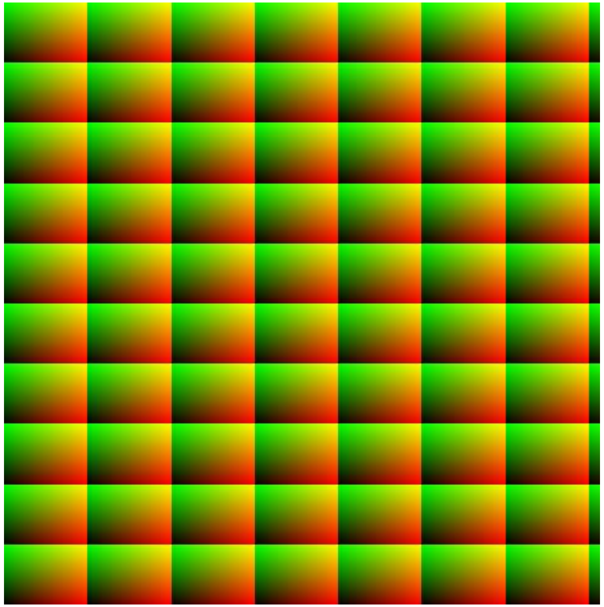

Referência: The Book of Shaders https://thebookofshaders.com/10/

Insper

Pergunta

Como ficará a tela agora?

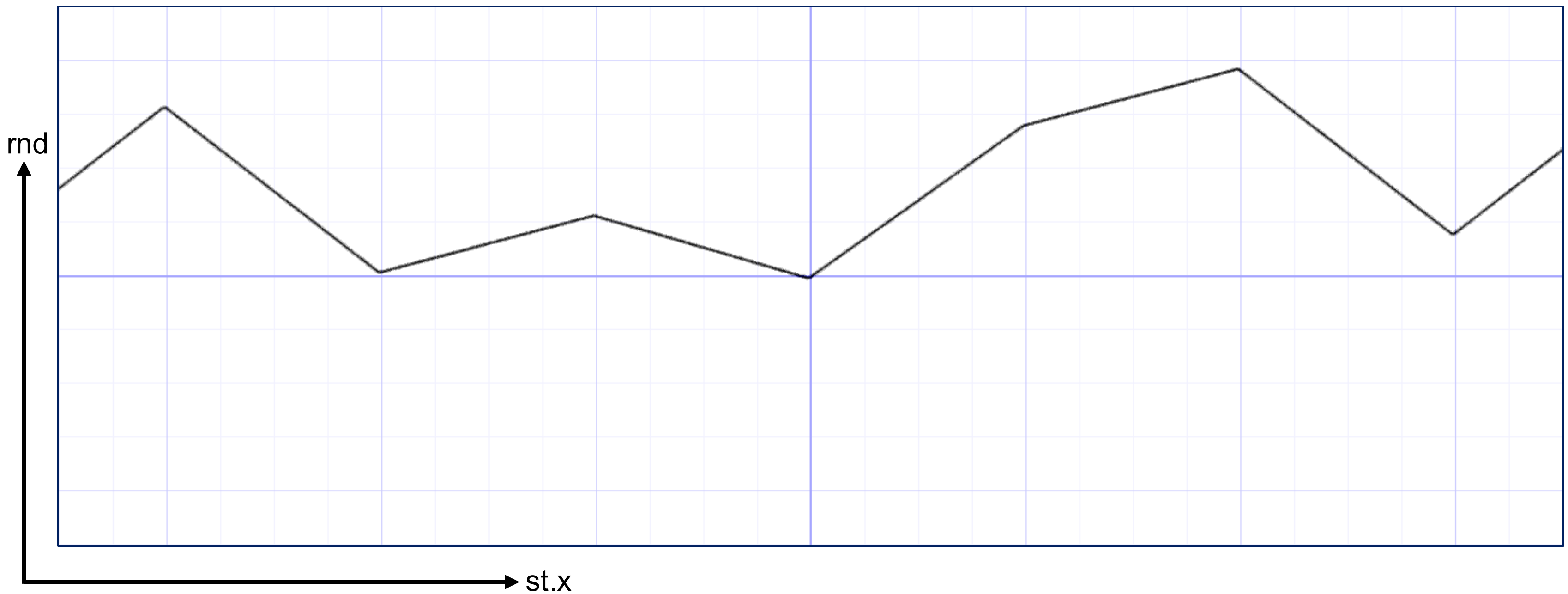
```
float rand(vec2 st) {
  float val = dot(st.xy,vec2(12.9898,78.233));
  return fract(sin(val)*43758.5453123);
}


void mainImage(out vec4 fragColor, in vec2 fragCoord){
  vec2 st = fragCoord.xy/iResolution.xy;
  st *= 10.0;
  vec2 fpos = fract(st);
  fragColor = vec4(fpos, 0.0 ,1.0);
}
```



Referência: The Book of Shaders https://thebookofshaders.com/10/

Funções aleatórias

Como você acha que ficaria agora?



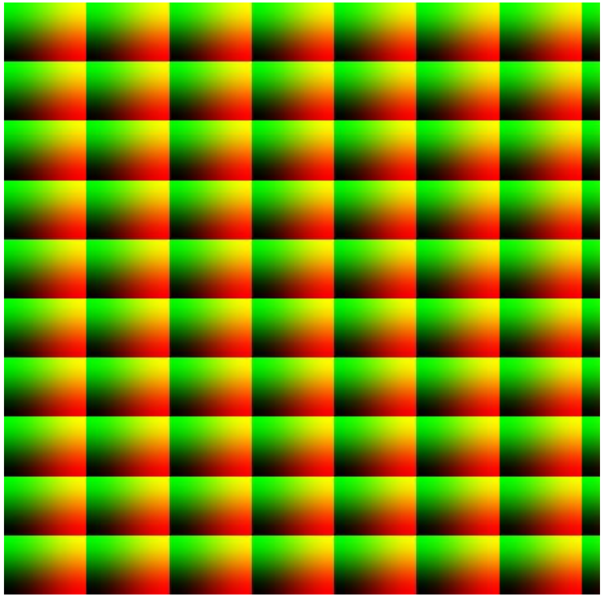

```
vec2 ipos = floor(st); // inteiro
vec2 fpos = fract(st); // fração
y = mix(rand(ipos), rand(ipos + 1.0), fpos);
```


Referência: The Book of Shaders https://thebookofshaders.com/10/

Pergunta

E agora?

```
float rand(vec2 st) {
  float val = dot(st.xy,vec2(12.9898,78.233));
  return fract(sin(val)*43758.5453123);
}


void mainImage(out vec4 fragColor, in vec2 fragCoord){
  vec2 st = fragCoord.xy/iResolution.xy;
  st *= 10.0;
  vec2 fpos = fract(st);
  fragColor = vec4(smoothstep(0.,1.,fpos), 0.0 ,1.0);
}
```

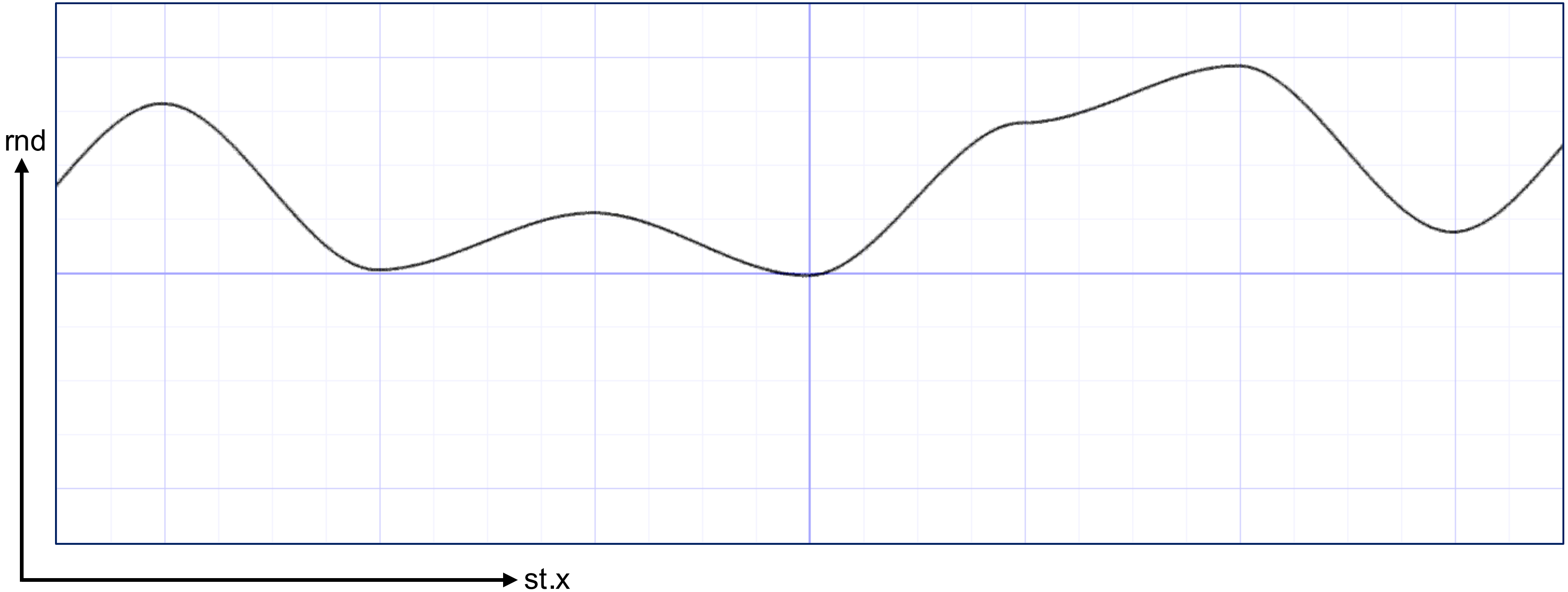


Referência: The Book of Shaders https://thebookofshaders.com/10/

Insper

Funções aleatórias

E agora?



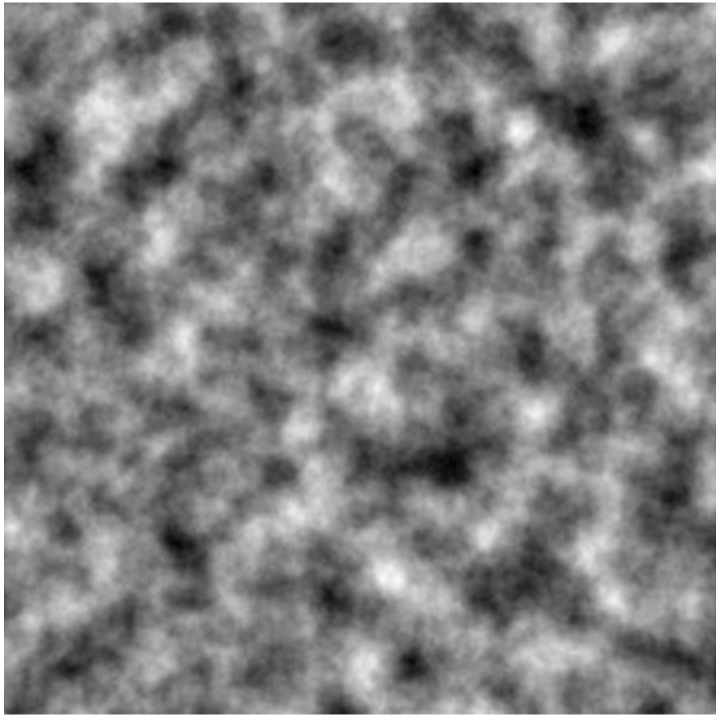

```
vec2 ipos = floor(st); // inteiro
vec2 fpos = fract(st); // fração
y = mix(rand(ipos), rand(ipos + 1.0), smoothstep(0.,1.,fpos));
```


Referência: The Book of Shaders https://thebookofshaders.com/10/

O que é o Perlin Noise

Geração aleatória de valores

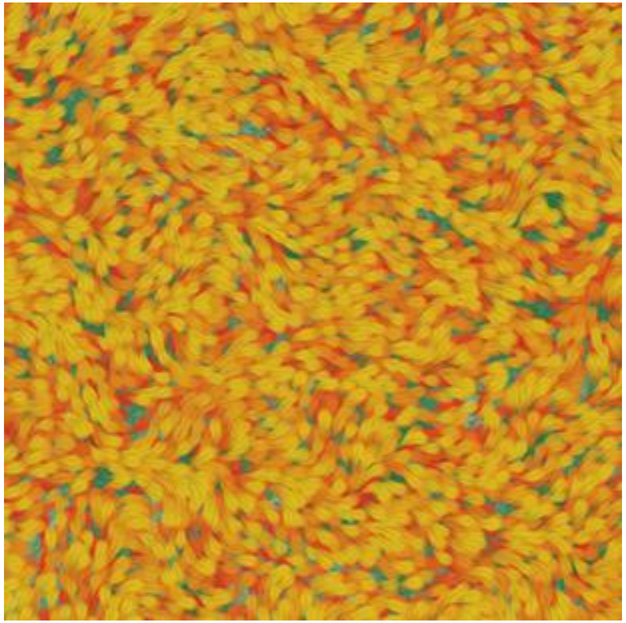
- Função n-dimensional
- Completa aleatoriedade
- Coerência



Perlin Noise

Gera padrões mais orgânicos

Percebe-se uma certa suavidade interna



fonte: https://sighack.com/post/getting-creative-with-perlin-noise-fields

Exemplos usando Perlin Noise

terrenos

rugosidades

FurryBall 4.8

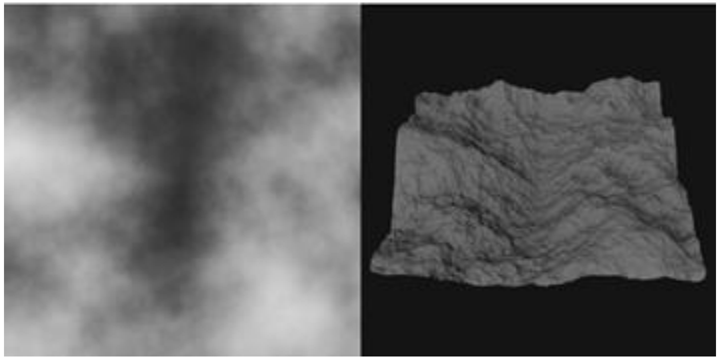
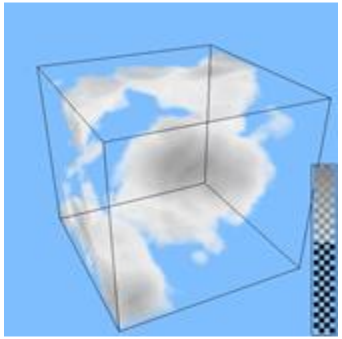


Fig. 2. Seven layers of stacked Perlin noise patterns (left) and the...

CMSC 425: Lecture 14 Procedural Generation: Perlin Noise

nuvens



caustics

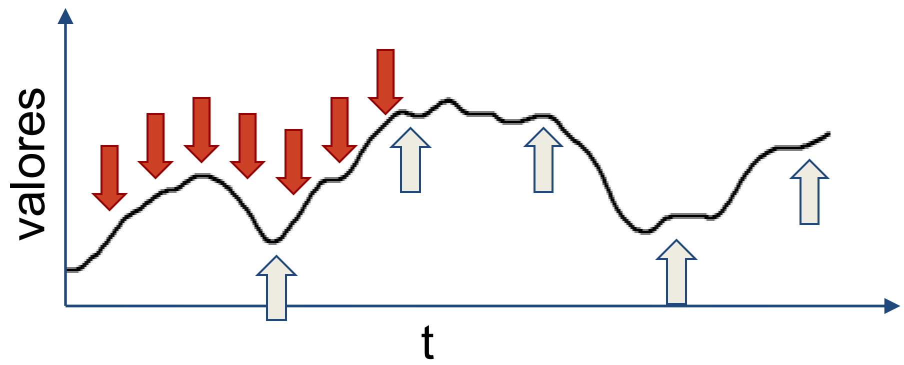
https://www.youtube.com/watch?v=n6eaQqKb4y0

https://github.com/BrutPitt/PerlinNoise4D

https://www.tvpaint.com/forum/viewtopic.php?t=11197

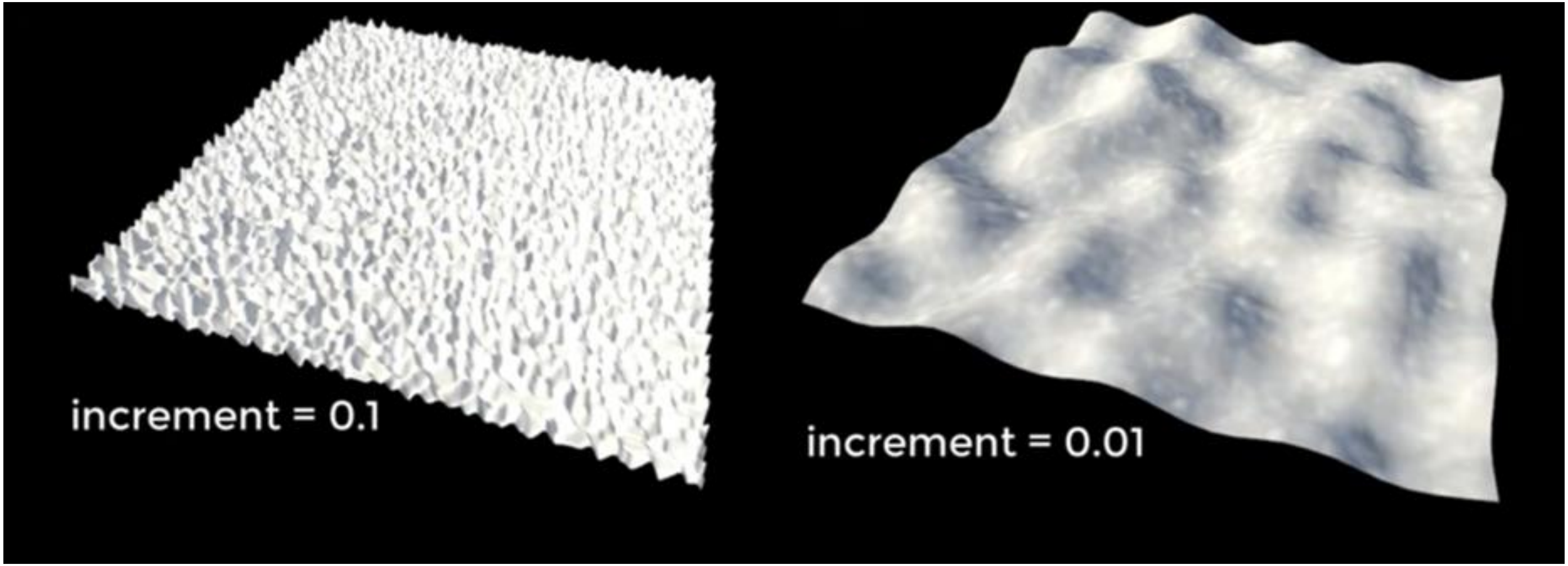
Função Parametrizada

função noise(t)



amostrando sobre a função.

Perlin Noise em 2D



fonte: https://www.youtube.com/watch?v=H6FhG9VKhJg

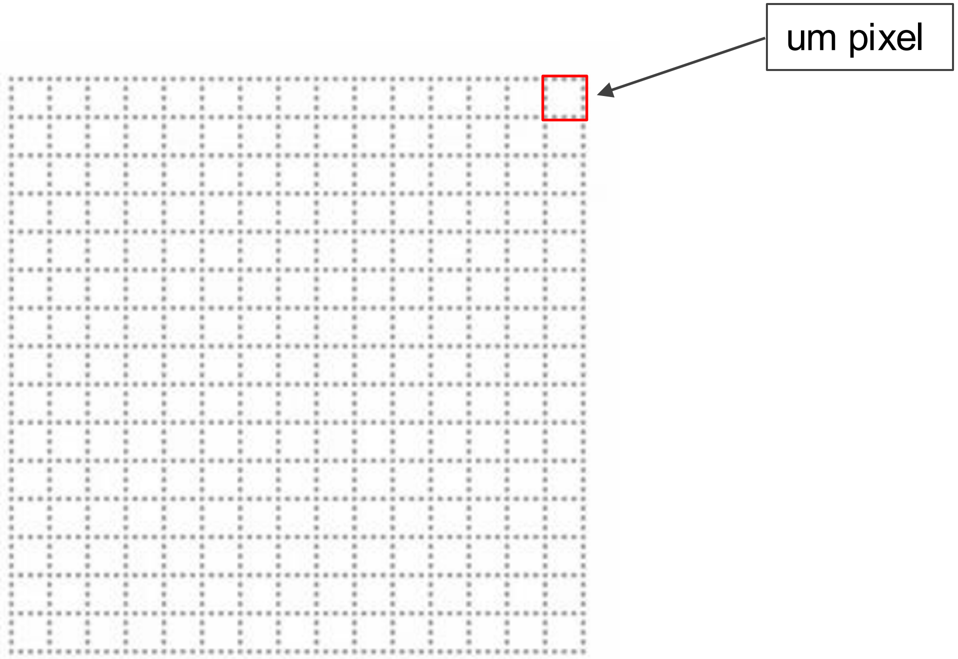
Explicações da Implementação do Perlin Noise

Passos básicos:

1. Definição do grid
2. Produto escalar do gradiente semi-aleatórios e vetores de distância
3. Interpolação dos produtos escalares

Imagem de referência para desenharmos

Por exemplo vamos usar uma imagem de 15x15 pixels.



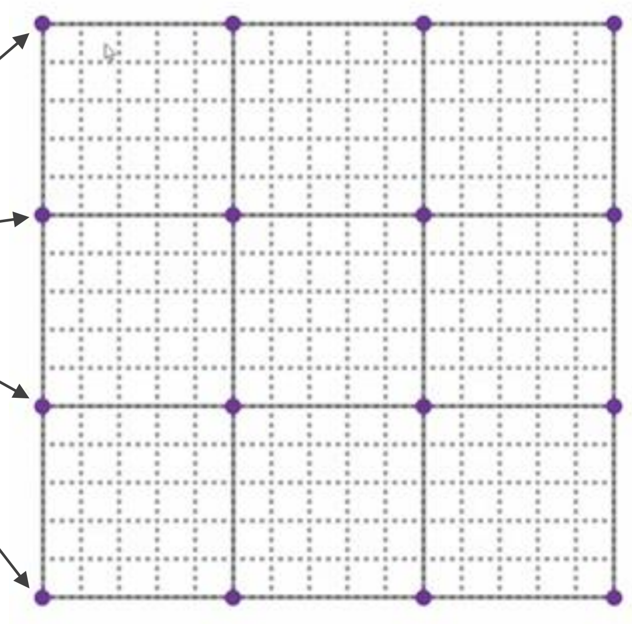
fonte: http://fataho.com/

Definição do Grid com valores aleatórios

Uma subdivisão é gerada para identificar os valores aleatórios. No caso os pontos violetas são os valores aleatórios.

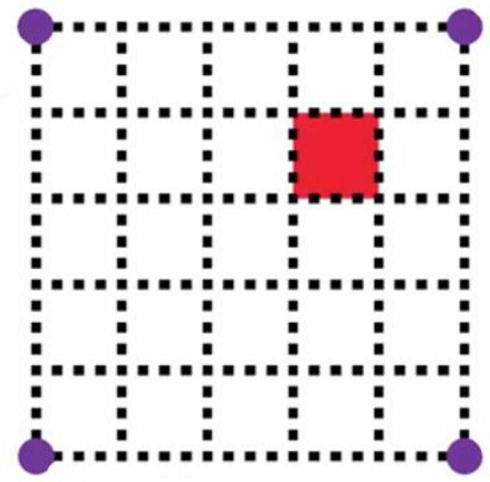
Aqui temos um grid de 4x4.

Cada um desses vértices tem um valor aleatório.



fonte: http://fataho.com/

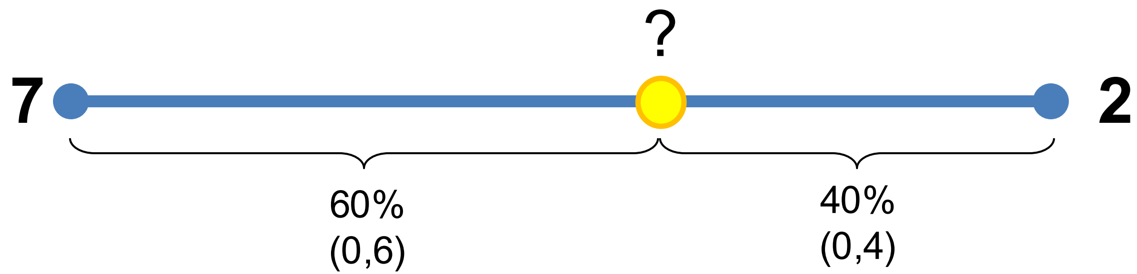
Calculando um valor na célula de um grid



fonte: http://fataho.com/

Interpolação Linear

Qual o valor do ponto?

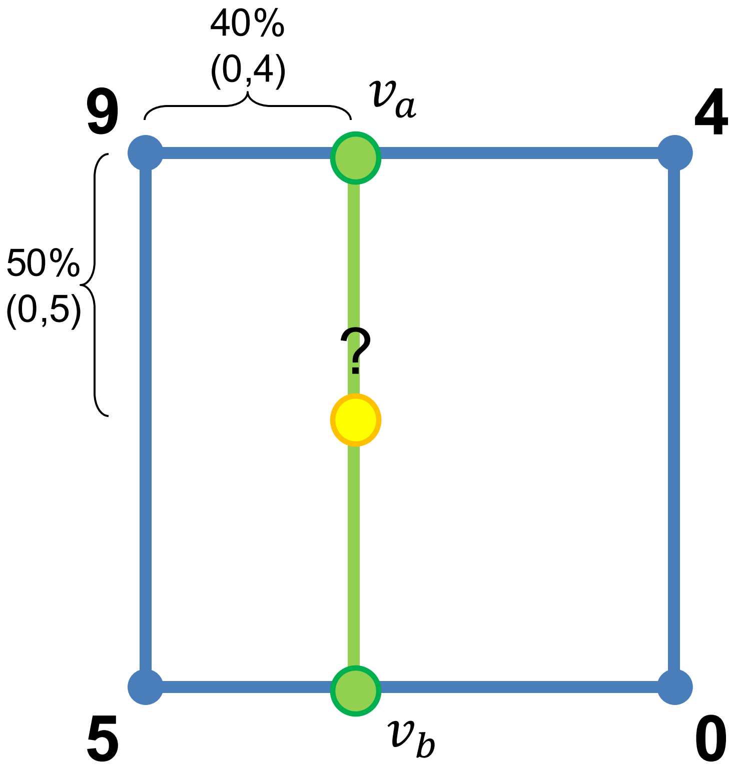


$$v = v_0 + t(v_1 - v_0)$$

$$v = 7 + 0{,}6(2 - 7)$$
$$v = 4$$

Filtro Bilinear

Qual o valor do ponto agora?



$$v = v_0 + t(v_1 - v_0)$$

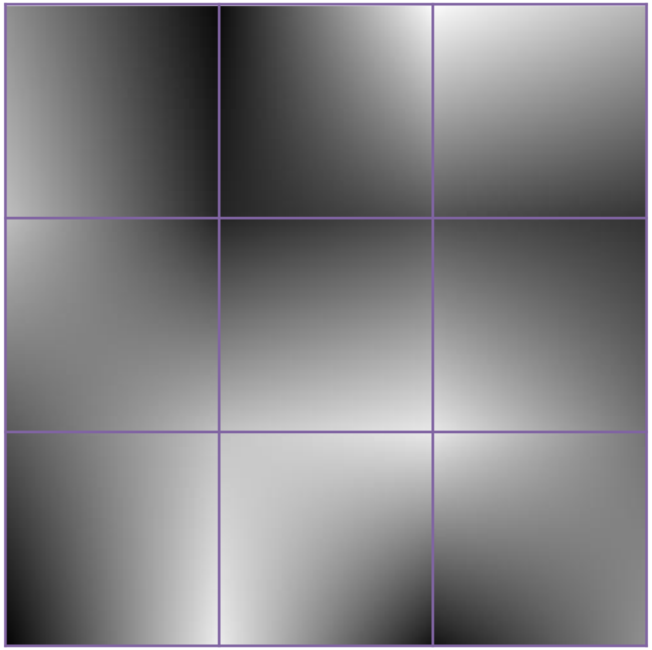
$$v_a = 9 + 0{,}4(4 - 9) = 7$$
$$v_b = 5 + 0{,}4(0 - 5) = 3$$

$$\boxed{v = 7 + 0{,}5(3 - 7) = 5}$$

Insper

Calculando o valor em GLSL

```
uniform vec2 u_resolution;
float random(vec2 st) {
  float val = dot(st.xy,vec2(12.9898,78.233));
  return fract(sin(val)*43758.5453123);
}

float noise (in vec2 st) {
  vec2 i = floor(st);
  vec2 f = fract(st);

  // Quatro cantos da célula
  float u00 = random(i);
  float u10 = random(i + vec2(1.0, 0.0));
  float u01 = random(i + vec2(0.0, 1.0));
  float u11 = random(i + vec2(1.0, 1.0));

  // Interpolação Bilinear
  float u0 = mix(u00, u10, f.x);
  float u1 = mix(u01, u11, f.x);
  return mix(u0, u1, f.y);
}

void main() {
  vec2 st = gl_FragCoord.xy/u_resolution.xy;
  vec2 pos = vec2(st*3.0);
  float n = noise(pos);
  gl_FragColor = vec4(vec3(n), 1.0);
}
```



Referência: The Book of Shaders https://thebookofshaders.com/10/

Problemas de Interpolação

Se uma interpolação linear for diretamente utilizada, poderão aparecer artefatos entre os subgrids.

Primeira proposta para solucionar o problema foi:

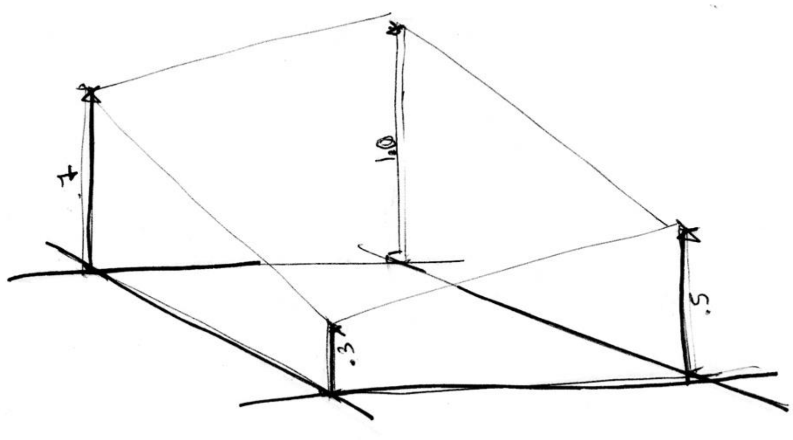
$$3t^2 - 2t^3$$

Que já conhecemos essa interpolação do Hermite, e que no GLSL pode ser usada diretamente com a chamada:

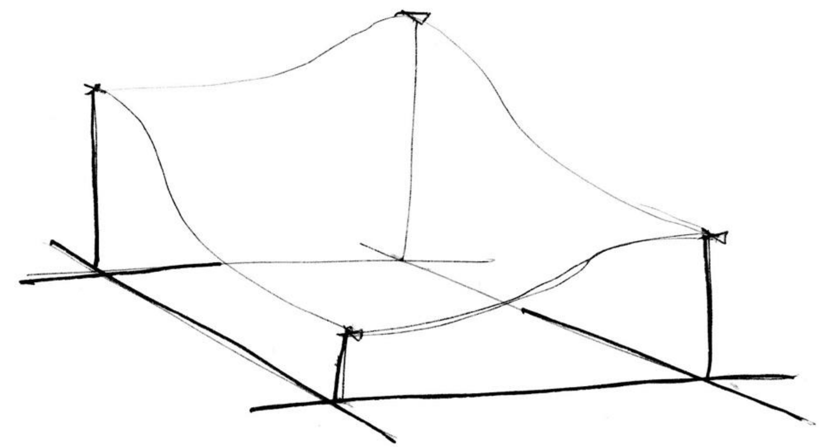
smoothstep()

Insper

Interpolação Cúbica



Linear

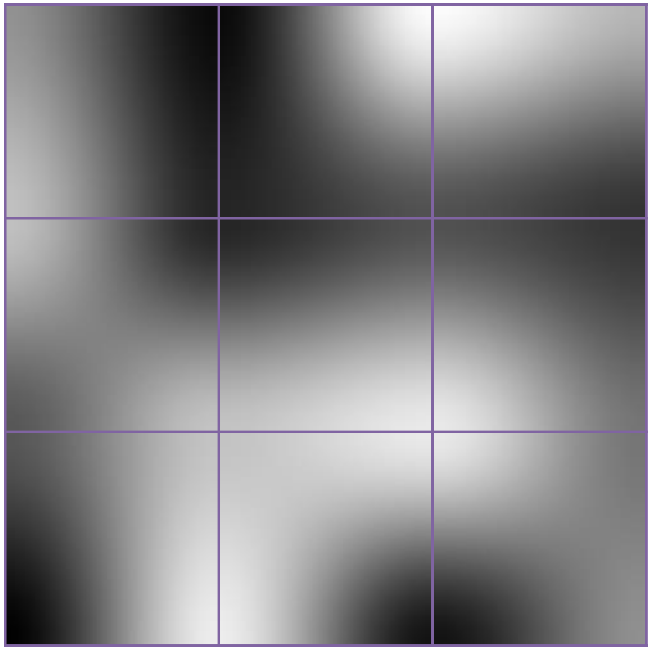


Cúbica (Hermite)

Referência: The Book of Shaders https://thebookofshaders.com/10/

Calculando o valor em GLSL

```
uniform vec2 u_resolution;
float random(vec2 st) {
  float val = dot(st.xy,vec2(12.9898,78.233));
  return fract(sin(val)*43758.5453123);
}

float noise (in vec2 st) {
  vec2 i = floor(st);
  vec2 f = fract(st);

  // Quatro cantos da célula
  float u00 = random(i);
  float u10 = random(i + vec2(1.0, 0.0));
  float u01 = random(i + vec2(0.0, 1.0));
  float u11 = random(i + vec2(1.0, 1.0));

  // Hermite
  vec2 u = smoothstep(0.,1.,f);

  // Interpolação Bilinear
  float u0 = mix(u00, u10, u.x);
  float u1 = mix(u01, u11, u.x);
  return mix(u0, u1, u.y);
}

void main() {
  vec2 st = gl_FragCoord.xy/u_resolution.xy;
  vec2 pos = vec2(st*3.0);
  float n = noise(pos);
  gl_FragColor = vec4(vec3(n), 1.0);
}
```



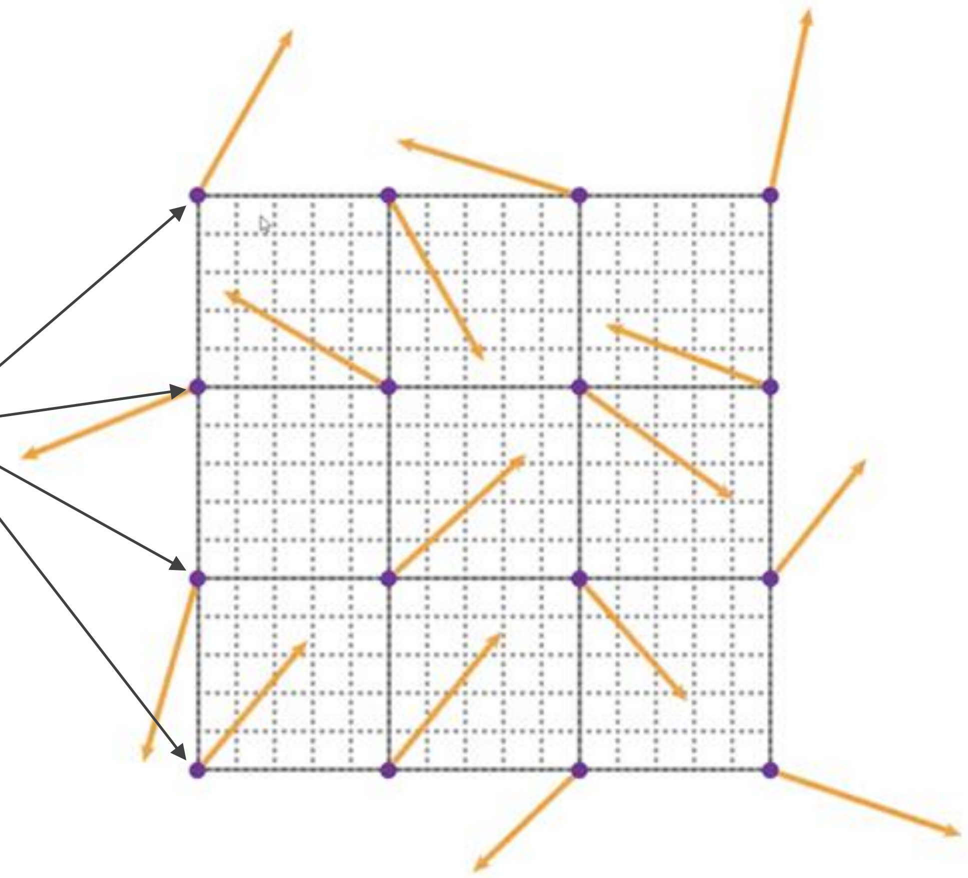
Referência: The Book of Shaders https://thebookofshaders.com/10/

Insper

Grid dos vetores aleatórios

Vetores aleatórios são gerados por vértice e depois produtos escalares são realizados.

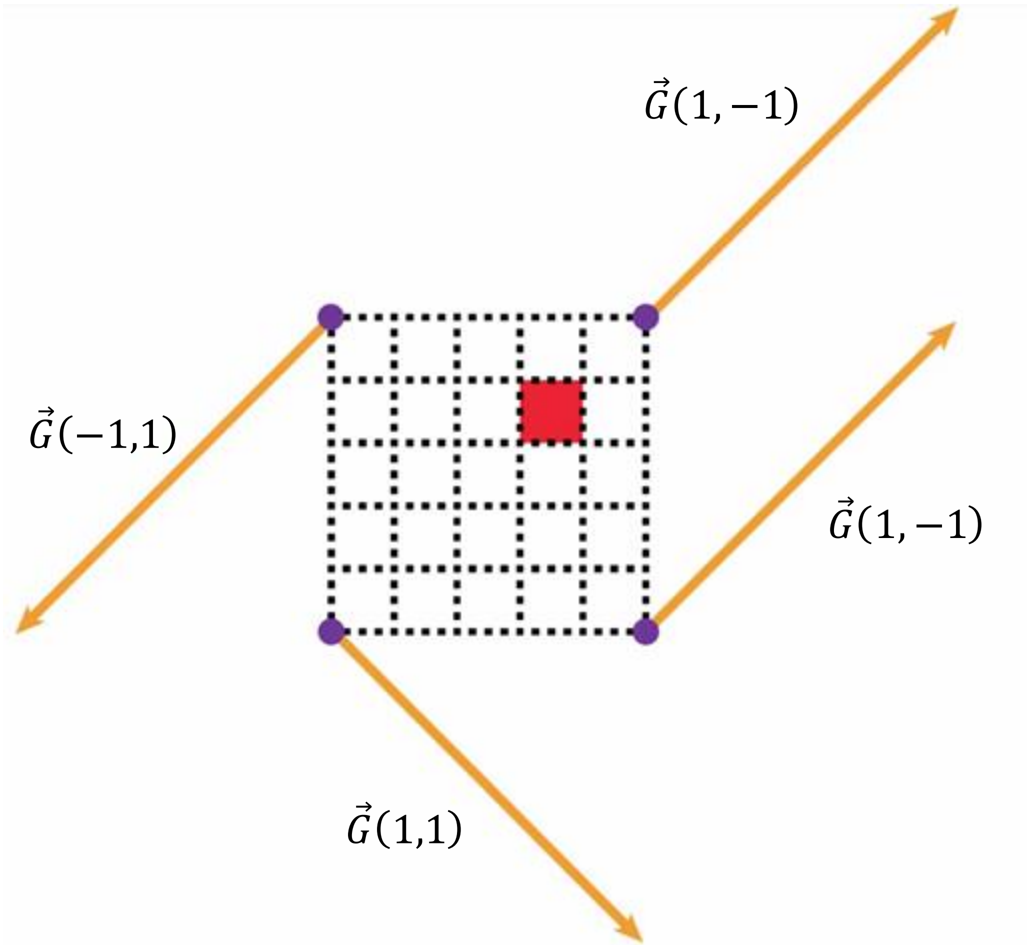
Cada um desses vértices tem um vetor aleatório.



fonte: http://fataho.com/

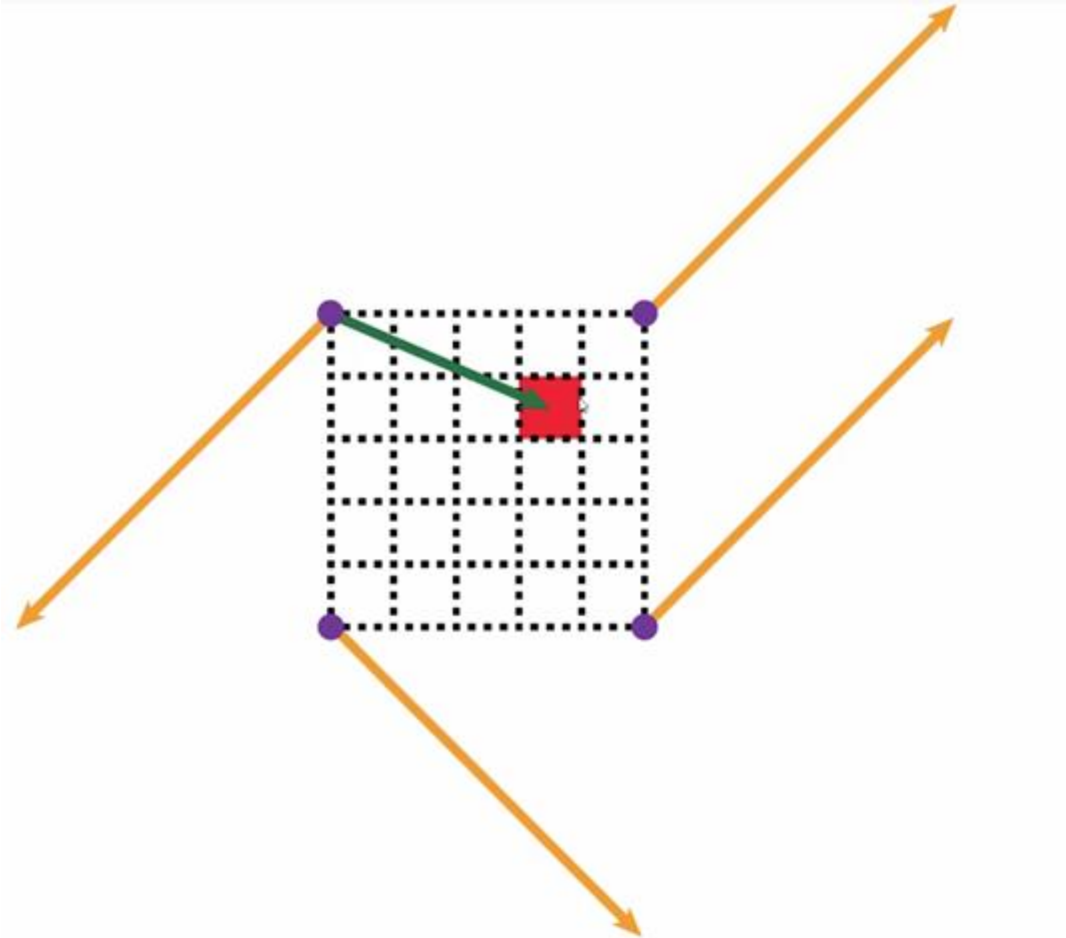
Insper

Calculando um valor no grid



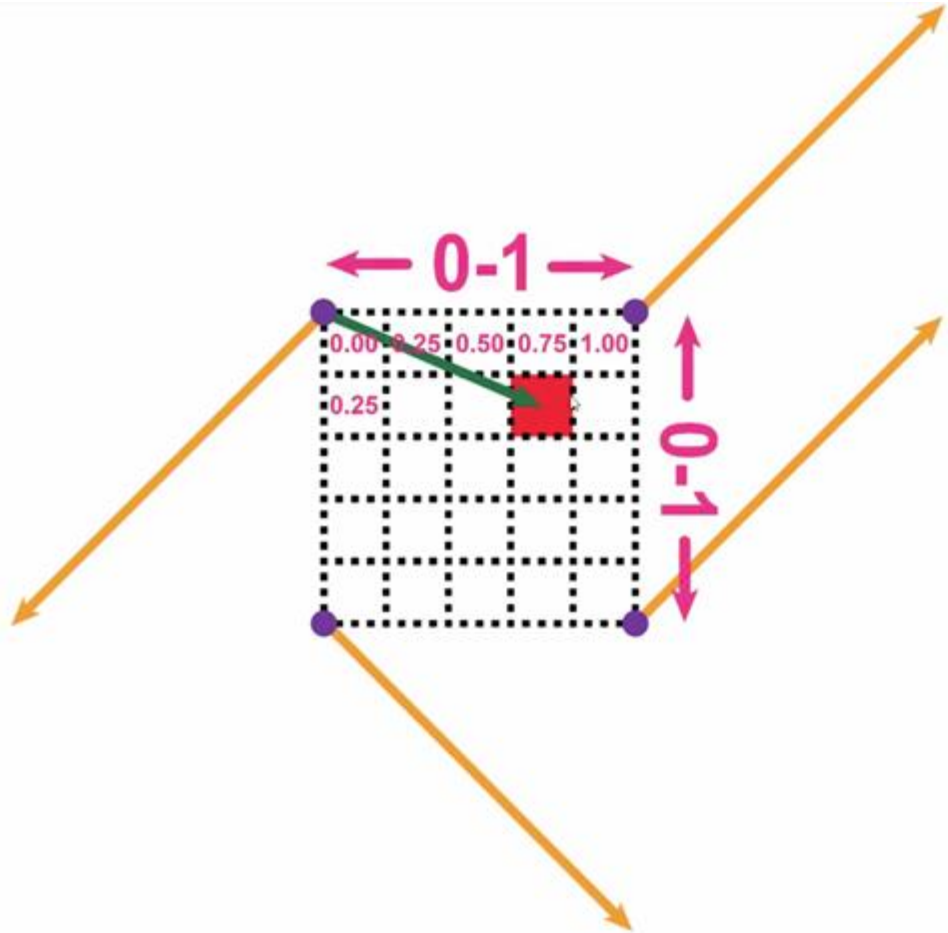
fonte: http://fataho.com/

Vetor Distância



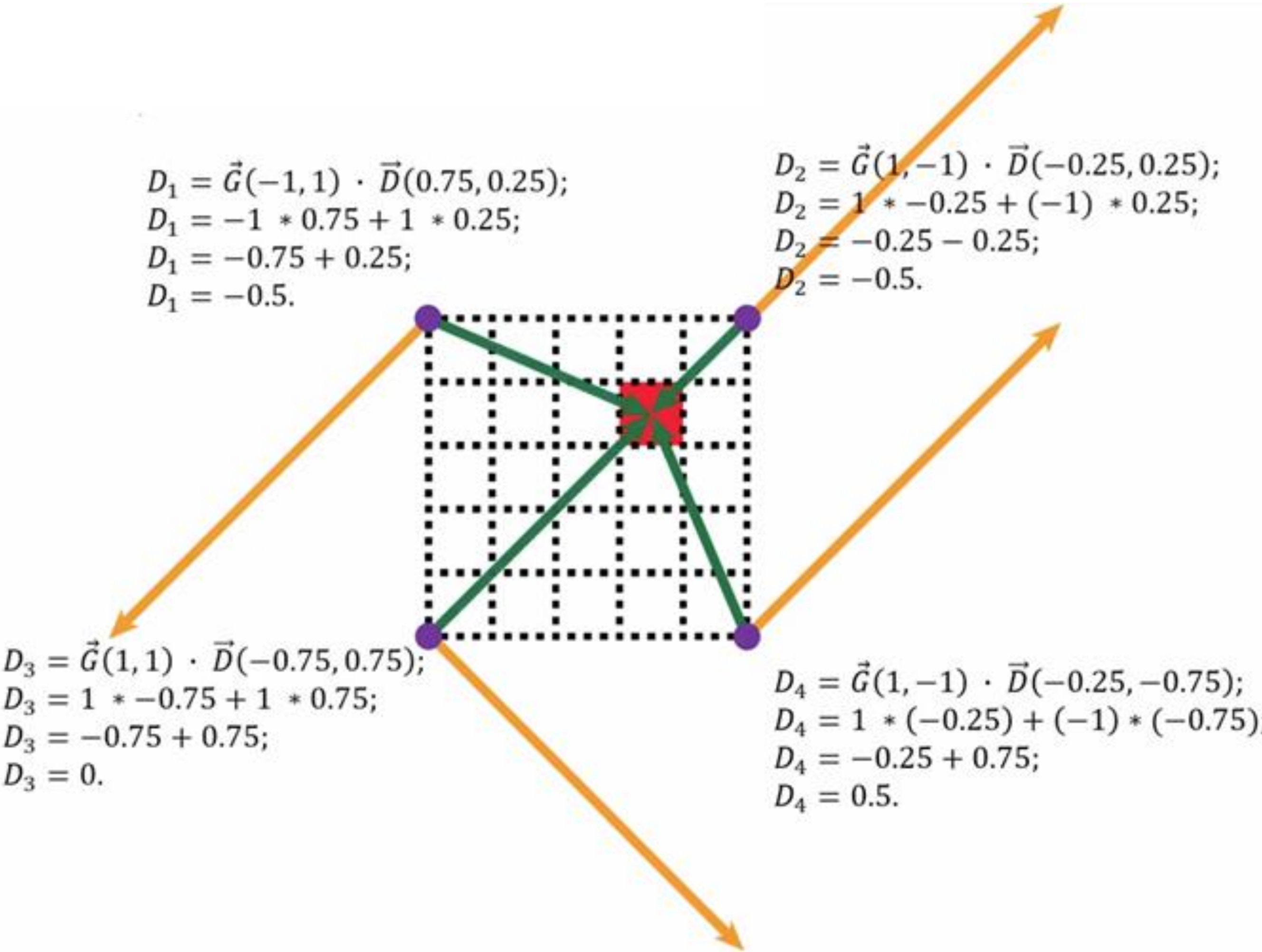
fonte: http://fataho.com/

Vetor Distância



fonte: http://fataho.com/

Produtos Escalares dos 4 cantos

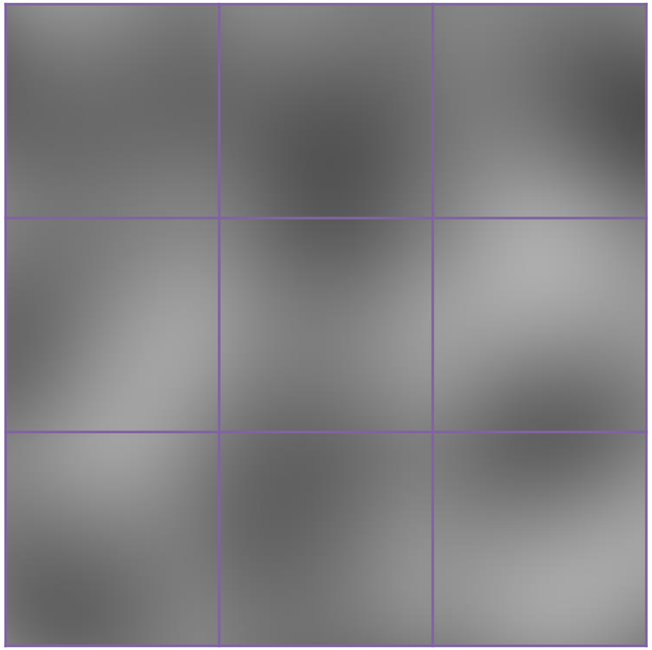


fonte: http://fataho.com/

Insper

Calculando o valor em GLSL

```
uniform vec2 u_resolution;
vec2 random2(vec2 st){
  vec2 val = vec2(dot(st,vec2(127.1,311.7)),
  dot(st,vec2(269.5,183.3)) );
  return -1.0 + 2.0*fract(sin(val)*43758.5453123);
}

float noise(vec2 st) {
  vec2 i = floor(st); vec2 f = fract(st);

  // Quatro cantos da célula
  float u00 = dot(random2(i + vec2(0.0,0.0)), f - vec2(0.0,0.0));
  float u10 = dot(random2(i + vec2(1.0,0.0)), f - vec2(1.0,0.0));
  float u01 = dot(random2(i + vec2(0.0,1.0)), f - vec2(0.0,1.0));
  float u11 = dot(random2(i + vec2(1.0,1.0)), f - vec2(1.0,1.0));

  // Hermite
  vec2 u = smoothstep(0.,1.,f);

  // Interpolação Bilinear
  float u0 = mix(u00, u10, u.x);
  float u1 = mix(u01, u11, u.x);
  return mix(u0, u1, u.y)*.5+.5;
}

void main() {
  vec2 st = gl_FragCoord.xy/u_resolution.xy;
  vec2 pos = vec2(st*3.0);
  float n = noise(pos);
  gl_FragColor = vec4(vec3(n), 1.0);
}
```



Referência: The Book of Shaders https://thebookofshaders.com/10/

Mais Problemas de Interpolação

A equação do Hermite funcionou bem, porém apresenta descontinuidade na sua segunda derivada, dessa forma a equação usada na versão melhorada (improved) é chamada de Fade Function e tem a seguinte formulação:

$$6t^5 - 15t^4 + 10t^3$$

Funções

Linear

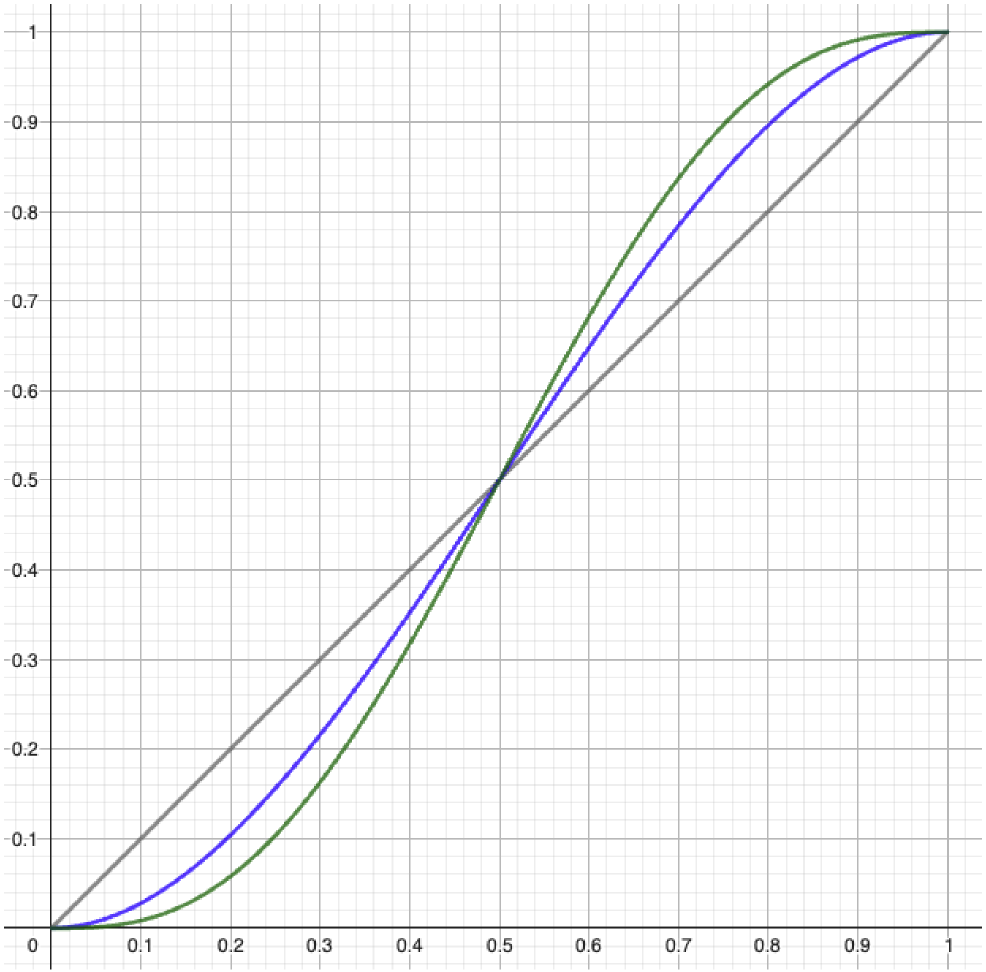
$l(t) = t$

Hermite

$h(t) = 3t^2 - 2t^3$

Fade

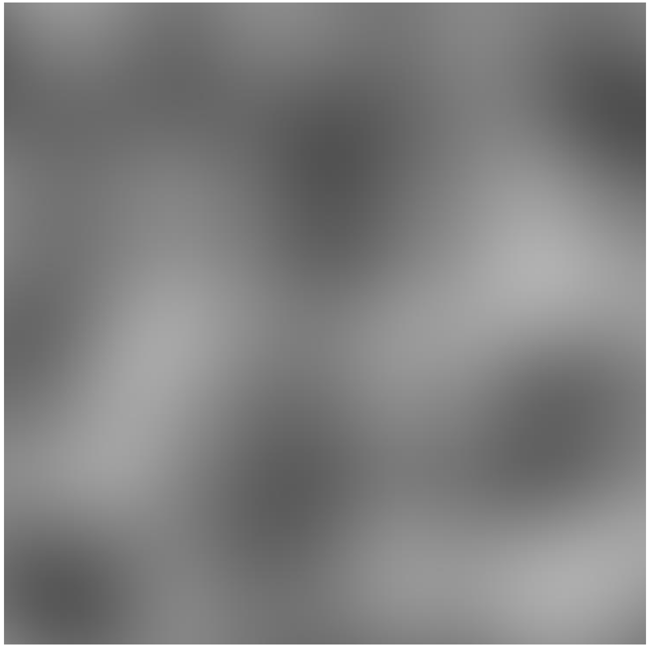
$f(t) = 6t^5 - 15t^4 + 10t^3$



Insper

Calculando o valor em GLSL

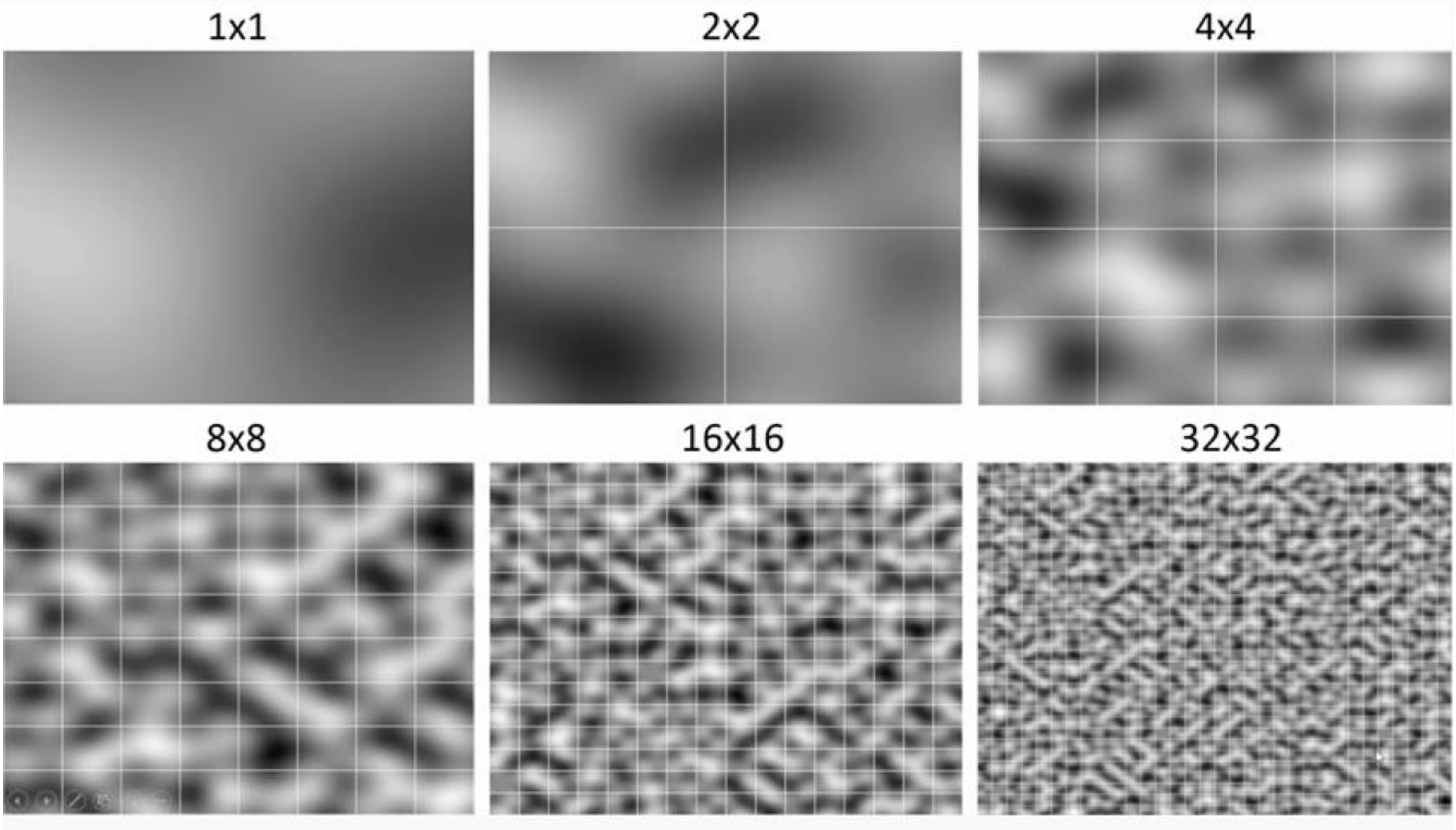
```
uniform vec2 u_resolution;
vec2 random2(vec2 st){
  vec2 val = vec2(dot(st,vec2(127.1,311.7)),
  dot(st,vec2(269.5,183.3)) );
  return -1.0 + 2.0*fract(sin(val)*43758.5453123);
}

float noise(vec2 st) {
  vec2 i = floor(st); vec2 f = fract(st);

  // Quatro cantos da célula
  float u00 = dot(random2(i + vec2(0.0,0.0)), f - vec2(0.0,0.0));
  float u10 = dot(random2(i + vec2(1.0,0.0)), f - vec2(1.0,0.0));
  float u01 = dot(random2(i + vec2(0.0,1.0)), f - vec2(0.0,1.0));
  float u11 = dot(random2(i + vec2(1.0,1.0)), f - vec2(1.0,1.0));

  // Equação quíntica (Fase function)
  vec2 u = f*f*f*(f*(f*6.-15.0)+10.0);

  // Interpolação Bilinear
  float u0 = mix(u00, u10, u.x);
  float u1 = mix(u01, u11, u.x);
  return mix(u0, u1, u.y)*.5+.5;
}

void main() {
  vec2 st = gl_FragCoord.xy/u_resolution.xy;
  vec2 pos = vec2(st*3.0);
  float n = noise(pos);
  gl_FragColor = vec4(vec3(n), 1.0);
}
```



Equação quíntica

Referência: The Book of Shaders https://thebookofshaders.com/10/

Número de células



fonte: http://fataho.com/

Insper

Artigo do Perlin Noise (Improved)

https://mrl.cs.nyu.edu/~perlin/paper445.pdf

Improving Noise

Ken Perlin
Media Research Laboratory, Dept. of Computer Science, New York University
perlin@cat.nyu.edu

ABSTRACT

Two deficiencies in the original Noise algorithm are corrected: second order interpolation discontinuity and unoptimal gradient computation. With these defects corrected, Noise both looks better and runs faster. The latter change also makes it easier to define a uniform mathematical reference standard.

Keywords

procedural texture

1 INTRODUCTION

Since its introduction 17 years ago [Perlin 1984; Perlin 1985; Perlin and Hoffert 1989], Noise has found wide use in graphics [Foley et al. 1996; Upstill 1990]. The original algorithm, although efficient, suffered from two defects: second order discontinuity across coordinate-aligned integer boundaries, and a needlessly expensive and somewhat problematic method of computing the gradient. We (belatedly) remove these defects.

2 DEFICIENCIES IN ORIGINAL ALGORITHM

As detailed in [Ebert et al 1998], Noise is determined at point **(x,y,z)** by computing a pseudo-random gradient at each of the eight nearest vertices on the integer cubic lattice and then doing splined interpolation. Let **(i,j,k)** denote the eight points on this cube, where **i** is the set of lower and upper bounding integers on **x**: $\{\lfloor x \rfloor, \lfloor x \rfloor+1\}$, and similarly $j = \{\lfloor y \rfloor, \lfloor y \rfloor+1\}$ and $k = \{\lfloor z \rfloor, \lfloor z \rfloor+1\}$. The eight gradients are given by $g_{i,j,k} = G[P[P[P[i]+j]+k]]$ where precomputed arrays **P** and **G** contain, respectively, a pseudo-random permutation, and pseudo-random unit-length gradient vectors. The successive application of **P** hashes each lattice point to de-correlate the indices into **G**. The eight linear functions $g_{i,j,k} \cdot (x-i,y-j,z-k)$ are then trilinearly interpolated by $s(x-\lfloor x \rfloor)$, $s(y-\lfloor y \rfloor)$ and $s(z-\lfloor z \rfloor)$, where $s(t) = 3t^2-2t^3$.

The above algorithm is very efficient but contains some deficiencies. One is in the cubic interpolant function's second derivative **6-12t**, which is not zero at either **t=0** or **t=1**. This non-zero value creates second order discontinuities across the coordinate-aligned faces of adjoining cubic cells. These discontinuities become noticeable when a Noise-displaced surface is shaded; then the surface normal (which is itself a derivative operator) has a visibly discontinuous derivative (Figure 1a).

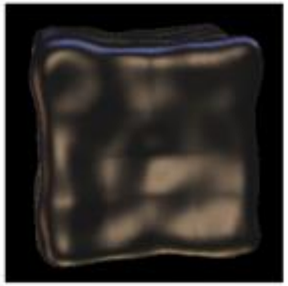


Figure 1a: Noise-displaced superquadric with old interpolants

Figure 1b: Noise-displaced superquadric with new interpolants

The second deficiency is that whereas the gradients in **G** are distributed uniformly over a sphere, the cubic grid itself has directional biases, being shortened along the axes and elongated on the diagonals between opposite cube vertices. This directional asymmetry tends to cause a sporadic clumping effect, where nearby gradients that are almost axis-aligned, and therefore close together, happen to align with each other, causing anomalously high values in those regions (Figure 2a).

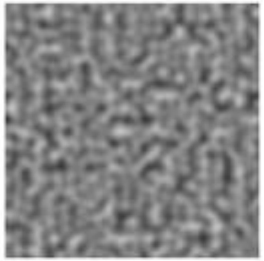


Figure 2a: High-frequency Noise, with old gradient distributions

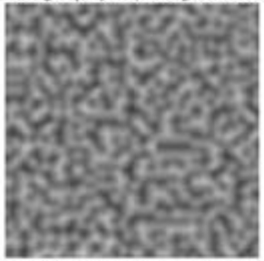


Figure 2b: High-frequency Noise, with new gradient distributions

3 MODIFICATIONS

The above deficiencies are addressed as follows. $3t^2-2t^3$ is replaced by $6t^5-15t^4+10t^3$, which has zero first and second derivatives at both **t=0** and **t=1**. The absence of artifacts can be seen in Figure 1b.

The key to removing directional bias in the gradients is to skew the set of gradient directions away from the coordinate axes and long diagonals. In fact, it is not necessary for **G** to be random at all, since **P** provides plenty of randomness. The corrected version replaces **G** with the 12 vectors defined by the directions from the center of a cube to its edges:

$$(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),$$
$$(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),$$
$$(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1)$$

Gradients from this set are chosen by using the result of **P**, modulo 12. This set of gradient directions was chosen for two reasons: *(i)* it avoids the main axis and long diagonal directions, thereby avoiding the possibility of axis-aligned clumping, and *(ii)* it allows the eight inner products to be effected without requiring any multiplies, thereby removing 24 multiplies from the computation.

To avoid the cost of dividing by 12, we pad to 16 gradient directions, adding an extra **(1,1,0),(-1,1,0),(0,-1,1)** and **(0,-1,-1)**. These form a regular tetrahedron, so adding them redundantly introduces no visual bias in the texture. The final result has the same non-directional appearance as the original distribution but less clumping, as can be seen in Figure 2b.

4 PERFORMANCE

In a timing comparison (C implementations on the Intel optimizing compiler running on a Pentium 3), the new algorithm runs approximately ten percent faster than the original. The cost of the extra multiplies required to compute the three corrected interpolants is apparently outweighed by the savings from the multiplies no longer required to compute the eight inner products. Examination of the assembly code indicates that the Intel processor optimizes by pipelining the successive multiplies of the three interpolant calculations since no memory fetches are required within this block of computations.

Rather than use a 12-entry table to avoid inner product multiplies, the **G** table can also be expanded and used to replace the last lookup into **P**. Whether this method is more efficient is processor dependent. For example, 3D inner products are single operations on both **nVidia** and **ATI** pixel processors.

5 CONCLUSIONS

The described changes result in an implementation of Noise which is both visually improved and computationally more efficient. Also, with the pseudo-random gradient table removed, the only pseudo-random component left is the ordering of the permutation table **P**. Once a standard permutation order is determined, it will at last be possible to give a uniform mathematical definition for the Noise function, identical across all software and hardware environments.

ACKNOWLEDGEMENTS

Thanks to the reviewers for constructive criticisms that improved the paper, to Denis Zorin for his invaluable suggestions, and to Nathan Wardrip-Fruin and Chris Poultney, who helped greatly in the rush of production.

References

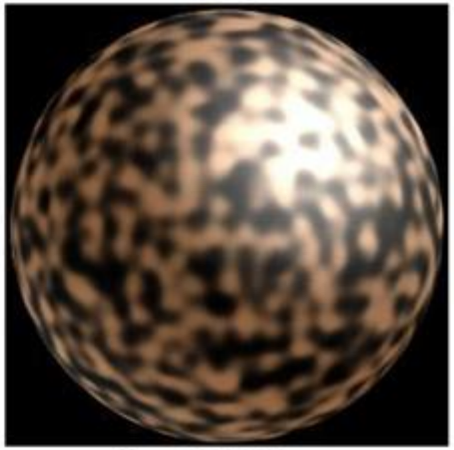
EBERT, D. ET AL. 1998. *Texturing and Modeling: A Procedural Approach.* Second Edition. AP Professional, Cambridge.
FOLEY, J. ET AL. 1996. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading.
PERLIN, K., ACM SIGGRAPH 84 conference, course in "Advanced Image Synthesis."
PERLIN, K. 1985. An Image Synthesizer. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85), 24, 3.*
PERLIN, K. AND HOFFERT, E. 1989. Hypertexture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 89), 23, 3.*
UPSTILL, S. 1990. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley.

Código do Perlin Noise (Improved)

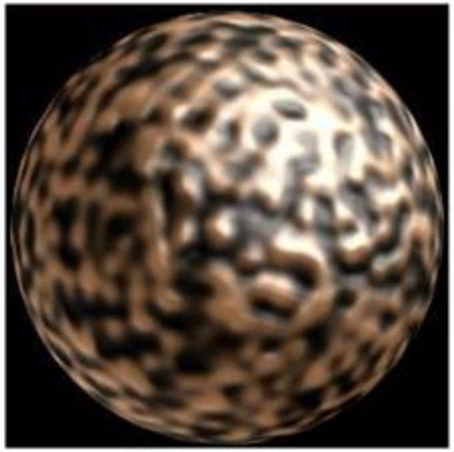
https://cs.nyu.edu/~perlin/noise/

This code implements the algorithm I describe in a corresponding [SIGGRAPH 2002 paper](#).

```
// JAVA REFERENCE IMPLEMENTATION OF IMPROVED NOISE - COPYRIGHT 2002 KEN PERLIN.

public final class ImprovedNoise {
   static public double noise(double x, double y, double z) {
      int X = (int)Math.floor(x) & 255,                  // FIND UNIT CUBE THAT
          Y = (int)Math.floor(y) & 255,                  // CONTAINS POINT.
          Z = (int)Math.floor(z) & 255;
      x -= Math.floor(x);                                // FIND RELATIVE X,Y,Z
      y -= Math.floor(y);                                // OF POINT IN CUBE.
      z -= Math.floor(z);
      double u = fade(x),                                // COMPUTE FADE CURVES
             v = fade(y),                                // FOR EACH OF X,Y,Z.
             w = fade(z);
      int A = p[X  ]+Y, AA = p[A]+Z, AB = p[A+1]+Z,      // HASH COORDINATES OF
          B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;      // THE 8 CUBE CORNERS,

      return lerp(w, lerp(v, lerp(u, grad(p[AA  ], x  , y  , z   ),  // AND ADD
                                     grad(p[BA  ], x-1, y  , z   )), // BLENDED
                             lerp(u, grad(p[AB  ], x  , y-1, z   ),  // RESULTS
                                     grad(p[BB  ], x-1, y-1, z   ))),// FROM  8
                     lerp(v, lerp(u, grad(p[AA+1], x  , y  , z-1 ),  // CORNERS
                                     grad(p[BA+1], x-1, y  , z-1 )), // OF CUBE
                             lerp(u, grad(p[AB+1], x  , y-1, z-1 ),
                                     grad(p[BB+1], x-1, y-1, z-1 ))));
   }
   static double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
   static double lerp(double t, double a, double b) { return a + t * (b - a); }
   static double grad(int hash, double x, double y, double z) {
      int h = hash & 15;                      // CONVERT LO 4 BITS OF HASH CODE
      double u = h<8 ? x : y,                 // INTO 12 GRADIENT DIRECTIONS.
             v = h<4 ? y : h==12||h==14 ? x : z;
      return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
   }
   static final int p[] = new int[512], 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
   };
   static { for (int i=0; i < 256 ; i++) p[256+i] = p[i] = permutation[i]; }
}
```

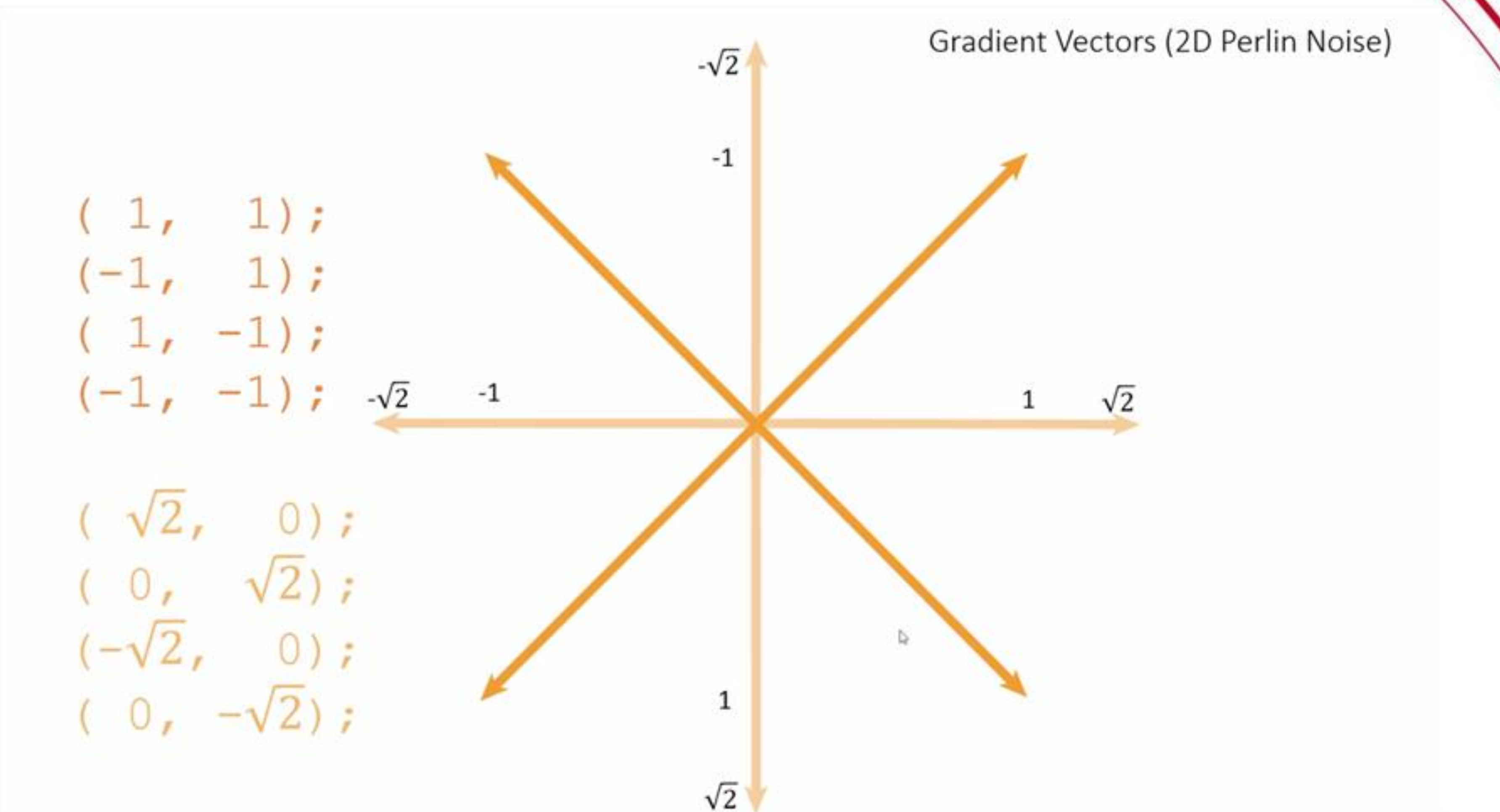


Smooth ball demo



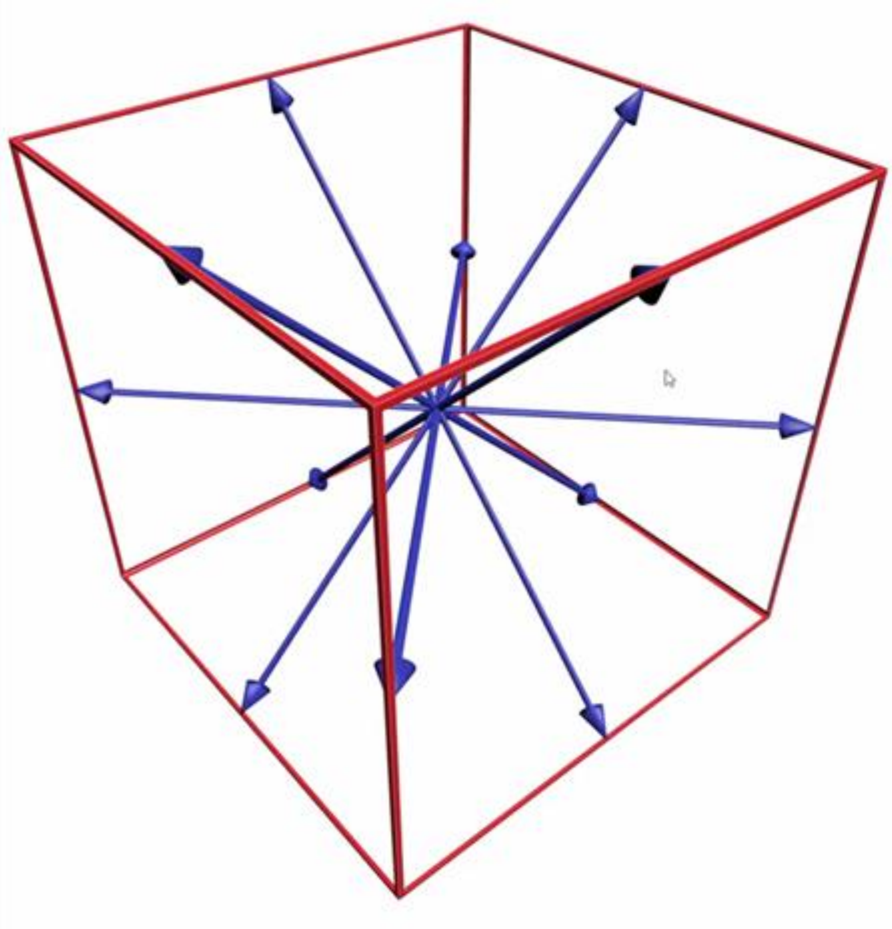
Bumpy ball demo

Vetores Gradiente 2D



fonte: http://fataho.com/

Vetores Gradiente 3D



Gradient Vectors (3D Perlin Noise)

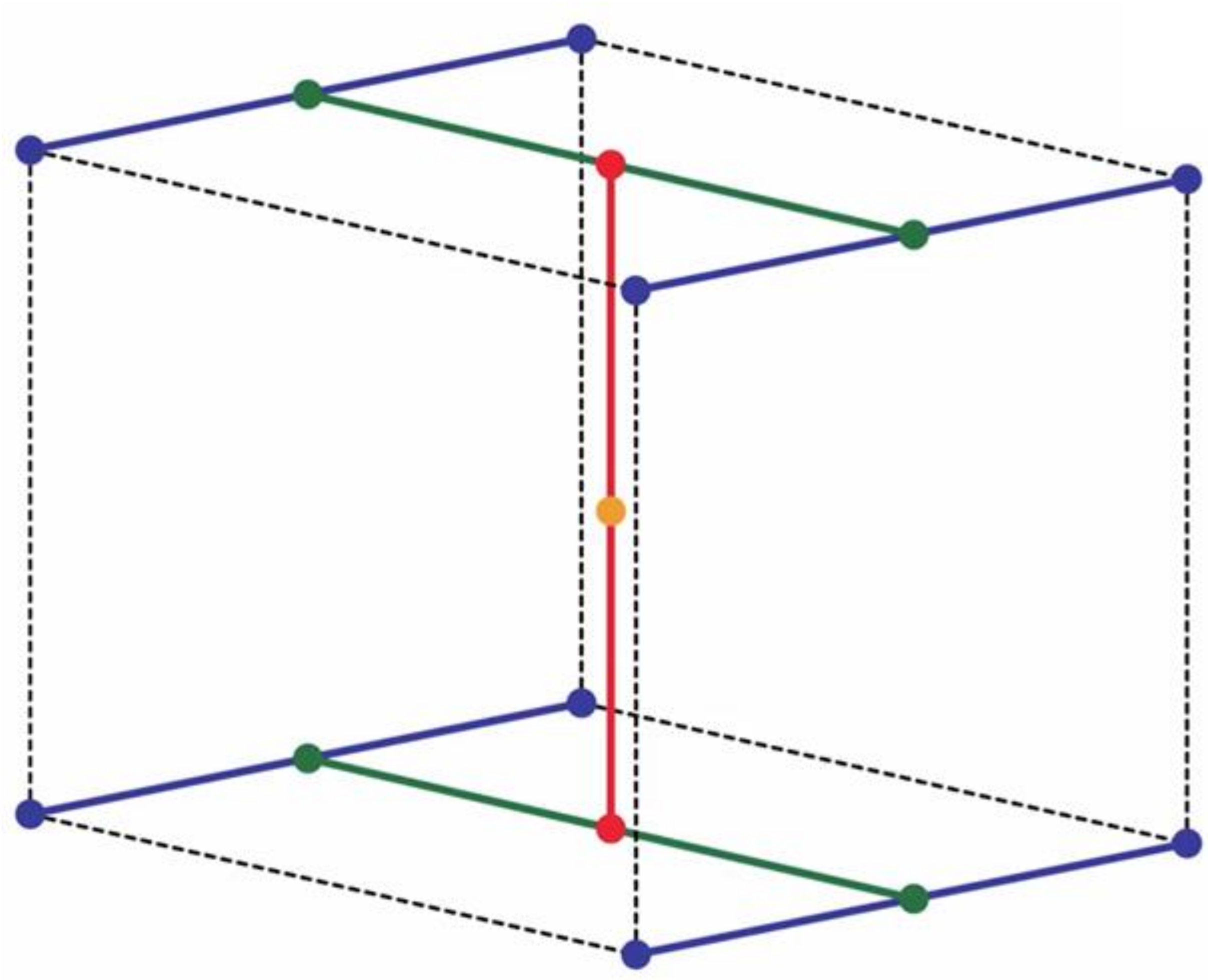
```
( 1,  1,  0); (-1,  1,  0);
( 1, -1,  0); (-1, -1,  0);
( 1,  0,  1); (-1,  0,  1);
( 1,  0, -1); (-1,  0, -1);
( 0,  1,  1); ( 0, -1,  1);
( 0,  1, -1); ( 0, -1, -1).
```

Additional Vectors (For Better Performance)

```
( 1,  1,  0); (-1,  1,  0);
( 0, -1,  1); ( 0, -1, -1).
```

fonte: http://fataho.com/

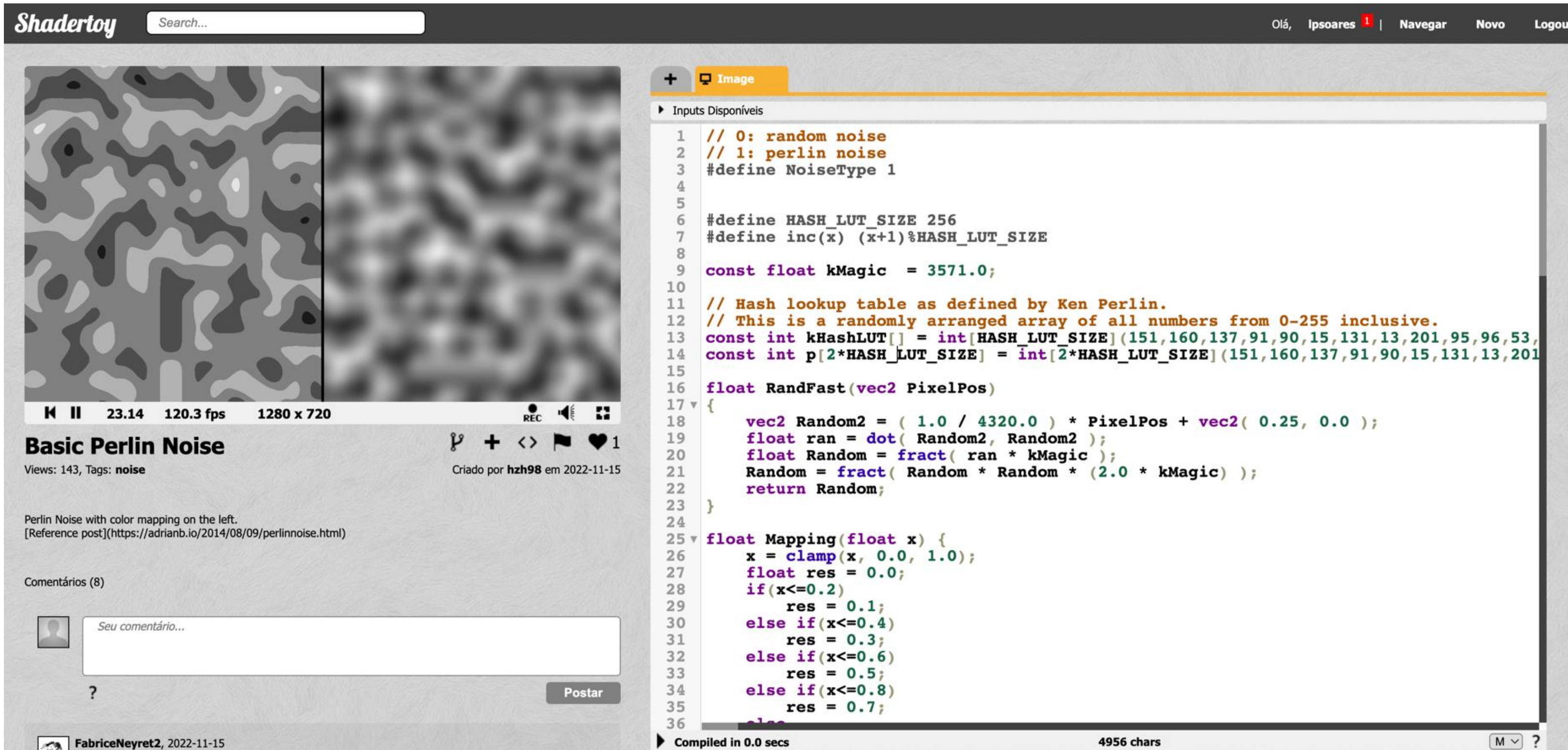
Interpolação Trilinear



fonte: http://fataho.com/

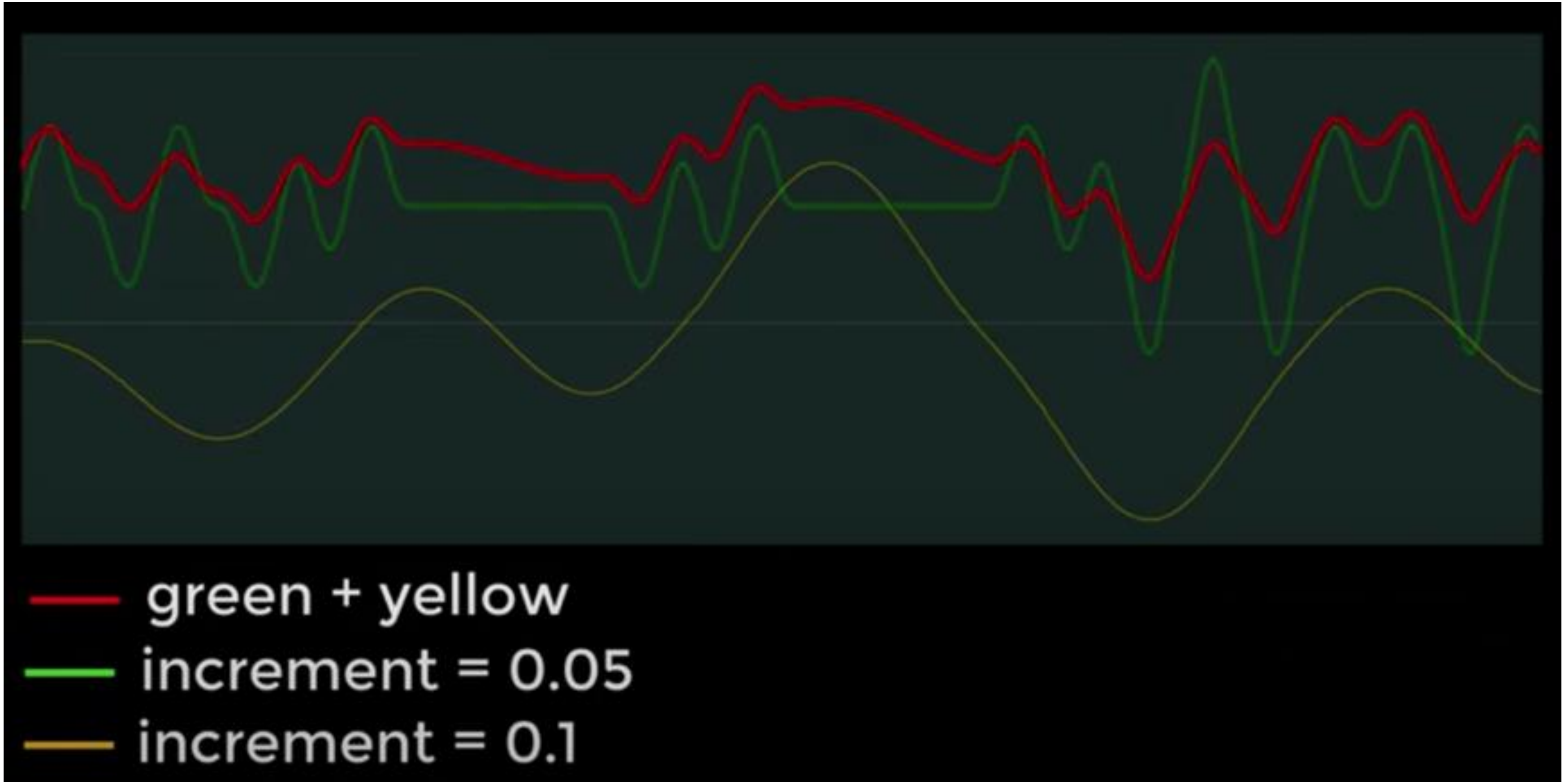
Insper

Exemplo no Shadertoy



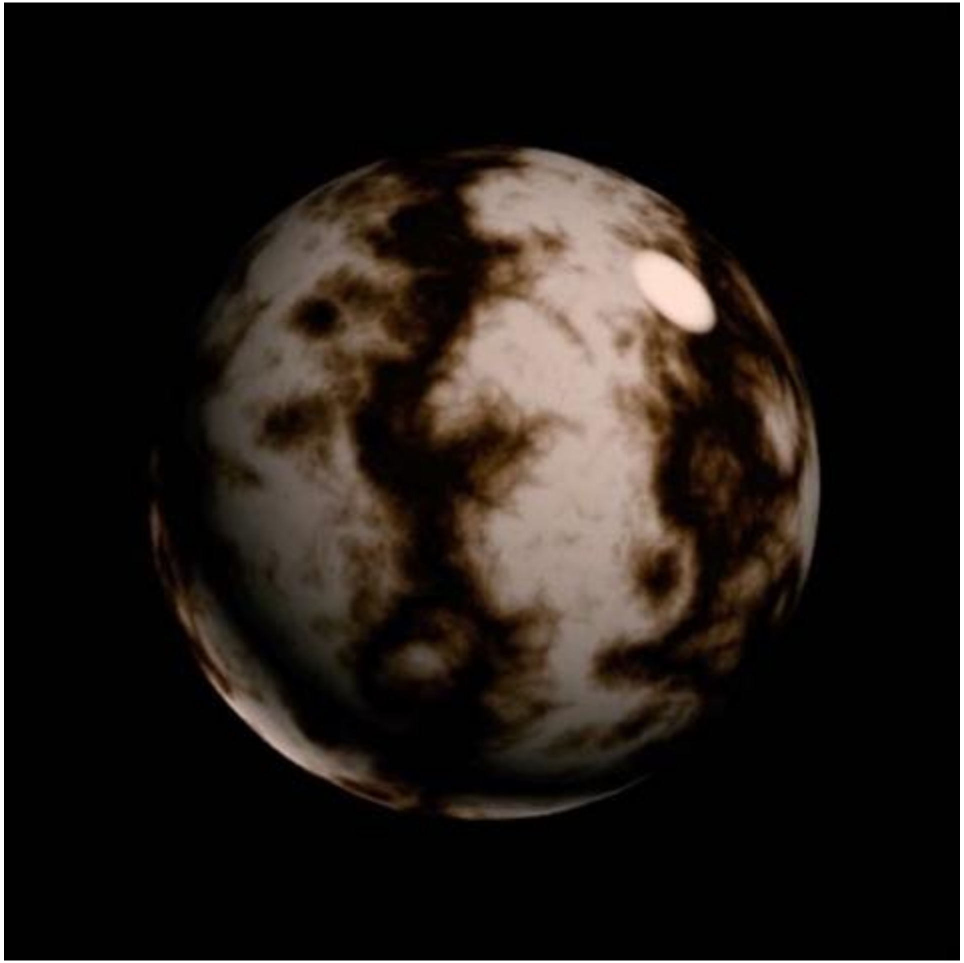
https://www.shadertoy.com/view/NlKBzm

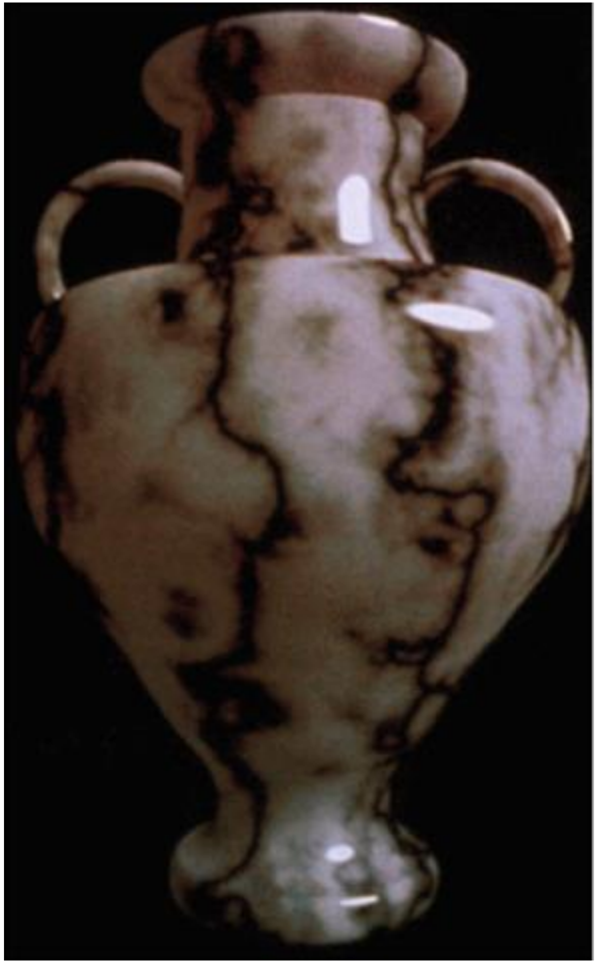
Combinando padrões (Turbulência)



fonte: https://www.youtube.com/watch?v=H6FhG9VKhJg

Ruído Procedural em 3D + Modelagem de Sólidos





Perlin noise, Ken Perlin

Live coding ruído em blocos

https://www.shadertoy.com/view/4XXfW8

Computação Gráfica

Luciano Soares
<lpsoares@insper.edu.br>

Fabio Orfali
<fabioo1@insper.edu.br>

Gustavo Braga
<gustavobb1@insper.edu.br>