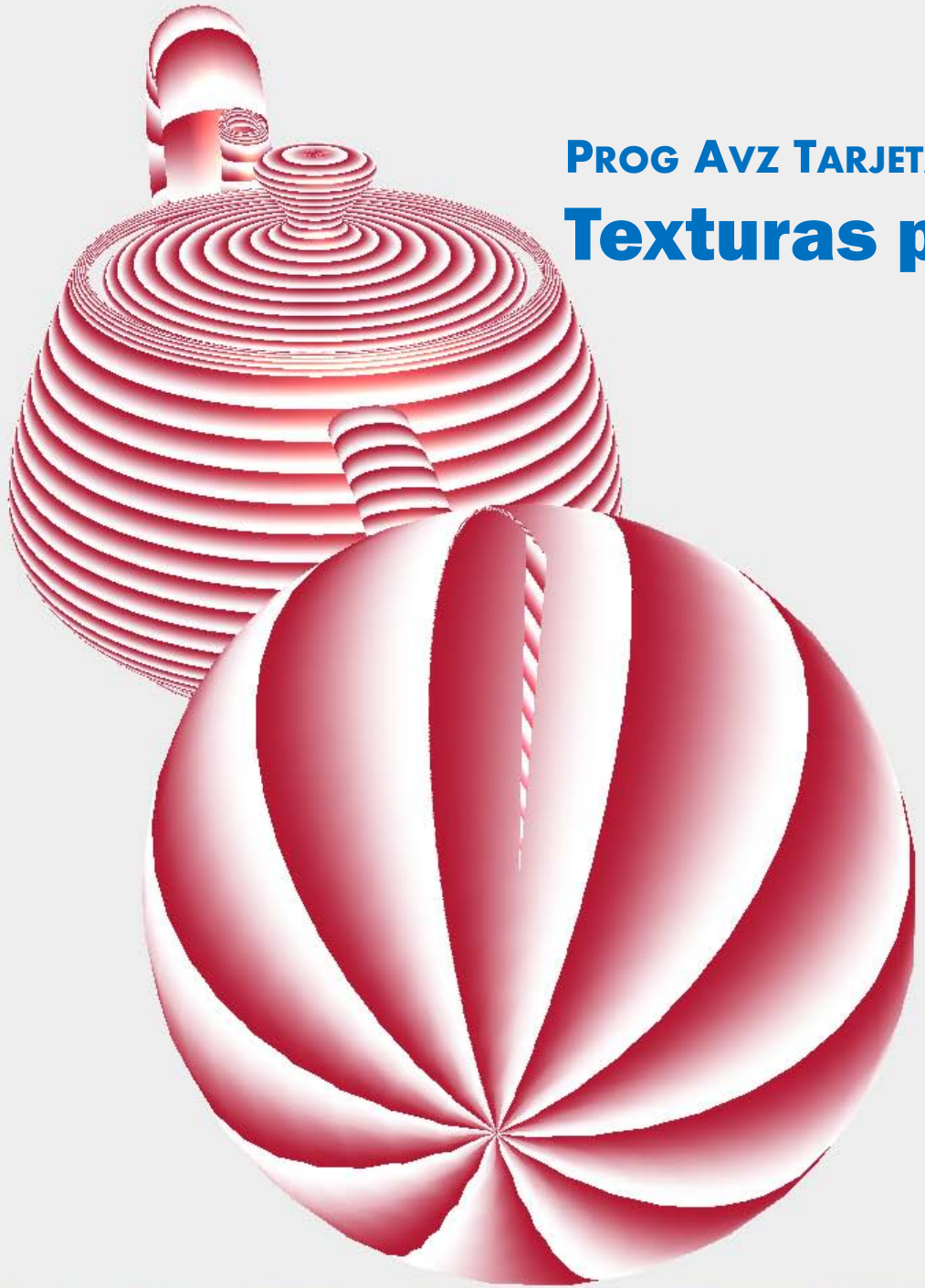


Prog Avz Tarjetas Gráficas

Texturas procedurales

Texturas procedurales. Creación.

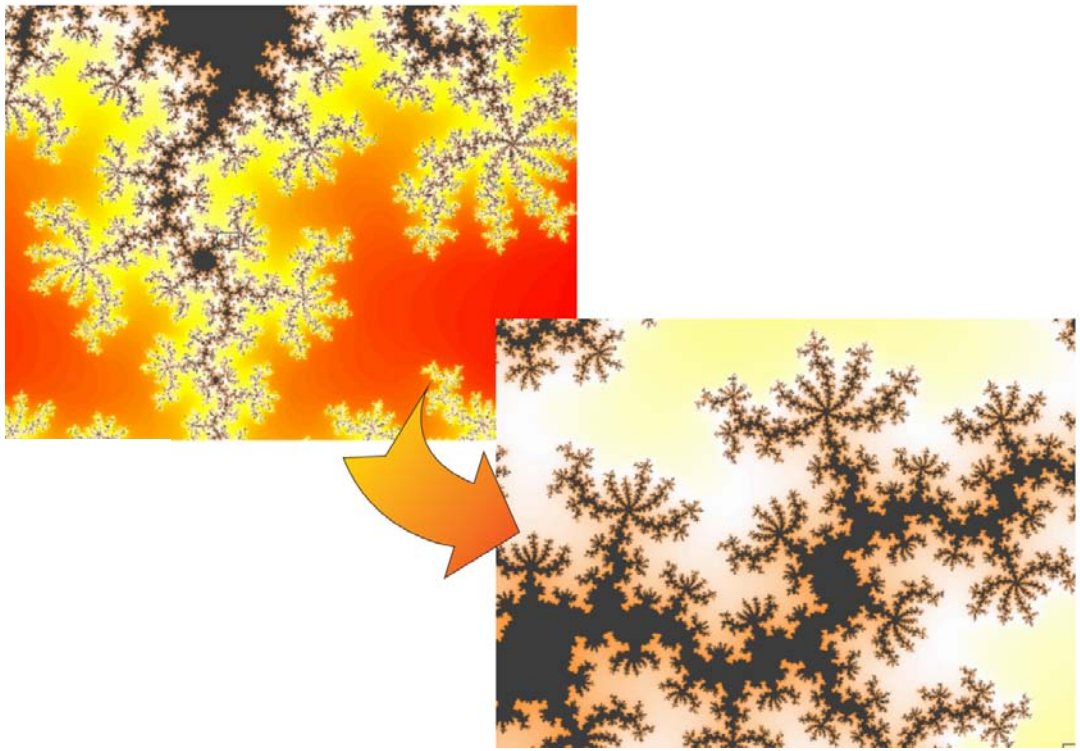
- Las texturas procedurales se generan mediante algoritmos matemáticos: programas a partir de los que se construye una imagen de la textura.
- Surgen como una forma sencilla de sintetizar texturas.

Texturas procedurales. Ventajas.

- Son densas: Las TP requieren de muy poca memoria comparada con las texturas tradicionales. Pocos kbytes versus cientos kbytes de uso del acelerador gráfico.
- Son infinitas y ocupan muy poco espacio. La única representación de la textura está en el algoritmo definido por el código en el shader.
- Escalables y ampliables: Las TP no tienen área fija o resolución. Pueden aplicarse a objetos de cualquier escala con resultados exactos a diferencia de las texturas almacenadas. No se ven los pixels al acercarlas.

Texturas paramétricas. Ventajas.

- Control paramétrico. Los shaders de la TP pueden ser escritos para parametrizar aspectos claves del algoritmo que pueden cambiarse fácilmente para producir una variedad de efectos.



Texturas procedurales. Desventajas.

- No se dibujan. No todos tienen la habilidad y técnicas de programación y codificación de algoritmos.
- Debugging difícil: patrones implícitos no triviales.
- Alto coste computacional que la GPU absorbe fácilmente, pero se precisa optimizar el código.
- Tratamiento de aliasing. No tenemos el “dibujo” de la señal 2D “premuestreado” para filtrarlo a una resolución adecuada al trozo de pantalla correspondiente.

Cuál escoger?

- Usar un shader procedural o una textura de imagen es una decisión pragmática.
- Objetos que son muy importantes en el acabado final (caras de personajes) pueden ser renderizadas con texturas tradicionales.
- Cosas triviales que pueden cubrir grandes áreas son candidatas a usar TP (paredes, piso, terreno, agua, etc.)
- Una textura híbrida puede ser una correcta solución.

Ejemplo

- Textura generada por fragment shader
- El vertex shader recibe las coordenadas de textura
- Estas coordenadas son usadas en el fragment shader.

Vertex shader

```
void main(){
   vTexCoord = aTexCoord;
   gl_Position = uModelViewProjMatrix *
   aPosition;
 }
```

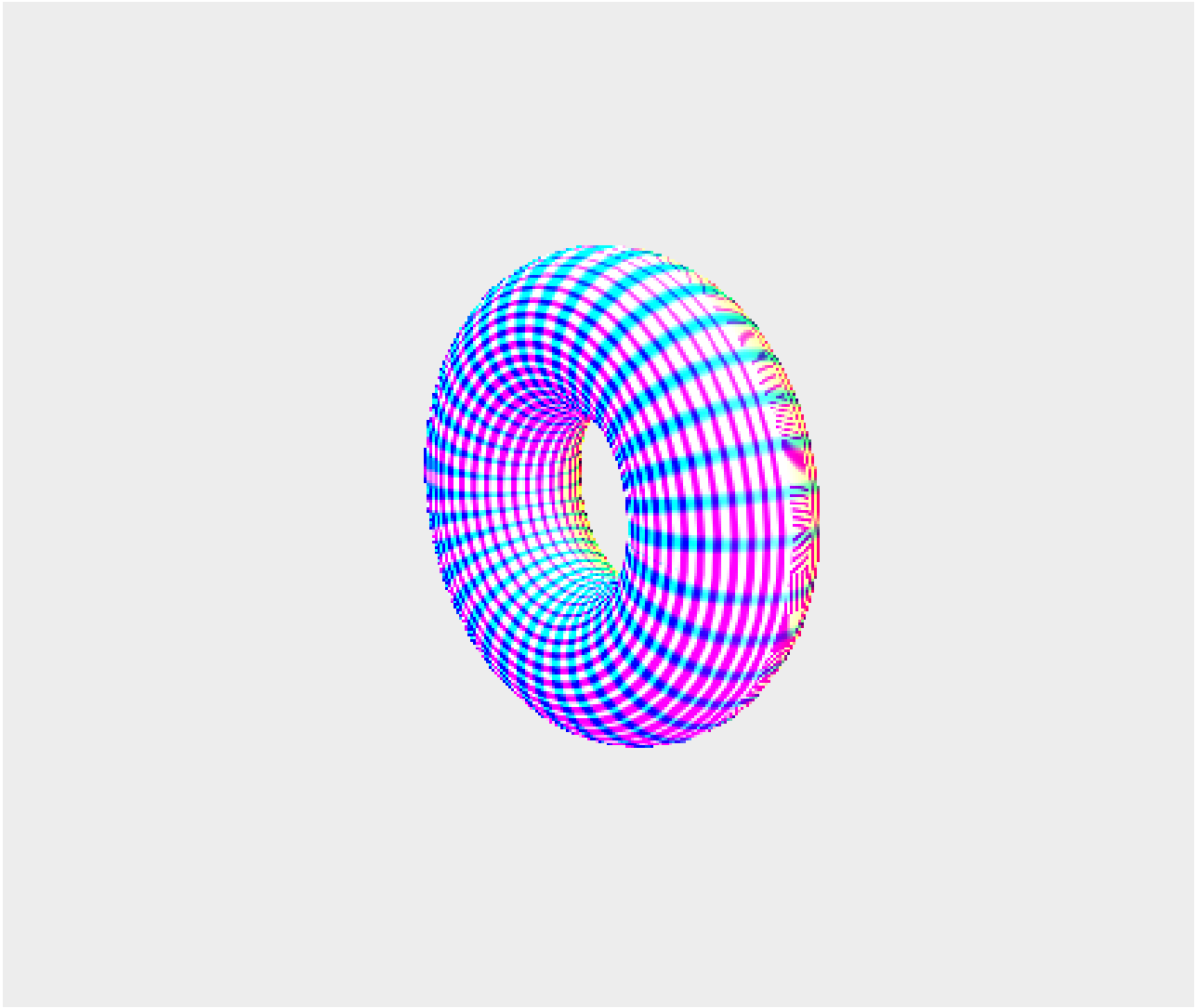
Ejemplo

- El fragment shader es una función de las coordenadas de textura.

Fragment shader

```
void main(){
  fFragColor = vec4(1.0,1.0,0.0,1.0);
  float a = sin(vTexCoord.s*30)/2 + .5;
  float b = sin(vTexCoord.t*30)/2 + .5;
  fFragColor = vec4(a,b,1.0,0.0);
}
```

Resultado



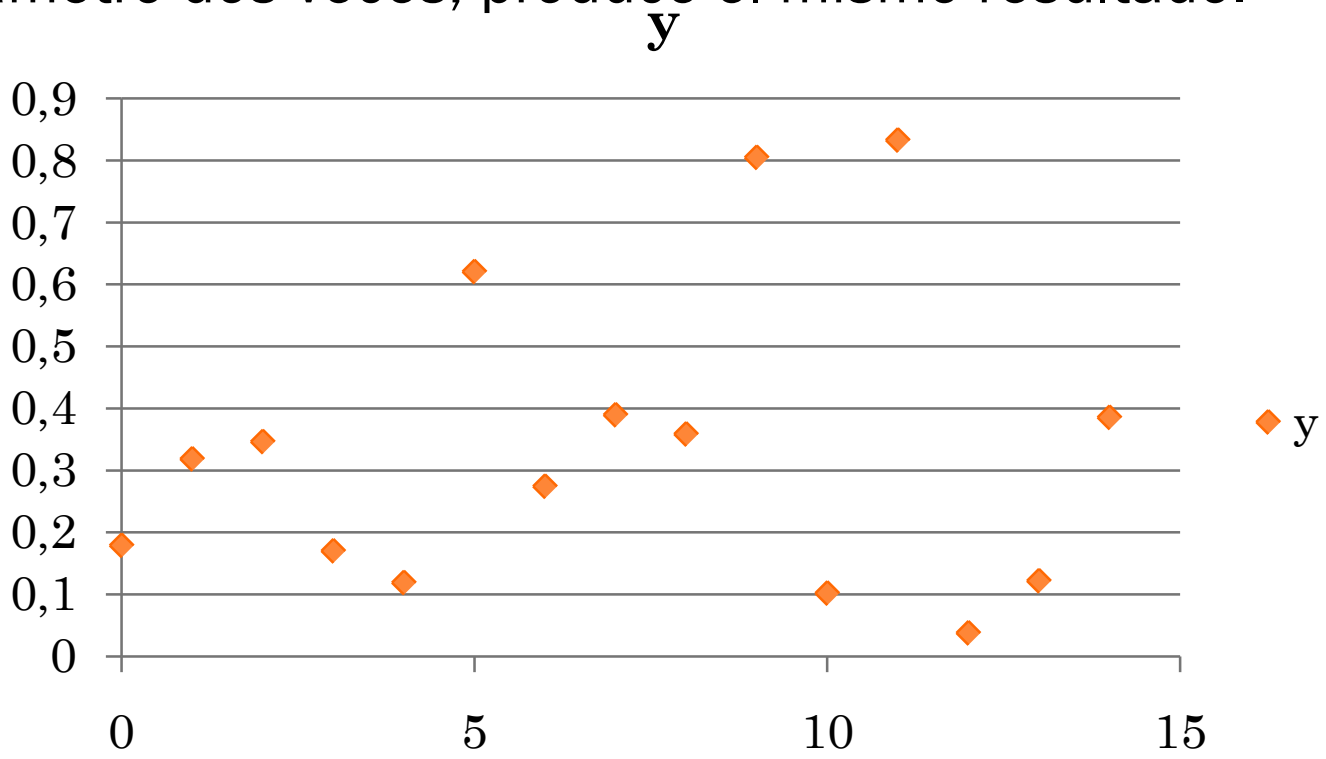
Perlin noise.

- El ruido es importante para añadir un aspecto natural a las texturas procedurales.
- Valor repetible y seudo aleatorio para cada valor de entrada.
- Rango definido entre -1 y 1.
- Ancho de banda limitado.
- No aparecen patrones que se repiten.

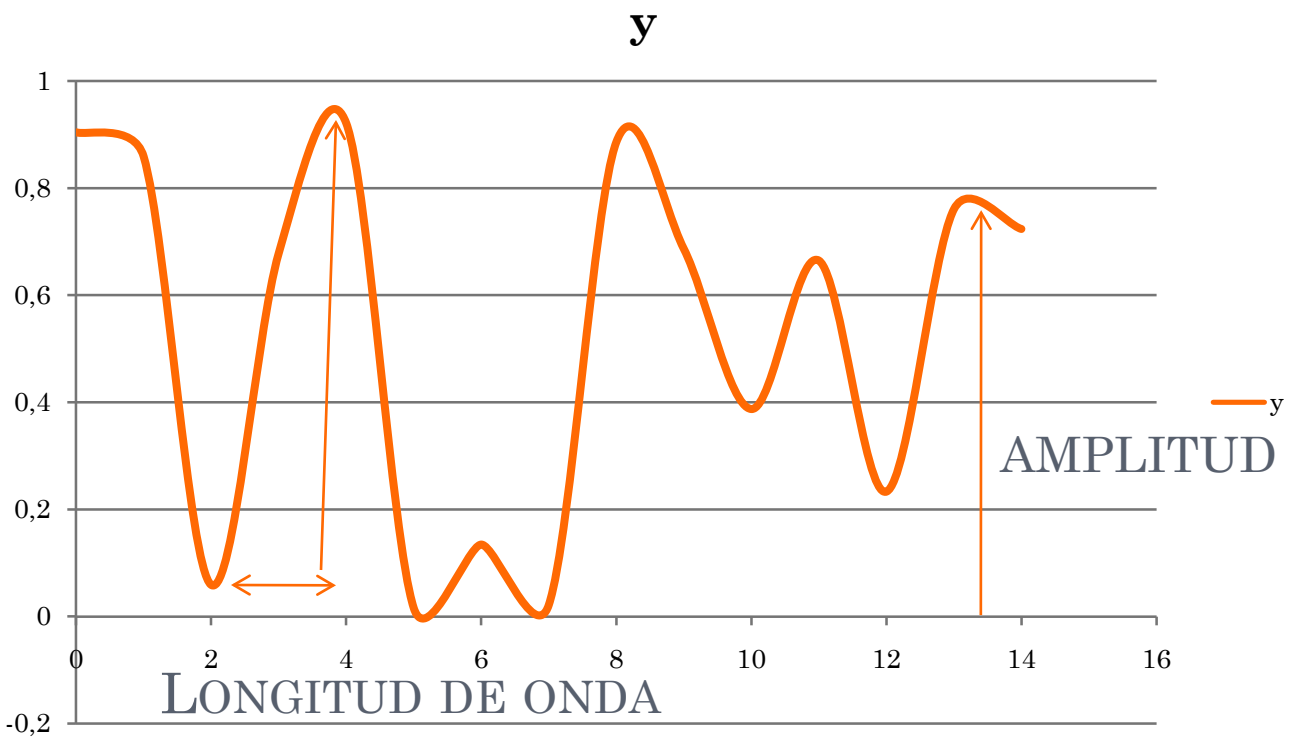
- Para crear una función Perlin noise, necesitamos:
  - Función Noise
  - Función de Interpolación

Función noise

- Es esencialmente generador de números aleatorios.
- Toma un número como parámetro y devuelve un número aleatorio basado en ese parámetro. Si se pasa el mismo parámetro dos veces, produce el mismo resultado.

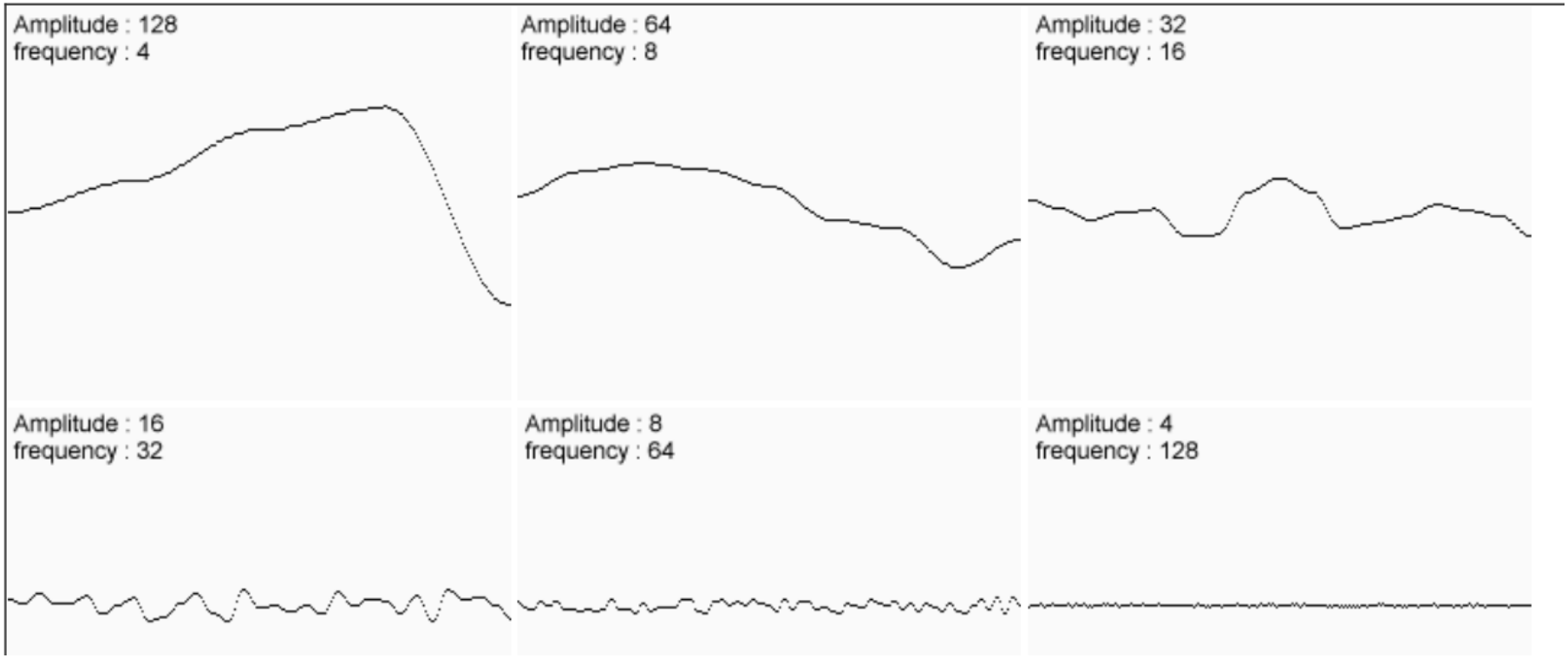


FUNCTION NOISE



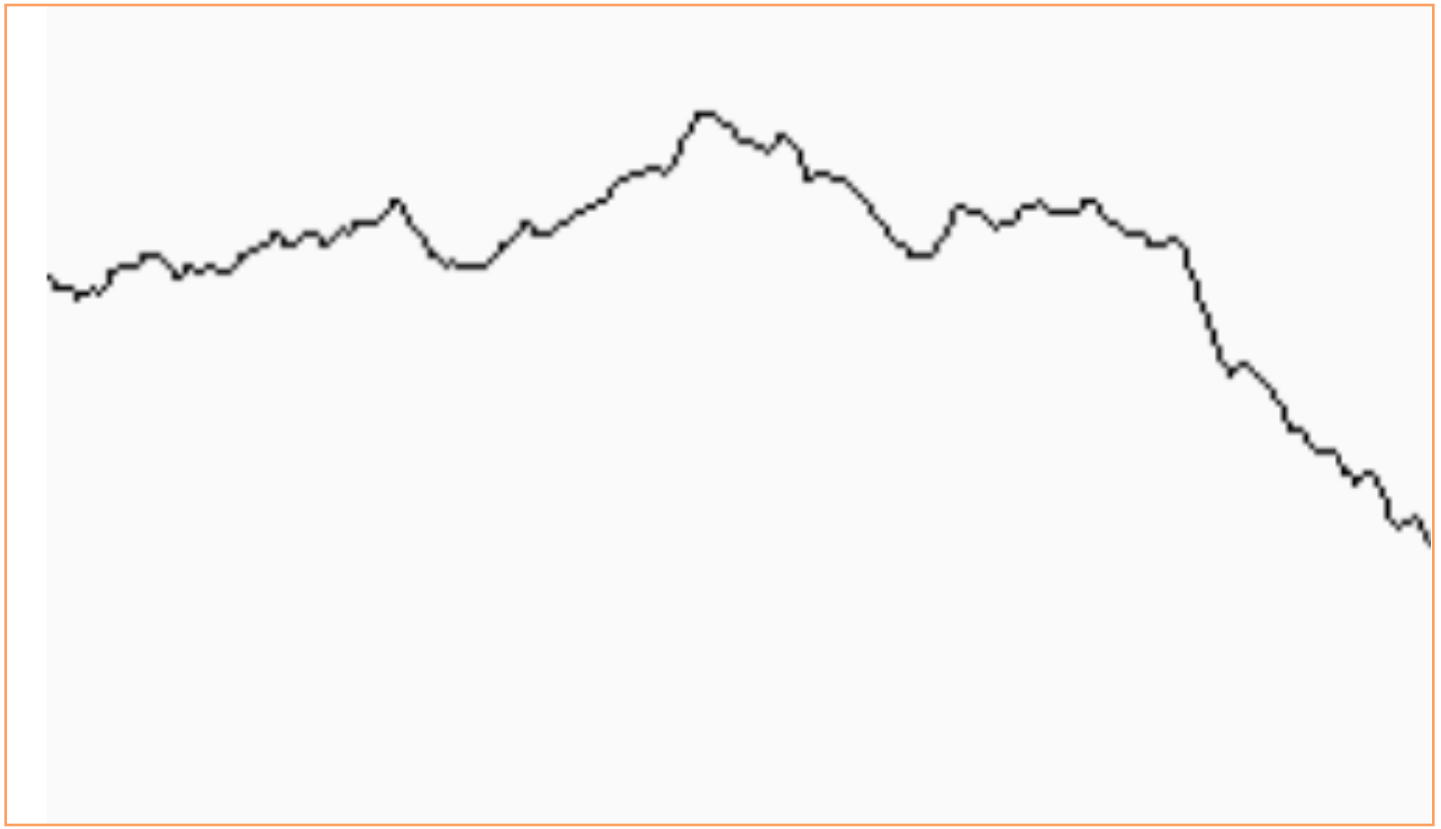
- La función anterior con interpolación suavizada.
- Podemos definir una función continua que toma un float como parámetro.

Creación de una función noise



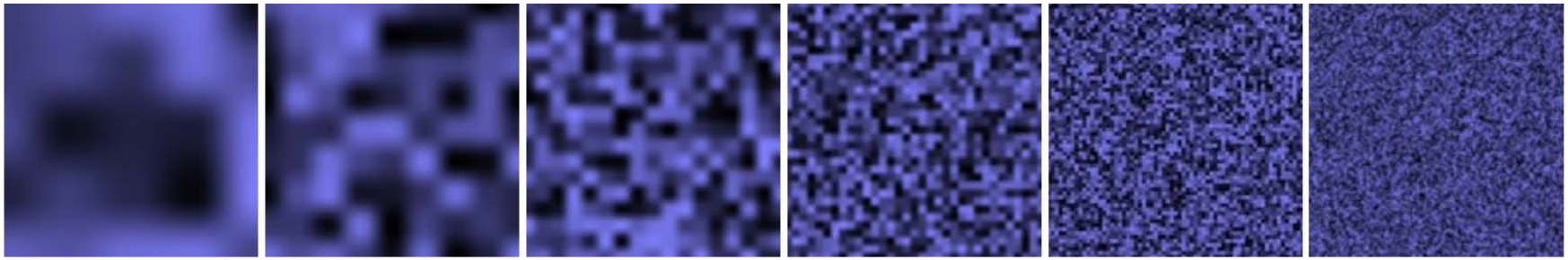
Si sumamos todas estas funciones, obtenemos:

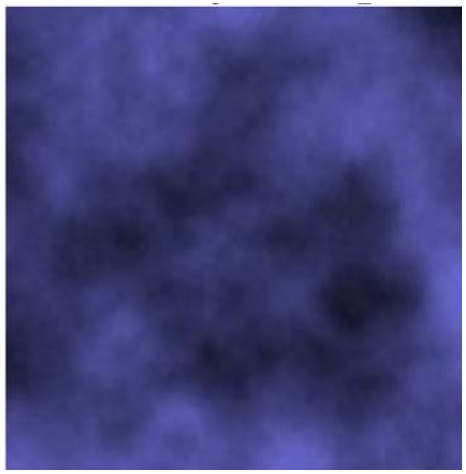
Perlin noise = suma de funciones noise



- Función con grandes, medianas y pequeñas variaciones. Muchos terrenos generados por computadora se obtienen usando este método.

Funciones noise en 2D





- La sumatoria de estas funciones producen una textura ruidosa.

Aplicaciones:

- Renderizar fenómenos naturales:
  nubes, fuego, humo, efectos del viento, etc.
- Materiales: granito, madera, montañas.
- Otros materiales: asfalto, cemento, stucco
- Para agregar imperfecciones a los modelos:
  suciedad, dientes, arrugas.
- Agregar imperfecciones al movimiento: jitters, bumps.
- Y más…

2D Perlin noise pseucodigo

```
function Noise1(integer x, integer y)
  n = x + y * 57
  n = (n<<13) ^ n;
  return ( 1.0 - ( (n * (n * n * 15731 + 789221) +
  1376312589) & 7fffffff) / 1073741824.0);
end function


function SmoothedNoise1(float x, float y)
  corners = ( Noise(x-1, y-1)+Noise(x+1, y-1)+Noise(x-
  1, y+1)+Noise(x+1, y+1) ) / 16
  sides = ( Noise(x-1, y) +Noise(x+1, y) +Noise(x, y-1)
  +Noise(x, y+1) ) / 8
  center = Noise(x, y) / 4
  return corners + sides + center
end function
```

Pseudocodigo

```
function InterpolatedNoise_1(float x, float y)
  integer_X = int(x)
  fractional_X = x - integer_X integer_Y = int(y)
  fractional_Y = y - integer_Y
  v1 = SmoothedNoise1(integer_X, integer_Y)
  v2 = SmoothedNoise1(integer_X + 1, integer_Y)
  v3 = SmoothedNoise1(integer_X, integer_Y + 1)
  v4 = SmoothedNoise1(integer_X + 1, integer_Y + 1)

  i1 = Interpolate(v1 , v2 , fractional_X)
  i2 = Interpolate(v3 , v4 , fractional_X)

  return Interpolate(i1 , i2 , fractional_Y)
end function
```

Pseudocodigo

```
function PerlinNoise_2D(float x, float y)
  total = 0 p = persistence
  n = Number_Of_Octaves - 1
  loop i from 0 to n
       frequency = 2^i
       amplitude = p^i
       total = total + InterpolatedNoise_i(x * frequency, y * frequency) * amplitude
  end of i loop
  return total
end function
```

Gems imp

```
float3 fade(float3 t){
  return t*t*t*(t*(t*6-15)+10);
}
float perm(float x){
  return tex1D(permSampler, x / 256.0) * 256;
}
float grad(float x, float 3){
  return dot(tex1d(gradSampler, x), p);
}
//3D version
Float inoise(float3 p){
  float3 P = fmod(floor(p), 256.0);
  p -= floor(p);
  float3 f = fade(p);
```

```
//Hash coordinates for 6 of the 8 cube corners
  float A = perm(P.x) + P.y;
  float AA = perm(A) + P.z;
  float AB = perm(A +1) + P.z;
  float B = perm(P.x + 1) + P.y;
  float BA = perm(B) + P.z;
  float BB = perm(B + 1) + P.z;

//And add blended results from 8 corners of cube
return lerp(
  lerp(lerp(grad(perm(AA), p),
            grad(perm(BA), p + float3(-1,0,0), f.x),
       lerp(grad(permAB), p + float3(0,-1.0)),
            grad(perm(BB), p + float3(-1,-1,0)), f.x), f.y),
  lerp(lerp(grad(perm(AA+1), p + float3(0,0,-1)),
            grad(perm(BA+1), p + float3(-1,0,-1)), f.x),
       lerp(grad(perm(AB+1), p + float3(0,-1,-1)),
            grad(perm(BB+1), p + float3(-1,-1,-1)), f.x), f.z);
}
```


Programa Ejemplo
ES
19H36

Bibliografía.

- Rost, Randi J. “OpenGL®Shading Language”, Second Edition, Addison Wesley Professional, 2006. ISBN-13 978-0-321-33489-3 ☺
- GPU Gems 2.