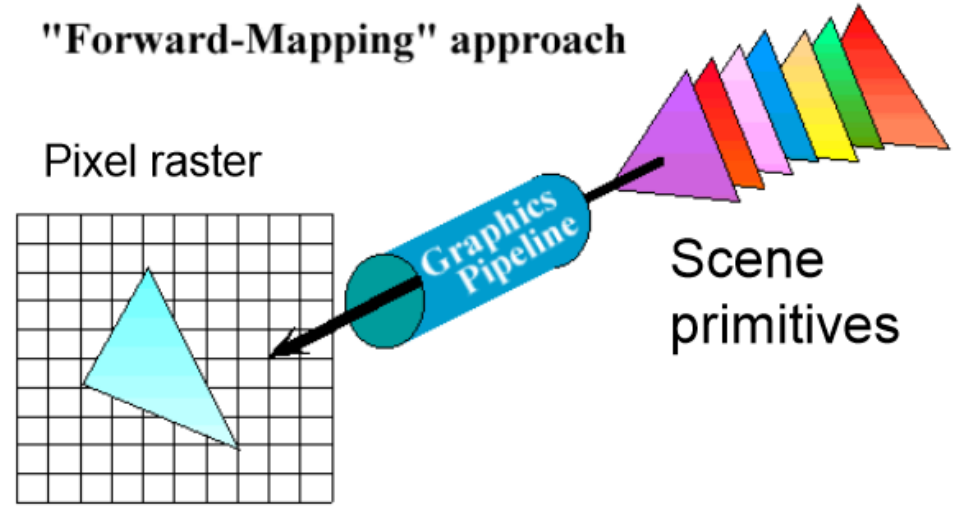
Advanced image synthesis

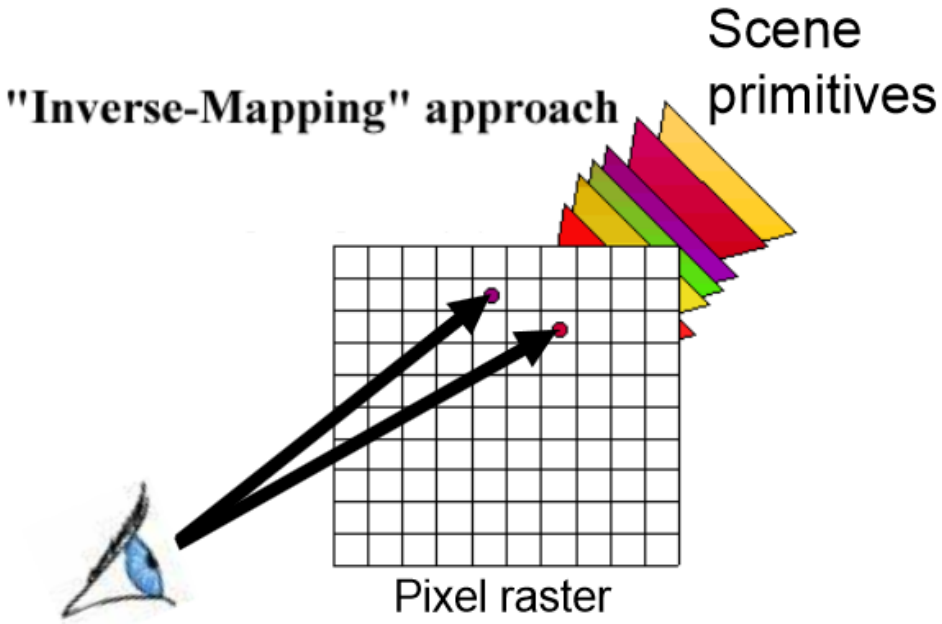
Romain Vergne – 2014/2015

Rasterization VS ray-casting

- For each triangle
  - Project triangle to image plane
  - For each pixel
    - Check pixel in triangle
    - Resolve visibility with z-buffer

- For each pixel
  - Compute pixel ray
  - For each triangle
    - Check ray-triangle intersection
    - Get closest intersection

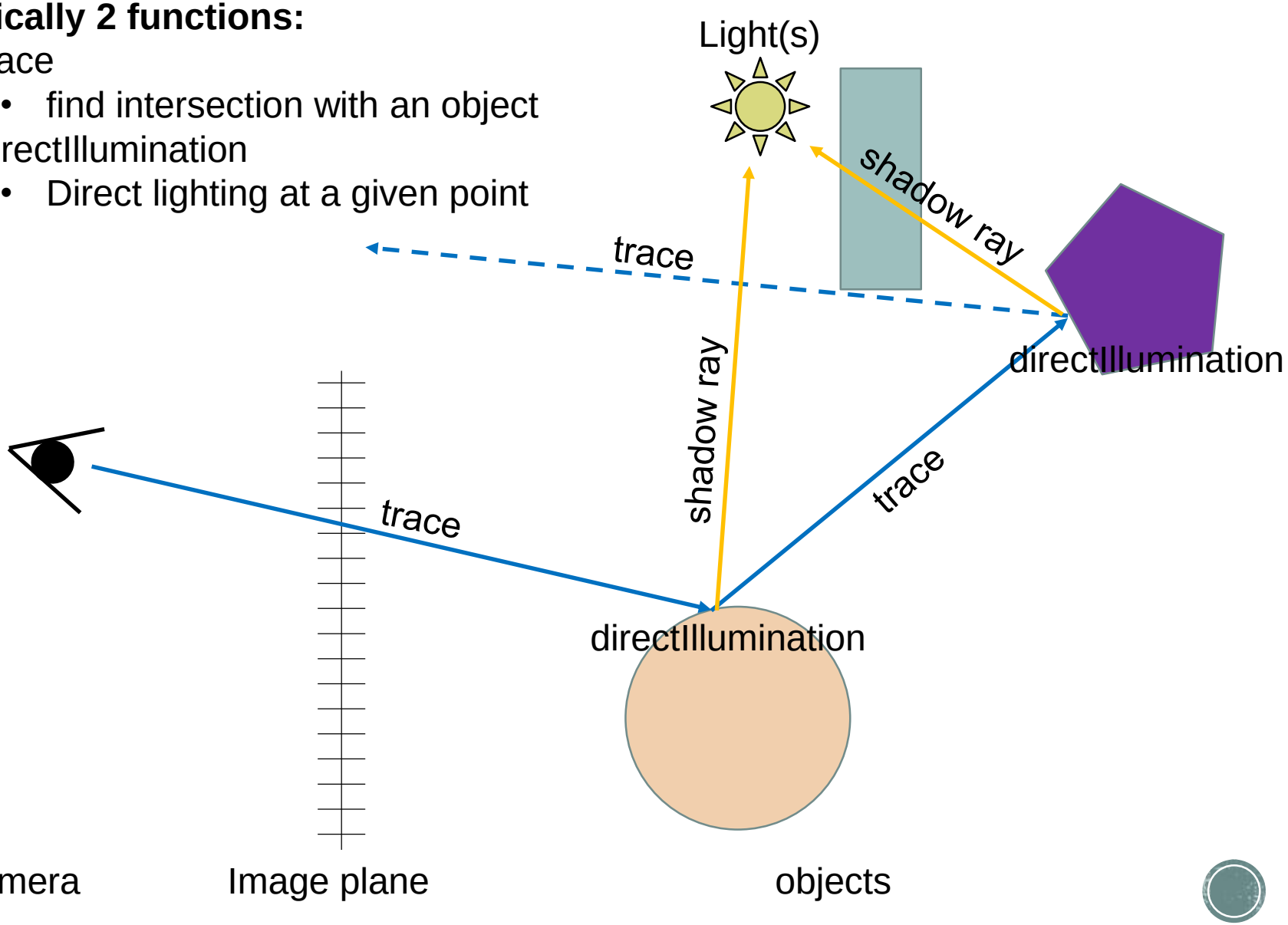




Ray tracing

Basically 2 functions:

- trace
  - find intersection with an object
- directIllumination
  - Direct lighting at a given point



Ray tracing

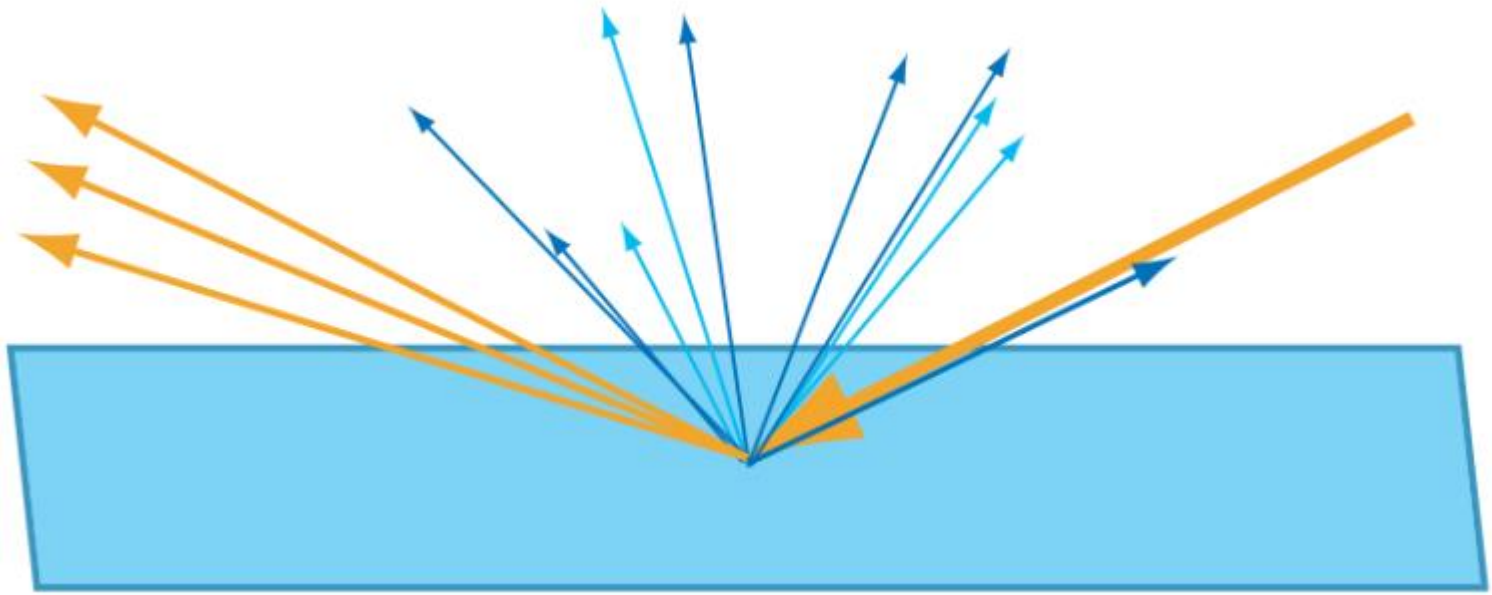
- color trace(ray) {
  - hit = intersectScene(ray)
  - if(hit) {
    - color = directIllumination(hit)
    - if hit is reflective
      - color += c_refl * trace(reflected ray)
    - if hit is transmissive
      - color += c_trans * trace(refracted ray)
  - } else
    - color = background_color
  - return color
- }

Ray tracing

- color directIllumination(hit) {
  - color = (0,0,0)
  - for each light L {
    - T = cast shadow ray to L
    - if hit is not shadowed by L
      - color += Ambient+diffuse+specular terms(L,hit)
  - }
  - return color
- }

Physics → Maths

- Radiance
  - radiometric quantity used mesure the amount of light along a single ray
  - Spectral quantity (RGB in practice), Watt per steradian per square meter
- If shading can be handled locally, light response depends on
  - Light direction
  - View direction



Physics → Maths

- **B**idirectionnal
- **R**eflectance
- **D**istribution
- **F**unction

$$f(\mathbf{l}, \mathbf{v})$$

$$L_o(\mathbf{v}) = \int_\Omega f(\mathbf{l}, \mathbf{v}) \otimes L_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l}) d\omega_i$$

Outgoing radiance

BRDF

Ingoing radiance

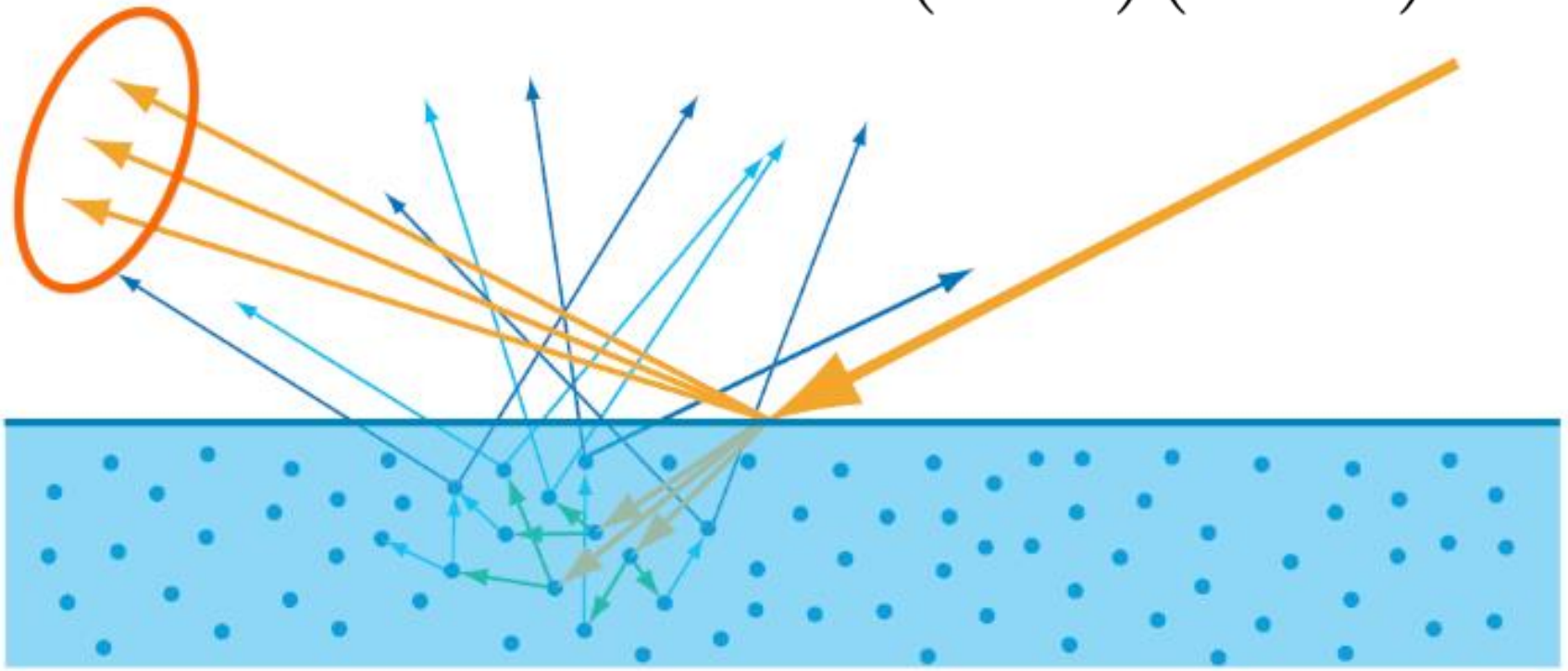
Surface orientation

Reflectance equation

Microfacet theory

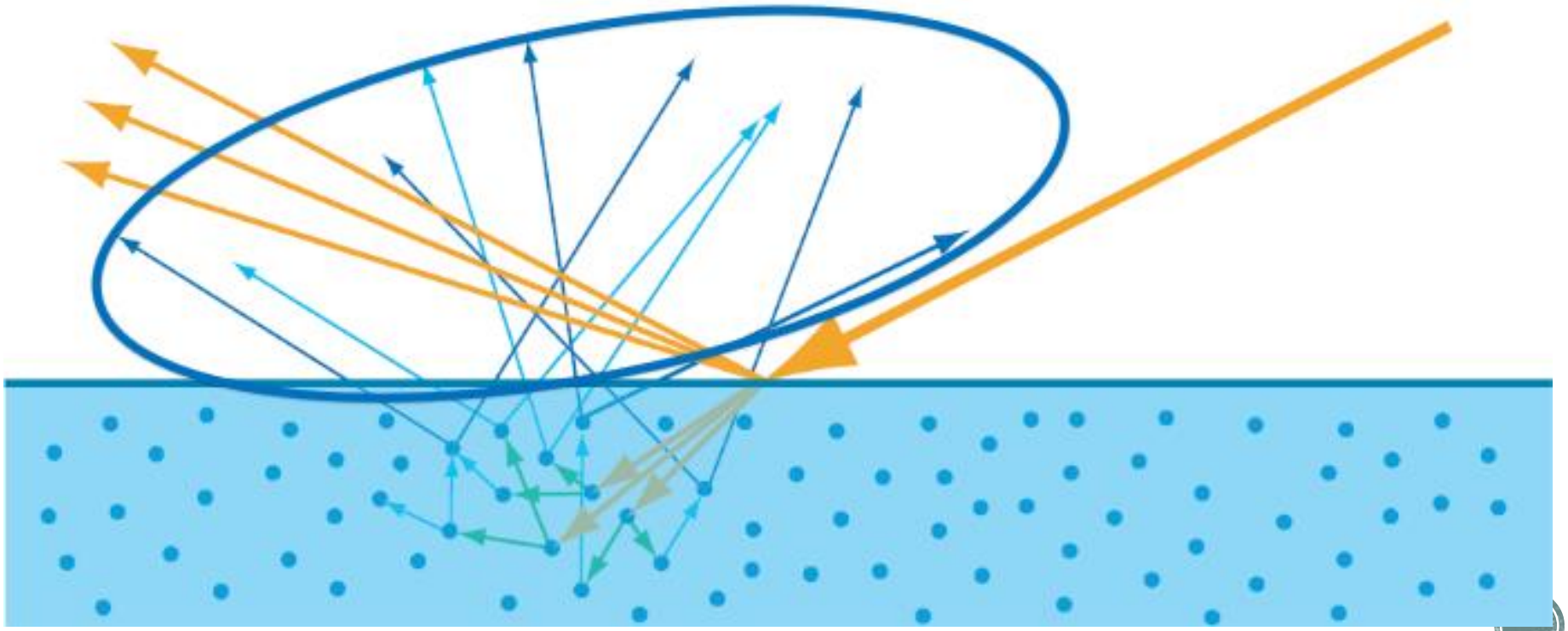
- Surface reflection (specular term)

$$f(\mathbf{l},\mathbf{v}) = \frac{F(\mathbf{l},\mathbf{h})G(\mathbf{l},\mathbf{v},\mathbf{h})D(\mathbf{h})}{4(\mathbf{n}\cdot\mathbf{l})(\mathbf{n}\cdot\mathbf{v})}$$



Microfacet theory

- Subsurface reflection (diffuse term)
  - Constant: $f_{\text{Lambert}}(\mathbf{l}, \mathbf{v}) = \frac{\mathbf{c}_{\text{diff}}}{\pi}$



Spatial variation

- All materials seen so far are the same everywhere…

Spatial variation

- All materials seen so far are the same everywhere…
  - No real reason to make this assumption



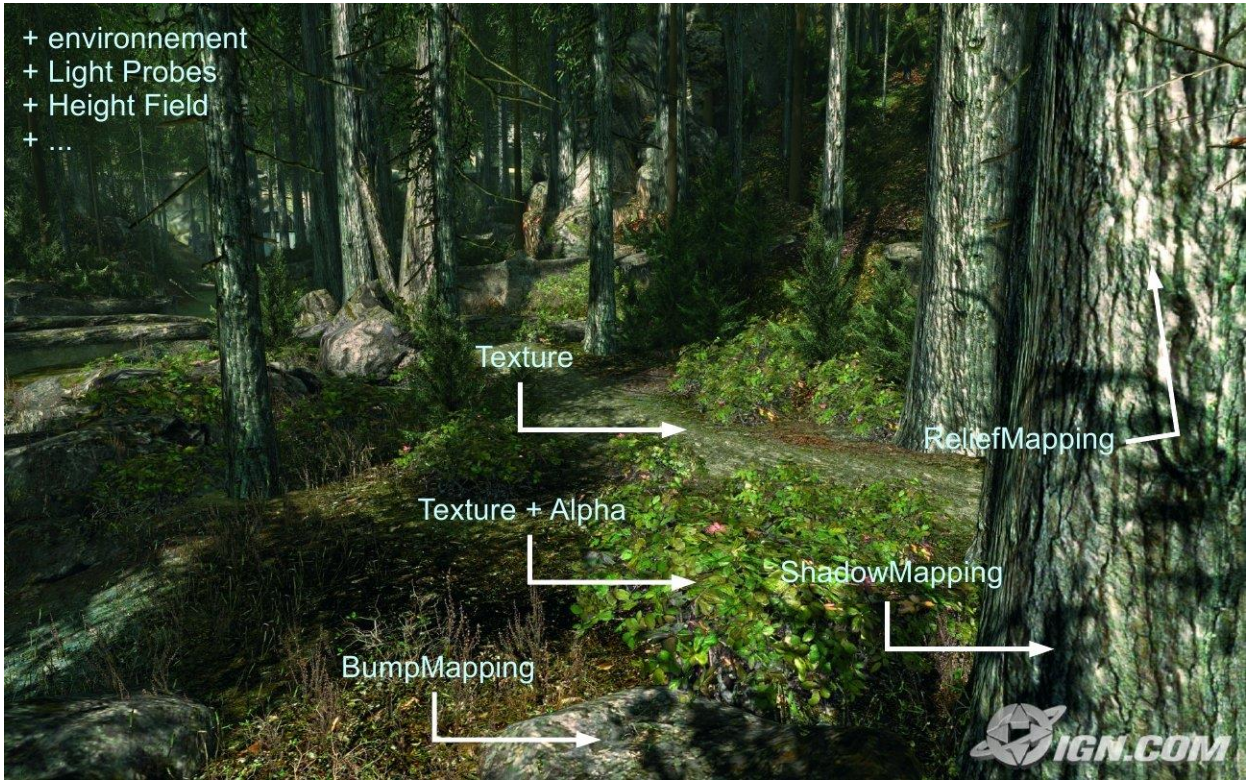



Spatial variation

- Allow BRDF parameters to vary over space
  - Diffuse / specular colors, roughness, normal

Spatial variation

- Textures are used for representing all elements
  - Material (BRDFs, BTFs)
  - Geometry (normal / bump / displacement maps)
  - Lighting (light / environment maps)

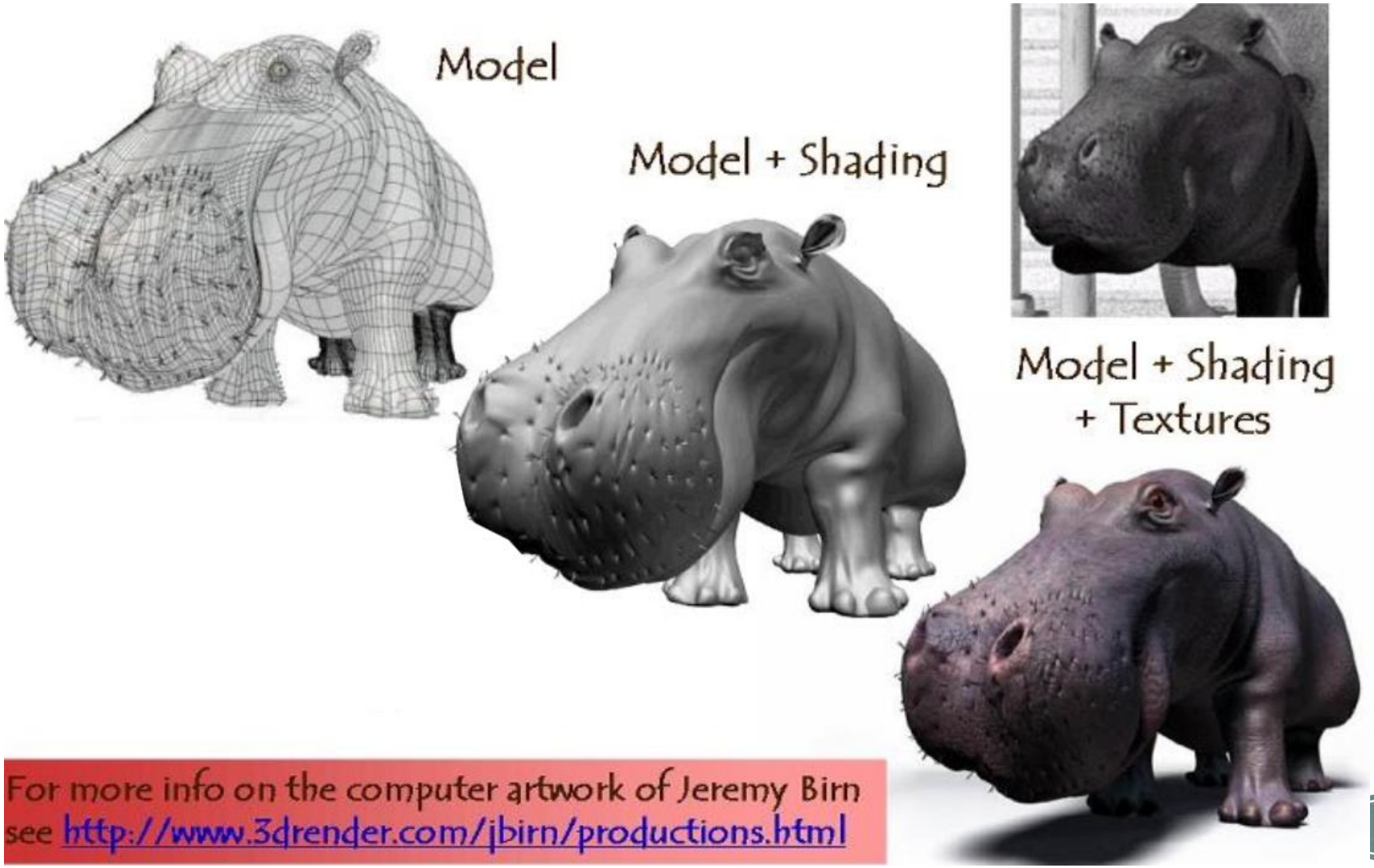


The bad idea

- Store BRDF properties on each vertex
  - Refine / tesselate if necessary
- On a given point, interpolate properties (barycentric coords)
- But:
  - Details mean (much) more geometry/memory
  - Only works on meshes

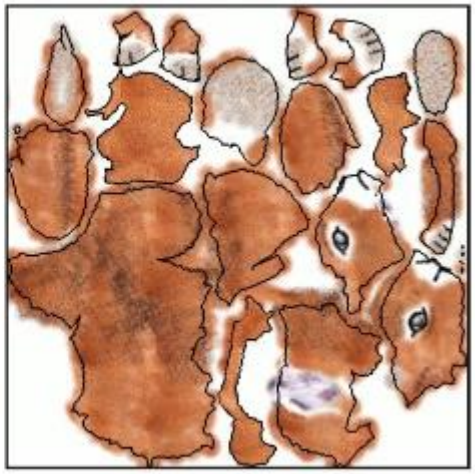
The good idea

- Use textures



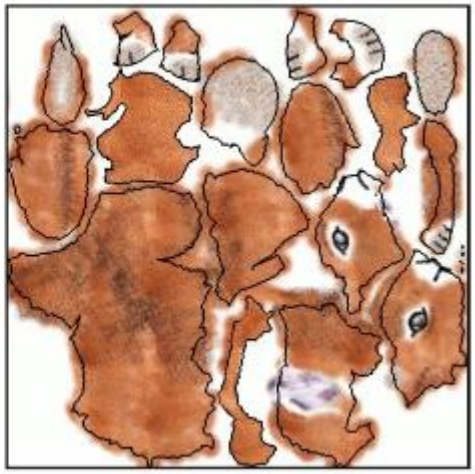
2 approaches

- From data
  - Colors, coefs, normals stored in 2D images

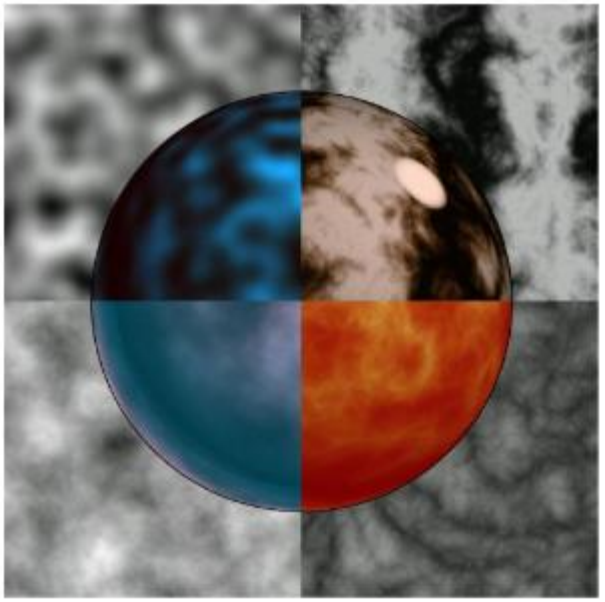


2 approaches

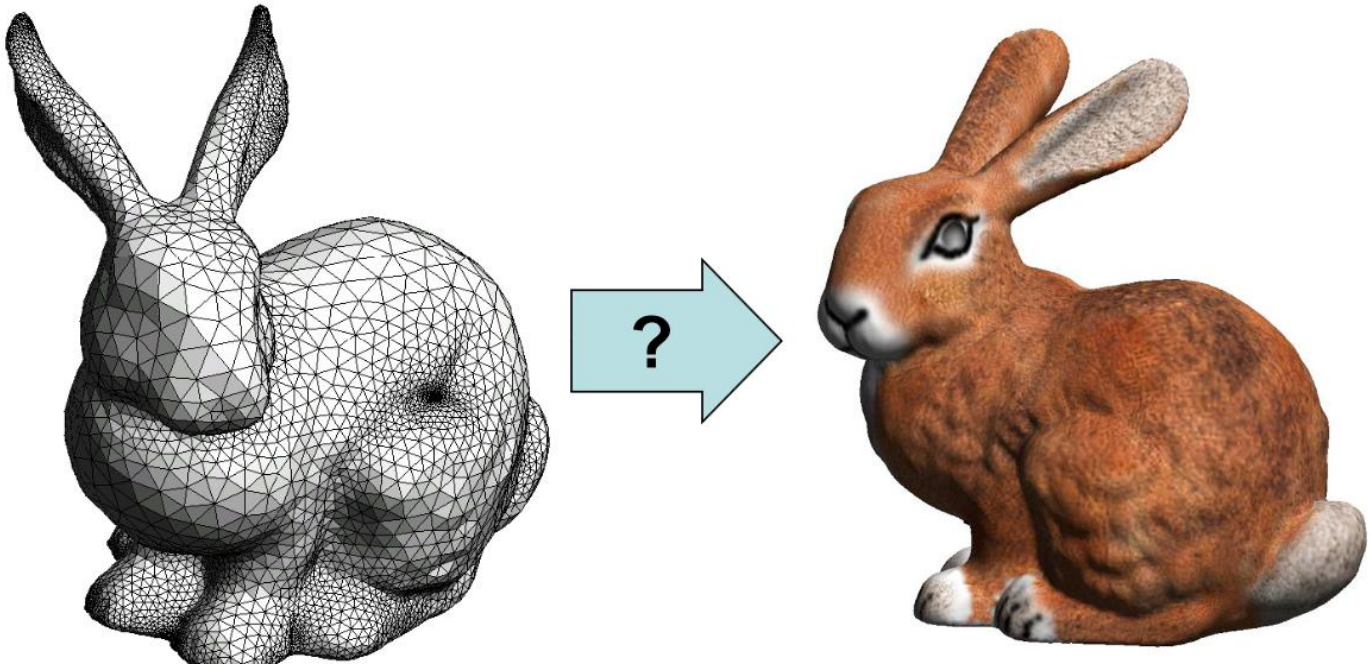
- From data
  - Colors, coefs, normals stored in 2D images



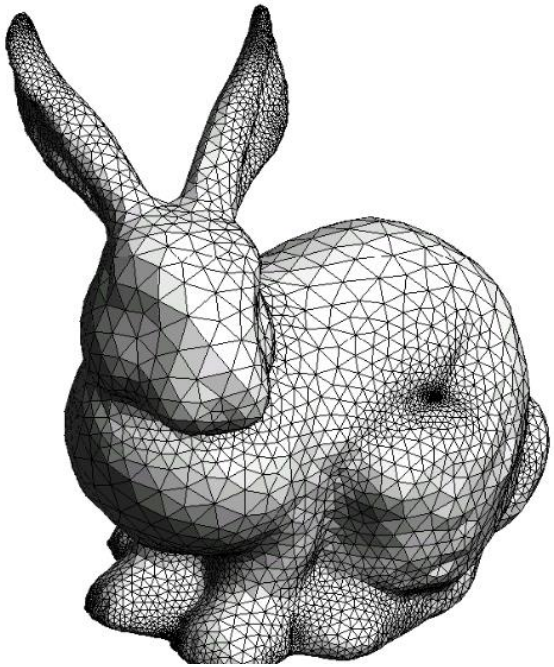
- Procedural shader
  - Little program that compute info at a given position



Texture mapping

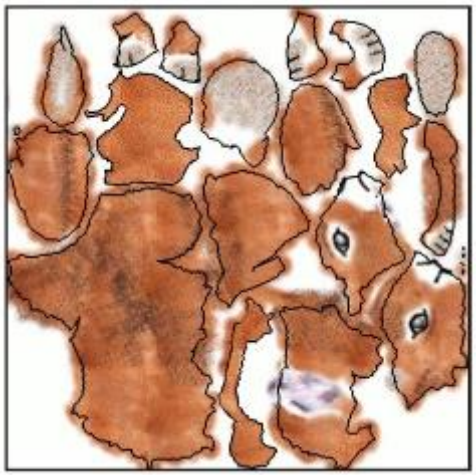


Texture mapping



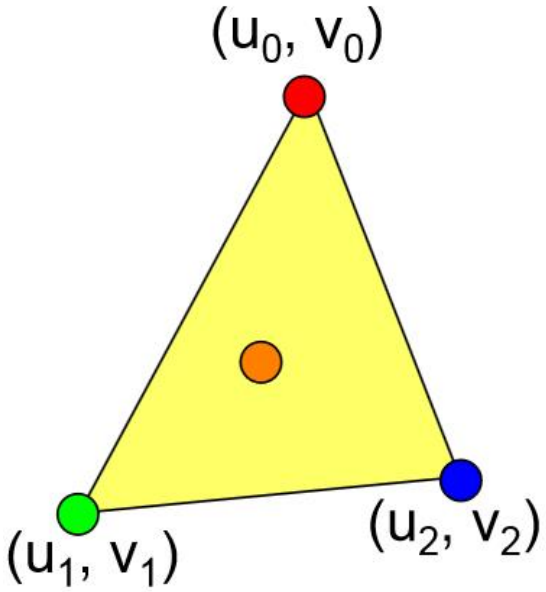
?

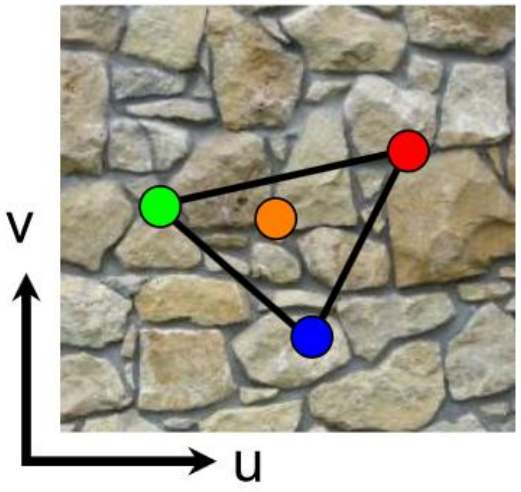
R3→R2



UV coordinates

- Store uv coords on each vertex of the input mesh
- Interpolate using barycentric coordinates





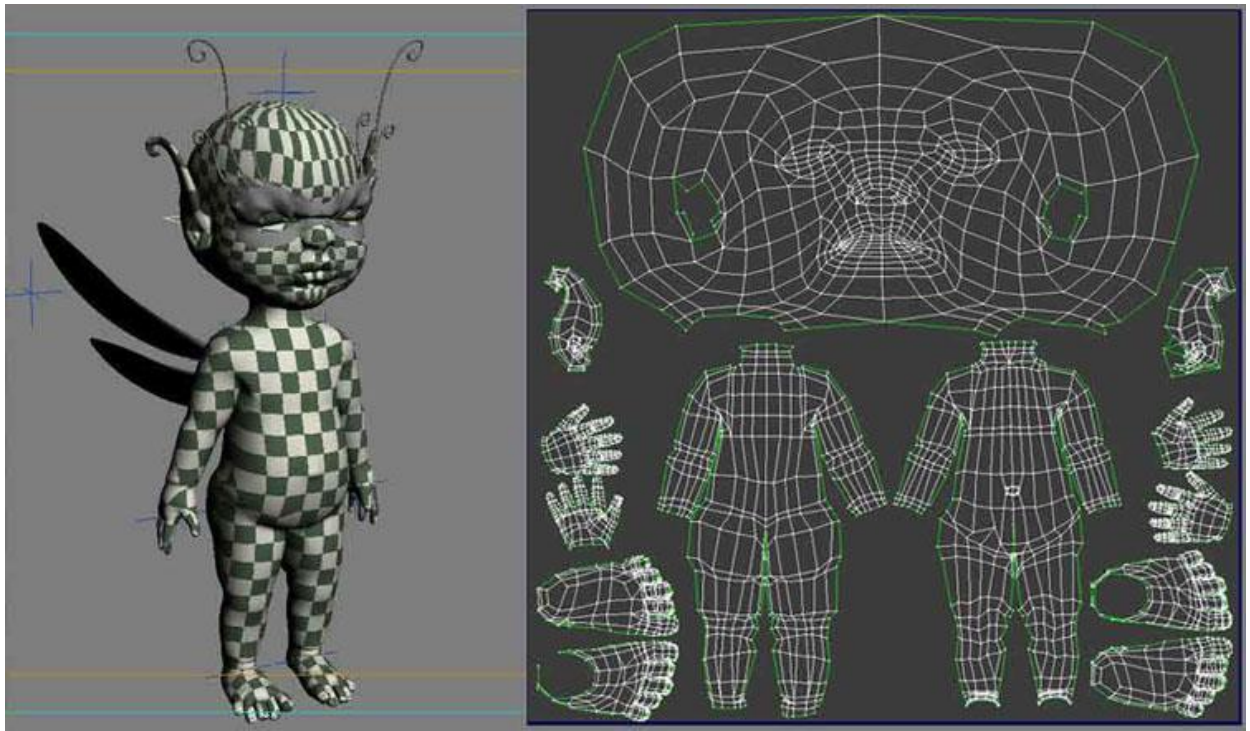
UV coordinates

- Store uv coords on each vertex of the input mesh
- Interpolate using barycentric coordinates

- How to generate coordinates?
  - Manually (time consumming, difficult)
  - Automatically (parametrization optimization)
  - Mathematical mapping (ivertex independent)

Parametrization optimization

- Flatten 3D object onto 2D UV coordinates
- Minimize distortion for each vertex:
  - Distances in UV correspond to distances on mesh
  - Angles in UV plane correspond to angles on 3D mesh
  - Usually requires cuts



Parametrization optimization

- Flatten 3D object onto 2D UV coordinates
- Minimize distortion for each vertex:
  - Distances in UV correspond to distances on mesh
  - Angles in UV plane correspond to angles on 3D mesh
  - Usually requires cuts

SIGGRAPH course:
http://www.inf.usi.ch/hormann/parameterization/

Rendering a 3D mesh

- Per vertex attributes:
  - Position
  - Normal
  - Tangent
  - UV coordinates
  - BRDF (const)
  - …
- + textures

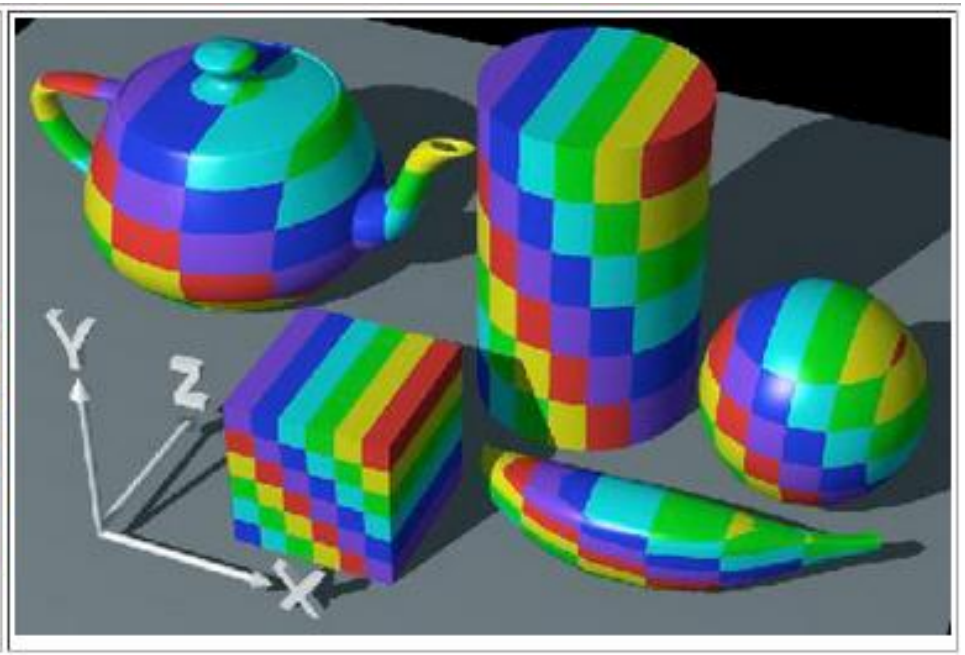
Mathematical mapping

- What if non-triangular geometry?
  - No vertices…

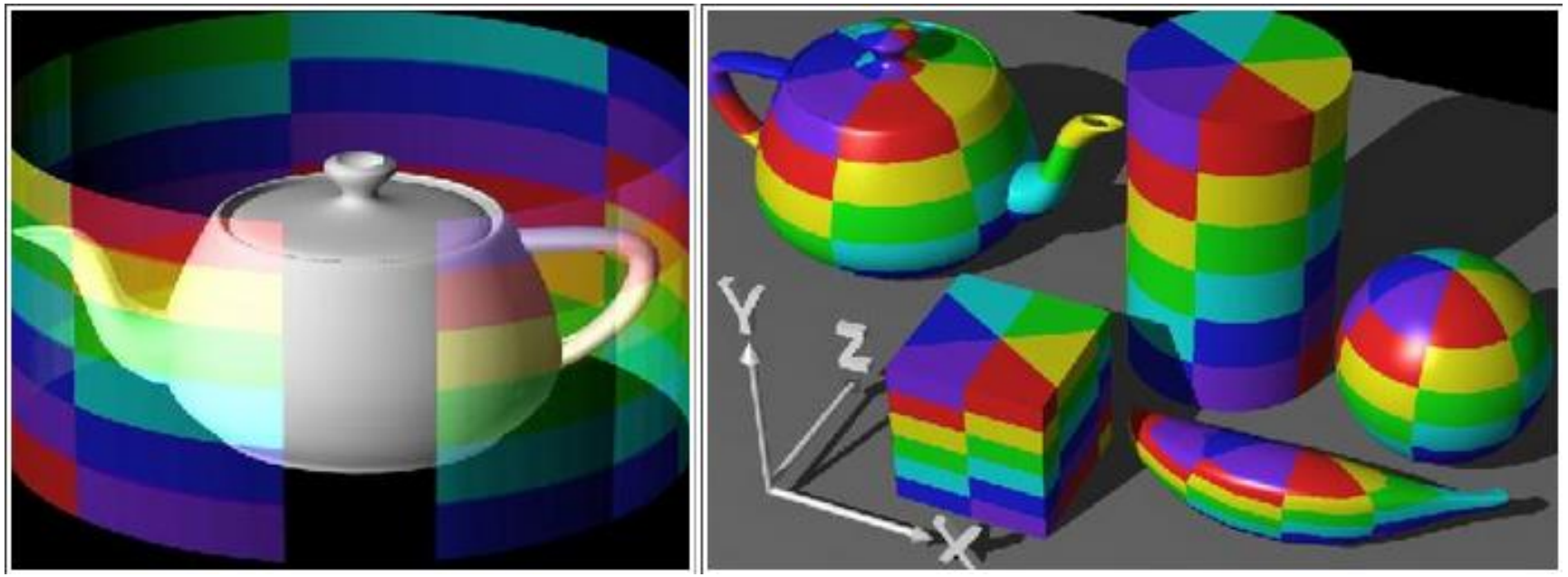
Mathematical mapping

- What if non-triangular geometry?
  - No vertices…
- Use parametric texturing:
  - Deduce (u,v) from (x,y,z)
  - Various possible mappings:

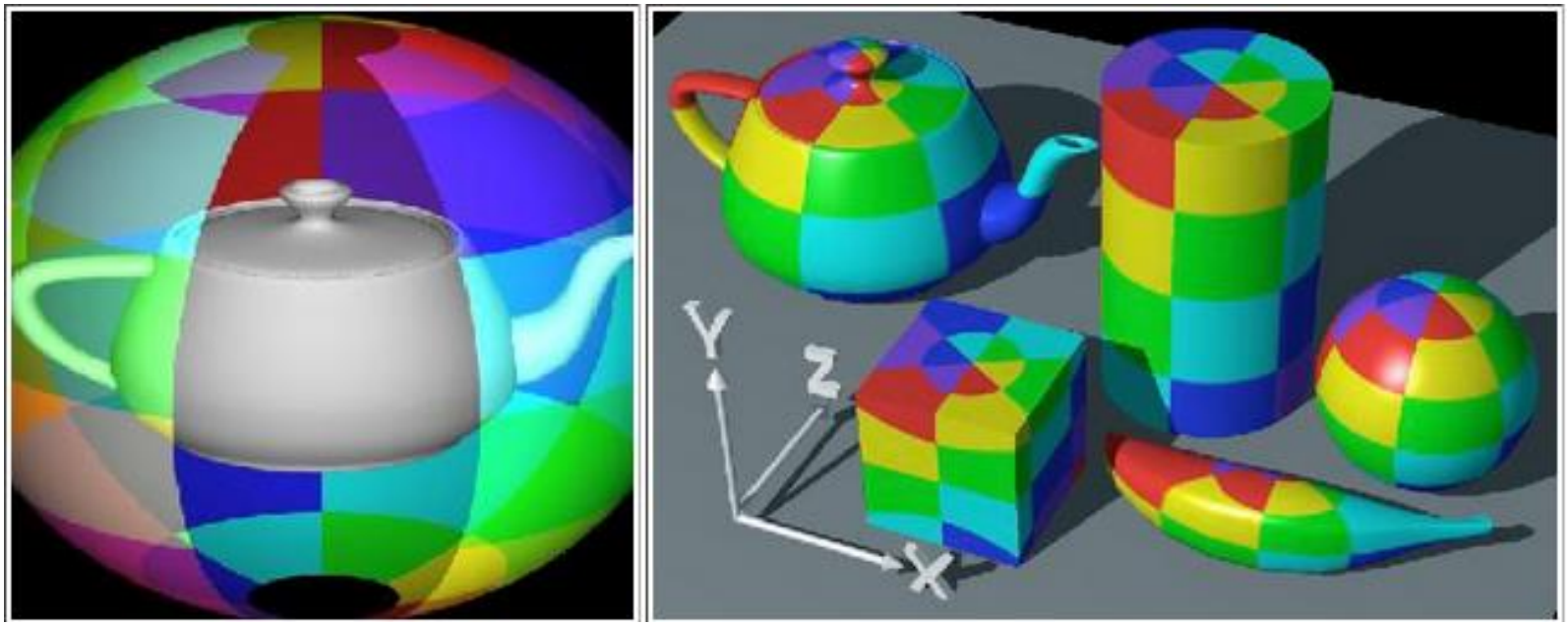
Planar mapping



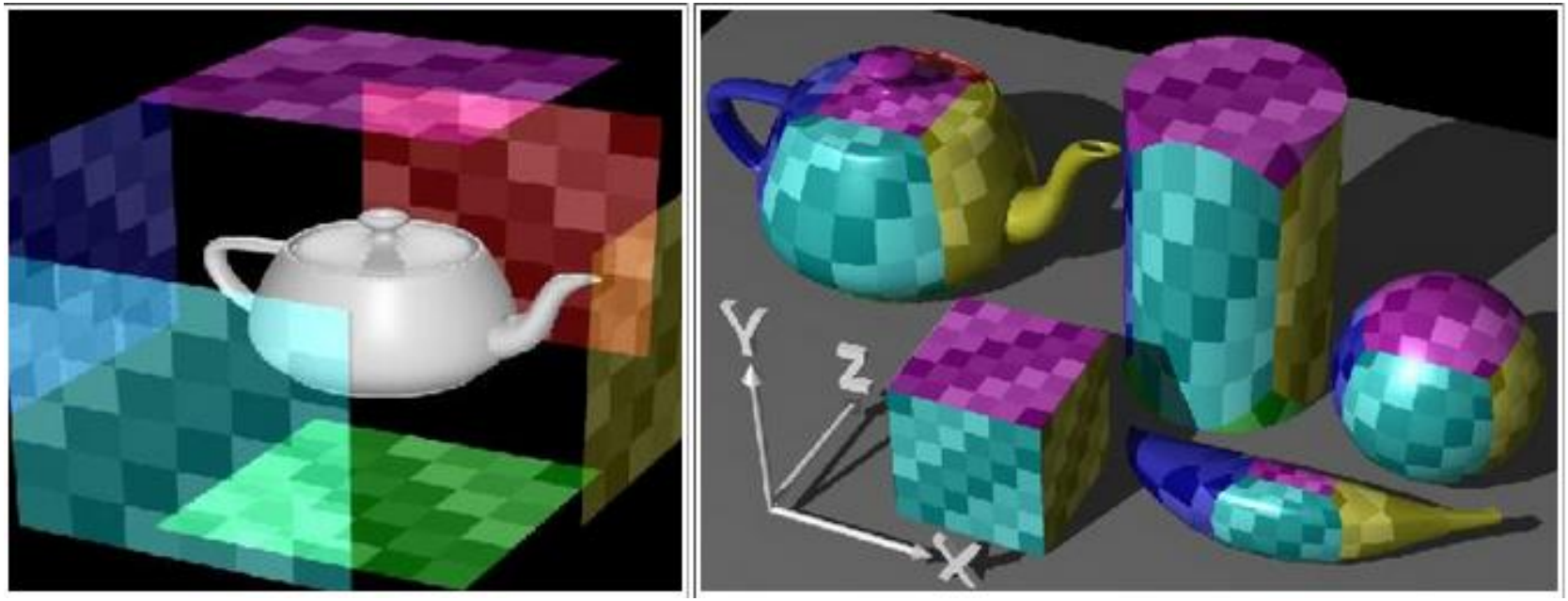
Cylindrical mapping



Spherical mapping



Cube mapping



Projective mapping

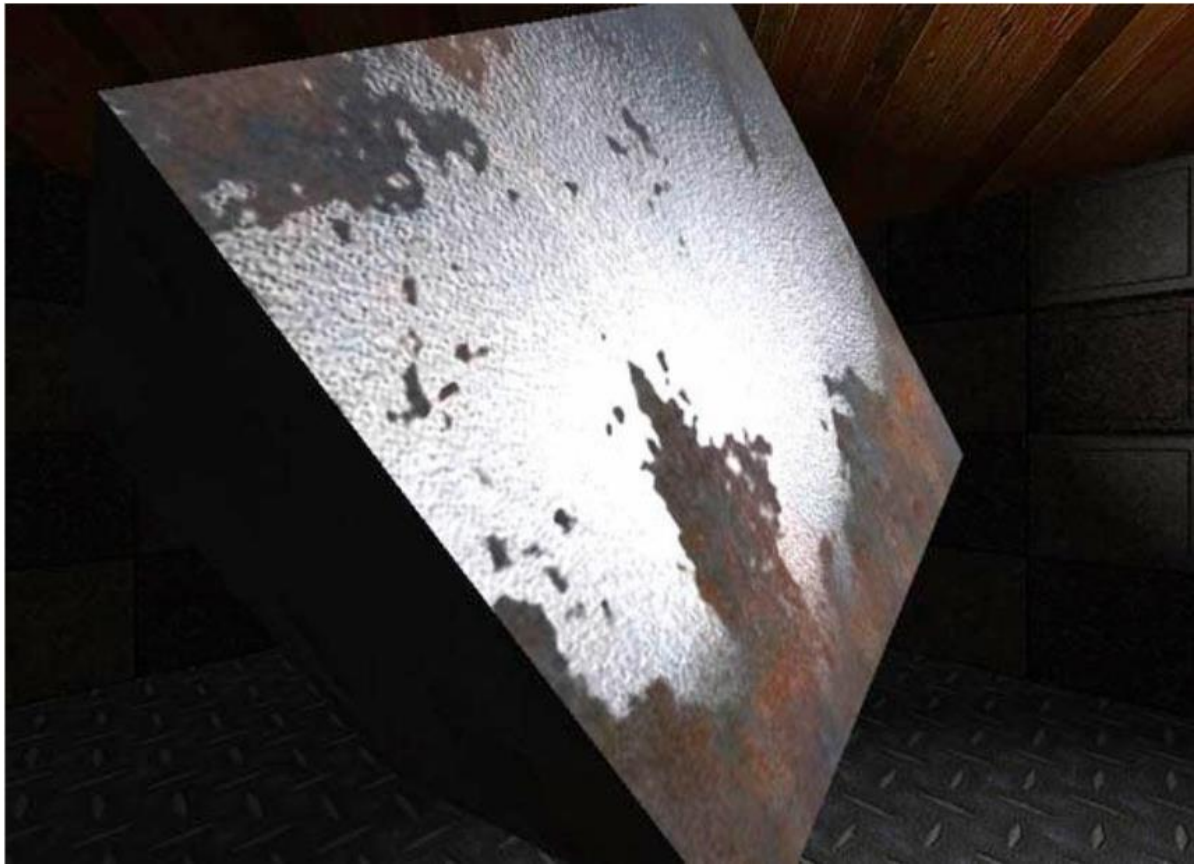
- Slide projector
  - Analogous to a camera
  - No need to specify texture coords explicitely

Projective mapping

- Slide projector https://www.youtube.com/watch?v=RPhGEiM_6lM
  - Analogous to a camera
  - No need to specify texture coords explicitely
- For a given 3D point:
  - Project onto 2D space
  - Use result as texture coordinate

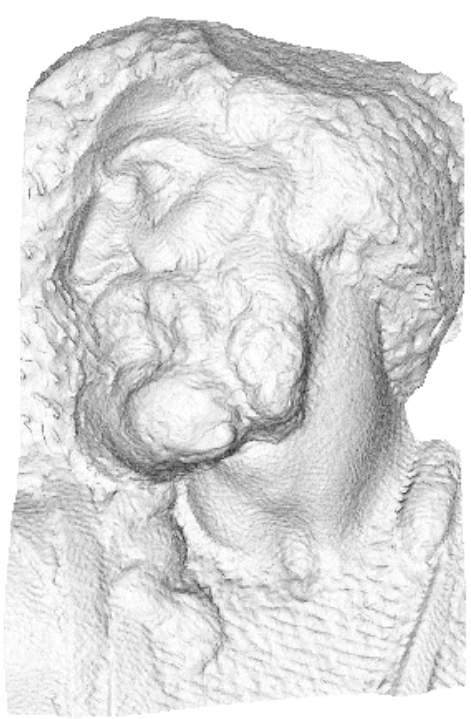
Textures for what?

- BRDF variations

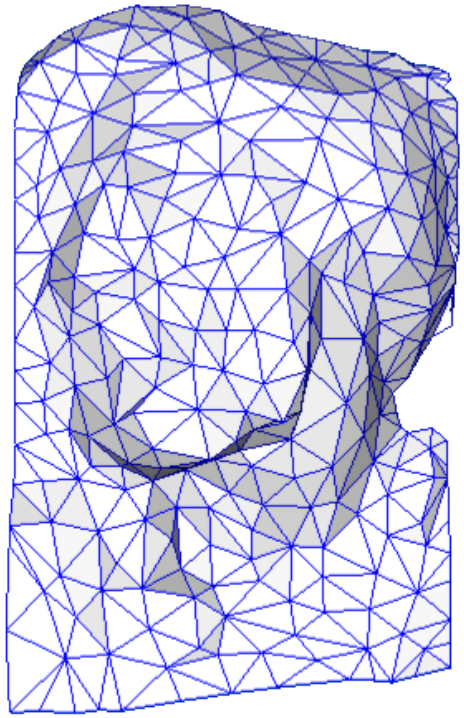


Textures for what?

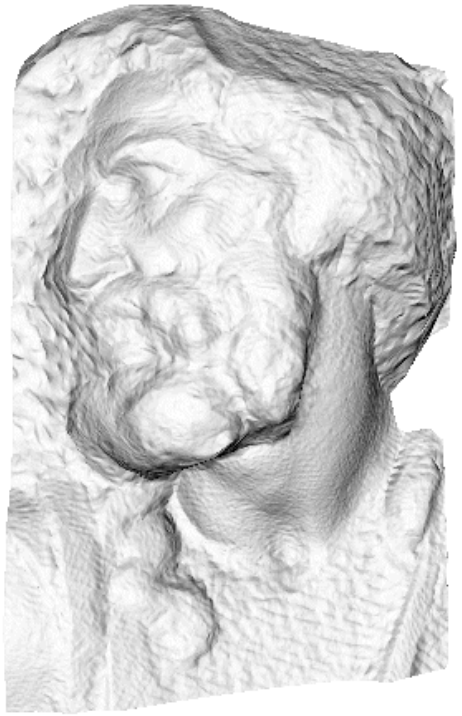
- Geometry variations



original mesh
4M triangles



simplified mesh
500 triangles



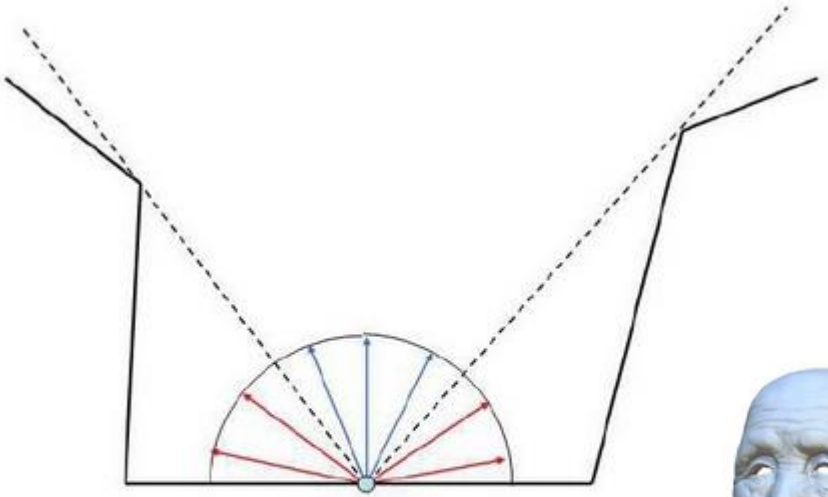
simplified mesh
and normal mapping
500 triangles

Textures for what?

- Geometry variations

Textures for what?

- Visibility (ambient occlusion)



Coarse shadows approximation

$$A = \frac{1}{\pi} \int V(\omega)(\mathbf{n} \cdot \omega) d\omega$$

Original model

With ambient occlusion

Extracted ambient occlusion map

Textures for what?

- Lighting environments

Textures for what?

- Lighting environments

Extreme case

- Image based rendering
  - Produce an image from other images!

Extreme case

- Image based rendering
  - Produce an image from other images!

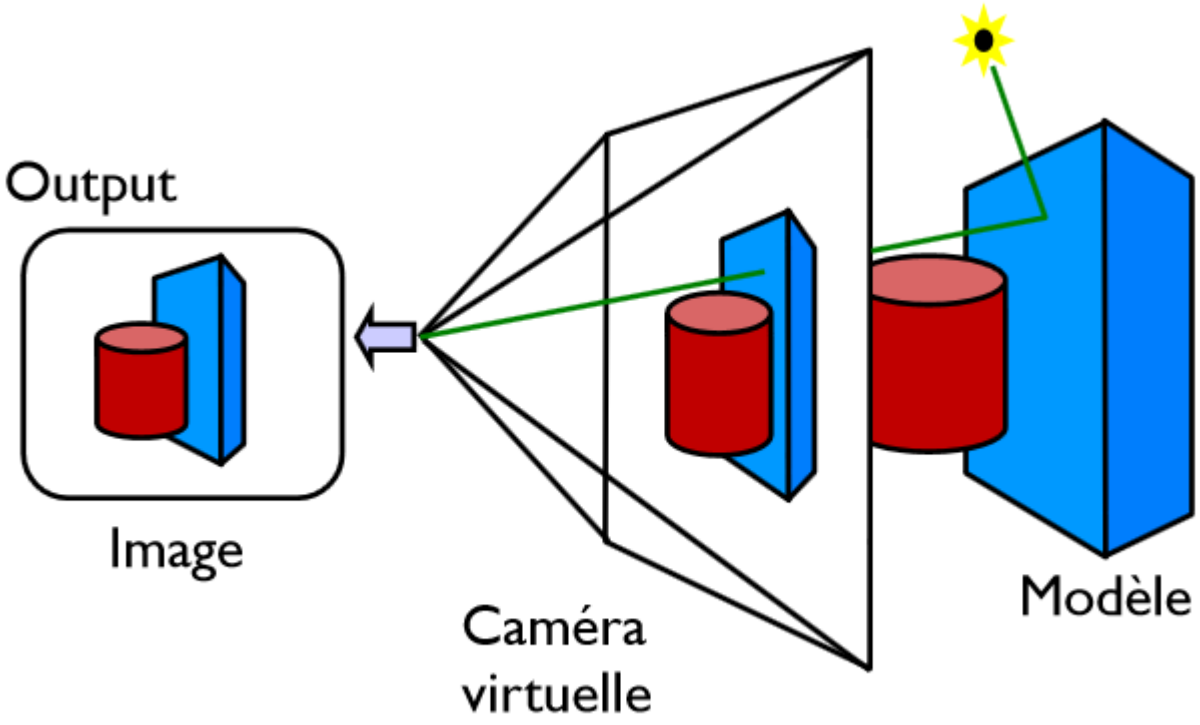
synthesis

- Easy to create new worlds
- Easy to manipulate
  - Viewpoints
  - Objects
- Difficult to make it realistic!

photography

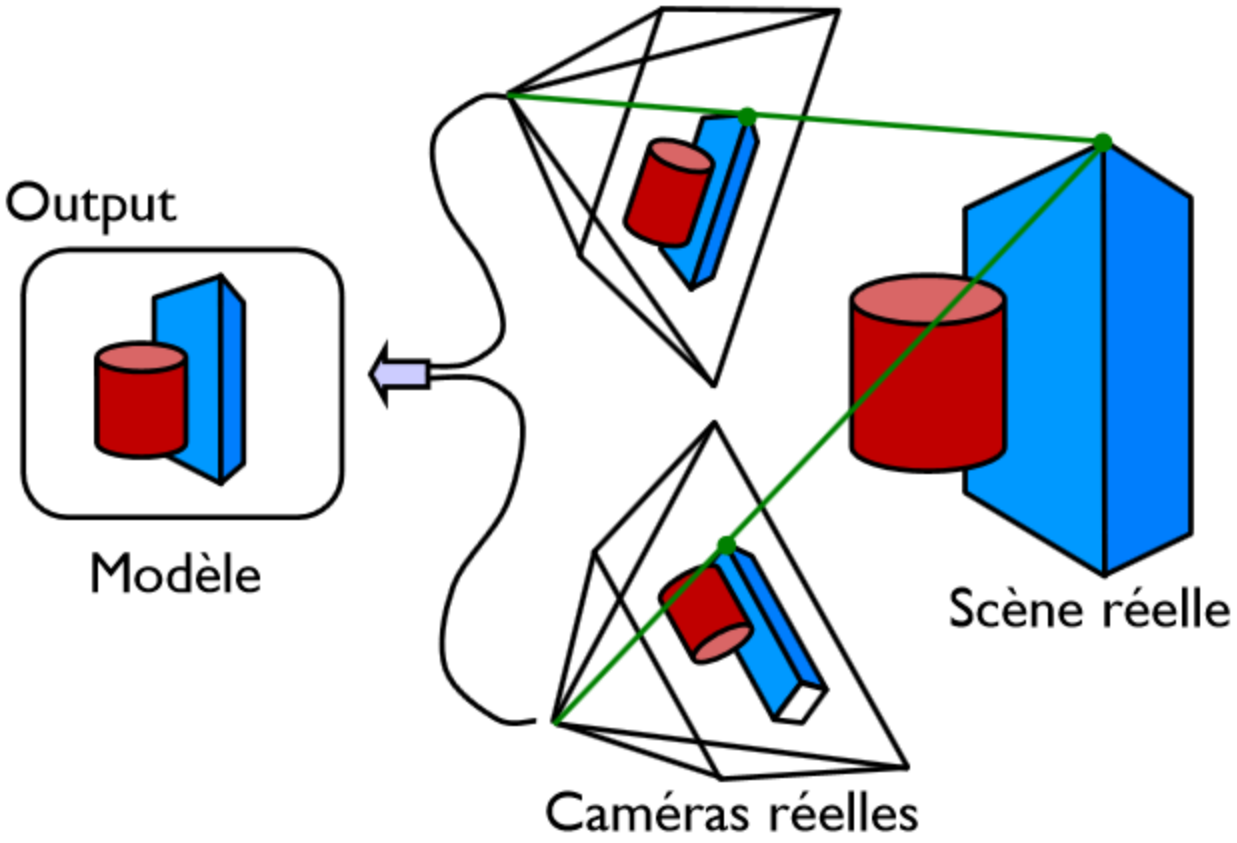
- Realistic…
- Simple aquisition (clic)
- Difficult to manipulate
  - Viewpoints
  - objects

Image synthesis



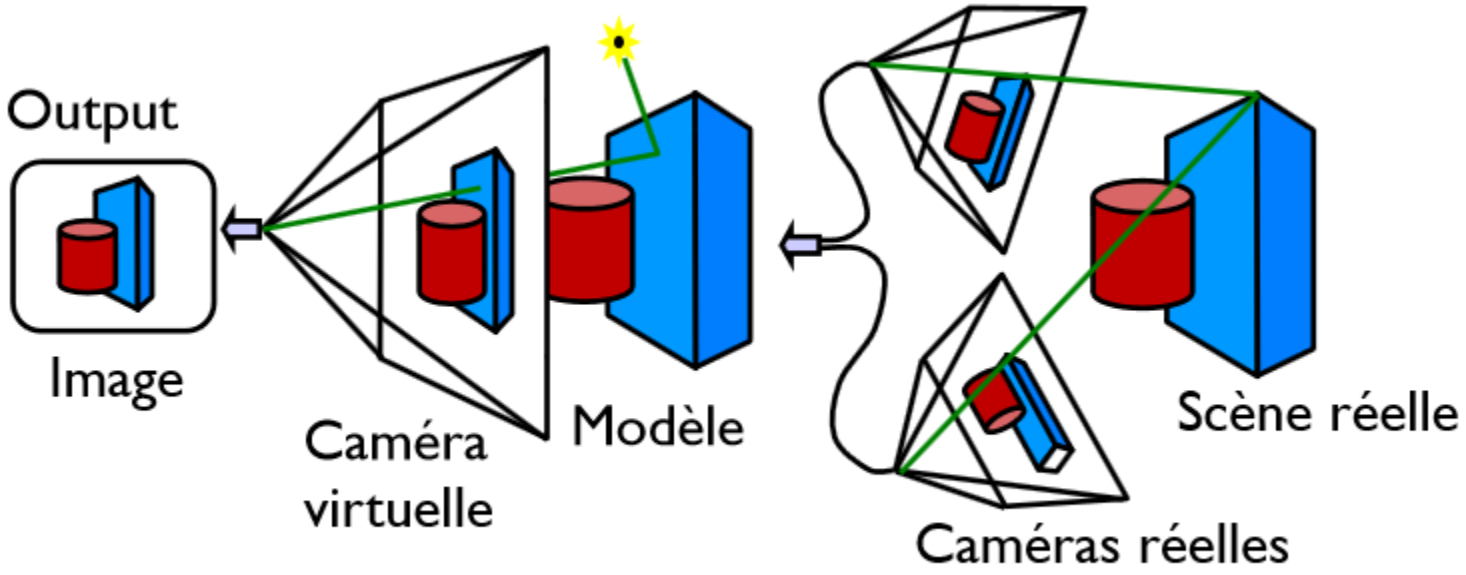
From Pierre Benard lecture on IBR

Vision



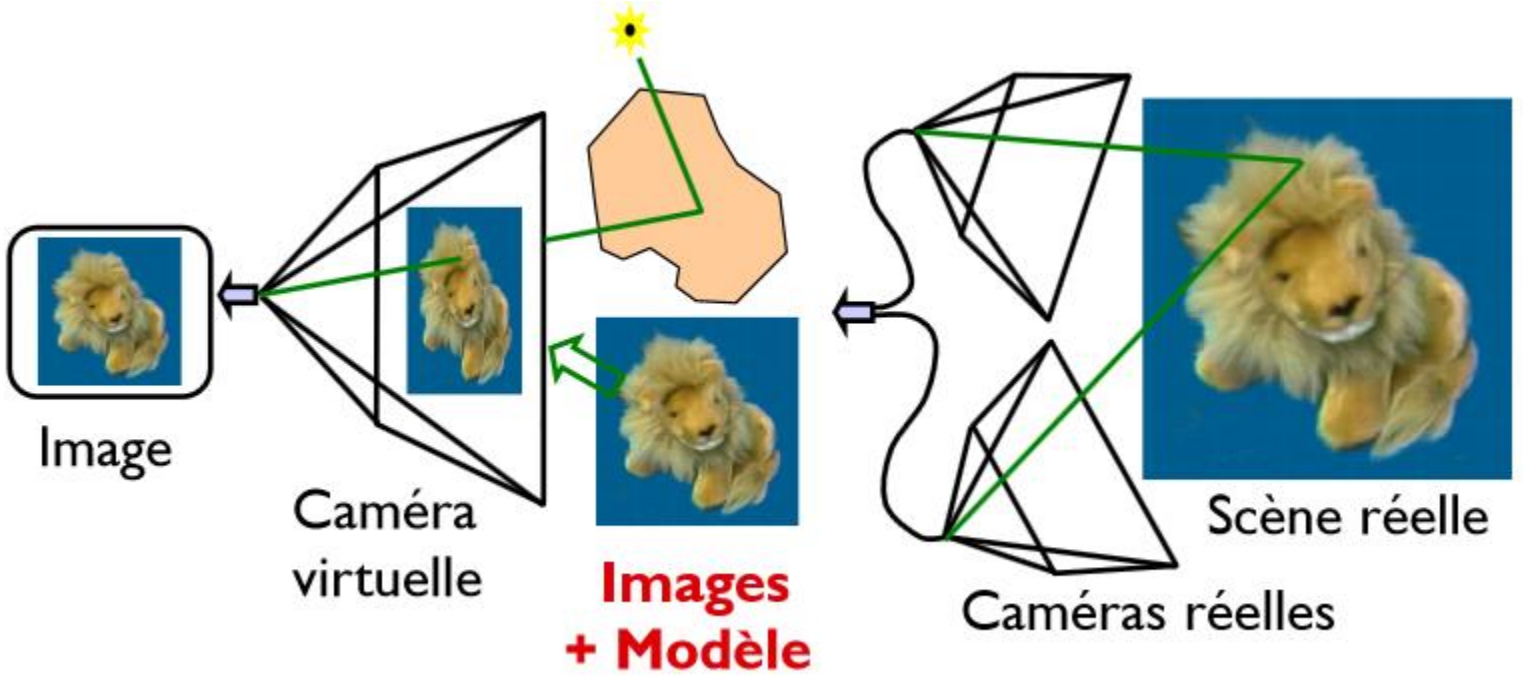
From Pierre Benard lecture on IBR

Combine both



From Pierre Benard lecture on IBR

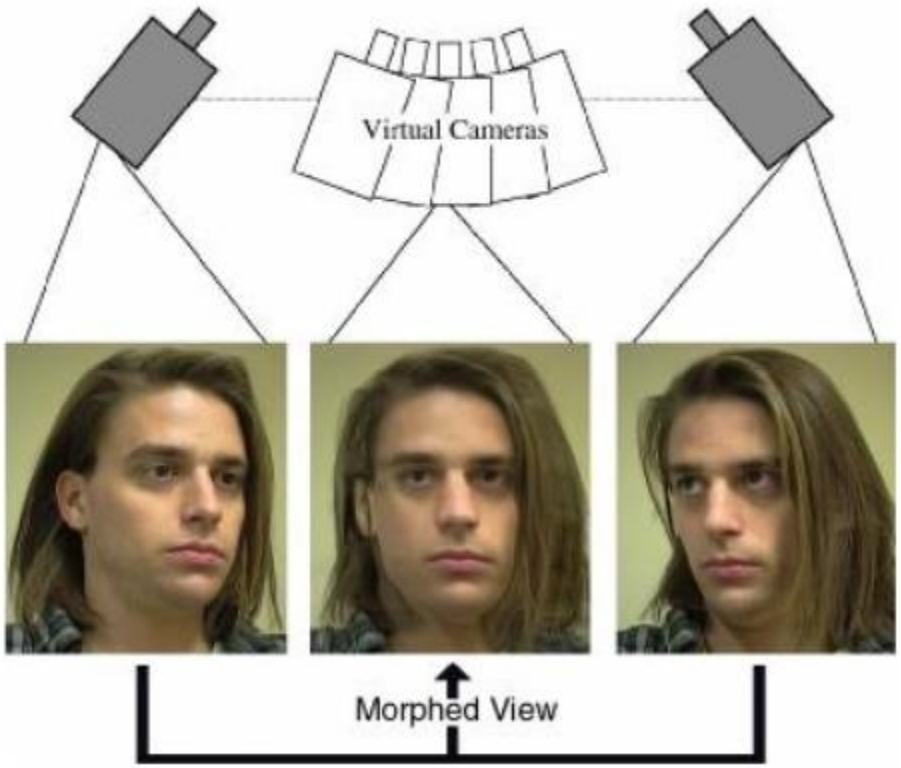
Combine both



From Pierre Benard lecture on IBR

Interpolate viewpoints

[Beier and Neely '92, Chen and Williams '93, Seitz and Dyer '96]



From Pierre Benard lecture on IBR

Morphing

https://www.youtube.com/watch?v=nUDIoN-_Hxs

From Pierre Benard lecture on IBR

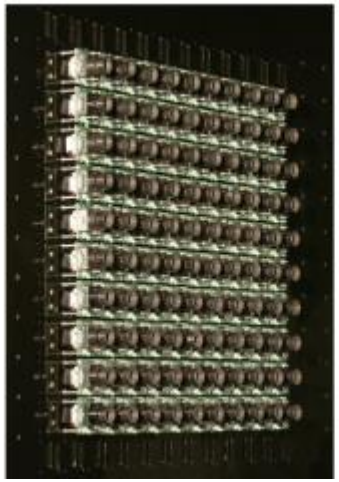
Plenoptic function

$$P(\theta, \phi, \lambda, t, V_x, V_y, V_z)$$

- Capture all possible images around u
- IBR = reconstruct P from samples

From Pierre Benard lecture on IBR

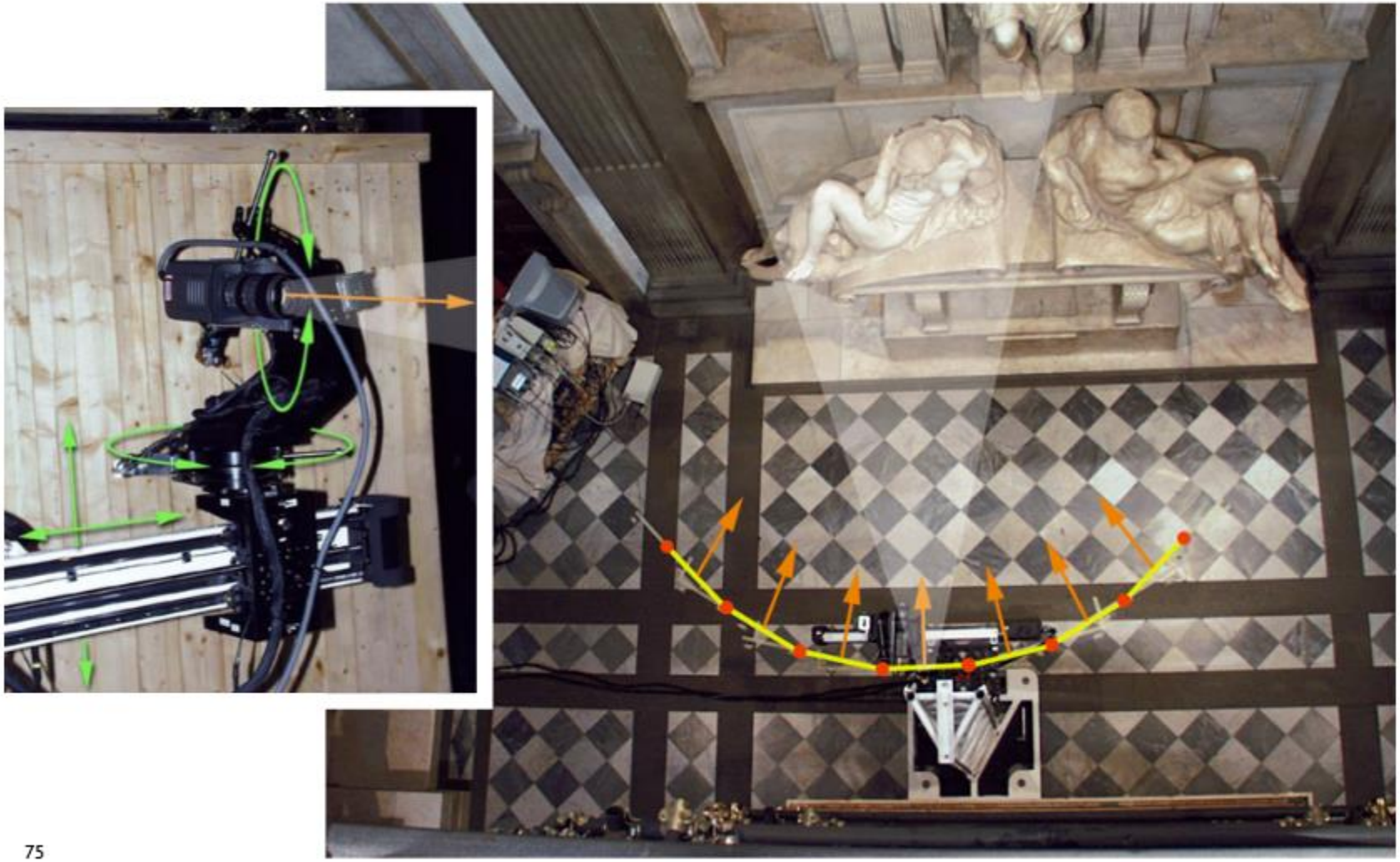
Standford cameras



- Array of cameras
  - 640X480 pixels X 30 FPS X 128 cameras
  - X 30 FPS

From Pierre Benard lecture on IBR

Digital Michelangleo project



75

From Pierre Benard lecture on IBR

Light field viewer

http://lightfield.stanford.edu/lfs.html

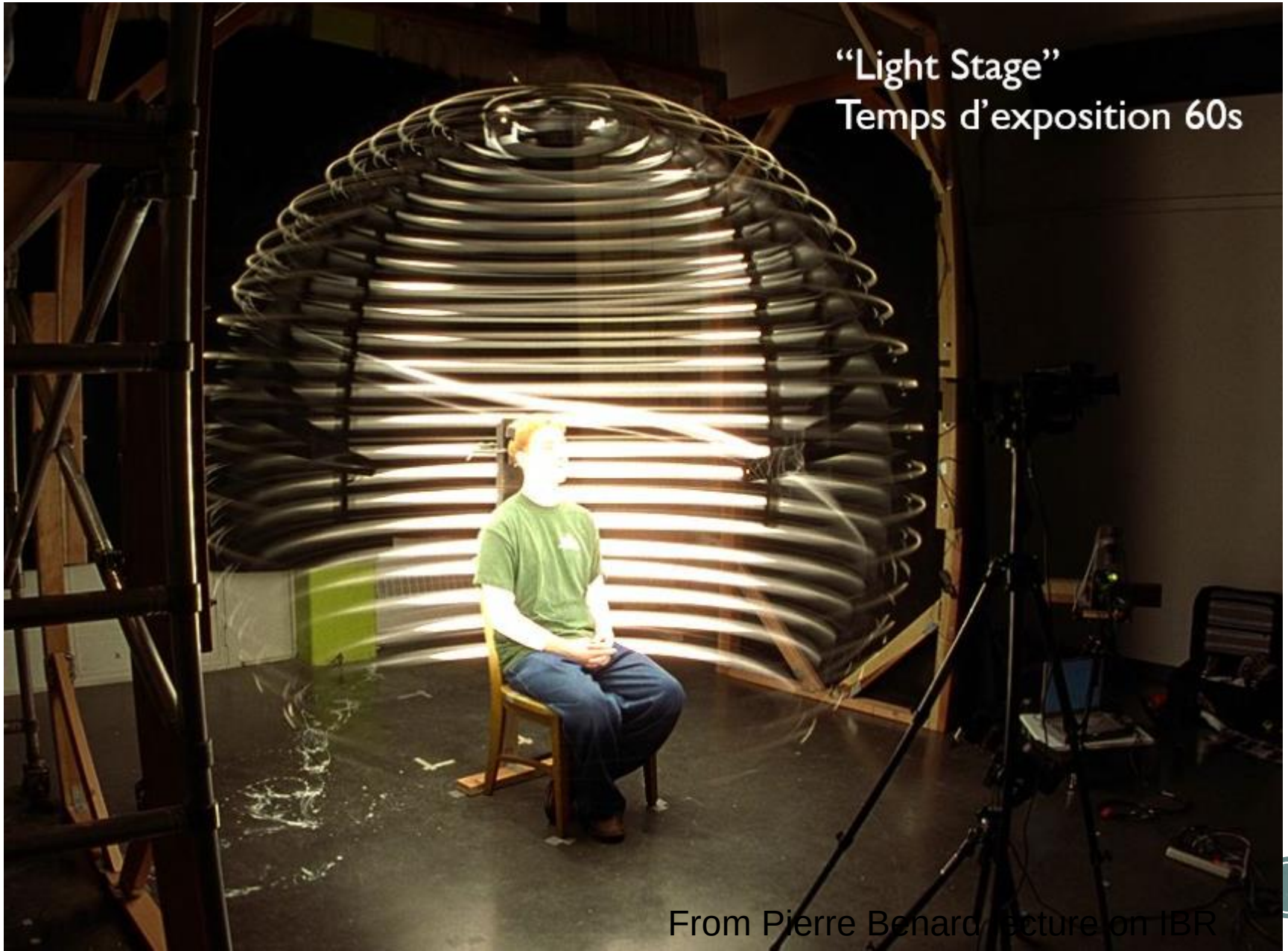
From Pierre Benard lecture on IBR

Time splice

http://www.timesplice.com.au/camera-array.html

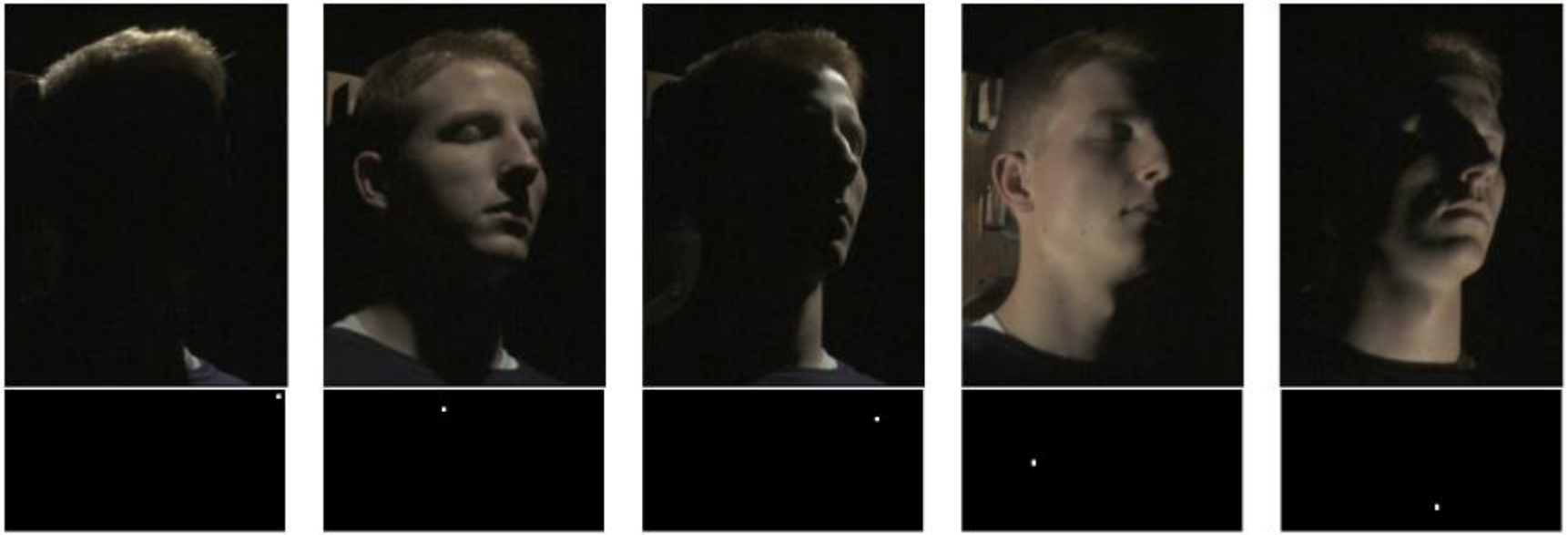
From Pierre Benard lecture on IBR

Relighting



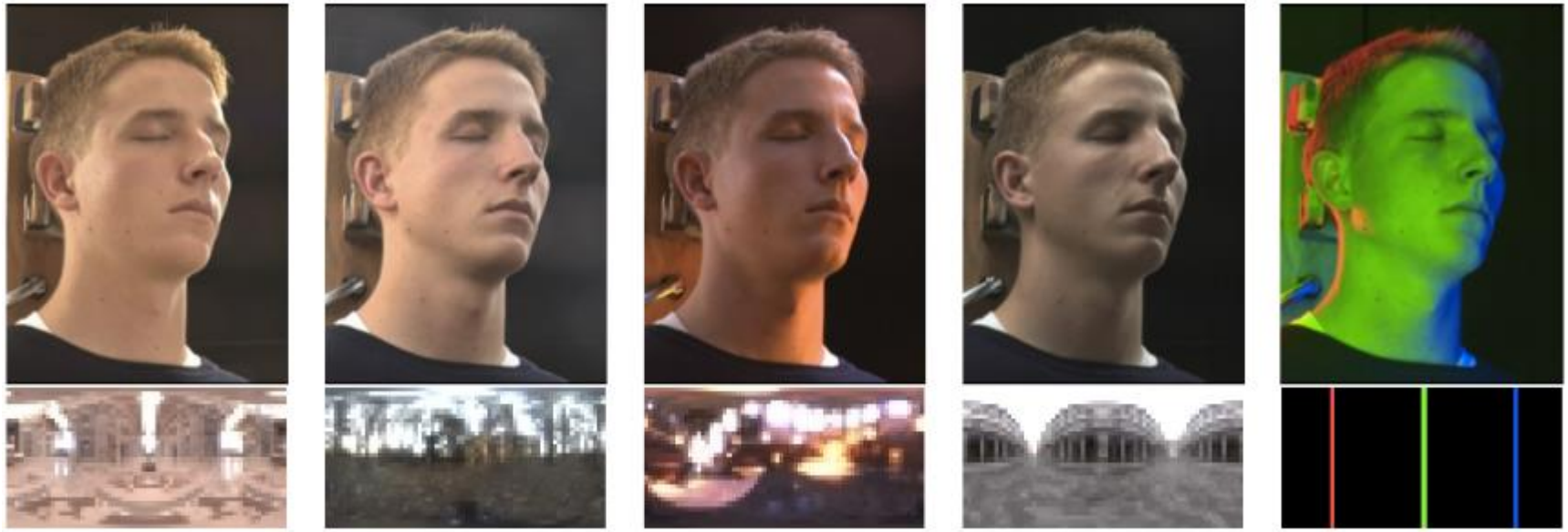
From Pierre Benard : cture on IBR

Relighting



From Pierre Benard lecture on IBR

Relighting

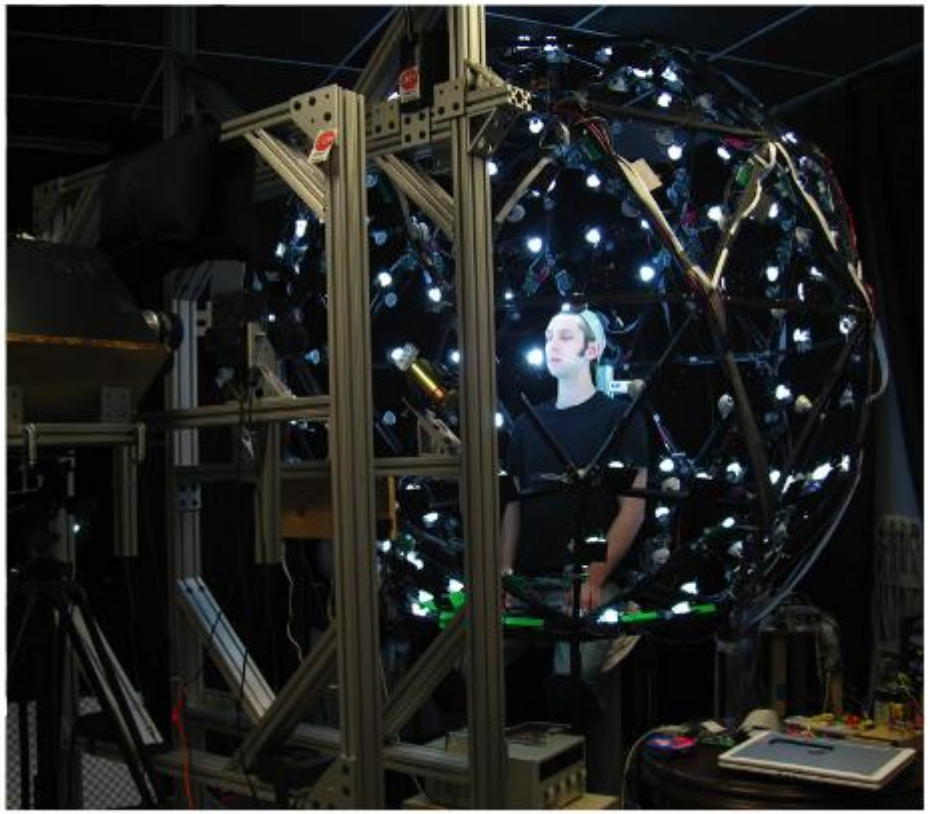


From Pierre Benard lecture on IBR

Relighting

The Curious Case of Benjamin Button

Avatar



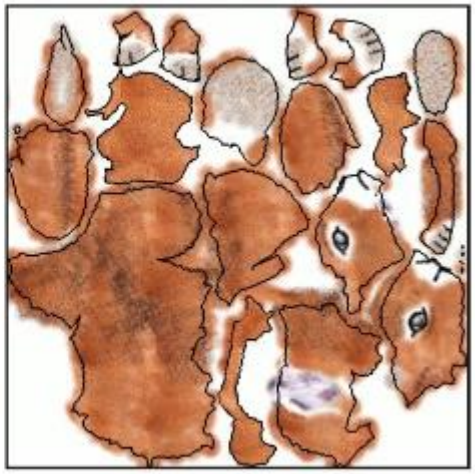
https://www.youtube.com/watch?v=piJ4Zke7EUw

Digital Emily project

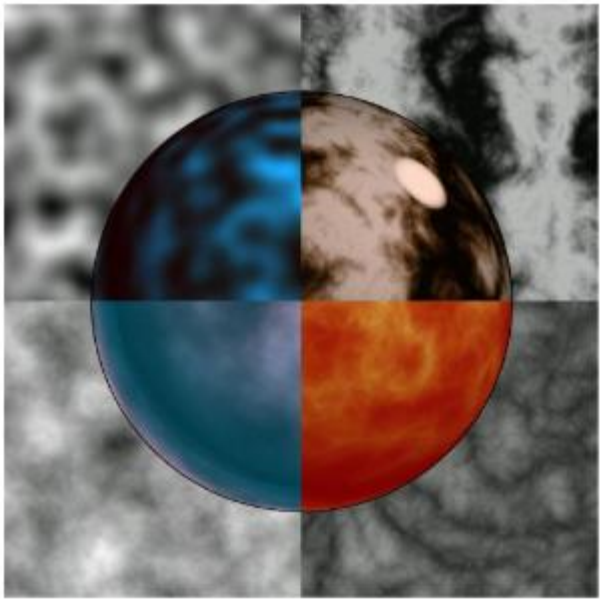
From Pierre Benard lecture on IBR

Lets go back to our textures!

- From data
  - Colors, coefs, normals stored in 2D images

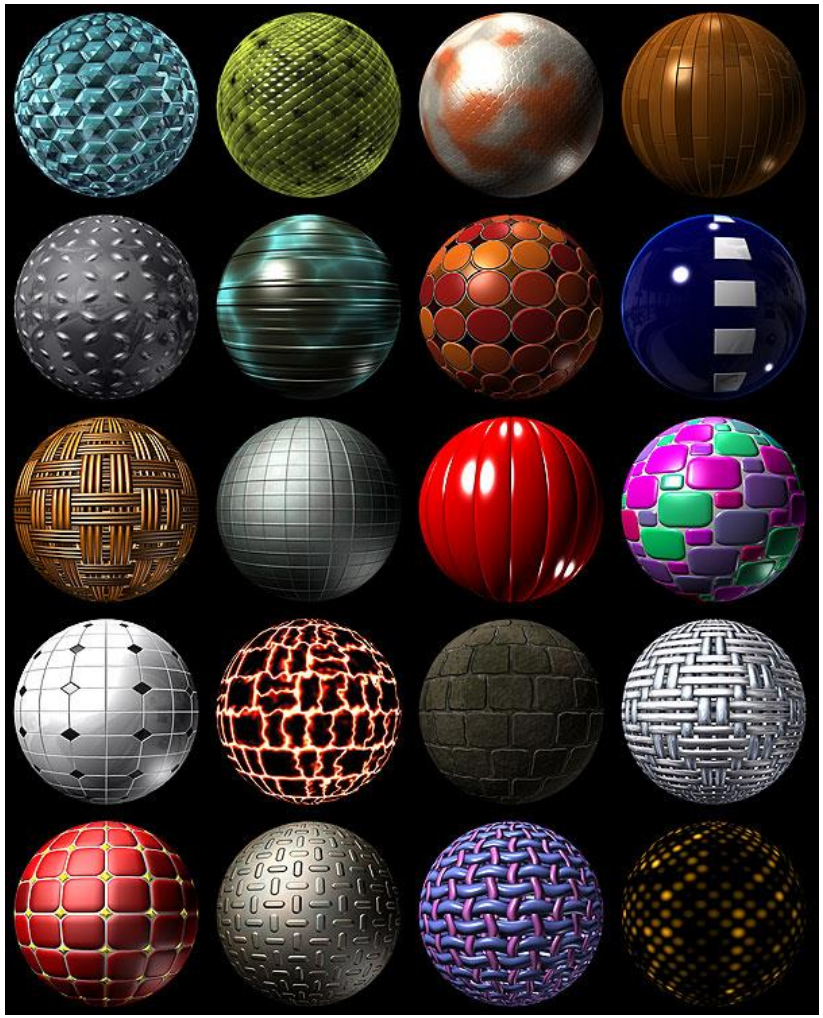


- Procedural shader
  - Little program that compute info at a given position



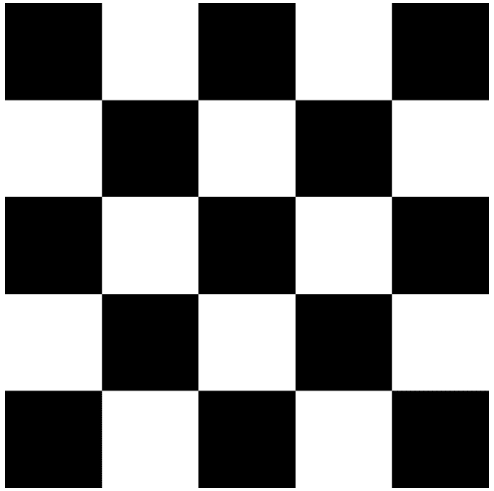
Procedural textures

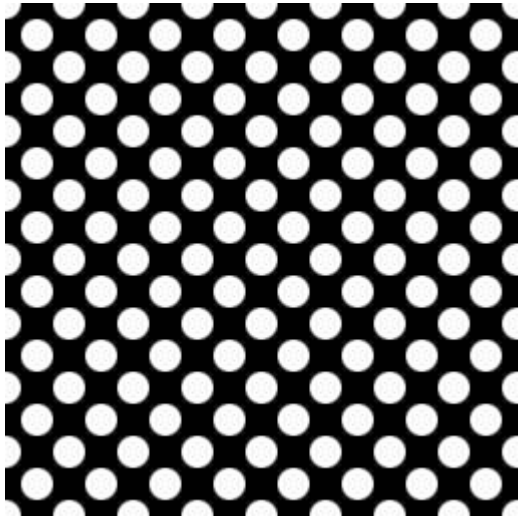
- pros
  - Easy to implement (ray tracing)
  - Compact
  - Infinite resolution
- Cons
  - Non-intuitive
  - Difficult to match existing textures

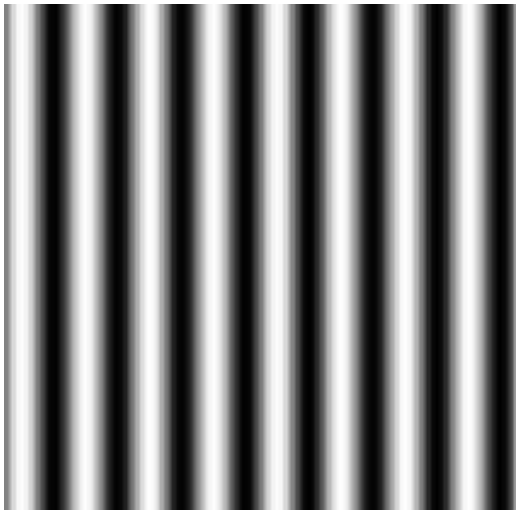


Simple math functions

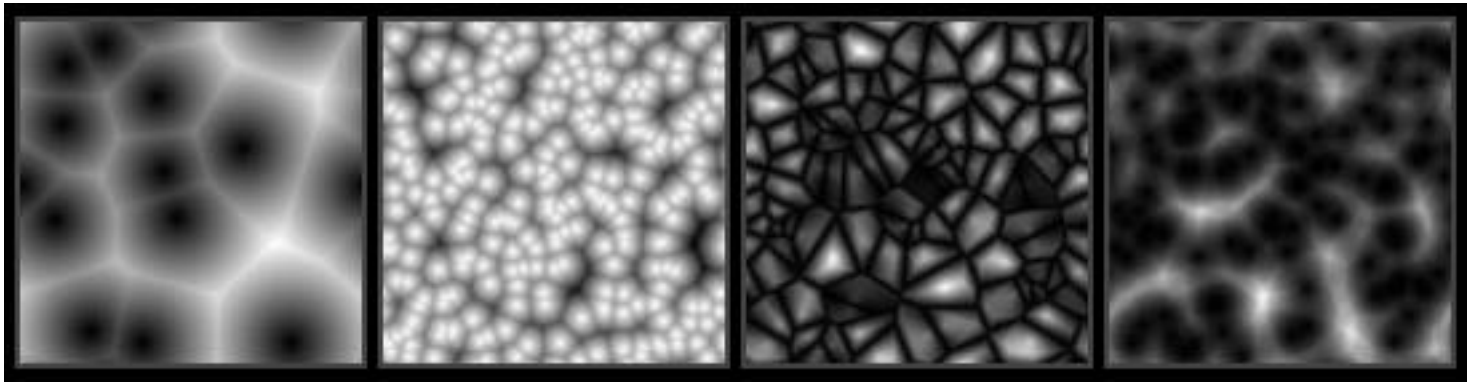
- Combination of simple functions
  - Mod
  - Clamp
  - Mix
  - Sin / cos / tan
  - Pow
  - Exp
  - Etc…







Cellular textures

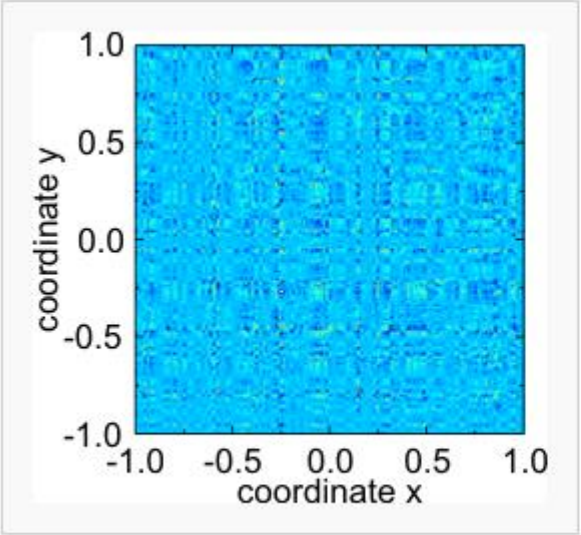


- Generate a bunch of random points
- For each pixel
  - Find the nearest distance to the nearest couple points
  - Use these values to determine a color
- Voronoi-like

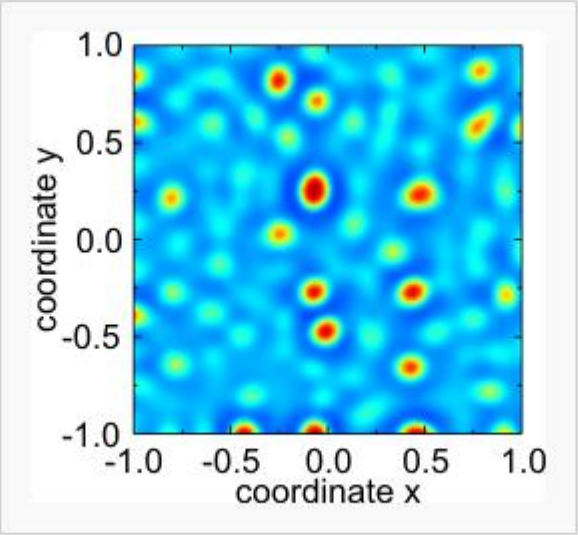
https://www.google.fr/search?q=cellular+texture&tbm=isch&tbo=u&source=univ&sa=X&ei=FADJVKG8McXwUpfRg6AK&ved=0CCIQsAQ&biw=1920&bih=969

Minimizing functions

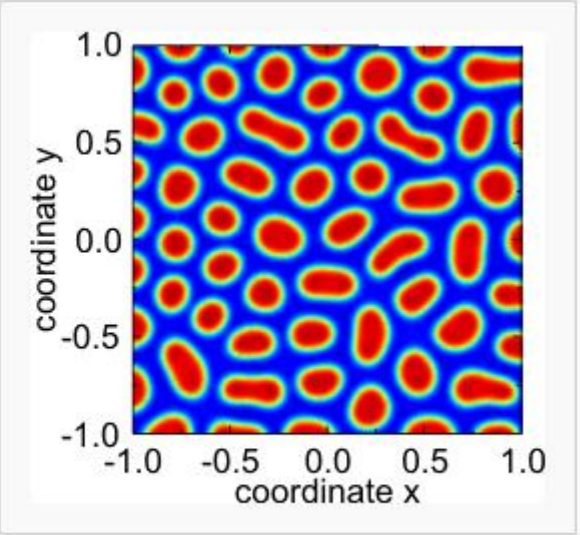
- Genetic algorithms
- Reaction-diffusion
- …



Noisy initial conditions at $t = 0$.



State of the system at $t = 10$.



Almost converged state at $t = 100$.

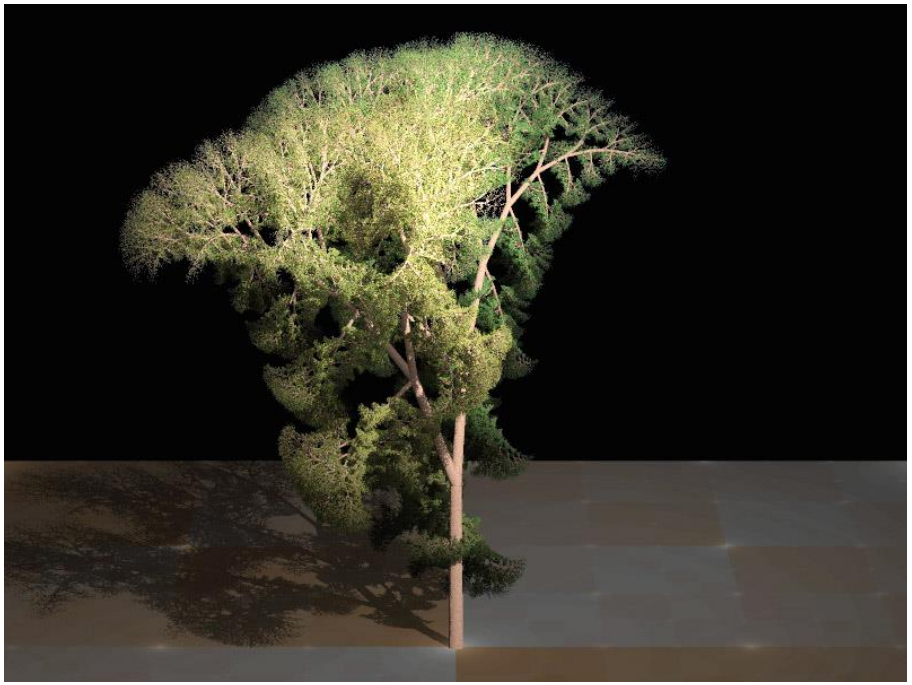
L-systems

- Formal grammar

  - Alphabet : V = {A, B}
  - Constantes : S = {}
  - Axiome de départ : w = A
  - Règles : (A → AB) ∧ (B → A)

Notation :

```
Algue
{
Axiom A
A=AB
B=A
}
```



L-systems

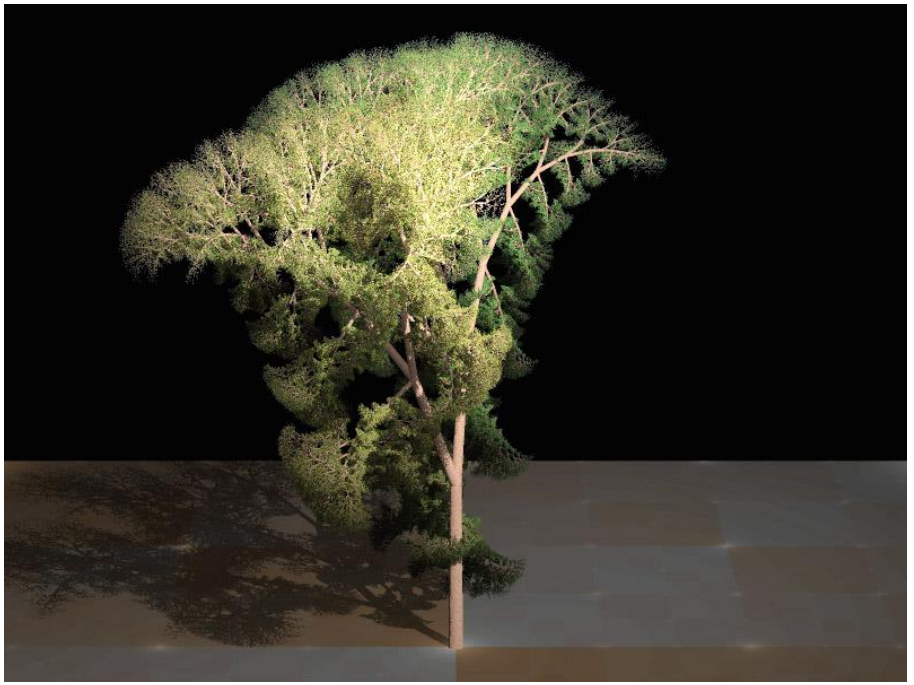
- Formal grammar

  - Alphabet : V = {A, B}
  - Constantes : S = {}
  - Axiome de départ : w = A
  - Règles : (A → AB) ∧ (B → A)

Notation :

```
Algue
{
Axiom A
A=AB
B=A
}
```

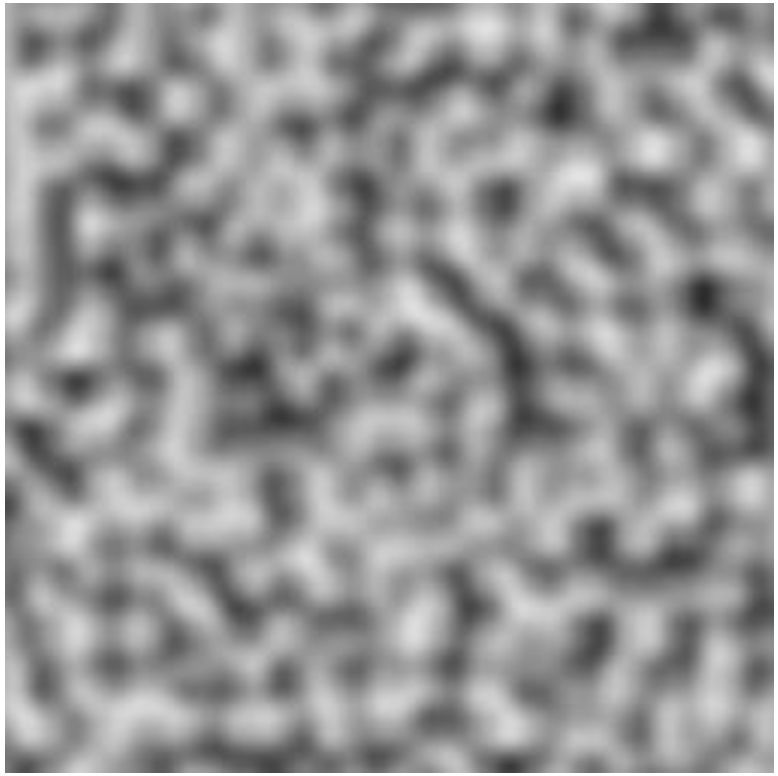
- n=0, A
- n=1, AB
- n=2, AB A
- n=3, AB A AB
- n=4, AB A AB AB A
- n=5, AB A AB AB A AB A AB
- n=6, AB A AB AB A AB A AB AB A AB AB A



http://www.kevs3d.co.uk/dev/lsystems/

Perlin-like textures

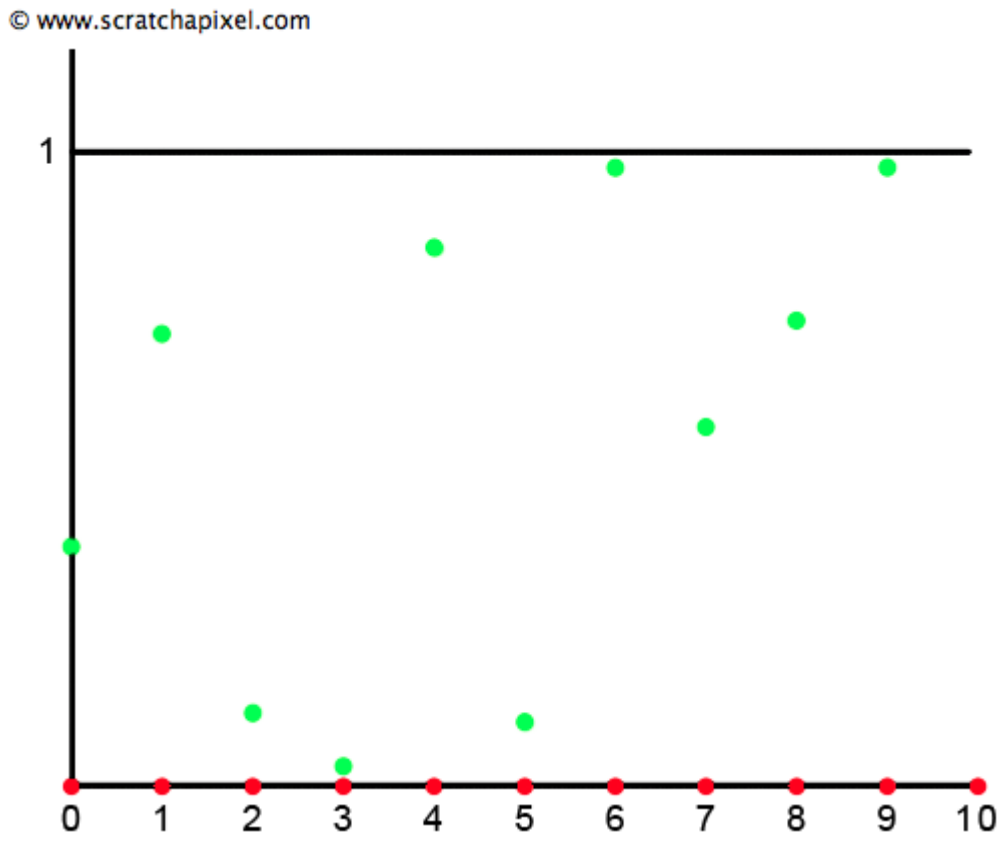
- Requirements
  - Pseudo random
  - Arbitrary dimension
  - Smooth
  - Band pass (one scale)
  - Little memory usage
  - Implicit evaluation



Perlin-like textures

Value noise 1D case

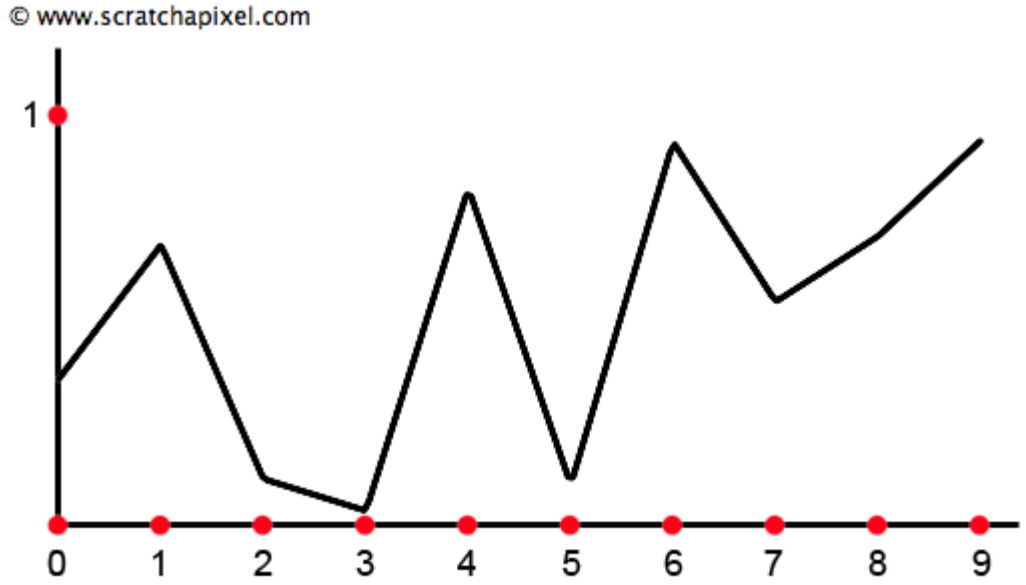
- Distribute random values at particular locations (a grid)…



Perlin-like textures

Value noise 1D case

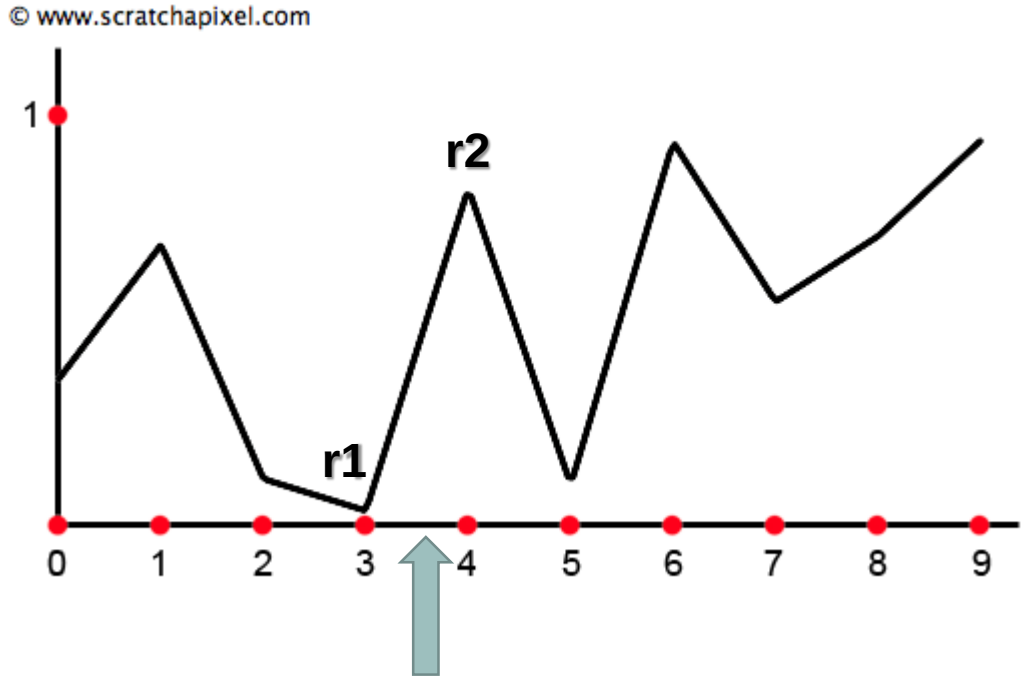
- Distribute random values at particular locations (a grid)…
- … and interpolate



Perlin-like textures

Value noise 1D case

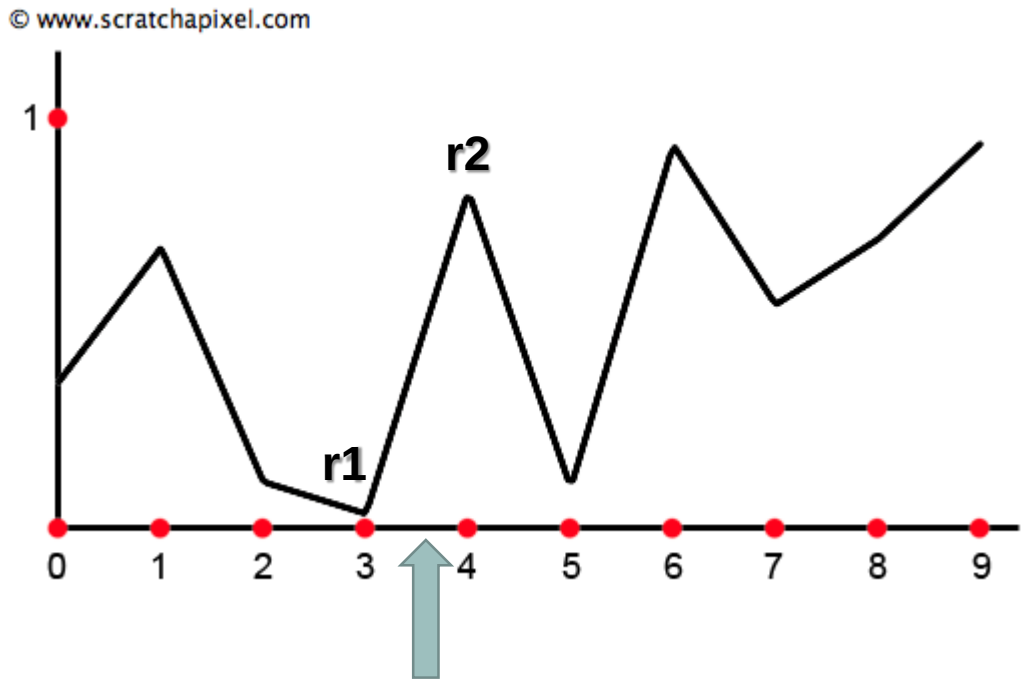
- Evaluation algorithm for a given point x



Perlin-like textures

Value noise 1D case

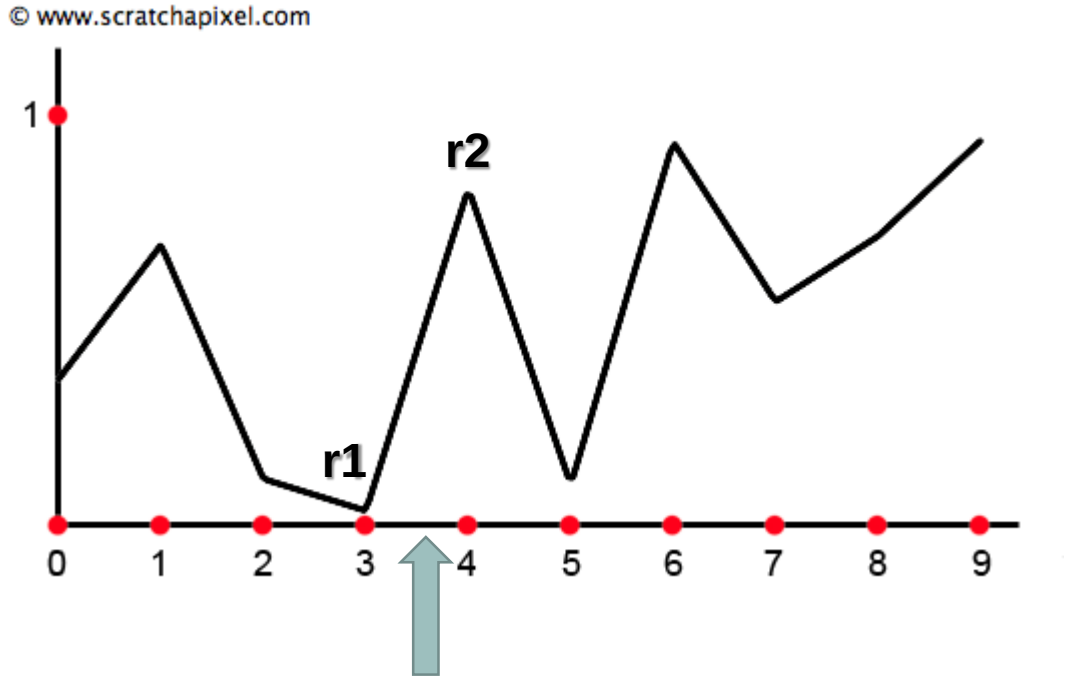
- Evaluation algorithm for a given point x
- Get the associated 2 random values?



Perlin-like textures

Value noise 1D case

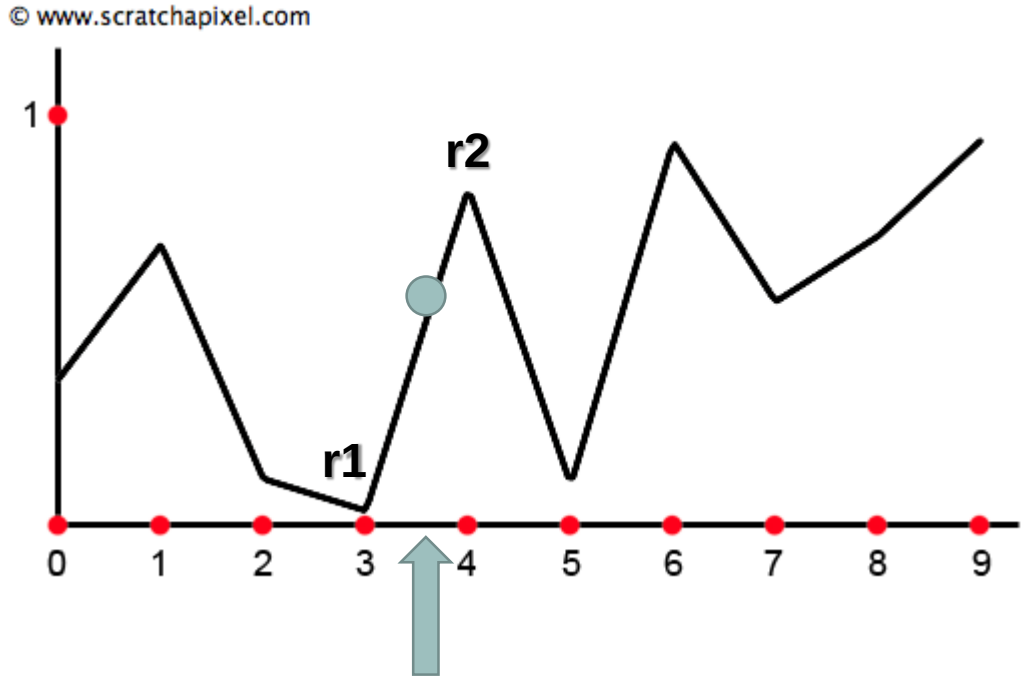
- Evaluation algorithm for a given point x
- Get the associated 2 random values?
  - Pseudo random function
  - Precomputed in an array



Perlin-like textures

Value noise 1D case

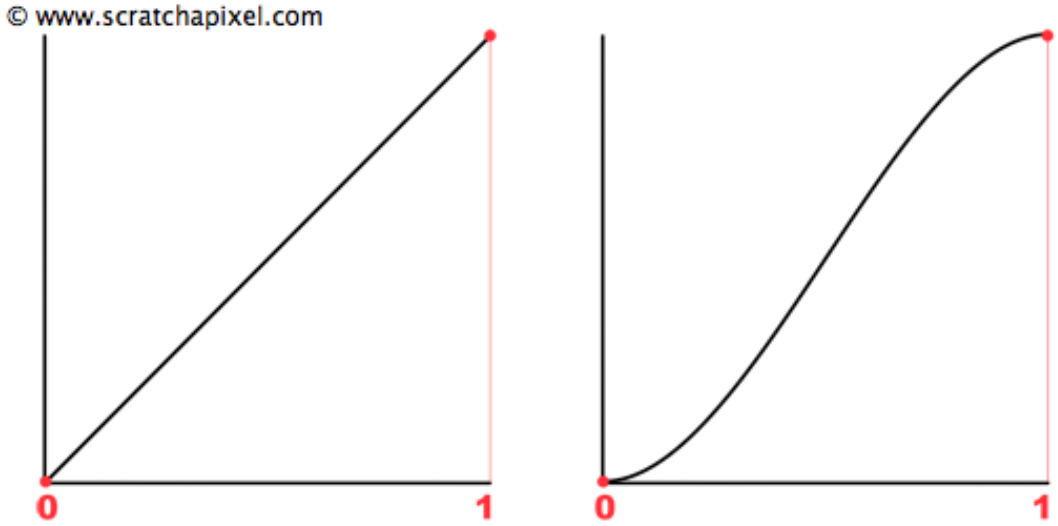
- Evaluation algorithm for a given point x
- Get the associated 2 random values?
  - Pseudo random function
  - Precomputed in an array
- Get relative position of x (between 0 and 1)
- Mix



Perlin-like textures

Value noise 1D case

- Evaluation algorithm for a given point x
- Get the associated 2 random values?
  - Pseudo random function
  - Precomputed in an array
- Get relative position of x (between 0 and 1)
- Mix

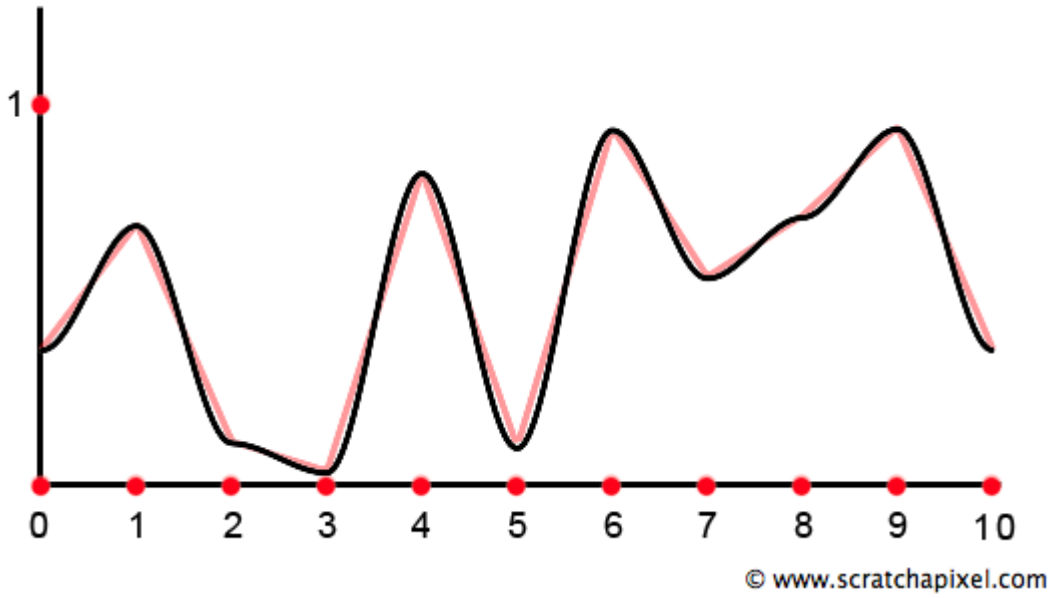


S-shaped function $t = t^2(3 - 2t)$

Perlin-like textures

Value noise 1D case

- Evaluation algorithm for a given point x
- Get the associated 2 random values?
  - Pseudo random function
  - Precomputed in an array
- Get relative position of x (between 0 and 1)
- Mix



Perlin-like textures

Value noise 1D case

- Evaluation algorithm for a given point x
- Get the associated 2 random values?
  - Pseudo random function
  - Precomputed in an array

Pros/cons?

- Get relative position of x (between 0 and 1)
- Mix

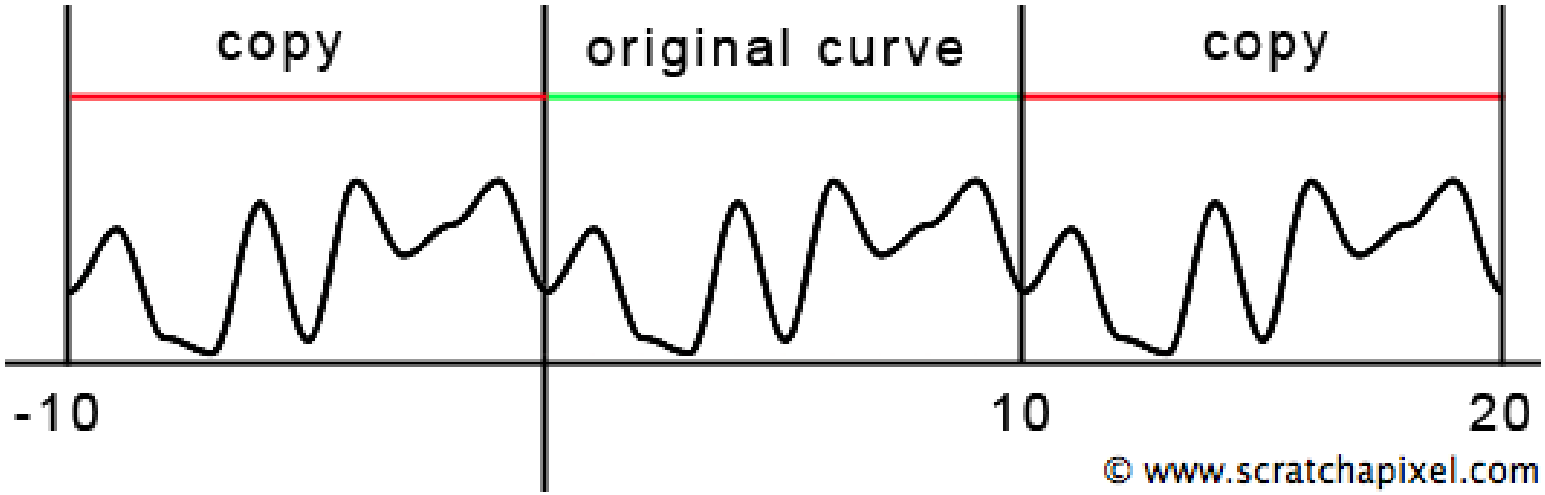
Perlin-like textures

Value noise 1D case

- Evaluation algorithm for a given point x
- Get the associated 2 random values?
  - Pseudo random function
  - Precomputed in an array

Pros/cons?

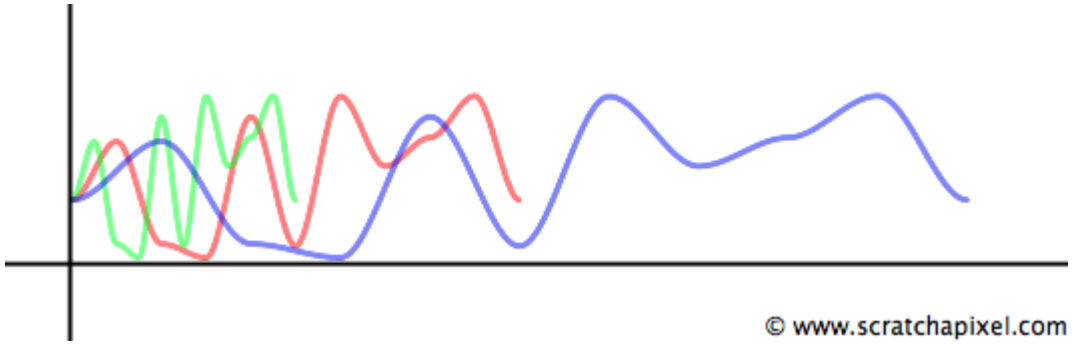
- Get relative position of x (between 0 and 1)
- Mix



Perlin-like textures

Value noise 1D case

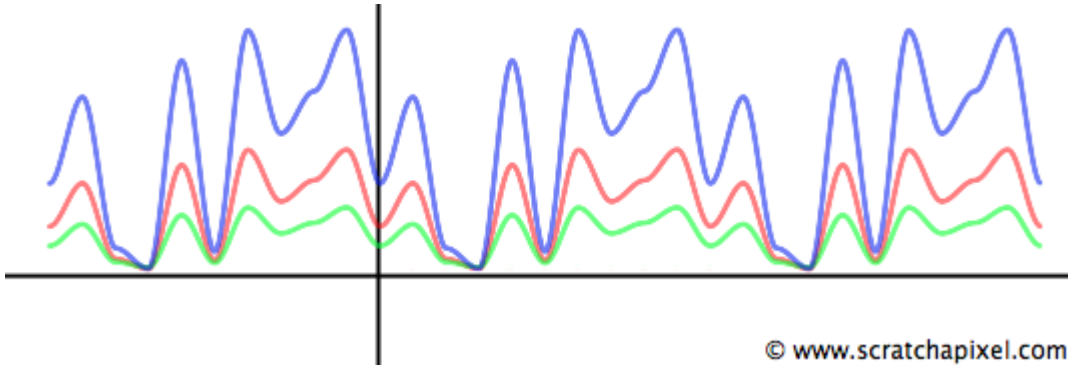
- Controls
  - Frequency: evalNoise(x * freq)



Perlin-like textures

Value noise 1D case

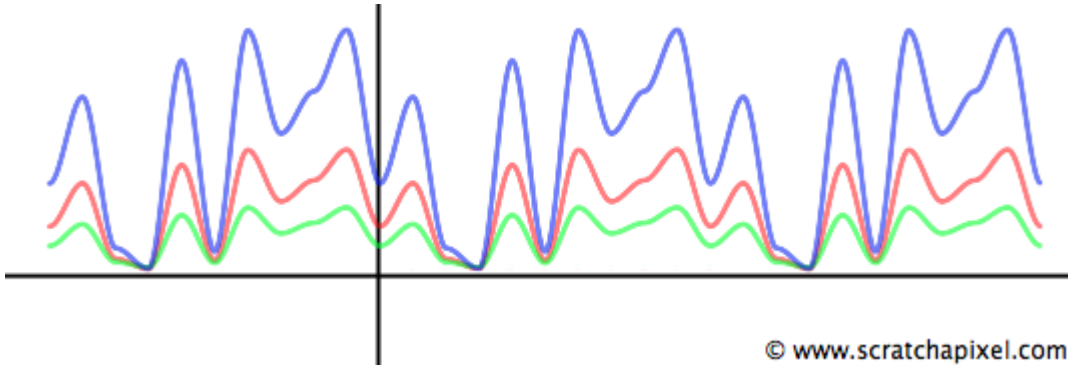
- Controls
  - Frequency: evalNoise(x * freq)
  - Amplitude: evalNoise(x) * amplitude



Perlin-like textures

Value noise 1D case

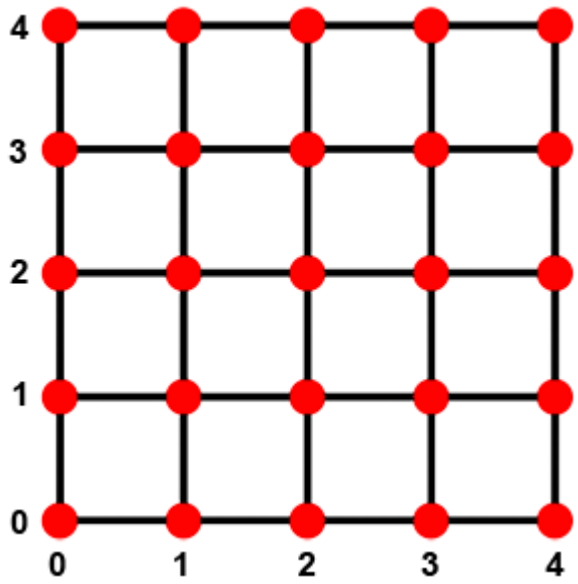
- Controls
  - Frequency: evalNoise(x * freq)
  - Amplitude: evalNoise(x) * amplitude
  - Offsetting: evalNoise(x + offset)



Perlin-like textures

Value noise 2D case

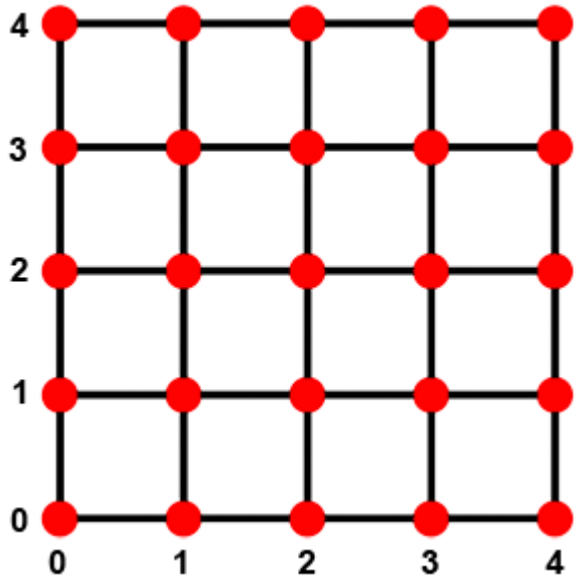
- Same principle, using a 2D grid

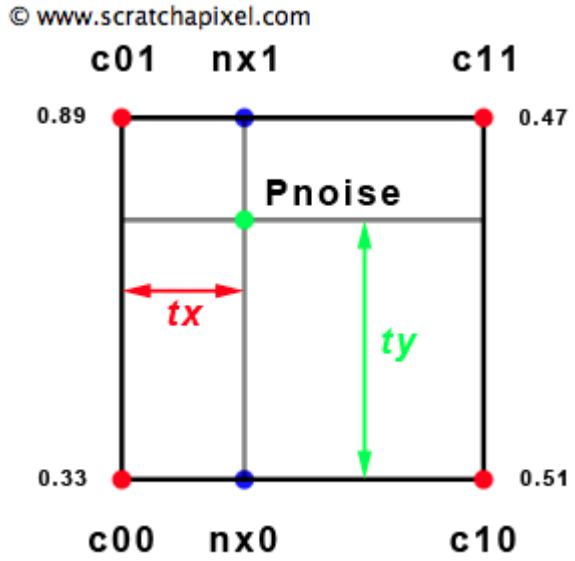


Perlin-like textures

Value noise 2D case

- Same principle, using a 2D grid
- Need 3 interpolations instead of 1



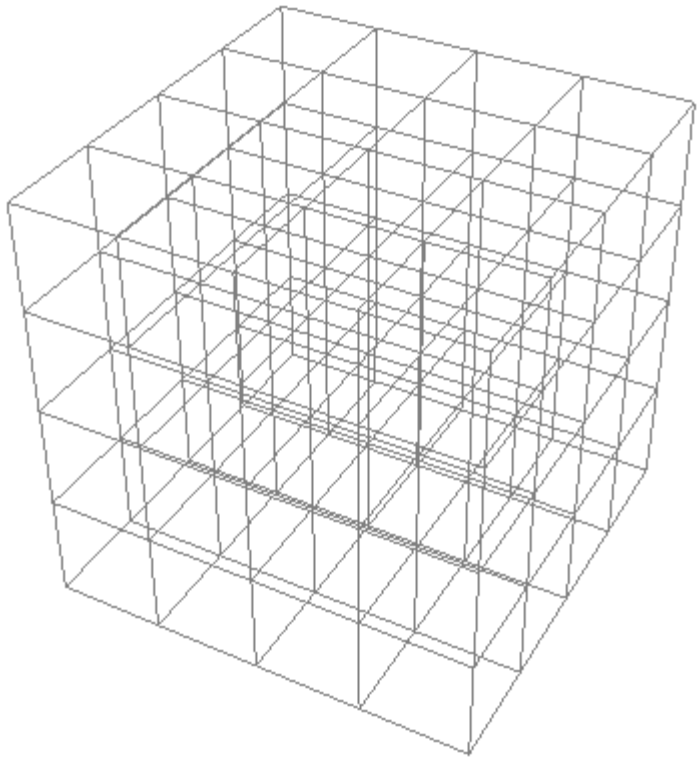


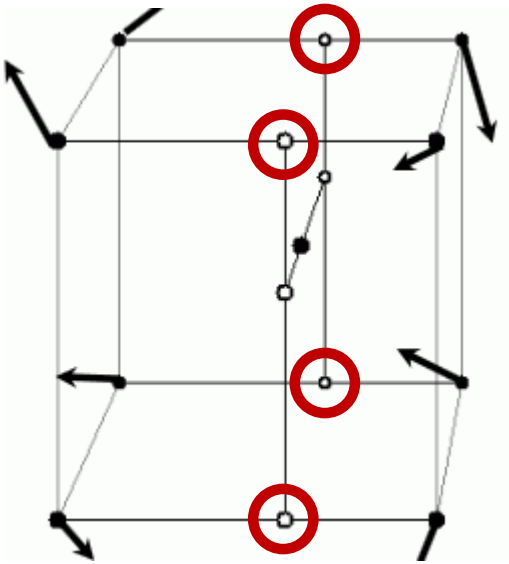
https://www.shadertoy.com/view/lsf3WH

Perlin-like textures

Value noise 3D case

- Same principle, using a 3D grid
- Need 7 interpolations

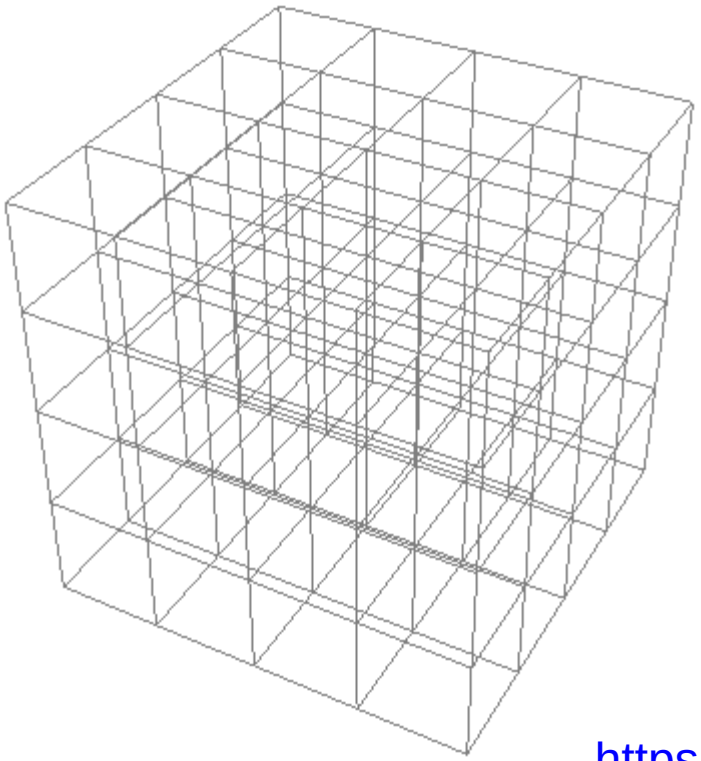


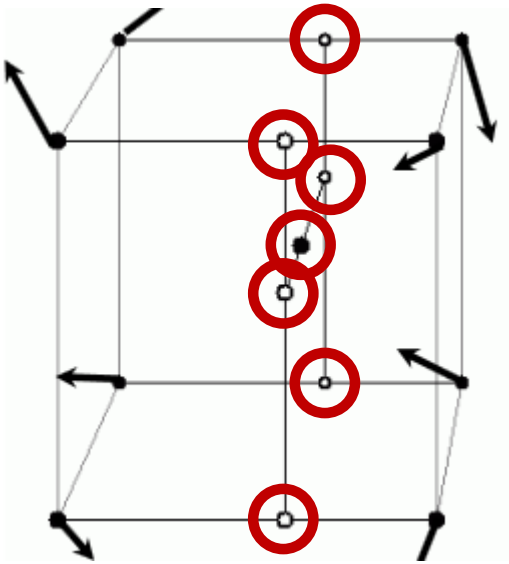


Perlin-like textures

Value noise 3D case

- Same principle, using a 3D grid
- Need 7 interpolations



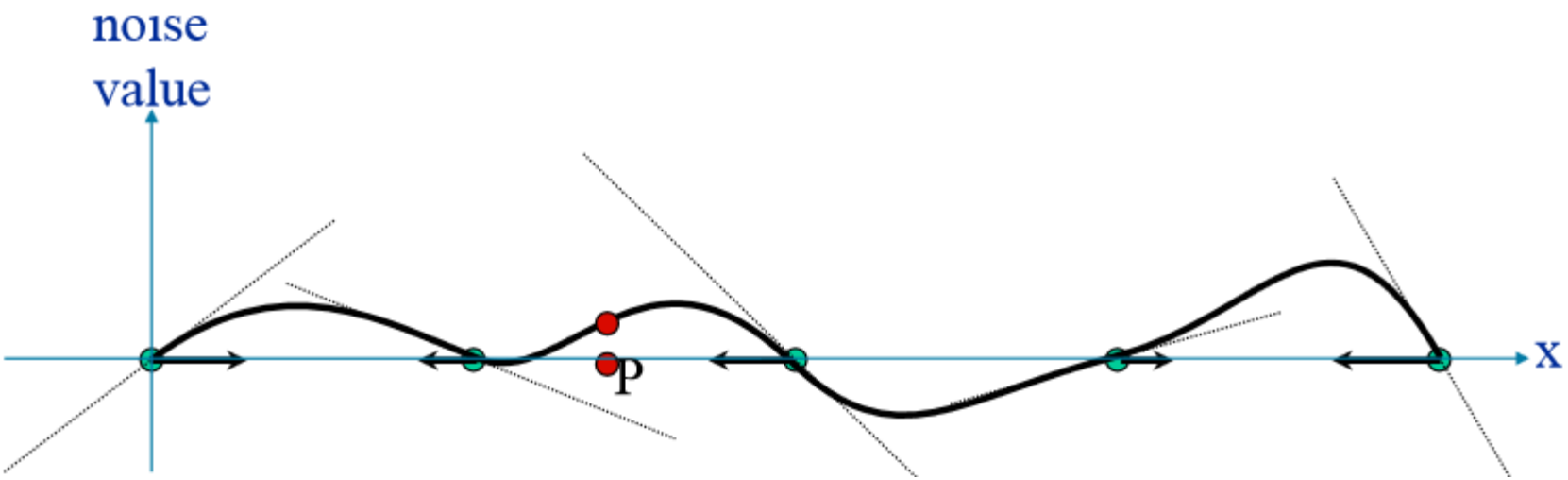


https://www.shadertoy.com/view/4sfGzS

Perlin-like textures

Gradient noise (1D case)

- Instead of distributing random positions:
  - Consider positions at 0
  - Distribute random gradients
  - Interpolate

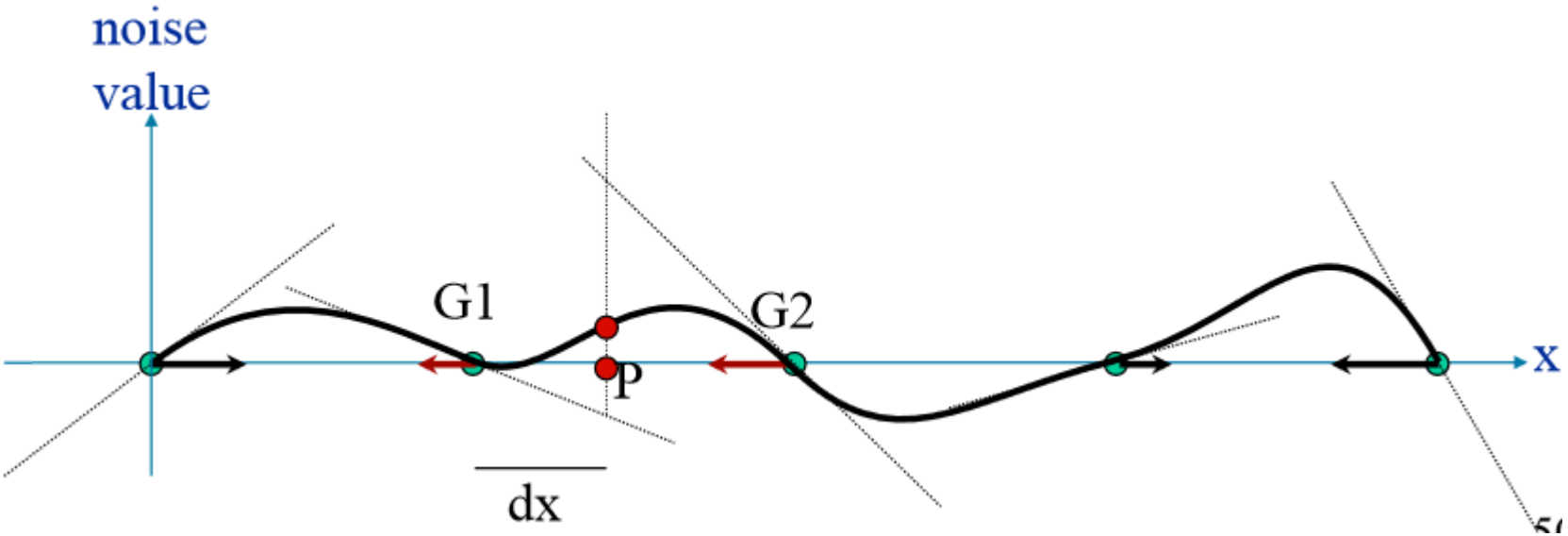


Perlin-like textures

Gradient noise (1D case)

- Instead of distributing random positions:
  - Consider positions at 0
  - Distribute random gradients
  - Interpolate

Available: G1, G2 and dx

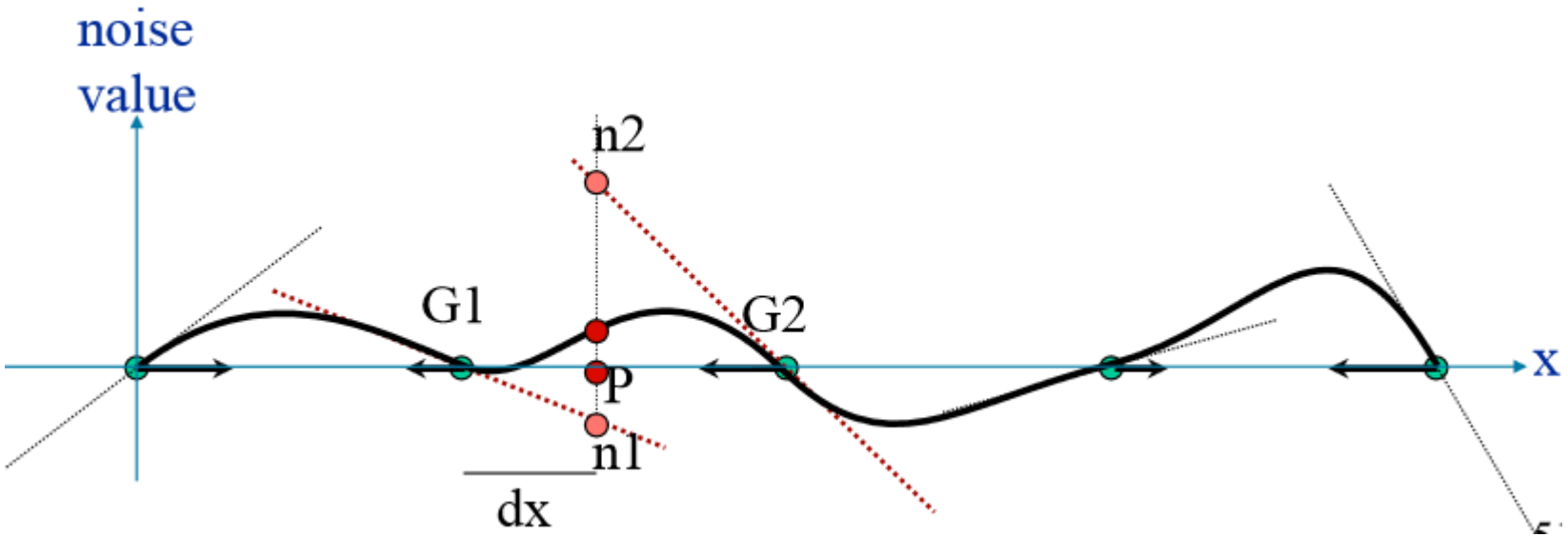


Perlin-like textures

Gradient noise (1D case)

- Instead of distributing random positions:
  - Consider positions at 0
  - Distribute random gradients
  - Interpolate

Available: G1, G2 and dx
n1 = dx*G1
n2 = (dx-1)*G2



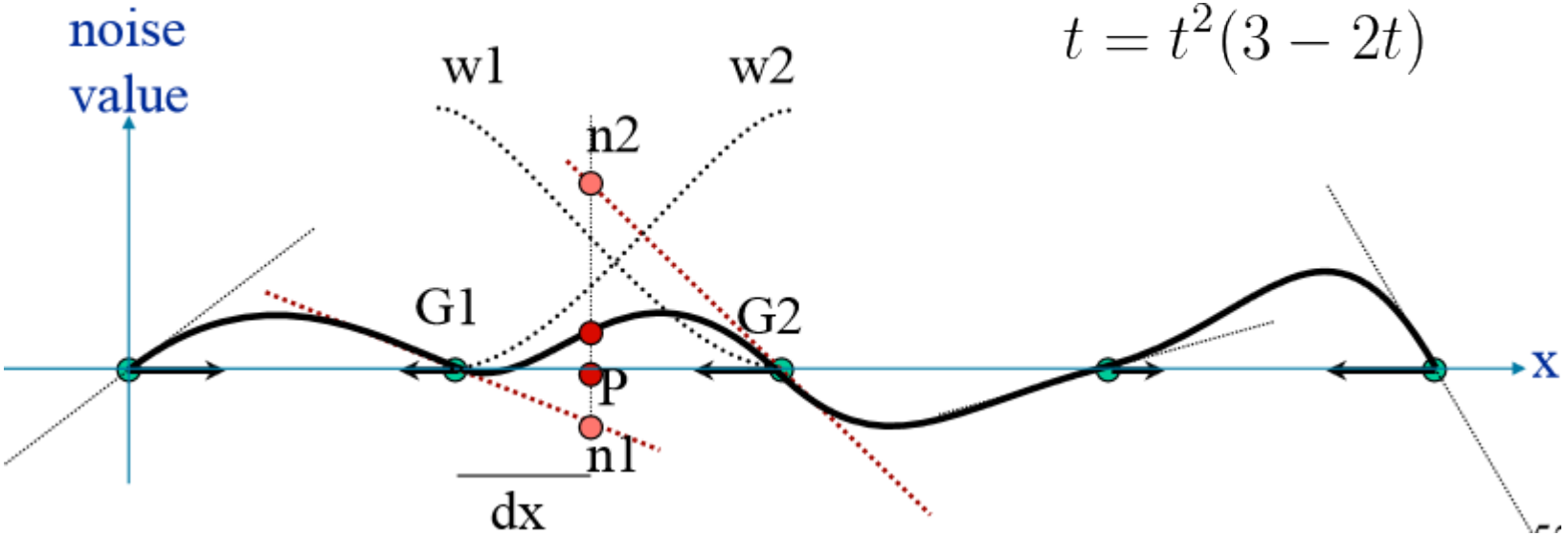
Perlin-like textures

Gradient noise (1D case)

- Instead of distributing random positions:
  - Consider positions at 0
  - Distribute random gradients
  - Interpolate

Available: G1, G2 and dx
n1 = dx*G1
n2 = (dx-1)*G2

P = w1.G1.dx + w2.G2.(dx-1)



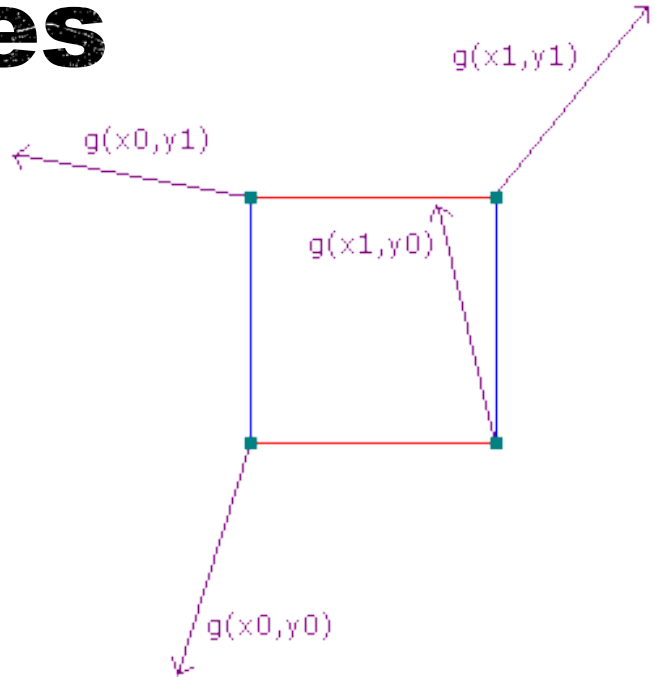
$$t = t^2(3 - 2t)$$

http://en.wikipedia.org/wiki/Gradient_noise

Perlin-like textures

Gradient noise (2D case)

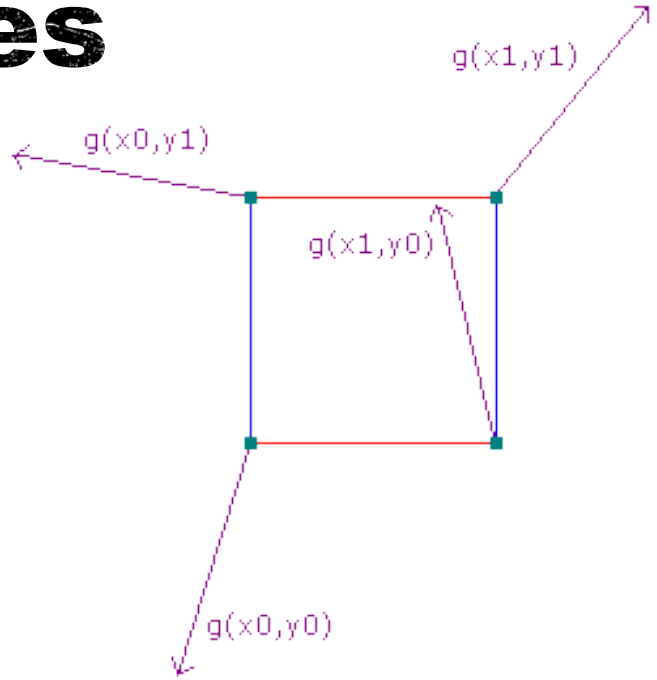
- Same principle
  - 2D gradients on each point

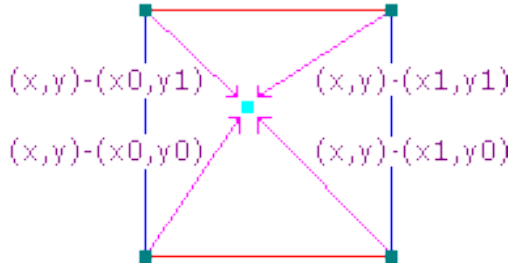


Perlin-like textures

Gradient noise (2D case)

- Same principle
  - 2D gradients on each point
  - Compute positional differences



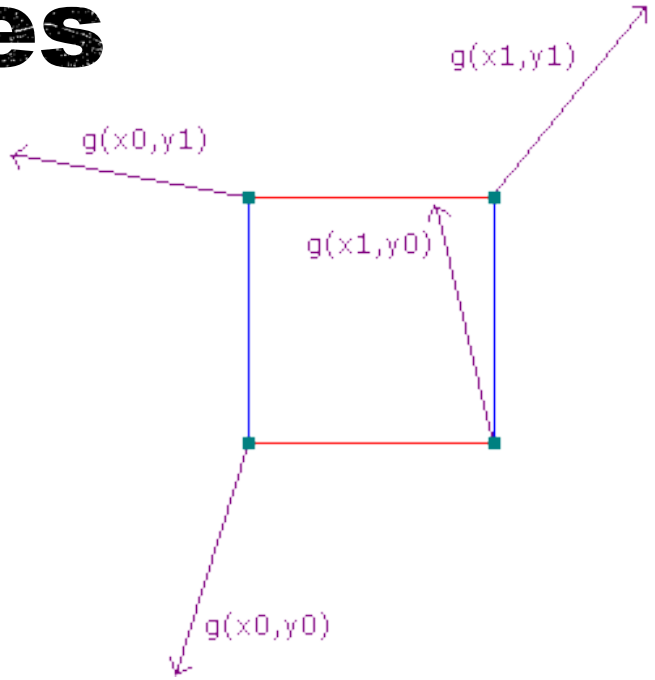


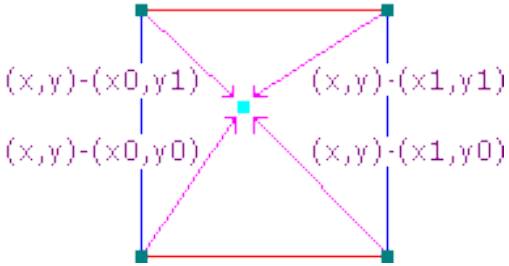
Perlin-like textures

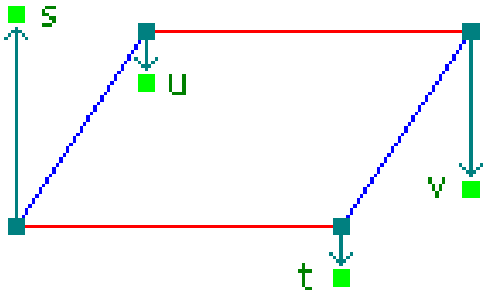
Gradient noise (2D case)

- Same principle
  - 2D gradients on each point
  - Compute positional differences
  - Compute corner values

$$s = g(x_0, y_0) \cdot ((x, y) - (x_0, y_0)),$$
$$t = g(x_1, y_0) \cdot ((x, y) - (x_1, y_0)),$$
$$u = g(x_0, y_1) \cdot ((x, y) - (x_0, y_1)),$$
$$v = g(x_1, y_1) \cdot ((x, y) - (x_1, y_1)).$$





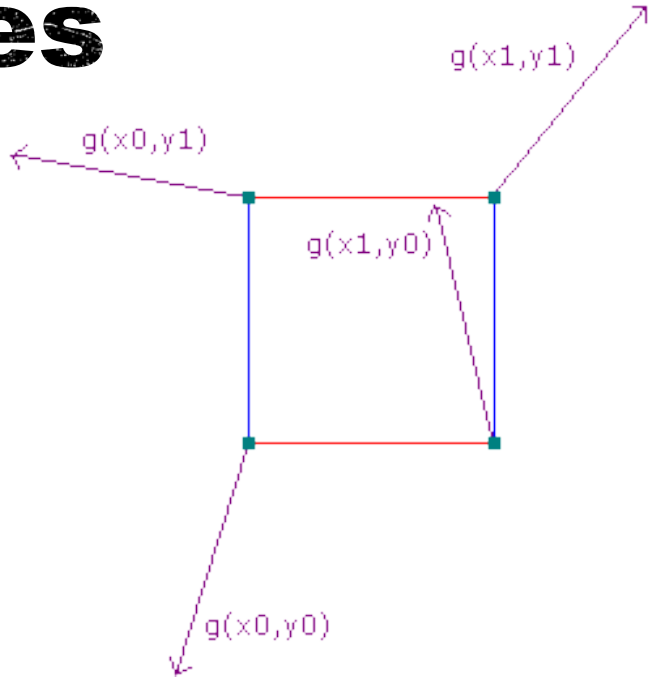


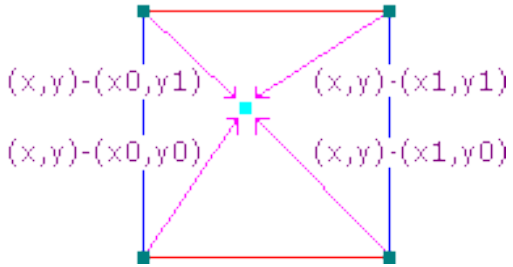
Perlin-like textures

Gradient noise (2D case)

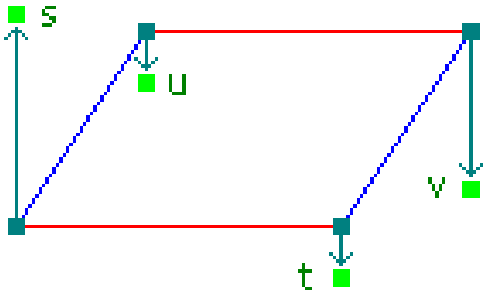
- Same principle
  - 2D gradients on each point
  - Compute positional differences
  - Compute corner values

$$s = g(x_0, y_0) \cdot ((x, y) - (x_0, y_0)),$$
$$t = g(x_1, y_0) \cdot ((x, y) - (x_1, y_0)),$$
$$u = g(x_0, y_1) \cdot ((x, y) - (x_0, y_1)),$$
$$v = g(x_1, y_1) \cdot ((x, y) - (x_1, y_1)).$$





- And interpolate, as before

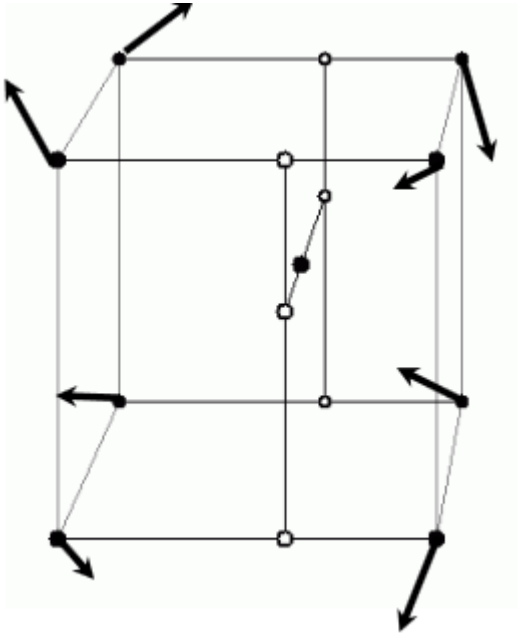


https://www.shadertoy.com/view/XdXGW8

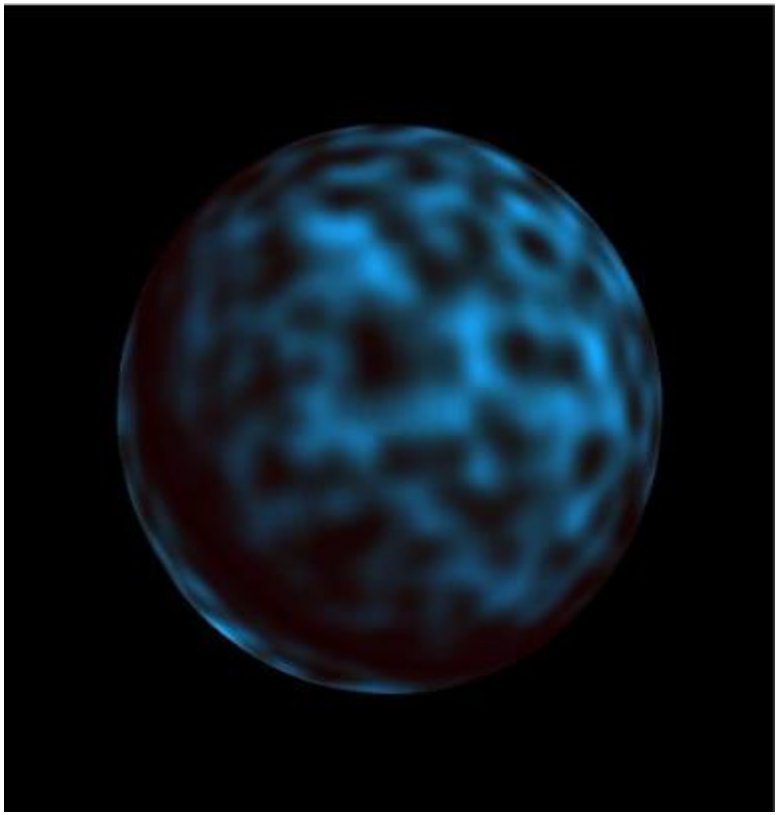
Perlin-like textures

Gradient noise (3D case)

- Same principle… again
  - 3D gradients on each point
  - Compute positional differences
  - Compute corner values
  - And interpolate, as before



Perlin-like textures



Gradient noise (3D case)

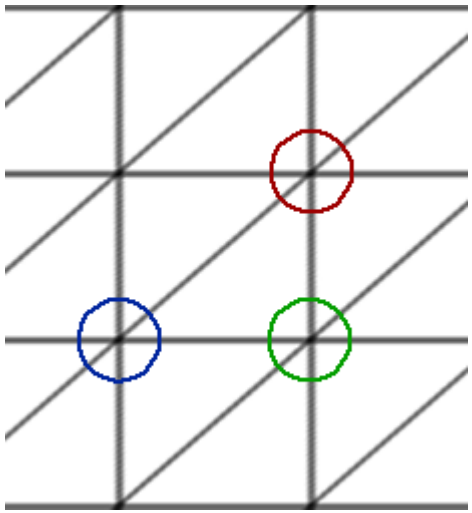
- Same principle… again
  - 3D gradients on each point
  - Compute positional differences
  - Compute corner values
  - And interpolate, as before

- Speeding it up (precompute gradients)
  - Precompute (1D) table of n gradients $G[n]$
  - Precompute (1D) permutation $P[n]$
  - For 3D grid point i, j, k :
    $G(i,j,k) = G[\ (\ i + P[\ (j + P[k]) \bmod n\ ]\ ) \bmod n\ ]$

Perlin-like textures

Gradient noise (3D case)

- Simplex noise
  - Use triangles instead of quads
  - Sum contributions instead of interpolating

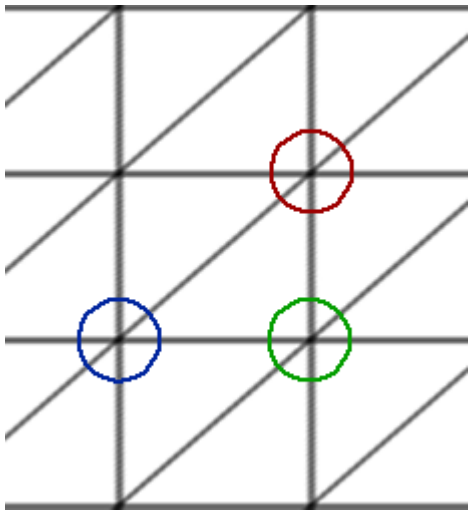


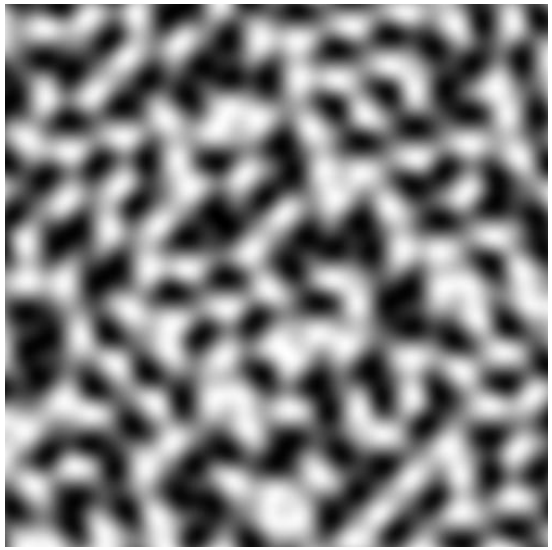
http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

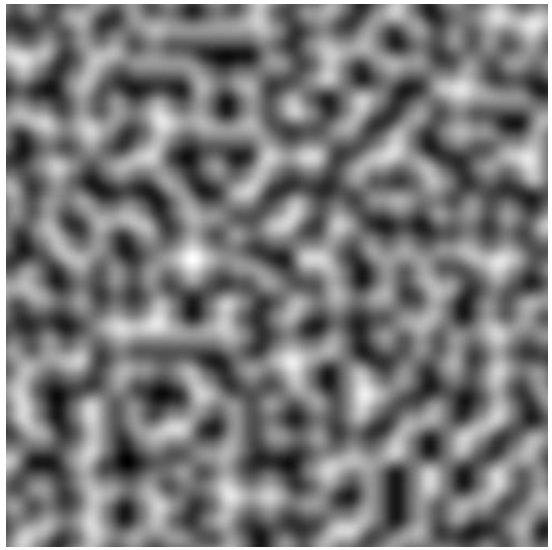
Perlin-like textures

Gradient noise (3D case)

- Simplex noise
  - Use triangles instead of quads
  - Sum contributions instead of interpolating



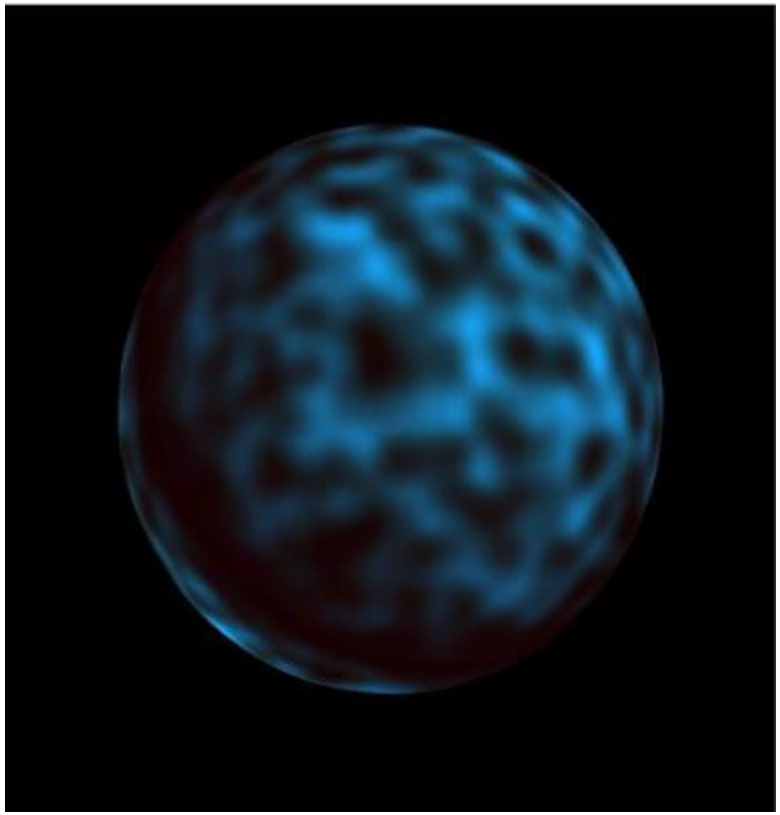




http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

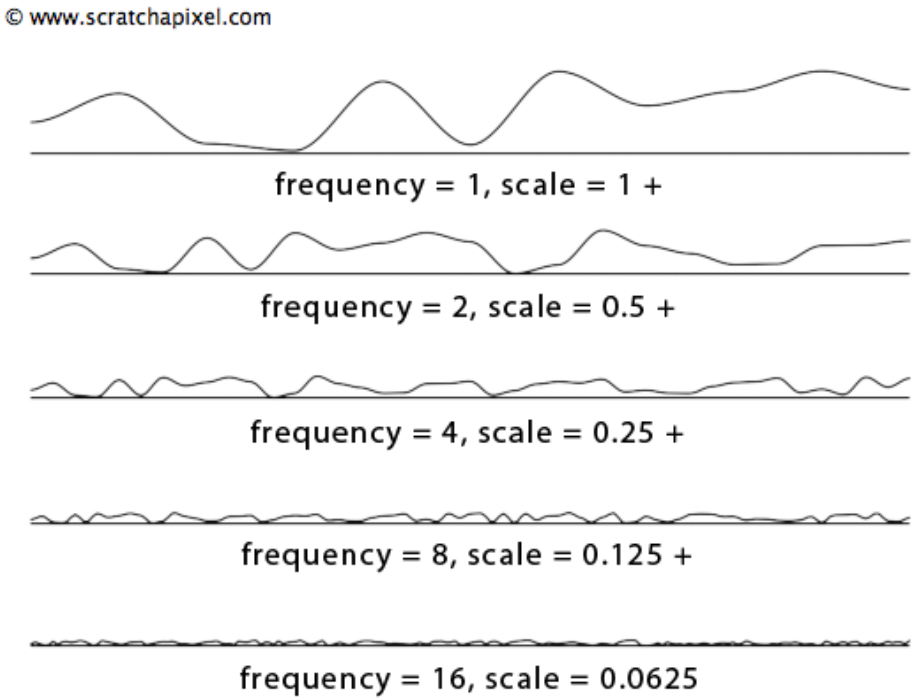
Fractal textures

- Noise at one scale = 1 octave



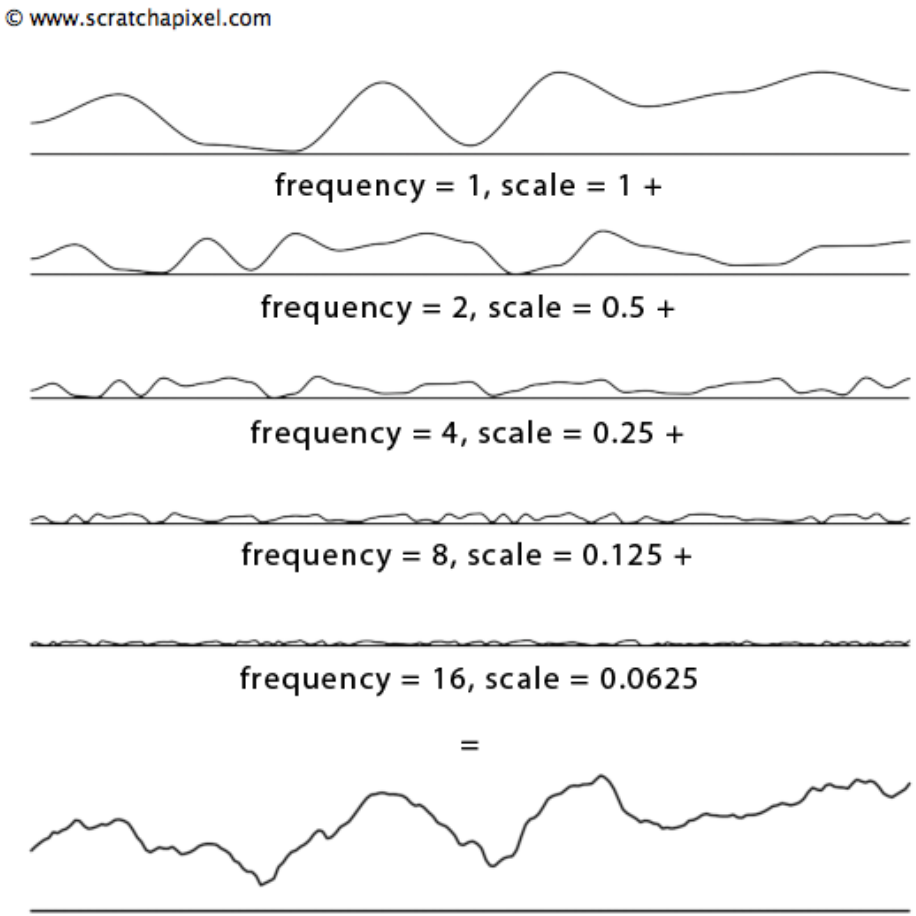
Fractal textures

- Noise at one scale = 1 octave
- Multiple octave usually used
  - Frequency multiplied by 2 each time
  - Hence the name octave
  - Different amplitudes too



Fractal textures

- Noise at one scale = 1 octave
- Multiple octave usually used
  - Frequency multiplied by 2 each time
  - Hence the name octave
  - Different amplitudes too
- Sum of all noises =
  - Fractal noise



Fractal textures

- Computing the ith noise texture:

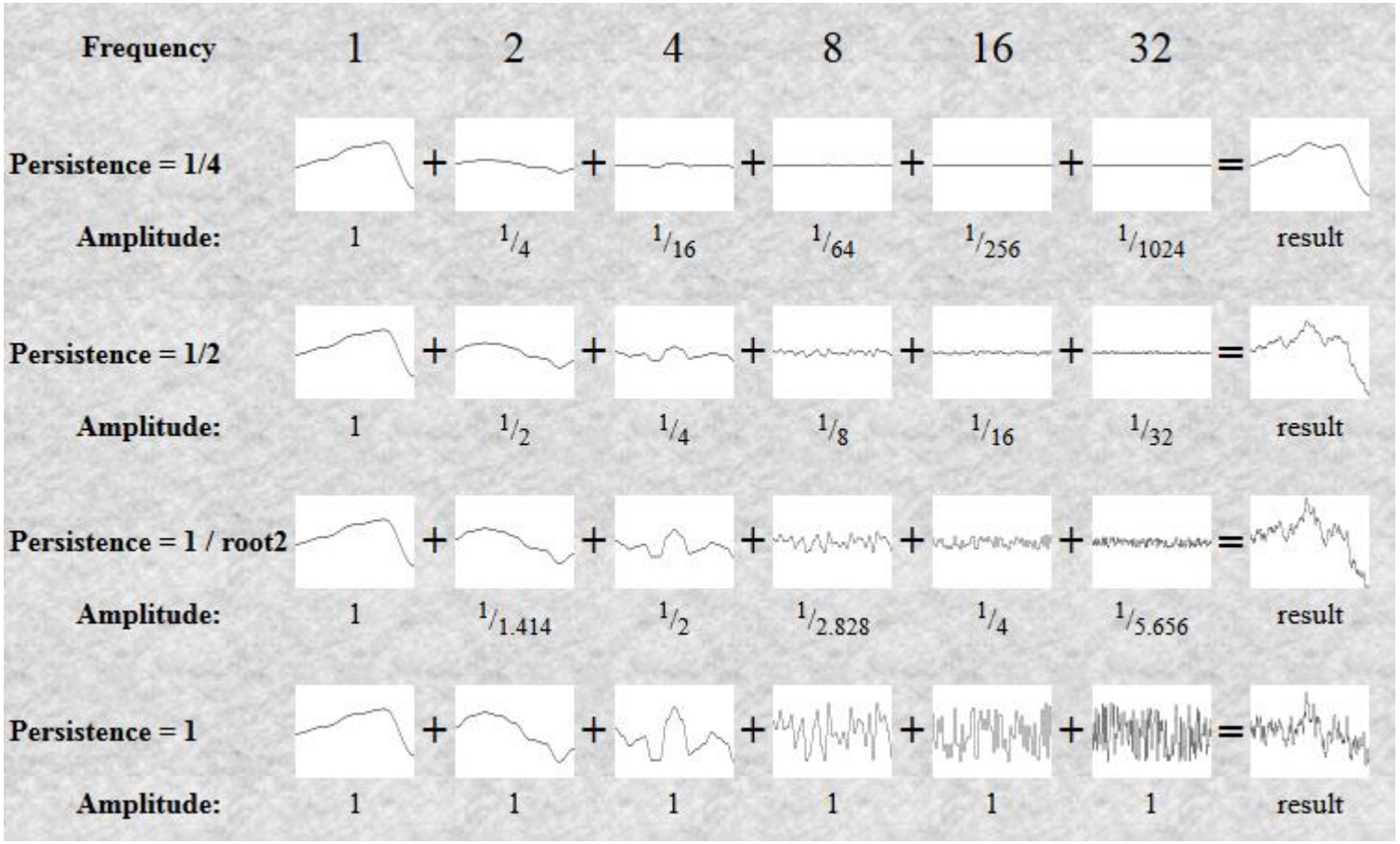
$$frequency = 2^i$$
$$amplitude = persistence^i$$

Fractal textures

- Computing the ith noise texture:

$$frequency = 2^i$$
$$amplitude = persistence^i$$

  - Persistence controls the relation between frequency and amplitude:

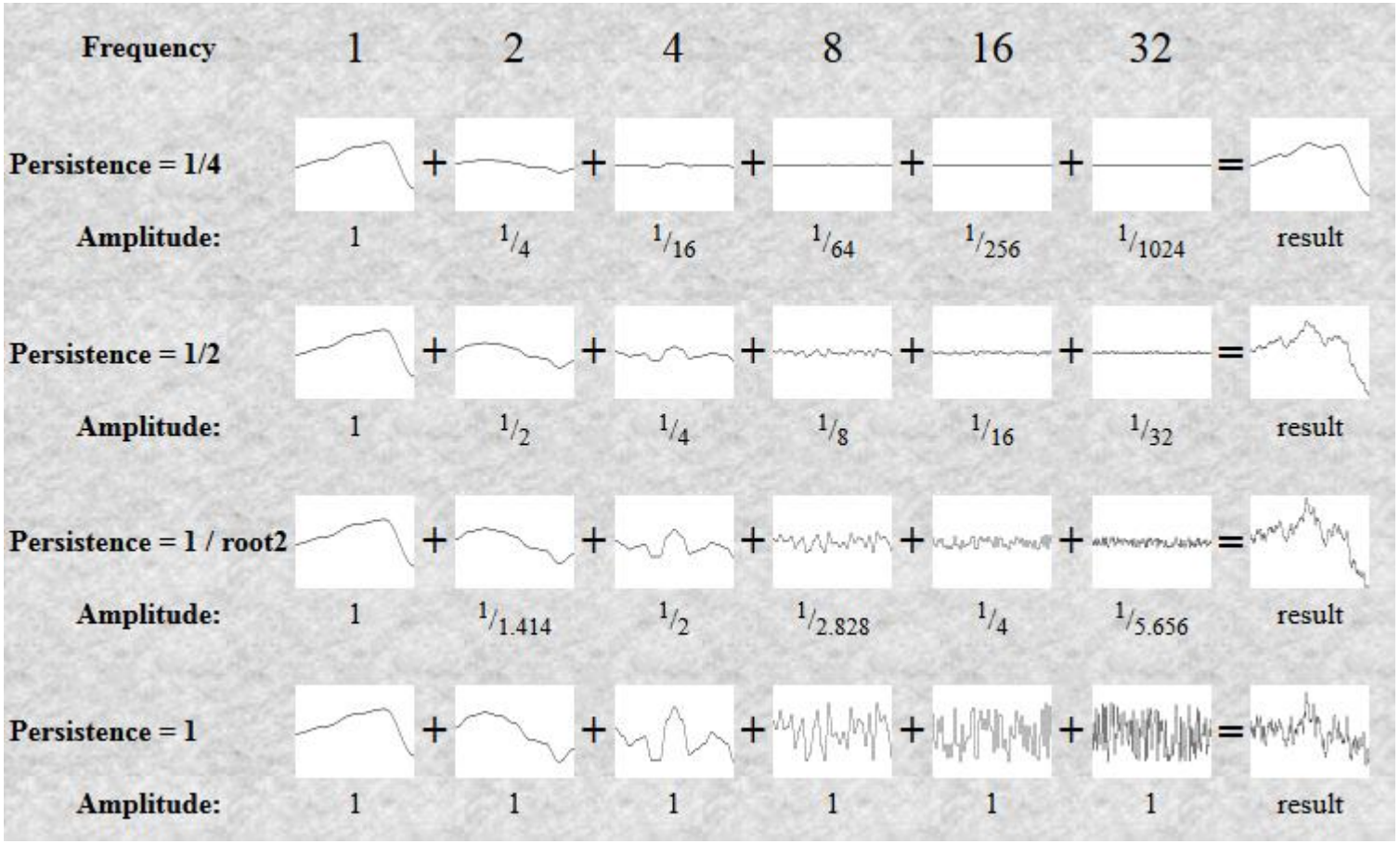


Fractal textures

- Computing the ith noise texture:

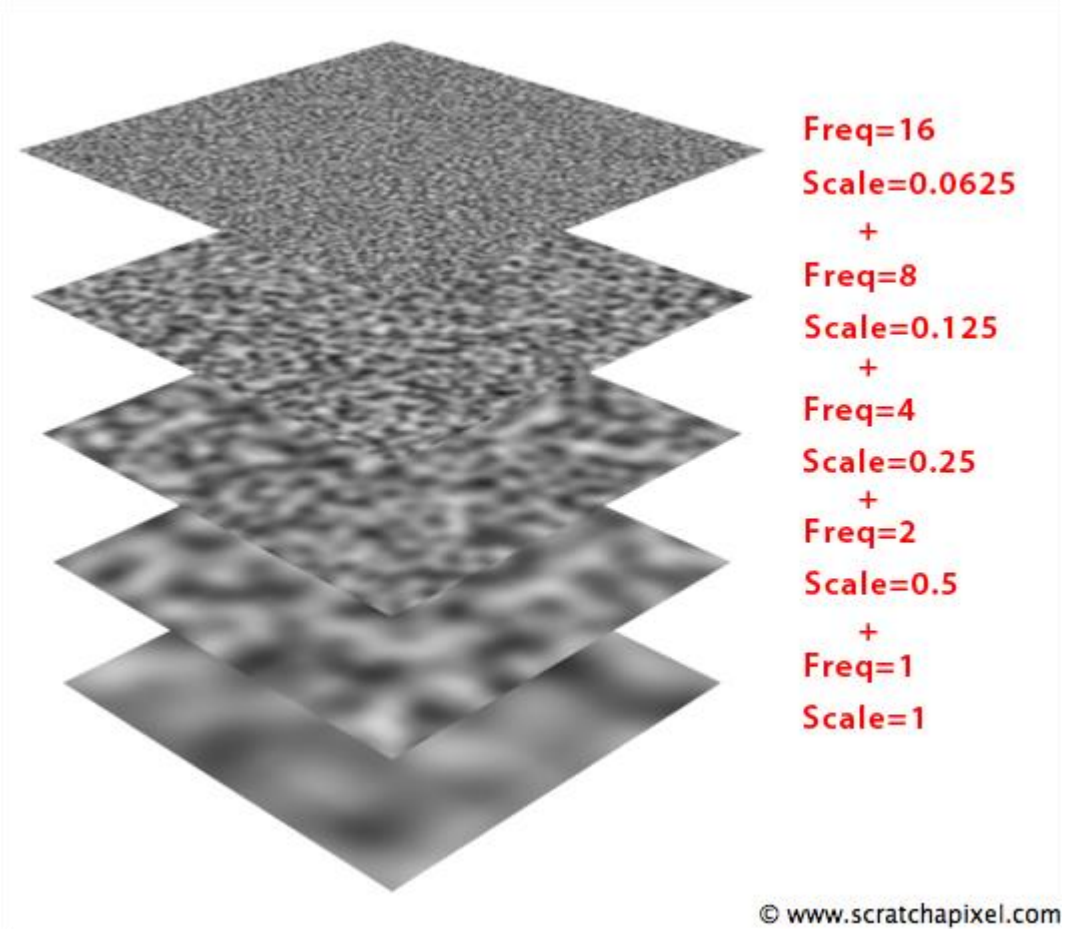
$$frequency = 2^i$$
$$amplitude = persistence^i$$

  - Persistence controls the relation between frequency and amplitude:



Fractal textures

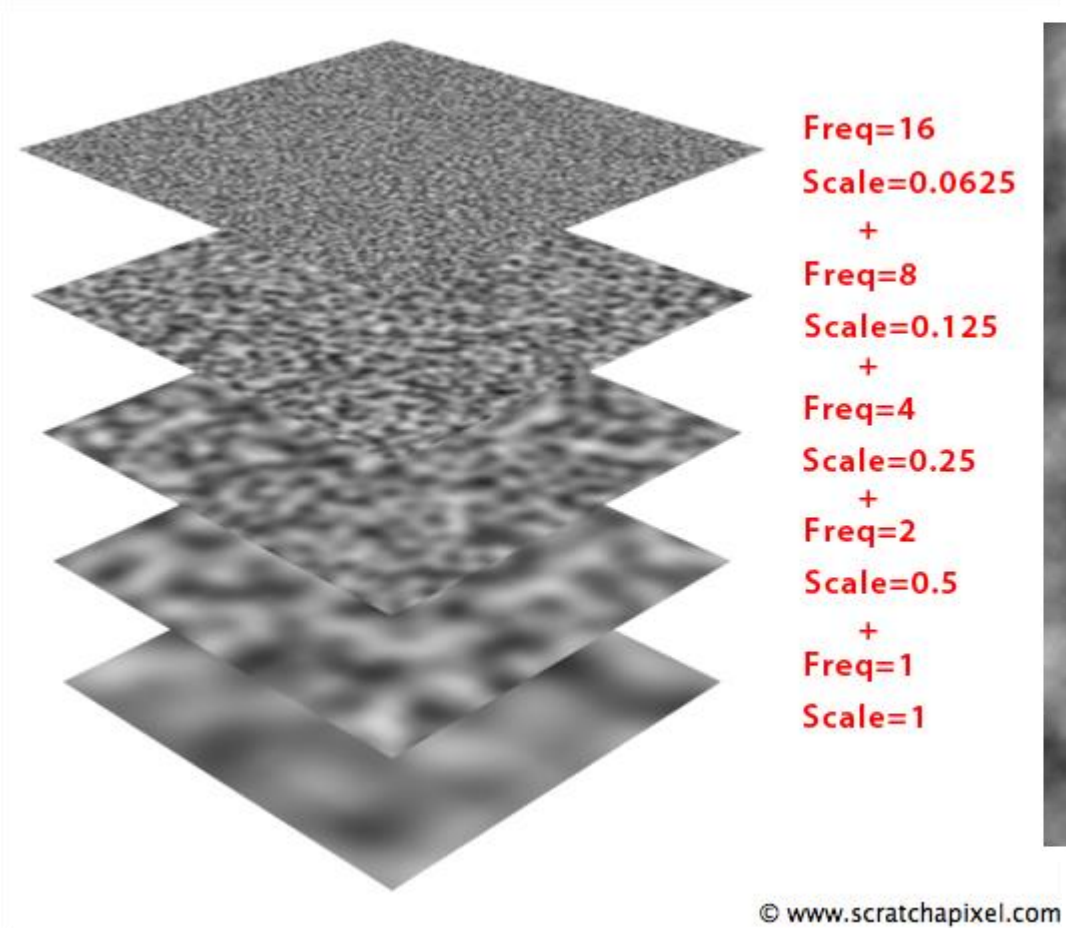
- 2D example

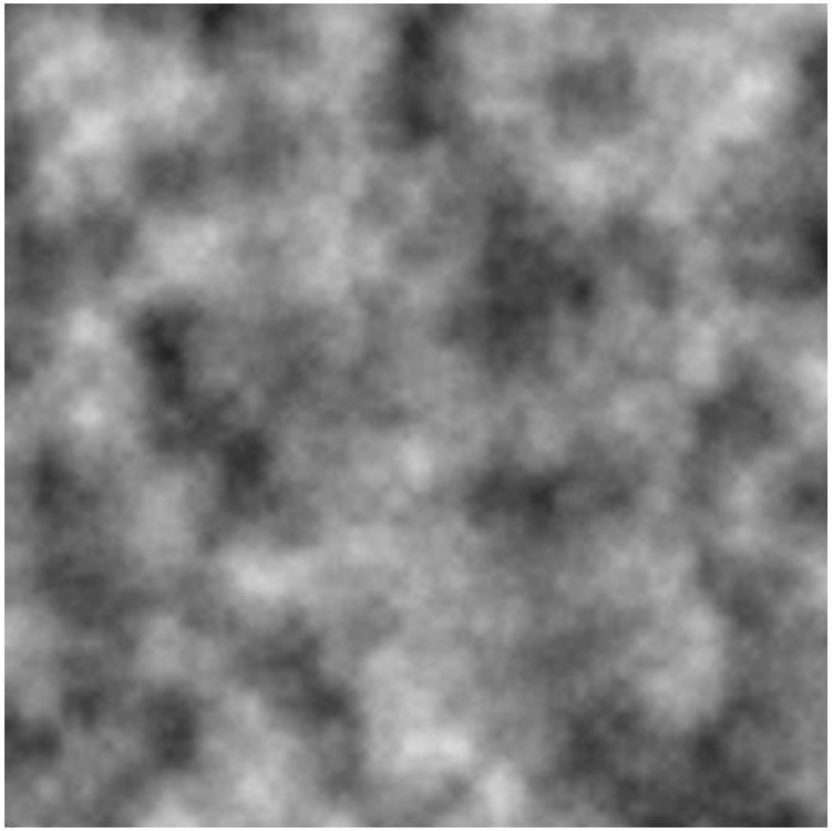


Fractal textures

- 2D example

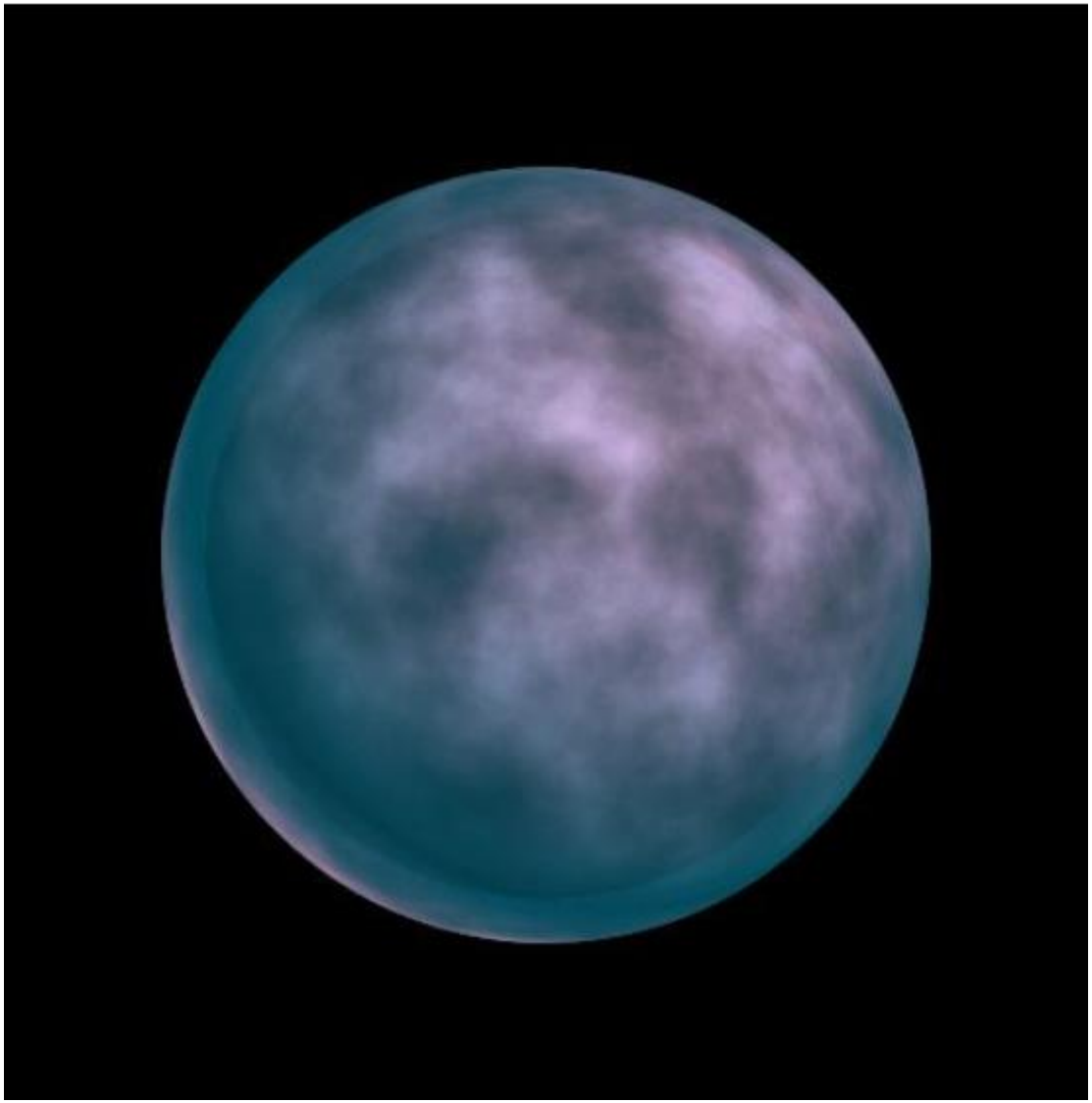
Each octave f has weight 1/f





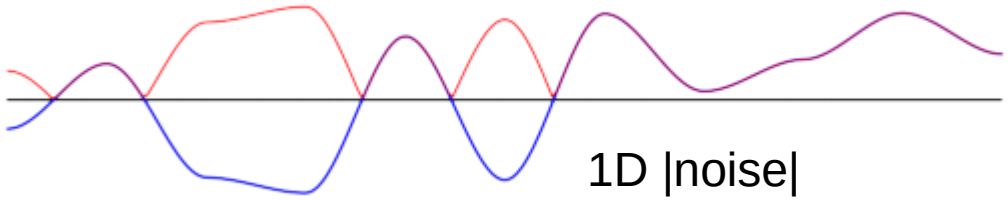
Fractal textures

- 3D example Sum 1/f(noise)



Fractal textures

- Taking the absolute value of the noise Sum 1/f(|noise|)

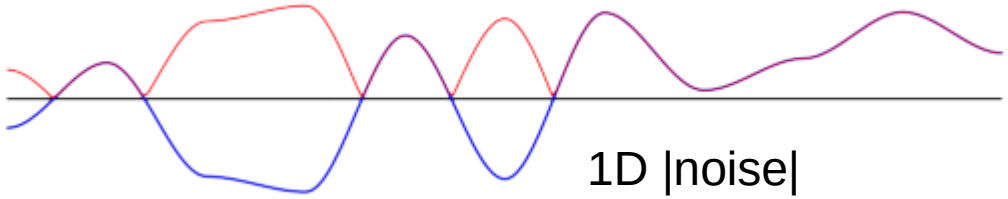


1D |noise|

Creates C0 discontinuities

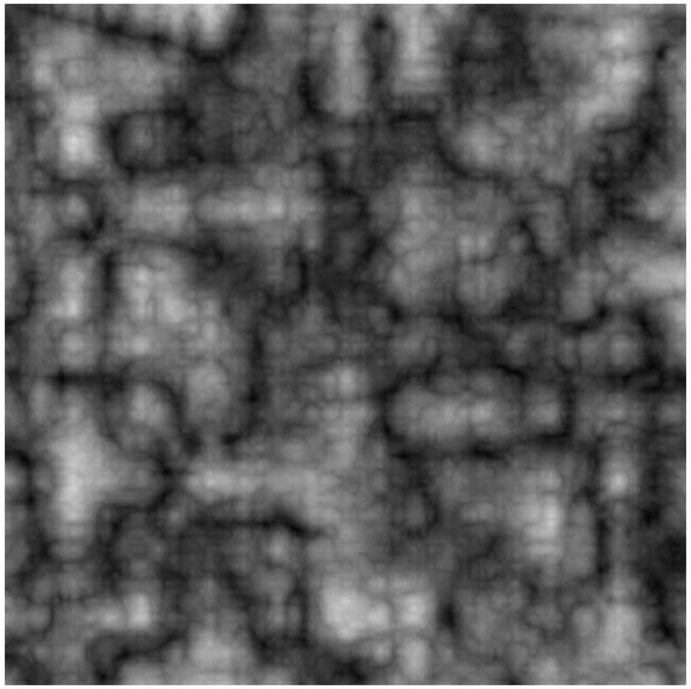
Fractal textures

- Taking the absolute value of the noise Sum 1/f(|noise|)



1D |noise|

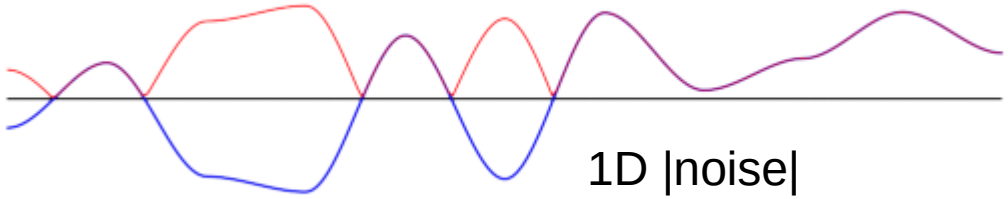
Creates C0 discontinuities



2D |noise|

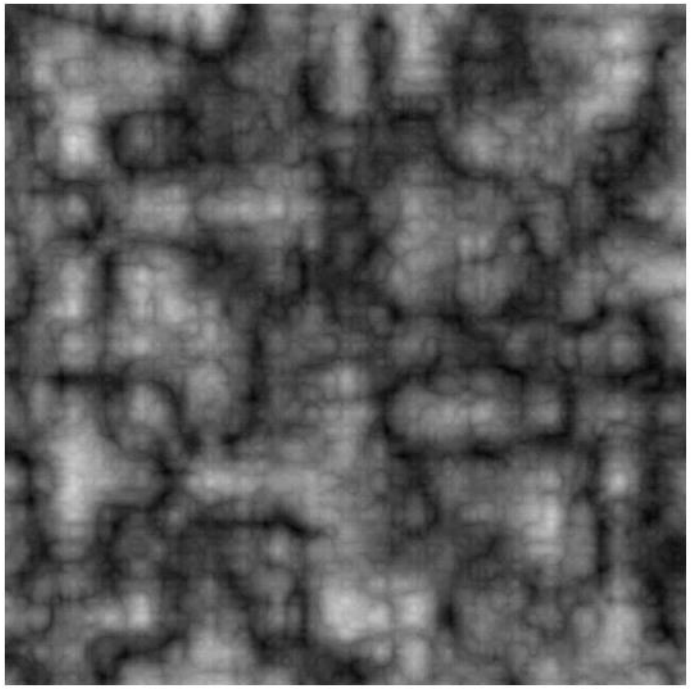
Fractal textures

- Taking the absolute value of the noise Sum 1/f(|noise|)



1D |noise|

Creates C0 discontinuities

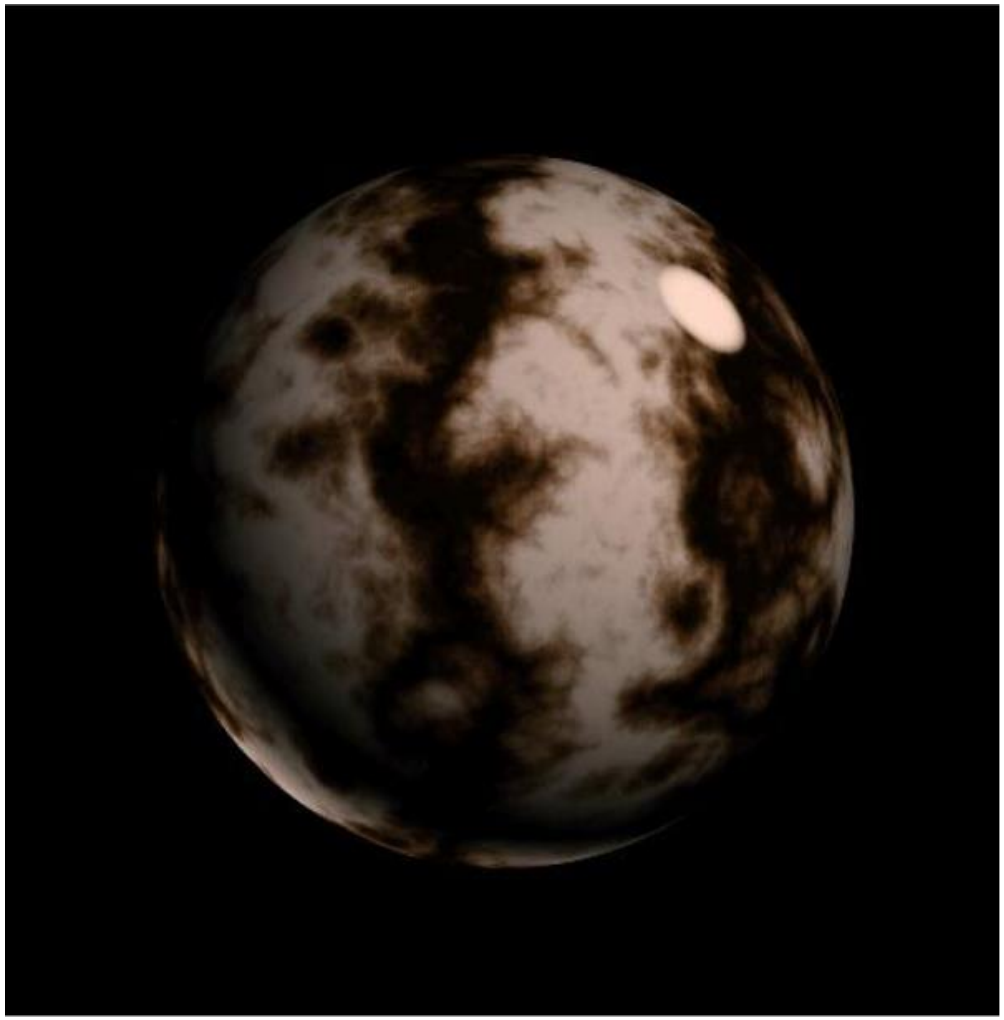


2D |noise|

3D |noise|

Fractal textures

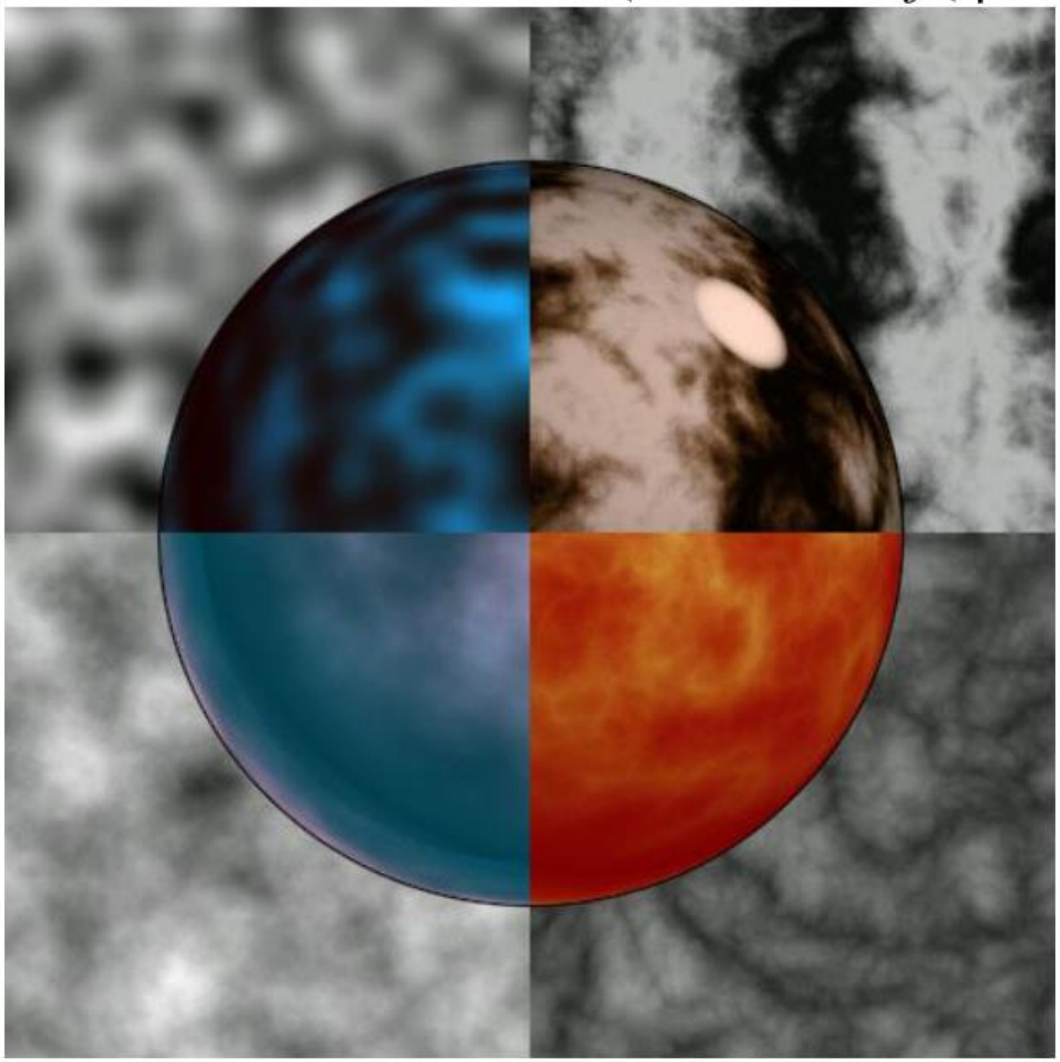
- Even better: Sin (x + Sum 1/f(noise))



Fractal textures

- *noise*

$sin(x + \text{sum } 1/f(\ |noise|\ ))$



sum 1/*f*(*noise*)

sum 1/*f*(|*noise*|)

Fractal textures

- Marble

Sin (x + Sum 1/f(noise))
=
Colormap(Sin (x + turbulence))

- Wood

Colormap(Sin (radius + turbulence))

Fractal textures

- Texture
- Terrain
- Clouds
- Fire

- But also,
- Density for distributing elements
  - Forest
  - Plants
  - Etc

References

- MIT:
  - http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-837-computer-graphics-fall-2012/lecture-notes/
- Standford:
  - http://candela.stanford.edu/cs348b-14/doku.php
- Siggraph:
  - http://blog.selfshadow.com/publications/s2014-shading-course/
  - http://blog.selfshadow.com/publications/s2013-shading-course/
- Image synthesis & OpenGL:
  - http://romain.vergne.free.fr/blog/?page_id=97
- Path tracing and global illum:
  - http://www.graphics.stanford.edu/courses/cs348b-01/course29.hanrahan.pdf
  - http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/realistic_raytracing.html
- GLSL / Shadertoy:
  - https://www.opengl.org/documentation/glsl/
  - https://www.shadertoy.com/
  - http://www.iquilezles.org/
- http://fileadmin.cs.lth.se/cs/Education/EDAN30/lectures/L2-rt.pdf
- http://csokavar.hu/raytrace/imm6392.pdf
- http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/realistic_raytracing.html