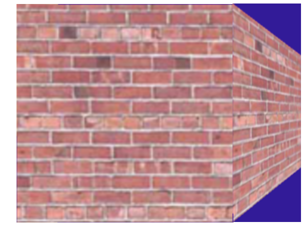
EECS 487: Interactive Computer Graphics

Lecture 26:

- Bump mapping
- Solid and procedural texture

Bump Mapping



2D texture map looks unrealistically smooth across different material, especially at low viewing angle

Fool the human viewer:

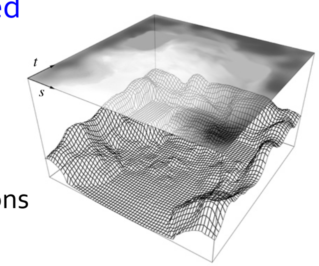
- perception of shape is determined by shading, which is determined by surface normal
- use texture map to perturb the surface normal per fragment
  - does not actually alter the geometry of the surface
  - shade each fragment using the perturbed normal as if the surface were a different shape

Sphere w/Diffuse Texture | Swirly Bump Map | Sphere w/Diffuse Texture & Bump Map

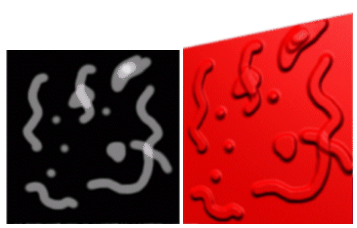
Bump Mapping

Treat the texture as a single-valued height function (height map)

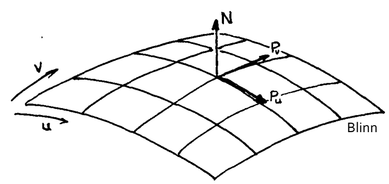
- grayscale image stores height: black, high area; white, low (or vice versa)
- difference in heights determines how much to perturb $\mathbf{n}$ in the (u, v) directions of a parametric surface
  - $\partial b/\partial u = b_u = (h[s+1, t] - h[s-1, t])/ds$
  - $\partial b/\partial v = b_v = (h[s, t+1] - h[s, t-1])/dt$
- compute a new, perturbed normal from (b_u, b_v)

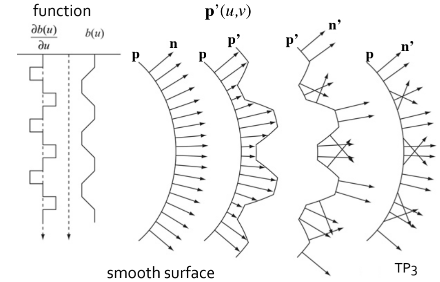


TP3



Computing Perturbed Normal





TP3

$\mathbf{n} = \mathbf{p}_u \times \mathbf{p}_v$

$\mathbf{p}_u = \frac{\partial}{\partial u}\mathbf{p}(u,v),\ \mathbf{p}_v = \frac{\partial}{\partial v}\mathbf{p}(u,v)$

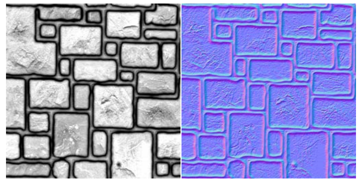
$\mathbf{p}' = \mathbf{p}(u,v) + b(u,v)\mathbf{n} \Leftarrow$ perturbed surface

$\mathbf{p}'_u = \frac{\partial}{\partial u}(\mathbf{p}(u,v) + b(u,v)\mathbf{n}) = \mathbf{p}_u + b_u\mathbf{n} + b(u,v)\mathbf{n}_u = \mathbf{p}_u + b_u\mathbf{n}$

$\mathbf{p}'_v = \frac{\partial}{\partial v}(\mathbf{p}(u,v) + b(u,v)\mathbf{n}) = \mathbf{p}_v + b_v\mathbf{n} + b(u,v)\mathbf{n}_v = \mathbf{p}_v + b_v\mathbf{n}$

0: $\mathbf{n} \perp \mathbf{p}$

Perturbed Normal

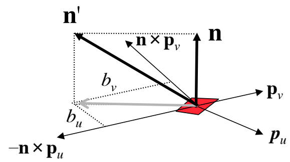


$\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v \Leftarrow$ normal of perturbed surface

$$\mathbf{p}'_u = \mathbf{p}_u + b_u\mathbf{n}$$
$$\mathbf{p}'_v = \mathbf{p}_v + b_v\mathbf{n}$$
$$\mathbf{n}' = (\mathbf{p}_u + b_u\mathbf{n}) \times (\mathbf{p}_v + b_v\mathbf{n})$$
$$= \mathbf{p}_u \times \mathbf{p}_v + b_u(\mathbf{n} \times \mathbf{p}_v) + b_v(\mathbf{p}_u \times \mathbf{n}) + b_u b_v(\mathbf{n} \times \mathbf{n})$$
$$= \mathbf{n} + b_u(\mathbf{n} \times \mathbf{p}_v) - b_v(\mathbf{n} \times \mathbf{p}_u)$$

Recall:

$$\mathbf{a} \times (k\mathbf{b} + \mathbf{c}) = k(\mathbf{a} \times \mathbf{b}) + (\mathbf{a} \times \mathbf{c})$$
$$\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$$



Bump Map vs. Normal Map

Computing **n'** requires the height samples from 4 neighbors

- each sample by itself doesn't perturb the normal
- an all-white height map renders exactly the same as an all-black height map

Instead of encoding only the height of a point, a normal map encodes the normal of the desired surface, in the tangent space of the surface at the point

- can be obtained from:
  - a high-resolution 3D model
  - photos (http://zarria.net/nrmphoto/nrmphoto.html)
  - a height-map (with more complex offline computation of perturbed normals)
  - filtered color texture (Photoshop, Blender, Gimp, etc. with plugin)

[Hastings-Trew]

Normal Map

Interpret the RGB values per texel as the perturbed normal, not height value

height map

[Hastings-Trew]

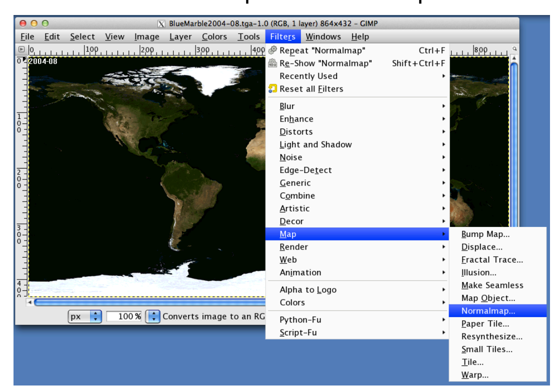
normal map

$(n_x, n_y, n_z) = (r, g, b)$

Normal Map Creation on Gimp

On Mac OS X, run Gimp-2.6.11 (not 2.8.4)

- load RGB file, then select Filters→Map→Normalmap



For Windows, see http://code.google.com/p/gimp-normalmap/

Normal Mapping: Complications

1. Normalized normals range $[-1, 1]$, but RGB values range $[0,1]$, convert normals by $\mathbf{n}' = (\mathbf{n}+1)/2$

   Values in normal map must be converted back before use: $\mathbf{n} = \mathbf{n}'*2-1$

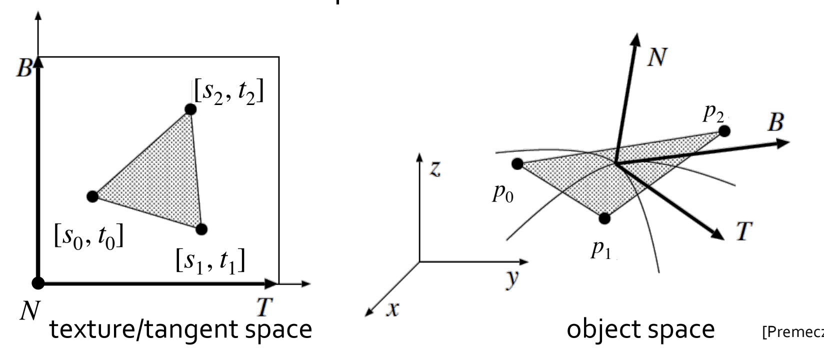
2. Normals are in object space, so normals must be transformed whenever object is transformed

   Instead, most implementations store normals in tangent space, but then light and view vectors must be transformed to tangent space

Tangent Space

Is a coordinate system attached to the local surface with basis vectors comprising the normal vector ($\mathbf{N}$), perpendicular to the surface, and two vectors tangent to the surface: the tangent ($\mathbf{T}$) and bitangent ($\mathbf{B}$)

We want $\mathbf{T}$ and $\mathbf{B}$ to span our texture:



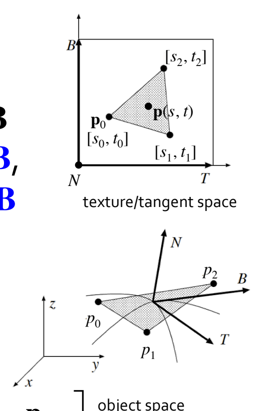
[Premecz]

Tangent Space

A point $\mathbf{p}(s, t) = \mathbf{p}_0 + (s-s_0)\mathbf{T} + (t-t_0)\mathbf{B}$
3D vectors $\mathbf{p}_1-\mathbf{p}_0 = (s_1-s_0)\mathbf{T} + (t_1-t_0)\mathbf{B}$,
and $\mathbf{p}_2-\mathbf{p}_0 = (s_2-s_0)\mathbf{T} + (t_2-t_0)\mathbf{B}$
Let $\Delta s_i = (s_i-s_0)$ and $\Delta t_i = (t_i-t_0)$, then

$$\begin{bmatrix} \mathbf{p}_1-\mathbf{p}_0 \\ \mathbf{p}_2-\mathbf{p}_0 \end{bmatrix} = \begin{bmatrix} \Delta s_1 & \Delta t_1 \\ \Delta s_2 & \Delta t_2 \end{bmatrix} \begin{bmatrix} \mathbf{T} \\ \mathbf{B} \end{bmatrix}$$

$$\begin{bmatrix} \mathbf{T} \\ \mathbf{B} \end{bmatrix} = \frac{1}{\Delta s_1 \Delta t_2 - \Delta s_2 \Delta t_1} \begin{bmatrix} \Delta t_2 & -\Delta t_1 \\ -\Delta s_2 & \Delta s_1 \end{bmatrix} \begin{bmatrix} \mathbf{p}_1-\mathbf{p}_0 \\ \mathbf{p}_2-\mathbf{p}_0 \end{bmatrix}$$



in texture space, $\mathbf{T}$, $\mathbf{B}$, $\mathbf{N}$ are orthonormal, but not necessarily so in object space, use Gram-Schmidt Orthogonalization: $\mathbf{B}' = \mathbf{N}\times\mathbf{T}$; $\mathbf{T}' = \mathbf{B}'\times\mathbf{N}$

[Premecz, Lengyel]

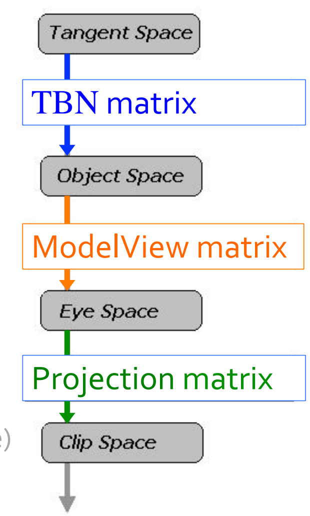
Tangent to Object Space

Given $\mathbf{T}'$, $\mathbf{B}'$, $\mathbf{N}$ orthonormal, $[\mathbf{T}'\mathbf{B}'\mathbf{N}]$ matrix transforms from tangent to object space ($[\mathbf{T}'\mathbf{B}'\mathbf{N}]^T$ matrix in Direct3D)

[We assume $\mathbf{TBN}$ orthonormal in the figure and in subsequent slides and we drop the "prime" sign]

Bitangent is sometimes called binormal, second normal, which is applicable to curves, but not to surfaces (see lecture on Frenet frame)

See also: http://www.terathon.com/code/tangent.html

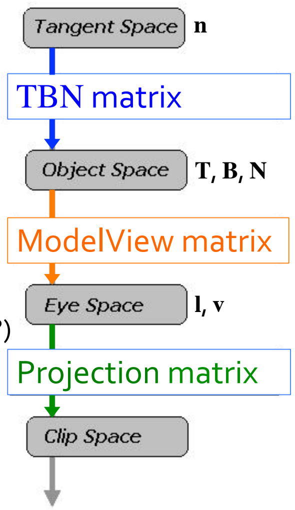


[Lengyel, Dreijer Madsen]

Lighting Computation

What we have:
- per-texel **n** in tangent space stored in normal map
- **T**, **N** in object space
- **l**, **v** in eye space

To compute lighting with normal map in tangent space:
1. transform **l** and **v** to tangent space (how?)
2. sample per-texel **n** from normal map
3. compute lighting in tangent space



[Lengyel, Dreijer Madsen]

Tangent-Space Lighting

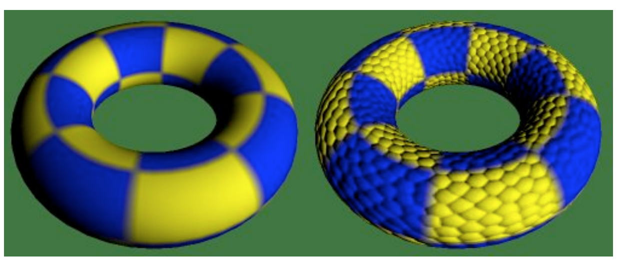
In app:
- load normal map into its own texture unit
- compute **T** per triangle and assign it to all three vertices
  - average out **T** of shared vertices for curved surface
- pass per triangle **N** and **T** to vertex shader

In vertex shader:
- transform **N** and **T** from object to eye space (how?)
- compute **B** and orthonormal eye-to-tangent matrix (how?)
- transform **l** and **v** to tangent space, normalize, and pass them, interpolated, to the fragment shader

In fragment shader:
- sample normal map per texel (**n**)
- compute lighting in tangent space using normalized **n**, **l**, and **v**

Bump/Normal Mapping Limitations

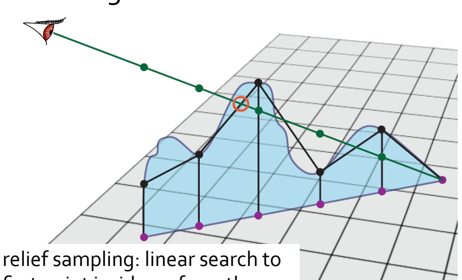
Smooth silhouette
Smooth when viewed at low viewing angle
No self-shadowing/self-occlusion



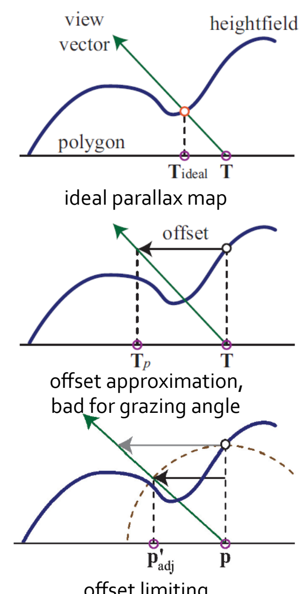
O'Brien08

Parallax/Relief Mapping

Parallax Mapping and Relief Mapping*:
- use height field and view vector to compute which "bump" is visible
- how to avoid computing view vector and height field intersection?



relief sampling: linear search to first point inside surface, then binary search between this point and last point outside surface

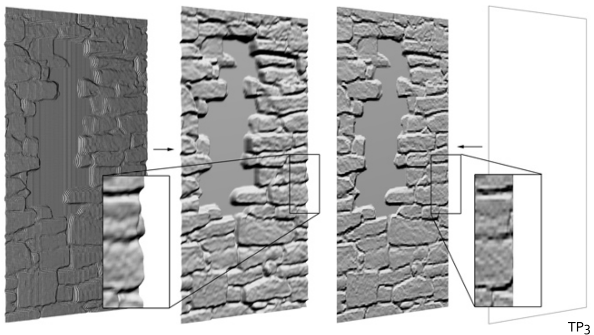


* a.k.a. Parallax Occlusion Mapping or Steep Parallax Mapping

RTR3

Displacement Mapping

Interpret texel as offset vector to actually displace fragments:
$\mathbf{p}' = \mathbf{p} + h(\mathbf{p})\mathbf{n}$
- correct silhouettes and shadows
- must be done before visibility determination
- complicates collision detection, e.g., if done in vertex shader

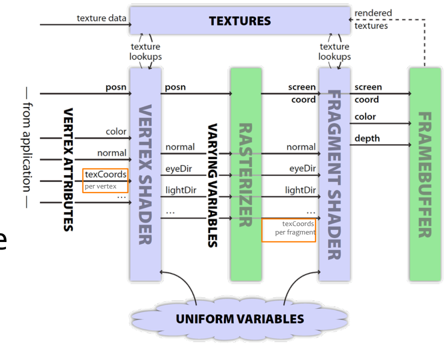


TP3

Vertex Texture Fetch

Traditionally, during vertex shading, the only texture-related computation is computing texture coordinates per vertex

With Shader Model 3.0, vertex shader can use texture map to process vertices, e.g., for displacement mapping, fluid simulation, particle systems



Marschner08

Texture Mapping

Alternative definition: a general technique for storing and evaluating functions

Textures are not just for shading parameters any more!

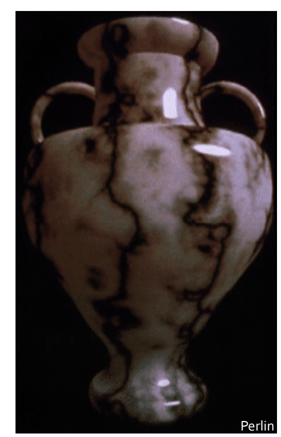
Marschner

Solid Textures

2D mapping 3D "carving"

Solid textures:
- create a 3D parameterization (s, t, r) for the texture
- map this onto the object
- the easiest parameterization is to use the model-space coordinates to index into a 3D texture $(s, t, r) = (x, y, z)$
- like "carving" the object from the material

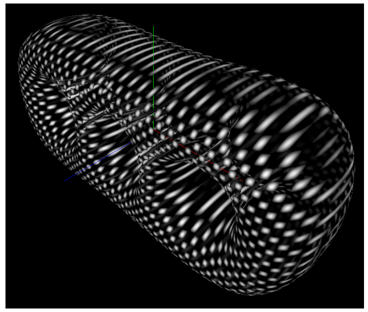
Solid procedural textures:
- more generally, instead of using the texture coordinates as an index, use them to compute a function that defines the texture



Solid Procedural Texture Example

Instead of an image, use a function

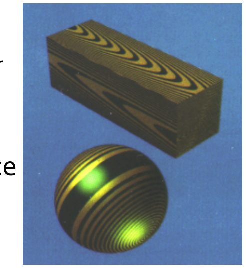
```
// vertex shader
varying vec3 pos;
...
pos = gl_Position.xyz;
...

// fragment shader
varying vec3 pos;
...
color = sin(pos.x)*sin(pos.y);
...
```



Procedural Textures

Advantages over image texture:

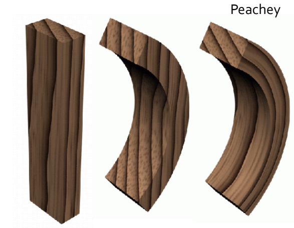
- infinite resolution and size
- more compact than texture maps
  - $f(x, y, z)$ may be a subroutine in the fragment shader
- no need to parameterize surface
- no worries about distortion and deformation
- objects appear sculpted out of solid substance
- can animate textures



Peachey

Disadvantages:

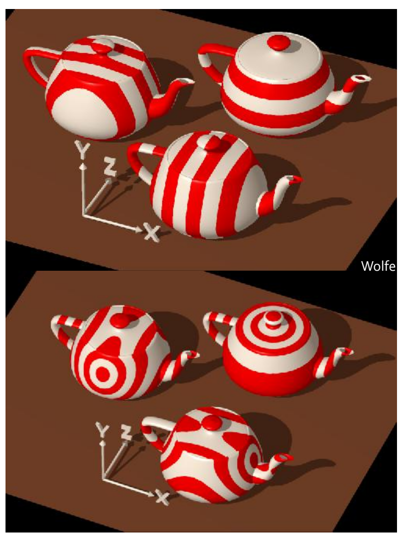
- difficult to match existing texture
- not always predictable
- more difficult to code and debug
- perhaps slower
- aliasing can be a problem



Simple Procedural Textures

Stripe: color each point one or the other color depending on where `floor`(z) (or `floor`(x) or `floor`(y)) is even or odd

Rings: color each point one or the other color depending on whether the `floor`(distance from object center along two coordinates) is even or odd

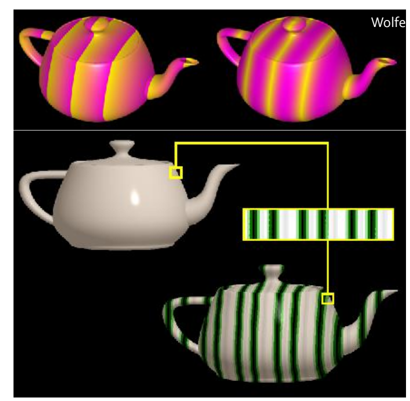


Simple Procedural Textures

Ramp functions:

- $\text{ramp}(x, y, z, a) = ((\text{float})\ \text{mod}(x, a))/a$
  - $\text{mod}(3.75, 2.0)/2.0 = 1.75/2.0 = .875$
- $\text{ramp}(x, y, z) = (\sin(x)+1)/2$

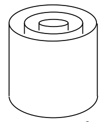
Combination: procedural color table lookup:
$f(x, y, z)$ computes an index into a color table, e.g., using the ramp function to compute an index



Wood Texture

Classify texture space into cylindrical shells

$$f(s, t, r) = (s^2 + t^2)$$

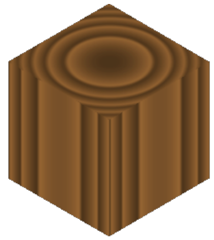


$f(s, t, r) = (s^2 + t^2)$

0 1

woodmap($f(\mathbf{p})$)

Outer rings closer together, which simulates the growth rate of real trees



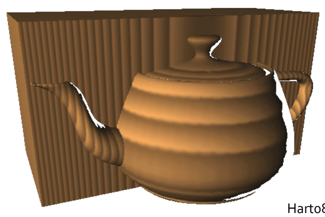
wood($\mathbf{p}$)

Wood colored color table

- woodmap(0) = brown "earlywood"
- woodmap(1) = tan "latewood"

wood($\mathbf{p}$) = woodmap($f(\mathbf{p})$ mod 1)

$\mathbf{p} = (s, t, r)$



Harto8

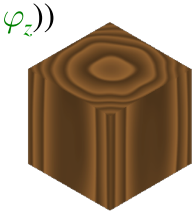
Adding Noise

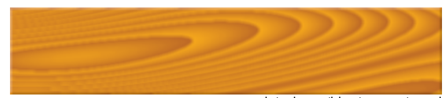
Add noise to cylinders to warp wood:

wood($x^2 + y^2 + A$* noise($f_x * x + \varphi_x, f_y * y + \varphi_y, f_z * z + \varphi_z$))

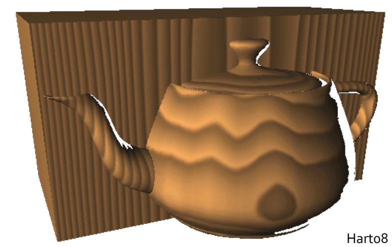
controls:

- frequency (f): coarse vs. fine detail, number and thickness of noise peaks, noise($f_x\, x, f_y\, y, f_z\, z$)
- phase (φ): location of noise peaks noise($x + \varphi_x, y + \varphi_y, z + \varphi_z$)
- amplitude (A): controls distortion due to noise effect





connectedpixel.com/blog/texture/wood



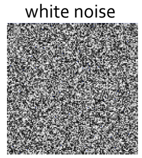
Harto8

Perlin Noise

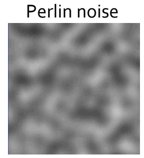
noise($\mathbf{p}$): pseudo-random number generator with the following characteristics:

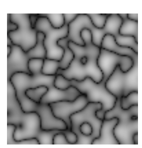
- memoryless
- repeatable
- isotropic
- band limited (coherent): difference in values is a function of distance
- no obvious periodicity
- translation and rotation invariant (but not scale invariant)
- known range [-1, 1]
  - scale to [0,1] using 0.5(noise()+1)
    - abs(noise()) creates dark veins at zero crossings

white noise



Perlin noise





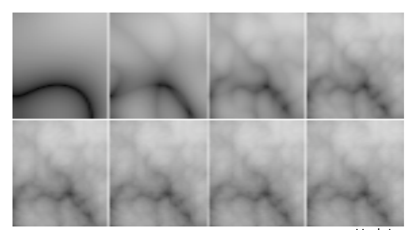
Hodgins07

Turbulence

Fractal:

- sum multiple calls to noise:

$$\text{turbulance}(\mathbf{p}) = \sum_{i=1}^{\text{octaves}} \frac{1}{2^i f} \text{noise}(2^i f \cdot \mathbf{p})$$



1-8 octaves of turbulence

Hodgins

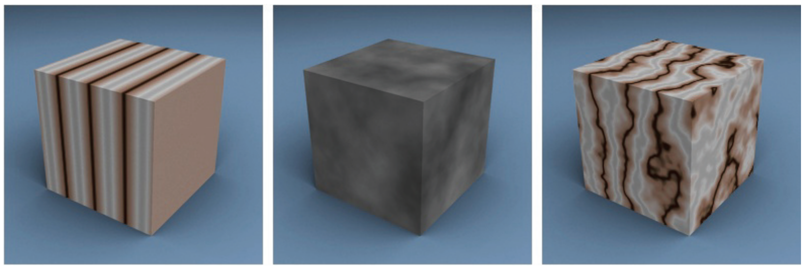
- each additional term adds finer detail, with diminishing return

Marble Texture

Use a sine function to create the stripes:

$$\text{marble} = \texttt{sin}(f * x + A * \texttt{turbulence}(x, y, z))$$

- the frequency (f) of the sine function controls the number and thickness of the veins
- the amplitude (A) of the turbulence controls the distortion of the veins



legakis.net/justin/MarbleApplet/