UMD CS Computer Graphics

Texture Mapping

Pete Willemsen

University of Minnesota Duluth

April 22, 2014

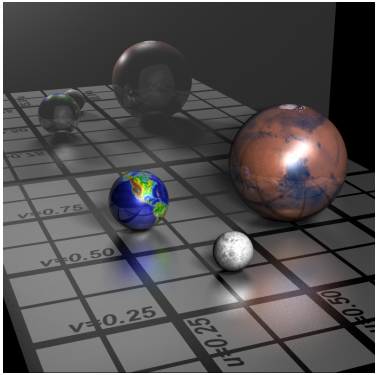
Outline

Texture Mapping

Utilize external data to replace reflectance (*or other*) information

Data can take on different forms:

- A function for generating the data or an image file for reading in the data - This is called a *texture*.
- A mapping between geometry and the various texture data - called the *texture map*
- The method or algorithms for applying data is called *texture mapping*

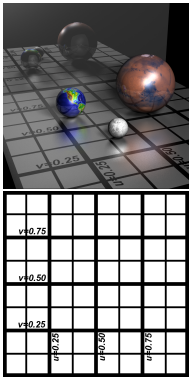


Texture Mapping

Utilize external data to replace reflectance (*or other*) information

More efficient to control reflectance with image (or procedural) data rather than through explicit creation with geometry!

- Augment reflectance properties in shaders - replace the static data within a shader with data that is looked up in a texture, such as the
  - Diffuse Reflectance
  - Specular Reflectance
- Augment surface attributes. We can even supplement or replace surface information, such as the normals or even offsets from the point of intersection:
  - Normal maps
  - Displacement maps
  - *More on these later...*



Texture Mapping Considerations

There are many things to consider with Texture Mapping.

- Texture type and data - is it an image, or is it procedurally generated

Texture Mapping Considerations

There are many things to consider with Texture Mapping.

- Texture type and data - is it an image, or is it procedurally generated
- Dimension of the texture - 1D, 2D, or 3D
  - 1D - consider a color legend, like temperature, mapping linear values to different colors - texture color $= f(x)$
  - 2D - map an image using two coordinate - texture color $= f(x, y)$
  - 3D - consider a block of some material, like marble or wood- texture color $= f(x, y, z$

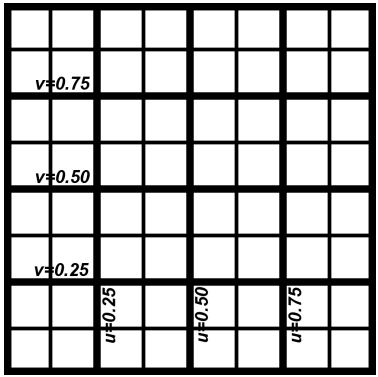
Texture Mapping Considerations

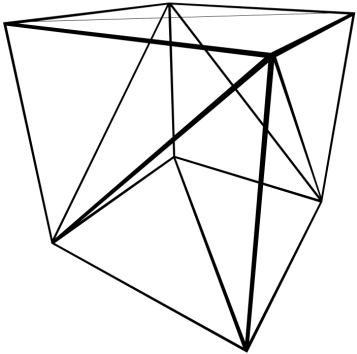
There are many things to consider with Texture Mapping.

- Texture type and data - is it an image, or is it procedurally generated
- Dimension of the texture - 1D, 2D, or 3D
  - 1D - consider a color legend, like temperature, mapping linear values to different colors - texture color $= f(x)$
  - 2D - map an image using two coordinate - texture color $= f(x, y)$
  - 3D - consider a block of some material, like marble or wood- texture color $= f(x, y, z$
- Texture Coordinates - how does the data fit onto or *map* to the specific geometry; texture coordinate generation can be quite complex; we will focus on simple mechanism to achieve the mapping

Texture Mapping

Consider texture mapping the image below onto each side of the cube:

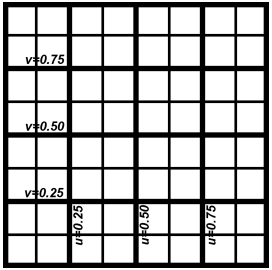


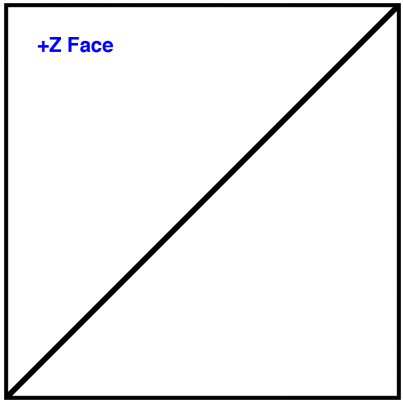


What components do you have to consider?

Texture Mapping

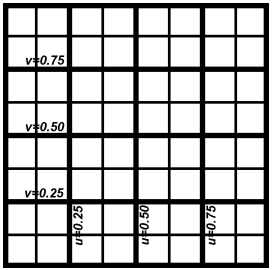
For each face, you need to understand how you want the image to map to the face:

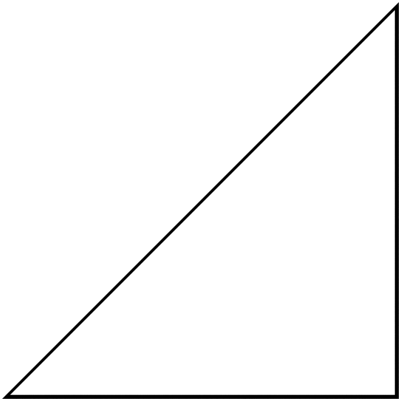




Texture Mapping

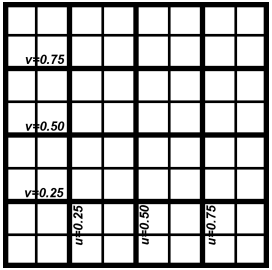
And, thusly, for each triangle, which is our lowest level of geometry, we need to understand how you want the image to map to the triangle:

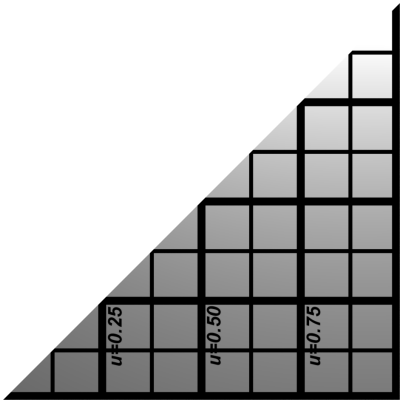




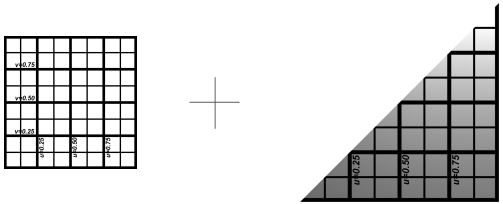
Texture Mapping

Resulting in the image being mapped onto the geometry.





Texture Mapping

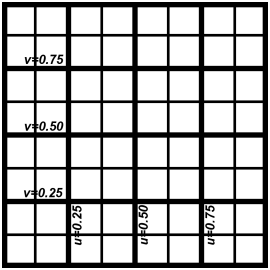


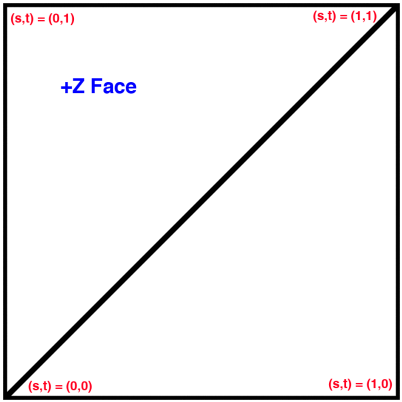
How does this mapping work in this case?

- Use normalized coordinates so that we do not have to create specific mappings for specific resolution input data.
- Refer to 2D texture coordinates with either (s, t) or (u, v)
- In other words, texture coordinates are specified in the following mapping: $s \in [0, 1]$ and $t \in [0, 1]$ map to $image_{width} \in [0, width - 1]$ and $image_{height} \in [0, height - 1]$, respectively
- Mappings can go outside this range in s and t, but this simply means that we would create tilings of the image data.

Texture Mapping

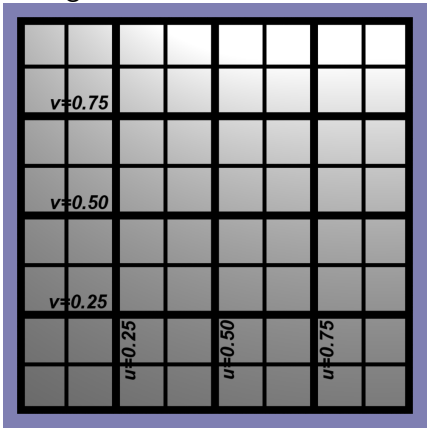
So, with texture coordinates applied, we have the following mapping:





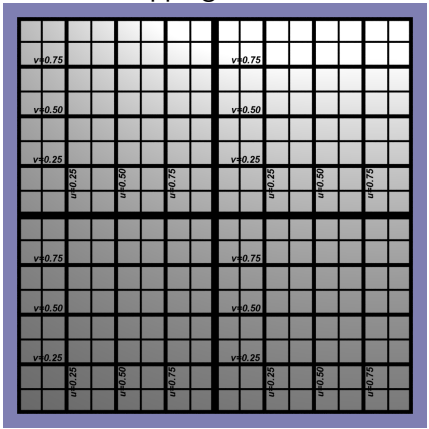
Texture Mapping

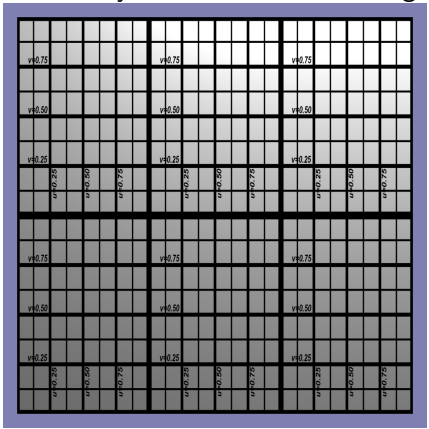
with the resulting rendered image:



Texture Mapping

Other alternate mappings that increase the s and t beyond 1. Notice the tiling:





Texture Mapping

Recall, that we use textures to *replace* or *augment* our shader data. For instance, we can use the data within the texture image to represent diffse reflectance.

Because of the mapping, we need a way to *look up* the texture data. For instance:

$$k_d = textureLookup(textureCoordinate)$$

which would allow shaders, such as this Lambertian model

$$L = \sum_{i=1}^{N} k_d I_i \max(0, \vec{n} \cdot \vec{l_i})$$

to use data other than the specified diffuse reflectance to shade the object.

Texture Lookup

The texture lookup function simply applies your mapping to the texture data to return a *texel*.

texel
A *texel* refers to a texture element. In other words, for an image, this is a single pixel's RGB data.

Texture Mapping a Face

Texture mapping involves the process of mapping the intersection points in a known way to texture data.

Given an image to use as a texture with dimensions n_x by n_y, find correspondence to points on the object.

- Use two dimensional (s, t) texture coordinates to tie the two together.
- Recall that $s \in [0, 1]$ and $t \in [0, 1]$

The texture lookup function must then:

- Convert s and t texture coordinates into pixel coordinates for the image

Texture Mapping - Nearest Neighbor

Now, how do you convert u and v into pixel indices, given an image with texture dimensions n_x by n_y? The simplest way:

$$i = \lfloor s n_x \rfloor$$

$$j = \lfloor t n_y \rfloor$$

The, texture color is determined by simple lookup into texture data (syntax related to our PNG images):

$$c_{(s,t)} = textureArray[j][i]$$

The functionality above is known as *Nearest Neighbor* lookup.

- You can get a smoother result by implementing bilinear interpolation, combining neighboring texel elements with a weighted average.

Texture Mapping - Bilinear Interpolation

To obtain a filtered, less pixelated texture lookup, let

$$s' = n_x s - \lfloor n_x s \rfloor$$

$$t' = n_y t - \lfloor n_y t \rfloor$$

Then, the color of the lookup becomes a weighted average of the neighbors:

$$c_{(s,t)} = (1 - s')(1 - t')c_{ij} + s'(1 - t')c_{(i+1)j} + (1 - s')t'c_{i(j+1)} + s't'c_{(i+1)(j+1)}$$

Other types of textures

What we've seen so far is a 2D texture. Other textures exist, such as

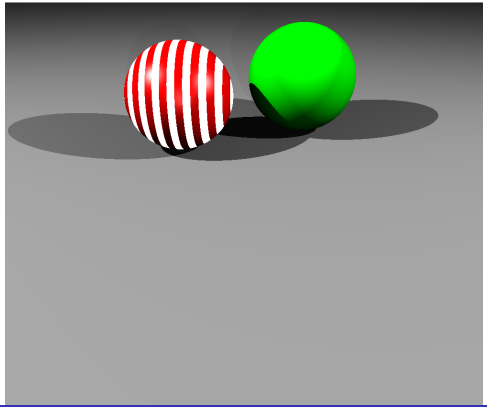
- 1D textures (lookup with a single texture coordinate)
- 3D textures (lookup into a 3D array)
- Procedural textures
- Environment Maps (aka Cube Map Textures)

Simple Procedural 3D Texture

Using procedural computation to generate a stripe texture:

$$k_d = textureLookup(p(t))$$

{*Given point of intersection, p(t), compute stripe color*}
if $(sin(p(t)_x) > 0)$ **then**
    return Vector3D(1.0, 0.0, 0.0);
else
    return Vector3D(1.0, 1.0, 1.0);
end if

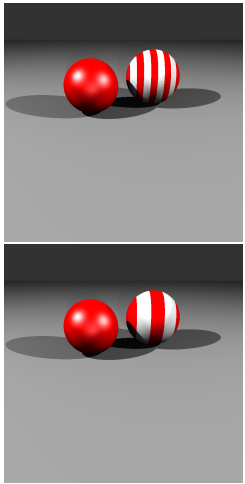


Simple Procedural 3D Texture

Width can be modified as well:

$$k_d = textureLookup(p(t))$$

{*Given point of intersection and stripe width, compute stripe color*}
if $(sin(\pi p(t)_x)/w > 0)$ **then**
    return Vector3D(1.0, 0.0, 0.0);
else
    return Vector3D(1.0, 1.0, 1.0);
end if



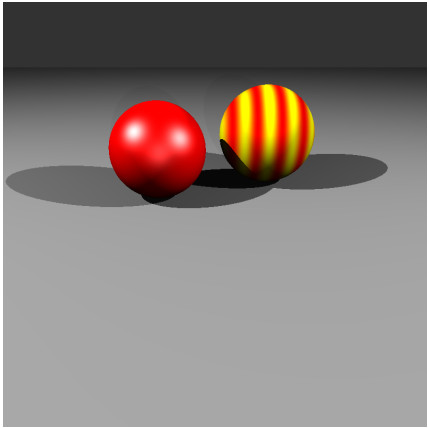
Simple Procedural 3D Texture

Colors can be smoothly blended through interpolation:

$$k_d = textureLookup(p(t))$$

{*Given point of intersection and stripe width, compute stripe color*}
$t = (1 + sin(\pi p(t)_x / w))/2$
return $(1 - t)$ Vector3D(1.0, 0.0, 0.0) + t Vector3D(1.0, 1.0, 0.0);

Note the linear interpolation based on a computed t value!



2D Texture Mapping the Sphere

Spheres are not flat objects! Try taking a piece of paper and making a sphere out of it!

We need to apply a special mapping that takes our 3D coordinates of the sphere and maps them to a 2D (s, t) value. We can start with the equation for a sphere, given azimuth and elevation angles (θ and ϕ, respectively):

$$x = x_c + Rcos\phi sin\theta$$

$$y = y_c + Rsin\phi sin\theta$$

$$z = z_c + Rcos\theta$$

and then calculate the mapping. First, note, that in this form of the equation, Z is up! In the coordinate frame we've been using througout the semester, the coordinate frame has Y up!

2D Texture Mapping the Sphere

After re-arranging:

$$x = x_c + R cos\phi sin\theta$$

$$z = z_c + R sin\phi sin\theta$$

$$y = y_c + R cos\theta$$

Then, solving for (θ, ϕ):

$$\theta = arccos(\frac{y - y_c}{R})$$

$$\phi = arctan2(z - z_c, x - x_c)$$

With this, you can find u and v:

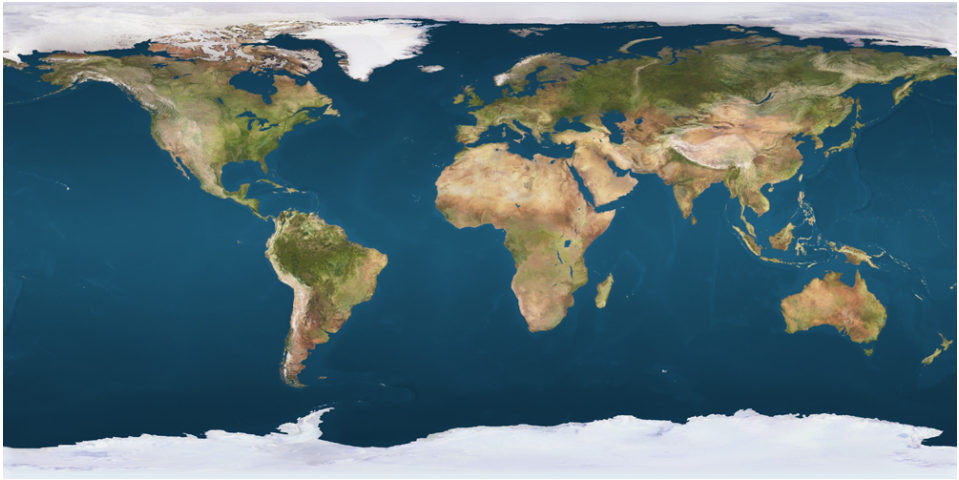
$$u = \frac{\phi}{2.0\pi}$$

$$v = \frac{\pi - \theta}{\pi}$$

Recall that $(\theta, \phi) \in [0, \pi]x[-\pi, \pi]$.

Cylindrical Mapping

Diffuse texture map for Earth is 2D, and is distorted:

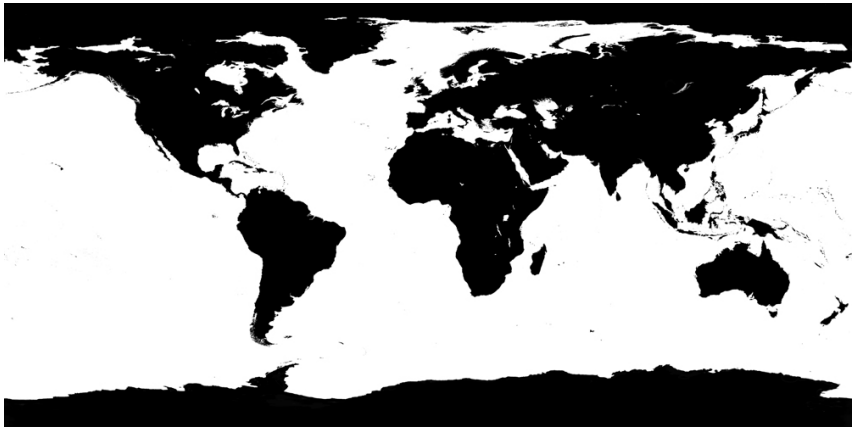


Applying the Cylindrical Map

After applying the cylindrical map to the diffuse component of the BlinnPhong shader:

Cylindrical Mapping

We can texture map additional elements, not just the diffuse component. Here is specular texture map for Earth:



Applying Diffuse and Specular Texture Maps to Earth

After applying the diffuse and specular maps to each component in the BlinnPhong shader:

Solid Noise

Solid Noise - Procedurally, and smoothly generate texture data from random numbers!

- Usually called Perlin noise, after its creator Ken Perlin (he has a nice website with code and examples)
- *Basic Idea*:
  - Build up random values in a 3D lattice
  - Look up those values based on real world X, Y, Z coordinates
  - Interpolate with neighbors
  - Apply additional filtering and/or scaling for different effects
- In practice, applying the basic idea as it is given here results in 3D random values that are very obvious and repeating. Perlin added several steps to hide the 3D lattice. Essentially, focusing on permutations of lookups in the 3D lattice.

Perlin Noise

Basic equation:

$$noise(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor + 1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor + 1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor + 1} \Omega_{ijk}(x - i, y - j, z - k)$$

What does this do? Simply, it looks into a 3D lattice of random numbers and returns a interpolated combination of neighbors in the lattice.

What if values are not interpolated? Salt and pepper noise; high frequency, not smoothed

Perlin Noise

Further detail:

$$\Omega_{ijk}(u, v, w) = \omega(u)\omega(v)\omega(w)(\Gamma_{ijk} \cdot (u, v, w))$$

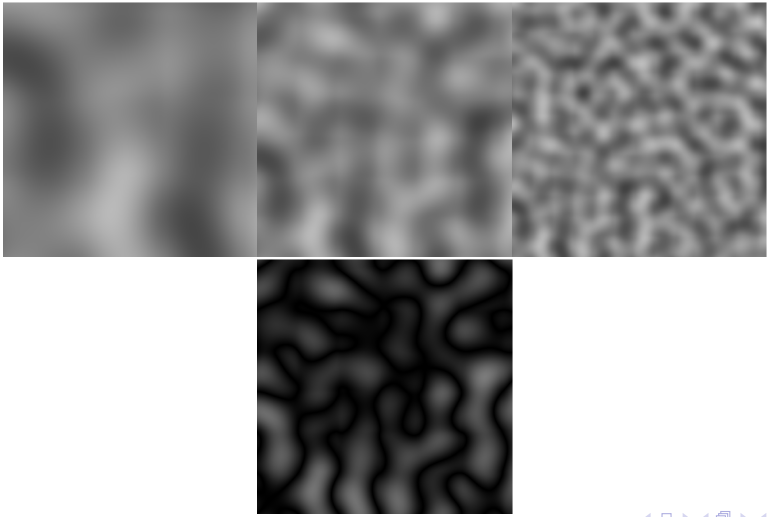
with $\omega(t)$ being a cubic weighted function:

$$\omega(t) = \begin{cases} 2|t|^3 - 3|t|^2 + 1, & \text{if } |t| < 1 \\ 0, & \text{otherwise} \end{cases}$$

$$\Gamma_{ijk} = \mathbf{G}(\phi(i + \phi(j + \phi(k))))$$

where **G** is a precomputed array of random unit vectors. $\phi(i) = P[i \bmod n]$ where P is an array of the integers 0 through n-1.

Examples of Noise



Turbulence

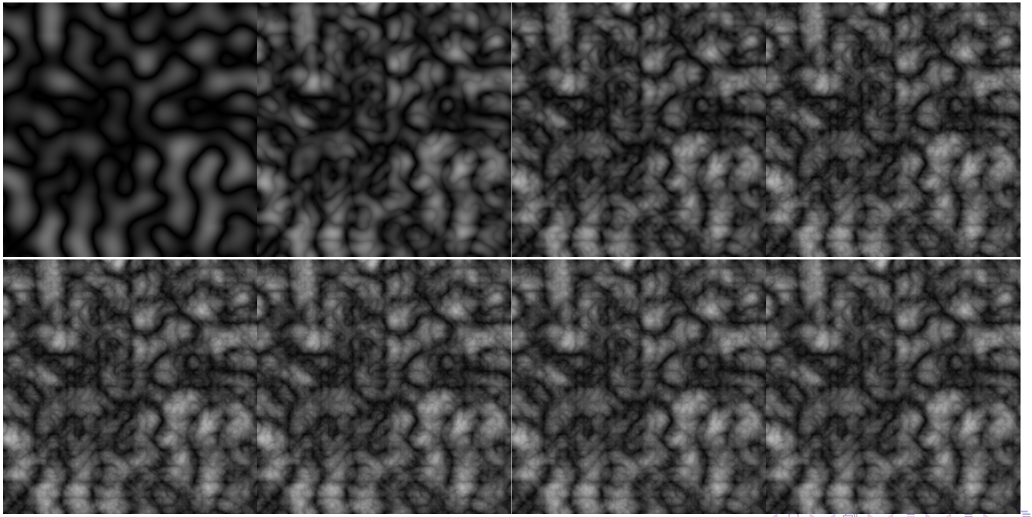
With Perlin Noise, you can create different patterns that mimic natural patterns by combining different scales and ranges of noise together.

For instance, turbulence:

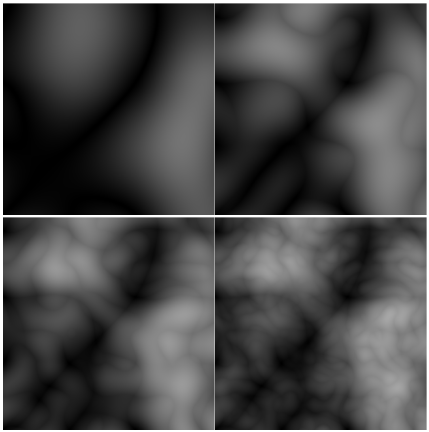
$$turbulence(\vec{pt}) = \sum_i \frac{|noise(2^i * \vec{pt})|}{2^i}$$

Repeatedly sum scaled copies of the noise function with itself.

Turbulence Levels



Closer Look at Turbulence



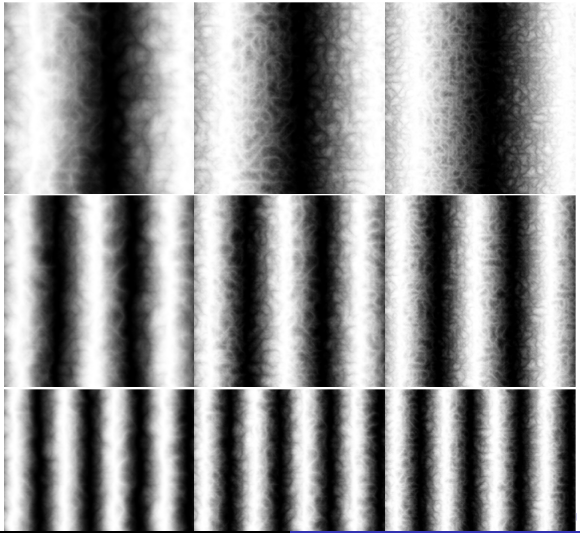
Turbulent Striping - Marble

Using turbulence function, you can manipulate values across positions, such as to acheive a marble striping:

$$t = (1 + sin(k_1 x_p + turbulence(k_2 \vec{p}))/width)/2.0$$

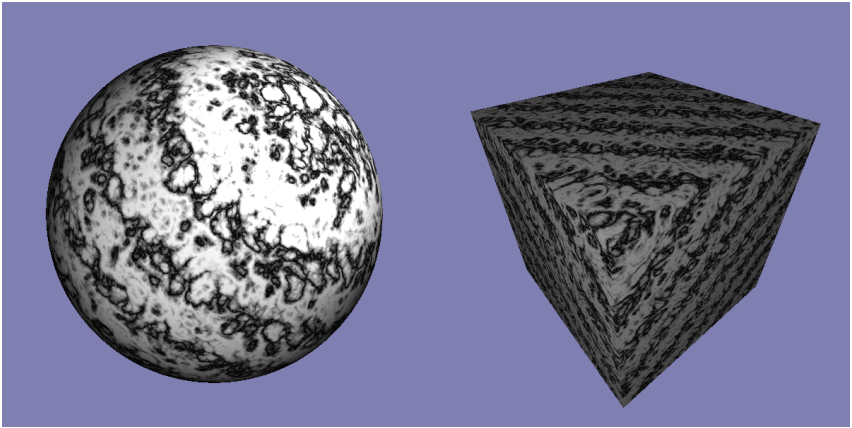
You can then interpolate between colors: t * Vector3D(1.0, 0.0, 0.0) + (1.0 - t) * Vector3D(1.0, 1.0, 1.0)

Turbulent Striping - Marble



Turbulent Striping - Marble

When you apply that to a sphere or a box, you can get the following:

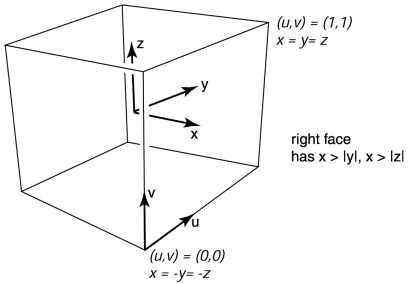


Environment Mapping

Environment maps, reflection, and cube maps. Very similar and provide a means to lookup information about the surround environment:

Basic Idea

- Surround object (or scene) with *infinitely* large axis-aligned cube.
- Texture each face of the cube with information about the environment
- Direction vectors about the unit sphere can be used to *intersect* the cube and apply texture mapping.
- In reality, no intersection is applied as the direction vector can be used to directly query the cube.



Environment Mapping

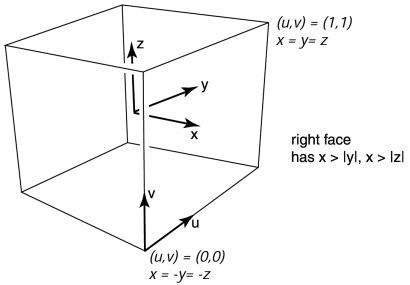
Given a reflection, or direction vector, $\vec{b}$, determine the face it hit and then compute (s, t):

- Face is determined by the major axis of the direction vector, $\vec{b}$. This is the largest positive or negative component. Another way to think about this is which face is the direction most heading.
- Once the face is determined, work in the plane of the other two dimensions and find the s, t) value by dividing each other component by the absolute value of the largest component. For instance, with +X being the selected face:

$$s = y/|x|$$

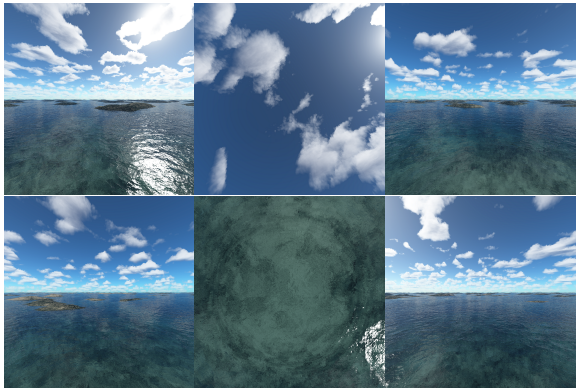
$$t = z/|x|$$

but those values are in the range $[-1, 1]$. You can then convert to the $[0, 1]$ range needed for (s, t) lookups by adding 1.0 and dividing by 2.0.



Environment Mapping

Apply texture to each face of the cube:



Environment Mapping

Results in images like this: