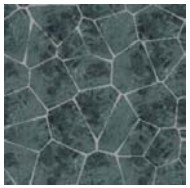
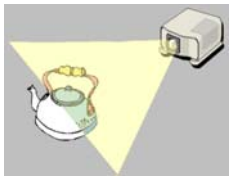
Texture-Mapping Tricks

- Combining Textures
- Filtering Textures
- Textures and Shading
- Bump Mapping
- Solid Textures

Texture Mapping Modes

- Label textures
- Projective textures
- Environment maps
- Shadow maps
- ...





The Best of All Worlds

All these texture mapping modes are great!

The problem is, no one of them does everything well.

Suppose we allowed several textures to be applied to each primitive during rasterization.

Multipass vs. Multitexture

Multipass (the old way) - Render the image in multiple passes, and "add" the results.

Multitexture - Make multiple texture accesses within the rasterizing loop and "blend" results.

Blending approaches:

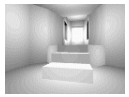
- Texture modulation
- Alpha Attenuation
- Additive textures
- Weird modes

Texture Mapping in Quake

Quake uses *light maps* in addition to texture maps. Texture maps are used to add detail to surfaces, and light maps are used to store pre-computed illumination. The two are multiplied together at run-time, and cached for efficiency.

	Texture Maps	Light Maps
Data	RGB	Intensity
Instanced	Yes	No
Resolution	High	Low

Light map image
by Nick Chirkov



Sampling Texture Maps

When texture mapping it is rare that the screen-space sampling density matches the sampling density of the texture. Typically one of two things can occur:

Oversampling of the texture or Undersampling of the texture

In the case of oversampling we already know what to do... Interpolation (review on Antialiasing and Resampling). But, how do we handle undersampling?

How Bad Does it Look?

Let's take a look at what undersampling looks like:

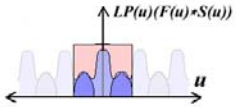
Notice how details in the texture, in particular the mortar between the bricks, tend to pop (disappear and reappear).

This popping is most noticeable around details (parts of the texture with a high-spatial frequency). This is indicative of aliasing (high-frequency details showing up in areas where we expect to see low frequencies).

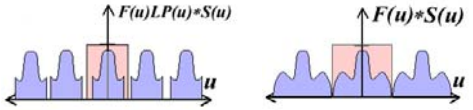
We've Seen this Sort of Thing Before

Remember... Aliasing?

This was the phenomenon that occurred when we undersampled a signal. It caused certain high frequency features to appear as low frequencies.



To eliminate aliasing we had to either band limit (low pass filter) our input signal or sample it at a higher rate:



Spatial Filtering

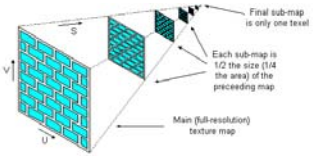
In order to get the sort of images the we expect, we must *prefilter* the texture to remove the high frequencies that show up as artifacts in the final rendering. The prefiltering required in the undersampling case is basically a spatial integration over the extent of the sample.

We could perform this filtering while texture mapping (during rasterization), by keeping track of the area enclosed by sequential samples and performing the integration as required. However, this would be expensive. The most common solution to undersampling is to perform prefiltering prior to rendering.

MIP Mapping

MIP Mapping is one popular technique for precomputing and performing this prefiltering. MIP is an acronym for the latin phrase *multium in parvo*, which means "many in a small place". The technique was first described by Lance Williams. The basic idea is to construct a *pyramid of images* that are prefiltered and resampled at sampling frequencies that are a binary fractions (1/2, 1/4, 1/8, etc) of the original image's sampling.

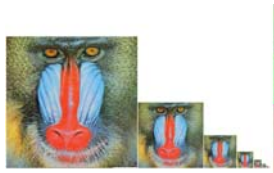
While rasterizing we compute the index of the decimated image that is sampled at a rate closest to the density of our desired sampling rate (rather than picking the closest one can in also interpolate between pyramid levels).



Computing this series of filtered images requires only a small fraction of addtional storage over the original texture (How small of a fraction?).

Storing MIP Maps

One convenient method of storing a MIP map is shown below. (It also nicely illustrates the 1/3 overhead of maintaining the MIP map).



10-level mip map

Memory format of a mip map

We must make a few small modifications to our rasterizer to compute the MIP map level. Recall the equations that we derived for mapping screen-space interpolants to their 3-space equivalent.

$$u = u_1 + s(u_2 - u_1)$$

$$s = \frac{t\, w_2}{w_1 + t(w_2 - w_1)}$$

Finding the MIP level

What we'd like to find is the step size that a uniform step in screen-space causes in three-space, or, in other words how a screen-space change relates to a 3-space change. This sounds like the derivatives, *(du/dt, dv/dt)*. They can be computed simply using the chain rule:

$$\frac{du}{dt} = \frac{du}{ds}\frac{ds}{dt} = (u_2 - u_1)\frac{w_1\, w_2}{(w_1 + t(w_2 - w_1))^2}$$

$$\frac{dv}{dt} = (v_2 - v_1)\frac{w_1\, w_2}{(w_1 + t(w_2 - w_1))^2}$$

Notice that the term being squared under the numerator is just the *w* plane equation that we are already computing. The remaining terms are constant for a given rasterization. Thus all we need to do to compute the derivative is a square the *w* accumulator and multiply it by a couple of constants.

Now, we know how a step in screen-space relates to a step in 3-space. So how do we translate this to an index into our MIP table?

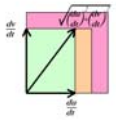
MIP Indices

Actually, you have a choice of ways to translate this **gradient value** into a MIP level.

This also brings up one of the shortcomings of MIP mapping. MIP mapping assumes that both the *u* and v components of the texture index are undergoing a uniform scaling, while in fact the terms *du/dt* and *dv/dt* are relatively independent. Thus, we must make some sort of compromise. Two of the most common approaches are given below:

$$level = \log_2\left(\sqrt{\left(\frac{du}{dt}\right)^2 + \left(\frac{dv}{dt}\right)^2}\right)$$

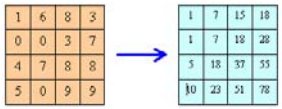
$$level = \log_2\left(Max\left(\left|\frac{du}{dt}\right|, \left|\frac{dv}{dt}\right|\right)\right)$$



The differences between these level selection methods is illustrated by the accompanying figure.

Summed-Area Tables

There are other approaches to computing this prefiltering integration on the fly. One, which was introduced by Frank Crow is called a *summed-area table*. Basically, a summed-area table is a tabularized two-dimensional cumulative distribution function. Imagine having a 2-D table of numbers the cumulative distribution function could be found as shown below.



To find the sum of region contained in a box bounded by (x_0, y_0) and (x_1, y_1):

$$\mathbf{T}(x_1, y_1) - \mathbf{T}(x_0, y_1) - \mathbf{T}(x_1, y_0) + \mathbf{T}(x_0, y_0)$$

This approach can be used to compute the integration of pixels that lie under a pixel by dividing the resulting sum by the area of the rectangle,

$$(y_1 - y_0)(x_1 - x_0).$$

With a little more work you can compute the area under any four-sided polygon (How?).

Summed-Area Tables

- How much storage does a summed-area table require?
- Does it require more or less work per pixel than a MIP map?
- What sort of low-pass filter does a summed-area table represent?
- What do you remember about this sort of filter?


No filtering


MIP mapping


Summed-Area Table

These nice images were originally grabbed from a link on the http://www.gris.uni-tuebingen.de/ web page.

Phototextures

To this point we've only used textures as labels over geometry.

Their applications are much broader, however.

Simple extensions include:

- Spatially varying surface properties:
  - The diffuse shading coefficient, k_d, could be stored as a texture.
  - Likewise for the intrinsic color, k_a, k_s, and n_{shiny}.
- Textures can also be used in a lot of other ways

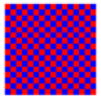
Adding Texture Mapping to Illumination

Texture mapping can be used to alter some or all of the constants in the illumination equation. We can simply use the texture as the final color for the pixel, or we can just use it as diffuse color, or we can use the texture to alter the normal, or... the possibilities are endless!

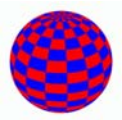
$$I_{total} = k_a I_{ambient} + \sum_{i=1}^{lights} I_i \left(k_d \left(\hat{N} \cdot \hat{L} \right) + k_s \left(\hat{V} \cdot \hat{R} \right)^{n_{shiny}} \right)$$

Phong's Illumination Model

Constant Diffuse Color



Diffuse Texture Color



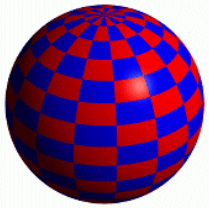
Texture used as Label

Texture used as Diffuse Color

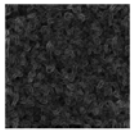
Bump Mapping

Textures can be used to alter the surface normal of an object. This does not change the actual shape of the surface -- we are only shading it as if it were a different shape! This technique is called *bump mapping*. The texture map is treated as a single-valued height function. The value of the function is not actually used, just its partial derivatives. The partial derivatives tell how to alter the true surface normal at each point on the surface to make the object appear as if it were deformed by the height function.

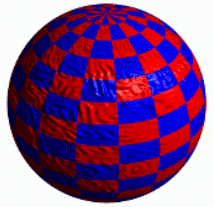
Since the actual shape of the object does not change, the silhouette edge of the object will not change. Bump Mapping also assumes that the Illumination model is applied at every pixel (as in Phong Shading or ray tracing).



Sphere w/Diffuse Texture



Swirly Bump Map



Sphere w/Diffuse Texture & Bump Map

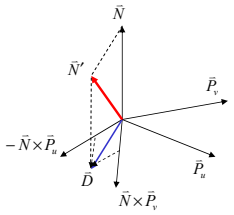
Bump mapping

$$\vec{P} = [x(u,v), y(u,v), z(u,v)]^T$$

$$\vec{N} = \vec{P}_u \times \vec{P}_v$$

$$\vec{P}' = \vec{P} + \frac{B(u,v)\vec{N}}{\left\|\vec{N}\right\|}$$

$$\vec{N}' \approx \vec{N} + \underbrace{\frac{B_u\left(\vec{N}\times\vec{P}_v\right) - B_v\left(\vec{N}\times\vec{P}_u\right)}{\left\|\vec{N}\right\|}}_{\vec{D}}$$



$$B_u = \frac{B(s-\Delta,t) - B(s+\Delta,t)}{2\Delta}$$

$$B_v = \frac{B(s,t-\Delta) - B(s,t+\Delta)}{2\Delta}$$

Compute bump map partials by numerical differentiation

Bump mapping derivation

$$\vec{P}' = \vec{P} + \frac{B(u,v)\vec{N}}{\left\|\vec{N}\right\|}$$

$$\vec{P}'_u = \vec{P}_u + \frac{B_u\vec{N}}{\left\|\vec{N}\right\|} + \frac{B\vec{N}_u}{\left\|\vec{N}\right\|} \quad (\vec{N}_u \approx 0)$$

Assume B is very small...

$$\vec{P}'_v = \vec{P}_v + \frac{B_v\vec{N}}{\left\|\vec{N}\right\|} + \frac{B\vec{N}_v}{\left\|\vec{N}\right\|} \quad (\vec{N}_v \approx 0)$$

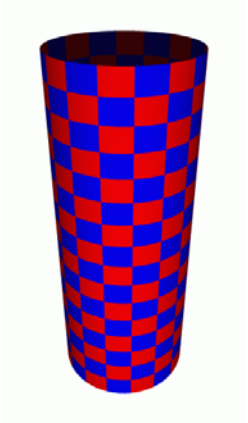
$$\vec{N}' = \vec{P}'_u \times \vec{P}'_v$$

$$\vec{N}' \approx \vec{P}_u \times \vec{P}_v + \frac{B_u(\vec{N}\times\vec{P}_v)}{\left\|\vec{N}\right\|} + \frac{B_v(\vec{P}_u\times\vec{N})}{\left\|\vec{N}\right\|} + \frac{B_u B_v(\vec{N}\times\vec{N})}{\left\|\vec{N}\right\|^2}$$

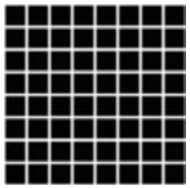
But $\vec{P}_u \times \vec{P}_v = \vec{N}$, $\vec{P}_u \times \vec{N} = -\vec{N}\times\vec{P}_u$ and $\vec{N}\times\vec{N} = 0$ so

$$\vec{N}' \approx \vec{N} + \frac{B_u(\vec{N}\times\vec{P}_v)}{\left\|\vec{N}\right\|} - \frac{B_v(\vec{N}\times\vec{P}_u)}{\left\|\vec{N}\right\|}$$

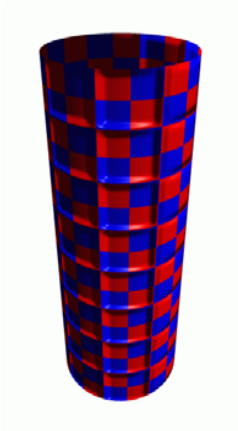
More Bump Map Examples



Cylinder w/Diffuse Texture Map

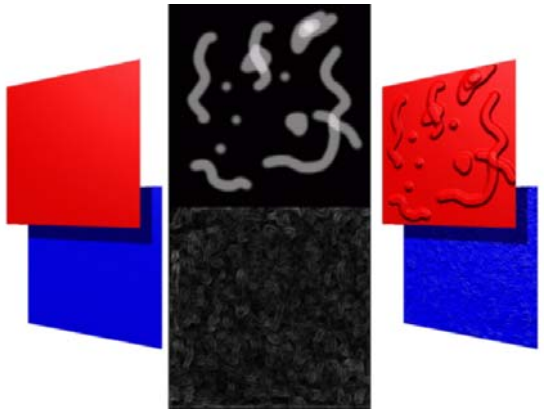


Bump Map



Cylinder w/Texture Map & Bump Map

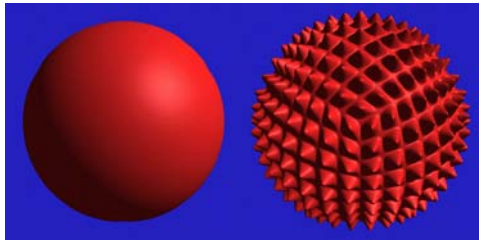
One More Bump Map Example



Notice that the shadow boundaries remain unchanged.

Displacement Mapping

We use the texture map to actually move the surface point. This is called *displacement mapping*. How is this fundamentally different than bump mapping?



The geometry must be displaced before visibility is determined. Is this easily done in the graphics pipeline? In a ray-tracer?

Displacement Mapping Example

It is possible to use displacement maps when ray tracing.

Image from

Geometry Caching for Ray-Tracing Displacement Maps

by Matt Pharr and Pat Hanrahan.

Another Displacement Mapping Example

Image from Ken Musgrave

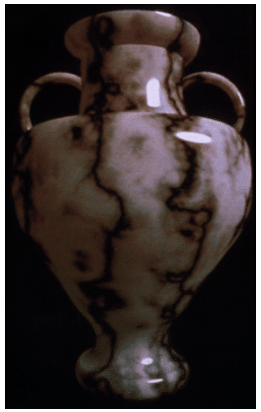
Three Dimensional or Solid Textures

The textures that we have discussed to this point are two-dimensional functions mapped onto two-dimensional surfaces. Another approach is to consider a texture as a function defined over a three-dimensional surface. Textures of this type are called *solid textures*.

Solid textures are very effective at representing some types of materials such as marble and wood. Generally, solid textures are defined procedural functionsrather than tabularized or sampled functions as used in 2-D (Any guesses why?)

The approach that we will explore is based on *An Image Synthesizer*, by Ken Perlin, SIGGRAPH '85.

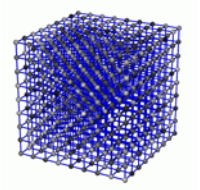
The vase to the right is from this paper.



Noise and Turbulence

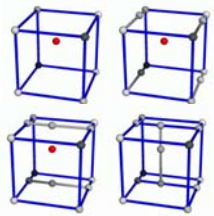
When we say we want to create an "interesting" texture, we usually don't care exactly what it looks like -- we're only concerned with the overall appearance. We want to add random variations to our texture, but in a controlled way. Noise and turbulence are very useful tools for doing just that.

A *noise function* is a continuous function that varies throughout space at a uniform frequency. To create a simple noise function, consider a 3D lattice, with a random value assigned to each triple of integer coordinates:



Interpolating Noise

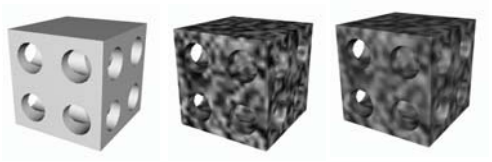
To calculate the noise value of any point in space, we first determine which cube of the lattice the point is in. Next, we interpolate the desired value using the 8 corners of the cube:



Trilinear interpolation is illustrated above. Higher-order interpolation can also be used.

Evaluating Noise

Since noise is a 3D function, we can evaluate it at any point we want. We don't have to worry about about mapping the noise to the object, we just use (x, y, z) at each point as our 3D texture coordinates! It is as if we are carving our object out of a big block of noise.



Original Object | Trilinear Noise | Triquadratic Noise

Turbulence

Noise is a good start, but it looks pretty ugly all by itself. We can use noise to make a more interesting function called turbulence. A simple turbulence function can be computed by summing many different frequencies of noise functions:

One Frequency | Two Frequencies | Three Frequencies | Four Frequencies

Now we're getting somewhere. But even turbulence is rarely used all by itself. We can use turbulence to build even more fancy 3D textures...

Marble Example

We can use turbulence to generate beautiful 3D marble textures, such as the marble vase created by Ken Perlin. The idea is simple. We fill space with black and white stripes, using a sine wave function. Then we use turbulence at each point to distort those planes.

By varying the frequency of the sin function, you get a few thick veins, or many thin veins. Varying the amplitude of the turbulence function controls how distorted the veins will be.

Marble = sin(f * (x + A*Turb(x,y,z)))

CLICK TO SEE EXAMPLE

Next Time - Radiosity