The Importance of Surface Texture

Objects in the real world have rich, detailed surface textures

- to produce believable scenes, we must replicate this detail
- uniformly colored surfaces only get us so far

Generated with Blue Moon Rendering Tools — www.bmrt.org

How Do We Model Intricate Surface Detail?

Approach #1: Explicit geometric representation

- actual polygons that model all the surface variations
- up to some finest level of detail
- may generate a *lot* of polygons

Approach #2: Geometry + texture images

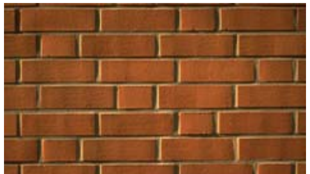
- geometry only describes the general shape of the object
- paste an image onto the wall to give the appearance of brick

Often We Use Simple Patterns

Generally useful for skin, bricks, stucco, granite, …

Typically need to repeat texture over the object

- must make sure there are no seams when texture is tiled



Or Given a Model and a Single Texture

Wrap the Texture onto the Model

Sample model from www.cyberware.com

Framework for Texture Mapping

The texture itself is just a 2-D raster image
- acquired from reality, hand-painted, or procedurally generated

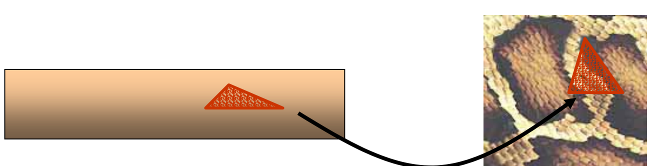
Establish a correspondence between surface points & texture

When shading a particular surface point
- look up the corresponding pixel in the texture image
- final color of point will be a function of this pixel

Texturing Polygonal Models

Polygonal models are not so easy
- they don’t have a natural 2-D parameterization
- we need to create one

For each vertex, we specify a texture coordinate
- a (u,v) pair that maps that point into the texture image
- a triangle on the surface will be mapped to a triangle in texture
- can interpolate texture coordinates over the triangle
- note that the size of the triangle may be quite different



Texturing and Rasterization

During rasterization, we traverse the pixels of a triangle
- at each pixel we interpolate the correct texture coordinate
- and we retrieve the corresponding texel (texture element)

What do we do with the contents of the texel?
- color — use it to fill in the current pixel
- reflectance — coefficient for illumination equation (e.g., k_d)
- transparency — an alpha value
- and *many* others, some of which we’ll discuss later

OpenGL Texture Modes

Determines how the contents of the texture are interpreted

For RGB images:

GL_MODULATE — multiply together with surface color

GL_BLEND — use as a *t* value to blend surface color and a predetermined color

GL_DECAL and GL_REPLACE — use texture color directly

Texturing with OpenGL

First, turn on texturing — glEnable(GL_TEXTURE_2D)

Next, pass the actual texture image to OpenGL

- glTexImage2D(GL_TEXTURE_2D, *level*, *channels*, *width*, *height*, *border*, *format*, *type*, *image*)
- for now, *level*=0 and *border*=0
- *channels* is usually 3 (RGB)
  - with *format*=GL_RGB and *type*=GL_UNSIGNED_BYTE

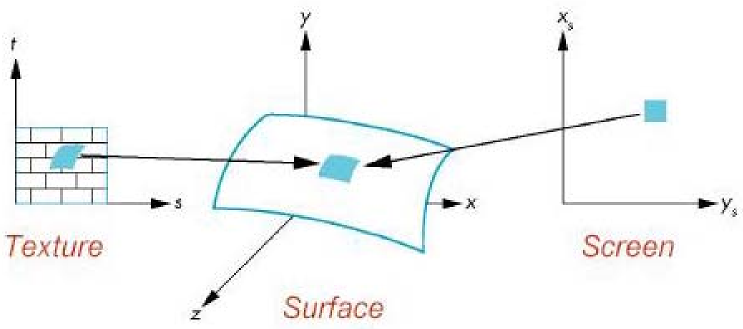
Have lots of options to control texturing behavior

- see glTexEnvf() and glTexParameterf() for details
  - texture coordinates clamped to [0,1] or do they wrap around?
  - how is the color of the texture applied to the surface?

Minification and Magnification

Minification: 1 pixel covers multiple texels

Magnification: 1 texel covers multiple pixels



Texturing with OpenGL

Here's an example setup

```
glEnable(GL_TEXTURE_2D);

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);


glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height, 0,
                   GL_RGB, GL_UNSIGNED_BYTE, image);
```

This configures the texturing system to

- combine (modulate) the texture color with the surface color
- wrap texture coordinates around outside unit square
- linearly average texels when “magnifying” and “minifying”

Texturing with OpenGL

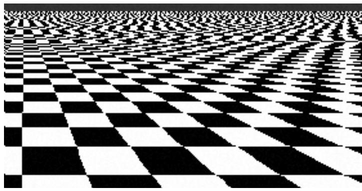
When drawing, just assign texture coordinates to vertices

```
glBegin(GL_TRIANGLES);
  glNormal3fv(n1);
  glTexCoord2f(s1, t1);
  glVertex3fv(v1);

  glNormal3fv(n2);
  glTexCoord2f(s2, t2);
  glVertex3f(v2);

  glNormal3fv(n3);
  glTexCoord2f(s3, t3);
  glVertex3fv(v3);
glEnd();
```

Texture Aliasing

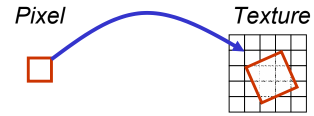
Recall the simple texture application method we discussed

- at each pixel we interpolate the correct texture coordinate
- and we retrieve the corresponding texel
- can lead to nasty aliasing



Why is this?

Consider mapping of pixel to texture



- may be mapped to several (fractional) texels
- but we're only selecting one of them to use in the pixel
- this kind of point sampling results in aliasing

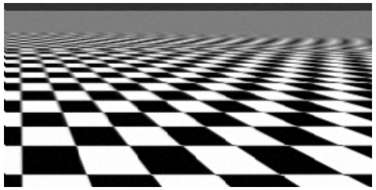
Texture Antialiasing

The antidote is to average (filter) all covered texels together

- need to choose appropriate averaging method

Removes objectionable artifacts

- but it's not magic
- very high frequency details just get smoothed over completely (e.g., gray on horizon)



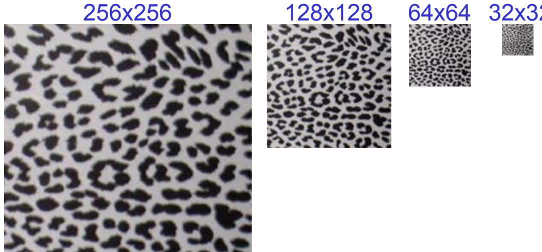
Unfortunately, there's a significant drawback here

- averaging covered texels can be very expensive
- for every pixel, might have to visit $O(n)$ texels
- this would really hurt rendering performance

Texture Antialiasing

For efficiency, we can use an image pyramid (mip–map)

- base of pyramid is the original image (level = 0)
- level 1 is the image down-sampled by a factor of 2
- level 2 is down-sampled by a factor of 4, and so on
- requires that original dimensions be a power of 2
- and it's not too big: 4/3 the size of the original image



Antialiasing with Image Pyramids

Image pyramids let us efficiently average large regions

- each texel in upper levels covers many base texels
  - at level k they are the average of 2^kx2^k texels
- can quickly assemble appropriate texels for averaging

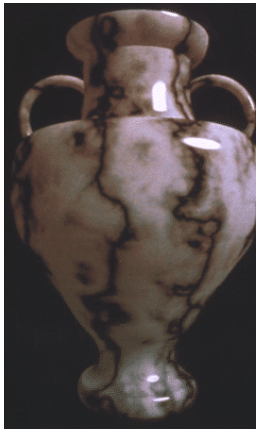
Fortunately, OpenGL can take care of most details

- gluBuild2DMipmaps() — automatically generate pyramid from base image
- control behavior with glTexParameter()
- OpenGL handles all filtering details during rasterization

Solid Texture

Instead of texture images, we can define texture volumes

- create a 3-D parameterization (u,v,w) for the texture
- map this onto the object
- the easiest parameterization $(u,v,w) = (x,y,z)$



Turns out to be a powerful technique

- for procedural generation
- more readily applies to implicit surfaces
- and other surfaces without natural 2-D parameterizations

Some Texturing Applications

First, there's the obvious one: realistic surface detail

- paste a fur, marble, face scan, … on a surface

We can also support illumination precalculation

- suppose we precompute some expensive lighting effects
  - soft shadows, indirect light (e.g., radiosity)
- can hard code this lighting into texture maps

Texturing can also be handy for faking objects

- billboards — place image on a polygon which always rotates to face the viewer (handy for things like trees)
- sprites used in video games are a similar idea

And texturing is useful for level of detail management

- can decouple resolution of texture from resolution of surface