#12: Texture Mapping

CSE167: Computer Graphics
Instructor: Ronen Barzel
UCSD, Winter 2006
Outline for today

- *Lighting & Shading wrapup*
- Texture Mapping
- Procedural Textures
- Fancy Texture Effects
Lighting Review

- Local Illumination Lighting Models
  - Ambient
    - Normals don’t matter
  - Diffuse (Lambert)
    - Angle between surface normal and light
  - Phong, Blinn
    - Surface normal, light, and viewpoint

- What they capture:
  - Direct illumination from light sources
  - Diffuse and Specular reflections
  - (Very) Approximate effects of global lighting

- Compute lighting on any point on a surface
  - Vertices in particular
Advanced Lighting & Rendering

- Shadows
- Reflection
- Translucency, refraction
- Caustics (light focusing)
- Global Illumination (light “spilling” from one surface to another)
- Area light sources/emitters
- Atmospheric effects (fog, clouds)
- Motion blur
- Depth of field
- Exposure & dynamic range
- ...

...
Advanced Lighting
Shading

- Choosing the color for each pixel
  - For photorealism, typically shading==lighting
  - But can take shortcuts, or do other things
- Various techniques:
  - Flat
  - Gouraud
  - Phong
  - Cartoon
  - Illustration
  - Procedural
Flat shading

- Compute shading at a representative point and apply to whole polygon
  - OpenGL uses one of the vertices

- Advantages:
  - Fast - one shading computation per polygon, fill entire polygon with same color

- Disadvantages:
  - Inaccurate
  - Faceted
Gouraud Shading

- Light each vertex with its own location and normal
  - Result is a color at each vertex
- Interpolate the colors across each triangle
- Default mode for most hardware

Advantages:
- Fast: incremental calculations when rasterizing
- Much smoother - use one normal per shared vertex to get continuity between faces

Disadvantages:
- Don’t get smooth specular highlights
- C^1-discontinuities in color cause *Mach bands*: perceptual illusions of edges
Phong Shading

- Want to recompute lighting at every pixel
  - Need normal vector at each pixel
- Interpolate vertex normals
  - Interpolate the normals while scan-converting
  - Typically, bilinearly interpolate x, y, z; then renormalize
  - Also interpolate or transform to get position in world/camera space
- Known as *Phong Shading* or *Phong Interpolation*
  - Not to be confused with Phong Lighting Model
  - (though both are often used at the same time)

**Advantages:**
- More accurate
- Better images

**Disadvantages:**
- Slow
- Still not completely accurate
- Modern GPUs can perform Phong shading via pixel shaders
Gouraud vs. Phong shading

- Gouraud misses specular highlights

solution:
- tessellate more finely: use smaller triangles
- that would help with the polygonal artifacts on the silhouettes too
Linear interpolation in screen space doesn’t align with linear interpolation in world space

Solutions:
- Do hyperbolic interpolation (see Buss)
- Break up into smaller triangles
Cartoon Shading

- **General approach:**
  - Compute simple lighting
    - diffuse (with shadows)
  - Remap continuous intensity to two (or a few) levels
  - Some trickery to draw dark edges
Illustration

- Use lighting & shadow to define “tone”
  - Not scan-conversion -- separate algorithms for drawing strokes

(Winkenbach & Salesin, University of Washington)
Procedural Shading

- Software or hardware renderer calls an arbitrary routine at every pixel (or sub-pixel)
  - Routine is known as a shader
  - Can typically implement Phong or any other lighting model
  - Used for patterns or other special effects
  - Used for cartoon shading
- Shader typically written in a special-purpose shading language
  - RenderMan Shading Language (Pixar)
  - Cg (Nvidia)
  - HLSL (DirectX)
  - C/C++ plugins, for software renderers
- Shader:
  - can get surface normal, position, material properties, light locations, …
  - computes a color value (also a transparency value)
  - can sometimes change the surface position! displacement shader
Programmable GPUs

- Two kinds of shaders:
  - *vertex shaders*: run once for each vertex
  - *pixel shaders*: run for each pixel

- Limits on
  - size of program
  - amount of memory available
  - types of memory and texture access (will talk about texture later)
  - etc.

- Technology still improving, capabilities keep expanding
Outline for today

- Lighting & Shading wrapup
- *Texture Mapping*
- Procedural Textures
- Fancy Texture Effects
Texture

- We know how to model and render uniform smooth surfaces
- Want more interesting *texture*:
  - variations in color
  - small-scale patterns, bumps, roughness
- In principle we could model everything
  - really tiny triangles
  - per-vertex color
  - per-vertex material properties
- Modeling everything unwieldy, impractical
  - Too much of the wrong kind of data
  - Don’t need that much fine-scale geometric data (triangle vertices)
  - Just want surface data, separate from geometric data
Texture data

Texture data can come from

Files:
- Typically, 2D color image file, known as a texture map or just texture
- Applied to surface using Texture Mapping techniques

“Painting” directly on surfaces:
- Using 3D paint tools integrated with 3D modeling systems
- Results typically stored in 2D image files
- i.e., this reduces to texture mapping

Programs:
- Procedural Texture
- Implemented in procedural shaders
Texture examples

Sphere with no texture
Texture image
Sphere with texture
Texture examples
Texture Map

- An image file
- Define *texture space*:
  - A normal 2D space
  - *Texture coordinates*: \((u,v)\) or \((s,t)\)
  - Lower-left corner of the image is at \((0,0)\)
  - Upper-right of the image is the coordinate \((1,1)\)
  - Data at each coordinate is (typically) a color value
- The actual texture map might be:
  - 512 x 256 pixels for example,
  - 24 bit color stored per pixel
- Common term: *texel* instead of *pixel*
  - “texture element” instead of “picture element”
Texture Mapping

- Store texture coordinates as vertex data
  - Coordinates interpolated to each pixel during scan conversion
  - Color looked up in texture map file, per pixel
  - Various ways to come up with the coordinate assignments

\[(1,1)\]
Texture Coordinate Assignment

- Various ways to assign the texture coordinates:
  - Parametric mapping
  - Orthographic Mapping
  - Perspective Mapping
  - Spherical Mapping
  - Cylindrical Mapping
  - Skin Mapping
  - Solid Texturing (3D mapping)
Parametric Mapping

- Also known as *decal* mapping
- Use the surface’s $u,v$ coordinates as the texture coordinates
Parametric Mapping

- Can transform the coordinates:

\[
\begin{bmatrix}
  s \\
  t \\
  1
\end{bmatrix} = \begin{bmatrix}
  a_s & b_s & d_s \\
  a_t & b_t & d_t \\
  0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
  u \\
  v \\
  1
\end{bmatrix}
\]
Orthographic Mapping

- Use object’s x, y, z coordinates
  - Perform orthographic projection to get s, t
  - Make sure to use object space!

\[
\begin{bmatrix}
  s \\
  t
\end{bmatrix} = \begin{bmatrix}
  a_s & b_s & c_s & d_s \\
  a_t & b_t & c_t & d_t
\end{bmatrix} \begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]
Perspective Mapping

- Perform a perspective projection of x,y,z coordinates
  - Often used in world space, for simulated lighting effects
  - Easy for texture to span multiple objects

\[
\begin{bmatrix}
  s' \\
t' \\
w
\end{bmatrix} = \begin{bmatrix}
a_s & b_s & c_s & d_s \\
a_t & b_t & c_t & d_t \\
a_w & b_w & c_w & d_w
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix};
\begin{bmatrix}
s \\
t
\end{bmatrix} = \begin{bmatrix}
s'/w \\
t'/w
\end{bmatrix}
\]
Spherical Mapping

- Many ways of mapping a sphere to a flat plane
  - all have some distortion
  - *polar mapping*: the s,t coordinates are longitude and latitude
- Place sphere around object
  - “shrink wrap” down onto the object
  - Useful for round objects, such as heads
Cylindrical Mapping

- Like spherical mapping, but with a cylinder
  - also useful for faces
Skin Mappings

- A variety of fancy mapping techniques to “unfold” surface
  - Conformal mappings (mappings that preserve angles)
  - Area preserving mappings
  - ...

[Image of skin mapping process with text: [Piponi2000]]
Solid Textures

- Instead of a 2D image, use 3D data
  - Often procedural -- 3D file would be too large
  - Common examples: marble, granite, wood
- Effect is like having carved the model out of a solid piece
Decals

- Overlay a texture on a model
  - Used for labels
  - Used in games for bullet holes, other “damage”
    - texture coordinates assigned on the fly
    - texture has transparency (alpha) so as to not be square
  - Typically orthographic or parametric projection
Texture Packing

- Put many different texture parts in one file
  - Often have limited number of texture maps available
  - Often have limited texture memory, want to use it all
  - May simply want just one file for convenience

charts
atlas
surface

[Sander2001]
Texture Mapping

- Store texture coordinates as vertex data
  - Coordinates interpolated to each pixel during scan conversion
  - Color looked up in texture map file, per pixel
  - Various ways to come up with the coordinate assignments

\[(1,1)\]
Texture Interpolation

- Given \((s, t)\) texture coordinates at vertices
- During scan conversion, interpolate bilinearly
  - Same as color and depth values:
    - compute \(ds/dy\) and \(dt/dy\) for the left and right edges
    - compute \(ds/dx\) and \(ds/dy\) for each scanline span
  - once we have interpolated \((s, t)\) value at the pixel
    - look up the color of that texel in the texture map
    - use the color for the pixel
    - or use the color as the base color for lighting
    - typically ignore the model’s per-vertex color data
- More work per pixel: texture coordinate interpolation, and data lookup
  - GPU has dedicated hardware to support this
  - Limited number/size of textures
Perspective Correction

- Linear interpolation in screen space isn’t linear in world space
  - Doesn’t properly capture foreshortening
  - If object moves or viewing angle changes, texture will warp/stretch within the triangle: *texture swimming*
- Solutions:
  - Break up into more triangles. Avoid having individual triangles that span large Z changes.
  - Interpolate in homogeneous coordinates:
    - *hyperbolic interpolation*
    - Interpolate $s^*, t^*, w$ and perform perspective divide

![Diagram showing perspective correction with and without hyperbolic interpolation](image.png)
Tiling

- Image exists from (0,0) to (1,1) in texture space
- (s,t) texture coordinates may go outside that range
- *Tiling* or *wrapping* rules for out-of-range coordinates
Tiling

- Repeat the texture.
- There will be seams unless the texture lines up on the edges

Texture Space
Clamping

- Use the edge value everywhere outside the data
- Or, ignore the texture outside 0-1
  - Use this for decals

Texture Space
Mirroring

- Flip left-to-right and top-to-bottom
  - makes all the edges line up

Texture Space

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(1,1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0,0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Magnification

- As object approaches the camera
  - Each texel might cover several pixels
  - Image of texture on screen is larger than original texture
- Known as *magnification*
- Can define various magnification behaviors such as:
  - Point sampling
  - Bilinear sampling
  - Bicubic sampling
  - known as *filtering* (we’ll come back to this later in the course)
Magnification

- **Point sampling**:  
  - Each pixel uses the color of the texel at the texture coordinate  
  - Causes the individual texels to appear as solid colored rectangles

- **Bilinear interpolation**:  
  - Each pixel uses a bilinear blend of the nearest 4 texel centers.  
  - Texture appears smoother when viewed up close  
  - When viewed too close, the bilinear blending can be noticeable

- **Bicubic interpolation**:  
  - Sample a 4x4 grid of texels and perform a bicubic blend.  
  - Smoother (C1-continuous)  
  - More expensive
Magnification: point sampling
Magnification: Bilinear interpolation
Minification

- When far away, many texels may map to a single pixel.
  - known as *minification*
  - What color should we use?
- Also, when we view flat surfaces edge-on:
  - Texels map to pixels in a very stretched way
  - Can be handled by similar techniques to minification
Minification

- Point sampling:
  - Use each pixel uses color of the texel at the texture coordinate
  - Causes parts of the texture to be skipped
  - Visual effect: shimmering or buzzing
  - A form of *aliasing*, more about this in a later class
Minification: point sampling
Minification

- Ideally, final color should be a blend of all texels that fall in the pixel
- This would be expensive to compute
  - Large memory access cost
  - Computation cost for blending all the texels
  - Gets worse as objects get farther hence smaller: more texels per pixel

- **Mipmapping**
  - A level-of-detail method
  - “mip” stands for “multum in parvo” (Lance Williams, 1983)
  - Reasonable compromise between performance and quality
  - Used in most graphics hardware
  - Used in most software renderers too
Mipmapping

- In addition to storing the texture image, store several **mipmaps**
  - Each mipmap is a scaled down-version of the original image
  - Computed with a decent quality algorithm (such as bicubic interpolation)
  - Each mipmap is half the resolution of the previous, in both x and y
- For example, if we have a 512x512 texture, store 9 mipmaps:
  - 256x256, 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2, 1x1
- This adds 1/3 extra memory per texture (1/4 + 1/16 + 1/64…= 1/3)
- Generally, use texturemaps with power-of-two sizes
  - If not, first mipmap can be original rounded down to nearest power of 2
  - Non-square textures are OK.
    - E.g. 16x4 texture would have mipmaps: 8x2, 4x1, 2x1, 1x1
Mipmaps
Mipmapping

- To render using a mipmapped texture:
  - Computed interpolated (s,t) texture coordinate as before
  - Determine texel (s,t) span from ds/dx, ds/dy, dt/dx, dt/dy
    - e.g. if span is roughly 10, 10, 10, 10 texels, use the 8x8 mipmap
  - Determine mipmap in which span is closest to one texel
  - Use that mipmap, point sampled or with bilinear interpolation

- What does that look like…
Nearest mipmap, point-sampled
Nearest mipmap, bilinear interpolation
Trilinear mipmapping

- Nearest mipmap: visible “pop” between mipmap levels
- Instead:
  - Choose the 2 nearest mipmaps (one larger, one smaller)
    - e.g., if ds and dt values are about 10, choose 8x8 and 16x16 mipmaps
  - Perform bilinear interpolation in both mipmaps
  - Perform linear blend between the two results
  - known as *trilinear mipmapping*
  - The preferred method, supported by hardware.
  - Requires 8 texels to be sampled per pixel
Trilinear mipmapping
Mipmapping Limitation

- Each mipmap level is uniformly smaller in s and t
- When looking at surfaces edge-on, texel span might not be square
  - e.g., ds/dx, dt/dx, might be 10,10 but ds/dy, dt/dy might be 3,3
  - to avoid buzzing in y, will blend between 2x2 and 4x4 mipmaps
  - but this is too blurry in x.
- Net effect: when objects are edge-on, texture gets blurry
Anisotropic Mipmapping

- Store rectangular mipmaps in addition to square
  - Usually limit to aspect ratios of 2x1 and 4x1
  - E.g. a 512x512 texture with 4x1 anisotropic mipmapping would have:
    - 512x512, 512x256, 256x512, 512x128, 128x512, 256x256, 256x128, 128x256, 256x64, 64x256, 128x128, 128x64, 64x128, 128x32, 32x128, 64x64, 64x32, 32x64, 64x16, 16x64, 32x32, 32x16, 16x32, 32x8, 8x32, 16x16, 16x8, 8x16, 16x4, 4x16, 8x8, 8x4, 4x8, 8x2, 2x8, 4x4, 4x2, 2x4, 4x1, 1x4, 2x2, 2x1, 1x2, 1x1

- Regular mipmapping (1x1 aspect) adds 1/3 memory per texture
- 2x1 aspect adds 2/3 extra memory per texture
- 4x1 aspect adds 5/6 extra memory per texture

- Improves image quality for edge-on viewing, but not perfect
  - improved when the view is lined up with x or y
  - blurs for diagonal views
  - if camera or object rotates, blur could vary noticeably

- Supported by modern graphics hardware.
Elliptical Weighted Averaging

- In hardware, anisotropic mipmapping is the state of the art today
- In software, some enhancements are available
- Highest quality technique (though not the fastest!) is Elliptical Weighted Averaging (EWA)
- Observation: A circle in pixel space maps to an ellipse in texture space
  - We want to sample all of the texture that falls within the pixel,
  - But weight the texels higher based on how close they are to the center of the pixel
  - Use a radial distribution weighting function to define how the pixel is sampled.
  - Each concentric circle in this distribution maps to a ellipse in texture space
- Direct EWA: sample all the texels, weighted by location in the ellipse region
  - Theoretically best way to sample the texture
  - But prohibitively expensive in practice
- Mipmapped EWA techniques exist
  - Potential to provide best of both: accurate while reasonably efficient
  - Not (yet) used in production hardware
Outline for today

- Lighting & Shading wrapup
- Texture Mapping
- *Procedural Textures*
- Fancy Texture Effects
Procedural textures

- Stored-data texture maps aren’t always convenient
  - Use lots of memory
  - Must choose fixed resolution in advance:
    - Resolution too low -- image is blocky or blurry
    - Resolution too high -- uses too much memory
  - Not easy to have lots of minor variations
    - Each would need its own texture map

- **Procedural textures**
  - Compute color value on the fly, in a procedural shader
  - Can take into account pixel size
  - Can have parameters that control the behavior
Procedural Textures
noise functions

- Procedural shaders can do the usual programming tasks
  - including looking data up in texture maps
  - including calculating lighting
- Particularly useful are pseudo-random “noise” functions
  - *Perlin noise* (Ken Perlin, NYU, 1983)
    - smoothly varying pattern
    - useful for swirly turbulent effects
  - *Worley cellular noise* (Steve Worley, 1996)
    - pattern with edges
    - useful for rocky scaly effects
- Both compute very quickly
  - Typically used by combining several calls to noise at different scales, to create a fractal pattern.
Perlin Noise
Worley Cellular Noise
Procedural Texture

- a RenderMan shader:

```plaintext
surface turbulence (float Kd=.8, Ka=.2)
{
    float a,scale,sum;
    point M;
    M = transform( "marble", P );
    scale = 1;
    sum = 0;
    a = sqrt(area(M));
    while( a < scale ) {
        sum += scale * float noise(M/scale);
        scale *= 0.5;
    }
    Oi = sum;
    Ci = Cs * Oi * (Ka + Kd * I.N * I.N / (I.I * N.N));
}
```

- xform to object space
- fractal
- Perlin noise
- diffuse lighting (variant)
Outline for today

- Lighting & Shading wrapup
- Texture Mapping
- Procedural Textures
- *Fancy Texture Effects*
Fancy Texture Effects

- We can look up data at each pixel… What can we do with it?

- Given procedural shaders, can do most anything we want!

- Here are some common techniques
  - Often supported by renderer without procedural shading
  - Some supported directly by hardware
Environment Mapping

- A simple but technique to fake mirror-like reflections of an environment
- Precompute, photograph, or paint an environment map:
  - A view of the distant environment (ground, sky, horizon, etc.) from the center of the scene
  - Can be stored in a single spherically-projected texture
  - Can be stored in 6 faces of a cube
- Imagine that the scene is enclosed in a huge sphere or cube, textured with that map
- For each vertex or point to be shaded:
  - compute the vector $e$ from the point to the eye
  - compute the reflection vector $R$
  - find out where $R$ intersects the environment cube/sphere, and use that texture coordinate
    - (because the environment is huge, we don’t need to take into account the position of the point)
  - add the texture color to the point’s color, with some constant $k_e$
Environment Mapping

For a spherical environment with polar mapping:

\[ \mathbf{R} = 2(\mathbf{e} \cdot \mathbf{n})\mathbf{n} - \mathbf{e} \]

\[ s = \frac{\text{atan2}(R_z, R_x) + \pi}{2\pi} \]

\[ t = \frac{\text{asin}(R_y) + \pi/2}{\pi} \]
Environment Mapping
Environment Mapping

(www.sparse.org)
Bump Mapping

- An easy way to make a smooth surface bumpy
  - Use a texture to represent the variation in surface height
  - With Phong interpolation we have a normal for each pixel
  - Use the texture value to perturb the normal
  - Then use the perturbed normal for the per-pixel lighting

- Texture can be stored or procedural
- Used for rough surfaces
- Used for “embossing”
- Modern hardware supports bump mapping

- Limitation: bumps are fake
  - silhouette edges betray smoothness
Bump mapping
Displacement Mapping

- Like bump mapping
  - but instead of faking by perturbing normals…
  - …actually move the surface point
  - Gives proper silhouettes,
  - Gives self-occlusion
  - Gives self-shadowing (once we have shadows…)

- Expensive and hard to do well
  - Supported by some software renderers
  - Supported by some recent hardware
Displacement Mapping
Other Texture Effects

- Textures can be used to map most any property onto a surface
  - Not just color
  - Bump or displacement
  - Diffuse coefficient? Specular Coefficient?
  - Parameters to procedural patterns
- Many texture maps can be used at once
  - Different layers of effect
  - e.g. Base color, then smudges, scratches, dents, rust, etc.
    - each might affect color, bumps, displacement, lighting, reflectivity, ...
  - Hardware systems have limited amounts of texture memory
  - In production, it’s not uncommon to have a dozen or more textures on each surface
- Texture maps can themselves be computed or animated
  - E.g. To simulate effects such as patina and aging
  - E.g. To show a TV picture in an animation:
    - each frame, use a different texture map on the TV
  - E.g. For raindrops dripping down a window
Done

- Next class: Shadows