BRDF in Matrix II & III
Spatial Variation

• All materials seen so far are the same everywhere
  – In other words, we’re assuming the BRDF is independent of the surface point x
  – No real reason to make that assumption
Spatial Variation

- We’ll allow the BRDF parameters to vary over space
  - This’ll give us much more complex surface appearance
- e.g. diffuse color $k_d$ vary with $x$
- Other parameters/info can vary too: $ks$, exponent, normal
Two approaches

• from data: texture mapping
  – read color and other information from 2D images

• procedural: shader
  – write little programs that compute color/info as a function of location
Effect of Textures

For more info on the computer artwork of Jeremy Birn see [http://www.3drenderer.com/jbirn/productions.html](http://www.3drenderer.com/jbirn/productions.html)
Texture Mapping

3D model

Texture mapped model

Image: Praun et al.
Texture Mapping

Texture mapped model

We need a function that associates each surface point with a 2D coordinate in the texture map

Texture map (2D image)
Texture Mapping

Texture mapped model

For each point rendered, look up color in texture map

Texture map (2D image)
UV Coordinates

• Each vertex $P$ stores 2D $(u, v)$ “texture coordinates”
  – UVs determine the 2D location in the texture for the vertex
  – We’ll see how to specify them later

• Then we interpolate using barycentrics

\[
(\alpha u_0 + \beta u_1 + \gamma u_2, \\
\alpha v_0 + \beta v_1 + \gamma v_2)
\]
UV Coordinates

- Each vertex P stores 2D (u, v) “texture coordinates”
  - UVs determine the 2D location in the texture for the vertex
  - We’ll see how to specify them later
- Then we interpolate using barycentrics
Pseudocode – Ray Casting

- Ray cast pixel \((x, y)\), get visible point and \(\alpha, \beta, \gamma\)
- Get texture coordinates \((u, v)\) at that point
  - Interpolate from vertices using barycentrics
- Look up texture color using UV coordinates
UV Coordinates?

- Per-vertex \((u, v)\) “texture coordinates” are specified:
  - Manually, provided by user (tedious!)
  - Automatically using parameterization optimization
  - Mathematical mapping (then not necessarily per vertex)
Texture UV optimization

- Goal: “flatten” 3D object onto 2D UV coordinates
- For each vertex, find coordinates U,V such that distortion is minimized
  - distances in UV correspond to distances on mesh
  - angle of 3D triangle same as angle of triangle in UV plane
- Cuts are usually required (discontinuities)

Figure 1.2: Application of parameterization: texture mapping for anthropomorphic maps implemented in the Open-Source Blender modeler.
To learn more

- For this course, assume UV given per vertex
- “Mesh Parameterization: Theory and Practice”
  Kai Hormann, Bruno Lévy and Alla Sheffer
  ACM SIGGRAPH Course Notes, 2007
  redirect=0&Paper=SigCourseParam@2007&Author=
  Levy
Creating Torso Portion in Max

3D Model

UV Mapping
3D model

• Information we need:
• Per vertex
  – 3D coordinates
  – Normal
  – 2D UV coordinates
• Other information
  – BRDF (often same for the whole object, but could vary)
  – 2D Image for the texture map
Questions?

Figure 4.8: Some results computed by stretch $L_2$ minimization (parameterized models courtesy of Pedro Sander and Alla Sheffer).
Mathematical mapping

• What of non-triangular geometry?
  – Spheres, etc.

• No vertices, can’t specify UVs that way!

• Solution: Parametric Texturing
  – Deduce (u, v) from (x, y, z)
  – Various mappings are possible....
Common Texture Coordinate Mappings

- **Planar**
  - Vertex UVs and linear interpolation is a special case!

- **Cylindrical**

- **Spherical**

- **Perspective Projection**
Projective Mappings

• A slide projector
  – Analogous to a camera!
  – Usually perspective projection tells us where points project to in our image plane
  – This time we’ll use these coordinates as UVs

• No need to specify texture coordinates explicitly
Projective Mappings

• We are given the camera matrix $H$ of the slide projector
• For a given 3D point $P$
• Project onto 2D space of slide projector: $HP$
  – results in 2D texture coordinates
Projective Texture Example

• Modeling from photographs
• Using input photos as textures

Figure from Debevec, Taylor & Malik
http://www.debevec.org/Research

MIT EECS 6.837 – Durand

Tuesday, November 9, 2010
Questions?
Texture Tiling

- Specify a texture coordinate \((u,v)\) at each vertex
- Canonical texture coordinates \((0,0) \rightarrow (1,1)\)
  - Wrap around when coordinates are outside \((0,1)\)

Note the range \((0,1)\) unlike normalized screen coordinates!
Questions?
Texture Mapping & Illumination

• Texture mapping can be used to alter some or all of the constants in the illumination equation
  – Diffuse color $k_d$, specular exponent $q$, specular color $k_s$...
  – Any parameter in any BRDF model!

$$L_o = \left[ k_a + k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{v} \cdot \mathbf{r})^q \right] \frac{L_i}{r^2}$$

  – $k_d$ in particular is often read from a texture map
Gloss Mapping Example

Spatially varying $k_d$ and $k_s$
Questions?
We Can Go Even Further...

• The normal vector is really important in conveying the small-scale surface detail
  – Remember cosine dependence
  – The human eye is really good at picking up shape cues from lighting!

• We can exploit this and look up also the normal vector from a texture map
  – This is called “normal mapping” or “bump mapping”
  – A coarse mesh combined with detailed normal maps can convey the shape very well!
Normal mapping

- For each shaded point, normal is given by a 2D image `normalMap` that stores the 3D normal

For a visible point

interpolate UV using barycentric

// same as texture mapping

Normal = normalMap[U,V]

compute shading (BRDF) using this normal

\[
L_o = \left[ k_a + k_d (n \cdot l) + k_s (v \cdot r)^q \right] \frac{L_i}{r^2}
\]
Normal Map Example

Original Mesh
4M triangles
Normal Map Example

Simplified mesh, 500 triangles

Simplified mesh + normal mapping
Normal Map Example

Models and images: Trevor Taylor

Final render

Diffuse texture $k_d$

Normal Map

MIT EECS 6.837 – Durand
Generating Normal Maps

• Model a detailed mesh
• Generate a UV parametrization for the mesh
  – A UV mapping such that each 3D point has a unique image in the 2D texture map
  – This is a difficult problem, but tools are available
    • E.g., the DirectX SDK has functionality to do this
• Simplify the mesh (again, see DirectX SDK)
• Overlay simplified and original model
• For each point \( P \) on the simplified mesh, find closest point \( P' \) on original model (ray casting)
• Store the normal at \( P' \) in the normal map. Done!
Normal Map Details

• You can store an object-space normal
  – Convenient if you have a unique parameterization
• ....but if you want to use a tiling normal map, this won’t do
  – Must account for the curvature of the object!
  – Think of mapping this diffuse+normal map combination on a cylindrical tower
• Solution: Tangent space normal map
  – Encode a “difference” from the geometric normal in a local coord. system
Questions?
Questions?
Shaders (Material class)

- Functions executed when light interacts with a surface
- Constructor:
  - set shader parameters
- Inputs:
  - Incident radiance
  - Incident & reflected light directions
  - Surface tangent basis (anisotropic shaders only)
- Output:
  - Reflected radiance
Shader

• Initially for production (slow) rendering
  – Renderman in particular

• Now used for real-time (Games)
  – Evaluated by graphics hardware
  – More later in the course

• Often makes heavy use of texture mapping
Questions?
Procedural Textures

• Alternative to texture mapping
• Little program that computes color as a function of x, y, z:
  \[ f(x, y, z) \rightarrow \text{color} \]

Image by Turner Whitted
Procedural Textures

• Advantages:
  – easy to implement in ray tracer
  – more compact than texture maps (especially for solid textures)
  – infinite resolution

• Disadvantages
  – non-intuitive
  – difficult to match existing texture
Questions?
Perlin noise

- Critical component of procedural textures
- Pseudo-random function
  - But continuous
  - band pass (single scale)
- Useful to add lots of visual detail

http://www.noisemachine.com/talk1/index.html
http://mrl.nyu.edu/~perlin/doc/oscar.html
http://mrl.nyu.edu/~perlin/noise/
http://en.wikipedia.org/wiki/Perlin_noise
http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
  (not really Perlin noise but very good)
http://portal.acm.org/citation.cfm?id=325247
Requirements

- Pseudo random
- For arbitrary dimension
  - 4D is common for animation
- Smooth
- Band pass (single scale)
- Little memory usage

- How would you do it?
Perlin noise

• Cubic lattice
• Zero at vertices
  – To avoid low frequencies
• Pseudo-random gradient at vertices
  – define local linear functions
• Splines to interpolate the values to arbitrary 3D points
1D noise

- 0 at integer locations
- pseudo-random derivative (1D gradient) at integer locations
  - define local linear functions
- Interpolate at location P

MIT EECS 6.837, Durand

Tuesday, November 9, 2010
1D noise: reconstruct at P

- dx: fractional x coordinate
- Gradients $G_1$ and $G_2$ at neighboring vertices
  - scalar in 1D, will be 3D vectors in 3D
- We know that noise is zero at vertices

MIT EECS 6.837, Durand
1D noise: reconstruct at P

- Compute the values from the two neighboring linear functions: \( n_1 = dx \times G_1 \); \( n_2 = (dx - 1) \times G_2 \)
  - dot product in 3D
1D noise: reconstruct at P

- Compute the values from the two neighboring linear functions: $n_1 = dx \cdot G_1$; $n_2 = (dx-1) \cdot G_2$
  - dot product in 3D
- Weight $w_1 = 3dx^2 - 2dx^2$ and $w_2 = 3(1-dx)^2 - 2(x-1)^2$
  - ie: $\text{noise} = w_1 \cdot G_1 \cdot dx + w_2 \cdot G_2 \cdot (dx-1)$
Algorithm in 3D

• Given an input point P
• For each of its neighboring grid points:
  – Get the "pseudo-random" gradient vector G
  – Compute linear function (dot product G.dP)
• Take weighted sum, using separable cubic weights
  – [demo in 2D]
Computing the pseudo-random gradient

- Precompute (1D) table of n gradients \( G[n] \)
- Precompute (1D) table of permutations \( P[n] \)
- For 3D grid point \( i, j, k \):
  \[
  G = G[ (i + P[ (j + P[k]) \mod n ] ) \mod n ]
  \]

- in practice only 256 gradients are stored!
  - But optimized so that they are well distributed
Noise at one scale

- A scale is also called an octave in noise parlance
Noise at multiple scales

- A scale is also called an octave in noise parlance
- But multiple octaves are usually used, where the scale between two octaves is multiplied by 2
  - hence the name octave
Sum 1/f noise

- That is, each octave f has weight 1/f
Sum 1/f \textit{noise}

- Absolute value introduces C1 discontinuities
- aka turbulence
\[
\sin (x + \sum \frac{1}{f} |\text{noise}|)
\]

- Looks like marble!
Comparison

• noise

\[ \sin(x + \sum \frac{1}{f(|\text{noise}|)}) \]

MIT EECS 6.837, Durand
Question?
Noise for solid textures

- **Marble**
  - recall $\sin(x + \sum 1/f |\text{noise}|)$
  - i.e. parallel plane + noise
  - Color = color map ($x + \text{turbulence}$)
  - [http://legakis.net/justin/MarbleApplet/](http://legakis.net/justin/MarbleApplet/)

- **Wood**
  - replace $x$ (or parallel plane) by radius
  - Color = color map ($r + \text{turbulence}$)
  - [http://www.connectedpixel.com/blog/texture/wood](http://www.connectedpixel.com/blog/texture/wood)
Corona

• The corona was made as follows:
  – Create a smooth gradient function the drops off radially from bright yellow to dark red.
  – Phase shift this function by adding a turbulence texture to its domain.
  – Place a black cutout disk over the image.

• Animation
  – Scale up over time
  – Use higher dim noise (for time)
    – http://www.noisemachine.com/talk1/imgs/flame500.html
Other cool usage: displacement, fur
Question?
Shaders

- Noise: one ingredient of shaders
- Can also use textures
- Shaders control diffuse color, but also specular components, maybe even roughness (exponent), transparency, etc.
- Shaders can be layered (e.g. a layer of dust, peeling paint, mortar between bricks).
- Notion of shade tree
  - Pretty much algebraic tree
- Assignment 5: checkerboard shader based on two shaders
Bottom line

• Programmable shader provide great flexibility
• Shaders can be extremely complex
  – 10,000 lines of code!
• Writing shaders is a black art