Texturing

Outline

• Texture mapping
  - Going from 3-D to 2-D
  - Rasterization issues
What is Texture Mapping?

• Spatially-varying modification of surface appearance

• Characteristics
  - Bumpiness
  - Shininess
  - Transparency
  - Etc.

Texture mapping: Examples

Examples of bumpiness, shininess, transparency with identical sphere geometry
Why use texture mapping?

- More detail without the cost of more complicated geometry
  - Modeling
  - Display
- “Lookup table” for precomputed quantities
  - Lighting
  - Shadows

Texture mapping applications: Lightmaps

![Image](image_url)
Texture mapping: Steps

- **Creation**: Where does the texture image come from?
- **Geometry**: Transformation from 3-D shape locations to 2-D texture image coordinates
- **Rasterization**: What to draw at each pixel

Texturing: Creation

- Photographs, or handpainted
- Synthesis
  - Lightmaps
  - Procedurally-built
    - Randomness, pattern-generating rules, etc.
Texturing: Geometry

1. Compute object space location \((x, y, z)\) from screen space location \((i, j)\)
2. Use \textbf{projector} function to obtain \((u, v)\)
3. \textbf{Corresponder} function to find texel coordinates \((s, t)\)

Projector Functions

- Idea: Way to get from 3-D point to 2-D surface coordinates as an intermediate step
- Project from \textbf{inside} object onto shape's surface
  - Plane
  - Cylinder
  - Sphere
  - Cube
Planar projector

Orthographic projection onto XY plane:
\[ u = x, \ v = y \]

...onto YZ plane ...onto XZ plane

Cylindrical projector

- Convert rectangular coordinates \((x, y, z)\) to cylindrical \((r, \theta, h)\), use only \((\theta, h)\) to index texture image
Cylindrical projectors with different axes

Spherical projector

- Convert rectangular coordinates \((x, y, z)\) to spherical \((\theta, \phi)\)
Spherical projectors

Cube projector

• Use intersection with cube of ray from center of shape through \((x, y, z)\) on shape surface to pick one of six side textures
Cube projector

Corresponder function

- Maps from \((u, v)\) to texture image coordinates \((s, t)\) in units of pixels
- \((u, v)\) in range \([0, 1]\) are displayed; corresponder decides:
  - What part of texture image this references (all or only part)
  - Wrapping: How to handle values outside range \([0, 1]\) (including negative)
Corresponder function

- Maps from \((u, v)\) to texture image coordinates \((s, t)\) in units of pixels
- \((u, v)\) in range \([0, 1]\) are displayed; corresponder decides:
  - What part of texture image this references (all or only part)
  - Wrapping: How to handle values outside range \([0, 1]\) (including negative)

Wrapping modes

- **repeat**: Start entire texture over
- **mirror**: Flip copy of texture in each direction
  - Get continuity of pattern
**Texturing: Rasterization**

- Assume we can compose the previous steps (projector, corresponder, etc.) to define a transformation function $M$ from integer screen coordinates $(i, j)$ to real-valued texture coordinates $(s, t)$ such that $M(i, j) = (s, t)$

**Nearest neighbor interpolation (NN)**

- Idea: Just use closest integer-valued texel
NN issues

- Problem is that it can cause big **aliasing** effects
- Why? Because the `round()` function causes discontinuous switches in which texel is nearest and hence is the color drawn

\[
\begin{align*}
(s, t) & \quad \quad \quad \quad (s+1, t) \\
(s+1, t) & \quad \quad \quad \quad (s+1, t+1) \\
(s, t+1) & \quad \quad \quad \quad (s+1, t+1)
\end{align*}
\]

NN aliasing

rotate 45°, scale 1.5
Bilinear Interpolation (BLI)

• Idea: Blend four texel values surrounding source, weighted by nearness

\[
I(i, j) = (1-b, b) \begin{bmatrix}
I_{tex}(s, t) & I_{tex}(s+1, t) \\
I_{tex}(s, t+1) & I_{tex}(s+1, t+1)
\end{bmatrix} \begin{bmatrix}
1-a \\
a
\end{bmatrix}
\]

Vertical blend  Horizontal blend

Bilinear interpolation

• Blending eliminates abrupt color changes, reducing aliasing artifacts

rotate 45\(^\circ\), scale 1.5
Texel Interpolation approaches:
NN vs. BLI

More rasterization techniques

• Filtering
• Interpolation
Magnification and minification

- **Magnification**: Single screen pixel maps to area less than or equal to one texel
- **Minification**: Single screen pixel area maps to area greater than one texel

Filtering for minification

- Aliasing problem much like line rasterization
  - Pixel maps to quadrilateral (**pre-image**) in texel space
Supersampling

- Rasterize at higher resolution
  - Regular grid pattern around each "normal" image pixel
- Combine multiple samples into one pixel via **weighted average**
  - "Box" filter: All samples associated with a pixel have equal weight (i.e., directly take their average)
  - Gaussian filter: Sample weights inversely proportional to distance from associated pixel

Mipmaps

- Filtering for minification is expensive
- Idea:
  - Prefilter entire texture image at different resolutions
  - For each screen pixel, pick texture in mipmap at **level of detail (LOD)** that minimizes minification (i.e., pre-image area closest to pixel area)
  - Do nearest or linear filtering in appropriate LOD texture image
Mipmaps in OpenGL

- Create with `gluBuild2DMipmaps()`
- Use by calling `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, param)`, where `param` is:
  - `GL_NEAREST_MIPMAP_NEAREST`: `GL_NEAREST` filtering in closest LOD mipmap
  - `GL_LINEAR_MIPMAP_NEAREST`: `GL_LINEAR` filtering in closest LOD mipmap