Texture Mapping

Computer Graphics
COMP 770 (236)
Spring 2007

Instructor: Brandon Lloyd
From last time...

- Triangle rasterization
  - Scan-line
  - Edge equations
- Parameter interpolation
- Beyond triangles
Topics for today

- Texture mapping overview
  - notation
  - wrapping

- Perspective-correct interpolation

- Texture filtering
  - Bilinear interpolation
  - MIP maps
  - Summed-area tables
Texture maps in OpenGL

- Specify normalized texture coordinates at each of the vertices \((u, v)\)
- Texel indices
  \((s,t) = (u, v) \cdot (\text{width, height})\)
- Previous to OpenGL 2.0, texture dimensions had to be a power of 2

```c
glBindTexture(GL_TEXTURE_2D, texID)
glBegin(GL_POLYGON)
  glTexCoord2d(0,1); glVertex2d(-1,-1);
  glTexCoord2d(1,1); glVertex2d( 1,-1);
  glTexCoord2d(1,0); glVertex2d( 1, 1);
  glTexCoord2d(0,0); glVertex2d(-1, 1);
glEnd()
```
The behavior of texture coordinates outside of the range \([0,1)\) is determined by the texture wrap options.

\[
\begin{align*}
\text{glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, wrap\_mode }) \\
\text{glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, wrap\_mode })
\end{align*}
\]
Linear interpolation of texture coordinates

- Simple linear interpolation of $u$ and $v$ over a triangle leads to unexpected results
  - Distortion when the triangle vertices do not have the same depth
Linear interpolation of texture coordinates

- Uniform steps along the edge projection in screen space do not correspond to uniform steps along the actual edge in eye space.
Linear interpolation of texture coordinates

**screen space**

\[ p(\tau_s) = p_1 + \tau_s (p_2 - p_1) \]

\[ = \frac{x_1}{z_1} + \tau_s \left( \frac{x_2}{z_2} - \frac{x_1}{z_1} \right) \]

**eye space**

\[ \dot{v}(\tau_e) = \dot{v}_1 + \tau_e (\dot{v}_2 - \dot{v}_1) \]

\[ p(\dot{v}(\tau_e)) = \frac{x(\tau_e)}{z(\tau_e)} = \frac{x_1 + \tau_e (x_2 - x_1)}{z_1 + \tau_e (z_2 - z_1)} \]
Correcting the interpolation

- We want to interpolate in eye space, but in terms of our screen space $\tau_s$. So we solve $p(\tau_s) = p(\vec{v}(\tau_e))$ for $\tau_e$ in terms of $\tau_s$:

$$p(\tau_s) = \frac{x_1}{z_1} + \tau_s \left( \frac{x_2}{z_2} - \frac{x_1}{z_1} \right) = \frac{x_1 + \tau_e (x_2 - x_1)}{z_1 + \tau_e (z_2 - z_1)} = p(\vec{v}(\tau_e))$$

$$\tau_e = \frac{\tau_s z_1}{z_2 + \tau_s (z_1 - z_2)}$$

- In screen space, we don’t have $z_1$ and $z_2$. But before the perspective divide we do have $w_1 = z_1$ and $w_2 = z_2$:

$$\tau_e = \frac{\tau_s w_1}{w_2 + \tau_s (w_1 - w_2)}$$
Correcting the interpolation

Now we plug this value of $\tau_e$ into the equation to used to linearly interpolate parameters like $(u,v)$ over the triangle in eye space:

$$u(\tau_e) = u_1 + \tau\epsilon (u_2 - u_1)$$

$$u(\tau_\theta) = u_1 + \frac{\tau_\theta w_1}{w_2 + \tau_\theta (w_1 - w_2)} (u_2 - u_1) = \frac{u_1 w_2 + \tau_\theta (u_2 w_1 - u_1 w_2)}{w_2 + \tau_\theta (w_1 - w_2)}$$

$$u(\tau_\varsigma) = \frac{u_1}{w_1} + \tau_\varsigma (\frac{u_2}{w_2} - \frac{u_1}{w_1}) \quad \text{; multiply by } \frac{w_1 w_2}{w_1 w_2}$$

Linearly interpolate the numerator and denominator separately and do the divide once per pixel

- Numerator is the linear interpolation of a parameter pre-divided by its corresponding $w$ value
- Denominator is the linearly interpolated $1/w$ value
Another formulation

\[
\begin{bmatrix} x & y & w \end{bmatrix}^T \Rightarrow \begin{bmatrix} x/w \\ y/w \end{bmatrix}
\]

Homogeneous (clip) coordinates \hspace{1cm} non-homogeneous (NDC) coordinates

- In clip coordinates, a triangle is still planar so a parameter can be interpolated over its surface as:
  \[
  u = \frac{A_u x + B_u y + C_u w}{1/A_o x + B_o y + C_o w}
  \]

- Division by \( w \) produces the perspective-correct interpolation equation in screen space:
  \[
  \frac{u/w}{1/w} = \frac{A_u x/w + B_u y/w + C_u}{A_o x/w + B_o y/w + C_o} = \frac{A_u x + B_u y + C_u}{A_o x + B_o y + C_o}
  \]
Another formulation

Because the coefficients are the same in homogeneous and non-homogeneous coordinates, we can compute them directly from homogeneous coordinates without doing a perspective divide:

\[
\begin{bmatrix}
A_u & B_u & C_u \\
\end{bmatrix} = \begin{bmatrix}
u_0 & u_1 & u_2 \\
y_0 & y_1 & y_2 \\
w_0 & w_1 & w_2 \\
\end{bmatrix}^{-1}
\]

The coefficients for interpolating 1 are given by using the parameter vector \([1 \quad 1 \quad 1]\)

\[
\begin{bmatrix}
A_o & B_o & C_o \\
\end{bmatrix} = \begin{bmatrix}
1 & 1 & 1 \\
y_0 & y_1 & y_2 \\
w_0 & w_1 & w_2 \\
\end{bmatrix}^{-1}
\]
Another formulation

- The edge equations can be defined as a pseudo-parameter that is 0 at the two end points and 1 at the opposite vertex

\[
\begin{bmatrix}
A_{\varepsilon_0} & B_{\varepsilon_0} & C_{\varepsilon_0} \\
A_{\varepsilon_1} & B_{\varepsilon_1} & C_{\varepsilon_1} \\
A_{\varepsilon_2} & B_{\varepsilon_2} & C_{\varepsilon_2}
\end{bmatrix}
= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix}
x_0 & x_1 & x_2 \\
y_0 & y_1 & y_2 \\
w_0 & w_1 & w_2
\end{bmatrix}^{-1}
\]

- After division by 1/w these edge equations give perspective-correct interpolation of barycentric coordinates

\[
\alpha_i = \frac{A_i}{A} \quad \alpha_0 + \alpha_1 + \alpha_2 = 1
\]

\[
u = \begin{bmatrix} u_0 & u_1 & u_2 \end{bmatrix}
\begin{bmatrix}
\alpha_0 \\
\alpha_1 \\
\alpha_2
\end{bmatrix}
\]
For obvious reasons, this method of interpolation is called **perspective-correct interpolation**.

Actually it is simply **correct interpolation**.

Not all 3-D graphics APIs implement perspective-correct interpolation.
Dealing with incorrect interpolation

- The perceived artifacts of non-perspective correct interpolation can be corrected by subdividing the texture-mapped triangles into smaller triangles.
  - why does this work?

- Screen-space interpolation of projected parameters is inherently flawed.
Sampling texture maps

- The uniform sampling pattern in screen space corresponds to some sampling pattern in texture space that is not necessarily uniform.
Sampling density mismatch

- Sampling density in texture space rarely matches the sample density of the texture itself

Oversampling (magnification)  Undersampling (minification)
Handling oversampling

How do we compute the color to assign to this sample?
Handling oversampling

- How do we compute the color to assign to this sample?
- Nearest neighbor – take the color of the closest texel
Handling oversampling

- How do we compute the color to assign to this sample?
- Nearest neighbor – take the color of the closest texel
- Bilinear interpolation

\[ \alpha = \frac{x-x_0}{x_1-x_0} \quad \beta = \frac{y-y_0}{y_1-y_0} \]

\[ c = ((1-\alpha)c_o + \alpha c_1)(1-\beta) + ((1-\alpha)c_2 + \alpha c_3)\beta \]
Undersampling

- Details in the texture tend to pop (disappear and reappear)
  - mortar in the brick
- High-frequency details lead to strange low-frequency patterns
  - aliasing
Spatial filtering

- To avoid aliasing we need to prefilter the texture to remove high frequencies
- Prefiltering is essentially a spatial integration over the texture
- Integrating on the fly is expensive
  - perform integration in a pre-process

samples and their extents

proper filtering removes aliasing
MIP mapping

- MIP is an acronym for the Latin phrase *multium in parvo*, which means "many in one place".

- Constructs an image pyramid. Each level is a prefiltered version of the level below resampled at half the frequency.

- While rasterizing use the level with the sampling rate closest to the desired sampling rate. (can also interpolate between pyramid levels).

- How much storage overhead is required?

\[
\text{mip map size} = \sum_{i=0}^{\infty} \left( \frac{1}{4} \right)^i = \frac{1}{1 - \frac{1}{4}} = \frac{4}{3}
\]
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
Finding the MIP level

- Use the projection of a pixel in screen into texture space to figure out which level to use

\[ u^*(x, y) = \frac{u}{w} = A_u x + B_u y + C_u \]
\[ o^*(x, y) = \frac{1}{w} = A_o x + B_o y + C_o \]
\[ s = u \cdot \text{texWidth} \]

\[ \frac{d}{dx} = \frac{du}{dx} \quad \frac{ds}{du} = \text{texWidth} \]
\[ \frac{du}{dx} = \frac{d}{dx} \left( \frac{u^*(x, y)}{o^*(x, y)} \right) = \frac{A_o o^*(x, y) - A_w u^*(x, y)}{o^*(x, y)^2} \]

Other derivatives can be found in the same way.
Finding the MIP level

- Use the lengths of the projected pixel in texture space to define a measure of mismatch between sampling densities:

\[ m = \max\left(\|\frac{dp}{dx}\|, \|\frac{dp}{dy}\|\right) = \max\left(\sqrt{\left(\frac{ds}{dx}\right)^2 + \left(\frac{dt}{dx}\right)^2}, \sqrt{\left(\frac{ds}{dy}\right)^2 + \left(\frac{dt}{dy}\right)^2}\right) \]

\[ \approx \max\left(\max\left(\left|\frac{ds}{dx}\right|, \left|\frac{dt}{dx}\right|\right), \max\left(\left|\frac{ds}{dy}\right|, \left|\frac{dt}{dy}\right|\right)\right) \]

- Now choose the appropriate level:

\[ l = \left\lfloor \log_2 m \right\rfloor \]

- Drawback of MIP maps - isotropic filtering
Texture filtering in OpenGL

- **Automatic creation**
  
  ```
  gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, width, height,
  GL_RGBA, GL_UNSIGNED_BYTE, data)
  ```

- **Manual creation**
  
  ```
  glTexImage2D(GL_TEXTURE_2D, level, 4, width, height, 0,
  GL_RGBA, GL_UNSIGNED_BYTE, data)
  ```

- **Filtering**
  
  ```
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, filter)
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, filter)
  ```

  where filter is:

  - `GL_NEAREST`
  - `GL_LINEAR`
  - `GL_LINEAR_MIPMAP_LINEAR`
  - `GL_NEAREST_MIPMAP_NEAREST`
  - `GL_NEAREST_MIPMAP_LINEAR`
  - `GL_LINEAR_MIPMAP_NEAREST`

  for MIN_FILTER only

  inter-level intra-level
Summed-area tables

- Another way performing the prefiltering integration on the fly
- Each entry in the summed area table is the sum of all entries above and to the left:

\[
\begin{array}{cccc}
1 & 6 & 8 & 3 \\
0 & 0 & 3 & 7 \\
4 & 7 & 8 & 8 \\
5 & 0 & 9 & 9 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 7 & 15 & 18 \\
1 & 7 & 18 & 28 \\
5 & 18 & 37 & 55 \\
50 & 23 & 51 & 78 \\
\end{array}
\]

What is the sum of the highlighted region?

\[T(x_1, y_1) - T(x_1, y_0) - T(x_0, y_1) + T(x_0, y_0)\]

Divide out area \((y_1 - y_0)(x_1 - x_0)\)
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 implement?
- Is this a good filter?
Next time

- Illumination and shading
  - empirical models