Projector Textures, Sampling 1

Dinesh K. Pai

Textbook Chapters 15.4, 16

Several slides courtesy of M. Kim

---

Today

- Announcements
  - No class on Monday. Remembrance Day
  - Reminder: Quiz 3 on Friday Nov 17
  - I will post a couple of Quiz 3 practice questions on Piazza by Tuesday. Will discuss answers on Wednesday
- Projector Texture mapping
- Sampling and Aliasing
Projector texture mapping

- There are times when we wish to glue our texture onto our triangles using a projector model, instead of the affine gluing model.
- For example, we may wish to simulate a slide projector illuminating some triangles in space.

Geometry of Projector Textures

Transformations are similar to shadow mapping.
Projector texture mapping

- The slide projector is modeled using 4 by 4, modelview and projection matrices, $M_s$ and $P_s$

$$\begin{bmatrix} x_t, w_t \\ y_t, w_t \\ - \\ w_t \end{bmatrix} = P_s M_s \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

Projector texture mapping

- With the texture coordinates defined as
  $$x_t = \frac{x_t w_t}{w_t} \quad \text{and} \quad y_t = \frac{y_t w_t}{w_t}$$

- To color a point on a triangle with object coordinates $[x_o, y_o, z_o, 1]^T$, we fetch the texture data stored at location $[x_t, y_t]^T$
Projector texture mapping

- The three quantities $x_w$, $y_w$, and $w_t$ are all affine functions of $(x_o, y_o, z_o)$. Thus these quantities will be properly interpolated over a triangle when implemented as varying variables.
- In the fragment shader, we need to divide by $w_t$ to obtain the actual texture coordinates.
- When doing projector texture mapping, we do not need to pass any texture coordinates as attribute variables to our vertex shader.

---

Projector texture mapping

- We simply use the object coordinates already available to us, and **compute the texture coordinates**.
- We do need to pass in, using uniform variables, the necessary projector matrices.
Projector texture mapping

- Projector vertex shader
  
  ```glsl
  #version 330
  
  uniform mat4 uModelViewMatrix;
  uniform mat4 uProjMatrix;
  
  uniform mat4 uSProjMatrix;
  uniform mat4 uSModelViewMatrix;
  
  in vec4 aVertex;
  out vec4 vTexCoord;
  
  void main(){
    vTexCoord = uSProjMatrix * uSModelViewMatrix * aVertex;
    gl_Position = uProjMatrix * uModelViewMatrix * aVertex;
  }
  ```

  Vertex shader generates texture coordinates!
  But not normalized

- Projector fragment shader
  
  ```glsl
  #version 330
  
  uniform sampler2D vTexUnit0;
  
  in vec4 aTexCoord;
  out vec4 fragColor;
  
  void main(){
    vec2 tex2;
    tex2.x = vTexCoord.x/vTexCoord.w;
    tex2.y = vTexCoord.y/vTexCoord.w;
    vec4 texColor0 = texture2D(vTexUnit0, tex2);
    fragCoor = texColor0;
  }
  ```
Projector texture mapping

- Conveniently, OpenGL even gives us a special call `texture2DProj(vTexUnit0, pTexCoord)`, that actually does the divide for us.
- Inconveniently, when designing our slide projector matrix `uSProjMatrix`, we have to deal with the fact that the canonical texture image domain in OpenGL is the unit square, whose lower left and upper right corners have coordinates `[0,0]^t` and `[1,1]^t` used for the display window.

Texture mapping tips and learning resources

- Read Texture Viewport (Textbook 12.3)
- Check out this excellent demo of transformations: [http://www.realtimerendering.com/udacity/transforms.html](http://www.realtimerendering.com/udacity/transforms.html)
- Nice online animations of many things we cover in this course, esp. related to textures [http://acko.net/files/fullfrontal/fullfrontal/webglmath/online.html](http://acko.net/files/fullfrontal/fullfrontal/webglmath/online.html)
Sampling

Two views of images

- A continuous image, $I(x_w, y_w)$, is a bivariate function.
  - range is a linear color space.
- A discrete image $I[i][j]$ is a two dimensional array of color values.
- We associate each pair of integers $i, j$, with the continuous image coordinates $x_w = i$ and $y_w = j$
Sampling

- The simplest and most obvious method to go from a continuous to a discrete image is by point sampling.
- To obtain the value of a pixel $i, j$, we sample the continuous image function at a single integer valued domain location:
  \[ I[i][j] \leftarrow I(i,j) \]
- This can result in unwanted artifacts.

Aliasing and anti-aliasing
Aliasing

- Scene made up of black and white triangles: jaggies at boundaries
  - Jaggies will crawl during motion
  - If triangles are small enough then we get random values or weird patterns.

Aliasing

- The heart of the problem: too much information in one pixel
Anti-aliasing

- Intuitively: the single sample is a bad value, we would be better off setting the pixel value using some kind of average value over some appropriate region.
- In the above examples, perhaps some gray value.

Mathematically this can be modeled using Fourier analysis.
- Breaks up the data by “frequencies” and figures out what to do with the un-representable high frequencies.
Box filter

- We often choose the filters \( F_{i,j}(x,y) \) to be something non-optimal, but that can more easily computed with.
- The simplest such choice is a box filter, where \( F_{i,j}(x,y) \) is zero everywhere except over the 1-by-1 square center at \( x = i, y = j \).
- Calling this square \( \Omega_{i,j} \), we arrive at

\[
I[i][j] \leftarrow \iint_{\Omega_{i,j}} I(x,y) \, dx \, dy
\]

Box filter

- In this case, the desired pixel value is simply the average of the continuous image over the pixel’s square domain.
Over-sampling

- Even that integral is not really easy to compute
- Instead, it is approximated by some sum of the form:
  \[ I[i][j] \leftarrow \frac{1}{n} \sum_{k=1}^{n} I(x_k, y_k) \]
  where \( k \) indexes some set of locations \((x_k, y_k)\) called the sample locations.
- The renderer first produces a “high resolution” color and z-buffer “image”,
  - where we will use the term *sample* to refer to each of these high resolution pixels.