COEN 290 Computer Graphics I

Santa Clara University

Evening's Goals

- Discuss displaying and reading image primitives
- Describe texture mapping
- Discuss OpenGL modes and settings for texture mapping

Image Primitives

- Color information for every pixel primitive
  - lots of information - lots of storage
- Passed to OpenGL as an array of color values
- OpenGL doesn't understand image file formats

Rendering Image Primitives

- Position image primitive in world coordinates
- Pass image data to OpenGL
  - images come in two forms
    - bitmaps
    - color images

Positioning Images

```
glRasterPos3f( x, y, z );
```

- Transformed just like a vertex
- Raster position is lower left corner of image
- Images are clipped based upon whether their raster position in inside the viewport

Corner aligned with Raster Position

Rendering Bitmaps

- Bitmaps are 1-bit deep images
- Use the current color to update pixel in framebuffer
  - sort of like a pixel mask

```
glBitmap( width, height, xorig,
  yorig, xmove, ymove, bitmap );
```

Rendering Images

glDrawPixels(*width, height, format, type, pixels*);

- Write color information directly into the framebuffer
- *format* describes how pixels are laid out in host memory
- *type* is storage type of *pixels* array

Rendering Image Example

```
GLubyte checkerBoard[2][2][3] = {
  {0,0,0},{1,1,1},{1,1,1},{0,0,0}
};

glRasterPos2f( 100.0, 273.0 );
glDrawPixels( 2, 2, GL_RGB,
  GL_UNSIGNED_BYTE, checkerBoard );
```

Reading Pixels

glReadPixels(*x, y, width, height, format, type, pixels*);

- Copy pixels from the framebuffer into host memory
- *(x,y)* is the window coordinate of lower left corner of block of pixels

Zooming Images

- Images can be zoomed in or out

```
glPixelZoom( xZoom, yZoom );
```

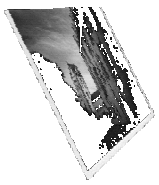
$$\begin{cases} zoom < -1 & \text{reflect and stretch} \\ -1 < zoom < 0 & \text{reflect and shrink} \\ 0 > zoom > 1 & \text{shrink} \\ zoom > 1 & \text{strecth} \end{cases}$$

Images Have Some Limitations

- Always aligned to the window
- Can not be rotated easily
  - no concept of world space
- Pixel zoom inexact
  - in particular, difficult to match world space dimensions
- Some performance limitations

A Much More Flexible Solution

- Imagine combining geometric rendering but using an image for the shading
- Principal idea behind *texture mapping*
  - also known as *image mappings*



Benefits of Texture Mapping

- Greater realism with less modeling
  - without texture mapping, need to model geometry at pixel resolution
- Fairly simple to implement
  - interpolate indices into texture map
    - similar interpolation to Gouraud shading

Disadvantages with Texture Mapping

- Considerable performance requirements
  - easily implemented in hardware, but filling pixels require much more work than Gouraud shading
  - dedicated texture memory in graphics hardware is usually a limited resource
- Aliasing
  - requires texture filtering
    - more math per pixel

Texture Mapping

- Use image data to determine the color of pixels for rasterizing geometric primitives
- Can texture in 1, 2 or 3 dimensions
  - 3D texturing very useful for *volume rendering*
- Two step process:
  ① set up image as a texture map
  ② render geometry with *texture coordinates* to reference texture

Passing Image Data as a Texture

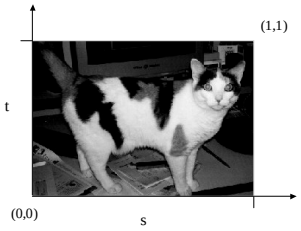
`glTexImage2D``( target, level, components, width, height, border, format, type, pixels );`

- images must have dimensions $2^n \times 2^m$
  - may additionally have a one pixel border around image
- set `target` to `GL_TEXTURE_2D`
- set `level` to `0`
  - we'll use `levels` later when we talk about *mipmaps*

Other Parameters

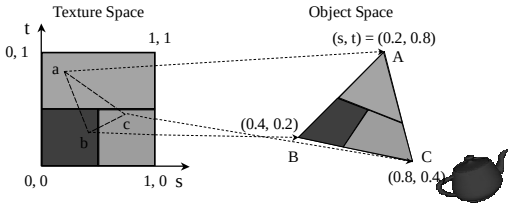
- `components` represents number of color channels per pixel
  - generally
    - 1 for intensity images
    - 3 for RGB images
  - may include an alpha channel
- `format` represents how *texels* are stored in memory
  - *texel* is short for *texture element*

Texture Space



Texture Coordinates

- *Texture coordinates* determine which texels are selected from the texture map.



Specifying Texture Coordinates

- Methods
  - explicitly specify texture coordinates
    - more control
    - more modeling
  - have OpenGL generate them for us
    - easy
    - limited applicability

Explicit Texture Coordinates

```
glTexCoord2f( s, t );
```

- Vertex attribute
  - like color or lighting normals

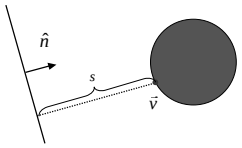
```
glBegin( GL_TRIANGLES );
  glTexCoord2f( 0, 0 );
  glVertex3fv( v1 );
  glTexCoord2f( 0, 1 );
  glVertex3fv( v2 );
  …
glEnd();
```

Generating Texture Coordinates

- OpenGL can generate texture coordinates based on a vertex's position in space
- Compute distance from a user specified plane in space

$$s = \hat{n} \cdot \vec{v} + D$$

$$\begin{pmatrix} n_x \\ n_y \\ n_z \\ D \end{pmatrix}$$



Setting up How to Generate Coordinates

`glTexGenfv``( coord, prop, params );`

- *`coord`* is which texture coordinate we want to generate coords for
  - `GL_S`, `GL_T`, `GL_R`, `GL_Q`
- set *`prop`* to `GL_TEXTURE_GEN_MODE`
- *`param`* defines which coordinate space we're going to generate coords in
  - `GL_OBJECT_LINEAR`
  - `GL_EYE_LINEAR`

Object Linear vs. Eye Linear

- `GL_OBJECT_LINEAR` computes texture coordinates in world coordinates
  - relationship between texture plane and object remains the same
  - object looks like its been covered in wallpaper
- `GL_EYE_LINEAR` computes texture coordinates in eye coordinates
  - objects looks like its "swimming" through the texture

Setting up Texture Generation Plane

glTexGenfv(*coord, prop, params*);

- *coord* is which texture coordinate we want to generate coords for
  - GL_S, GL_T, GL_R, GL_Q
- *prop* defines which plane we're setting
  - GL_OBJECT_PLANE
  - GL_EYE_PLANE

Setting up Texture Generation Plane (cont.)

- *params* defines the plane equation
  - plane transformed by current ModelView matrix

```
GLfloat plane[] = {1,0,0,1};
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE,
   GL_OBJECT_LINEAR );
glTexGenfv( GL_S, GL_OBJECT_PLANE,
   plane );
glEnable( GL_TEXTURE_GEN_S );
```

Texture Coordinate Modes

- Recall that texture coordinates are defined only in [0,1]
- Two options if outside that range
  - clamp values to range
  - ignore integer part and only use fractional part

Repeat Mode

Clamp Mode

Setting Texture Wrap Modes

glTexParameteri(*target, prop, param*);

- *target* is GL_TEXTURE_2D
- *prop* is GL_TEXTURE_WRAP_{S,T}
- *param*
  - GL_CLAMP
  - GL_REPEAT

Telling OpenGL how to Apply a Texture

- Several options in how we should use texture's colors for our primitive
  - replace primitives color with that of texture
  - combine the texture and primitive color
- We need to set *texture environment*
  - describes how textures should be applied

Texture Environments

glTexEnvf(*target, prop, params*);

- *prop* is GL_TEXTURE_ENV_MODE
- *param* is either
  - GL_DECAL $Color_{pixel} = Color_{texture}$
  - GL_BLEND $Color_{pixel} = Color_{fragment} \cdot Color_{texture}$
    - recall that a *fragment* is an OpenGL pixel

Texture Objects

- Its convenient to bundle all this stuff together

glBindTexture(*target, texId*);

- Use twice in your program
  - when defining a texture object
  - when using that texture to render
- Use **glGenTextures()** to create unique texture identifiers

A Complete Example

```
GLuint   texId;
void init( void )
  {
  GLfloat  pixels[w][h][3] = { … };
  glGenTextures( 1, &texId );
  glBindTexture( GL_TEXTURE_2D, texId );
  glTexParameteri( GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_S, GL_REPEAT );
  glTexParameteri( GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_T, GL_REPEAT );
```

A Complete Example (cont.)

```
  glTexImage2D( GL_TEXTURE_2D, 0, 3
     w, h, 0, GL_RGB, GL_FLOAT, pixels );

  glEnable( GL_TEXTURE_2D );
}
```

A Complete Example (cont.)

```
void render( void )
  {
     …
     glBindTexture( GL_TEXTURE_2D, texId );
     glBegin( GL_TRIANGLES );
     glTexCoord2fv( t0 );
     glVertex3fv( v0 );
     glTexCoord2fv( t1 );
     glVertex3fv( v1 );
     glTexCoord2fv( t2 );
     glVertex3fv( v2 );
     glEnd();
  }
```

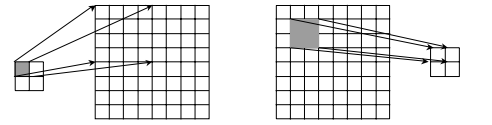
And that would almost work ...

- OpenGL's default state makes an assumption which won't quite get us there
- OpenGL defines one more texture feature, called *texture filtering*

Texture Filtering

- Ideally, texel resolution would match pixel resolution
  - but then, we'd just be drawing images
- When they don't match, we need to compensate
  - *minification* - when pixels are smaller than texels
  - *magnification* - when pixels are larger than texels

Filter Modes



Magnification *Minification*

Texture Magnification

- Not enough information to be able to fill in every pixel "correctly"
- Need to set OpenGL's magnification filter
- Two options
  - repeat texels to fill pixels
  - average closest texels to fill pixels

Texture Magnification (cont.)

`glTexParameteri``( target, prop, param );`

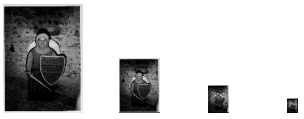
- *`prop`* is `GL_TEXTURE_MAG_FILTER`
- *`param`* is either
  - `GL_NEAREST`
  - `GL_LINEAR`

Texture Minification

- Opposite problem of magnification
  - too much information
  - more processing options
- Same sampling options as magnification
  - `GL_NEAREST`
  - `GL_LINEAR`
- Additional technique to reduce aliasing
  - mipmapping

Mipmaps

- Multiple resolution versions of the same image



Level 0 *Level 1* *Level 2* *Level 3*

Creating Mipmaps

`gluBuild2DMipMaps``(` *`target, components, width, height, format, type, pixels`* `);`

- Automatically builds mipmap levels from source image.
- Automatically scales image if not dimensions are not a power of two.

Texture Minification (cont.)

glTexParameteri(*target, prop, param*);

- *prop* is GL_TEXTURE_MIN_FILTER
- *param* is either
  - GL_NEAREST
  - GL_LINEAR
  - GL_NEAREST_MIPMAP_NEAREST
  - GL_NEAREST_MIPMAP_LINEAR
  - GL_LINEAR_MIPMAP_NEAREST
  - GL_LINEAR_MIPMAP_LINEAR