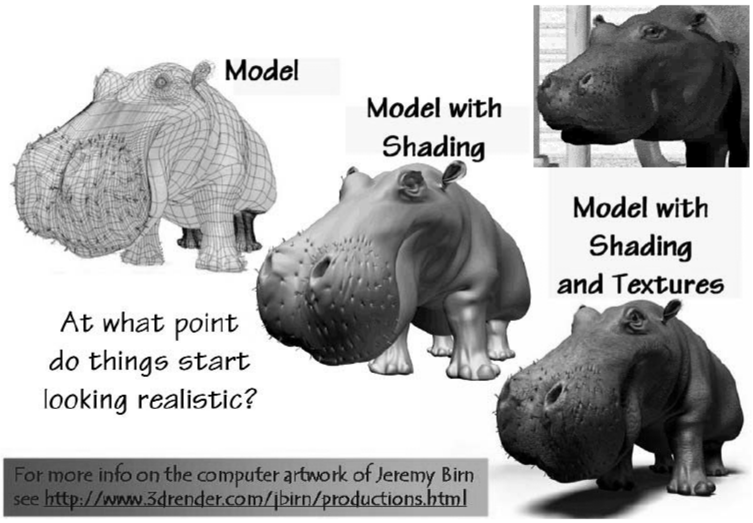
Texture Mapping

May 4, 2006

Many slides are borrowed from UNC-CH COMP236 Course (Spring 2003) taught by Leonard McMillan
http://www.unc.edu/courses/2003spring/comp/236/001/handouts.html

The Quest for Visual Realism



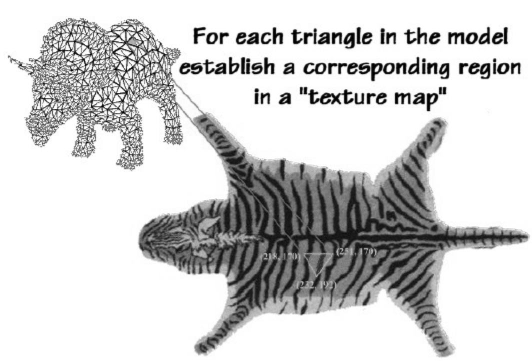
At what point do things start looking realistic?

For more info on the computer artwork of Jeremy Birn see http://www.3drender.com/jbirn/productions.html

Decal Textures

The concept is very simple!

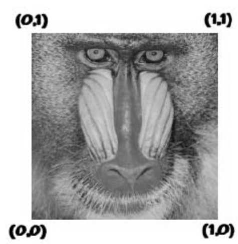
For each triangle in the model establish a corresponding region in a "texture map"



During rasterization interpolate the coordinate indices within the texture map

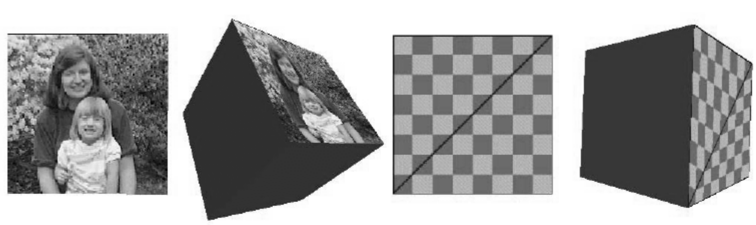
Simple OpenGL Example

- Specify a texture coordinate at each vertex *(s, t)*
- Canonical coordinates where s and t are between 0 and 1




```
public override void Draw() {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glTranslated(centerx, centery, depth);
        glMultMatrixf(Rotation);
              :
        // Draw Front of the Cube
        glEnable(GL_TEXTURE_2D);
        glBegin(GL_QUADS);
        glColor3d(0.0, 0.0, 1.0);
        glTexCoord2d(0, 1);
        glVertex3d( 1.0, 1.0, 1.0);
        glTexCoord2d(1, 1);
        glVertex3d(-1.0, 1.0, 1.0);
        glTexCoord2d(1, 0);
        glVertex3d(-1.0,-1.0, 1.0);
        glTexCoord2d(0, 0);
        glVertex3d( 1.0,-1.0, 1.0);
        glEnd();
        glDisable(GL_TEXTURE_2D);
              :
        glFlush();
}
```


Linear Interpolation of Textures



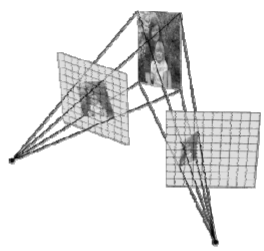
At first, you might think that we could simply apply the linear interpolation methods that we used to interpolate colors in our triangle rasterizer. However, if you implement texturing this way, you don't get the expected results.

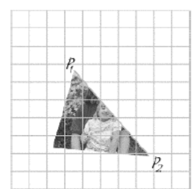
Notice how the texture seems to bend and warp along the diagonal triangle edges. Lets take a closer look at what is going on.

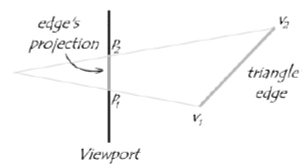
Texture Index Interpolation

Interpolating texture indices is not as simple as the linear interpolation of colors that we discussed when rasterizing triangles. Let's look at an example.

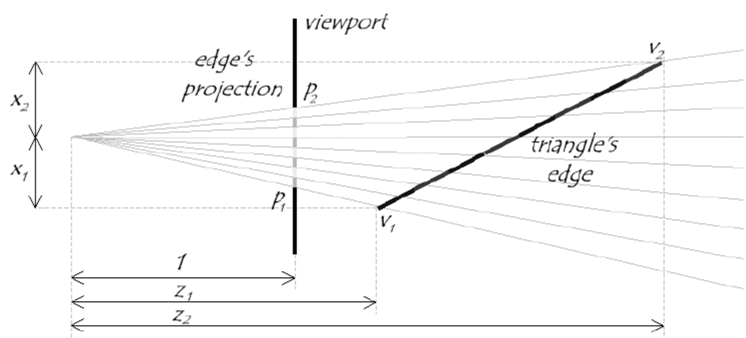
First, lets consider one edge from a given triangle. This edge and its projection onto our viewport lie in a single common plane. For the moment, lets look only at that plane, which is illustrated below:







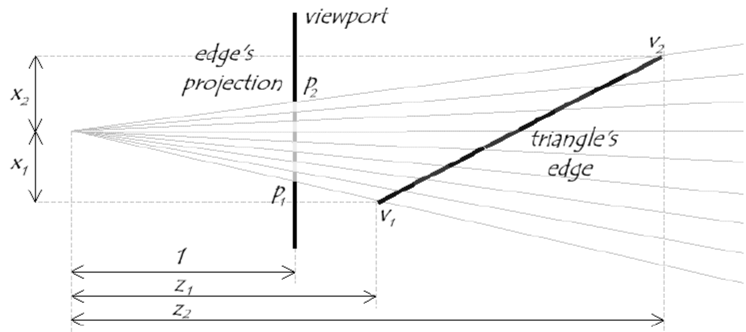
Texture Interpolation Problem



Notice that uniform steps on the image plane do not correspond to uniform steps along the edge.

Without loss of generality, let's assume that the viewport is located 1 unit away from the center of projection.

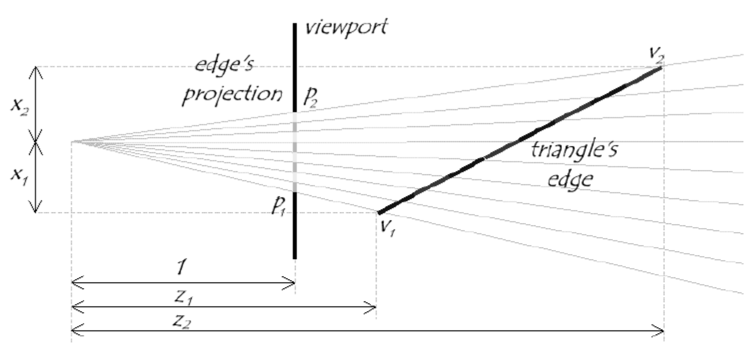
Linear Interpolation in Screen Space



Compare linear interpolation in screen space

$$p(t) = p_1 + t(p_2 - p_1) = \tfrac{x_1}{z_1} + t\left(\tfrac{x_2}{z_2} - \tfrac{x_1}{z_1}\right)$$

Linear Interpolation in 3-Space



to interpolation in 3-space:

$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} x_1 \\ z_1 \end{bmatrix} + s\left(\begin{bmatrix} x_2 \\ z_2 \end{bmatrix} - \begin{bmatrix} x_1 \\ z_1 \end{bmatrix}\right) \qquad P\left(\begin{bmatrix} x \\ z \end{bmatrix}\right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

How to make them Mesh

Still need to scan convert in screen space... so we need a mapping from *t* values to *s* values. We know that the all points on the 3-space edge project onto our screen-space line. Thus we can set up the following equality:

$$\frac{x_1}{z_1} + t\left(\frac{x_2}{z_2} - \frac{x_1}{z_1}\right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

and solve for *s* in terms of *t* giving:

$$s = \frac{t\, z_1}{z_2 + t\,(z_1 - z_2)}$$

Unfortunately, at this point in the pipeline (after projection) we no longer have z_1 and z_2 lingering around (Why?). However, we do have $w_1 = 1/z_1$ and $w_2 = 1/z_2$.

$$s = \frac{t\,\frac{1}{w_1}}{\frac{1}{w_2} + t\,(\frac{1}{w_1} - \frac{1}{w_2})} = \frac{t\, w_2}{w_1 + t\,(w_2 - w_1)}$$

Interpolating Parameters

We can now use this expression for *s* to interpolate arbitrary parameters, such as texture indices (*u, v*), over our 3-space triangle. This is accomplished by substituting our solution for *s* given *t* into the parameter interpolation.

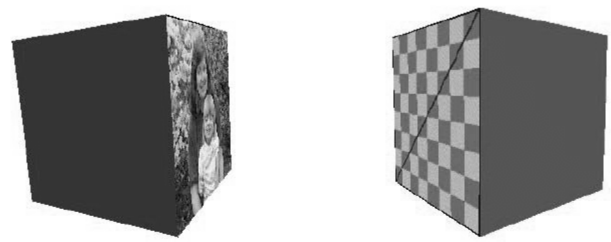
$$u = u_1 + s(u_2 - u_1)$$

$$u = u_1 + \frac{t\, w_2}{w_1 + t\,(w_2 - w_1)}(u_2 - u_1) = \frac{u_1 w_1 + t\,(u_2 w_2 - u_1 w_1)}{w_1 + t\,(w_2 - w_1)}$$

Therefore, if we premultiply all parameters that we wish to interpolate in 3-space by their corresponding *w* value and add a new plane equation to interpolate the *w* values themselves, we can interpolate the numerators and denominator in screen-space. We then need to perform a divide a each step to get to map the screen-space interpolants to their corresponding 3-space values. This is a simple modification to the triangle rasterizer that we developed in class.

Demonstration

For obvious reasons this method of interpolation is called *perspective-correct interpolation*. The fact is, the name could be shortened to simply *correct interpolation*. You should be aware that not all 3-D graphics APIs implement perspective-correct interpolation.



Compare the above with what we discussed previously…

(Note the different meaning of s and t.)

Derivation of *s* and *t*

- Two end points $P_1=(x_1, y_1, z_1)$ and $P_2=(x_2, y_2, z_2)$. Let $P_3=(1-t)P_1+(t)P_2$
- After projection, P_1, P_2, P_3 are projected to (x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3) in screen coordinates. Let $(x'_3, y'_3)=(1-s)(x'_1, y'_1) + s(x'_2, y'_2)$.

- (x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3) are obtained from P_1, P_2, P_3 by:

$$\begin{bmatrix} x'_1 w_1 \\ y'_1 w_1 \\ z'_1 w_1 \\ w_1 \end{bmatrix} = M \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}, \qquad \begin{bmatrix} x'_2 w_2 \\ y'_2 w_2 \\ z'_2 w_2 \\ w_2 \end{bmatrix} = M \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x'_3 w_3 \\ y'_3 w_3 \\ z'_3 w_3 \\ w_3 \end{bmatrix} = M \begin{bmatrix} x_3 \\ y_3 \\ z_3 \\ 1 \end{bmatrix} = M((1-t)\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} + t\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix})$$

Since

$$M\begin{bmatrix}x_1\\y_1\\z_1\\1\end{bmatrix}=\begin{bmatrix}x'_1 w_1\\y'_1 w_1\\z'_1 w_1\\w_1\end{bmatrix}, \qquad M\begin{bmatrix}x_2\\y_2\\z_2\\1\end{bmatrix}=\begin{bmatrix}x'_2 w_2\\y'_2 w_2\\z'_2 w_2\\w_2\end{bmatrix}$$

We have:

$$\begin{bmatrix}x'_3 w_3\\y'_3 w_3\\z'_3 w_3\\w_3\end{bmatrix}=(1-t)M\begin{bmatrix}x_1\\y_1\\z_1\\1\end{bmatrix}+t\cdot M\begin{bmatrix}x_2\\y_2\\z_2\\1\end{bmatrix}$$

$$=(1-t)\begin{bmatrix}x'_1 w_1\\y'_1 w_1\\z'_1 w_1\\w_1\end{bmatrix}+t\begin{bmatrix}x'_2 w_2\\y'_2 w_2\\z'_2 w_2\\w_2\end{bmatrix}$$

When P_3 is projected to the screen, we get (x'_3, y'_3) by dividing by w, so:

$$(x'_3, y'_3)=(\frac{(1-t)x'_1 w_1+t\cdot x'_2 w_2}{(1-t)w_1+t\cdot w_2},\frac{(1-t)y'_1 w_1+t\cdot y'_2 w_2}{(1-t)w_1+t\cdot w_2})$$

But remember that

$$(x'_3, y'_3)=(1-s)(x'_1, y'_1) + s(x'_2, y'_2)$$

Looking at x coordinate, we have

$$(1-s)x_1+s\cdot x_2=\frac{(1-t)x'_1 w_1+t\cdot x'_2 w_2}{(1-t)w_1+t\cdot w_2}$$

We may rewrite s in terms of t, w_1, w_2, x'_1, and x'_2.

In fact, $$s = \frac{t \cdot w_2}{(1-t)w_1 + t \cdot w_2} = \frac{t \cdot w_2}{w_1 + t(w_2 - w_1)}$$

or conversely

$$t = \frac{s \cdot w_1}{s \cdot w_1 + (1-s)w_2} = \frac{s \cdot w_1}{s(w_1 - w_2) + w_2}$$

Surprisingly, x'_1 and x'_2 disappear.

Texture Mapping II

What You Will Learn Today?

- Bump maps
- Mipmapping for antialiased textures
- Projective textures
- Shadow maps
- Environment maps

The Limits of Geometric Modeling

- Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena
  - –Clouds
  - –Grass
  - –Terrain
  - –Skin

Modeling an Orange

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
  - –Too simple
- Replace sphere with a more complex shape
  - –Does not capture surface characteristics (small dimples)
  - –Takes too many polygons to model all the dimples

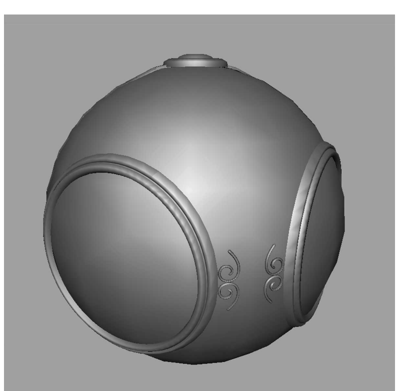
Modeling an Orange (2)

- Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  - –This process is texture mapping
- Still might not be sufficient because resulting surface will be smooth
  - –Need to change local shape
  - –Bump mapping

Three Types of Mapping

- Texture Mapping
  - –Uses images to fill inside of polygons
- Environmental (reflection mapping)
  - –Uses a picture of the environment for texture maps
  - –Allows simulation of highly specular surfaces
- Bump mapping
  - –Emulates altering normal vectors during the rendering process

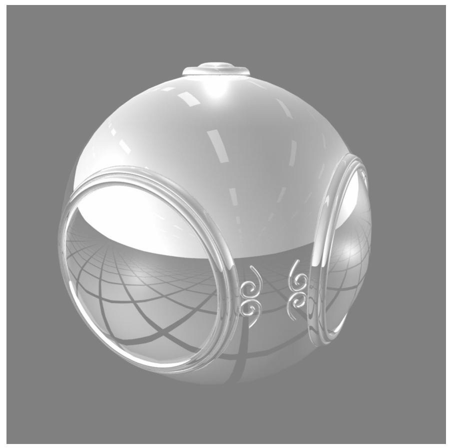
Texture Mapping



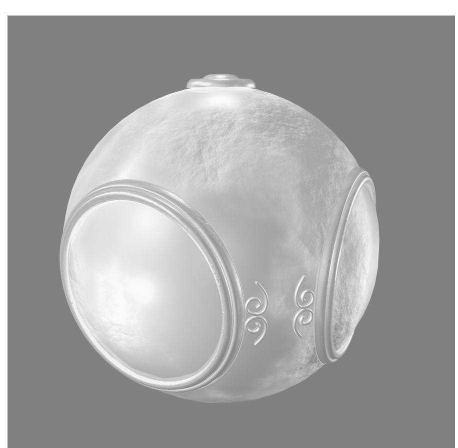
geometric model

texture mapped

Environment Mapping



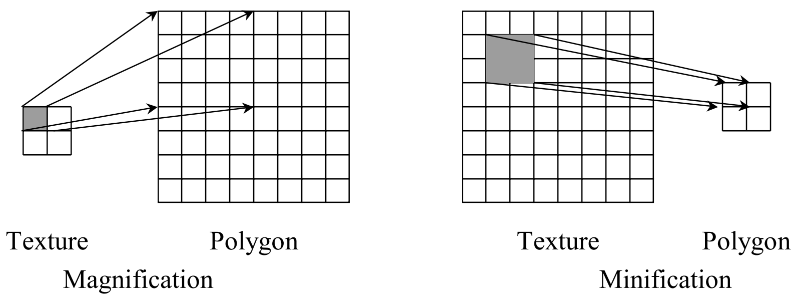
Bump Mapping



Magnification and Minification

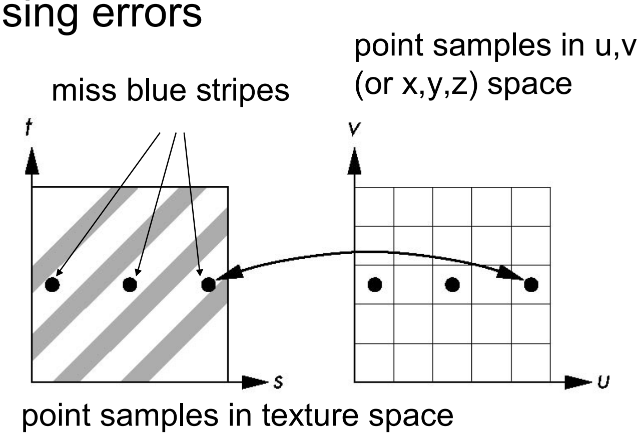
More than one texel can cover a pixel (*minification*) or more than one pixel can cover a texel (*magnification*)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values



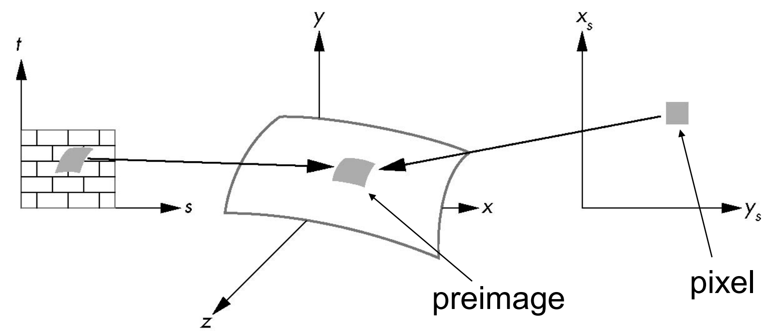
Aliasing

- Point sampling of the texture can lead to aliasing errors



Area Averaging

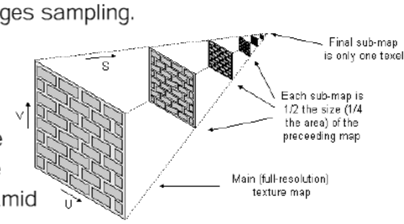
A better but slower option is to use *area averaging*

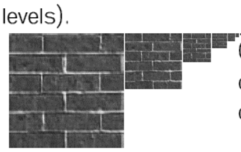


Note that *preimage* of pixel is curved

MIP Mapping

- MIP Mapping is one popular technique for precomputing and performing this prefiltering. MIP is an acronym for the Latin phrase *multium in parvo*, which means "many in a small place". The technique was first described by Lance Williams. The basic idea is to construct a *pyramid of images* that are prefiltered and resampled at sampling frequencies that are a binary fractions (1/2, 1/4, 1/8, etc) of the original images sampling.
- While rasterizing we compute the index of the decimated image that is sampled at a rate closest to the density of our desired sampling rate (rather than picking the closest one can in also interpolate between pyramid levels).

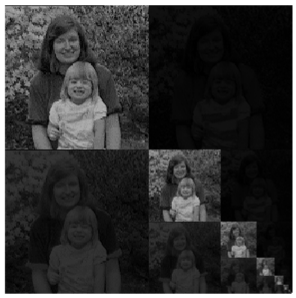


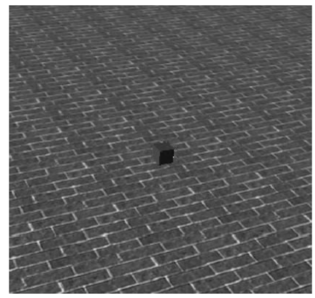


Computing this series of filtered images requires only a small fraction of additional storage over the original texture (How small of a fraction?).

Storing MIP Maps

- One convienent method of storing a MIP map is shown below (It also nicely illustrates the 1/3 overhead of maintaining the MIP map).





- The rasterizer must be modified to compute the MIP map level. Remember the equations that we derived last lecture for mapping screen-space interpolants to their 3-space equivalent.

$$u = u_1 + s(u_2 - u_1)$$

$$s = \frac{t\, w_2}{w_1 + t(w_2 - w_1)}$$

OpenGL Code Example

Incorporating MIPmapping into OpenGL applications is surprisingly easy.

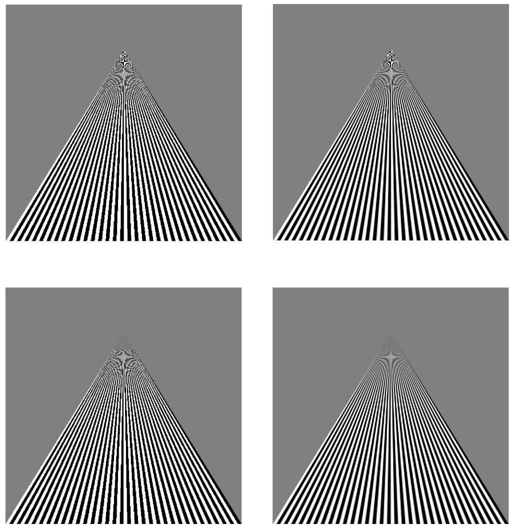
```
// Boilerplate Texture setup code

glTexImage2D(GL_TEXTURE_2D, 0, 4, texWidth, texHeight, 0, GL_RGBA,GL_UNSIGNED_BYTE, data);
gluBuild2DMipmaps(GL_TEXTURE_2D, 4, texWidth, texHeight, GL_RGBA, GL_UNSIGNED_BYTE, data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
                                          GL_LINEAR_MIPMAP_LINEAR
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

OpenGL also provides a facility for specifying the MIPmap image at each level using multiple calls to the glTexImage*D() function. This approach provides more control over filtering in the MIPmap construction and enables a wide range of tricks.

The gluBuildMipmaps() utility routine will automatically construct a mipmap from a given texture buffer. It will filter the texture using a simple box filter and then subsample it by a factor of 2 in each dimension. It repeats this process until one of the texture's dimensions is 1. Each texture ID, can have multiple levels associated with it. GL_LINEAR_MIPMAP_LINEAR trilinearly interploates between texture indices and MIPmap levels. Other options include GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, and GL_LINEAR_MIPMAP_NEAREST.

Example



point sampling

linear filtering

mipmapped point sampling

mipmapped linear filtering

Automatic Texture Coordinate Generation

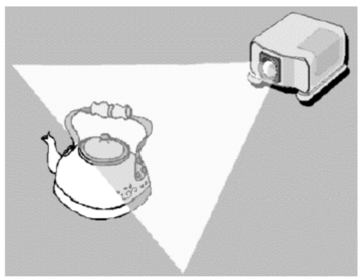
- OpenGL can generate texture coordinates automatically

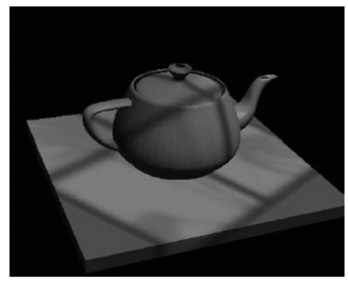
  `glTexGen{ifd}[v]()`

- generation modes
  - **`GL_OBJECT_LINEAR`**
  - **`GL_EYE_LINEAR`**
  - **`GL_SPHERE_MAP`** **(used for environmental maps)**
- Check the OpenGL Red Book!
  - 4th Ed., Chapter 8, pp.422-432, 446-450.

Projective Textures

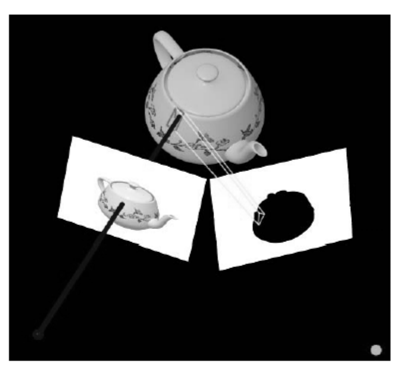
- Treat the texture as a light source (like a slide projector)
- No need to specify texture coordinates explicitly
- A good model for shading variations due to illumination (cool spotlights)
- A fair model for view-dependent reflectance (can use pictures)





The Mapping Process

During the Illumination process:
For each vertex of triangle
(in world or lighting space)
Compute ray from the projective textures origin to point
Compute homogeneous texture coordinate, $[ti, tj, t]^T$
(use equation from last slide)
During scan conversion
(in projected screen space)
Interpolate all three texture coordinates in 3-space
(premultiply by w of vertex)
Do normalization at each rendered pixel
$i = t\,i / t, \quad j = t\,j / t$
Access projected texture

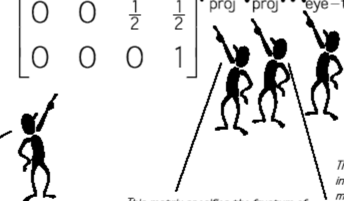


This is the same process, albeit with an additional transform, as perspective correct texture mapping. Thus, we can do it for free! Almost.

Another Frame "Texture Space"

- OpenGL is able to insert this extra projection transformation for textures by including another matrix stack, called GL_TEXTURE.
- The transform we want is:

$$T_{eye} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}_{proj} \mathbf{V}_{proj} \mathbf{M}_{eye-to-world}$$



This matrix undoes the world-to-eye transform on the MODEL_VIEW matrix stack, so the projective texture can be specified in world coordinates. Note: If you specify your lights in eye-space then this matrix is identity.

This extra matrix maps from normalized device coordinates ranging from [-1,1] to valid texture coordinates ranging from [0,1].

This matrix specifies the frustum of the projector . It is a non-affine, projection matrix. You can use any of the projection transformations to establish it, such as glFrustum(), glOrtho() or gluPerspective () .

This matrix positions the projector in the world, much like the viewing matrix positions the eye within the the world. (HINT, you can use gluLookAt() to set this up if you want.

OpenGL Example

Here is a code fragment implementing projective textures in OpenGL

```
// The following information is associated with the current active texture
// Basically, the first group of setting says that we will not be supplying texture coordinates.
// Instead, they will be automatically established based on the vertex coordinates in "EYE-SPACE"
// (after application of the MODEL_VIEW matrix).

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);
glTexGeni(GL_Q, GL_TEXTURE_GEN_MODE, (int) GL_EYE_LINEAR);

// These calls initialize the TEXTURE_MAPPING function to identity. We will be using
// the Texture matrix stack to establish this mapping indirectly.

float [] eyePlaneS = { 1.0f, 0.0f, 0.0f, 0.0f };
float [] eyePlaneT = { 0.0f, 1.0f, 0.0f, 0.0f };
float [] eyePlaneR = { 0.0f, 0.0f, 1.0f, 0.0f };
float [] eyePlaneQ = { 0.0f, 0.0f, 0.0f, 1.0f };

glTexGenfv(GL_S, GL_EYE_PLANE, eyePlaneS);
glTexGenfv(GL_T, GL_EYE_PLANE, eyePlaneT);
glTexGenfv(GL_R, GL_EYE_PLANE, eyePlaneR);
glTexGenfv(GL_Q, GL_EYE_PLANE, eyePlaneQ);
```

OpenGL Example (cont)

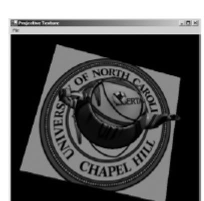
The following code fragment is inserted into Draw( ) or Display( )

```
if (projTexture) {
   glEnable(GL_TEXTURE_2D);
   glEnable(GL_TEXTURE_GEN_S);
   glEnable(GL_TEXTURE_GEN_T);
   glEnable(GL_TEXTURE_GEN_R);
   glEnable(GL_TEXTURE_GEN_Q);
   projectTexture();
}
// ... draw everything that the texture is projected onto
if (projTexture) {
   glDisable(GL_TEXTURE_2D);
   glDisable(GL_TEXTURE_GEN_S);
   glDisable(GL_TEXTURE_GEN_T);
   glDisable(GL_TEXTURE_GEN_R);
   glDisable(GL_TEXTURE_GEN_Q);
}
```


OpenGL Example (cont)

Here is where the extra "Texture" transformation on the vertices is inserted.

```
private void projectTexture() {
   glMatrixMode(GL_TEXTURE);
   glLoadIdentity();
   glTranslated(0.5, 0.5, 0.5);   // Scale and bias the [-1,1] NDC values
   glScaled(0.5, 0.5, 0.5);        // to the [0,1] range of the texture map
   gluPerspective(15, 1, 5, 7);    // projector "projection" and view matrices
   gluLookAt(lightPosition[0],lightPosition[1],lightPosition[2], 0,0,0, 0,1,0);
   glMatrixMode(GL_MODELVIEW);
}
```



Shadow Map

- Similarly, by clever use of glTexGen(), we can cast shadows on objects.

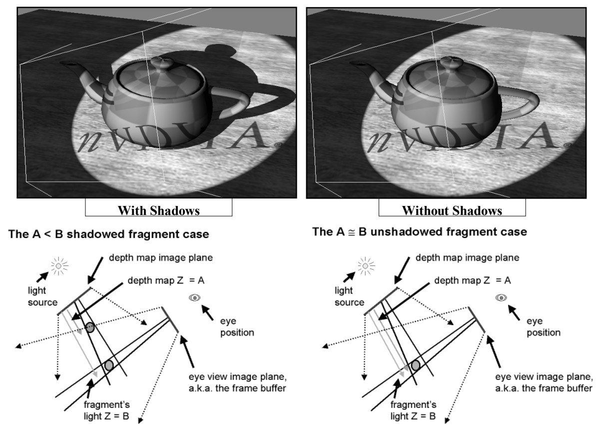


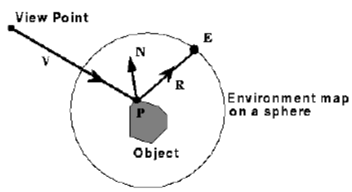
Figure 1. These diagrams were taken from Mark Kilgard's shadow mapping presentation at GDC 2001. They illustrate the shadowing comparison that occurs in shadow mapping.

More Detail

- For projective texture, see: http://developer.nvidia.com/object/Projective_Texture_Mapping.html
- For shadow map, see: http://developer.nvidia.com/object/hwshadowmap_paper.html

Environment Maps

If, instead of using a transform of the vertex to index the projected texture, we can instead use the transformed *surface normal* as an index into the texture map. This can be used to simulate reflections. This approach is not completely accurate. It assumes that all reflected rays begin from the same point, and that all objects in the scene are the same distance from that point.



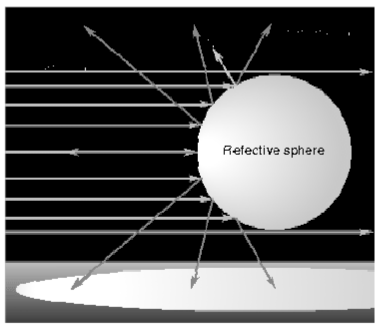
3/31/2003 Lecture 18 21

Question:

Aren't shadow and reflection global illumination effects? Why can we do it in the hardware pipeline?

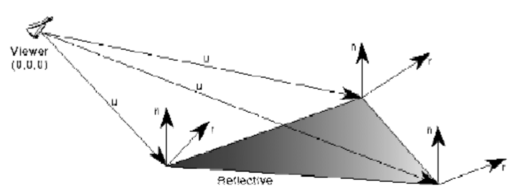
Sphere Mapping Basics

- OpenGL provides special support for a particular form of Normal mapping called sphere mapping. It maps the normals of the object to the corresponding normal of a sphere. It uses a texture map of a sphere viewed from infinity to establish the color for the normal.



Sphere Mapping

- Mapping the normal to a point on the sphere



$$\vec{R} = \hat{V} - 2(\hat{N} \cdot \hat{V})\hat{N}$$

$$p = \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

$$s = \frac{R_x}{2p} + \frac{1}{2} \quad t = \frac{R_y}{2p} + \frac{1}{2}$$

OpenGL code Example

```
// this gets inserted where the texture is created
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, (int) GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, (int) GL_SPHERE_MAP);

// Add this before rendering any primatives
if (texWidth > 0) {
 glEnable(GL_TEXTURE_2D);
 glEnable(GL_TEXTURE_GEN_S);
 glEnable(GL_TEXTURE_GEN_T);
}
```