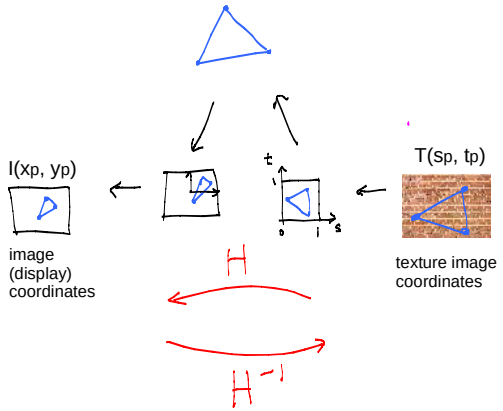
lecture 17

- more on texture mapping
  - graphics pipeline
  - MIP mapping
  - procedural textures
- procedural shading
  - Phong shading (revisited)
  - bump mapping, normal mapping

Recall texture mapping from last lecture...



Texture mapping in OpenGL (recall last lecture)

For each polygon vertex, we texture coordinates (s, t).

```
glBegin(GL_POLYGON)
  glTexCoord2f(0, 0)
  glVertex( x0, y0, z0 )
  glTexCoord2f(0, 1)
  glVertex( x1, y1, z1 )
  glTexCoord2f(1, 0)
  glVertex( x2, y2, z2 )
glEnd()
```

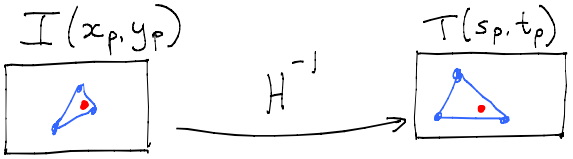
We use these coordinates to define the homography which allows us to inverse map (*correctly*) the interior points of the polygon.

for each pixel (x_p, y_p) in the image projection of the polygon {

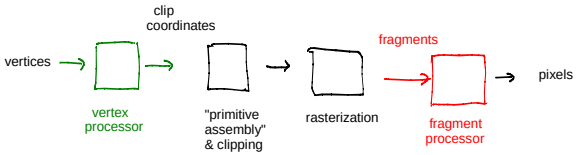
use homography to compute texel position (s_p, t_p).

use texture image $T(s_p, t_p)$ to determine $I(x_p, y_p)$

}



Where does texture mapping occur in graphics pipeline ?

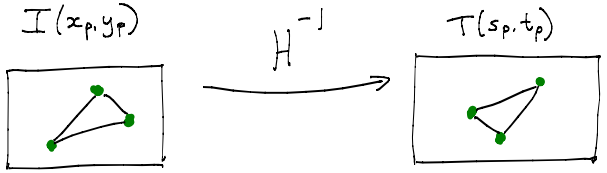


Vertex Processing

for each vertex of the polygon {

Use the assigned texture coordinates (s,t) to look up the RGB value in the texture map.

}

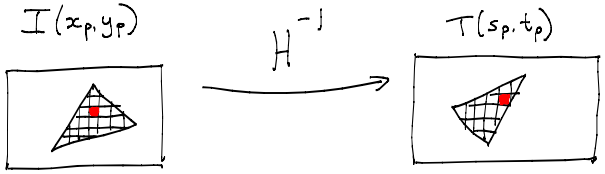


Fragment Processing

for each pixel in the image projection of the polygon {
Use homography to compute corresponding texel position

[ADDED: Texture coordinates for each fragment are computed during rasterization, not during fragment processing.]

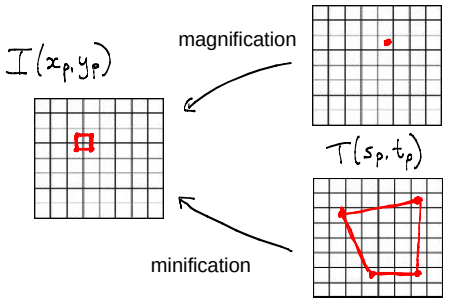
Use texture RGB to determine image pixel RGB
}



lecture 17

- more on texture mapping
  - graphics pipeline
  - MIP mapping
  - procedural textures
- procedural shading
  - Phong shading (revisited)
  - bump mapping, normal mapping

Recall from last lecture.



What does OpenGL do ?

Notation: pixels coordinates are grid points (not squares).

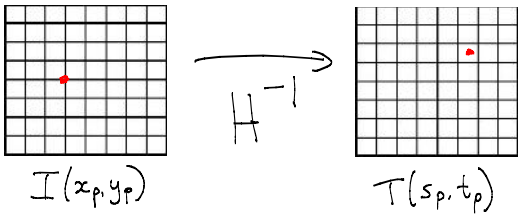
Q: How does OpenGL handle magnification ?

A: Two options (you can look up details):

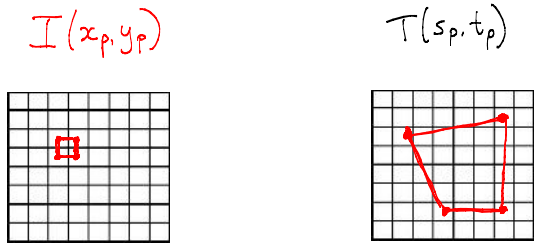
https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexParameter.xml

GL_NEAREST (take value from closest pixel)

GL_LINEAR (use bilinear interpolation for nearest 4 neighbors)



Case 2: texture minification



Possibe Solution (?): take average of intensities within the quad (inverse map of square pixel)

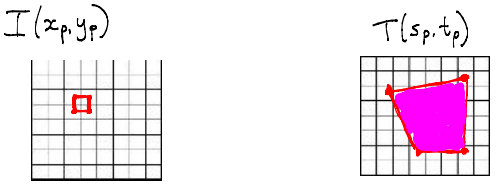
Q: How does OpenGL handle minification ?

A: It can use GL_NEAREST or GL_LINEAR as on previous slide.

What about the technique I mentioned last class and on the previous slide (Take average of intensities within the quad) ?

No, OpenGL doesn't do this.

Instead, OpenGL uses a technique called MIP mapping

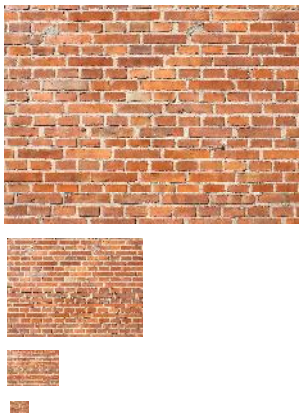


MIP mapping [Williams, 1983]

"MIP" comes from LATIN *multum in parvo*, which means "much in little"

I like to think of it as "Multiresolution Texture Mapping"

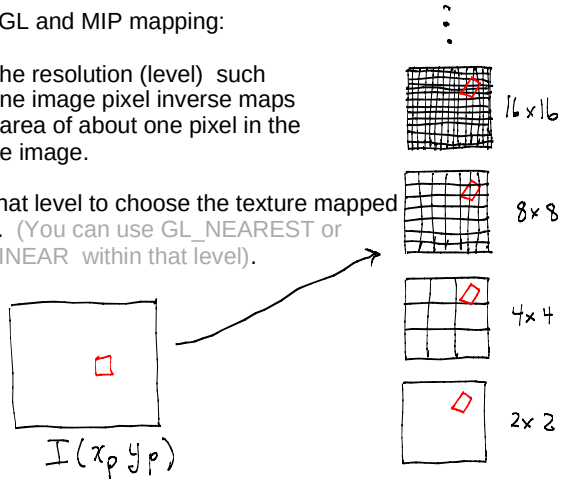
It uses multiple copies of the texture image, at different sizes i.e. resolutions (typically powers of 2).



OpenGL and MIP mapping:

Find the resolution (level) such that one image pixel inverse maps to an area of about one pixel in the texture image.

Use that level to choose the texture mapped value. (You can use GL_NEAREST or GL_LINEAR within that level).



"Procedural Textures"

Up to now our textures have been images $T(s_p, t_p)$

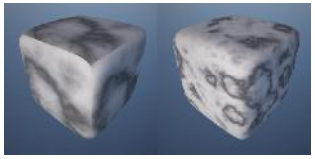
e.g. brick, rock, grass, checkerboard, ...

An alternative is to define $T(s_p, t_p)$ by a function that you write/compute.

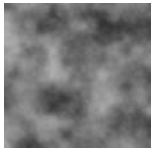
It can be anything. It can include random variables. You are limited only by your imagination.

Procedural Textures: Examples

wood



marble



fractal ("Perlin noise")

etc

Q: Texture Images versus Procedural Textures:

What are the advantages/disadvantages ?

A: - Procedural textures use less memory.

- Procedural textures allow multiple levels of detail while avoiding the approximation of MIP mapping. [ADDED: $T(s_p, t_p)$ could use float parameters rather than integer parameters.]

- Procedural textures require more computation (not just lookup and interpolate).

lecture 17

- more on texture mapping
  - graphics pipeline
  - MIP mapping
  - procedural textures
- procedural shading
  - Phong shading (revisited)
  - Toon shading (briefly)
  - bump mapping, normal mapping

What is "procedural shading" ?

- More general than procedural texture.
- RGB values are computed, rather than looked up

  i.e. Computed: Blinn-Phong model

  Looked up: glColor(), basic texture mapping

OpenGL 1.x shading: Blinn-Phong model

(recall from lecture 12)

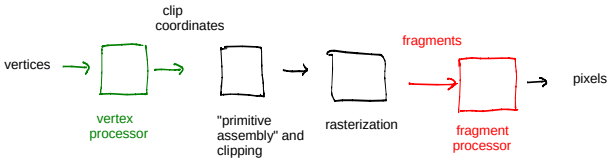
$$I_{diffuse}(x) = I_{light}\, k_{diffuse}(x)\, \max(\vec{n}\cdot\hat{l}, 0)$$

$$I_{specular}(\vec{x}) = I_{light}\, k_{specular}(x)\, (\vec{H}\cdot\vec{n})^{e}$$

$$I_{ambient}(\vec{x}) = I_{light}\, k_{ambient}(x)$$

$$I(x) = I_{diffuse}(x) + I_{specular}(x) + I_{ambient}(x)$$

Q: Where in the pipeline is shading computed?



A: It depends.

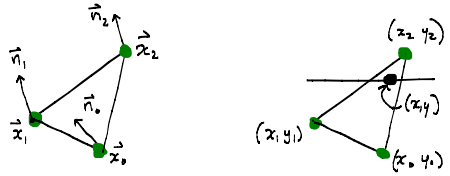
Gouraud shading or Phong shading ?
(covered at end of lecture 12)

Gouraud (smooth) shading

Vertex processor computes RGB values at each 3D vertex using the Phong model.

Rasterizer linearly interpolates these RGB values, for each pixel/fragment in the projected polygon.

Nothing for fragment processor to do.



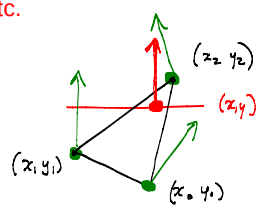
This and flat shading are the only options for OpenGL 1.x

Phong shading

Vertex processor maps vertices and normals from object to clip coordinates, but does *not* compute RGB values.

Rasterizer interpolates the normal and assigns a normal to each pixel/fragment (pixel) in the polygon's interior.

Fragment processor computes the RGB values from the normal, light vector, etc.



Possible with Open 2.x and beyond.

I will show examples of vertex and fragment shader code in a few slides. But first...

What coordinate systems are used in shading ?

- object
- world (model)
- camera/eye (model_view)

- clip (model_view_projection)
- NDC
- pixel (display coordinates)

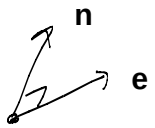
Let **n** be surface normal in world coordinates.

How do we obtain the surface normal **n** in camera coordinates ?
(see similar argument in lecture 5)

$\mathbf{M}_{GL\_MODELVIEW}\ \mathbf{n}$? No. Why not?

Let **n** and **e** be in world coordinates.

Let **e** be parallel to the surface and **n** be the surface normal.



Then, $\mathbf{n} \cdot \mathbf{e} = 0$

Let $\mathbf{M}$ be any invertible 4x4 matrix.

$$\mathbf{n}^T\mathbf{e} = \mathbf{n}^T\mathbf{M}^{-1}\mathbf{M}\,\mathbf{e}$$

$$= ((\mathbf{n}^T\mathbf{M}^{-1})^T)^T\mathbf{M}\,\mathbf{e}$$

$$= (\mathbf{M}^{-T}\mathbf{n})^T(\mathbf{M}\,\mathbf{e})$$

In particular, the surface normal in camera coordinates is:

$$\left(\mathbf{M}_{GL\_MODELVIEW}^{-1}\right)^T\mathbf{n}$$

Vertex shader for Phong shading (GLSL)

https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/lighting.php

```
void main(void)
{
  v = vec3( gl_ModelViewMatrix * gl_Vertex )

  N = normalize( gl_NormalMatrix * gl_Normal )

  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex

}
```

In modern OpenGL, you need to specify all these transformation explicitly.

Fragment shader for Phong shading (GLSL)

https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/lighting.php

```
varying vec3 N;
varying vec3 v;

void main (void)
{
   vec3 L = normalize(gl_LightSource[0].position.xyz - v);
   vec3 E = normalize(-v); // we are in Eye Coordinates, so EyePos is (0,0,0)
   vec3 R = normalize(-reflect(L,N));

   //calculate Ambient Term:
   vec4 Iamb = gl_FrontLightProduct[0].ambient;

   //calculate Diffuse Term:
   vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
   Idiff = clamp(Idiff, 0.0, 1.0);

   // calculate Specular Term:
   vec4 Ispec = gl_FrontLightProduct[0].specular
                * pow(max(dot(R,E),0.0),0.3*gl_FrontMaterial.shininess);
   Ispec = clamp(Ispec, 0.0, 1.0);

   // write Total Color:
   gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
}
```

Toon Shading (as in cartoon)

Also known as 'cel' or 'Gooch' (1998) shading.

Uses a "non-photorealistic" model based on light, normals,...

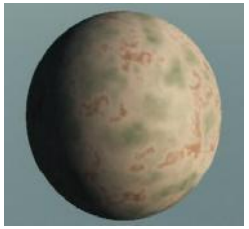
Note outline effects !

The examples from lecture 13 also used procedural shading.

lecture 17

- more on texture mapping
  - graphics pipeline
  - texture mapping for curved surfaces
  - MIP mapping
  - procedural textures
- procedural shading
  - Phong shading (revisited)
  - Toon shading
  - bump mapping, normal mapping

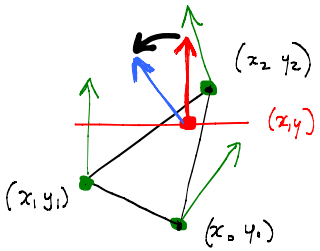
Bump mapping (Blinn 1974)

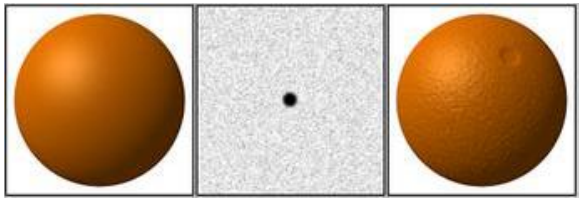


without bumps

with bumps

Interpolate surface normal as in Phong shading... but add a **perturbation to the normal** which is consistent with "bumps". These bumps are specified by a "bump map".

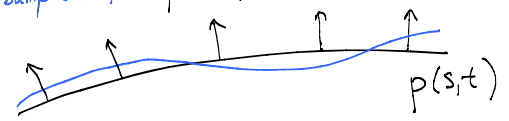




without bumps bump map with bumps

Add bumps:

$p_{bump}(s,t) = p(s,t) + b(s,t)\,\vec{n}(s,t)$



What are the normals of $p_{bump}(s,t)$?

What are the normals of $p(s,t)$?

$$\vec{n}(s,t) = \frac{\frac{\partial p}{\partial s} \times \frac{\partial p}{\partial t}}{\left| \frac{\partial p}{\partial s} \times \frac{\partial p}{\partial t} \right|}$$

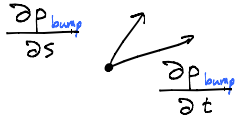
What are the normals of $p_{bump}(s,t)$?

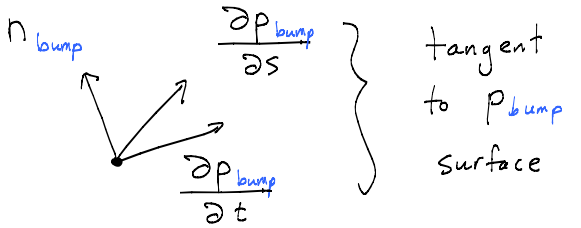
$$\vec{n}_{bump} = \frac{\frac{\partial p_{bump}}{\partial s} \times \frac{\partial p_{bump}}{\partial t}}{\left| \frac{\partial p_{bump}}{\partial s} \times \frac{\partial p_{bump}}{\partial t} \right|}$$

Applying chain rule from Calculus:

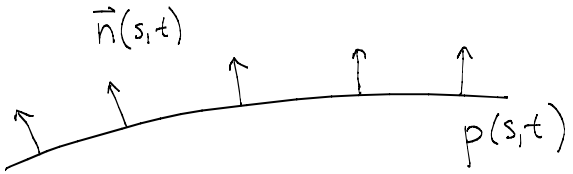
$$\frac{\partial p_{bump}}{\partial s} = \frac{\partial p}{\partial s} + \frac{\partial b}{\partial s} \cdot n + b \frac{\partial n}{\partial s}$$

$$\frac{\partial p_{bump}}{\partial t} = \frac{\partial p}{\partial t} + \frac{\partial b}{\partial t} \cdot n + b \frac{\partial n}{\partial t}$$





$$\vec{n}_{bump} = \frac{\frac{\partial p_{bump}}{\partial s} \times \frac{\partial p_{bump}}{\partial t}}{\left| \frac{\partial p_{bump}}{\partial s} \times \frac{\partial p_{bump}}{\partial t} \right|}$$



To compute $\vec{n}_{bump}$, assume that $\frac{\partial n}{\partial s}$ and $\frac{\partial n}{\partial t}$ are ≈ 0.

Then, "turn the crank"

(see lecture notes for the math)

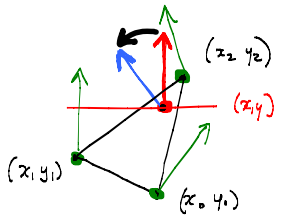
Bottom line: if we have

$$\frac{\partial p}{\partial s}, \frac{\partial p}{\partial t}, \frac{\partial b}{\partial s}, \frac{\partial b}{\partial t}$$

for any (s,t) then we get a formula for $\vec{n}_{bump}(s,t)$.

Thus, given an underlying smooth surface p(s,t) and given a bump map b(s, t), you can *estimate* the normal n(s, t) of the surface that you would get if you added these bumps to the surface.

Similar to Phong shading, but now use the perturbed normal instead of the original normal when computing the RGB values.



Normal Mapping

Given a bump map b(s,t), **pre-compute** the surface normal perturbations and store them in a texture.

b(s,t) normal_perb(s, t) I(x, y)

viewed obliquely for illustration purposes

Resulting image of quad, rendered with fake normal perturbations.

Issues to be aware of:

- For both bump mapping and normal mapping, the surface geometry remains smooth

  e.g. the outline of a bump mapped sphere is a circle, (not a bumpy circle). The lack of bumps on the outline is sometimes detectable.

  More complicated methods ("displacement mapping") have been developed to handle this.

- Writing fragment shaders for these methods is non-trivial. To compute the perturbed normal at a fragment, it is best to use the local coordinate system of surface (tangent plane + smooth normal).

Summary of high level concepts:

- Vertex vs. fragment processing
- Smooth shading vs. Phong shading
- Texture mapping (lookup) vs. (procedural) shading

Keep these in mind when working through the low level details.