In the first half of today's lecture, I elaborated on the ideas of texture mapping that were introduced last lecture. In the second half, I discussed the idea of procedural shading, and looked at a specific example known as bump mapping.

Texture mapping (continued from last lecture)

OpenGL pipeline

At the end of last lecture, I mentioned how texture coordinates (s, t) are assigned to vertices in OpenGL. We discussed the mapping between pixel coordinates (x, y) and texture coordinates (s, t) and we saw how this mapping was given by a composition of matrices, ultimately resulting in 3×3 invertible matrix known as a homography. The homography maps from texture coordinates to pixel coordinates. The inverse maps points on the interior of (the image projection of a) polygon to their corresponding texture coordinates.

I began today's lecture by recalling the OpenGL pipeline and distinguishing which aspects of texture mapping involve vertex processing and which involve fragment processing. The vertex processing stage involves little extra work for texture mapping (beyond the usual mapping from object coordinates to clipping coordinates). The work for texture mapping begins at the rasterization stage, where the fragments are generated. That would be the time that the texture coordinates for each fragment are generated, using the inverse homography. (Note that I am not distinguishing whether the inverse homography maps to (s, t) or to (s_p, t_p) space here.) Finally, the lookup of the texture color occurs at the fragment processing stage.

OpenGL: Magnification, minification, and MIP mapping

In the slides from last lecture, I briefly mentioned the magnification and minification issue. Let's have a brief look at how this is done in OpenGL. See
`https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexParameter.xml` for details.

In OpenGL texture mapping, you can specify one of several methods for sampling the texture. If you have magnification, then there are two options. You can choose the integer coordinates (s_p, t_p) that are closest to the (non-integer) inverse mapped values. Alternatively, you can compute the RGB value by bilinearly interpolating the values of the four nearest neighbors. To choose between these two methods, you set a texture parameter to be either `GL_NEAREST` or `GL_LINEAR`.

The above method/parameters are available for minification as well. But there is another method as well which is more subtle. Recall that with minification, a square pixel $I(x_p, y_p)$ is inverse mapped to a region in (s_p, t_p) space that is larger than a pixel – sometimes much larger, as in the case at the horizon. The method I suggested last class where you average over that region in texture space can be expensive in this case, since for each pixel in the image you need to average over many pixels in the texture and this can be slow.

One method for speeding things up is called MIP mapping[1] which can be thought of as multiresolution texture mapping. This method pre-computes multiple versions of a texture of different sizes. For simplicity, suppose the original texture is $2^n \times 2^n$. Then a MIP mapped texture would also have sizes $2^{n-1} \times 2^{n-1}, 2^{n-2} \times 2^{n-2}, etc. 2 \times 2$. Texture $2^{n-l} \times 2^{n-l}$ is said to be "level" l. These

[1] It was proposed in 1983 by Lance Williams.

images are just lower resolution version of the original. Because each extra level takes 1/4 as much space as the previous level, this extra texture takes up relatively little texture memory overall.

There are many details involved in using MIPmapping in OpenGL so let me just sketch out the basic ideas. First, you need to specify that you want to use MIP mappings and you need to tell OpenGL to (pre)compute the MIP maps. (I will not go into *how* the different resolution textures are computed, since this requires some background in signal processing which most of you don't have.) Once MIP maps are computed, they can be used roughly as follows. During rasterization, when a fragment is generated, the image pixel (x_p, y_p) is inverse mapped to a texture coordinate (s, t) as before. But now at the fragment processing stage, it must be decided roughly how many pixels m are occupied by the inverse mapped 1×1 little square pixel centered at (x_p, y_p). (There is an OpenGL command for this query.) Similarly it must be decided which level of the MIP map is appropriate, that is, we want the $i - th$ level MIP map which is of size $m \approx 2^{n-i} * 2^{n-i}$. Typically the m satisfying will fall between two levels i and $i + 1$, and in that case the RGB texture value must be linearly interpolated from the values of these two levels (albeit with more processing cost). For the values at each of the two MIP map levels, i and $i + 1$, one also needs to specify whether to use `GL_NEAREST` or `GL_LINEAR`. You can read up a bit more about these parameter settings here: `http://www.glprogramming.com/red/chapter09.html`

Procedural textures

All the texture mapping discussion up to now has assumed that we are working with an existing texture image $T(s_p, t_p)$. However, many aspects of the texture mapping mechanism that we've described do not depend on this assumption. Many aspects of texture mapping would still be applicable if $T(s_p, t_p)$ were not a pre-existing image, but rather it were a function that could be computed at runtime.

There are a few advantages to defining a texture by a function (or method, or procedure). One advantage is that we could save on memory, since we wouldn't need to store the texture. A second advantage is that the parameters of the function might not need to be integers (s_p, t_p) but rather could be floats (s, t) and so the issues of aliasing and interpolation might vanish.

There are disadvantages too, though. Having to compute the texture values at runtime is computationally more expensive than just looking up the values in memory.

See the slides for some examples of texture mapped images that used procedural textures.

Procedural shading

Until now, we have thought of texture mapping as just copying (painting) an RGB value onto a surface point Let's next generalize our notion of texture mapping. In lecture 12, we introduced several shading models for determining the intensity of light reflected from a point. We saw that OpenGL uses the Blinn-Phong model. There the intensity was determined by parameters such as diffuse k_d and specular reflectance coefficients k_s, e which are defined at each vertex. The model also depends on the relative positions of the vertex position $\mathbf{x}$, the light source $\mathbf{x}_l$, and the camera.

One can combine texture mapping with a shading model. For example, material properties such as k_d, k_s, e could be stored as texture images $T_d(s_p, t_p)$, $T_s(s_p, t_p)$, $T_e(s_p, t_p)$. When choosing the RGB values for a vertex, one could compute the texture position (s_p, t_p) corresponding to each pixel position (x_p, y_p) as before, but now one could look up the three values from the three texture images,

and plug these values into the Phong lighting model. Moreover, one should not be restricted to the Blinn-Phong model when choosing RGB values. One can define whatever model one wishes. All one needs is access to the parameter values of the model, and some code for computing the RGB values given those parameters. This more general idea of computing (rather than just looking up) the RGB values is called *procedural shading*.

Gouraud versus Phong shading, and the OpenGL pipeline

Before we turn more examples of procedural shading, let's recall an important difference between smooth (Gouraud) shading and Phong shading. Suppose we have a triangle. With Gouraud shading, we choose an RGB value for each vertex of the polygon. We then use linear interpolation to fill the polygon with RGB values. Choosing the RGB values of the vertices is done with the vertex shader. These RGB values are then the color of the vertex. The fill occurs at the rasterization stage, when the polygon is "broken down" into fragments. In OpenGL 1.x, the rasterizer linearly interpolates the colors for each fragment of the polygon. This happens automatically when you set the shading mode to `GL_SMOOTH`. There is nothing left for the fragment processor to do.

With Phong shading (not be confused with the Phong lighting model), the RGB filling is done differently. Instead of interpolating the RGB values from the vertices, one interpolates the surface normals. This interpolation is done by the rasterizer, which assigns an interpolated surface normal to each fragment that it generates. The rasterizer does not assign RGB values, however. Rather, the RGB values are computed by the next stage in the pipeline – the fragment shader. (OpenGL 1.x does *not* do Phong shading.)

To do Phong shading or any of the more general examples of procedural shading such as we will see later in the lecture, one has to use OpenGL 2.x or beyond and one has to write the fragment shader. This is not particular difficult to do. One just specifies code for computing RGB values from all the parameters. For an example of what the code looks like, see:

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

Coordinate systems for shading calculations

In which coordinate system are these shading computations performed? Consider a vertex and its associated normal. The vertex processor maps from object coordinates to clip coordinates. If it were also to map the surface normals (as direction vectors) to clip coordinates, these transformed normals could not be useful for the Blinn-Phong lighting model. As I mentioned at the end of lecture 5 (just prior to the Appendix), the mapped surface normal will not be normal to the mapped polygon. Moreover, the angles between the normal and light source, and other angles needed by the Blinn-Phong model would be wrong. We simply cannot perform Blinn-Phong calculations in clip coordinates.

To compute RGB values using the Blinn-Phong model, one typically uses eye/camera coordinates. The vertices are mapped to eye coordinates using only the `GL_MODELVIEW` matrix. This allows one to use real 3D distances in the model, for example, the distance from the vertex to the light source in the case of a spotlight. (Note that the vertices themselves still need to be mapped using both the `GL_MODELVIEW` and `GL_PROJECTION` matrices, so that clipping and rasterization can be done. But shading computations – whether in the vertex shader or fragment shader – need a separate representation of the vertex positions and normals which is in camera coordinates.)

If a surface normal vector $\mathbf{n}$ were transformed by the `GL_MODELVIEW` matrix, there is no guarentee that its angle with respect to the underlying surface will remain the same. (See Exercises.) If we can't use the `GL_MODELVIEW` matrix to transform the normal to eye coordinates in general, then what can we use? We discussed a similar issue at the end of lecture 5 so let's revisit it.

Let $\mathbf{e}$ be a 4×1 direction vector whose direction lies in the plane of the polygon e.g. it could be the vector from one vertex of the polygon to another vertex of the polygon, $\mathbf{e} = \mathbf{x}_2 - \mathbf{x}_1$. Let $\mathbf{n}$ be a 4×1 vector in the direction of the surface normal vector. By assumption, we have $\mathbf{n}^T\mathbf{e} = 0$.

We are representing $\mathbf{n}$ and $\mathbf{e}$ as direction vectors, namely 4-vectors whose 4th component is a 0. Let $\mathbf{M}$ be any invertible 4×4 matrix, for example, the `GL_MODELVIEW` matrix. Using linear algebra matrix tricks (MATH 223),

$$0 = \mathbf{n}^T\mathbf{e} = \mathbf{n}^T\mathbf{M}^{-1}\mathbf{M}\ \mathbf{e} = ((\mathbf{n}^T\mathbf{M}^{-1})^T)^T\mathbf{M}\ \mathbf{e} = (\mathbf{M}^{-T}\mathbf{n})^T(\mathbf{M}\ \mathbf{e})$$

where $\mathbf{M}^{-T}$ means $(\mathbf{M}^{-1})^T$ or equivalently $(\mathbf{M}^T)^{-1}$ (by linear algebra). Thus, to transform $\mathbf{n}$ so that it is perpendicular to $\mathbf{Me}$, we need to multiply $\mathbf{n}$ by $\mathbf{M}^{-T}$.

In OpenGL 2.x and beyond, the matrices $\mathbf{M}_{\texttt{GL\_MODELVIEW}}$ and $\mathbf{M}^{-T}_{\texttt{GL\_MODELVIEW}}$ are predefined. Their names are `gl_ModelViewMatrix` and `gl_NormalMatrix`, respectively. See the above URL (or the slides) for an example of how this is used in a vertex shader.

Other examples of Procedural Shading

Toon/Cel shading, Gooch shading (1998)

Several methods have been proposed for making images that have simplified shading, as in cartoons. See the slides for examples. Also see `http://en.wikipedia.org/wiki/Cel_shading` One of the nice ideas here is to automatically draw outlines around objects. This can be done by setting pixels to black when the surface normal is perpendicular to the viewing direction.

Let's consider another example, but this time go into some of the technical details.

Bump mapping (Blinn 1974)

Phong shading smoothly interpolates the surface normal across the face of a polygon (or patch of bicubic). Bump mapping is a way of choosing the surface normals to give the surface a bumpy or wrinkled appearance. The idea is to add small bumps to a smooth surface, *without re-meshing the surface*, that is, without defining a fine mesh to capture the little bumps. For example, a surface such as an orange might be modelled as a sphere with many little bumps on it. We don't change the geometry of the sphere. Instead, when we compute the intensitites on the sphere we pretend as if the surface normal is varying in a more complicated way than it would on a sphere.

How can we define these variations in the normal? Suppose we have a smooth surface $\mathbf{p}(s,t)$ such as a triangle or bicubic. The surface normal is in direction $\mathbf{n}(s,t)$ which is parallel to $\frac{\partial \mathbf{p}(s,t)}{\partial s} \times \frac{\partial \mathbf{p}(s,t)}{\partial t}$. We define a bumpy version of this surface, by defining a *bump map* $b(s,t)$. We wish to imitate the surface normal variations that arise if one were to displace the surface by a distance $b(s,t)$ in the direction of the surface normal, that is, if we had a bumpy surface

$$\mathbf{p}_{bump}(s,t) = \mathbf{p}(s,t) + b(s,t)\ \mathbf{n}(s,t)$$

where $\mathbf{n}$(s,t) is the unit normal to the surface at $\mathbf{p}(s,t)$. This is the unit normal to the original surface, not the surface with bumps added on it.

Consider the partial derivative

$$\frac{\partial \mathbf{p}_{bump}(s,t)}{\partial s} = \frac{\partial \mathbf{p}(s,t)}{\partial s} + \frac{\partial b(s,t)}{\partial s}\mathbf{n}(s,t) + b(s,t)\frac{\partial \mathbf{n}(s,t)}{\partial s}$$

which is a vector tangent to the surface. In Blinn's paper (1978), we simplifies the model by assuming that the original surface $\mathbf{p}(s,t)$ is so smooth that

$$\frac{\partial}{\partial s}\mathbf{n} \approx 0.$$

For example, if the original surface were a plane then this partial derivative would be exactly 0! In addition, we assume that the bumps are small and so $b(s,t)$ is small. Putting these two assumptions together, we approximate the partial derivative by dropping the last term, giving:

$$\frac{\partial \mathbf{p}_{bump}(s,t)}{\partial s} \approx \frac{\partial \mathbf{p}(s,t)}{\partial s} + \frac{\partial b(s,t)}{\partial s}\mathbf{n}(s,t)$$

Similarly,

$$\frac{\partial \mathbf{p}_{bump}(s,t)}{\partial t} \approx \frac{\partial \mathbf{p}(s,t)}{\partial t} + \frac{\partial b(s,t)}{\partial t}\mathbf{n}(s,t)$$

Taking their cross product gives an approximation of the direction of the surface normal of the *bumpy surface*.

$$\begin{aligned}\frac{\partial \mathbf{p}_{bump}}{\partial s} \times \frac{\partial \mathbf{p}_{bump}}{\partial t} \quad &\approx \quad \frac{\partial \mathbf{p}}{\partial s} \times \frac{\partial \mathbf{p}}{\partial t} \;+\; (\frac{\partial \mathbf{p}}{\partial s} \times \frac{\partial b}{\partial t}\mathbf{n}) \;-\; (\frac{\partial \mathbf{p}}{\partial t} \times \frac{\partial b}{\partial s}\mathbf{n}) \\ &= \quad \frac{\partial \mathbf{p}}{\partial s} \times \frac{\partial \mathbf{p}}{\partial t} \;+\; \frac{\partial b}{\partial t}(\frac{\partial \mathbf{p}}{\partial s} \times \mathbf{n}) \;-\; \frac{\partial b}{\partial s}(\frac{\partial \mathbf{p}}{\partial t} \times \mathbf{n})\end{aligned}$$

There is also a term that contains $\mathbf{n} \times \mathbf{n}$ but this term disappears because $\mathbf{n} \times \mathbf{n} = 0$.

The above vector is the normal that is used for shading the surface at $\mathbf{p}(s,t)$, instead of the original normal to the surface $\mathbf{n}(s,t)$ at that point. Of course, the above vector needs to be divided by its length, so we have a unit vector.

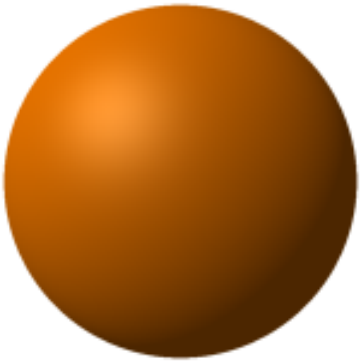
[ASIDE: Notice that there are three terms on the right side. The first term is parallel to the normal on the original smooth surface. The second two terms are parallel to the smooth surface's tangent plane. It is the second two components that define the perturbation of the normal on the surface, and these terms depend on the bump map $b(s,t)$.]

Given the above approximation, one can define the surface normal for the bumpy surface by

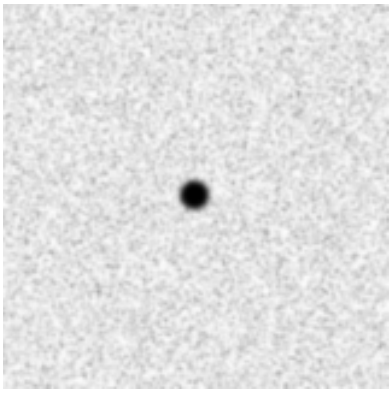
$$\mathbf{n}_{bump}(s,t) = (\frac{\partial \mathbf{p}_{bump}}{\partial s} \times \frac{\partial \mathbf{p}_{bump}}{\partial t}) \;\; / \;\; | \; \frac{\partial \mathbf{p}_{bump}}{\partial s} \times \frac{\partial \mathbf{p}_{bump}}{\partial t} \; | \; .$$

Assuming it is possible to find the texture coordinates (s,t) that corresponds to a given image pixel (x_p, y_p), one can compute a surface normal $\mathbf{n}_{bump}(s,t)$ using the above approximation. I will spare you the detail on this, since you really only need to go down that rabbit hole if you want to implement this.

An example (taken from the web) is shown below. I encourage you to search under "bump mapping" for other examples on the web.



smooth sphere



bump map

bumpy sphere

Normal mapping

The essential idea of bump mapping is that computes a fake normal at each point on the surface and uses that normal to compute the RGB image intensities. ()This work would be done by the fragment shader, not vertex shader).

Normal mapping is a similar idea to bump mapping. WIth normal mapping, though, we precompute the perturbed normals and store them as an RGB texture. That is, RGB = (n_x, n_y, n_z). See Exercises.

In the slides, I gave an example of normal mapping. You can find that example along with more details here: `http://en.wikipedia.org/wiki/Normal_mapping`