CMSC 425: Lecture 12
Texture Mapping
Thursday, Mar 14, 2013

Surface Detail: We have discussed the use of lighting as a method of producing more realistic images. This is fine for smooth surfaces of uniform color (plaster walls, plastic cups, metallic objects), but many of the objects that we want to render have some complex surface finish that we would like to model. In theory, it is possible to try to model objects with complex surface finishes through extremely detailed models (e.g. modeling the cover of a book on a character by character basis) or to define some sort of regular mathematical texture function (e.g. a checkerboard or modeling bricks in a wall). But this may be infeasible for very complex unpredictable textures.

Textures and Texture Space: Although originally designed for textured surfaces, the process of *texture mapping* can be used to map (or "wrap") any digitized image onto a surface. For example, suppose that we want to render a picture of the Mona Lisa, or wrap an image of the earth around a sphere, or draw a grassy texture on a soccer field. We could download a digitized photograph of the texture, and then map this image onto surface as part of the rendering process.

There are a number of common image formats which we might use. We will not discuss these formats. Instead, we will think of an image simply as a 2-dimensional array of RGB values. Let us assume for simplicity that the image is square, of dimensions $n \times n$ (OpenGL requires that n is a power of 2 for its internal representation. If you image is not of this size, you can pad it out with unused additional rows and columns.) Images are typically indexed row by row with the upper left corner as the origin. The individual RGB pixel values of the texture image are often called *texels*, short for *texture elements*.

Rather than thinking of the image as being stored in an array, it will be a little more elegant to think of the image as function that maps a point (s, t) in 2-dimensional *texture space* to an RGB value (see Fig. 1). That is, given any pair (s, t), $0 \leq s, t < 1$, the texture image defines the value of $T(s, t)$ is an RGB value. Note that the interval $[0, 1)$ does not depend on the size of the images. This has the advantage an image of a different size can be substituted without the need of modifying the wrapping process.

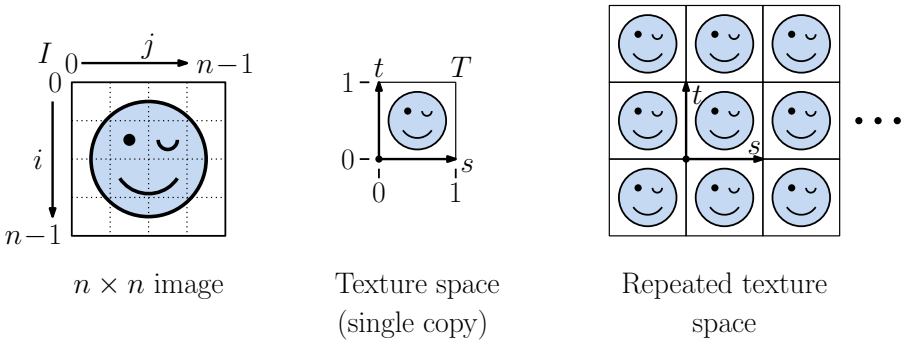

Fig. 1: Texture space.

For example, if we assume that our image array $I[n][n]$ is indexed by row and column from 0 to $n-1$ with (as is common with images) the origin in the upper left corner. Our texture space $T(s, t)$ has two

axes, labeled s (horizontal) and t (vertical), such that the origin is in the lower-left corner. We could then apply the following function to round a point in image space to the corresponding array element:

$$T(s,t) = I\big[\,\lfloor(1-t)n\rfloor\,\big]\big[\,\lfloor sn\rfloor\,\big], \quad \text{for } 0 \le s,t < 1.$$

In many cases, it is convenient to think of the texture is an infinite function. We do this by imagining that the texture image is repeated cyclically throughout the plane. This is useful when applying a small repeating texture to a very large surface (like a brick pattern on a wall). This is sometimes called a *repeated texture*. In this case we can modify the above function to be

$$T(s,t) = I\big[\,\lfloor(1-t)n\rfloor \bmod n\big]\big[\,\lfloor sn\rfloor \bmod n\big], \quad \text{for any } s,t.$$

Inverse Wrapping Function and Parametrization: Suppose that we wish to "wrap" a 2-dimensional texture image onto the surface of a 3-dimensional ball of unit radius, that is, a unit sphere. We need to define a wrapping function that achieves this. The surface resides in 3-dimensional space, so the wrapping function would need to map a point (s,t) in texture space to the corresponding point (x,y,z) in 3-space. That is, the wrapping function can be thought of as a function $W(s,t)$ that maps a point in 2-dimensional texture space to a point (x,y,z) in three dimensional space.

Later we will see that it is not the wrapping function that we need to compute, but rather its inverse. So, let us instead consider the problem of computing a function W^{-1} that maps a point (x,y,z) on the sphere to a point (s,t) in parameter space. This is called the *inverse wrapping function*.

This is typically done by first computing a 2-dimensional *parametrization* of the surface. This means that we associate each point on the object surface with two coordinates (u,v) in *surface space*. Implicitly, we can think of this as three functions, $x(u,v)$, $y(u,v)$ and $z(u,v)$, which map the parameter pair (u,v) to the x,y,z-coordinates of the corresponding surface point.

Our approach to solving the inverse wrapping problem will be to map a point (x,y,z) to the corresponding parameter pair (u,v), and then map this parameter pair to the desired point (s,t) in texture space. At the end of this lecture, we present an example of how to derive a parametrization for a sphere.

Texture Mapping in OpenGL: Recall that all objects in OpenGL are rendered as polygons or generally meshes of polygons. This simplifies the texture mapping process because it means that we need only provide the inverse wrapping function for the vertices, and we can rely on simple interpolation to fill in the polygon's interior. For example, suppose that a triangle is being drawn. When the vertices of the polygon are given, the user also specifies the corresponding (s,t) coordinates of these points in texture space. These are called the vertices' *texture coordinates*. This implicitly defines the inverse wrapping function from the surface of the polygon to a point in texture space.

As with surface normals (which were used for lighting computations) texture vertices are specified *before* each vertex is drawn. For example, a texture-mapped object in 3-space with shading might be drawn using the following general form, where $\vec{n} = (n_x, n_y, n_z)$ is the surface normal, (s,t) are the texture coordinates, and $p = (p_x, p_y, p_z)$ is the vertex position.

```
glBegin(GL_POLYGON);
  glNormal3f(nx, ny, nz); glTexCoord2f(s, t); glVertex3f(px, py, pz);
  // ...
glEnd();
```

Interpolating Texture Coordinates: Given the texture coordinates, the next question is how to interpolate the texture coordinates for the points in the interior of the polygon. An obvious approach is to first project the vertices of the triangle onto the viewport. This gives us three points p_0, p_1, and p_2 for the vertices of the triangle in 2-space. Let q_0, q_1 and q_2 denote the three texture coordinates, corresponding to these points. The simplest approach is to apply a simple linear interpolation procedure, mapping each point of the $\triangle p_0 p_1 p_2$ to its corresponding point in $\triangle q_0 q_1 q_2$.

What is wrong with this simple approach? The first problem has to do with perspective. The problem is that projective transformations are not linear transformations, but a linear interpolation is. An illustration of what can go wrong is shown in Fig. 2(c).

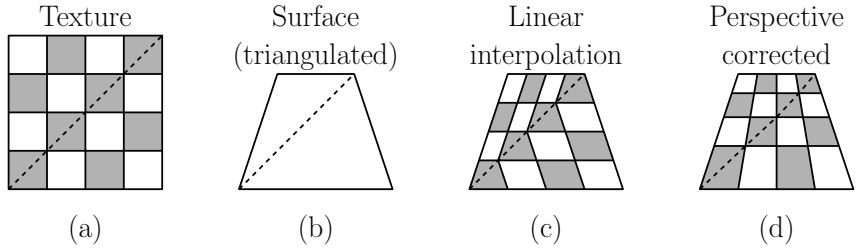

Fig. 2: Perspective correction.

There are a number of ways to fix this problem. One approach is to use a more complex formula for interpolation, which corrects for perspective distortion. (See the text for details. An example of the result is shown in Fig. 2(d)). This can be activated using the following OpenGL command:

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

(The other possible choice is GL_FASTEST, which does simple linear interpolation.)

The other method involves slicing the polygon up into sufficiently small polygonal pieces, such that within each piece the amount of distortion due to perspective is small. Recall that with Gauroud shading, it was often necessary to subdivide large polygons into small pieces for accurate lighting computation. If you have already done this, then the perspective distortion due to texture mapping may not be a significant issue for you.

Aliasing and mipmapping: A second problem with the aforementioned simple approach to interpolating texture coordinates has to do with something called *aliasing*. After determining the region of texture space onto which the pixel projects, we should blend the colors of the texels in this region to determine the color of the pixel. The problem with the above procedure does no blending, it determines the color on the basis of just a single texel. In situations where the pixel corresponds to a point in the distance and hence covers a large region in texture space, this may produce very strange looking results, because the color of the entire pixel is determined entirely by a single point in texture space that happens to correspond to the pixel's center coordinates (see Fig. 3(a)).

Dealing with aliasing in general is a deep topic, which is a main issue in the general field of signal processing. The process of blending samples together to smooth over the effects of aliasing is called *filtering*. In the field of signal processing there are many techniques (some quite sophisticated) for filtering in general. OpenGL applies a very simple method for dealing with the aliasing of this sort.

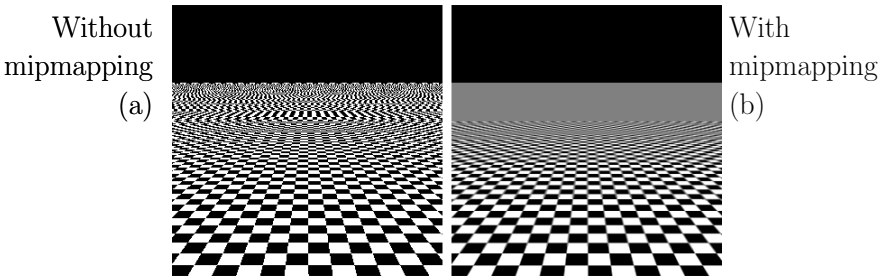

Fig. 3: Aliasing and mipmapping.

The method is called *mipmapping*. (The acronym "mip" comes from the Latin phrase *multum in parvo*, meaning "much in little.")

The idea behind mipmapping is to generate a series of texture images at decreasing levels of resolution. For example, if you originally started with a 128×128 image, a mipmap would consist of this image along with a 64×64 image, a 32×32 image, a 16×16 image, etc. All of these are scaled copies of the original image. Each pixel of the 64×64 image represents the average of a 2×2 block of the original. Each pixel of the 32×32 image represents the average of a 4×4 block of the original, and so on (see Fig. 4).

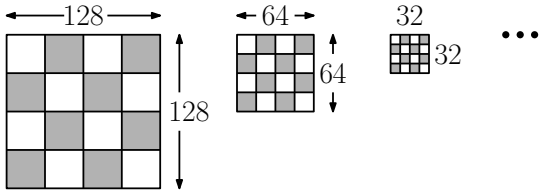

Fig. 4: Mipmapping.

Now, when OpenGL needs to apply texture mapping to a screen pixel that overlaps many texture pixels (that is, when "minimizing" the texture), it determines the mipmap in the hierarchy that is at the closest level of resolution, and uses the corresponding averaged pixel value from that mipmapped image. This results in more nicely blended images (see Fig. 4(b)).

Warning: If you use mipmapping in OpenGL (and this is not only a good idea, but the default for texture minimization), you *must* construct mipmaps for your texture. This can be done automatically for you by the command gluBuild2DMipmaps. (We will not discuss this command explicitly, but an example is given below.) If you fail to build mipmaps when they are needed, OpenGL will not produce an error message, it will simply disable texture mapping.

Texture mapping in OpenGL: OpenGL supports a fairly general mechanism for texture mapping. The process involves a bewildering number of different options. You are referred to the OpenGL documentation for more detailed information. By default, objects are not texture mapped. If you want your objects to be colored using texture mapping, you need to enable texture mapping before you draw them. This is done with the following command.

```
glEnable(GL_TEXTURE_2D);
```

After drawing textured objects, you can disable texture mapping.

If you plan to use more than one texture, then you will need to request that OpenGL generate texture objects. This is done with the following command:

```
glGenTextures(GLsizei n, GLuint* textureIDs);
```

This requests that n new texture objects be created. The n new texture id's are stored as unsigned integers in the array $textureIDs$. Each texture ID is an integer greater than 0. (Typically, these are just integers 1 through n, but OpenGL does not require this.) If you want to generate just one new texture, set $n = 1$ and pass it the address of the unsigned int that will hold the texture id.

By default, most of the texture commands apply to the "active" texture. How do we specify which texture object is the active one? This is done by a process called *binding*, and is defined by the following OpenGL command:

```
glBindTexture(GLenum target, GLuint textureID);
```

where $target$ is one of GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D, and $textureID$ is one of the texture IDs returned by glGenTextures. The $target$ will be GL_TEXTURE_2D for the sorts of 2-dimensional image textures we have been discussing so far. The $textureID$ parameter indicates which of the texture IDs will become the active texture. If this texture is being bound for the first time, a new texture object is created and assigned the given texture ID. If $textureID$ has been used before, the texture object with this ID becomes the active texture.

Presenting your Texture to OpenGL: The next thing that you need to do is to input your texture and present it to OpenGL in a format that it can access efficiently. It would be nice if you could just point OpenGL to an image file and have it convert it into its own internal format, but OpenGL does not provide this capability. You need to input your image file into an array of RGB (or possibly RGBA) values, one byte per color component (e.g. three bytes per pixel), stored row by row, from upper left to lower right. By the way, OpenGL requires images whose height and widths are powers of 2.

Once the image array has been input, you need to present the texture array to OpenGL, so it can be converted to its internal format. This is done by the following procedure. There are many different options, which are partially explained in below.

```
glTexImage2d(GL_TEXTURE_2D, level, internalFormat, width, height,
  border, format, type, image);
```

The procedure has an incredible array of options. Here is a simple example to present OpenGL an RGB image stored in the array myPixelArray. The image is to be stored with an internal format involving three components (RGB) per pixel. It is of width $nCols$ and height $nRows$. It has no border ($border = 0$), and we are storing the highest level resolution. (Other levels of resolution are used to implement the averaging process, through a method called *mipmaps*.) Typically, the level parameter will be 0 ($level = 0$). The format of the data that we will provide is RGB (GL_RGB) and the type of each element is an unsigned byte (GL_UNSIGNED_BYE). So the final call might look like the following:

```
glTexImage2d(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
```

In this instance, your array $myPixelArray$ is an array of size $256 \times 512 \times 3 = 393,216$ whose elements are the RGB values, expressed as unsigned bytes, for the 256×512 texture array. An example of a typical texture initialization is shown in the code block below. (The call to gluBuild2DMipmpaps is needed only if mipmapping is to be used.)

Initialization for a Single Texture

```
GLuint textureID;                       // the ID of this texture
glGenTextures(1, &textureID);           // assign texture ID
glBindTexture(GL_TEXTURE_2D, textureID); // make this the active texture
//
// ... input image nRows x nCols into RGB array myPixelArray
//
glTexImage2d(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
                                        // generate mipmaps - very important!
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, nCols, nRows, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
```

Texturing Options: Once the image has been input and presented to OpenGL, we need to tell OpenGL how it is to be mapped onto the surface. Again, OpenGL provides a large number of different methods to map a surface. These parameters are set using the following function:

```
glTexParameteri(target, param_name, param_value);
glTexParameterf(target, param_name, param_value);
```

The first form is for integer parameter values and the second is for float values. For most options, the argument $target$ will be set to GL_TEXTURE_2D. There are two common parameters to set. First, in order to specify that a texture should be repeated or not (clamped), the following options are useful.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

These options determine what happens of the s parameter of the texture coordinate is less than 0 or greater than 1. (If this never happens, then you don't need to worry about this option.) The first causes the texture to be displayed only once. In particular, values of s that are negative are treated as if they are 0, and values of s exceeding 1 are treated as if they are 1. The second causes the texture to be wrapped-around repeatedly, by taking the value of s modulo 1. Thus, $s = 5.234$ amd $s = 76.234$ are both equivalent to $s = 0.234$. This can independently be set for he t parameter of the texture coordinate, by setting GL_TEXTURE_WRAP_T.

Filtering and Mipmapping: Another useful parameter determines how rounding is performed during *magnification* (when a screen pixel is smaller than the corresponding texture pixel) and "*minification*" (when a screen pixel is larger than the corresponding texture pixel). The simplest, but not the best looking, option in each case is to just use the *nearest pixel* in the texture:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

A better approach is to use linear filtering when magnifying and mipmaps when minifying. An example is given below.

Combining Texture with Lighting: How are texture colors combined with object colors? The two most common options are GL_REPLACE, which simply makes the color of the pixel equal to the color of the texture, and GL_MODULATE (the default), which makes the colors of the pixel the product of the color of the pixel (without texture mapping) times the color of the texture. The former is for painting textures that are already *prelit*, meaning that lighting has already been applied. Examples include skyboxes and precomputed lighting for the ceiling and walls of a room. The latter is used when texturing objects to which lighting is to be applied, such as the clothing of a moving character.

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

Drawing a Texture Mapped Object: Once the initializations are complete, you are ready to start drawing. First, bind the desired texture (that is, make it the active texture), set the texture parameters, and enable texturing. Then start drawing your textured objects. For each vertex drawn, be sure to specify the texture coordinates associated with this vertex, prior to issuing the glVertex command. If lighting is enabled, you should also provide the surface normal. A generic example is shown in the code block below.

Drawing a Textured Object

```
glEnable(GL_TEXTURE_2D);              // enable texturing
glBindTexture(GL_TEXTURE_2D, textureID); // select the active texture
                                      // (use GL_REPLACE below for skyboxes)
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
                                      // repeat texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
                                      // reasonable filter choices
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
glBegin(GL_POLYGON);                  // draw the object(s)
  glNormal3f( ... );                  // set the surface normal
  glTexCoord2f( ... );                // set texture coords
  glVertex3f( ... );                  // draw vertex
  // ... (repeat for other vertices)
glEnd()
glDisable(GL_TEXTURE_2D);             // disable texturing
```

Parametrization of a Sphere (Optional): Let's make this more concrete with an example. Our shape is a unit sphere centered at the origin. We want to find the inverse wrapping function W^{-1} that maps any point (x, y, z) on the of the sphere to a point (s, t) in texture space.

We first need to come up with a surface parametrization for the sphere. We can represent any point on the sphere with two angles, representing the point's latitude and longitude. We will use a slightly different approach. Any point on the sphere can be expressed by two angles, φ and θ, which are sometimes called *spherical coordinates*. (These will take the roles of the parameters u and v mentioned above.)

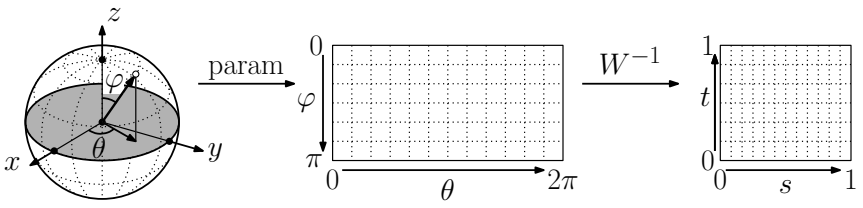

Fig. 5: Parametrization of a sphere.

Consider a vector from the origin to the desired point on the sphere. Let φ denote the angle in radians between this vector and the z-axis (north pole). So φ is related to, but not equal to, the latitude. We have $0 \leq \varphi \leq \pi$. Let θ denote the counterclockwise angle of the projection of this vector onto the xy-plane. Thus $0 \leq \theta < 2\pi$. (This is illustrated in Fig. 5.)

Our next task is to determine how to convert a point (x, y, z) on the sphere to a pair (θ, φ). It will be a bit easier to approach this problem in the reverse direction, by determining the (x, y, z) value that corresponds to a given parameter pair (θ, φ).

The z-coordinate is just $\cos\varphi$, and clearly this ranges from 1 to -1 as φ increases from 0 to π. To determine the value of θ, let us consider the projection of this vector onto the x, y-plane. Since the vertical component is of length $\cos\varphi$, and the overall length is 1 (since it's a unit sphere), by the Pythagorean theorem the horizontal length is $\ell = \sqrt{1 - \cos^2\varphi} = \sin\varphi$. The lengths of the projections onto the x and y coordinate axes are $x = \ell\cos\theta$ and $y = \ell\sin\theta$. Putting this all together, it follows that the (x, y, z) coordinates corresponding to the spherical coordinates (θ, φ) are

$$z(\varphi, \theta) = \cos\varphi, \qquad x(\varphi, \theta) = \sin\varphi \cdot \cos\theta, \qquad y(\varphi, \theta) = \sin\varphi \cdot \sin\theta.$$

But what we wanted to know was how to map (x, y, z) to (θ, φ). To do this, observe first that $\varphi = \arccos z$. It appears at first that θ will be much messier, but there is an easy way to get its value. Observe that $y/x = \sin\theta/\cos\theta = \tan\theta$. Therefore, $\theta = \arctan(y/x)$. In summary:

$$\varphi = \arccos z \qquad \theta = \arctan(y/x),$$

(Remember that this can be computed accurately as $\mathsf{atan2}(y, x)$.)

The final step is to map the parameter pair (θ, φ) to a point in (s, t) space. To get the s coordinate, we just scale θ from the range $[0, 2\pi]$ to $[0, 1]$. Thus, $s = \theta/(2\pi)$.

The value of t is trickier. The value of φ increases from 0 at the north pole π at the south pole, but the value of t decreases from 1 at the north pole to 0 at the south pole. After a bit of playing around

with the scale factors, we find that $t = 1 - (\varphi/\pi)$. Thus, as φ goes from 0 to π, this function goes from 1 down to 0, which is just what we want. In summary, the desired inverse wrapping function is $W^{-1}(x, y, z) = (s, t)$, where:

$$\begin{aligned} s &= \frac{\theta}{2\pi}, \quad \text{where} \quad \theta = \arctan\frac{y}{x} \\ t &= 1 - \frac{\varphi}{\pi}, \quad \text{where} \quad \varphi = \arccos z. \end{aligned}$$

Note that at the north and south poles there is a singularity in the sense that we cannot derive a unique value for θ. This is phenomenon is well known to cartographers. (What is the longitude of the north or south pole?)

To summarize, the *inverse wrapping* function $W^{-1}(x, y, z)$ maps a point on the surface to a point (s, t) in texture space. This is often done through a two step process, first determining the parameter values (u, v) associated with this point, and then mapping (u, v) to texture-space coordinates (s, t).This "unwrapping" function, maps the surface back to the texture. For this simple example, let's just set this function to the identity, that is, $W^{-1}(u, v) = (u, v)$. In general, we may want to stretch, translate, or rotate the texture to achieve the exact placement we desire.