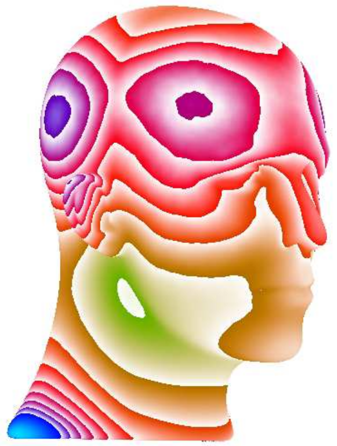
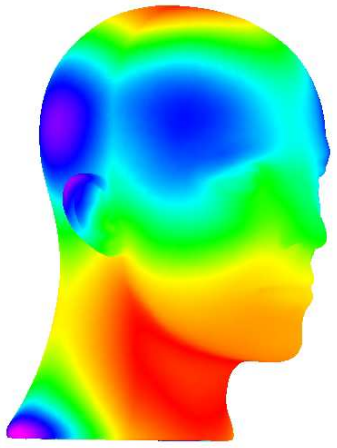
Textures and Bumps

Two dimensional texture pattern $T(s, t)$

The independent variables s and t are known as **texture coordinates**. At this point we can think of T as continuous, although, in reality, it is stored in texture memory as an $n \times m$ array of texture elements called **texels**.

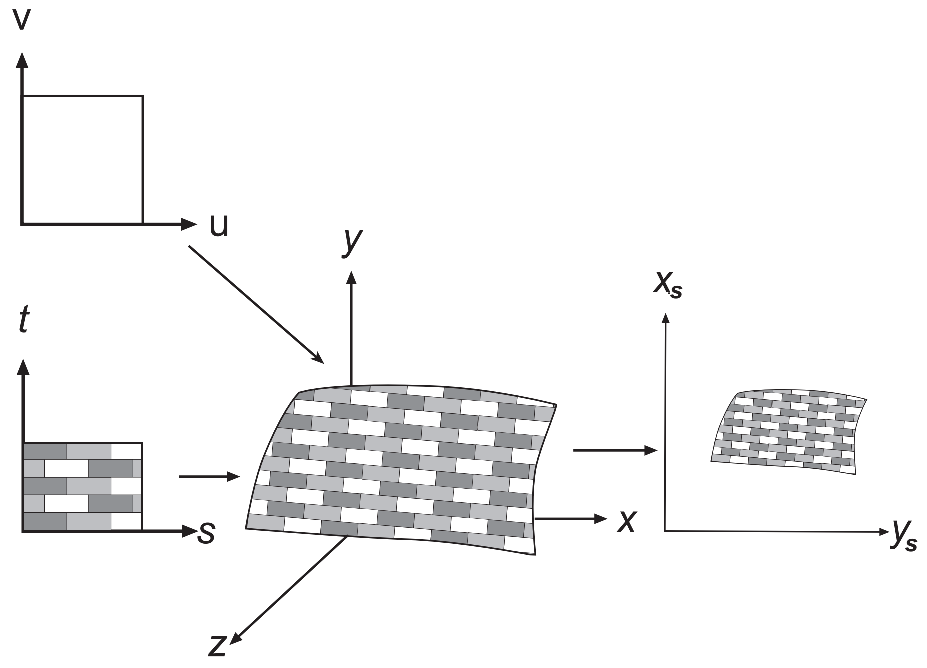
A **texture map** associates a unique point of T with each point on a geometric object that is itself mapped to screen coordinates for display.





Difficulties:

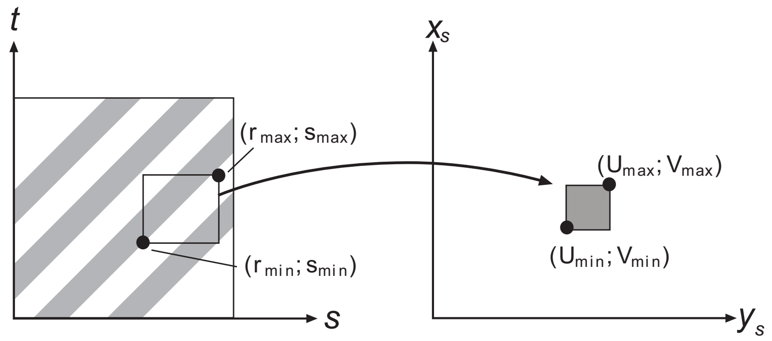
1. We must determine the map from texture coordinates to geometric coordinates.
2. Due to the nature of the rendering process, which works on a pixel-by-pixel basis, we are more interested in the inverse map from screen coordinates to texture coordinates.
3. Because we calculate the shade for pixels, each of which generates a color for a small rectangle on the display surface, we are interested in mapping not points to points, but rather area to areas.


v
u
t
s
y
x
z
x_s
y_s

Given a parametric surface, we can often map a point in the texture map $T(s,t)$ to a point on the surface $\mathbf{p}(u,v)$ with a linear map of the form

$$u = as + bt + c,$$

$$v = ds + et + f.$$



As long as $ae \neq bd$, this mapping is invertible. Linear mapping makes it easy to map a texture to a group of parametric surface patches. The patch determined by the corners $(s_{\min}, t_{\min})$ and $(s_{\max}, t_{\max})$ corresponds to the surface patch with corners $(u_{\min}, v_{\min})$

and $(u_{\max}, v_{\max})$, then the mapping is

$$u = u_{\min} + \frac{s - s_{\min}}{s_{\max} - s_{\min}}(u_{\max} - u_{\min}),$$
$$v = v_{\min} + \frac{t - t_{\min}}{t_{\max} - t_{\min}}(v_{\max} - v_{\min}).$$

Another approach to the mapping problem is to use a two-part mapping. The first step maps the texture to a simple three-dimensional intermediate surface, such as a sphere, cylinder, or cube. In the second step, the intermediate surface containing the mapped texture is mapped to the surface being rendered.

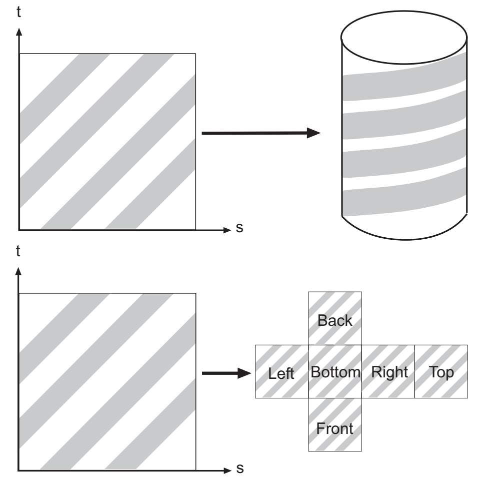
Suppose that our texture coordinates vary over the unit square, and that we use the surface of a cylinder of height h and radius r as our intermediate object.

$$x = r\cos(2\pi u),$$
$$y = r\sin(2\pi u),$$
$$z = v/h,$$

and u and v vary over $(0, 1)$. Hence, we can use the mapping

$$s = u,$$

$$t = v.$$



If we use a sphere of radius r as the intermediate surface, a possible mapping is

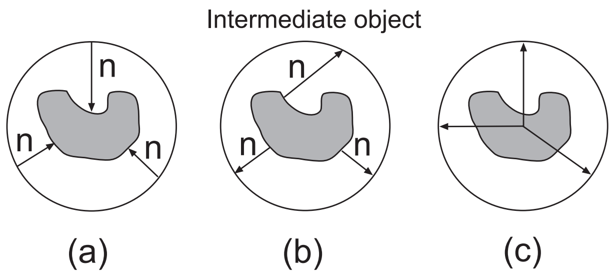
$$\begin{aligned} x &= \cos(2\pi u), \\ y &= \sin(2\pi u)\cos(2\pi v), \\ z &= \sin(2\pi u)\sin(2\pi v), \end{aligned}$$

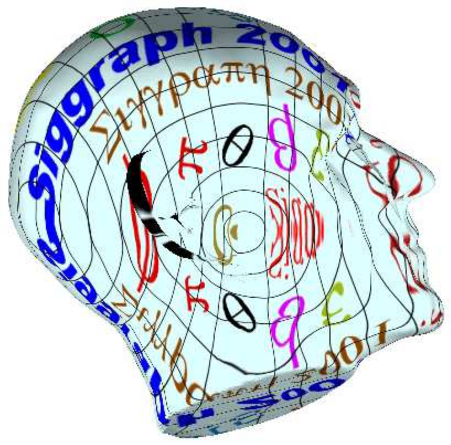
We can also use a rectangular box. Here, we map the texture to a box that can be unravelled, like a cardboard packing box. This mapping often is used with environment maps.

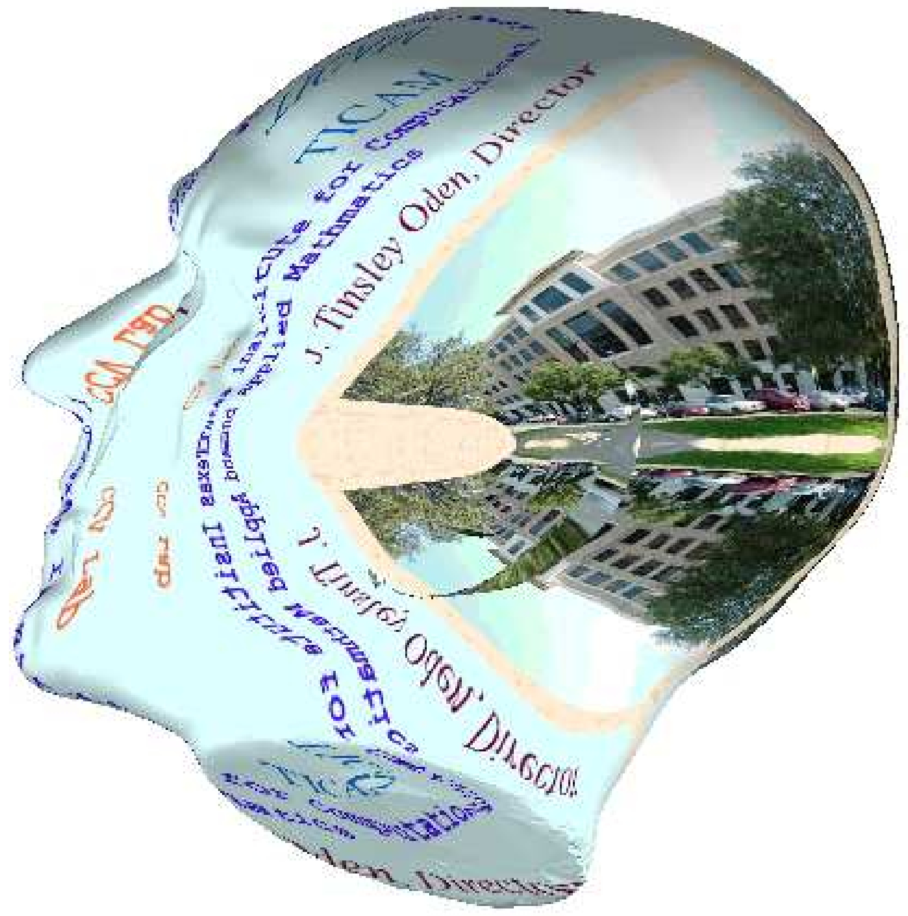
The second step is to map the texture values on the intermediate object to the desired surface.

1. We take the texture value at a point on the intermediate object, go from this point in the direction of the normal until we intersect the object, and then place the texture value at the point of intersection.
2. Reverse this method, starting at a point on the surface of the object and going in the direction of the normal at this point until we intersect the intermediate object, where we obtain the texture value.

3. If we know the center of the object, draw a line from the center through a point on the object, and to calculate the intersection of this line with the intermediate surface. The texture at the point of intersection with the intermediate object is assigned to the corresponding point on the desired object.






TICAM
Institute for Computational Engineering and Sciences
J. Tinsley Oden, Director

Texture Mapping in OpenGL

Two-dimensional texture mapping starts with an array of texels. Suppose that we have a 512×512 image `my_texels` that was generated by a program, or perhaps was read in from a file into an array

```
my_texels[512][512];
```

We specify that this array is to be used as a two-dimensional texture (usually as part of initialization) by

```
glTextImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0,
              GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

More generally, two-dimensional textures are specified through the function

```
glTextImage2D(GL_TEXTURE_2D, level, components, width, height,
              border, format, type, tarray);
```

The texture pattern `tarray` is stored in the `width` × `height` array. The `components` value is the number (1 through 4) of color components (RGBA) that we wish to affect with the map. The parameters `level` and `border` give us fine control over how texture is handled.

The texture map has two coordinates, s and t, both of which normally range over the interval (0.0, 1.0). For our example, the value (0.0, 0.0) corresponds to the point `my_texels[0][0]`, and (1.0, 1.0) corresponds to the point `my_texels[511][511]`.

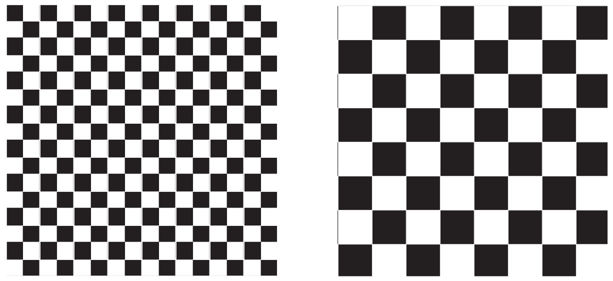
Assign texture coordinates to vertices through

```
glTexCood2f(s, t);
```

We must set the texture coordinate before we specify a vertex. If we want to assign our texture to a quadrilateral, then we use code such as

```
glBegin(GL_QUAD);
  glTexCood2f(0.0, 0.0);
  glVertex2f(x1, y1, z1);
  glTexCood2f(1.0, 0.0);
  glVertex2f(x2, y2, z2);
```

```
    glTexCood2f(1.0, 1.0);
    glVertex2f(x3, y3, z3);
    glTexCood2f(0.0, 1.0);
    glVertex2f(x4, y4, z4);
  glEnd();
```



(a) (b)

OpenGL has something called **mipmapping**. For objects that project to an area of screen space that is small compared with the size of the texel array, we do not need the resolution of the original texel array. OpenGL allows us to create a series of texture arrays at reduced sizes; it will then automatically use the appropriate size. For a 64×64 original array, we can set up 32×32, 16×16, 8×8, 4×4, 2×2, and 1×1 arrays through the GLU function

```
  gluBuid2DMipmaps(GL_TEXTURE_2D, 3, 64, 64, GL_RGB,
                    GL_UNSIGNED_BYTE, my_texels);
```

We can also set them through a set of `GL` functions. These mipmaps are invoked automatically if we specify

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST_MIPMAP_NEAREST);
```

Interaction Between Texture and Shading

For RGB colors, there are two options. The texture can modulate the shade that we would have assigned without texture mapping by multiplying the color components of the texture by the color components from the shader. Modulation is the default mode; it can be set by

```
glTexEnv(GL_TEX_ENV, GL_TEX_ENV_MODE, GL_MODULATE);
```

If we replace `GL_MODULATE` by `GL_DECAL`, the color of the texture determines the color of the object completely — a technique called **decaling**.

A growing technique for modulating the shade of textures (e.g. shadow on textured objects) is to use **multi-texture** units available on most current graphics cards.

Environment mapping is a form of texture mapping where the texture is derived from the environment. Once we obtain the required texture—either by scanning an image or through projecting a scene—OpenGL can automatically generate the tangent coordinates for a spherical mapping.

Bump Maps

The technique of **bump mapping** varies the apparent shape of the surface by perturbing the normal vectors as the surface is rendered; the colors that are generated by shading then show a variation in the surface properties.

We could perturb the normals in many ways; the following procedure for parametric surfaces is an efficient one. Let $\mathbf{p}(u, v)$ be a point on a parametric surface. The unit normal at that point is given by the cross product of the partial derivative vectors:

$$\mathbf{n} = \frac{\mathbf{p}_u \times \mathbf{p}_v}{|\mathbf{p}_u \times \mathbf{p}_v|}$$

where

$$\mathbf{p}_u = \begin{bmatrix} \frac{\partial x}{\partial u} \\ \frac{\partial y}{\partial u} \\ \frac{\partial z}{\partial u} \end{bmatrix} \qquad \mathbf{p}_v = \begin{bmatrix} \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial v} \\ \frac{\partial z}{\partial v} \end{bmatrix}$$

Suppose we display the surface in the normal direction by a given function called the **bump**

function, $d(u,v)$, which is assumed known and small $(|d(u,v)| \ll 1)$:

$$\mathbf{p}' = \mathbf{p} + d(u,v)\mathbf{n}.$$

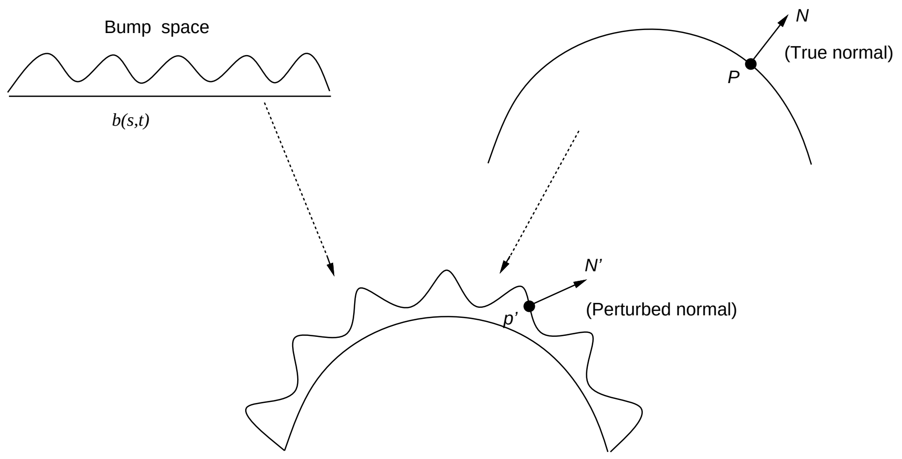
We would prefer not to display the surface; we just want to make it look as though we have displaced it. We can achieve the desired look by altering the normal $\mathbf{n}$, instead of $\mathbf{p}$, and using the perturbed normal in our shading calculations.

The normal at the perturbed point $\mathbf{p}'$ is given by the cross product

$$\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v.$$

We can compute the two partial derivatives by differentiating the equation for $\mathbf{p}'$, obtaining

$$\mathbf{p}'_u = \mathbf{p}_u + d_u\mathbf{n} + d(u,v)\mathbf{n}_u,$$
$$\mathbf{p}'_v = \mathbf{p}_v + d_v\mathbf{n} + d(u,v)\mathbf{n}_v.$$


Bump space
b(s,t)
N
(True normal)
P
N′
(Perturbed normal)
p′

Reading Assignment and News

Chapter 9 pages 487 - 492, of Recommended Text.

Please also track the News section of the Course Web Pages for the most recent Announcements related to this course.

(http://www.cs.utexas.edu/users/bajaj/graphics25/cs354/)