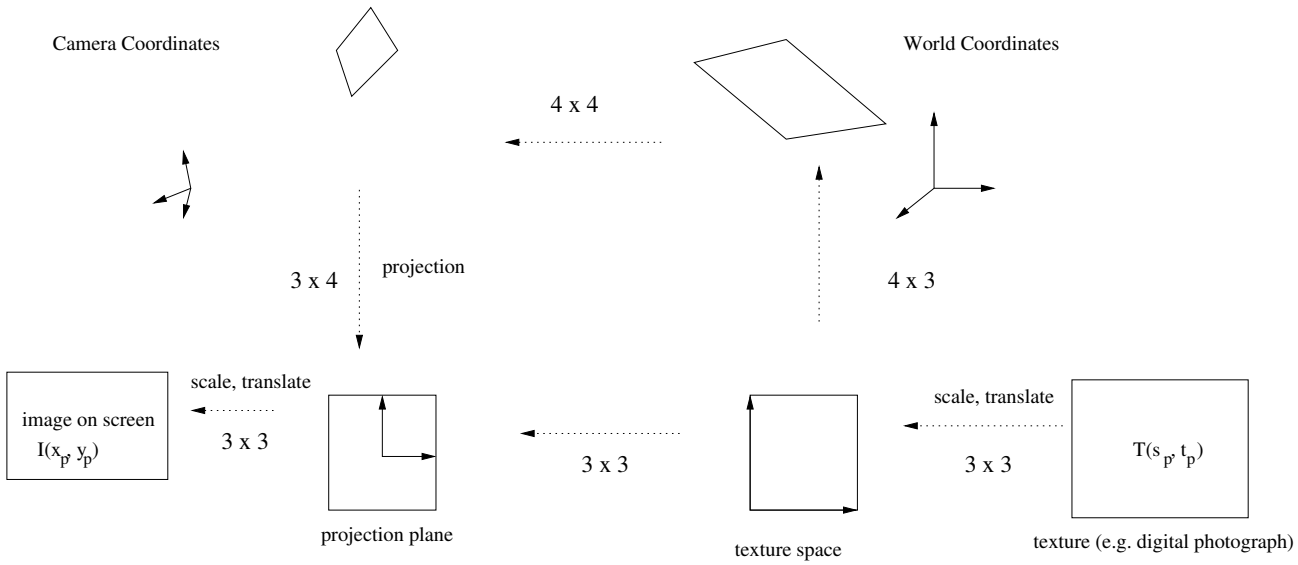
Texture Mapping

One often wishes to "paint" a surface with a certain pattern, or a *texture*. In general this is called *texture mapping*. The texture might be defined by a digital photograph $T(s_p, t_p)$ where (s_p, t_p) are pixels in the texture image. (These pixels are sometimes called "texels".) For example, suppose you want a ground surface to look like grass. You could model each blade of grass. Alternatively, you could just "paint" a grass pattern on the surface. The grass pattern could be just a photograph of grass. Similarly for a building: rather than making all the geometric details of the facade, you could justs use a single plane and paint the details. This might be sufficient if the building is in the background of the scene and the details are unlikely to be scrutinized.

To map a digital photograph onto a surface, one would need to compute the transformation that goes from the texture space (s_p, t_p) in which the digital photograph is defined, all the way to the pixel space (x_p, y_p) on the screen where the final image is displayed. The subscript p denotes that we are in discrete pixel space, rather than a continuum.

The mapping from the 2D texture space to the 2D pixel space can be illustrated with the sketch below. At the bottom right is a mapping from the texture image defined on (s_p, t_p) to the parameters (s, t) of the surface e.g. a square, a polygon. Then there is a mapping from these surface parameters (s, t) to world coordinates (which takes 2D to 3D) and a mapping from world coordinates to camera coordinates (3D to 3D) and the projection back to the 2D image domain (3D to 2D). Finally, there is the mapping from the image domain to the pixels (2D to 2D).

You've seen most of these mappings before. There an object to world and a world to camera ("modelview") and a projection. *In the lecture slides, I assumed that world coordinates were the same as camera coordinates, but in general that's not the case of course, and one needs to include that transformation in there.*



The projection is a bit subtle. It maps from camera coordinates to clip coordinates, which you recall are normalized device coordinates but prior to perspective division. So we have (wx, wy, wz, w). We are dropping the 3rd coordinate here, so it is just (wx, wy, w). The 3rd (z) coordinate is needed for hidden surface removal, but we are not dealing with that problem here.

The final mapping in the sequence is the window to viewport mapping, which we discussed in lecture 6.

You can see that the above sequence of linear mappings collapses to a 3×3 matrix, call it $\mathbf{H}$, and so we have the map:

$$\begin{bmatrix} wx_p \\ wy_p \\ w \end{bmatrix} = \mathbf{H} \begin{bmatrix} s_p \\ t_p \\ 1 \end{bmatrix}$$

The inverse of this map is:

$$\begin{bmatrix} ws_p \\ wt_p \\ w \end{bmatrix} = \mathbf{H}^{-1} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}.$$

To texture map the image $I(s_p, t_p)$ onto the polygon, we do the following:

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

    compute texel position using  (ws_p, wt_p, w) = H^{-1}(x_p, y_p, 1)
    I(x_p, y_p) = T(s_p, t_p)
}
```

Note that the computed texel position (s_p, t_p) generally will not be a pair of integers and so will not correspond to a unique texel. One therefore needs to be careful when choosing $T(s_p, t_p)$ for the color to write into the pixel. We will return to this issue later in the lecture.

Homography

The new concept here is the 3×3 linear transform $\mathbf{H}$ between texture coordinates $(s_p, t_p, 1)$ and the image plane coordinates $(x_p, y_p, 1)$. We examine this in more detail by looking at two components of the mapping, namely from $(s, t, 1)$ into world coordinates and then from world coordinates into the image projection plane.

The mapping from texture coordinates $(s, t, 1)$ into world coordinates is:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} a_x & b_x & x_0 \\ a_y & b_y & y_0 \\ a_z & b_z & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} \tag{1}$$

This mapping takes the origin $(s, t) = (0, 0)$ to (x_0, y_0, z_0). It takes the corner $(s, t) = (0, 1)$ to $(x_0 + b_x, y_0 + b_y, z_0 + b_z)$, and it takes the corner $(s, t) = (1, 0)$ to $(x_0 + a_x, y_0 + a_y, z_0 + a_z)$, etc. In homogeneous coordinates, it takes $(s, t, 1) = (0, 0, 1)$ to $(x_0, y_0, z_0, 1)$. It takes $(s, t, 1) = (1, 1, 1)$ to $(x_0 + a_x + b_x,\ y_0 + a_y + b_y,\ z_0 + a_z + b_z, 1)$, etc.

Let's assume, for simplicity, that world coordinates are the same as camera coordinates, i.e. there is an identity mapping between them. This is what I did in the slides. We then project points

from world/camera coordinates into the image plane $z = f$ by:

$$\begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This gives

$$\begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_x & b_x & x_0 \\ a_y & b_y & y_0 \\ a_z & b_z & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

We get 3×3 matrix, namely the product of a 3×4 matrix, and a 4×3 matrix. This gives a 3×3 linear transform from $(s, t, 1)$ to (wx, wy, w).

Note that in this example, for simplicity I am ignoring the normalization matrix $\mathbf{M}_{normalize}$. (Recall lecture 5.) In fact, this is only allowed in the special case that this matrix happens to be the identity matrix, e.g. if left = bottom = -1, top = right = 1.

In general, an invertible 3×3 matrix that operates on 2D points written in homogenous coordinates is called a *homography*. In the case of mapping from (s, t) to (x, y) space, we have in general

$$\begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}.$$

Note that mapping from normalized coordinates (s, t) to (x, y) or from pixel coordinates (s_p, t_p) to (x_s, y_s) are both homographies. Both are given by 3×3 matrices.

Example: a slanted ground plane

Let's consider a concrete example so you get a sense for how this works. Consider a plane:

$$z = z_0 - y \tan\theta$$

which we get by rotating the plane $z = 0$ by θ degrees about the x axis and then translating by $(0, 0, z_0)$. We parameterize the rotated 2D plane by (s, t) such that the origin $(s, t) = (0, 0)$ is mapped to the 3D point $(x, y, z) = (0, 0, z_0)$, the s axis is in the direction of the x axis. By inspection, the transformation from $(s, t, 1)$ to points on the rotated 3D plane, and then on to points on the image plane is:

$$\begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & 0 \\ 0 & -\sin\theta & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} \qquad (*)$$

Multiplying the two matrices in (*) gives the homography:

$$\mathbf{H} = \begin{bmatrix} f & 0 & 0 \\ 0 & f\cos\theta & 0 \\ 0 & -\sin\theta & z_0 \end{bmatrix}.$$

The inverse happens to be:

$$\mathbf{H}^{-1} = \begin{bmatrix} \frac{1}{f} & 0 & 0 \\ 0 & \frac{1}{f\cos\theta} & 0 \\ 0 & \frac{\tan\theta}{z_0 f} & \frac{1}{z_0} \end{bmatrix}$$

The 4×3 matrix mapping texture space to 3D may be mysterious at first glance, but note that the first mapping is just a rotation about the x axis, with the third column removed. This mapping can be thought of as rotating a plane $(s, t, 0)$, where we don't bother including the $z = 0$ in the mapping since it has no effect. Another way to think of this mapping is to break it down into its components.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & 0 \\ 0 & -\sin\theta & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} = s \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} 0 \\ \cos\theta \\ -\sin\theta \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ z_0 \\ 1 \end{bmatrix}$$

Although one can interpret the 4D vectors $(1, 0, 0, 0)$ and $(0, \cos\theta, -\sin\theta, 0)$ as points at infinity, this interpretation is not so helpful here. Rather think of $(1, 0, 0)$ and $(0, \cos\theta, -\sin\theta)$ as vectors that allow you to reach any point in the slanted plane, via linear combinations. (You need to use 4D vectors here because the slanted plane is shifted away from the origin and is not a 3D vector space i.e. it is not closed under addition.)

Aliasing

Recall from lecture 6 when I discussed how to scan convert a line:

```
m = (y1 - y0)/(x1 - x0)
y = y0
for x = round(x0) to round(x1)
   writepixel(x, Round(y), rgbValue)
   y = y + m
```

For each discrete `x` along the line, we had to choose a discrete value of `y` . To do so, we rounded off the `y` value. This gave us a set of pixels that approximated the original line.

If we represent a continuous object (line, curve, surface,..) with a finite discrete object then we have a many-to-one mapping and we lose information. We say that *aliasing* occur.s Here the word aliasing is used slightly differently from what you may be used to. In programming languages, aliasing refers to an object – or more generally to memory location – that has two different names.

```
x = new Dog()
y = x
```

The analogy in computer graphics (or signals in general) is that we have one discrete object (RGB values on a grid of pixels) that can arise from more than one continuous object (RGB values defined on a continuous parameter space).

To understand how aliasing arises in texture mapping, suppose I have continuous image $I(x, y)$ which is vertical stripes of width $w < 1$. The stripes have alternating intensities 0, 1, 0, 1, 0, 1 over

continuous intervals of width w. If you now sample this image along a row (fixed y), where the pixel positions are $x = 0, 1, 2, 3, ...$ then the values that we get will typically not be alternating 0's and 1's. For example, run the python code:

```
#  sample from alternating 0 and 1 values on intervals of width w
w = 0.38
n = 15                  # number of samples
result = zeros(n)
for x in range(n):
   if (x / w) % 2 > 1:
       result[x] = 1
print result
```

The output for the given parameters is:

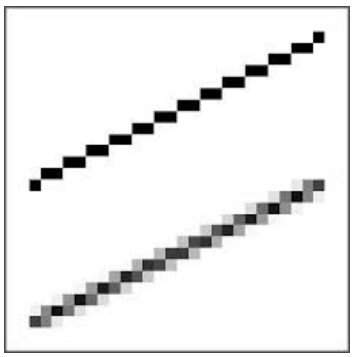
```
0  0  1  1  0  1  1  0  1  1  0  0  1  0  0
```

which is clearly not periodic. If you change w then you'll get a different output.

The main problem here is that the underlying image has structure that is at a finer scale than the sampling of the pixels. This problem often arises when you sample a continous signal. In the lecture slides I gave some other examples, including a Moiré pattern.

Anti-aliasing

In the case of scan converting a line, aliasing produces a jagged edge. Examples were given in the lecture slides. One idea to get rid of the jaggies (called "anti-aliasing") is to take advantage of the fact that RGB values are not binary but rather typically are 8 bit values (i.e. 0 to 255). For example, suppose that a line is supposed to be black (0,0,0) on a white (255,255,255) background. One trick to get rid of the jaggies is to make a pixel's intensity a shade of grey, if that pixel doesn't lie exactly on the line. The grey value of the pixel could depend on the closeness of the pixel to the continuous line. (There is no unique way to do this, and understand what's better or worse depends on factors take a while to explain – including issues in visual perception. We're not going down that rabbit hole.) See the example below. The first shows the jaggies; the second shows the "anti-aliased" line.



Aliasing and anti-aliasing in texture mapping

Aliasing arises in texture mapping as well. Recall the algorithm on page 2 of today's notes. For each pixel (x_p, y_p) in the image of a polygon, we map back to some pixel position (s_p, t_p) in the original texture. The simplest method would be to just round off (s_p, t_p) to the nearest integer and copy the (RGB) texture value $T(round(s_p), round(t_p))$ into $I(x_p, y_p)$.

A alternative approach is to consider a pixel as a small unit square, for example, centered around the point (x_p, y_p), namely the square $[x_p - \frac{1}{2}, x_p + \frac{1}{2}] \times [y_p - \frac{1}{2}, y_p + \frac{1}{2})]$. If we were to map the corners of this little square pixel back to the texture space (s_p, t_p) then we would get a four vertex polygon (quad) in that space. These four vertices typically would not correspond exactly to integer coordinates of the texture. GIven this lack of correspondence, how should one choose the color for the image pixel (x_p, y_p) ?

There are two general cases to consider. Here we consider texels and pixels to be unit squares in (s_p, t_p) and (x_p, y_p) space, respectively.

- *magnification*: This is the case that a texel in (s_p, t_p) maps forward to a region in (x_p, y_p) that is *larger* ("magnify") than a pixel in (x_p, y_p). Note that if we consider the mapping in the inverse direction, then a square image pixel centered at (x_p, y_p) would map back to a region that is *smaller* than a texel square in (s_p, t_p) space. Magnification can happen, for example, if the camera is close to the surface that is being texture mapped. (You may have noticed this when playing cheap video games, and you drive too close to a wall.)

- *minification*: This is the case that a texel in (s_p, t_p) maps forward to a region in (x_p, y_p) that is *smaller* ("minify") than a pixel in (x_p, y_p). In the inverse direction, a pixel in (x_p, y_p) maps back to a region that is *larger* than a texel in (s_p, t_p) space. In the case of a slanted ground plane, minification can happen pixels that are near the horizon.

Notice that smaller or larger is a bit ambiguous here since there is both an x and y direction. It could happen that there is a minification in one direction but a magnification in the other direction, for example, if you are close to a surface but the surface is highly slanted.

The method that one uses to choose the color $I(x_p, y_p)$ could depend on whether one has magnification or minification at that pixel. For example, if magnification occurs, then it can easily happen that the inverse map of an image pixel quad will not contain a texture pixel. To choose the color of the image pixel, one could ignore the inverse map of the image pixel quad and just consider the position of the inverse mapped pixel center. One could then choose the intensity based on the nearest texture pixel (s_p, t_p) or take a weighted average of the nearest four texture pixels, which form a square. This can be done with *bi-linear interpolation*. See the Exercises.

If one is minifying (shrinking) the texture, then the inverse map of the square image pixel might correspond to a larger region (quad) of the texture image. In this case, one might scan convert this region use the average of the RGB intensities.

As you can image, there is great flexibility in what one does here. There is no single approach that is best, since some approaches are more expensive than others, and the aliasing issues that arise depend on the texture being used and the geometry involved.

Texture Mapping in OpenGL

At the end of the lecture, I mentioned the basics of how texture mapping is done in OpenGL. First, you need to define a texture. For a 2D texture such as we have been discussing, you specify that it is 2D,[1] and various parameters such as its size (width and height) and the data i.e. RGB values.

```
glTexImage2D( GL_TEXTURE_2D, ...., size, .., data )
```

You also need to associate each vertex of your polygon with a texture coordinate in (s, t) space. You need to do this in order to define the mapping from texture coordinate space to your polygon, as discussed at the bottom of page 2 of these notes. This definition of (s, t) values for each vertex is occurs when you declare the polygon.

The texture coordinate, like a surface normal, is an OpenGL state. Each vertex is assigned the texture coordinate at that current state. If you want different vertices to have different texture coordinates – and you generally do want this when you do texture mapping – then you need to change the state from vertex to vertex. For example, here I define the texture coordinates to be the positions $(s, t) = (0, 0), (0, 1), (1, 0)$. You can define the texture coordinates to be any (s, t) values though.

```
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()
```

I'll say more about texture coordinates next lecture.

[1] Not all textures are 2D. You can define 1D and 3D textures too.