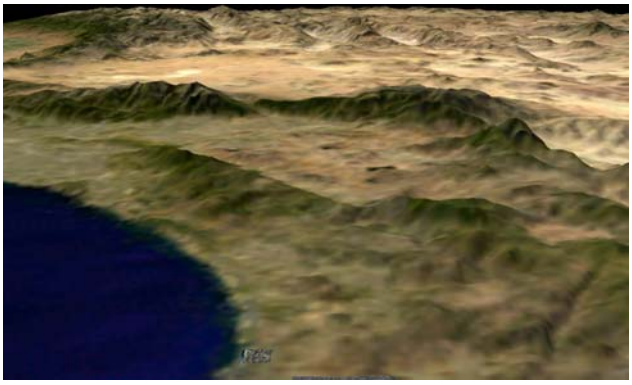
OpenGL Texture-Mapping Made Simpler

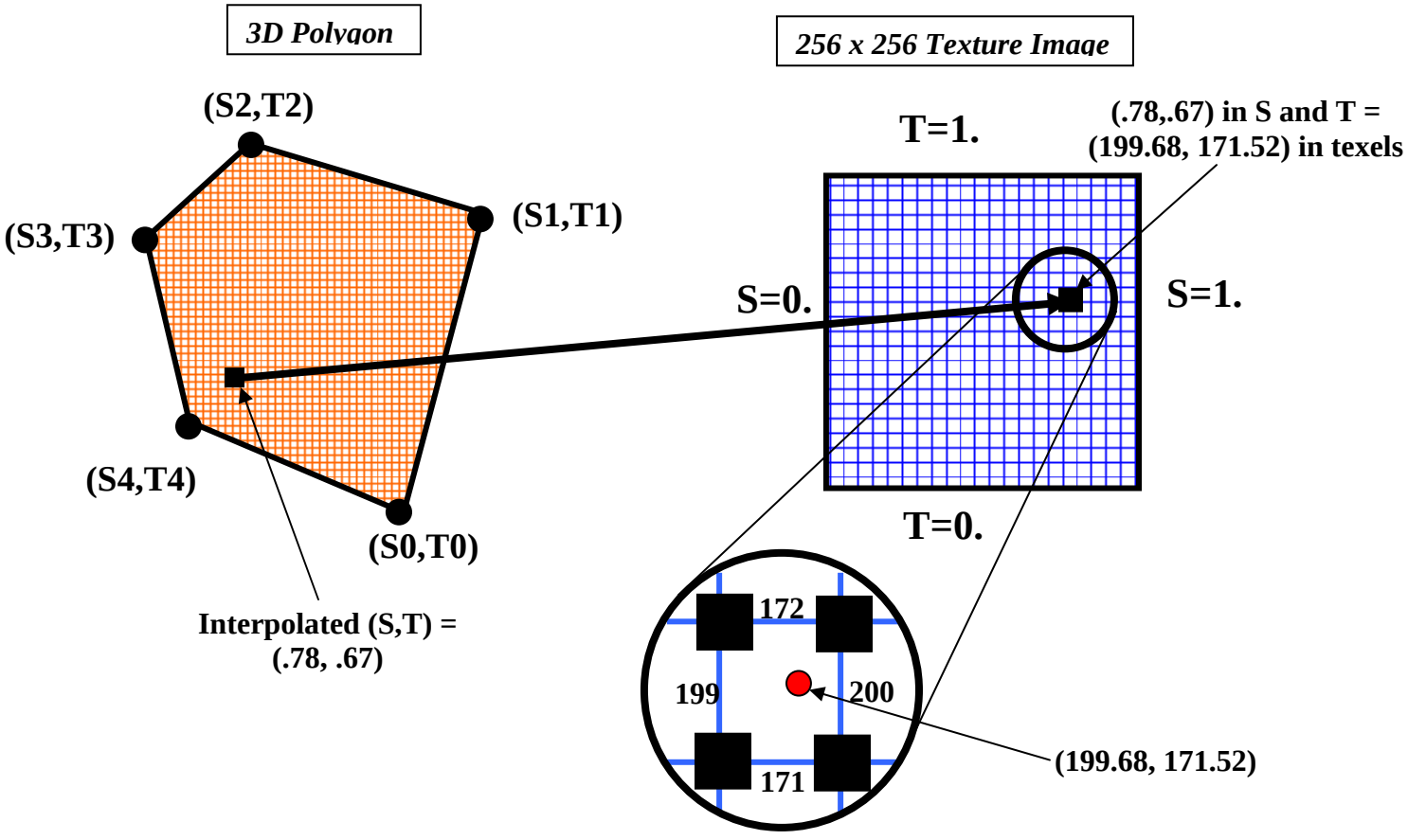
Introduction

Texture mapping is a computer graphics capability in which a separate image, referred to as the texture, is stretched onto a piece of 3D geometry and follows it however it is transformed. This image is also known as a *texture map*. This can be most any image, but its pixel dimensions must be a *power of two*. (This restriction is being lifted soon.) The X and Y dimensions do not need to be the *same* power of two, just a power of two. So, a 128x512 image would be OK, a 129x511 image would not.



Also, to prevent confusion, the texture pixels are not called *pixels*. A pixel is a dot in the final screen image. A dot in the texture image is called a texture element, or **texel**. Similarly, to avoid terminology confusion, a texture's width and height dimensions are not called *X* and *Y*. They are called **S** and **T**. A texture map is never indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always **S=0**., the right side is **S=1**., the bottom is **T=0**., and the top is **T=1.** Thus, you do not need to be aware of the texture's resolution when you are specifying coordinates that point into it. Think of S and T as a measure of what fraction of the way you are into the texture.

The mapping between the geometry of the 3D object and the S and T of the texture map works like this:



Getting a Texture for Your Program

Here are two ways to get a texture image for your program:

Create the texture yourself. There are different ways to store this information, depending on whether you are storing 1, 2, 3, or 4 values per texel. To create 3 values (RGB) per texel, for example:

```
unsigned char Texture[][3] =
{
        {     R0,   G0,   B0    },
        {     R1,   G1,   B1    },
        {     R2,   G3,   B2    },
        •••
};
```

where:

`R0, G0, B0`	the RGB values for the top-row, first-texel in the texture map, each in the range 0-255.
`R1, G1, B1`	the RGB values for the top-row, second texel in the texture map.

```
unsigned char Texture[][4] =
{
        {     R0,   G0,   B0,   A0    },
        {     R1,   G1,   B1,   A1    },
        {     R2,   G3,   B2,   A2    },
        •••
};
```

where:

`R0, G0, B0, A0`	the RGBA values for the top-row, first-texel in the texture map, each in the range 0-255.
`R1, G1, B1, A1`	the RGBA values for the top-row, second texel in the texture map.

Read the texture from an image file:

We will use code from nehe.gamedev.net to load a bmp file as a texture array.

This code can be found in the loadtex.h and loadtex.cpp files. The file *filename.bmp* needs to have dimensions that are powers of 2.

Preparing to Draw:

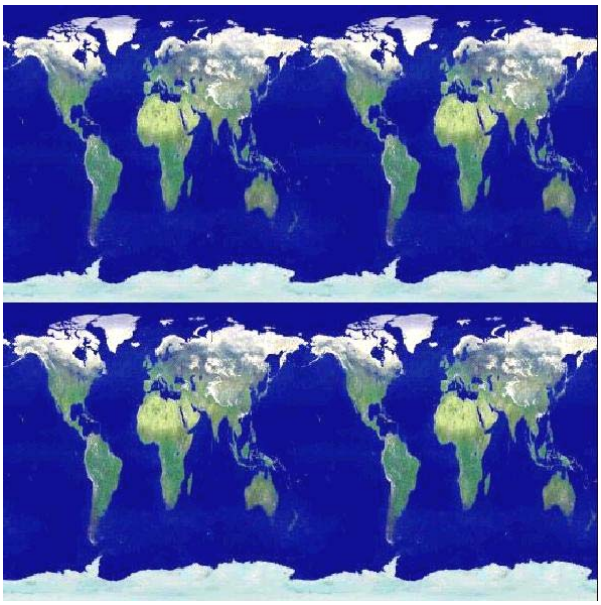
1. Define the texture wrapping parameters. This will control what happens when a texture coordinate greater than 1.0 or less than 0.0 is encountered:

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap );
```

where *wrap* is:

`GL_REPEAT`	specifies that this pattern will repeat (i.e., wrap-around) if transformed texture coordinates less than 0.0 or greater than 1.0 are encountered.
`GL_CLAMP`	specifies that the pattern will "stick" to the value at 0.0 or 1.0.

Clamping



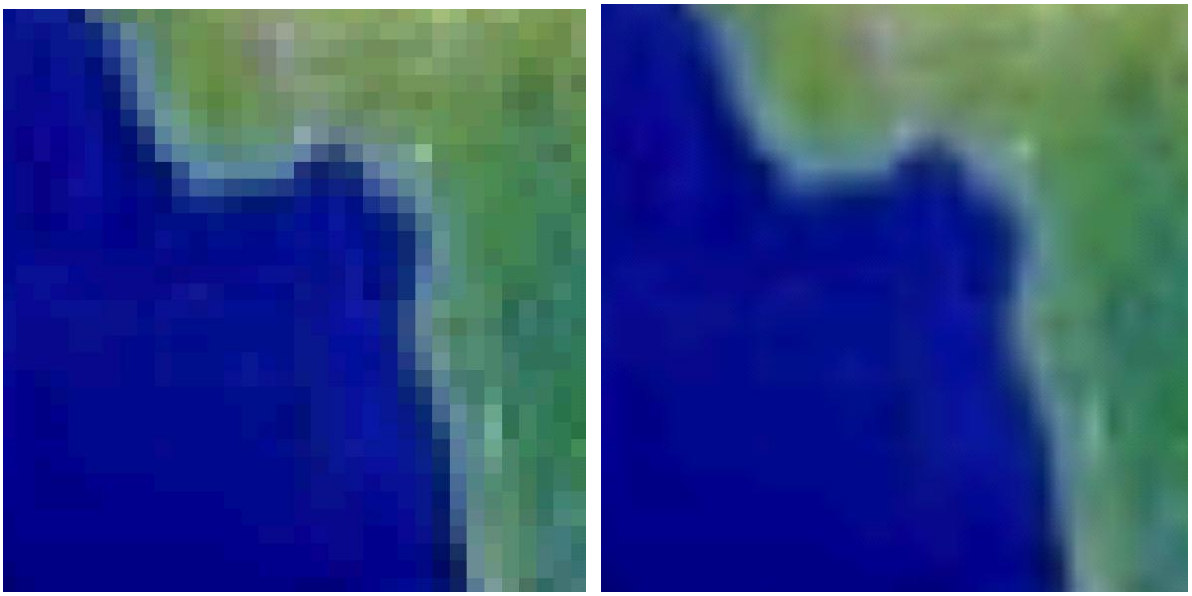
Repeating

2. Define the texture filter parameters. This will control what happens when a texture is scaled up or down.

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

where *filter* is:

`GL_NEAREST`	specifies that point sampling is to be used when the texture map needs to be magnified or minified.
`GL_LINEAR`	specifies that bilinear interpolation among the four nearest neighbors is to be used when the texture map needs to be magnified or minified.



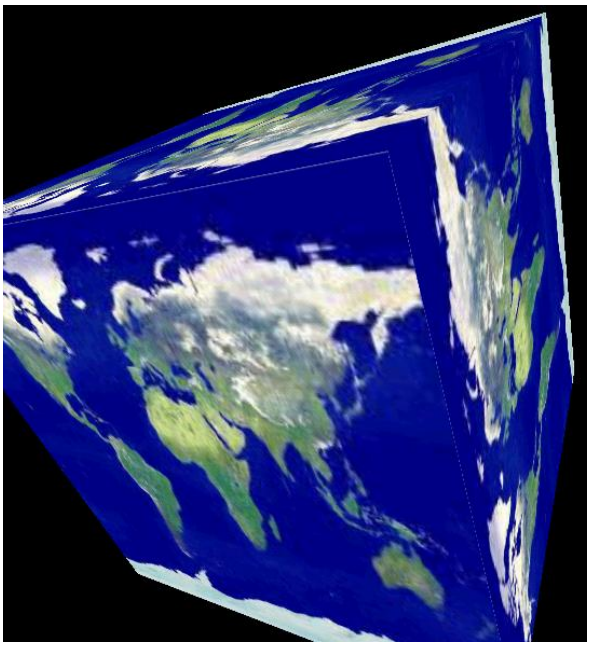
Nearest **Bilinear**

3. Define the texture environment properties.

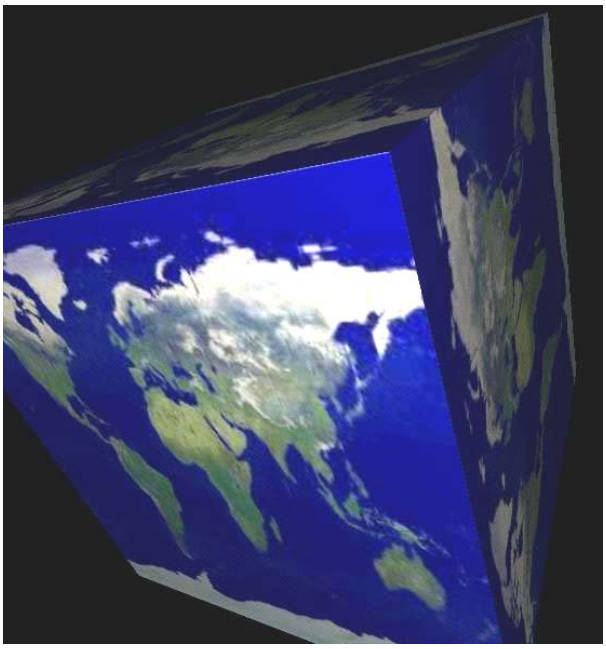
```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode );
```

where mode is either:

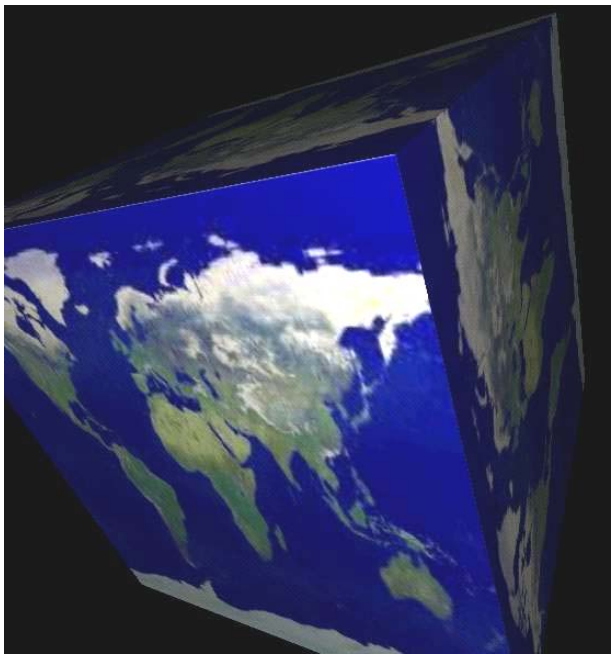
`GL_REPLACE`	specifies that the 3-component texture will be applied as an opaque image on top of the polygon.
`GL_MODULATE`	specifies that the 3-component texture will be applied as piece of colored plastic on top of the polygon. The polygon color "shines" through the plastic texture. This is very useful for applying lighting to textures: paint the polygon white with lighting and let it shine up through a texture.



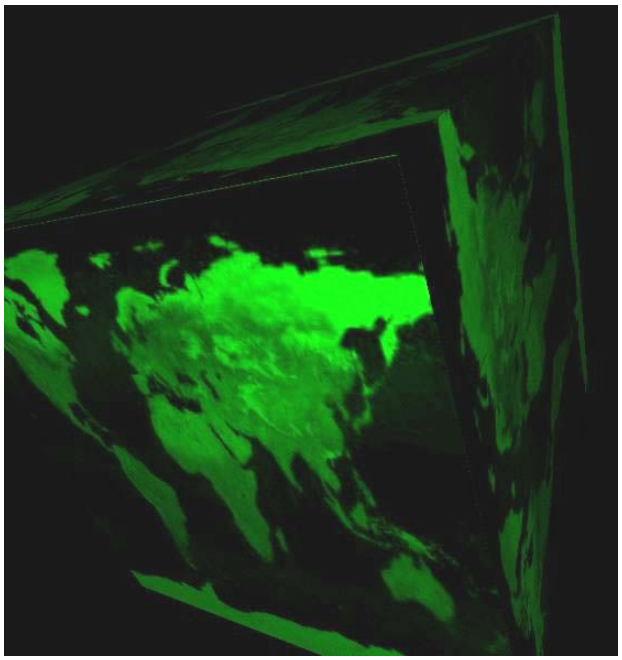
Replace



Modulate



Modulate with a White Cube



Modulate with a Green Cube

4. Tell the system what the current texture is:

```
int level, ncomps, width, height, border;
unsigned char *Texture;
•••
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glTexImage2D(  GL_TEXTURE_2D,  level,  ncomps,  width,  height,  border,
      GL_RGB, GL_UNSIGNED_BYTE, Texture );
```

where:

`level`	is used with mip-mapping. Use 0 for now.
`ncomps`	number of components in this texture: 3 if using RGB, 4 if using RGBA.
`width`	width of this texture map, in pixels.
`height`	height of this texture map, in pixels.
`border`	width of the texture border, in pixels. Use 0 for now.
`Texture`	the name of an array of unsigned characters holding the texel information.

Note that you can get away with specifying this ahead of time only if you are using a single texture. If you are using multiple textures, you must specify them in `Display()` right before you need them.

In `Display()`:

1. Set the current texture using `glTexImage2D()` if you are using multiple textures.

2. Enable texture mapping:

```
glEnable( GL_TEXTURE_2D );
```

3. Draw your polygons, specifying **s** and **t** at each vertex:

```
glBegin( GL_POLYGON );
      glTexCoord2f( s0, t0 );
      glNormal3f( nx0, ny0, nz0 );
      glVertex3f( x0, y0, z0 );

      glTexCoord2f( s1, t1 );
      glNormal3f( nx1, ny1, nz1 );
      glVertex3f( x1, y1, z1 );

      •••
glEnd();
```

If this geometry is static (i.e., will never change), it is a good idea to put this all into a display list.

4. Disable texture mapping:

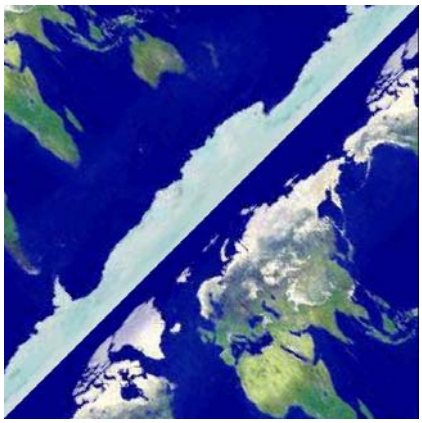
```
glDisable( GL_TEXTURE_2D );
```

Transforming Texture Coordinates

In addition to the projection and modelview matrices, OpenGL maintains a transformation for texture map coordinates **S** and **T** as well. You use all the same transformation routines you are used to: `glRotatef(), glScalef(), glTranslatef(),` but you must first specify the Matrix Mode:

```
glMatrixMode( GL_TEXTURE );
```

The only trick to this is to remember that you are transforming the *texture coordinates*, not the *texture image*. Of course, transforming the texture image forward is the same as transforming the texture coordinates backwards, so this should be pretty intuitive:



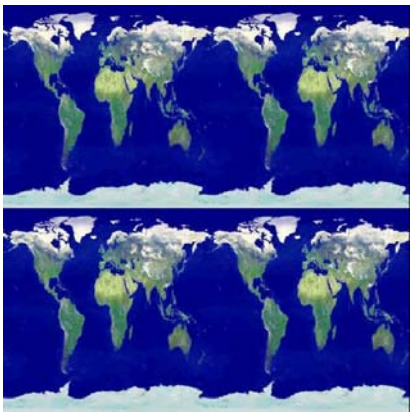
Rotate by -45°

Rotate by 0°

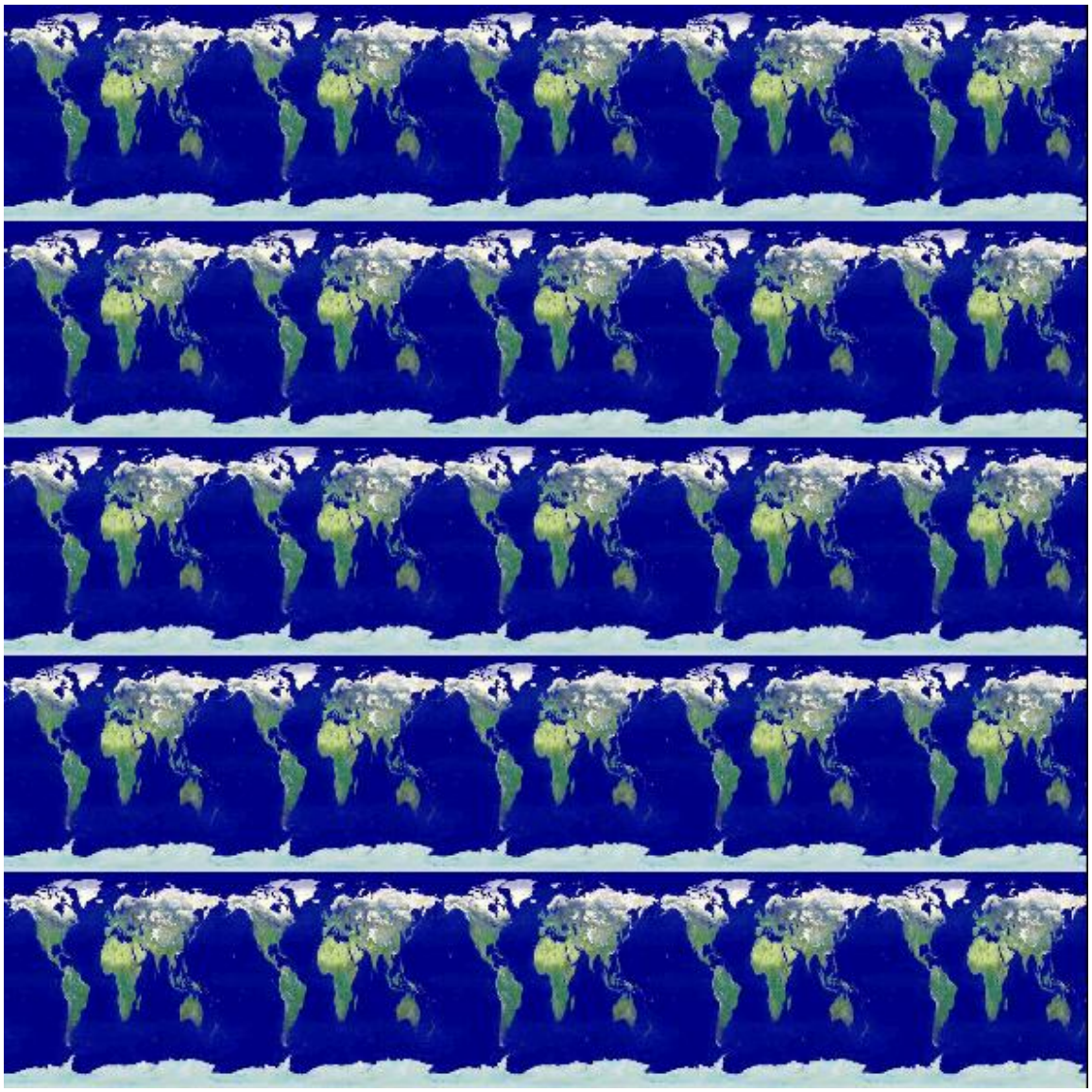
Rotate by +45°

Scale by 0.5

Scale by 1.



Scale by 2.



Scale by 5.

Texture Objects

If your scene has only one texture, you are lucky. But, if you have more than one texture, it may be important to maximize the efficiency of how you create, store, and manage those textures. In this case you should use **texture objects**.

Earlier in these notes, we created a texture by saying:

```
glTexImage2D( GL_TEXTURE_2D, level, ncomps, width, height, border,
      GL_RGB, GL_UNSIGNED_BYTE, Texture );
```

This worked, but if you do this for every texture you are managing, then each texture will be downloaded to the graphics card every time you call `Display()`. You fix this by using texture objects. Texture objects leave your textures on the graphics card and re-use them, which is always going to be faster than re-loading them.

Create a texture object by generating a texture name and then bind the texture object to the texture data and texture properties. The first time you execute `glBindtexture()`, you are creating the texture object. Subsequent times you do this, you are invoking the texture object. So, for example, you might say:

```
GLuint tex0, tex1;

. . .

glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );

glGenTextures( 1, &tex0 );
glGenTextures( 1, &tex1 );

. . .

glBindTexture( GL_TEXTURE_2D, tex0 );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );
glTexImage2D( GL_TEXTURE_2D, 0, 3, 256, 128, 0, GL_RGB,
      GL_UNSIGNED_BYTE, TextureArray0 );


glBindTexture( GL_TEXTURE_2D, tex1 );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
glTexImage2D( GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB,
      GL_UNSIGNED_BYTE, TextureArray1 );
```

Then, later on in `Display()`:

```
glEnable( GL_TEXTURE_2D );

glBindTexture( GL_TEXTURE_2D, tex0 );
glBegin( GL_QUADS );
      •••
glEnd();


glBindTexture( GL_TEXTURE_2D, tex1 );
glBegin( GL_TRIANGLE_STRIP );
      •••
glEnd();
```

Managing Texture Objects

At some point, you can end up with more texture than you have texture memory. OpenGL will react to this by discarding some of the textures, re-using the texture memory, and then re-loading the discarded textures. There are usually some set of textures that you know, from the nature of your application, it would be better if they were never discarded. This might be because they are large or because they are used more often. Regardless of the reason, you can prioritize the texture objects so that the most critical ones are the least likely to ever be discarded from on-board texture memory. To do this, say something like this:

```
GLuint   TextureNames[5];
GLclampf Priorities[5];

TextureNames[0] = tex0;       Priorities[0] = 0.5;
TextureNames[1] = tex1;       Priorities[1] = 1.0;
. . .

glPrioritizeTextures( 5, TextureNames, Priorities );
```

Elements of the `Priorities[]` array are in the range 0. to 1., where 0. represents the lowest priority (most likely to be discarded if necessary) and 1. represents the highest priority (least likely to be discarded).