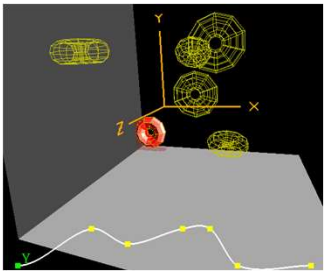
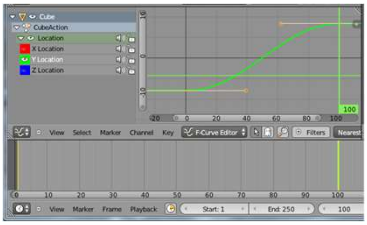
Simple Keytime Animation for CS 450/550



Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu



Approaches to Animation

1. Motion Capture ("MoCap")
2. Using the laws of physics (F=ma)
3. Using functional (target-driven) animation
4. **Using keyframing**

We'll talk more about these others in our Animation notes!

Keyframing

3

Keyframing involves creating certain *key* positions for the objects in the scene, and then the program later interpolating the animation frames *in between* the key frames.

In hand-drawn animation, the key frames are created by the senior animators, and the in-between frames are developed by the junior animators.

In our case, you are going to be the senior animator, and the computer will do the in-betweening.

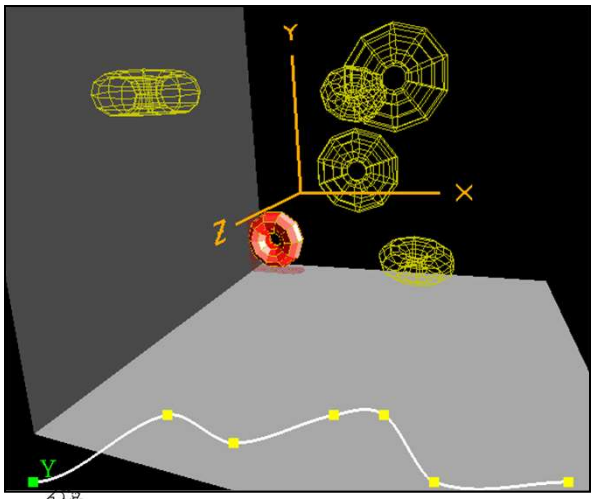
Oregon State University
Computer Graphics

mjb – August 22, 2024

3

4

The General Idea is to Interpolate the In-between Frames from the Smooth Curves Fit through the Key Frames



Oregon S
Universi
Computer Graphics

To make this simple to use, our goal is to just specify the keyframe *values*, not the *slopes*. We will let the computer compute the slopes for us, which will then let the in-between frames be computed.

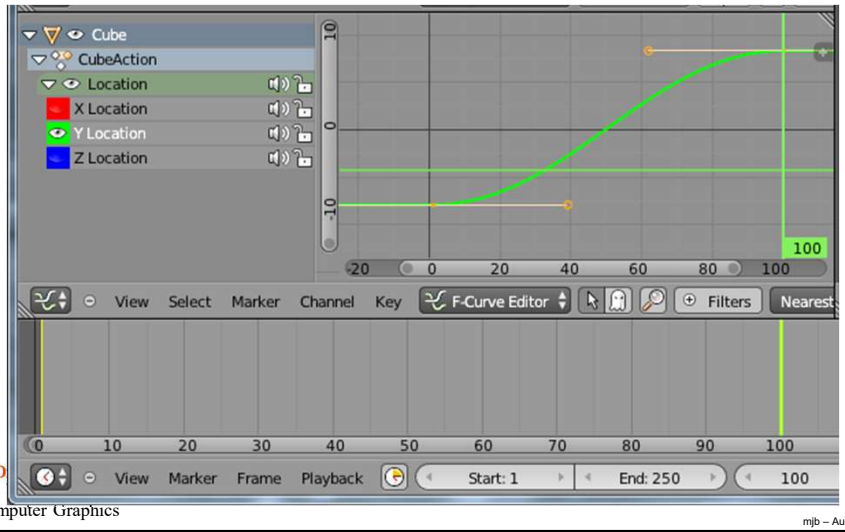
mjb – August 22, 2024

4

5

Many Professional Animation Packages Make You Sculpt the Slopes (but we won't . . .)

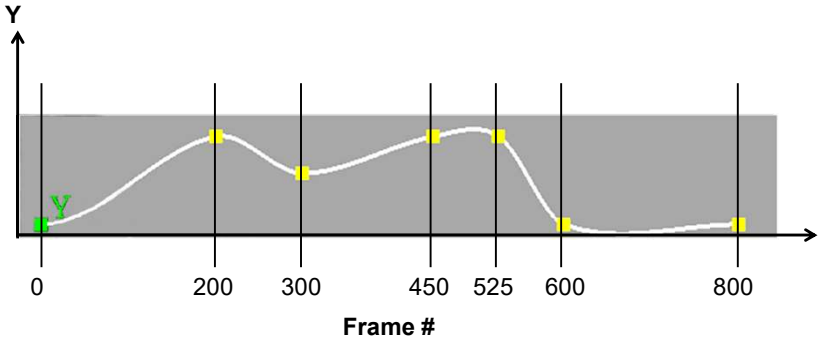
Blender:



5

6

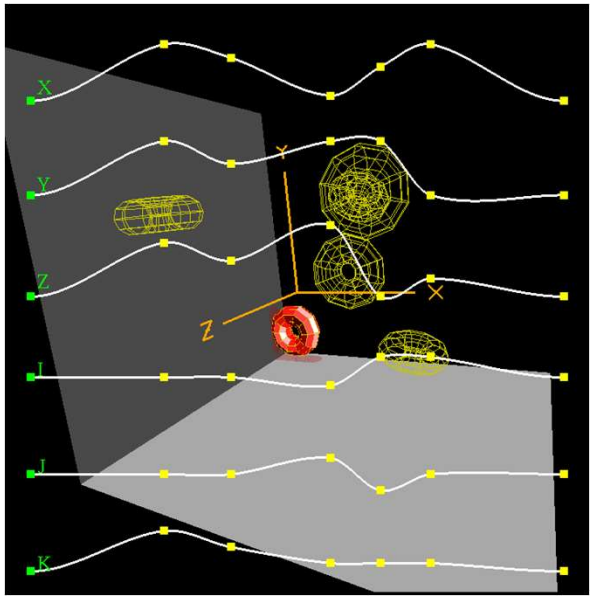
The "Y vs. Frame" Curve Looks Like This



6

7

Do This Same Thing for the X, Y, and Z Translations and the X, Y, and Z Rotations



X

Y

Z

θ_X

θ_y

θ_z

Oregon State University
Computer Graphics

7

8

Instead of Key *Frames*, I Like Specifying Key *Times* Better

We created a C++ class to do the interpolation for you

class Keytimes:

```
void   AddTimeValue( float time, float value );
float  GetFirstTime( );
float  GetLastTime( );
int    GetNumKeytimes( );
float  GetValue( float time );
void   Init( );
void   PrintTimeValues( );
```

Un-comment this line in your sample code:

```
//#include "keytime.cpp"
```

Oregon State University
Computer Graphics

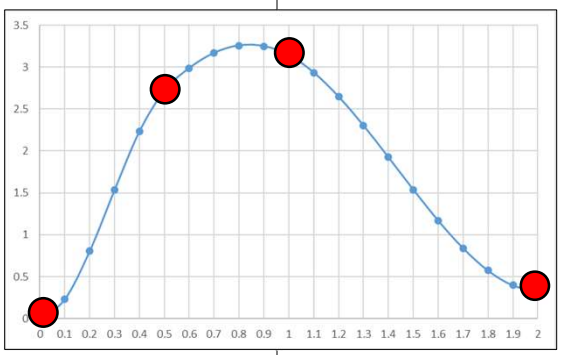
8

Instead of Key *Frames*, I Like Specifying Key *Times* Better

```
Keytimes Xpos;                          // global variable

int
main( int argc, char *argv[ ] )
{
        // do this in main or in InitGraphics( ):
        Xpos.Init( );
        Xpos.AddTimeValue(  0.0,  0.000 );
        Xpos.AddTimeValue(  2.0,  0.333 );
        Xpos.AddTimeValue(  1.0,  3.142 );
        Xpos.AddTimeValue(  0.5,  2.718 );
        fprintf( stderr, "%d time-value pairs:\n", Xpos.GetNumKeytimes( ) );
        Xpos.PrintTimeValues( );

        fprintf( stderr, "Time runs from %8.3f to %8.3f\n", Xpos.GetFirstTime( ), Xpos.GetLastTime( ) );

        for( float t = 0.f; t <= 2.f; t += 0.1f )
        {
                float v = Xpos.GetValue( t );
                fprintf( stderr, "%8.3f\t%8.3f\n", t, v );
        }

        . . .
```


Instead of Key *Frames*, I Like Specifying Key *Times* Better

```
(  0.00,   0.000)
(  0.00,   0.000)  (  2.00,   0.333)
(  0.00,   0.000)  (  1.00,   3.142)  (  2.00,   0.333)
(  0.00,   0.000)  (  0.50,   2.718)  (  1.00,   3.142)  (  2.00,   0.333)
4 time-value pairs
Time runs from    0.000 to    2.000
   0.000        0.000
   0.100        0.232
   0.200        0.806
   0.300        1.535
   0.400        2.234
   0.500        2.718
   0.600        2.989
   0.700        3.170
   0.800        3.258
   0.900        3.250
   1.000        3.142
   1.100        2.935
   1.200        2.646
   1.300        2.302
   1.400        1.924
   1.500        1.539
   1.600        1.169
   1.700        0.840
   1.800        0.574
   1.900        0.397
   2.000        0.333
```



Using the System Clock in Display( ) for Timing

```
#define MSEC       10000                    // i.e., 10 seconds
Keytimes Xpos, Ypos, Zpos;
Keytimes ThetaX, ThetaY, ThetaZ;

// in InitGraphics( ):

<< init the Keytime classes and add the keyframe values >>
. . .

// in Display( ):

      // # msec into the cycle ( 0 - MSEC-1 ):
      int msec = glutGet( GLUT_ELAPSED_TIME )  %  MSEC;

      // turn that into a time in seconds:
      float nowSecs = (float)msec  / 1000.f;
      glPushMatrix( );
            glTranslatef( Xpos.GetValue(nowSecs), Ypos.GetValue(nowSecs), Zpos.GetValue(nowSecs)  );
            glRotatef( ThetaX.GetValue(nowSecs),  1., 0., 0. );
            glRotatef( ThetaY.GetValue(nowSecs),  0., 1., 0. );
            glRotatef( ThetaZ.GetValue(nowSecs),  0., 0., 1. );
            << draw the object >>
      glPopMatrix( );
}
```

Number of msec in the animation cycle

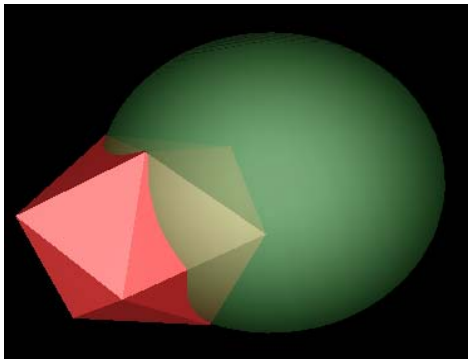
OpenGL Transparency

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

OpenGL has a nice feature that lets you display see-through objects. This is useful in visualization when wanting to see some objects through other objects.

OpenGL calls it **"transparency",** but in fact it is really **"blending"**. Be sure to remember this. Real transparency is a subtractive-color process. For example, looking at a pure red object through a pure green piece of glass will give you **black**.

OpenGL "transparency" would instead blend the RGB of the red object with the RGB of the green glass, giving a shade of yellow.



When Defining Your Object

Instead of using **glColor3f( )** to specify the red, green, and blue of an object, use **glColor4f( )** to specify red, green, blue, and *alpha*. Alpha is the transparency factor.

Or, if you are using lighting,

glMaterialfv(GL_FRONT, GL_AMBIENT, rgba);
glMaterialfv(GL_FRONT, GL_DIFFUSE, rgba);

- An alpha value of **0.0** means that this object is **completely transparent** (i.e., invisible – not too useful).
- An alpha value of **1.0** means that this object is completely **opaque** (also not useful as a transparency).

$$C' = \alpha C_{new} + (1. - \alpha) C_{old}$$

Oregon State University
Computer Graphics

In the Display( ) Callback

1. Draw the solid things first
2. Enable color blending:

   glEnable(GL_BLEND);

3. Make the Z-buffer read-only:

   glDepthMask(GL_FALSE);

   This is important because you don't want the presence of a transparent object close to your eye to prevent the writing of its blend with an object a little farther away.

Oregon State University
Computer Graphics

In the Display( ) Callback

4. Define how much of the about-to-be-written pixel color (the "source") and how much of the already-existing pixel color (the "destination") will end up being used:

   glBlendFunc(src, dst);

   The value for src multiplies the about-to-be-written source pixel color (S) and the value of dst multiplies the already-existing destination pixel color (D). While there are several options for src and dst, the most useful combination is:

Src (C_{new})	Dst (C_{old})	Result (C')
GL_SRC_ALPHA	GL_ONE_MINUS_SRC_ALPHA	C' = αS + (1-α)D

5. Draw the transparent things.

6. After drawing all the transparent elements of the scene, set the depth mask back to read-write and disable blending:

   glDepthMask(GL_TRUE);
   glDisable(GL_BLEND);

Oregon State University
Computer Graphics

Tips

- Remember that the way OpenGL implements this is not really transparency, it is *blending*. True transparency is a color-subtractive process. Blending is a color-additive process. If the goal is to make a yellow window look and behave like a real yellow window, this won't work correctly. If the goal is to just see inside something, this works very well.
- Recognize that OpenGL picking will know nothing about your transparency. As far as it is concerned, you drew all solid, opaque polygons.
- An example from the world of medical visualization:

University
Computer Graphics

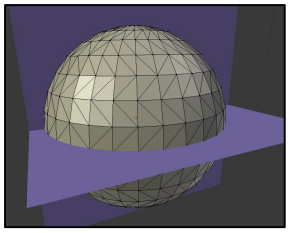
Texture Mapping

Oregon State University

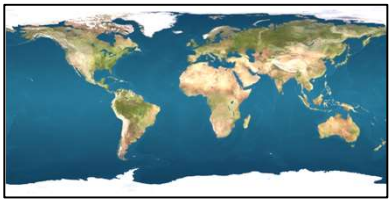
Mike Bailey

mjb@cs.oregonstate.edu

The Basic Idea: Wrap an Image Around a Piece of Geometry



+



=

In software, this is a very slow process. In hardware, this is very fast. The development of texture-mapping hardware was one of the most significant events in the history of computer graphics. This is really what finally enabled game development on a realistic scale.

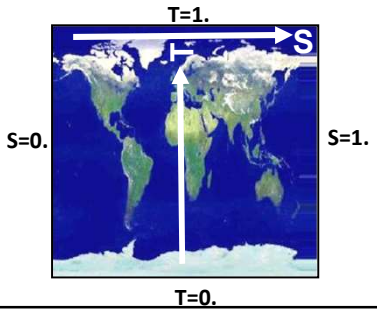
The Basic Ideas

To prevent confusion, the texture image 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 image's width and height dimensions are not called *X* and *Y*. They are called **S** and **T**.

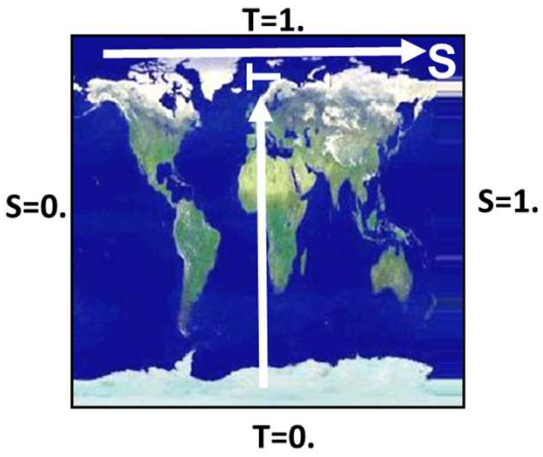
A texture image is not indexed by its actual resolution coordinates. Instead, it is indexed by a coordinate system that is resolution-independent. The left side is always **S=0**., the right side is **S=1**., the bottom is **T=0**., and the top is **T=1.**

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.



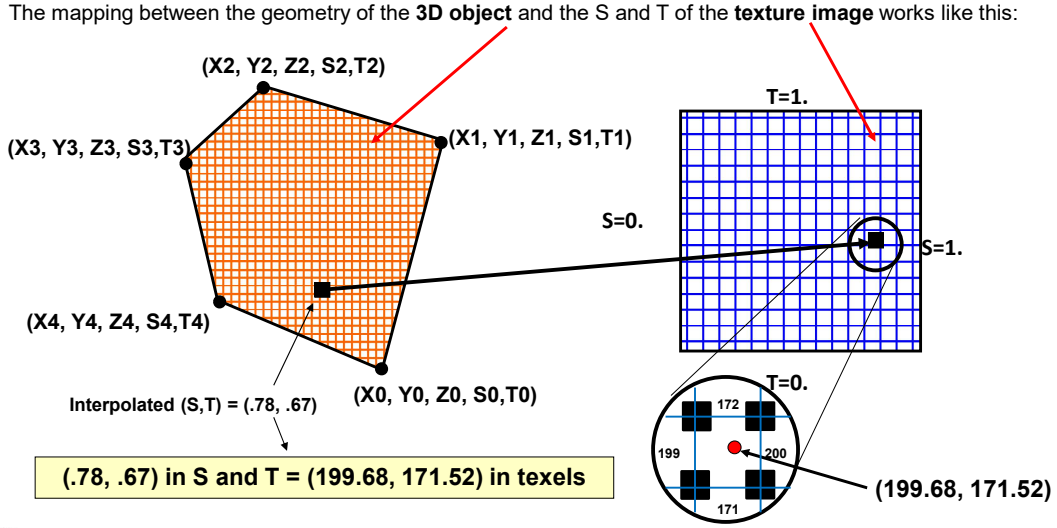
The Basic Ideas

Texture mapping is a computer graphics operation 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.



The Basic Ideas

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



Oregon State University
Computer Graphics

You specify an (s,t) pair at each vertex, along with the vertex coordinate. At the same time that OpenGL is interpolating the coordinates, colors, etc. inside the polygon, it is also interpolating the (s,t) coordinates. Then, when OpenGL goes to draw each pixel, it uses that pixel's interpolated (s,t) to lookup a color in the texture image.

Using a Texture: Assign an (s,t) to each vertex

Enable texture mapping:
glEnable(GL_TEXTURE_2D);

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

```
glBegin( GL_TRIANGLES );
        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***.)

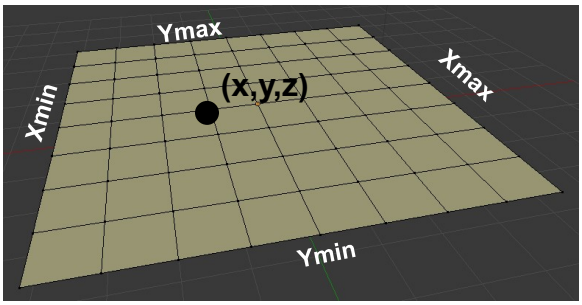
Disable texture mapping:
glDisable(GL_TEXTURE_2D);

Oregon State University
Computer Graphics

Using a Texture: How do you know what (s,t) to assign to each vertex?

glTexCoord2f(s0, t0);

The easiest way to figure out what s and t are at a particular vertex is to figure out what fraction across the object the vertex is living at. For a plane, this is pretty easy:

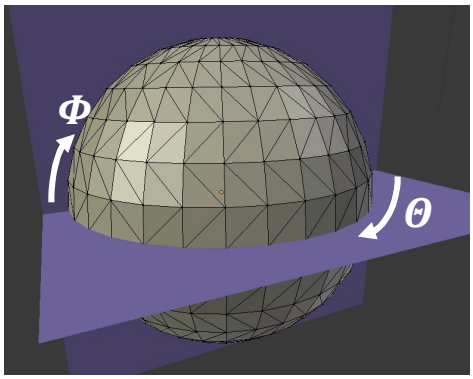


$$s = \frac{x - Xmin}{Xmax - Xmin} \qquad t = \frac{y - Ymin}{Ymax - Ymin}$$

Using a Texture: How do you know what (s,t) to assign to each vertex?

glTexCoord2f(s0, t0);

Or, for a sphere, you do the same thing you did for the plane, only the interpolated variables are angular (spherical) coordinates instead of linear coordinates



$$s = \frac{\Theta - (-\pi)}{2\pi} \qquad t = \frac{\Phi - (-\frac{\pi}{2})}{\pi}$$

The OsuSphere code does it like this:

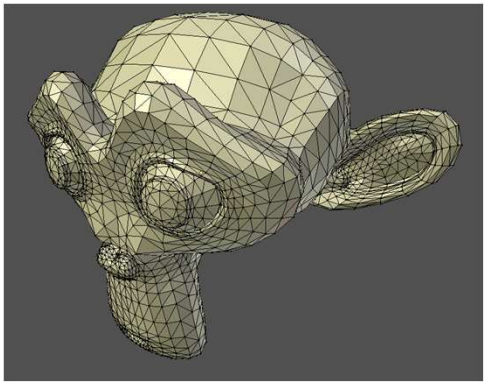
s = (lng + M_PI) / (2.*M_PI);

t = (lat + M_PI/2.) / M_PI;

Using a Texture: How do you know what (s,t) to assign to each vertex?

glTexCoord2f(s0, t0);

Uh-oh. Now what? Here's where it gets tougher…,



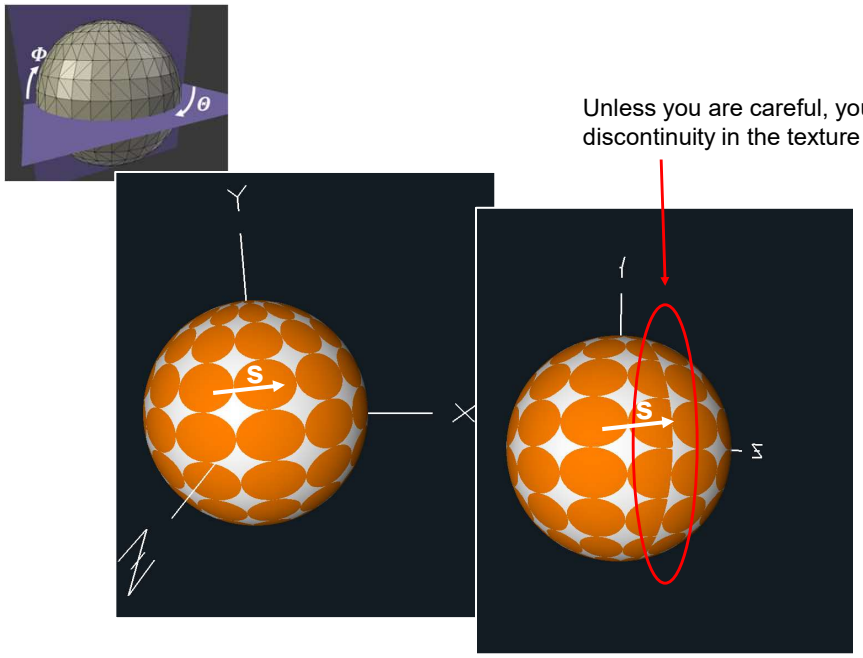
$s = ?$ $t = ?$

You really are at the mercy of whoever did the modeling and assigned the s,t coordinates…

Natural

Not-so natural

Be careful where *s* abruptly transitions from 1. back to 0.



Oregon State University
Computer Graphics

Reading a Texture from a BMP File


```
unsigned char *BmpToTexture( char *, int *, int * );
•••
int width, height;
unsigned char *texture = BmpToTexture( "filename.bmp", &width, &height );
```

This function is found in your sample code.

Note: BmpToTexture should be called *once*, and must be used at the end of *InitGraphics( )*.

Do not call BmpToTexture from the Display( ) function.
Do not call BmpToTexture from the Display( ) function.
Do not call BmpToTexture from the Display( ) function.
Do not call BmpToTexture from the Display( ) function.

Oregon State University
Computer Graphics

Texture Wrapping

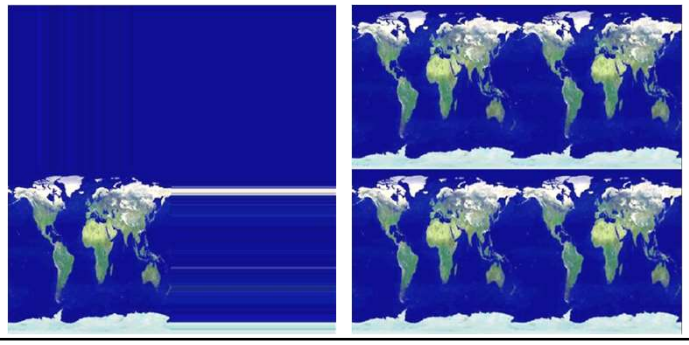
Define the texture wrapping parameters. This will control what happens when a texture coordinate is greater than 1.0 or less than 0.0:

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.



Oregon State University
Computer Graphics

Texture Filtering

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.

GL_NEAREST

GL_LINEAR

Oregon State University
Computer Graphics

Texture Environment

This tells OpenGL what to do with the texel colors when it gets them:

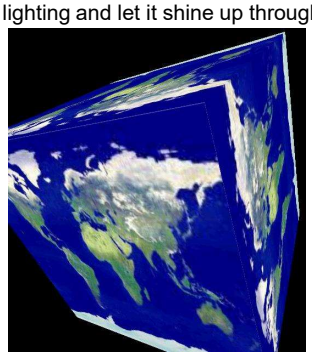
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode);

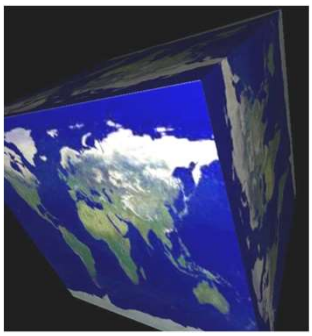
There are several *modes* that can be used. Two of the most useful are:

GL_REPLACE specifies that the 3-component texture will be applied as an opaque image on top of the polygon, replacing the polygon's specified color.

GL_MODULATE specifies that the 3-component texture will be applied as piece of colored plastic on top of the polygon. The polygon's specified 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.

GL_REPLACE





GL_MODULATE

Setting up the Texture in InitGraphics( )


```
int width, height;
unsigned char *texture = BmpToTexture( "filename.bmp", &width, &height );
int level=0, ncomps=3, border=0;
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

ncomps number of components in this texture: 3 if using RGB, 4 if using RGBA. Use 3

width width of this texture map, in texels.

height height of this texture map, in texels.

border width of the texture border, in texels. Use 0

texture the name of an array of unsigned characters holding the texel colors.

This function physically **transfers** the array of texels from the CPU to the GPU and makes it the currently-active texture. 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 make each texture current in **Display( )** right before you need it. See the upcoming section about ***binding*** texture objects.

Texture Transformation

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 functions 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*. Transforming the texture image forward is the same as transforming the texture coordinates backwards:

Scale = 0.5 | Scale = 1. | Scale = 2.

Texture Transformation

The only trick to this is to remember that you are transforming the texture coordinates, not the texture image. Transforming the texture image forward is the same as transforming the texture coordinates backwards:

Angle = -45. | Angle = 0. | Angle = 45.

Texture Objects

The OpenGL **glTexImage2D** function doesn't just use that texture, it ***downloads*** all those bytes from the CPU to the GPU, *every time that call is made*! After the download, this texture becomes the "current texture image".

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

If your scene has only one texture, this is easy to manage. Just do it once and forget about it.

But, if you have several textures, all to be used at different times on different objects, it will be important to maximize the efficiency of how you create, store, and manage those textures. In this case you should ***bind*** **texture objects**.

Texture objects leave your textures on the graphics card and then re-uses them, which is always going to be faster than re-loading them. Re-binding a texture object is basically "throwing a switch" in the GPU.

Texture Objects

Create a texture object by generating a texture name and then binding the texture object to the texture data and texture properties. The first time you execute **glBindTexture( ),** you fill the texture object. Subsequent times you do this, you are making that texture object current. So, create *global* Texture IDs like this:

```
int         Tex0, Tex1;                     // global variables
. . .
```

Then, at the end of **InitGraphics( )** you add:

```
int width0, height0, width1, height1;
unsigned char * textureArray0 = BmpToTexture( "image0.bmp", &width0, &height0 );
unsigned char * textureArray1 = BmpToTexture( "image1.bmp", &width1, &height1 );
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );

glGenTextures( 1, &Tex0 );                      // assign binding "handles" to texture objects
glGenTextures( 1, &Tex1 );
. . .
glBindTexture( GL_TEXTURE_2D, Tex0 );  // make Tex0 the current texture and store its parameters

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_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,   GL_LINEAR);
glTexImage2D( GL_TEXTURE_2D, 0, 3, width0, height0, 0, GL_RGB, GL_UNSIGNED_BYTE, textureArray0 );
```

Texture Objects


```
glBindTexture( GL_TEXTURE_2D, Tex1 );  // make Tex1 the current texture and store its parameters

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);
glTexImage2D( GL_TEXTURE_2D, 0, 3, width1, height1, 0, GL_RGB,GL_UNSIGNED_BYTE, textureArray1 );
```

Then, in **Display( ):**

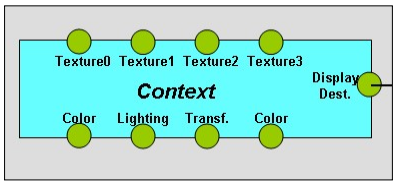
```
glEnable( GL_TEXTURE_2D );

glBindTexture( GL_TEXTURE_2D, Tex0 );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );
glCallList( DL0 );


glBindTexture( GL_TEXTURE_2D, Tex1 );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
glCallList( DL1 );

glDisable( GL_TEXTURE_2D );
```


What Does "Binding" Really Mean?

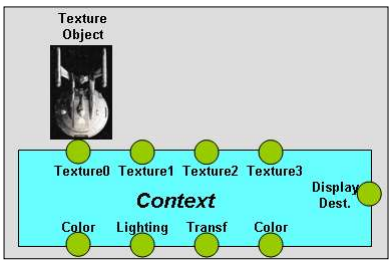
The OpenGL Rendering Context contains all the characteristic information necessary to produce an image from geometry. This includes the current transformations, colors, lighting, textures, where to send the display, etc.

The OpenGL term "binding" refers to "attaching" or "docking" (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will "flow" through the Context into the object.

Before Binding



After Binding



Some Great Uses for Texture Mapping you have seen in the Movies

Disney

Disney

Disney

Yes, I know, I know, these are older examples, but I especially like them because, at the time, the CG (and the textures) became part of the story-telling for the first time.

Some Great Uses for Texture Mapping you have seen in the Movies More Recently

Disney

Pixar

Disney

Pixar

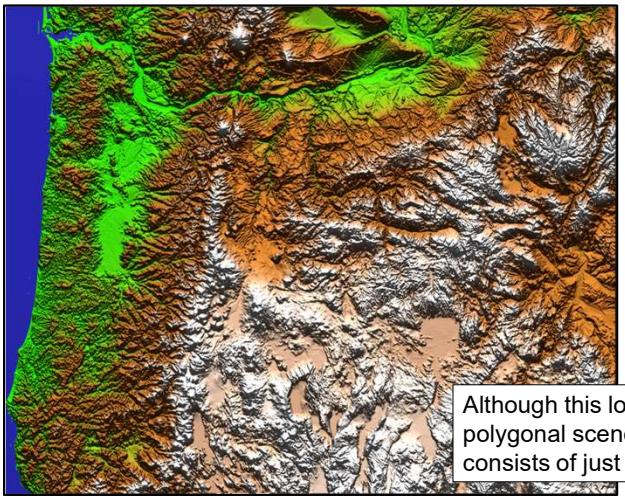
Disney

Disney

Bonus Topic: Procedural Texture Mapping

You can also create a texture from data on-the-fly. In this case, the fragment shader takes a grid of heights and uses cross-products to produce surface normal vectors for lighting.

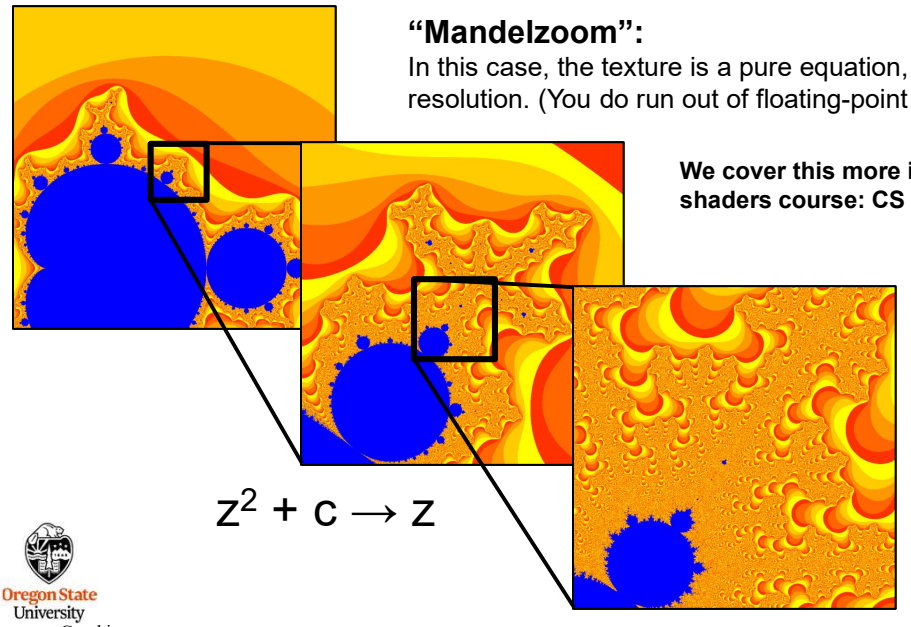
While this is "procedural", the amount of height data is finite, so you can still run out of resolution



We cover this more in the shaders course: CS 457/557

Although this looks like an incredible amount of polygonal scene detail, the geometry for this scene consists of just a ***single*** **quadrilateral**

Bonus Topic: Procedural Texture Mapping



"Mandelzoom":

In this case, the texture is a pure equation, so you never run out of resolution. (You do run out of floating-point precision, however.)

We cover this more in the shaders course: CS 457/557