Lecture on Texture Mapping

Texture mapping is one of the major innovations in CG in the 1990s. It allows us to add a lot of surface detail without adding a lot of geometric primitives (lines, vertices, faces). Think of how interesting Caroline's "loadedDemo" is with all the texture-mapping. Figure 1 contrasts the two.

1 Reading

As with everything in computer graphics, most of what I know about texture mapping I learned from Angel's book, so check that chapter first. Unfortunately, it's one of his weakest chapters, because it doesn't do a very good job of connecting the theory with the OpenGL code. A more practical presentation is a chapter of the Red Book (the Red OpenGL Programming Guide). You're encouraged to look at both.

2 Conceptual View

Texture mapping paints a picture onto a polygon. Although the name is *texture*-mapping, the general approach simply takes an array of pixels and paints them onto the surface. An array of pixels is just a picture. It might be something your program computes and uses. More likely, it will be something that you load from a file.

Demos: These all live in the `~cs307/public_html/pytw/demos/texture-mapping/` directory.

- `SimplestTextures.py`: This uses two extremely simple textures, one black and white, and one RGB, textured onto either a quad or the teapot.
- `USFlag.py`: This actually shows three flags: a checkered flag, a flag of increasing luminance values and a US flag. You can switch among the flags using the "u" key. You can change the size of the grayscale flags using the "." and "," keys. This is about the simplest texture-mapping code. Please look at it.
- `QuadPPM.py`: TW makes it easy to read a texture from a PPM file and texture-map it onto something. The code for this demo is relatively simple and is worth reading. This demo uses a PPM file of the US flag texture as the default, but you can specify your own on the command line. There's a directory of image files in

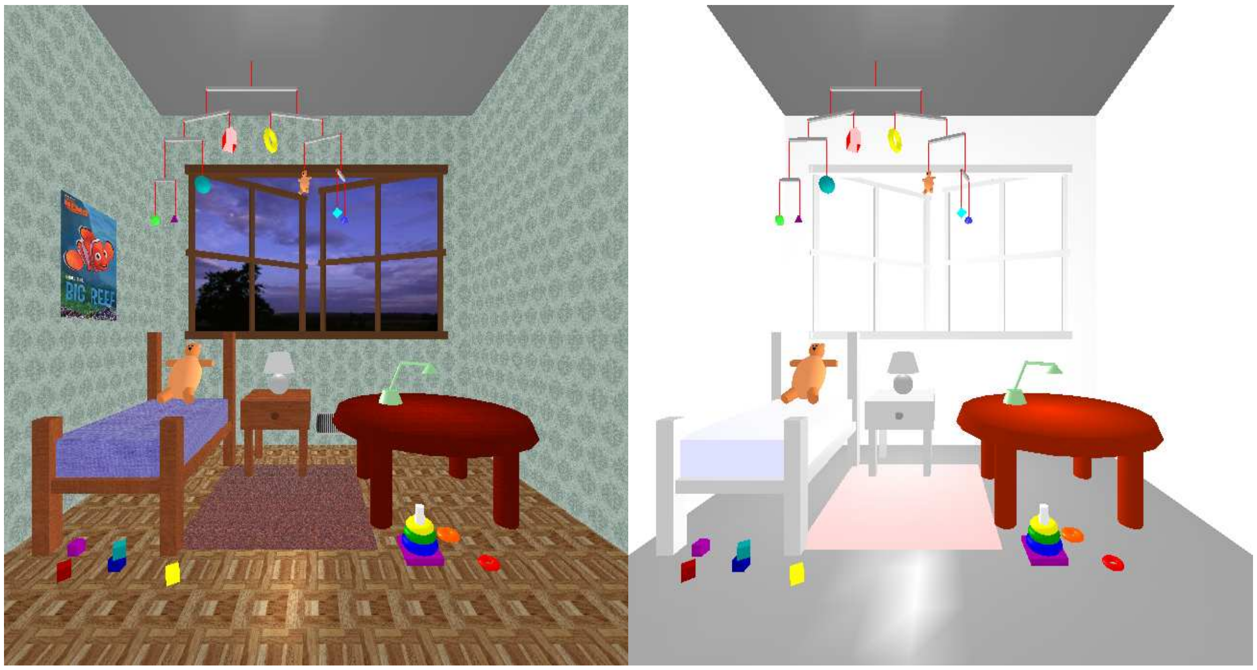

Figure 1: Caroline Geiersbach's loadedDemo with and without textures

```
~cs307/public_html/pytw/textures
```

Conceptually, to use textures, you must do the following:

- define a texture (a rectangular array of pixels — *texels*)
- specify a pair of texture coordinates (s, t) for each vertex of your polygon

The graphics system "paints" the texture onto the polygon.

3 How It Works

Texture mapping is a *raster* operation, unlike any of the other things we've looked at. Nevertheless, we apply textures to 2D surfaces in our 3D model, and the graphics system has to figure out how to modify the *pixels* during *rasterizing* (AKA *scan conversion*).

Since texture-mapping happens as part of the rasterizing process, let's start there.

3.1 Rasterizing

When the graphics card renders a polygon, it (conceptually)

- determines the pixel coordinates of each corner.
- determines the edge pixels of the polygon, using a line-drawing program (an imporant one is Bresenham's algorithm, which we won't have time to study).
- determines the color of the edge pixels on a single row (by linear interpolation from the vertex colors)
- walks down the row coloring each pixel (by linear interpolation from the two edge pixels).

Note: standard terminology is that the polygon is called a *fragment* (since it might be a fragment of a Bézier surface or some such). Thus, the graphics card applies a texture to a fragment.

This all happens in either in the *framebuffer* or an array just like it.

3.2 Texture Mapping

To do texture mapping, the graphics card must

- compute a texture coordinate for each pixel during the rasterizing process, using bi-linear interpolation
- look up the texture coordinates in the array of texels (either using the nearest or a linear interpolation of the four nearest)
- Either
  - Use the color of the texture as the color of the pixel, or
  - Combine the color of the texture with the color of the pixel

3.3 Other Innovations

The idea is to *map* a buffer of information onto a region of the framebuffer, thereby affecting the pixels.

- texture mapping: combine colors from the texture map with colors from the scene.
- bump mapping: perturb the surface normals from the scene when computing the light reflection
- environment (reflection) mapping: combine the colors from the material with a reflection of the scene; gives a result like ray tracing, but much faster.

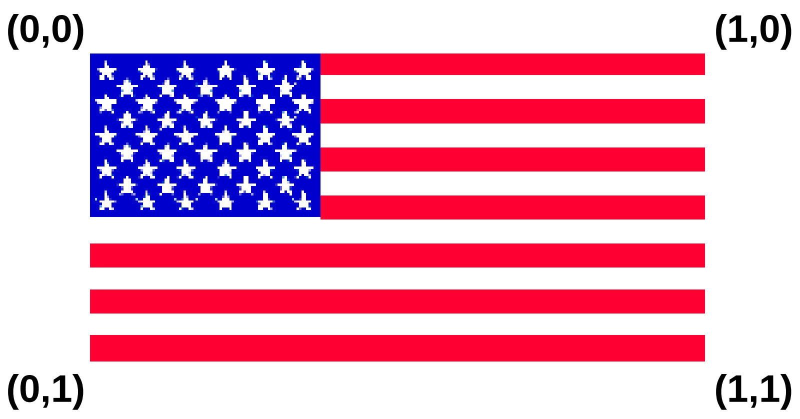

Figure 2: Texture Coordinates

3.4 Texture Space

We can have 1D or 2D textures. As with Bézier curves, the texture parameters will be in the range [0,1] in each dimension. Note that if your texture array isn't square and your polygon isn't square, you may have to deal with changes in *aspect ratio*.

Your texture is *always* an array and therefore is always a rectangle. Mapping a texture to rectangles (as OpenGL objects) is fairly easy; mapping it to other shapes is likely to cause distortion. We'll need to be careful in those cases.

Associate each vertex of our polygon with a texture parameter, just like we associate it with a normal, a color, and so forth. That means, as usual, that we must *precede* the vertex with the texture parameter.

For example, here is how we texture-map the entire texture onto a quad:

```
glBegin(GL_QUADS);
glTexCoord2f(0,1); glVertex3f( -1,-1,0);
glTexCoord2f(1,1); glVertex3f(  1,-1,0);
glTexCoord2f(1,0); glVertex3f(  1, 1,0);
glTexCoord2f(0,0); glVertex3f( -1, 1,0);
glEnd();
```

Notice that each vertex is *preceded* by its texture coordinates. That's because, like many other things in OpenGL, the attributes of a vertex precede it down the pipeline. (Remember, we did this with RGB colors as well.)

How do the texture coordinates relate to the 2D array of texels? This is easiest to explain with a picture such as figure 2.

- As you'd expect, the first element of the texel array, that is, element `[0][0]` is the same as texture coordinates (0,0).
- As we go down the first row of the array, until we get to element `[0][RowLength]`, we get to texture coordinates (1,0). This may seem odd, but it's true.
- As we go down the first column of the array, until we get to element `[ColLength][0]`, we get to texture coordinates (0,1). Again, this may seem odd, but it's true.
- Unsurprisingly, the last element of the texel array is the corner opposite the first element, so array element `[ColLength][RowLength]` corresponds to texture coordinates (1,1).

Conventionally, the texture coordinates are called (s, t), just as spatial coordinates are called (x, y, z). Thus, we can say that s goes along the *rows* of the texture (along the "fly" of the flag). The t coordinate goes along the *columns* of the texture (along the "hoist" of the flag).

Although you will *often* use the entire texture, so that all your texture coordinates are 0 or 1, that is not *necessary*. In fact, because the dimensions of texture arrays are required to be powers of two, the actual image that you want is often only a portion of the whole array.

The computed US flag array has that property. The array is 256 pixels wide by 128 pixels high, but the flag itself is 198 pixels wide by 104 pixels high. Thus, the maximum texture coordinates are:

$$\begin{aligned} \text{fly} &= 198/256 = 0.7734 \\ \text{hoist} &= 104/128 = 0.8125 \end{aligned}$$

Of course, we also need to ensure that the rectangle we are putting the flag on has the same aspect ratio as the US flag, namely: 1.9. See `http://cs.wellesley.edu/~cs307/flagspec.htm`.

The texture parameters can also be *greater* than 1, in which case, if we use `GL_REPEAT`, we can get *repetitions* of the texture. If s is some parameter where $0 < s < 1$, specifying some part of the texture partway along, then $1 + s$, $2 + s$ and so on are the same location in the texture.

3.5 Basic Demos

Please look at the code for the following demos. All of them are in `~cs307/public_html/demos/texture-mapping/`.

- `SimplestTextures.py` This is a simple example, using very small (4×4) textures. There are actually two textures: use "u" to switch.

3.6 Texture Mapping in OpenGL

Conceptually, to actually do texture mapping in OpenGL, you have to do all the following steps.

1. Create or load a 1D or 2D array of texels. All dimensions must be a power of two! Different kinds of data are possible:
   - RGB values
   - RGBA values
   - Luminance (grayscale)
   - ...

   Also, the data in the array can be in different formats (unsigned bytes, short floats, etc.). You must tell OpenGL what it is.
2. Set various modes. These have default values (see the man pages), so they can be skipped in some cases, but I tend to set them all. I copy/paste the code from some working example of texture-mapping, then change the modes as necessary.
3. Send the texel data to the graphics card.
4. Enable texture-mapping
5. Specify a texture coordinate for each vertex. Sometimes this is done automatically (as for the teapot) or is calculated (as for Bézier surfaces). We'll get into Bézier stuff later.

For coding, that means the following steps. We'll go through these functions in detail.

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height, 0,
             GL_RGB, GL_UNSIGNED_BYTE, texelArray);
```

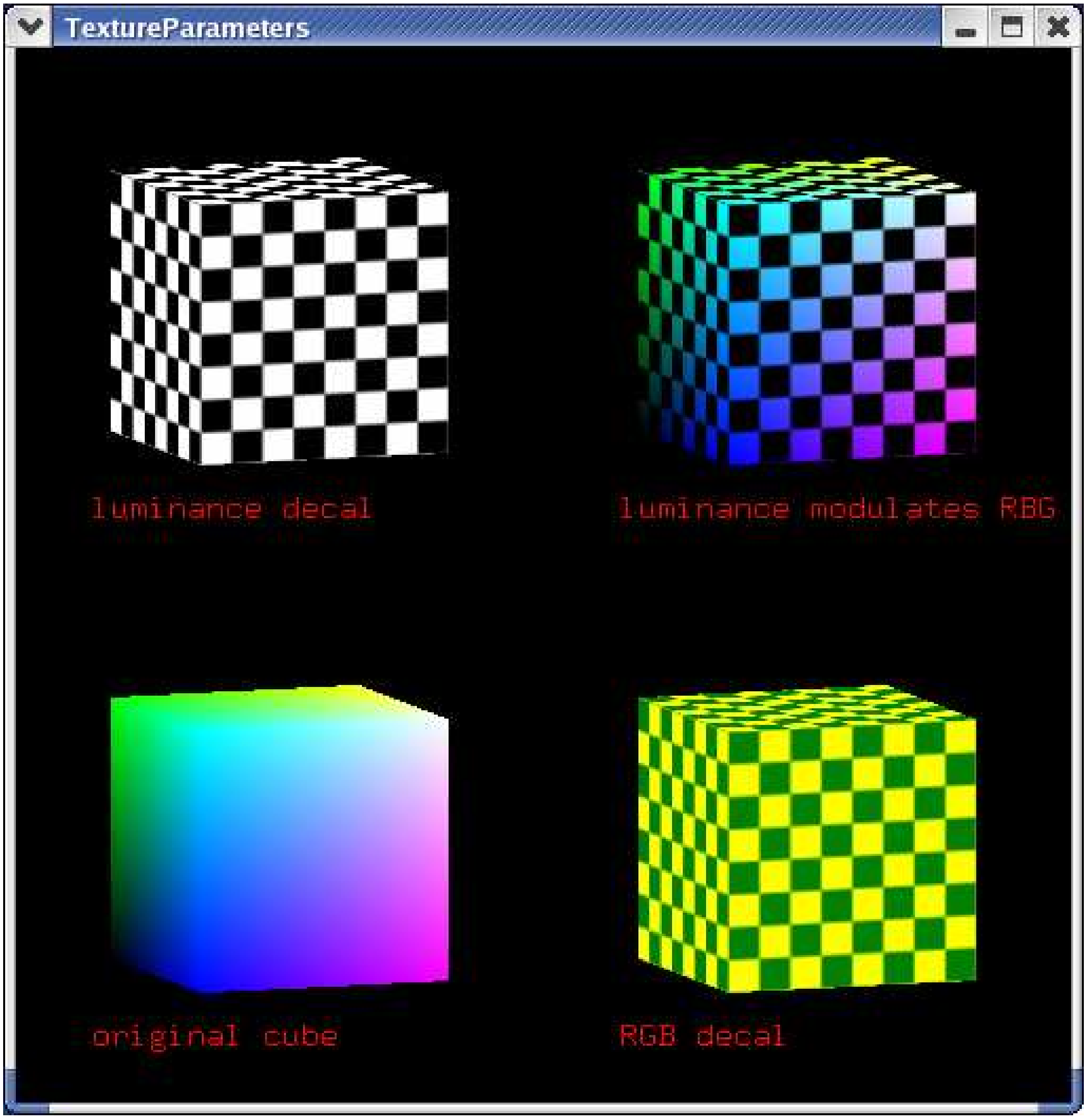

Figure 3: Demo of how textures can interact with fragment color: TextureParameters

```
glEnable(GL_TEXTURE_2D);

glBegin(...);
glTexCoord2f(0,1);
glVertex3f(0,0,0);
```

The `glTexEnvf` has several settings. The settings can mean different things depending on the format of the texture (such as luminance versus RGB) and the color model in OpenGL. Ignoring those nuances, here's a basic summary.

- `GL_DECAL` and `GL_REPLACE`. These do the same thing except when there's transparency. The intrinsic color of the fragment (the polygon in the model) is ignored and the color of the texel is used instead.
- `GL_MODULATE`. The color of the pixel is the *product* (plain old multiplication) of the color of the fragment and the color of the texel. The texel is often a luminance value and you use the texture to darken or lighten the color of the fragment.
- `GL_BLEND`. The color of the pixel is a mixture (weighted average) of the color of the fragment and the texture "environment" color, with the texel determining the weighting.

The texture can either *replace* the scene colors, like a decal, or it can blend with the scene colors, as with wood-grain finishes, or even adding surface smudges and dirt, to make things look more realistic. The parameters settings in `glTexEnvf` help to set this interaction between the color of the fragment (whether direct RGB color or material and lighting) and the texture.

We'll look at the `TextureParameters.py` demo and code (I'll port that to Python from C before class). You can see a screen-shot in figure 3. We'll also look at `~cs307/pub/tw/Tutors/texture`.

We'll talk about the `glTexParameteri` function calls later.

The `glTexImage2D` function has a lot of parameters. Most are fixed, though:

1. `GL_TEXTURE_2D` or `GL_TEXTURE_1D`. Those are the values we'll use
2. level. It's possible to give OpenGL several images at different resolutions, called "mipmaps," but it seems to be flakey. The OpenGL examples I've downloaded don't work. So, always use the base level, which is 0.
3. internal format: specifies the number of color components in the texture. Typically, this is 3, meaning RGB data.
4. width of the texture. Must be a power of 2 or one more than a power of 2 if you have a one-pixel border.
5. height. Same as width
6. border. zero or one.
7. format. The kind of pixel data. Typically `GL_RGB` or `GL_LUMINANCE`.
8. type. The datatype of the array. Typically `GL_UNSIGNED_BYTE`.
9. pixels. A pointer to the image data in memory.

We'll look at the `USflag.cc` file for an example of this.

4 Issues

Here are some issues to face and choices to make:

- **Aspect Ratio:** Your texture is always a rectangle. Even if your polygon is one, too, you'll have to deal with matching aspect ratios if you want the image to be undistorted. With a plain texture (such as grass or wood) this may not matter, but for pictures it may.

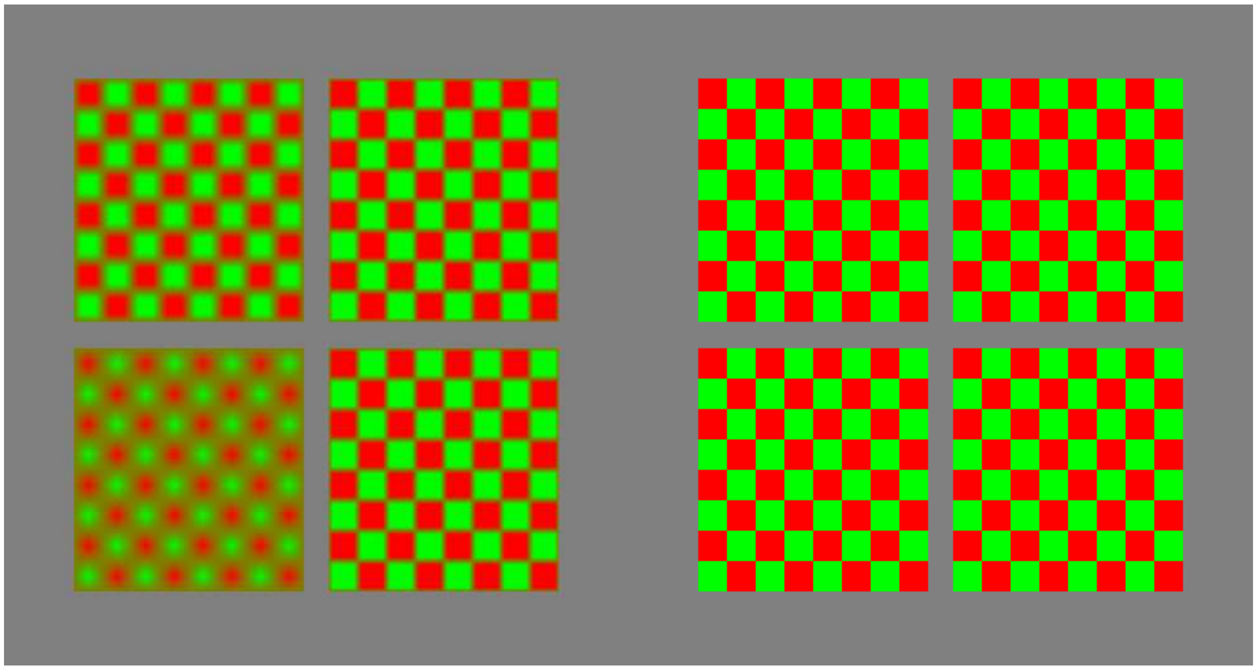

Figure 4: Both figures are checkerboard textures, stretched over a large number of pixels. Consequently, the texture coordinate values for many pixels fall “between” texel values. In the picture on the left, we use a “linear” interpolation between the texel values. In the picture on the right, we use the “nearest” texel value.

- **Wrapping:** What happens when your texture parameters fall outside the [0,1] range? We’ll try this with the tutor.
  - You can “wrap” around (essentially removing the integer part and using only the fractional part). This repeats the texture. For real textures, you often want to do this.
  - You can “clamp” the value at the edge pixel. If your texture has a border of some sort, that can work out well.
- **Filter:** What to do when the pixel doesn’t exactly match a texel? You get to specify this for both magnification (pixel smaller than texel) and minification (pixel larger than texel), but in practice, I think they are usually set to the same value.
  - Use the nearest (Manhattan distance) texel to the center of the pixel.
  - Use a weighted average of the four texels nearest to the center of the pixel.

  We’ll look at the `LinearNearest` demo to understand this.

  Note: the functions to set the filters appear not to have adequate defaults: if you don’t set them, you won’t get a texture!
- **density** of the texture repetition. Too little and it looks badly “stretched.” Too much can squeeze the texture too much. Look at `Grass.cc`. Try the three different textures. Use the “r” callback to reveal the vertices that are created. Look at the texture “from above,” by using the “Y” callback.

4.1 More Demos

Please look at the code for the following demos. All of them are in `~cs307/public_html/pytw/demos/texture-mapping/`.

- `Rainbow.py` This is lovely example of a 1D texture. Use the "R" keyboard callback to turn the rainbow on/off. The illusion is much better if you switch to "immerse" mode. Note that another version of this demo that doesn't use TW, called `RainbowSweet`, looks better because the illusion is much better if you're inside the scene. Or use TW's "immerse" mode. Original code from Michael Sweet.
- `TextureParameters.cc` This demonstrates uses of texture parameters, such as decal vs blending. It generates figure 3.
- `LinearNearest.cc` This demonstrates the difference between `LINEAR` and `NEAREST` for magnification/minification. You can see screen shots in figure 4.
- `LitUSFlag.cc` This demonstrates how to combine Bézier surfaces and texture mapping. Lighting, too.
- `~/pub/tw/Tutors/texture`. Nate Robins' tutor. Pretty slick, but he uses textures that are upside down, so it can be confusing, too.

5 Images and File Formats

Images come in dozens of formats, with different kinds of compression techniques and so forth. We will look at the following kinds:

- Compressed Formats. These are supported by all reasonable web browsers, and the file sizes are not excessive. The different formats have tradeoffs, though, and are complex, because of the compression algorithms. There are common libraries to read/write these.
  - GIF (Graphic Interchange Format): a compressed (loss-less) format limited to 256 colors. It was encumbered with a patent, but that has now expired. Allows index transparency, meaning chosen pixels can be in the "clear" color instead of a normal RGB color.
  - JPG (Joint Photographic Experts Group): a compressed (lossy) format that can handle full RGB color (millions of different colors in an image). No transparency. Tends to be best for pictures of realistic natural scenes.
  - PNG (Portable Network Graphics): an open-source, compressed (loss-less) format that removes some restrictions of GIF. The file format also stores "vector" information, if the image is produced by a drawing program. Fireworks uses this format as its native format.
- Uncompressed formats. These formats have a simple structure but the file sizes are large because there is no compression: it's just a 2D array of RGB values. These files are typically not supported by web browsers and shouldn't be used on the web anyhow because the file sizes are so large.
  - BMP (MS-Windows Bitmap format): this is an uncompressed Windows format.
  - TIFF (Tag Image File Format): an industry standard pixmap file format, common on Macs. Some digital cameras produce this.
  - PPM (Portable Pixmap): an open-source, uncompressed format. The format is:
    - P6: two ASCII characters identifying the file type
    - w, h: two decimal numbers with a space after them giving the width and height of the image
    - 255: the largest possible value of a color component
    - a carriage return character (ASCII 13)
    - data: $w \times h \times 3$ bytes giving the R, G, B values for each pixel.

    It's standard to store the image in top-to-bottom, left-to-right order. I got this info from
    `http://astronomy.swin.edu.au/~pbourke/dataformats/ppm/`

For TW, we will always use PPM format. You can convert images to/from PPM format to other formats using Windows, Mac or Linux graphics programs or various Linux commands, such as:

- ppmtogif
- ppmtojpeg
- bmptoppm
- *topnm
- pnmto*

PNM is a “portable anymap” file; the programs seem to be able to guess whether it’s black and white (PNB), grayscale (PGM) or color (PPM).

5.1 Demo

- Start Fireworks
- Draw something
- Save (default is PNG, so that’s fine)
- copy it to Puma, say with Fetch or WinSCP.
- convert to PPM:

  ```
  % display foo.png
  % pngtopnm -verbose foo.png > foo.ppm
  % display foo.ppm
  ```

- Run QuadPPM.py foo.ppm

5.2 Loading Images

We’ll explore the code of `QuadPPM.py`.

- You can read in an image from a file and use it as a texture.
- You should read the file in just once, so don’t call `twTex2D` from your display function.
- Note that most glut objects don’t have pre-defined texture coordinates. Only the teapot does. You can generate them for the others, using a fairly incomprehensible interface. We’ll try to learn more about this as the semester goes on.

6 Binding Textures

For additional speed when using several textures, you can load all the textures, associating each with an integer identifier (just like display lists) and then referring to them later.

Setup steps:

- Ask for a bunch of identifier numbers:

  ```
  glGenTextures(num_wanted,result_array);
  ```

- Then, for each texture you want, get one of the numbers out of the array and:

```
glBindTexture(GL_TEXTURE_2D, textureNumber);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, something);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, something);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, something);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, something);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, something);
glPixelStorei(GL_UNPACK_ALIGNMENT,1);
glTex2D(...);
   or
twTex2D(...);
```

Reference step: When you want a texture, just:

```
glBindTexture(GL_TEXTURE_2D, textureNumber);
```

As a convenience, you can replace each of the texture steps with

```
twLoadTexture(textureIDs[n], filename);
```

However, this function uses `GL_MODULATE`, `GL_REPEAT` and `GL_LINEAR`, which may not be what you want.

Demos: `TextureBinding.cc` and `USflag-binding.cc`. Try spinning either of these. Notice how relatively quick they are. This is because the texture is already loaded into memory on the graphics card, so almost nothing needs to be sent down the pipeline to draw the next frame of animation.

6.1 Saving Images

You can also save the contents of the framebuffer as a PPM file. Just hit the “S” key. This is accomplished thanks to an interesting function

```
void glReadPixels( GLint x,     // raster location of first pixel
                   GLint y,
                   GLsizei width,  // dimensions of pixel rectangle
                   GLsizei height,
                   GLenum format,  // GL_RGB
                   GLenum type,    // GL_UNSIGNED_BYTE
                   GLvoid *pixels );
glReadPixels(0,y,width,1,GL_RGB,GL_UNSIGNED_BYTE, (GLvoid *) pixels);
```

The file is saved as `saved_image01.ppm` in the current directory. If you hit “s” again, you get `saved_image02.ppm` and so forth. In honor of family and friends weekend, convert these to PNG and put them on your web page! Or email them!

Note that PPM files are big. In many of our examples, the framebuffer is 500 by 500. The file size is therefore

$$500 \times 500 \times 3 + \text{len(P6500 500 255)} + 1 = 750014$$

```
% ppmtojpeg -v saved-frame01.ppm > saved-frame01.jpg
ppmtojpeg: Input file has format P6.
It has 500 rows of 500 columns of pixels with max sample value of 255.
ppmtojpeg: No scan script is being used
% ls -l saved-frame01.*
-rw-rw-r--    1 cs307    cs307       25290 Nov  7 00:06 saved-frame01.jpg
-rw-r--r--    1 cs307    cs307      750014 Oct 31 14:33 saved-frame01.ppm
```

The JPG is a bit smaller! It’s 1/30th the size in this case, but your mileage may vary.

Since you have a finite filespace quota, manage your space carefully. Once you save a frame, you might convert it to a compressed format (probably PNG or JPEG) and then discard the PPM file.

7 Texture Mapping using Modulate

When you texture-map using `GL_MODULATE`, you have to think about the color of the underlying surface. In particular, if you're using material and lighting, you have to use material *and* textures.

Caroline's texture tutor can help: `~cs307/public_html/demos/textureTutor`

8 Texture Mapping Onto Odd Shapes

8.1 Triangles

There are actually two choices here. If you want a triangular region of your texture, there's no problem, just use the texture coordinates as usual. If you want to squeeze one edge of the texture down to a point, it would seem that all you have to do is use the same texture coordinates for both vertices, but that yields odd results. Instead, you can use *linear* Bézier surfaces to make a triangular region.

Demo: `TexturemapTriangles.cc`

8.2 Cylinders

If mapping onto a curved surface, we usually represent the surface with parametric equations and map texture parameters to curve parameters. For example, a cylinder:

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

With the easy mapping:

$$\begin{aligned} s = u \\ t = v \end{aligned}$$

Demo: `CylinderFlag.cc` This shows how to put a 2D texture onto a non-planar figure. It uses the US flag, since it's easy to see the orientation of the texture. Essentially, we have to build the figure ourselves out of vertices, so that we can define texture coordinates for each vertex. There are two ways to put a flag onto a cylinder: with the stripes going around the cylinder or along its length. This demo does either; the "l" keyboard callback switches the orientation. Understanding this code is not easy, but it really only requires understanding polar/cylindrical coordinates. The texture coordinates are relatively straightforward.

8.3 Bezier Surfaces

We've already seen this, and we got another dose of it when we looked at mapping onto triangles, but let's look at it again.

To map onto a surface with material and lighting, consider:

Demo: `LitUSFlag.cc`

8.4 Globes

In general, mapping a flat surface onto a globe (sphere) is bound to produce odd distortions. It's essentially a 3D version of the problem of mapping a rectangle onto a circle.

The reverse mapping is interesting to contemplate: namely a flat rectangle that shows the surface of the globe. This is a problem that cartographers have wrestled with for years. Indeed, both of the examples I gave above for circles and squares have equivalents in cartography.

The distortion problem presents several tradeoffs, the most important of which is shape distortion versus area distortion.

- area: To preserve the equal-area property, you have to compress the lines of latitude, particularly those far from the equator. A famous current example is the **Peters** projection.
- shape: To preserve shape, you end up expanding the lines of latitude, particularly those far from the equator. One important side effect of preserving shape is that a straight line on the map is a great circle, which makes these maps better for navigation. A famous current example is the **Mercator** projection.

Let's spend a few minutes discussing the pros and cons of these. There are some good web pages linked from the course home page.

To texture-map a globe, I created a globe by hand, iterating from the north pole (π) to the south pole ($-\pi$) and from 0 longitude around to 2π longitude. I converted each (longitude,latitude) pair into (x,y,z) values but also made a (s,t) texture-map pair. This works pretty well except possibly at the poles.

Math:

$$\begin{aligned}
x &= \cos(\text{latitude}) * \cos(\text{longitude}) \\
y &= \sin(\text{latitude}) \\
z &= \cos(\text{latitude}) * \sin(\text{longitude}) \\
s &= 1 - \text{longitude}/2\pi \\
t &= 1 - \text{latitude}/\pi + \pi/2
\end{aligned}$$

Demo: `GlobeTexture.cc`