Overview

Basics of Textures

Using Textures

Repetition and Clamping

Filtering and Mip maps

Specifying texture maps

Animating and transforming textures

Lighting textures

Multitexturing

Framebuffer Objects and Textures

Texture

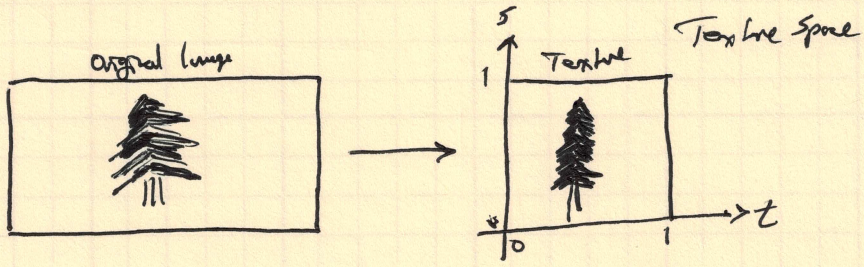
- **Basics** ≡ Texturing a surface is essentially painting a picture onto it
  - This provides 2 advantages
    1.) Realistic depiction of an object that happens to be painted in real life
    2.) Illusion of geometric detail: Rather than attempting to model an object we can simply use an image of it
  - Typically a texture is an image applied to a polygon or mesh.
  - The pixels in the texture are called texels, each storing color values
  - The picture can be from an external file or a procedural or synthetic texture.

- **Usage:**

```
glGenTextures(n, texture) ≡ returns n texture ids into the provided texture array. // setup

loadTextures() {
    imageFile *image[1];
    image[0] = getBMP("....")        ≡ Template for loading an image
}
...

glBindTexture(GL_TEXTURE_2D, texture[i]) ≡ binds the ~~current~~ texture id at i

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image[0]->width, image[0]->height,
             0, GL_RGBA, GL_UNSIGNED_BYTE, image[0]->data)
    ≡ specifies the texture image for the currently bound texture object

    glTexImage2D(target, level, internalFormat, width, height, border, externalFormat,
                 type, *pointer)
        ≡ specifies the texture image pointed to by pointer, which is of format
          externalFormat and of data type type; the kind of texture is target, the
          mipmap level is level, the size of the image is width x height, while internalFormat
          tells OpenGL how to store the image data. Border must be 0
}

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE)
glEnable(GL_TEXTURE_2D)   ≡ set texture environment vars        // setup

glBindTexture(GL_TEXTURE_2D, texture[id]) ≡ activates Texturing // draw scene
```

◦ Texture Space, Coordinates and Map

- A texture, once loaded, occupies a unit square (0,0), (1,0), (1,1), (0,1) of an imaginary (virtual) plane called texture space. Regardless of whether the original image is square.



- In the following example each statement pair maps the vertex of a polygon to a point in texture space.

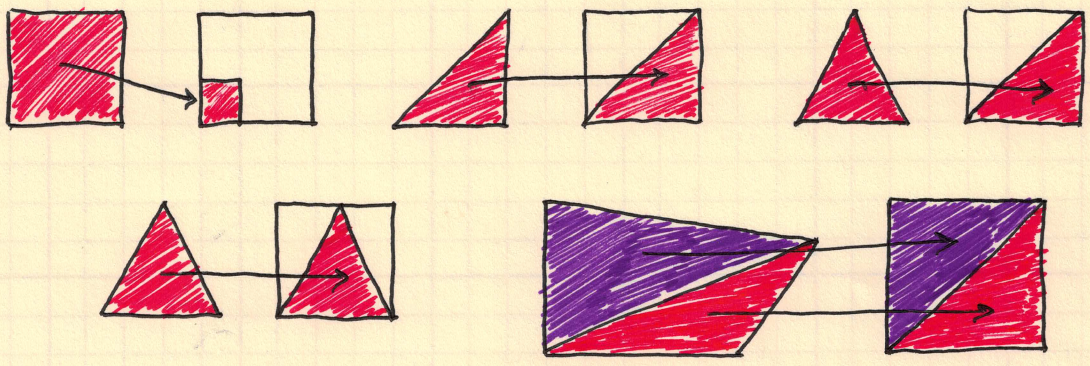
```
glBegin(GL_POLYGON);
    glTexCoord2f(0.0, 0.0); glVertex3f(-10.0, -10.0, 0.0)
    glTexCoord2f(1.0, 0.0); glVertex3f(10.0, -10.0, 0.0)
    glTexCoord2f(1.0, 1.0); glVertex3f(10.0, 10.0, 0.0)
    glTexCoord2f(0.0, 1.0); glVertex3f(-10.0, 10.0, 0.0)
glEnd(GL_POLYGON);
```

Thus mapping each vertex in world space to a point in texture space

- The coordinates of the mapped point in texture space are called the texture coordinates of the vertex

- The texture is painted onto the polygon by interpolating the mapping through the polygon and then applying the RGB value of the image to each point in the polygon.

- Example Mappings



◦ Repeating and Clamping

- Textures can be tiled across a ~~large~~ large area in both the s- and t-directions by specifying the wrapping mode to be repeated:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
```

- Another mode is clamped to the edge, where there is no repetition. This then stretches the texture across the surface. To set this mode use ~~GL_CLAMP~~ GL_CLAMP_TO_EDGE

- There are additional modes, which I leave you to explore.

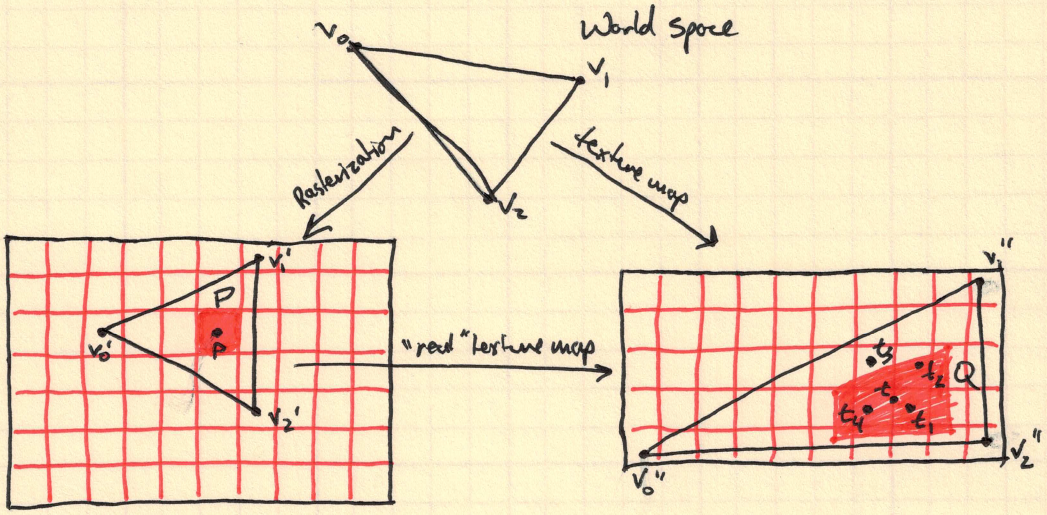
◦ Filtering

- Unfortunately texturing is not without its issues.

- Specifically visual artifacts such as aliasing can occur which can cause the apparent shimmer/flash/scintillate as the view changes.

  • This occurs due to the finite resolution of the display

  ◦ Because both raster pixels and texels are finite-size squares and not points we cannot exactly ~~map~~ map them one-to-one



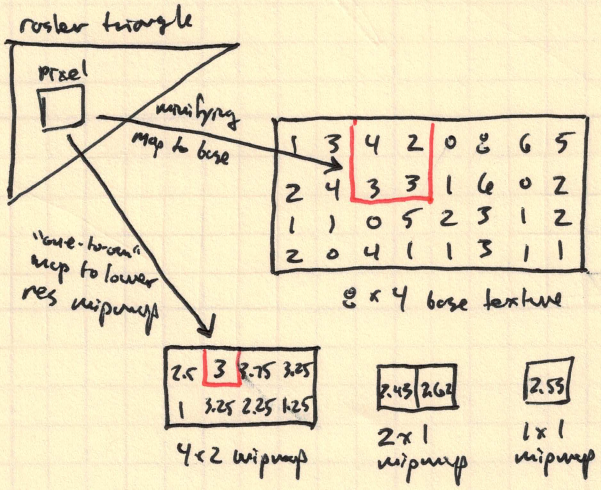
where Pixel P is mapped to quadrilateral Q containing many Texels

- How do we choose a color for P?

- This is handled by sampling the texel whose center is nearest to the center of the quadrilater, t_1.
- This is done by setting the minification and magnification filters to GL_NEAREST as follows.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
```

- Unfortunately if the rasterization changes, this can cause a small change in the mapping, leading to the shimmer, as we can see in the previous drawing small changes lead to selecting t_2, t_3, t_4 rather than t_1
- We can handle this by smoothing the color with the GL_LINEAR filter which averages the surrounding texels colors together
- Note there is a difference in minification and magnification here.
  - Minification: occurs when a pixel is mapped onto multiple texels, as a consequence, typically, of a textured surface moving into the distance to occupy a smaller part of the screen. Zoom-out
  - Magnification: inverse of minification, Zoom-in
- To handle minification, we can use a pre-assigned set of textures for different levels of minification. These are called mipmaps and are created from an initial base texture. This:
  - saves on run-time filtering
  - assures quality



- Generally, if a base texture of resolution $2^m \times 2^n$ is to be mipmapped then OpenGL requires mipmaps of resolution $2^{m-1} \times 2^{n-1}$, $2^{m-2} \times 2^{n-2}$, ... obtained by halving both width and height, until dimensions become 1.

- In addition to GL_NEAREST and GL_LINEAR there are 4 other filtering options. In order of increasing quality and computational cost:
  - GL_MIPMAP_NEAREST: applies the mipmap that's closest fit resolution-wise and then applies GL_NEAREST
  - GL_LINEAR_MIPMAP_NEAREST: applies mipmap that is closest fit, and then GL_LINEAR
  - GL_NEAREST_MIPMAP_LINEAR: Finds two closest mipmaps to rasterized primitive, then uses GL_NEAREST on both to produce two sets of color values, then takes a weighted average
  - GL_LINEAR_MIPMAP_LINEAR: Finds two closest mipmaps to rasterized primitive then uses GL_LINEAR on both and takes weighted average

- Note: Mipmaps are not used to handle magnification

- Generating mipmaps is done using:

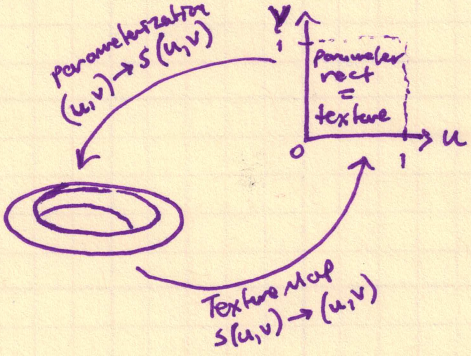
  glGenerateMipmap(target)
  → GL_TEXTURE_2D

Specifying the Texture Map

◦ Parameterized Surfaces

- With parameterized surfaces (such as a Torus [redacted]) we can let the parameterization drive the texture mapping
- In the example of a torus we have two parameters i, j which define the vertex v_{ij}. We can also use them to define a texture coordinate array by mapping i, j to $[0,1] \times [0,1]$ as follows.
  - As i goes from 0 to p, and
  - As j goes from 0 to q
  - We can map vertex (i,j) to a location in texture space (u,v), as follows

$$u = (i/p)$$
$$v = (j/q)$$



◦ Bézier and Quadric Surfaces

- OpenGL can automatically generate texture coordinates for Bézier and quadric surfaces
- The process is as follows:

1.) Create a Bezier surface s' in texture space

```
glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2, 0, 1, 4, 2, texturePoints[0][0])
```

- This is similar to the creation of a Bezier surface, but instead we are creating a surface in texture space, and the control points will be stored in texturePoints array
- The control [redacted] points stored are $(0,0), (0,1), (1,0), (1,1)$, which makes s' a rectangle $[0,1] \times [0,1]$ in texture space

2.) [redacted] Construct the same surface in world space

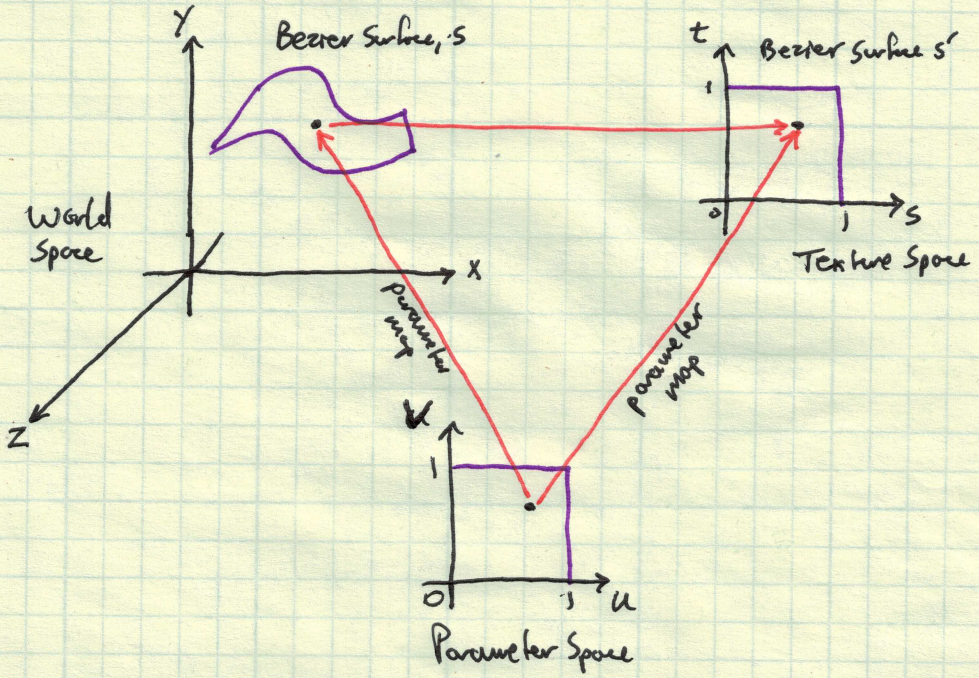
3.) We need to construct the texture map. OpenGL will do this by by translating each parameter point (u,v) on the world space surface s to the point (u,v) on s'

4.) Call the following 2 statements

```
glEnable(GL_MAP2_TEXTURE_COORD_2) - enable texture mapping
glMapGrid2f(...) - setup the grid in the parameter domain for surface s
```

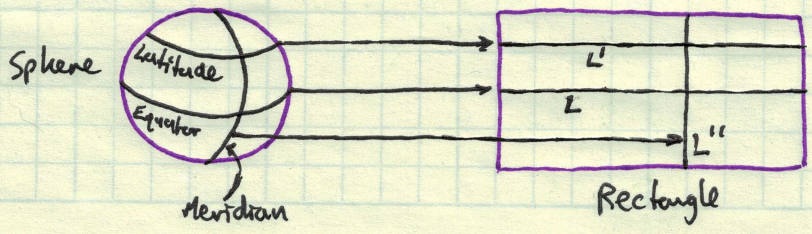
5.) Get OpenGL to generate texture coordinates

```
gluQuadricTexture(qobj, GL_TRUE) // in drawScene
                  ↳ pointer to object
```



Spheres and Mercator Projection

- Textured spheres are quite common, think of a globe, the question is how do we texture such objects?
- For this we use a Mercator Projection. Imagine a sphere and a rectangular sheet of paper, where the paper's height equals the diameter of the sphere.
- We then map each latitude (circle) of the sphere to the horizontal segment across the rectangle at the same altitude, scaling each latitude to fit its segment, such that the image of each meridian (longitudinal half-circle going the poles) is a vertical segment through the rectangle



- The Mercator projection obviously produces distortions which worsen as one moves away from the equator. Additionally:
  - Latitudinal circles can never map in a continuous one-to-one manner onto a line segment
  - The north and south poles are not circles, but points.
- Textures meant to be applied to a sphere are usually obtained from a Mercator projection of the coloring space.
- The mapping this defines works as follows:
  - We first note that we can define a generic point on a sphere's mesh as follows:

$$\begin{pmatrix} R\cos\left(\frac{-\pi}{2}+\frac{j}{q}\pi\right)\cos\left(\frac{2i}{p}\pi\right), \\ R\sin\left(\frac{-\pi}{2}+\frac{j}{q}\pi\right), \\ R\cos\left(\frac{-\pi}{2}+\frac{j}{q}\pi\right)\cos\left(\frac{2i}{p}\pi\right) \end{pmatrix}$$

  - From this, we can see that fixing a y-value on the sphere (equivalent to staying at a given latitude) implies fixing a j-value on the rectangular sample grid, which is equivalent to staying on a horizontal line of grid points.
  - Similarly, fixing an i-value is done by fixing on a meridian, and equivalent to staying on a vertical line in the sample grid.
  - Thus if we map a generic point above to a point (i, j) in the rectangular grid of size $p \times q$, we get a Mecator projection.
  - We can then map this to a unit square where $(i,j) \to (i/p, j/q)$, which will then place the Mercator projection into texture space.

Texture Matrix and Animating Textures

- Similar to the modelview matrix stack, there is the **texture matrix stack** which also contains **4x4 texture matrices**, the topmost being the **current texture matrix**

- Just as how the model view matrix works, texture coordinates are also transformed by multiplication from the left by the current texture matrix

- When transformations are applied, texture coordinates are written as a vector with the following form:

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix}$$

where if we have a 2D texture coordinate (s,t) this would be the vector:

$$\begin{bmatrix} s \\ t \\ 0 \\ 1 \end{bmatrix}$$

- As expected, the default texture matrix is the 4x4 identity.

- Texture Mode is entered with a call to:

```
glMatrixMode(GL_TEXTURE)
```

- The texture matrix stack is manipulated with the standard functions:

  - glPushMatrix()
  - glLoadIdentity()
  - glTranslatef()
  - ...

- Similar to how we can use matrix transformations to animate objects in world space, we can use these same transformations to animate textures.

Lighting Textures

- Lighting, color, and textures play well together within OpenGL
- In the initialization function an option is selected which sets up exactly how these will play together. Specifically this is done on the following call:

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, params)
```

- If params is GL_REPLACE, then the texture's color overwrites the current primitive pixel colors (thus the material colors and light source are ignored)
- On the other hand we can combine color, light, and texture by setting params to GL_MODULATE, which does the following:
  - Computes the RGB at the vertices and interpolates it through the interior based on the current lighting model
  - Uses the texture map to obtain the RGB values from the texture for each of the primitive's pixels
  - Determines the final RGB value at each pixel as the product of the values from the previous two steps.
- OpenGL separately computes the values of color and light as if there is no texture and the RGB for the texture as if there is no color or light, and finally, scales one with the other

Multitexturing

- OpenGL allows more than one texture to be applied to a polygon in a pipelined process, each texture being combined with its predecessor in a programmer specified manner. This is called multi-texturing

- Multitexturing requires more than one texture unit - a binding point for a texture - each with an id of the form GL_TEXTUREi. Such a texture unit is activated using code similar to the following

```
1.) glActiveTexture(GL_TEXTURE0);
2.) glEnable(GL_TEXTURE_2D);
3.) glBindTexture(GL_TEXTURE_2D, texture[0]);
4.) glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE))
```

① Selects the current active texture unit
② Enables 2D texturing
③ Binds texture object texture[0]
④ Surface colors come from the texture

- We then want to add a second texture using the same process, but we change the last statement to be:

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE)
```

Here the third parameter indicates that this texture will combine with the other texture using a texture combiner function, which is defined in the initialization routine using:

```
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_INTERPOLATE)
```
(② GL_COMBINE_RGB, ③ GL_INTERPOLATE)

where:

② Indicates that this defines the combiner function
③ Notes that we will use interpolation, as defined by

$$Arg0 * Arg2 + Arg1 * (1 - Arg2)$$

- The following defines these three arguments

```
① glTexEnvi(GL_TEXTURE_ENV, GL_SRC0_RGB, GL_TEXTURE0);
② glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND0_RGB, GL_SRC_COLOR);

③ glTexEnvi(GL_TEXTURE_ENV, GL_SRC1_RGB, GL_TEXTURE1);
   glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB, GL_SRC_COLOR);

④ glTexEnvi(GL_TEXTURE_ENV, GL_SRC2_ALPHA, GL_CONSTANT);
   glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND2_ALPHA, GL_SRC_ALPHA);
```

~~The first sta~~

- Where

① states that the combiner's 0th ~~argument~~ source has RGB values from the 0th texture unit

② states that the 0th operand (Arg0) reads its RGB values from the 0th source

Both ① and ② together imply that Arg0's values are from GL_TEXTURE0 RGB

③ This set of statements set Arg1 values to GL_TEXTURE1's RGB

④ This set of statements ~~set~~ say Arg2 ~~selects~~ is the alpha value of the texture environment variable GL_TEXTURE_ENV_COLOR, which is connoted by GL_CONSTANT

- The following statements in the display routine specify that GL_TEXTURE_ENV_COLOR values are read from the global array constColor

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, constColor);
constColor[3] = alpha;
```

- Finally we specify the texture coordinates for the two texture units, which is done within the polygon definition in the display routine using statements of the form glMultiTexCoord2f(GL_TEXTUREi, *, *);

- In addition to interpolation, other texture combiners, e.g., modulation, addition and subtraction, can be used to create various effects.

Rendering to Texture with Framebuffer Objects

- Often when developing applications we want to texture an object with contents of a video. This requires that we have the capability of rendering to the texture.

- This requires the use of framebuffer objects (FBO's)

- Framebuffer Objects

  - The final destination of the OpenGL rendering pipeline is the framebuffer provided by the windowing system, called the default framebuffer.

  - The framebuffer consists of multiple buffers

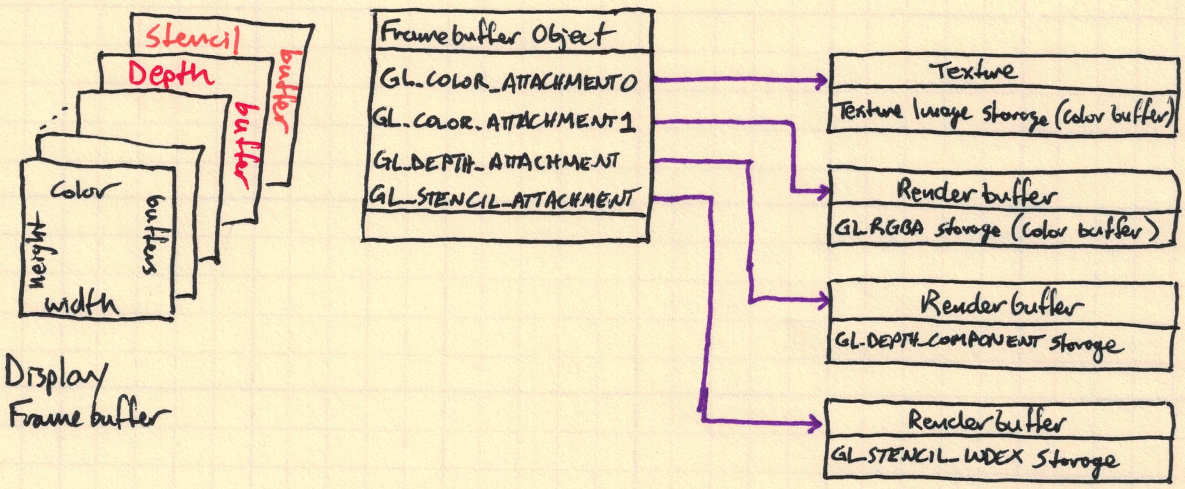
    - Color Buffers with RGBA values
    - Depth Buffer with z-values
    - Stencil Buffer

  - An FBO, on the other hand, is a "pretend" framebuffer. It resides in GPU memory and accessible to the OpenGL pipeline as long as it isn't deleted.

    - Upon creation an FBO contains no color, depth or stencil buffers, but rather the slots into which these can be connected.

    - The advantage of FBO's is that they act as an intermediate off-screen rendering station where we can perform processing without interrupting what is displayed.

    - The slots to attach color buffers are denoted GL_COLOR_ATTACHMENTi, where i runs from 0 to a system defined maximum. The slot for the depth buffer is denoted GL_DEPTH_ATTACHMENT. The slot for the stencil buffer is denoted GL_STENCIL_BUFFER



- The connected buffers can be one of two base types: a texture or a renderbuffer

- A texture comes with its own storage (typically an image, which, nevertheless, can serve as a color, depth, or stencil buffer). To attach a Texture as a color buffer to a FBO use:

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENTi,
                       GL_TEXTURE_2D, textureID, mipmap level)
```

- A renderbuffer is an object that must be created and bound first and then can be assigned with a command of the form

```
glRenderbufferStorage(GL_RENDERBUFFER, storageFormat, width, height)
```
(1) (2)

  - Where:

    (1) storageFormat can be

    - GL_RGBA → Buffer functions as a color buffer
    - GL_DEPTH_COMPONENT → Buffer functions as a depth buffer
    - GL_STENCIL_INDEX → Buffer functions as a stencil buffer

    (2) width, height specify the size of the buffer

- We then use the following command to attach the renderbuffer to an FBO.

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, attachSlot,
                          GL_RENDERBUFFER, renderbufferID)
```
(1) (2)

  - Where:

    (1) attachSlot may be GL_COLOR_ATTACHMENTi, GL_DEPTH_ATTACHMENT, or GL_STENCIL_ATTACHMENT, which specifies the type of buffer it will be

    (2) renderbufferID identifies the renderbuffer that will be used

- FBO's and renderbuffers are themselves constructed in the usual way using glGen...() and glBind...() commands

~~Finally we use the following~~

- Finally, we note, textures may not always be an image. Rather it may be any type of data we wish to exploit the rapid access the GPU provides.

For Next Time

Additional Notes