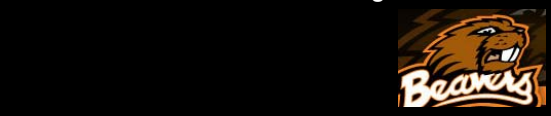
# Rendering to an Offscreen Framebuffer

and

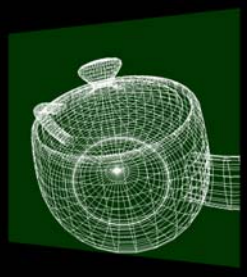
# Rendering to a Texture

**Mike Bailey**

**Oregon State University**

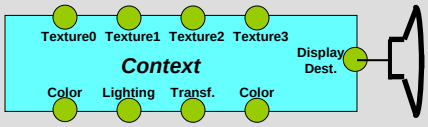






## Preliminary Background – the OpenGL *Rendering Context*

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




Some of these characteristics have a default value (e.g., lines are white, the display goes to the screen) and some have nothing (e.g., no textures exist)

mjb – July 31, 2007

## More Background – What is an OpenGL "Object"?

An OpenGL Object is pretty much the same as a C++, C#, or Java object: it encapsulates a group of data items and allows you to treat them as a single whole. For example, a Texture Object could be created in C++ by:

```
class TextureObject
{
    enum minFilter, maxFilter;
    enum storageType;
    int numComponents;
    int numDimensions;
    int numS, numT, numR;
    void *image;
};
```

Then, you could create any number of Texture Object instances, each with its own characteristics encapsulated within it. When you want to make that combination current, you just need to bring in ("bind") that entire object. You don't need to deal with the information one piece of information at a time.



## More Background – How do you Create an OpenGL "Object"?

In C++, objects are pointed to by their address.

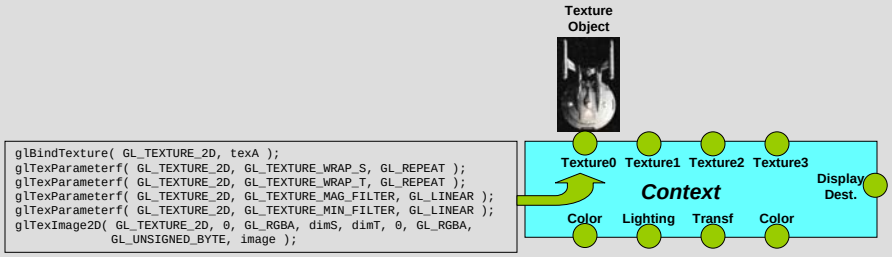
In OpenGL, objects are pointed to by an unsigned integer handle. You can assign a value for this handle yourself (not recommended), or have OpenGL generate one for you that is guaranteed to be unique. For example:

```
GLuint texA;

glGenTextures( 1, &texA );
```

This doesn't actually allocate memory for the texture object yet, it just acquires a unique handle. To allocate memory, you need to bind this handle to the Context.

## More Background -- "Binding" to the Context

The OpenGL term "binding" refers to "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.

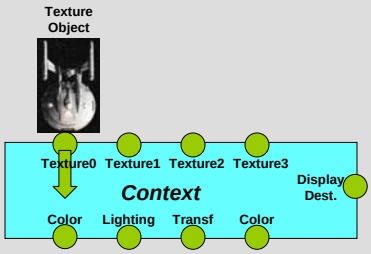



Memory for the object is allocated the first time the handle goes through the bind process.



## More Background -- "Binding" to the Context

When you want to use that Texture Object, just bind it again. All of the characteristics will then be active, just as if you had specified them again.




```
glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, texA );
```

## The Overall Render-to-Offscreen-Framebuffer Process

**You will be changing the Display Destination. Generate a handle for a Framebuffer Object. Generate handles for two (color+depth) Renderbuffer Objects. (These will later be attached to the Framebuffer Object)**

**Bind the Framebuffer Object to the Context**

**Bind the Depth Renderbuffer Object to the Context**
**Assign storage attributes to it**
**Attach it to the Framebuffer Object**

**Bind the Color Renderbuffer Object to the Context**
**Assign storage attributes to it**
**Attach it to the Framebuffer Object**

**Render as Normal**

**Un-bind the Framebuffer Object from the Context**

mjb – July 31, 2007

## Code for the Render-to-Offscreen-Framebuffer Process

**1. In InitGraphics( ), generate Framebuffer and Renderbuffer handles:**

```
GLuint FrameBuffer;
GLuint ColorBuffer;
GLuint DepthBuffer;

glGenFramebuffers(  1, &FrameBuffer );
glGenRenderbuffers( 1, &ColorBuffer );
glGenRenderbuffers( 1, &DepthBuffer );
```

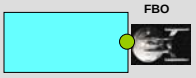
**2. Setup the size you want the offscreen rendering to be :**

```
int sizeX = 2048;
int sizeY = 2048;
```

**You will reference these sizes a few times, so it is a good idea to use variables for the sizes and not hardcode them as numbers in function calls !**

**3. Bind the offscreen framebuffer to be the current output display:**

```
glBindFramebuffer(  GL_FRAMEBUFFER,  FrameBuffer );
```




mjb – July 31, 2007

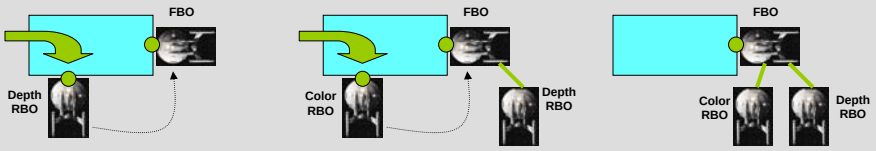
## Code for the Render-to-Offscreen-Framebuffer Process

**4. Bind the Depth Buffer to the context, allocate its storage, and attach it to the Framebuffer:**

```
glBindRenderbufferEXT( GL_RENDERBUFFER_EXT, DepthBuffer );
glRenderbufferStorageEXT( GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT, sizeX, sizeY );
glFramebufferRenderbufferEXT( GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
        GL_RENDERBUFFER_EXT, DepthBuffer );
```

**5. Bind the Color Buffer to the context, allocate its storage, and attach it to the Framebuffer:**

```
glBindRenderbufferEXT( GL_RENDERBUFFER_EXT, ColorBuffer );
glRenderbufferStorageEXT( GL_RENDERBUFFER_EXT, GL_RGBA, sizeX, sizeY );
glFramebufferRenderbufferEXT( GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
        GL_RENDERBUFFER_EXT, ColorBuffer );
```




**6. Check to see if OpenGL thinks the framebuffer is complete enough to use**:

```
GLenum status = glCheckFramebufferStatusEXT( GL_FRAMEBUFFER_EXT );
if( status != GL_FRAMEBUFFER_COMPLETE_EXT )
        fprintf( stderr, "FrameBuffer is not complete.\n" );
```

mjb – July 31, 2007

## Code for the Render-to-Offscreen-Framebuffer Process

**7. Now, render as you normally would. Be sure to set the viewport to match the size of the color and depth buffers:**

```
glClearColor( 0.0, 0.2, 0.0, 1. );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glEnable( GL_DEPTH_TEST );
glShadeModel( GL_FLAT );
glViewport( 0, 0, sizeX, sizeY );

glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluPerspective( 90., 1.,0.1, 1000. );

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );

glTranslatef( TransXYZ[0], TransXYZ[1], TransXYZ[2] );
glMultMatrixf( RotMatrix );
glScalef( scale, scale, scale );
glColor3f( 1., 1., 1. );

glutWireTeapot( 1. );
```

mjb – July 31, 2007

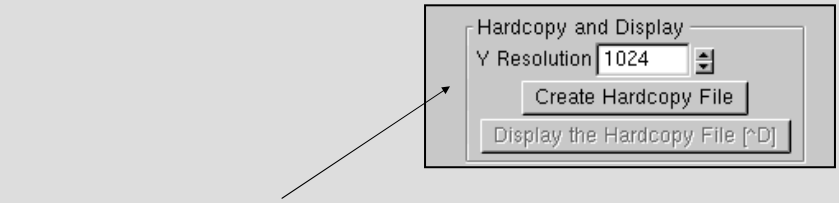
## Code for the Render-to-Offscreen-Framebuffer Process

**8. Read the pixels back and do something with them (such as writing an image file):**

```
unsigned char *image = new unsigned char [3*sizeX*sizeY ];
glPixelStorei( GL_PACK_ALIGNMENT, 1 );
glReadPixels( 0, 0, sizeX, sizeY, GL_RGB, GL_UNSIGNED_BYTE, image );
. . .
```

**9. Un-bind the Framebuffer object by telling OpenGL to go back to rendering to the display device framebuffer:**

```
glBindFramebuffer( GL_FRAMEBUFFER, 0 );
```




Render-to-Framebuffer is *great* for creating arbitrary-resolution hardcopy!



## Remembering the Old Viewport Settings

You can push the old viewport setting on the Attribute Stack so you can easily recall it later:

```
glPushAttrib( GL_VIEWPORT );

glClearColor( 0.0, 0.2, 0.0, 1. );
glClear( GL_COLOR_BUFFER_BIT  |
GL_DEPTH_BUFFER_BIT );
glEnable( GL_DEPTH_TEST );
glShadeModel( GL_FLAT );
glViewport( 0, 0,  sizeX, sizeY );

        // render to the offscreen framebuffer

glPopAttrib( );

glBindFramebuffer( GL_FRAMEBUFFER, 0 );
```

## The Overall Render-to-Texture Process

**You will be changing the Display Destination. Generate a handle for a Framebuffer Object. Generate handles for one (depth) Renderbuffer Object and for a Texture Object. (These will later be attached to the Framebuffer Object)**

**Bind the Framebuffer Object to the Context**

**Bind the Depth Renderbuffer Object to the Context**
**Assign storage attributes to it**
**Attach it to the Framebuffer Object**

**Bind the Texture Object to the Context**
**Assign storage attributes to it**
**Assign texture parameters to it**
**Attach it to the Framebuffer Object**

**Render as Normal**

**Un-bind the Framebuffer Object from the Context**

mjb – July 31, 2007

## Code for the Render-to-Texture Process

**1. In InitGraphics( ), generate a FrameBuffer handle, a RenderBuffer handle, and a Texture handle:**

```
GLuint FrameBuffer;
GLuint DepthBuffer;
GLuint Texture;

glGenFramebuffers(  1, &FrameBuffer );
glGenRenderBuffers( 1, &DepthBuffer );
glGenTextures(          1, &Texture );
```

**2. Setup the size you want the texture rendering to be :**

```
int sizeX = 2048;
int sizeY = 2048;
```

**3. Bind the offscreen framebuffer to be the current output display:**

```
glBindFramebufferEXT(  GL_FRAMEBUFFER,  FrameBuffer );
```

**4. Bind the Depth Buffer to the context, allocate its storage, and attach it to the Framebuffer:**

```
glBindRenderbufferEXT( GL_RENDERBUFFER_EXT, DepthBuffer );
glRenderbufferStorageEXT( GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT, sizeX, sizeY );
glFramebufferRenderbufferEXT( GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
                GL_RENDERBUFFER_EXT, DepthBuffer );
```

mjb – July 31, 2007

## Code for the Render-to-Texture Process

**5. Bind the Texture to the Context:**

```
glBindTexture(  GL_TEXTURE_2D,  Texture );
```

**6. Setup a NULL texture of the size you want to render into and set its properties:**

```
glTexImage2D( GL_TEXTURE_2D, 0, 4, sizeX, sizeY, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL );
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 );
```

**7. Tell OpenGL that you are going to render into the color planes of the Texture:**

```
glFramebufferTexture2DEXT( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, Texture, 0 );
```

**8. Check to see if OpenGL thinks the framebuffer is complete enough to use**:

```
GLenum status = glCheckFramebufferStatusEXT( GL_FRAMEBUFFER_EXT );
if( status != GL_FRAMEBUFFER_COMPLETE_EXT )
        fprintf( stderr, "FrameBuffer is not complete.\n" );
```

mjb – July 31, 2007

## Code for the Render-to-Texture Process

**9. Now, render as you normally would.  Be sure to set the viewport to match the size of the color and depth buffers:**

```
glClearColor( 0.0, 0.2, 0.0, 1. );
glClear( GL_COLOR_BUFFER_BIT  |  GL_DEPTH_BUFFER_BIT );
glEnable( GL_DEPTH_TEST );
glShadeModel( GL_FLAT );
glViewport( 0, 0,  sizeX, sizeY );

glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluPerspective( 90., 1.,0.1, 1000. );

glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );

glTranslatef( TransXYZ[0], TransXYZ[1], TransXYZ[2] );
glMultMatrixf( RotMatrix );
glScalef( scale, scale, scale );
glColor3f( 1., 1., 1. );

glutWireTeapot( 1. );
```

**10. Tell OpenGL to go back to rendering to the hardware framebuffer:**

```
glBindFramebufferEXT( GL_FRAMEBUFFER, 0 );
```

mjb – July 31, 2007

## Code for the Render-to-Texture Process

**11. (optional) Have OpenGL create the multiple mipmap layers for you:**

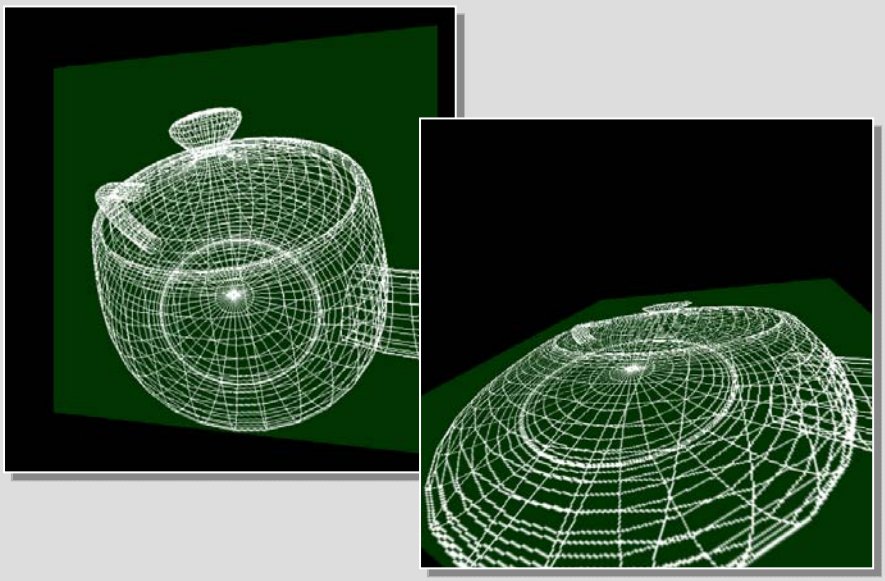
```
glGenerateMipmapEXT( GL_TEXTURE_2D );
```

**12. Now, render the rest of the scene as normal, using the Texture as you normally would:**

```
. . .

glEnable( GL_TEXTURE_2D );

glBindTexture( GL_TEXTURE_2D, Texture );
glBegin( GL_QUADS );
        glTexCoord2f( 0., 0. );
        glVertex2f( -1., -1. );
        glTexCoord2f( 1., 0. );
        glVertex2f(  1., -1. );
        glTexCoord2f( 1., 1. );
        glVertex2f(  1.,  1. );
        glTexCoord2f( 0., 1. );
        glVertex2f( -1.,  1. );
glEnd();

glDisable( GL_TEXTURE_2D );

. . .
```

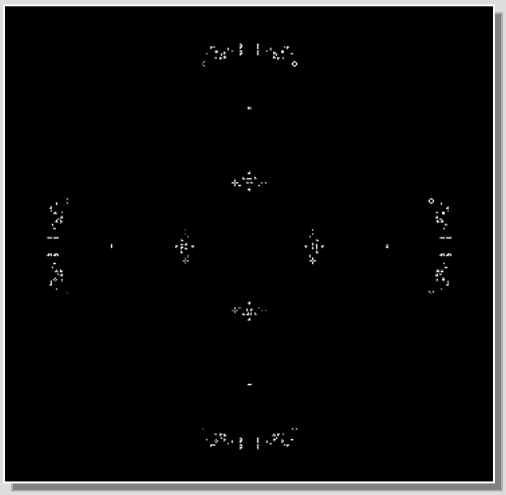
mjb – July 31, 2007

## Render-to-Texture
## A Rotating 3D Teapot Displayed on a Rotating Plane



mjb – July 31, 2007

## Rendering to a Texture and then using that Texture as both a Rendering Texture and as an Input to the Next Iteration: *The Game of Life*



Ping-pong between two different textures. One texture is being read from (the previous state) and the other is being written into (the next state).

mjb – July 31, 2007

## Using GLSL, you Can Bind Multiple Color Framebuffers and Write Separately to Each of Them

Usually in a GLSL Fragment Shader, you write color information to the vec4 gl_FragColor:

```
gl_FragColor = vec4( r, g, b, 1. );          // full color
```

Instead, you can write color information to an array of vec4's called gl_FragData[ ]:

```
gl_FragData[0] = vec4( r, 0., 0., 1. );      // red component
gl_FragData[1] = vec4( 0., g, 0., 1. );      // green component
gl_FragData[2] = vec4( 0., 0., b, 1. );      // blue component
```

Your OpenGL program must then declare what color attachment each gl_FragData[ ] element corresponds to:

```
gluint buffers[ ] = {
        GL_COLOR_ATTACHMENT2_EXT,     // gl_FragData[0]
        GL_COLOR_ATTACHMENT1_EXT,     // gl_FragData[1]
        GL_COLOR_ATTACHMENT3_EXT      // gl_FragData[2]
};
glDrawBuffers( 3, buffers );
```

This is called "Multiple Render Targets (MRT)

mjb – July 31, 2007