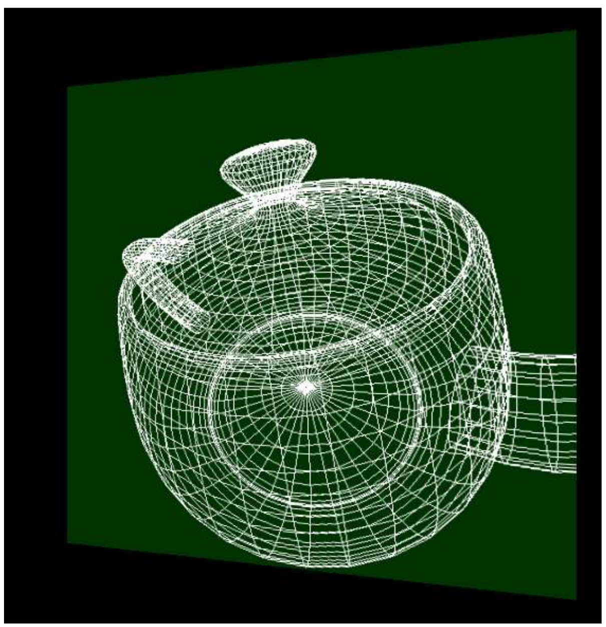
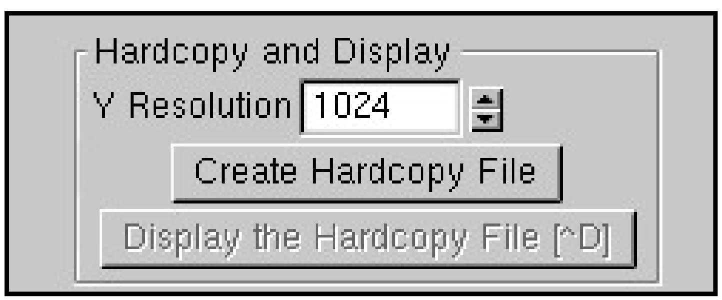
Rendering to a Texture:
A Good Way to Get a Texture for Use in a Shader Second Render Pass

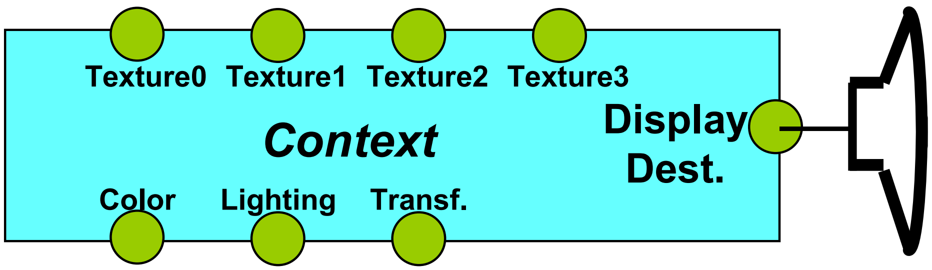
Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu





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)

More Background – What is an OpenGL "Object"?

An OpenGL Object is pretty much the same as a C, 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 emulated 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/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:

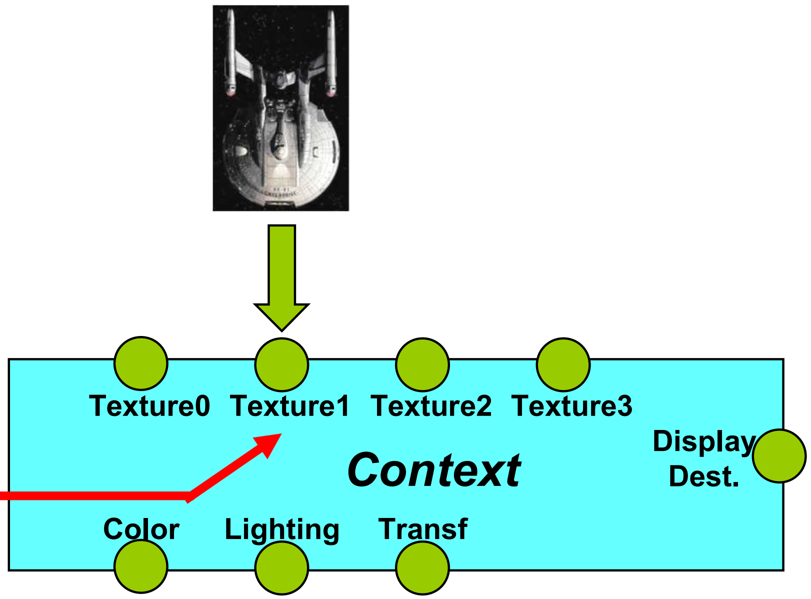
```
GLuint texA;

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 Star Trek-ish 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.

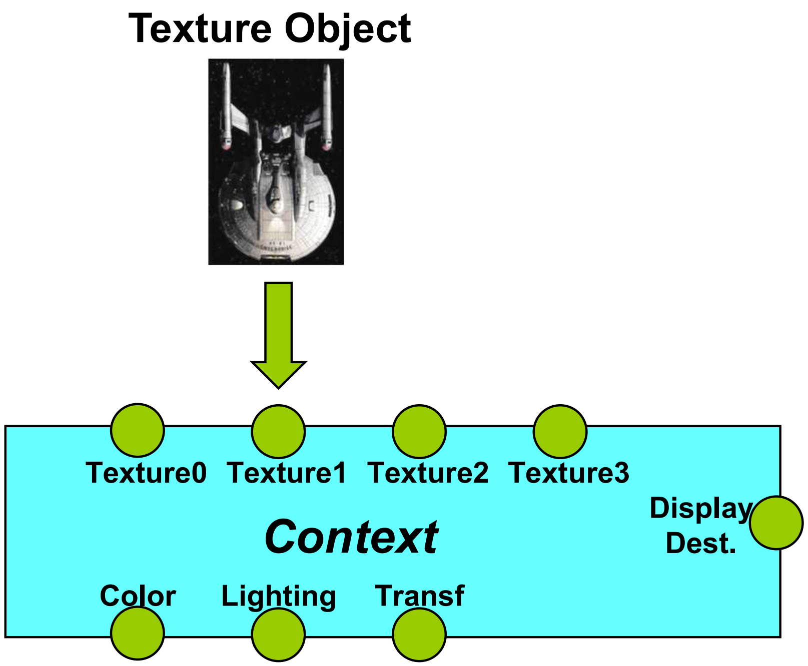
Texture Object




```
glActiveTexture( GL_TEXTURE1 );
glBindTexture( GL_TEXTURE_2D, texA );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, dimS, dimT, 0, GL_RGBA,
                     GL_UNSIGNED_BYTE, image );
```

More Background -- “Binding” to the Context

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




```
glActiveTexture( GL_TEXTURE1 );

glBindTexture( GL_TEXTURE_2D, texA );
```

The Overall Render-to-Texture Process

1 Generate a handle for a Framebuffer Object.

2 Generate handles for one Color Texture Object and for one Depth Texture Object.

3 Bind the Color Texture Object to the Context (glBindTexture).
Act as if you are downloading texels to it but set that array to NULL. Just give the texture size.
Assign texture parameters to it.
Attach it to the Framebuffer Object (glFramebufferTexture2D) as the GL_COLOR_ATTACHMENT0.

4 Bind the Depth Texture Object to the Context (glBindTexture).
Act as if you are downloading texels to it but set that array to NULL. Just give the texture size.
Assign texture parameters to it.
Attach it to the Framebuffer Object (glFramebufferTexture2D) as the GL_DEPTH_ATTACHMENT.

5 Bind the Framebuffer Object to the Context (glBindFramebuffer), thus un-binding the display monitor.

6 Render as normal. Be sure the size of the viewport matches the size of the textures you created.

7 Un-bind the Framebuffer Object from the Context, glBindFramebuffer(0),, thus re-binding the display monitor.

Code for the Render-to-Texture Process

In InitGraphics( ), generate a FrameBuffer handle, a ColorBuffer handle, and a DepthBuffer handle:

```
GLuint FrameBuffer;
GLuint ColorTexture;
GLuint DepthBuffer;

glGenFramebuffers(  1, &FrameBuffer );
glGenTextures(      1, &ColorTexture );
glGenTextures(      1, &DepthBuffer );
```

Bind the offscreen framebuffer to be the current output display:

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

Setup the size you want the texture rendering to be (this can be larger than the display window or even the display monitor):

```
int sizeS = 2048;
int sizeT = 2048;
```

Code for the Render-to-Texture Process

Bind the Color Buffer to the context, allocate its storage, and attach it to the current Framebuffer:

```
glBindTexturer(   GL_TEXTURE_2D, ColorBuffer );
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, sizeS, sizeT, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, ColorBuffer, 0);
```

Bind the Depth Buffer to the context, allocate its storage, and attach it to the current Framebuffer:

```
glBindTexturer(   GL_TEXTURE_2D, DepthBuffer );
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, sizeS, sizeT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, DepthBuffer, 0);
```

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

```
GLenum status = glCheckFramebufferStatus( GL_FRAMEBUFFER );
if( status != GL_FRAMEBUFFER_COMPLETE )
        fprintf( stderr, "FrameBuffer is not complete.\n" );
```

Code for the Render-to-Texture Process

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

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glEnable( GL_DEPTH_TEST );
glShadeModel( GL_SMOOTH );
glViewport( 0, 0, sizeS, sizeT );

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

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

glRotatef( RotY, 0., 1., 0. );
glRotatef( RotX, 1., 0., 0. );
glScalef( Scale, Scale, Scale );
glColor3f( 1., 1., 1. );

glutWireTeapot( 1. );
```

Tell OpenGL to go back to rendering to the display monitor:

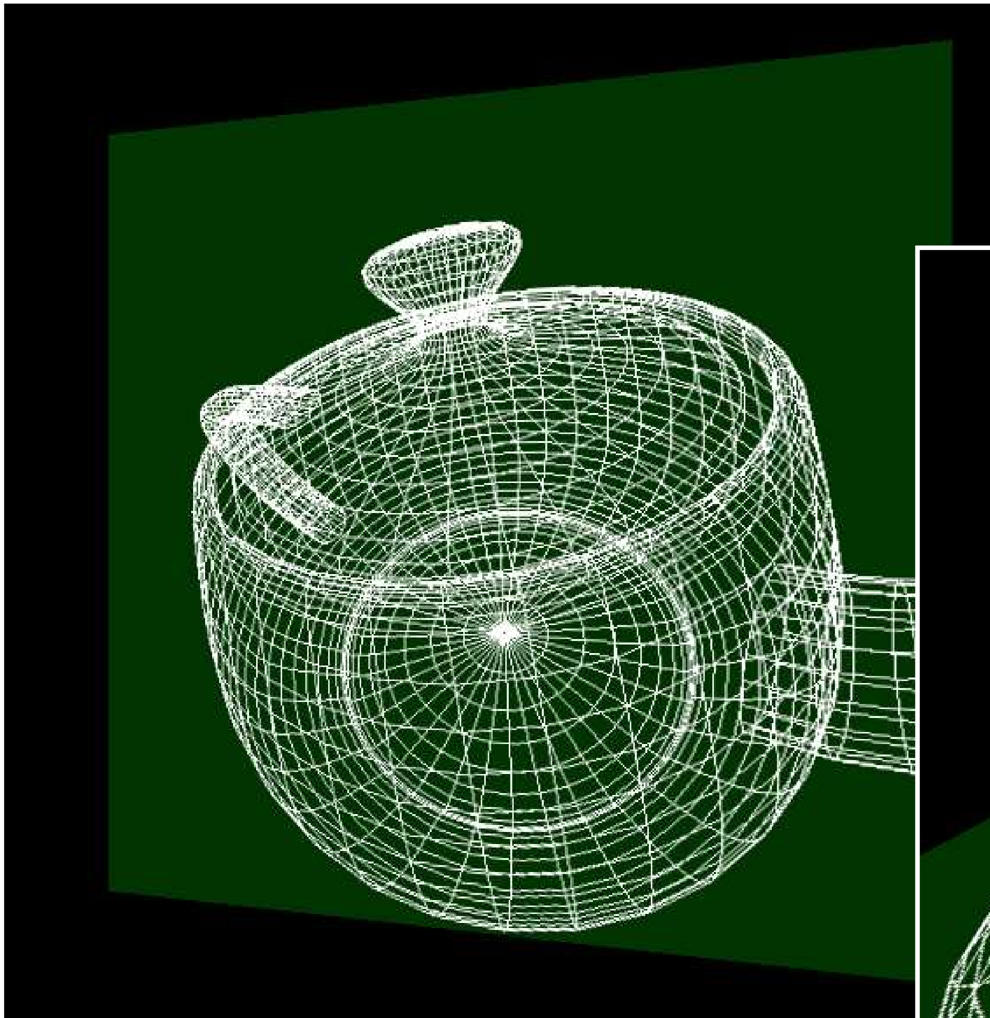
```
glBindFramebuffer( GL_FRAMEBUFFER, 0 );
```

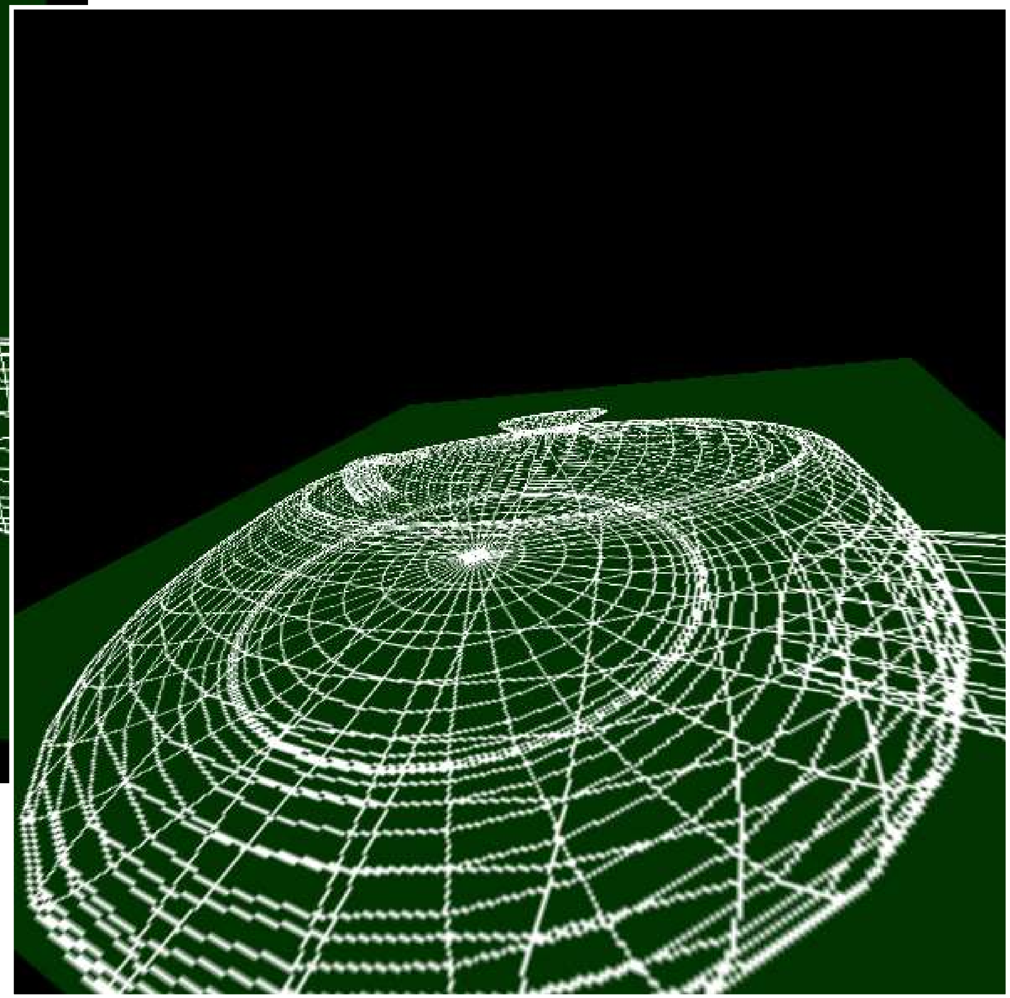
Code for the Render-to-Texture Process

Now, render the second pass of the scene as normal, mapping the Texture onto a quadrilateral:

```
glClear( GL_COLOR_BUFFER_BIT  |  GL_DEPTH_BUFFER_BIT );
glEnable( GL_DEPTH_TEST );
glShadeModel( GL_FLAT );
glViewport( 0, 0,  v, v );

glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluOrtho2D( -1., 1.,   -1., 1. );

glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );

glEnable( GL_TEXTURE_2D );
glActiveTexture( GL_TEXTURE1 );
glBindTexture( GL_TEXTURE_2D, ColorBuffer );
Pattern->Use( );
Pattern->SetUniforVariable( "uTexunit", 1 );
    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( );
Pattern->UnUse( );
glDisable( GL_TEXTURE_2D );
```

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





Render-to-Texture : An Image-filtered 3D Noisy-Ovaled Teapot using Glman

twopass.glib

```
##OpenGL GLIB
Perspective 90

Texture2D   6  1024 1024            # a 1024x1024 NULL texture
RenderToTexture  6                  # render to texture unit 6
Background  0. 0. 0.
Clear
LookAt  0. 0. 3.   0. 0. 0.   0. 1. 0.

Vertex      ovals.vert
Fragment   ovals.frag
Program    Ovals                                          \
           uAd <.01 .2 .5>  uBd <.01 .2 .5>               \
           uNoiseAmp <0. 0. 1.>  uNoiseFreq <0. 1. 2.>    \
           uTol <0. 0. 1.>

Teapot

RenderToTexture                     # render to the display monitor
Background 0. 0. 0.
Clear
LookAt  0. 0. 3.   0. 0. 0.   0. 1. 0.

Vertex      image.vert
Fragment   image.frag
Program    Filter       uTexUnit 6                        \
                        uEdgeDetect <true>                \
                        uTedge   <0. 0. 1.>               \
                        uTSharp <-3. 1. 10.>

QuadXY .2  2.
```

Multipass Algorithm to Render and then Image Process

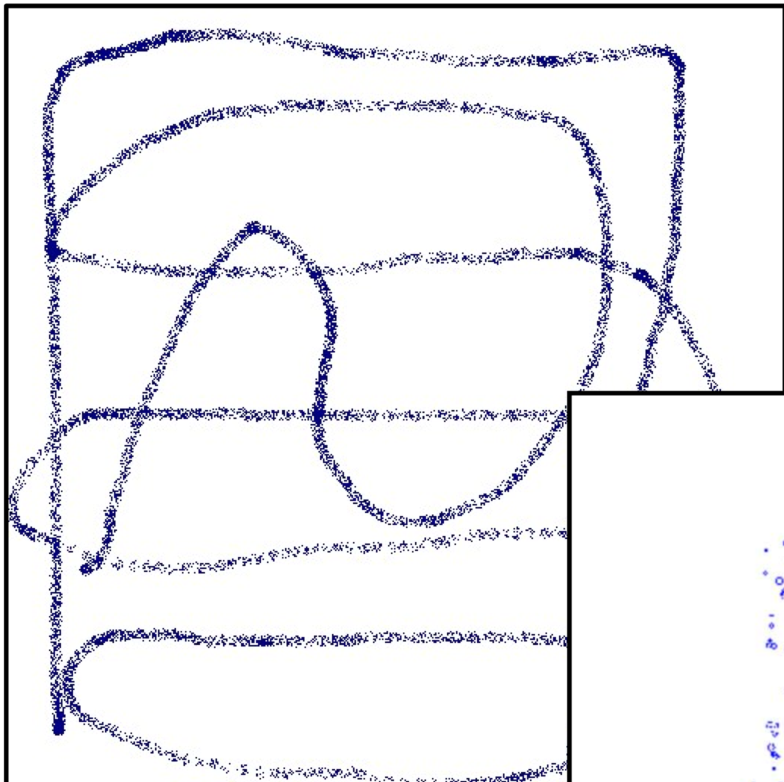
Original | Sharpened | Edge Detected

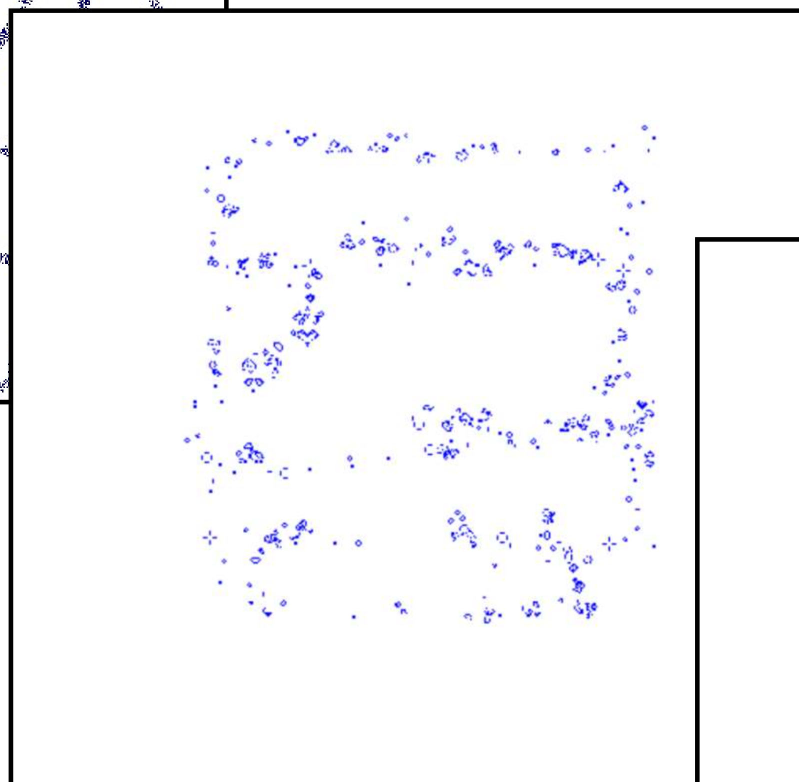
No Noise

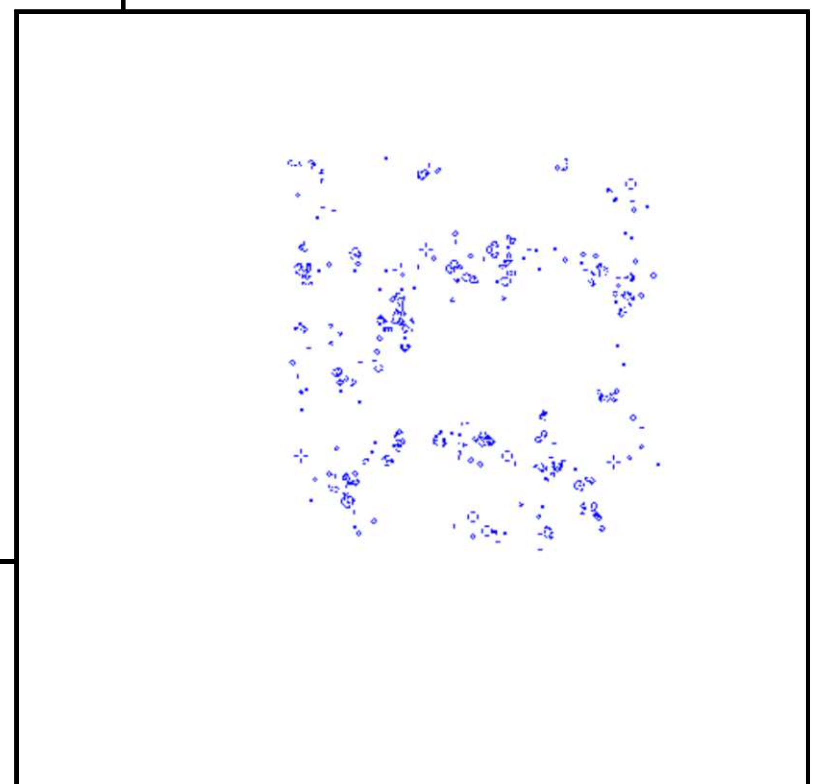
Noise

Multipass Algorithm to Implement Conway's 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).







life.glib

```
##OpenGL GLIB
Perspective 70

# setup the 2 textures:
Texture2D  5  paint0.bmp
Texture2D  6  512  512

# execute the first iteration:
RenderToTexture 6
Background 0. 0. 0.
Clear
Vertex        life.vert
Fragment   life.frag
Program   GameOfLife1    uTexUnit 5
TextureMatrix
Translate 0. 0. -3.08
QuadXY .2 2.

# render it so we can see it:
RenderToTexture
Background 0. .2 0.
Clear
Vertex         texture.vert
Fragment    texture.frag
Program Texture1    uTexUnit 6
ModelViewMatrix
Translate 0. 0. -3.08
QuadXY .2 2.
SwapBuffers
```

```
# execute the second iteration:
RenderToTexture 5
Background 0. 0. 0.
Clear
Vertex         life.vert
Fragment    life.frag
Program GameOfLife2   uTexUnit  6
QuadXY .2 2.

# render it so we can see it:
RenderToTexture
Background .2 0. 0.
Clear
Vertex        texture.vert
Fragment   texture.frag
Program Texture    uTexUnit 5
QuadXY .2 2.

# repeat:
animate
```

life.frag, I

```
uniform sampler2D uTexUnit;
in  vec2    vST;

const vec3  DEAD  =  vec3( 1., 1., 1. );
const vec3  ALIVE =  vec3( 0., 0., 1. );

const float TB = 0.20;        // color threshold
const float TR = 0.20;        // color threshold
const int T1 = 1;             // critical # of neighbors
const int T3 = 3;             // critical # of neighbors
const int T4 = 4;             // critical # of neighbors

void main( )
{
        ivec2 isize = textureSize( uTexUnit, 0 );
        vec2 st = vST;
        ivec2 ist     = ivec2( st.s*float(isize.s-1) , st.t*float(isize.t-1) );        // 0 -> dimension-1
        ivec2 istp0  = ivec2( 1, 0 );
        ivec2 ist0p  = ivec2( 0, 1 );
        ivec2 istpp  = ivec2( 1, 1 );
        ivec2 istpm = ivec2( 1, -1 );

        vec3 i00      =  texelFetch( uTexUnit, ist, 0 ).rgb;        // index using integer indices
        vec3 im10    =  texelFetch( uTexUnit, ist-istp0, 0 ).rgb;
        vec3 i0m1    =  texelFetch( uTexUnit, ist-ist0p, 0 ).rgb;
        vec3 ip10    =  texelFetch( uTexUnit, ist+istp0, 0 ).rgb;
        vec3 i0p1    =  texelFetch( uTexUnit, ist+ist0p, 0 ).rgb;
        vec3 im1m1 =  texelFetch( uTexUnit, ist-istpp, 0 ).rgb;
        vec3 ip1p1   =  texelFetch( uTexUnit, ist+istpp, 0 ).rgb;
        vec3 im1p1  =  texelFetch( uTexUnit, ist-istpm, 0 ).rgb;
        vec3 ip1m1  =  texelFetch( uTexUnit, ist+istpm, 0 ).rgb;
```

life.frag, II

```
        int sum = 0;
        if( im10.b   > TB  &&  im10.r   < TR )   sum++;
        if( i0m1.b   > TB  &&  i0m1.r   < TR )   sum++;
        if( ip10.b   > TB  &&  ip10.r   < TR )   sum++;
        if( i0p1.b   > TB  &&  i0p1.r   < TR )   sum++;
        if( im1m1.b  > TB  &&  im1m1.r  < TR )   sum++;
        if( ip1p1.b  > TB  &&  ip1p1.r  < TR )   sum++;
        if( im1p1.b  > TB  &&  im1p1.r  < TR )   sum++;
        if( ip1m1.b  > TB  &&  ip1m1.r  < TR )   sum++;

        vec3 newcolor = i00;
        if( sum == T3 )
        {
                newcolor = ALIVE;
        }
        else if( sum <= T1  ||  sum >= T4 )
        {
                newcolor = DEAD;
        }

        gl_FragColor = vec4( newcolor, 1. );
}
```

Oregon State
University
Computer Graphics

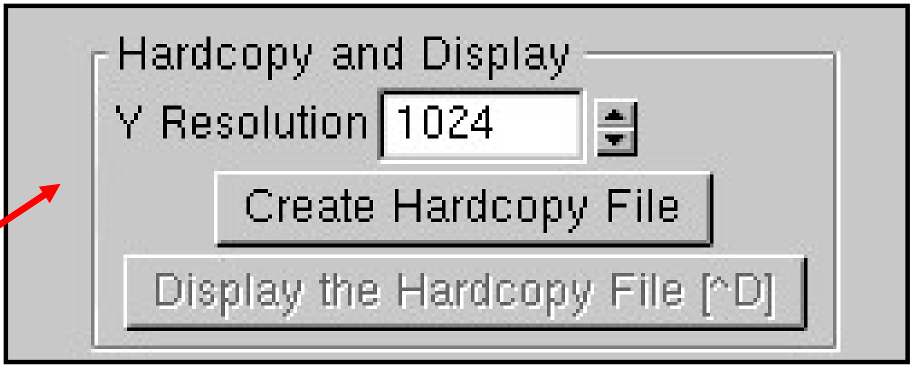
Code for Reading Pixels Back

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

```
glBindFramebuffer(  GL_FRAMEBUFFER,  FrameBuffer );
unsigned char *image = new unsigned char [ 3*sizeS*sizeT ];
glPixelStorei( GL_PACK_ALIGNMENT, 1 );
glReadPixels( 0, 0, sizeS, sizeT, GL_RGB, GL_UNSIGNED_BYTE, image );
. . .
```

Tell OpenGL to go back to rendering to the display monitor:

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



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

(Especially if you are using shader-generated procedural texture patterns!)