Shaders in Jitter

Powerful as Jitter is, simple appearing patches often run up against the performance limitations of the computer, especially if we are trying for high resolution. You can't do very many operations on 1.2 million pixels in 1/30th of a second. High performance games can produce visually stunning effects by reprogramming the computer graphics adapter. Jitter provides the same capability through custom shaders and jit.gl.slab.

The openGL language allows custom programming in the form of *shaders*. Shaders are somewhat like textures, originally intended to add detail to openGL worlds. Unlike textures, which are simply images to paint onto surfaces, shaders are code, and can react to and modify the position and color of each pixel in a scene. There are actually two programs in a shader, one that modifies the position of vertices, and a second that modifies the color of pixels. The two are separated by a multi-step rasterization process, but it is possible for the pixel processor to read variables generated by the vertex processor. The language specifications refer to pixels as fragments, so the second program is called the fragment processor.

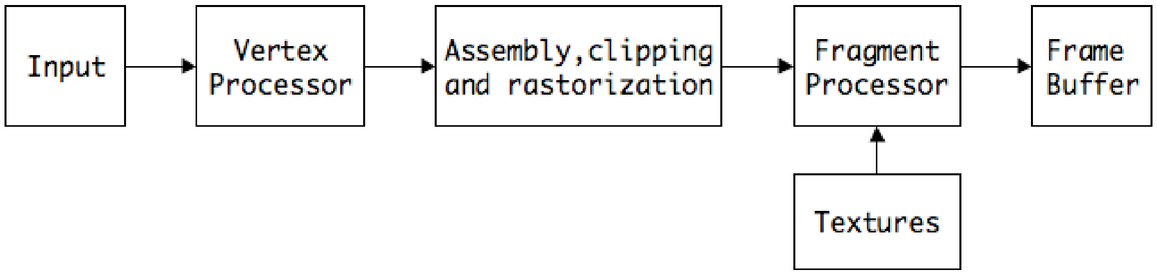


Fig 1. The openGL pipeline (simplified).

There are some functions that must be performed by any vertex processor-- these are the operations specified by rotate, scale and position parameters, as well as global settings such as point of view and lighting positions. A custom shader must perform these computations, but it's really just a matter of including some boilerplate code. The fragment shader should compute color based on variables passed from the vertex shader, but is otherwise fairly free. The default fragment shader applies the current texture, and custom shaders can access multiple textures.

Lack of coding experience is not a barrier to using shaders-- jitter includes more than 200 in the jitter-shaders folder. Many of these perform actions similar to the effects found in the gallery (brcosa, rota, etc), many expand the jitter bag of tricks. They are not documented, but it is not difficult to discover how to use them. The first step is to go through the jitter tutorials, starting with number 41.

Tutorial 41 introduces the jit.gl.shader object, which is used to process objects generated by jit.plato, jit.gridshape, and so forth. Note that shaders are managed in a manner similar to textures: the jit.gl.shader object is connected to the render object and given a name. Passing this name to a shape object invokes the shader. You can define as many shaders as you want, only one is used by a shape at a time. Since shaders are responsible for texturing, invoking a shader may turn the texture off or treat it oddly-- this is demonstrated in the jit.gl.shader help file.

Many shaders are just projected on jit.gl.videoplane. This provides raster style graphics with openGL performance. The basic scheme is shown in figure 2.

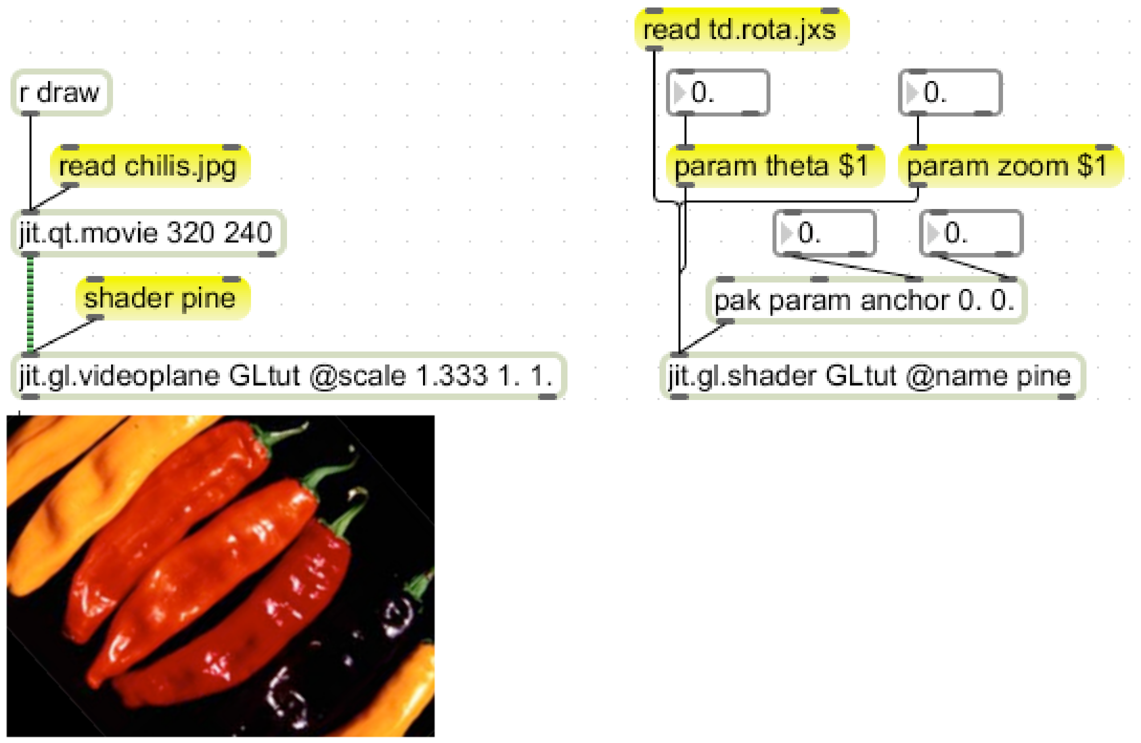


Figure 2. Simple application of shaders.

This assumes a render and window combination named Gltut. (See my open GL tutorial) In essence, the shader file is read into a jit.gl.shader object, which is given a unique name. Here the td.rota.jxs file is read into a shader object named pine. The effect is applied to the jit.gl.videoplane by the message "shader pine". The shader file td.rota.fx works much like rota. Parameters are fed to the jit.gl.shader object. Note that the parameter name is the first argument to a "param" message. The parameter names are not quite the same though. You can discover the parameter names with the "getparamlist" message.

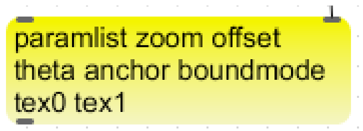

paramlist zoom offset theta anchor boundmode tex0 tex1

Figure 3.

The parameter name is followed by one or more arguments. You discover how many arguments with the "getparamval" message with a name. The result will tell you how to format the param message.

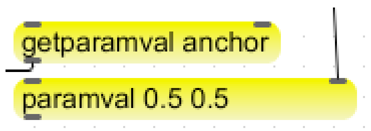


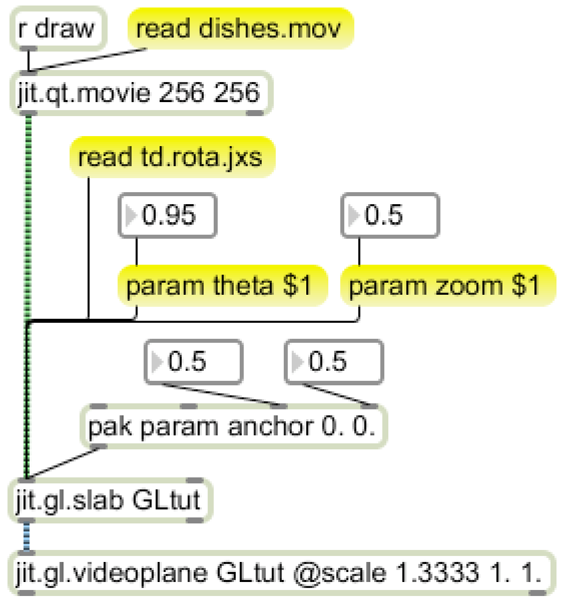
Figure 4.
Figure 4 is an example for the anchor attribute. The numbers are a bit different from what you might use with rota proper. Jit.rota anchors according to matrix address. The shader equivalent is really processing a texture, so locations are defined in terms of a height and width of 1.0 measured from the upper left corner.

You can apply the same shader to several objects (they will share parameter settings) or you can load the same shader file into several jit.gl.shader objects to apply a process with individual settings.

There are quite a few shader files supplied in the Max 5/Cycling74/Jitter-Shaders folder. They don't all work with jit.gl.shader, and some may not work with all video cards. The files that parallel the jitter effect set are pretty robust and about 100 times more efficient than their counterparts.

Jit.gl.slab

The second way to use shaders is in jit.gl.slab.



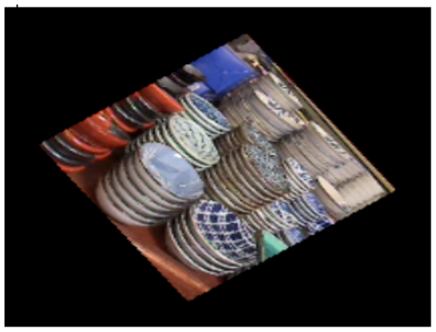
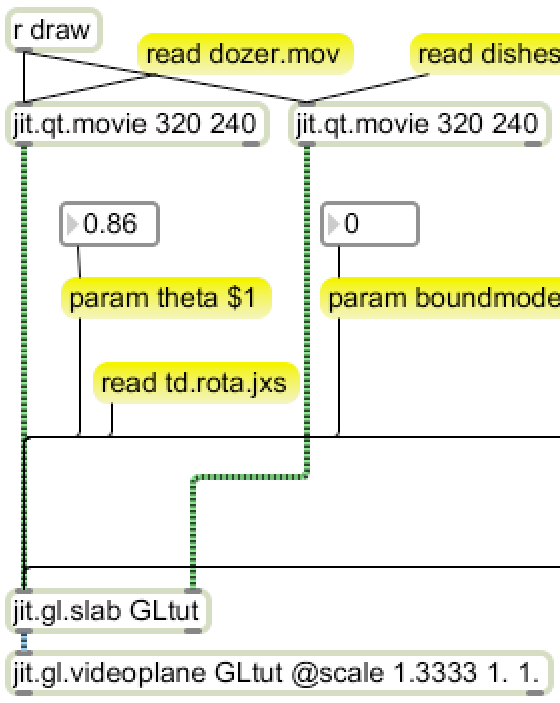


Figure 5.
Jit.gl.slab accesses the GPU in a more direct manner than jit.gl.shader. The output is a special matrix known as a jit_gl_texture. This is stored in video RAM and is available at high transfer rates to jit.gl.texture and jit.gl.videoplane. Transfer the slab output to raster graphics via a jit.matrix. (There may be some performance loss when doing this.) The file loading and parameter management are the same as jit.gl.shader, although it is not necessary to name slabs.

Most operations can be carried out as easily in jit.gl.slab as jit.gl.shader. There are a few shader files that run in one but not the other-- this seems to depend on subtle aspects of the internal code. Slab offers a variable number of inlets, so mix and transition effects are easily accomplished. Slab also works best when you want to chain effects.



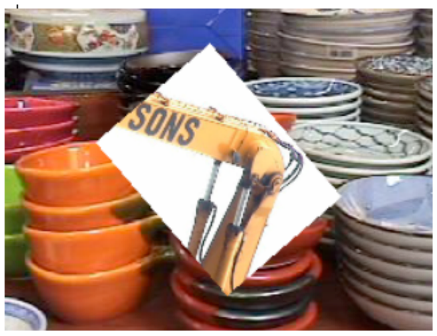


Figure 6.
Figure 6 shows an example of a slab with two inputs. In the case of the td.tota.jxs shader, the second input is used as the background in boundmode 0.

Shaders can generate images from scratch. Figure 7 shows the output of gn.bricks.jxs. Images that are generated this way can be very responsive to external events.

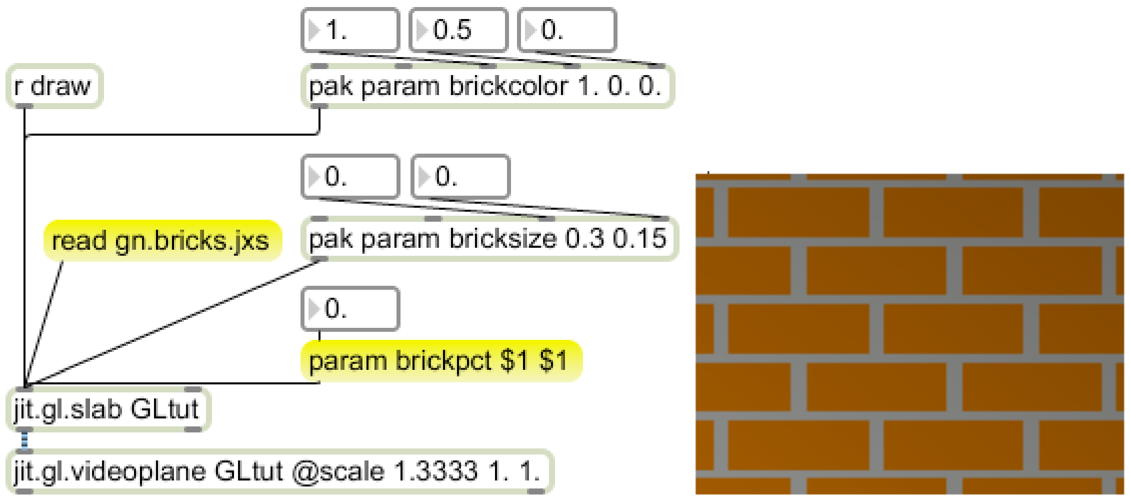

1.
0.5
0.
r draw
pak param brickcolor 1. 0. 0.
0.
0.
read gn.bricks.jxs
pak param bricksize 0.3 0.15
0.
param brickpct $1 $1
jit.gl.slab GLtut
jit.gl.videoplane GLtut @scale 1.3333 1. 1.

Figure 7.

Homebrew Shaders

Note: The following assumes some knowledge of programming in java or C. If you can write code for mxj or jsui you will be OK. Detailed specification for the glsl language can be found at www.opengl.org.

The most powerful feature of jitter shaders is the ability to write high performance code. Shaders are programmed in a fairly simple language that should be comfortable for anyone who knows Java or C. The OpenGL Shader Language is well documented on the web, and in a heavy tome known as the "Orange Book." Jitter's version of shader code is written in an XML file (.jxs) and compiled on the spot by the jit.gl.slab object. You can see the code behind any shader with the message dump source. The best way to learn GLSL is by experimenting with an existing shader. I've taken a standard shader and simplified it a bit-- that is found as tut.bricks.jxs. This is an example presented in most glsl tutorials.

The file starts with some lines of XML that allow jitter to patch Max messages to the code. The <jittershader> tag identifies this file as something slab can open. The first interesting tag is <param> which determines the name, type and default values for input parameters. After a few of these comes the <language> tag, which encloses most of the file. Within <language> the <bind> tags direct the parameters to either the vertex program (vp) or fragment program (fp).

```
<jittershader name="bricks">
        <description>Shader for generating procedural bricks</description>
        <param name="brickcolor" type="vec3" default="1.0 0.3 0.2" />
        <param name="mortarcolor" type="vec3" default="0.85 0.86 0.84" />
        <param name="bricksize" type="vec2" default="0.30 0.15" />
        <param name="brickpct" type="vec2" default="0.90 0.85" />
```

```
	<language name="glsl" version="1.0">
		<bind param="brickcolor" program="fp" />
		<bind param="mortarcolor" program="fp" />
		<bind param="bricksize" program="fp" />
		<bind param="brickpct" program="fp" />
```

The vertex progarm is named "vp". The easiest way to deal with vertices is to load boilerplate code from the shared folder. The file sh.passthrough.vp.glsl does what is required and provides information about pixel location that we need in the fragment processor.

```
<program name="vp" type="vertex" source="sh.passthrudim.vp.glsl" />
```

The fragment code starts here. It is enclosed in a <program> tag.

```
		<program name="fp" type="fragment">
<![CDATA[
uniform vec3  brickcolor, mortarcolor;
uniform vec2  bricksize;
uniform vec2  brickpct;

varying vec2  texcoord0;  // provided by passthru
varying vec2  texdim0;  // provided by passthru
void main(void)
{
   vec3  color;
   vec2  position, useBrick;
   vec2 index = texcoord0/texdim0;  // normalized index

   position = index / bricksize;  // find where we are in a brick

   if (fract(position.y * 0.5) > 0.5)
      position.x += 0.5;
   position = fract(position);


   useBrick = step(position, brickpct);
  color  = mix(mortarcolor, brickcolor, useBrick.x * useBrick.y);
   gl_FragColor = vec4 (color, 1.0);
}
]]>
		</program>
	</language>
</jittershader>
```

Shading code is something like C as it appeared back in 1980 or so. The most important distinction is there is no type conversion. If you place an int where a float is needed, there will be an error. The easy way to deal with this is always use floats. A nice feature is the inclusion of vector data types: vec2,vec3,vec4. These provide efficient manipulation of two and three dimensional data as well as 4x4 matrix operations. You can create a vec2 variable by declaring it and stuff it with a vec2 function:

```
vec2  center = vec2(0.5,0.5);
```

The members of these vectors may be accessed by letters commonly used for position, texture mapping, or color in the gl language:

```
vec4 cell;
cell.x, cell,y,cell.z,cell.w
cell.s cell,t,cell.p,cell.q     // texture coord r is called p in glsl
cell.r, cell,g,cell.b,cell.a
cell.xyz, cell.xy
cell.stp, cell.st
cell.rgb, cell.rg, cell.ra
```

are all legal, but not cell.xg.

The variables that are set by param messages or created in the vertex processor are global and must be declared before the main() function. There are two qualifiers used: uniform variables are set by the param messages, and cannot change from fragment to fragment. Varying variables are determined in the vertex processor and must be declared there as well.

The action is in the main() function, of course. This will be called for every pixel in the input matrix. All that is known by the vertex processor is the color of the input pixel: gl_Color. The job of the vertex processor is to fill the variable gl_FragColor with the appropriate output. The passthrudim code provides more information derived from the vertices of the primitive being processed. Texcoord0 and texcoord1 provide the pixel location in the matrices input to the left and right inlets of the jit.gl.slab object. Texcoord0 is a vec2 containing both x and y (or s and t) coordinates of current pixel. In openGL, these coordinates may be presented either in normalized form (0-1) or in rectangular form that follows the matrix addresses. Jitter uses the second option, so we also must know the dimensions of the input. This is provided in another vec2, texdim0. Many operations need to be scaled to the dimensions of the image, or were originally written for normalized coordinates, so the first line of code usually constructs a normalized index.

```
vec2 index = texcoord0/texdim0;  // normalized index
```

The rest of the code in the bricks generator determines which of two colors (brickcolor, mortarcolor) to apply to the output. First divide the screen into bricks:

```
position = index / bricksize;  // find where we are in a brick
```

Since both variables are vec2, two divisions are performed and the result is a vec2. Next determine if the pixel is on brick or mortar. This code depends on some functions that are predefined in glsl. Fract returns the fractional part of a floating point number. This bit of code will offset the even rows of bricks.

```
if (fract(position.y * 0.5) > 0.5)
      position.x += 0.5;
   position = fract(position);
```

The step function compares the members of vectors, returning 1.0 when the second argument is greater than the first. Again, the result is an appropriate vector.

```
useBrick = step(position, brickpct);
```

The mix function interpolates between members of its first arguments according to a third argument, which is generally a float. Multiplying useBrick.x and useBrick.y will choose the mortar color if either member is 0.

```
color  = mix(mortarcolor, brickcolor, useBrick.x * useBrick.y);
```

The final step is to fill gl_FragColor, which must be an RGBA vector.

```
gl_FragColor = vec4 (color, 1.0);
```

The hallmark of good shader code is efficiency. The video processor is performing every operation at once, albeit on different pixels that march down the pipe. The number of variables available and the number of operations that can be done is limited by the hardware, and we don't know we have hit that limit until slab refuses to compile the code. These limits vary from one video card to another, and not always in the sense that the newest is most capable. So avoid unnecessary operations, use vectors wherever possible, and use the built in functions.

Secrets of repose slab

We have already seen the effectiveness of jit.repos for interesting and efficient image processing. You will remember that the repos process specifies the color of a cell by a control matrix that contains the address of a different cell.   (If not, see my Repos Tutor.) There is a slab version, coded in the file td.repos.jxs. Unfortunately, we cannot use the control matrices generated for jit.repos directly in the slab version. Control matrices for td.repose.jxs must be float32 type with a range of 0.0 to 1.0. If relative displacement is used, the value 0.5 indicates no change. Its not difficult to convert existing control matrices, but I find it just as convenient to generate them directly. Figure 7 shows a patch to do so.

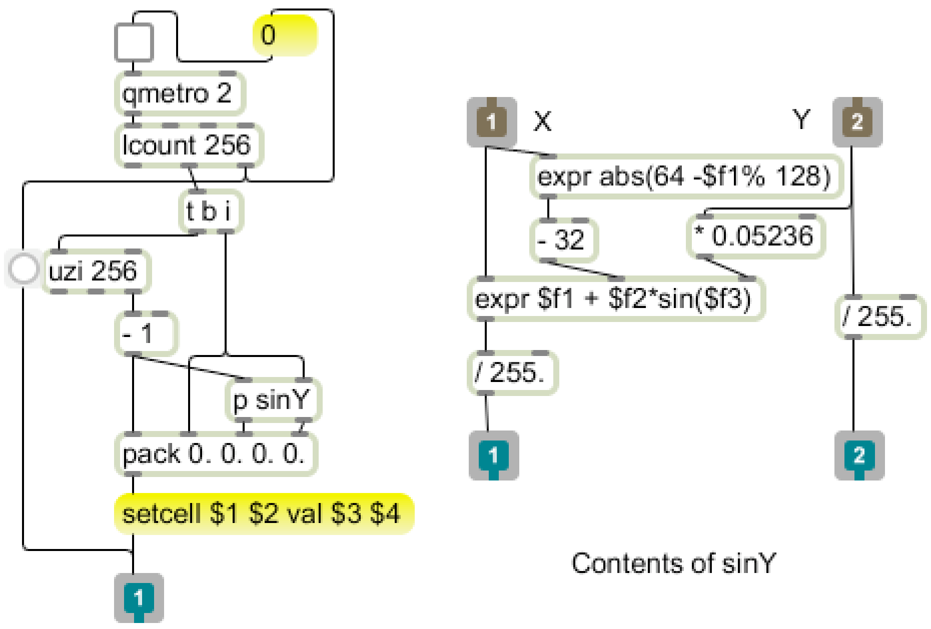


Figure 7. Generate control matrices for td.repos.jxs

Figure 8 shows the effect of this on the bricks image:

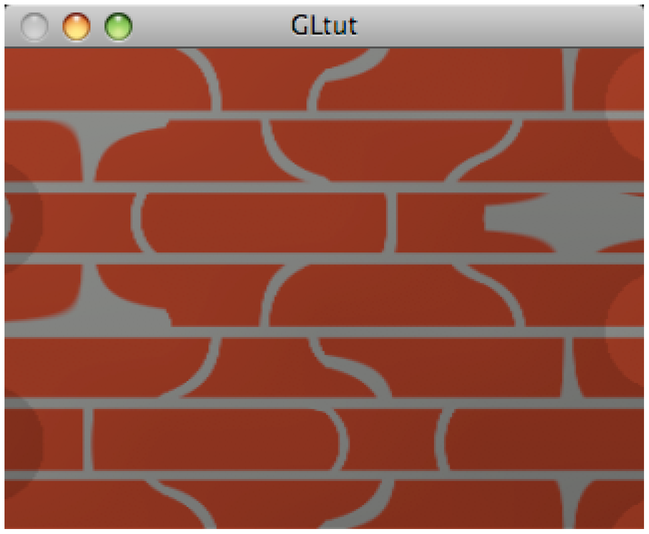


Figure 8. Repose of x coordinate by the sine of the Y coordinate.

The code in td.repose.jxs demonstrates how to get the pixel color from a texture. This starts with the usual varying and uniform variables-- as before the varying variables have their origin in the vertex processor. There is now a new type of variable, sampler2DRect. This provides a handle to the input texture-- the "Rect" indicates that the texture is indexed by pixel address. A plain sampler2D indexes by normalized address 0.0-1.0. This only works properly for square textures that are a power of 2 in size. Jitter uses the rectangle textures so any aspect is available. The names tex0, tex1 and so on are bound to the inlets of the slab object.

```
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texdim0;
uniform vec2 amt;
uniform vec2 mode;
uniform vec2 boundmode;
uniform sampler2DRect tex0;
uniform sampler2DRect tex1;

void main()
{
   vec4 look = texture2DRect(tex1,texcoord1);//sample control matrix
   vec2 rela = ((look.ra-0.5)*texdim0*amt*2.)+texcoord0;//relative
   vec2 abso = look.ra*texdim0*amt;//absolute
   vec2 coord = mix(abso,rela,clamp(mode, 0., 1.));

   vec2 wrap = mod(coord,texdim0);
   vec2 fold = mix(wrap,texdim0-wrap,floor(mod(coord,
                                   texdim0*2.)/texdim0));

   coord = mix(wrap,fold,clamp(boundmode, 0., 1.)); //choose boundmode
   vec4 repos = texture2DRect(tex0, coord);

   // output texture
   gl_FragColor = repos;
}
```

Example 2. Code for td.repos.jxs

The texture2DRect() function retrieves a four color sample from the texture at the coordinates specified. This is a vec4 in RGBA format. If the control matrix was created in jitter, this will be mapped to alpha and red. The slab inlet assumes a matrix is in ARGB format, but a texture is in RGBA. We will need to remember that when we write shaders to generate control matrices. The pertinent values will be available in look.ra.

The vec2s rela and abso are both calculated from the values in look.ra. The one to use is selected by the mode variable inside a mix function. The relative coordinates are scaled to produce a positive or negative displacement from the input of 0.0 to 1.0. In either case, the control value of 0-1 is expanded to the full dimensions of the left input (texdim1).

The chosen coordinates are then treated to stay within the range of texdim0. The mod function is all that is needed to produce a wrapping effect, and wrap coordinates are reversed for the fold.

Finally, one of the coordinate sets is chosen and used to sample the input (tex0). This is copied to gl_FragColor.

Using a shader to control Repos

The shader version of repos is probably a bit more efficient that jit.repos, but jit.repos is so fast on its own that any difference is barely noticeable. The real work of reposing is calculating the control matrices. Moving that task to a shader can produce some really spectacular effects. Here is some code that will provide an X displacement based on the sine of Y and vice versa.

```
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texdim0;
uniform vec2 amt;
uniform float angle;

void main()
{
	float theta = angle * 6.283185;
// make a new X value
	float thetaX = texcoord0.y * theta;
	float newX = texcoord0.x + amt.x * sin(thetaX);
// fold it if necessary
	if(newX > 1.0) newX = 1.0 - (newX - 1.0);
	if(newX < 0.0) newX = -newX;
// make a new Y value
	float thetaY = texcoord0.x * theta;
	float newY = texcoord0.y + amt.y * sin(thetaY);
//fold it if necessary
	if(newY > 1.0) newY = 1.0 - (newY - 1.0);
	if(newY < 0.0) newY = -newY;
// output control as a color
	vec4 repos = vec4(newX ,0.0,1.0,1.0 -newY);

   // output texture
   gl_FragColor = repos;
}
```

This code is not as compact as most shader code, but using extra variables makes it easy to follow. This really does a relative action when the repos shader is in absolute mode. I have to include my own fold function because the output is limited to 0.0 - 1.0.

The trick here is to place the newX and newY into gl_FragColor where the repos shader can find them. Since gl_FragColor is in RGBA mode, a dummy G and B are necessary, with the X value in R and the Y value in A. Note that the newY coordinate is inverted. This compensates for the difference between the qt origin at the upper left and the openGL texture origin at the lower left.

Naturally, there is a lot more to these things, but most of the work from here on in is figuring out what transforms you want to apply and how to code them. With a bit of research and a lot of experimentation you should see fantastic results.

pqe