Adventures in optimization

Doing complex math in expr, gen and pix

Every Max/Jitter programmer hits the wall eventually. By hitting the wall, I mean developing a patch that is just too much computation for one frame of video at 30 FPS. When you hit that wall, the framerate drops, and/or interface actions get sluggish and eventually hang up. Worst case, you have to force quit Max. With each generation of Max, things have gotten better (and computers have gotten better) but there is always another wall to hit.

The basics of optimization can be written down in a list, mostly of things not to do.

- Never use two objects where one will do. The slowest part of max is sending messages from one object to another. Some third party objects (like lobjects) will give a performance boost because they move complex operations into one object. The expr object is faster than a lot of loose math boxes.
- Don't do anything twice in a frame. Many of the help patches are driven by a 2 millisecond metro, but the screen is only redrawn every 15 ms, and every 30 ms is plenty. Convoluted patches often wind up with duplicate actions hidden in the tangle.
- Minimize the use of display objects. A single jit.pwindow can cost 5% in efficiency, and if your patch is littered with them it will be slow. A jit.pwindow in a closed subpatch is still costing something.
- Minimize the use of buttons to convert messages to bangs. These aren't as bad as pwindows, but still slow down the screen draw process. Use a trigger object instead.
- Keep your computer lean and clean. Don't run unnecessary applications, keep 20% of your hard drive clear. Mountain Lion is particularly nasty as it restarts apps that were running when you last shut down.
- Optimize your computer. Everything is memory hungry, especially as we move to 64 bit applications. Add memory if you can, update the graphics card when offered.

Chladni Patterns: a case study.

Chladni patterns are the shapes you get with you sprinkle four on a metal plate and bow it. They show the resonant nodes as you get various harmonics out of the plate.

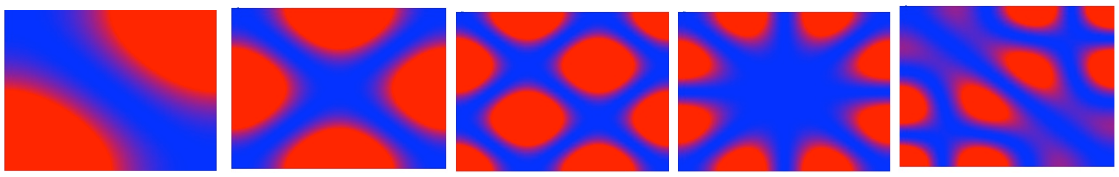


Figure 1. Some basic Chladni patterns

I find these figures interesting, especially if I overlay more than one harmonic at a time.

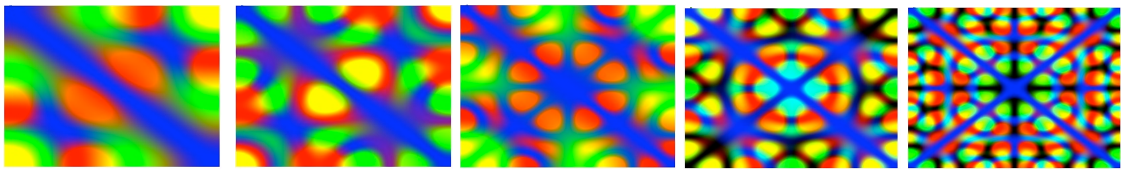


Figure 2. Chladni patterns in red and green

There are several approaches to computing these figures. The patterns in figure 1 and 2 were produced with this formula:

$$\cos(nx\pi/L_x)*\cos(my\pi/L_y) - \cos(mx\pi/L_x) * \cos(ny\pi/L_y)$$

This value is computed for each pixel. Each vibrational mode has two harmonic numbers, n and m, which represent the horizontal and vertical standing waves. The subtraction in the formula could be an addition instead--this has the effect of inverting the phase of the vertical wave. X and Y are the coordinates of the pixel, and Lx and Ly the dimensions of the plate. The use of cosine functions is a result of supporting the plate at the middle. Sine functions can be used instead--this produces a different series of often asymmetrical figures (supported at the edges.)

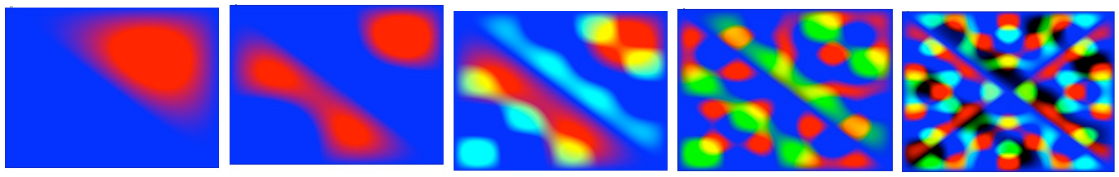
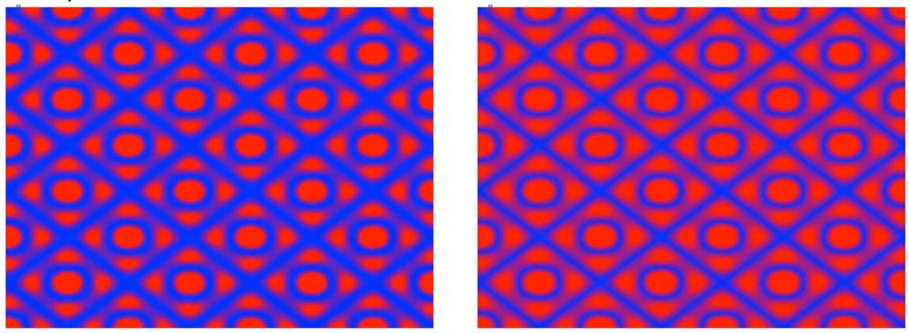


Figure 3.

All of these images are produced with two harmonics, applying one to the red plane and one to the green plane. The blue is calculated as the compliment of the sum of the two figures- in other words, deep blue appears where there is no vibration (nodes).

The values in each pattern were squared to intensify the edges. If squaring is not used, the absolute value must be found.


Output with squaring Output without squaring
Figure 4

Episode 1: Brute force.
Needless to say, this is a heavy computation, especially if you want decent definition. In the original version (which was pre-jitter) it would take nearly a minute to compute a 256 by 256 pattern. Of course this was in 1999 on a G-3 computer.

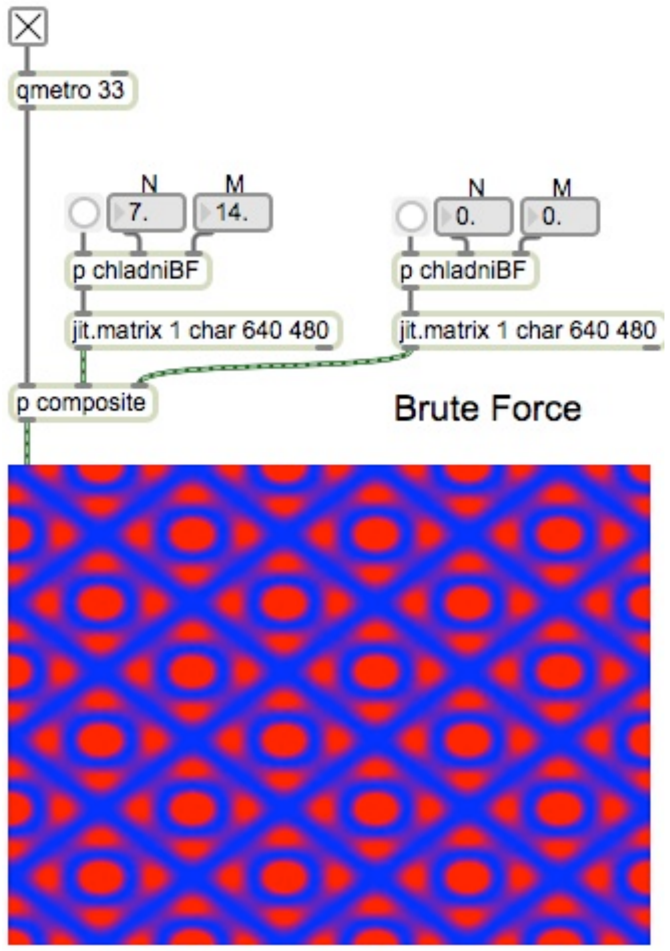


Figure 5.
Figure 5 shows the overall patch used. I compute patterns in two matrices and combine them with the composite subpatch. Figure 6 shows how the combination works. The red and green matrices are assigned to the proper layers with jit.pack.
Inlet 1 is a matrix of all 1s for the alpha layer.
Inlet 2 is the red layer
Inlet 3 is the green layer.
Inlet 4 is computed from the red and green layer. It will be blue where there is no red or green.

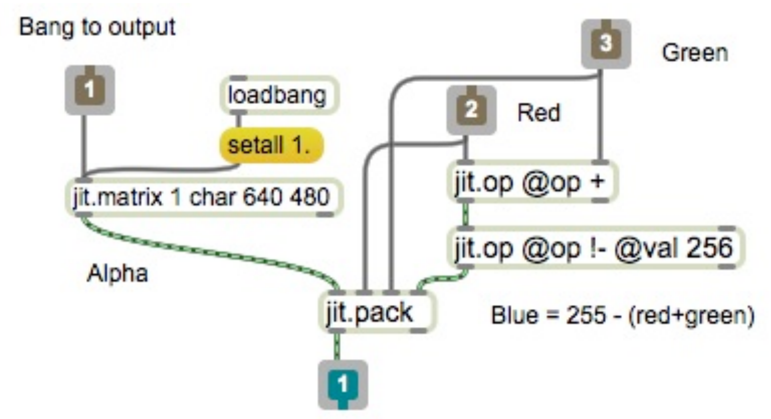


Figure 6.

The real work is in the chladniBF subpatch:

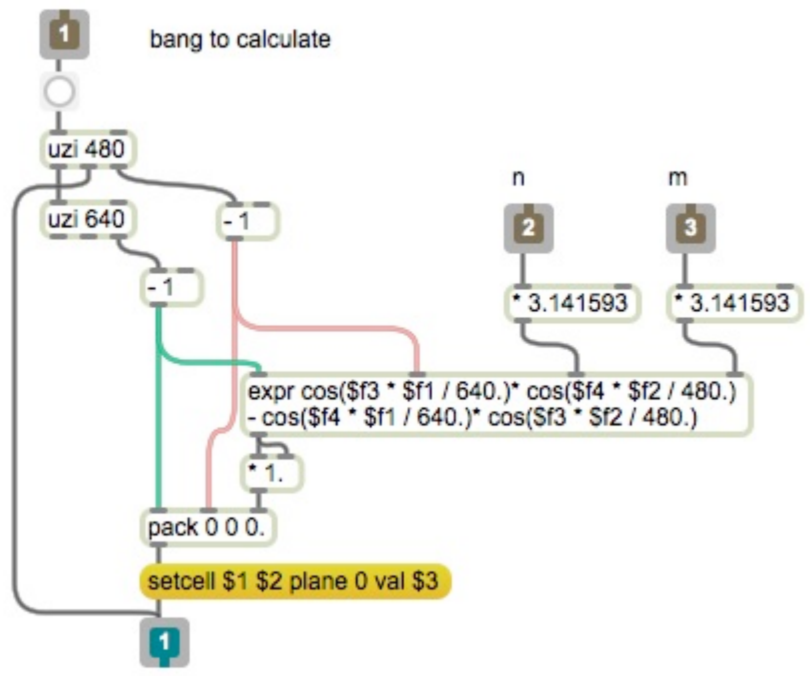

1
bang to calculate
uzi 480
uzi 640
- 1
- 1
n
m
2
3
* 3.141593
* 3.141593
expr cos($f3 * $f1 / 640.)* cos($f4 * $f2 / 480.)
- cos($f4 * $f1 / 640.)* cos($f3 * $f2 / 480.)
* 1.
pack 0 0 0.
setcell $1 $2 plane 0 val $3
1

Figure 7.
The top uzi generates the Y addresses. For each Y 640 X addresses are generated. Each step of the lower uzi triggers the expr object.

And of course the expression is key. Here is an enlarged shot of the expr:

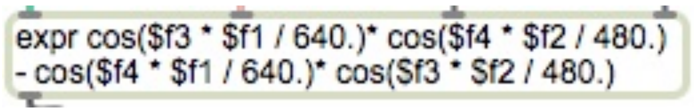

```
expr cos($f3 * $f1 / 640.)* cos($f4 * $f2 / 480.)
- cos($f4 * $f1 / 640.)* cos($f3 * $f2 / 480.)
```

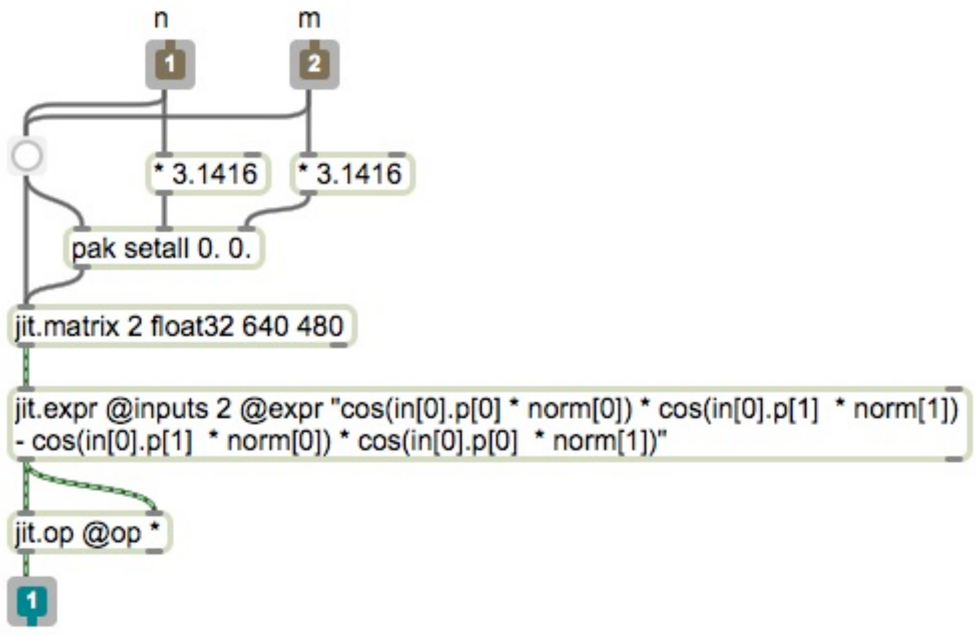
Figure 8.

The expression is pretty close to the formula, with $f1=X, $f2=Y. (We have to subtract 1 from the uzi index to get the address range of 0 to L-1.) I have left the multiplication of n and m by PI outside of the expr so they are only done once. So $f3 = nπ and $f4 = mπ. The L values will never change, so they are constants rather than inputs. Squaring is also outside of the expr, as the most efficient way to square a number is to multiply it by itself.

The addresses and values set cell values in the matrix. Setting individual cells is slow, so it takes nearly 5 seconds to generate a complete image. Upping the resolution to 1080 by 768 nearly triples the number of calculations. This is not what I would call a responsive system.

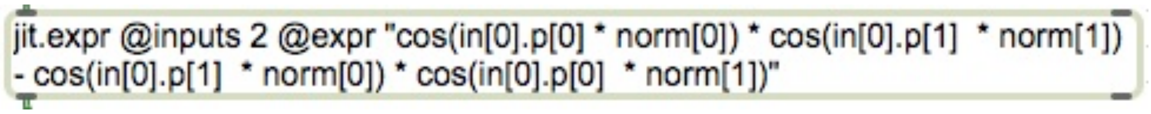
Episode 2. Using jit.expr

Jit.expr allows you to write a piece of code that will be applied to every cell in a matrix. This eliminates the biggest bottleneck in Max, message passing from object to object. In jitter, matrix data is not passed along the patchlines, just the name of a matrix where an object will find data to work on. The jit.expr version of the subpatch in figure 7 is in figure 9.



```
n  m
* 3.1416
* 3.1416
pak setall 0. 0.
jit.matrix 2 float32 640 480
jit.expr @inputs 2 @expr "cos(in[0].p[0] * norm[0]) * cos(in[0].p[1]  * norm[1]) - cos(in[0].p[1]  * norm[0]) * cos(in[0].p[0]  * norm[1])"
jit.op @op *
```

Figure 9.



```
jit.expr @inputs 2 @expr "cos(in[0].p[0] * norm[0]) * cos(in[0].p[1]  * norm[1]) - cos(in[0].p[1]  * norm[0]) * cos(in[0].p[0]  * norm[1])"
```

Figure 10.

Since jit.expr works entirely within a matrix, it needs an appropriately dimensioned matrix to work on. This is created by setting everything on the first plane of a 2 plane matrix to nπ and everything on the second plane to mπ. These are accessed by the operators in[0].p[0] and in[0].p[1]. Jit.expr offers the operators norm[0] to provide X/Lx and norm[1] for Y/Ly. All of this math occurs immediately after the matrix hits the jit.expr object, with no need to step through every address. This process is too fast to measure. Dragging on the float number box produces a smooth transition through intermediate figures. It only gets jumpy if I try to update all four harmonics at once.

Episode 3: Compiling code with Gen

The next step in speed comes with a slight extra charge in Max 6. This is gen. Gen allows you to construct your own object as if you had downloaded the object development kit, spent a month or so learning how to use it, and written C code specific to your task. A gen object is nothing less than a code compiler. The subpatch that contains the jit.gen object is even simpler than what we have seen before. There is a input matrix, but all that shows of jit.gen is the object.

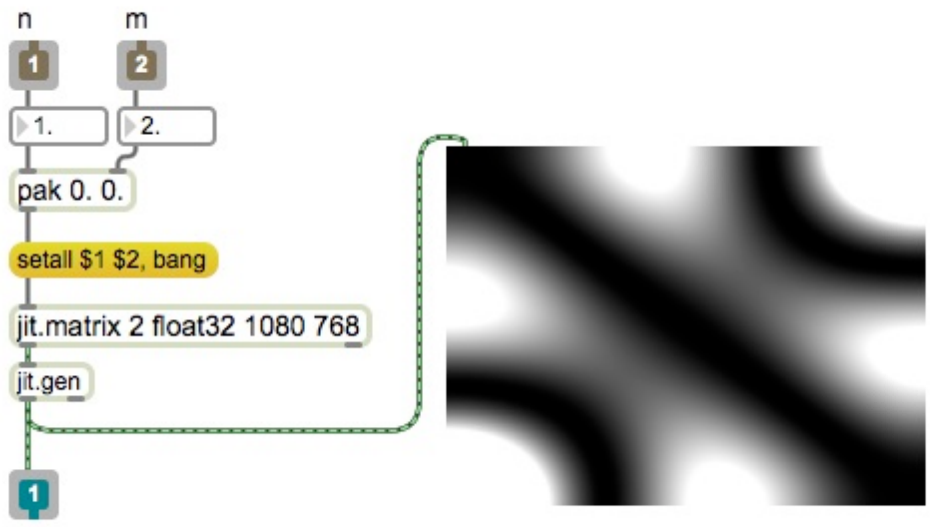
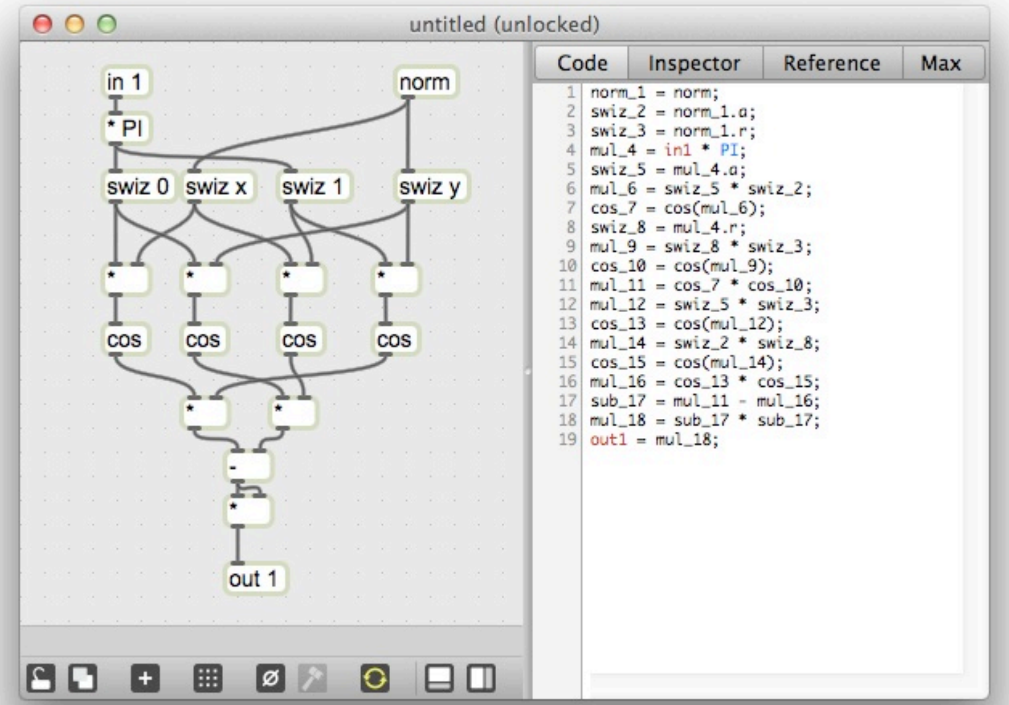


Figure 11.

If you double click on jit.gen, you see the window shown in figure 12.



```
norm_1 = norm;
swiz_2 = norm_1.a;
swiz_3 = norm_1.r;
mul_4 = in1 * PI;
swiz_5 = mul_4.a;
mul_6 = swiz_5 * swiz_2;
cos_7 = cos(mul_6);
swiz_8 = mul_4.r;
mul_9 = swiz_8 * swiz_3;
cos_10 = cos(mul_9);
mul_11 = cos_7 * cos_10;
mul_12 = swiz_5 * swiz_3;
cos_13 = cos(mul_12);
mul_14 = swiz_2 * swiz_8;
cos_15 = cos(mul_14);
mul_16 = cos_13 * cos_15;
sub_17 = mul_11 - mul_16;
mul_18 = sub_17 * sub_17;
out1 = mul_18;
```

Figure 12.

The jit.gen window has two panes- the left holds an ordinary looking patch, and the right shows the equivalent in code. Editing the patch edits the code. This is what gets compiled into a single object.

Gen has its own set of objects (and they are different in the different flavors of gen: gen~, jit.gen, jit.pix and jit.gl.pix). These are principally the operations available in jit.expr and jit.op. There are some odd ones which need explanation.

The cell values are passed in as vectors with all the values found in a single cell. Since the input matrix is 2 layered, the input vector in jit.gen has two values. All of the values in a vector are processed by any object- for instance, the multiply by PI affects both layers of the input.[1] If you want the individual values, as I do here, the swiz operation takes vectors apart. Swiz 0 produces the first value from the vector. You can also refer to the values in a vector as x y z w, or a r g b. The norm operator produces the current address as a vector also, so swiz x and swiz y are appropriate. The other operations should be pretty clear.

[1] It would still be more efficient to multiply n and m by π at the setcell stage. I just put it in here to show how it can be done.

Episode 4: Jit.pix

Jit.pix is an optimization of jit.gen. It is optimized by restricting the type of input it can deal with- only 2 dimensional matrices up to 4 layers may apply. This avoids the auto dimensioning and interpolation overhead as well as simplifying the operators themselves. The jit.pix version of our patch is the simplest looking yet:

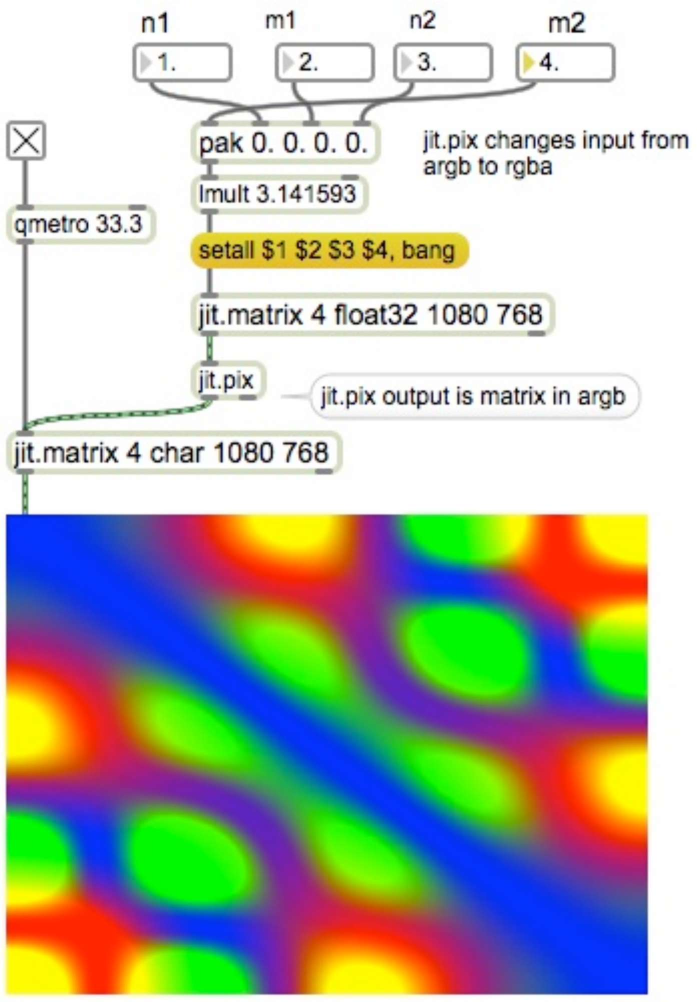

n1
m1
n2
m2
1.
2.
3.
4.
pak 0. 0. 0. 0.
jit.pix changes input from argb to rgba
lmult 3.141593
qmetro 33.3
setall $1 $2 $3 $4, bang
jit.matrix 4 float32 1080 768
jit.pix
jit.pix output is matrix in argb
jit.matrix 4 char 1080 768

Figure 13.

There are no subpatches. Everything will be run from one master 4 plane matrix, and compositing is handled in the gen code. Note that I am multiplying the n and m values by PI outside of the jit.pix object. All code in jit.pix is performed for every pixel. Once jit.pix is compiled, it is fast, but not so fast we can afford to throw basic efficiency to the winds. The inner patch is shown in figure 14.

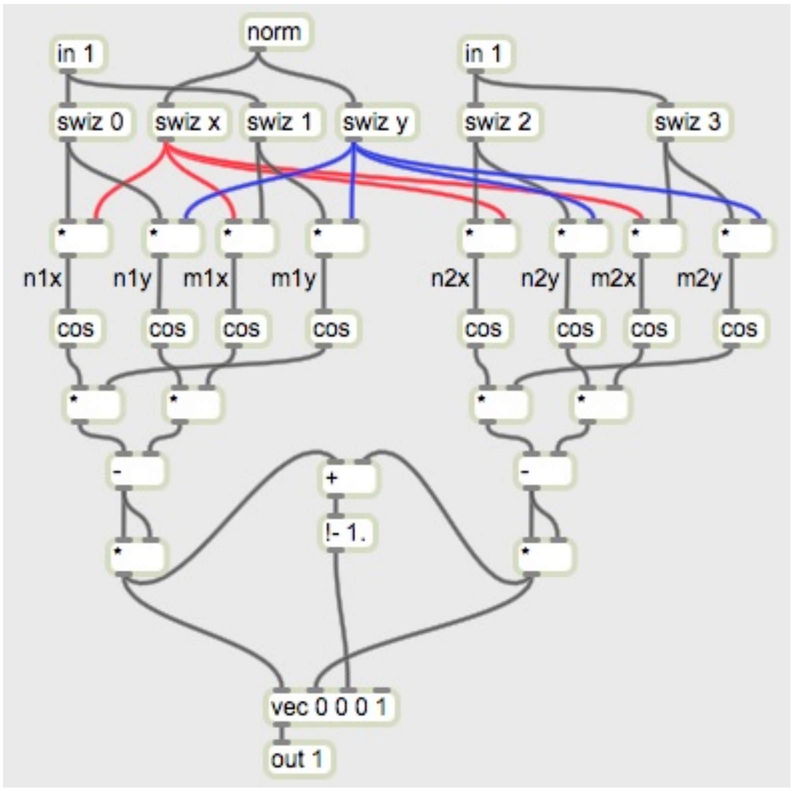


Figure 14.
It's very much like the code in figure 12 but with a code tree for the red image and the green image. The green image uses swiz 2 and swiz 3 for inputs. (Both objects called in 1 refer to the same data.) There is only one gotcha here-- when a matrix is transferred to jit.pix, the planes are rearranged from argb to rgba. This can be dealt with in many ways, but the quick and simple fix is to swap patchlines as shown in figure 13.

The composite function is handled by the objects in the center at the bottom. Vec is an operator that combines single numbers into a vector. Out 1 places each vector into the appropriate cell. Note that on the way out, images are again rearranged, this time from rgba to argb.

Episode 5: The old route to the GPU

The computer is not the most powerful machine sitting on your desk. That honor goes to the computer's graphic processor or GPU. In the tutorial on shaders in Jitter, I show how to run code directly on the GPU using jit.gl.slab. The Cliff notes version goes like this:

- Once an openGL context is set up with a named window and jit.gl.render, a jit.gl.slab object can access the GPU to run code for a custom shader (texture generator).
- Code is written in the glsl language (see www.opengl.org) and embedded in a XML wrapper with a .jxs extension. Among other things, the wrapper binds internal variables to parameter names
- Code is loaded with a read message as in figure 15.

- Parameters within the code are set with the param message.
- The output is a texture, which can only be routed to videoplanes or to other objects via the texture mechanism.

Figure 15 shows a patch to use slab.

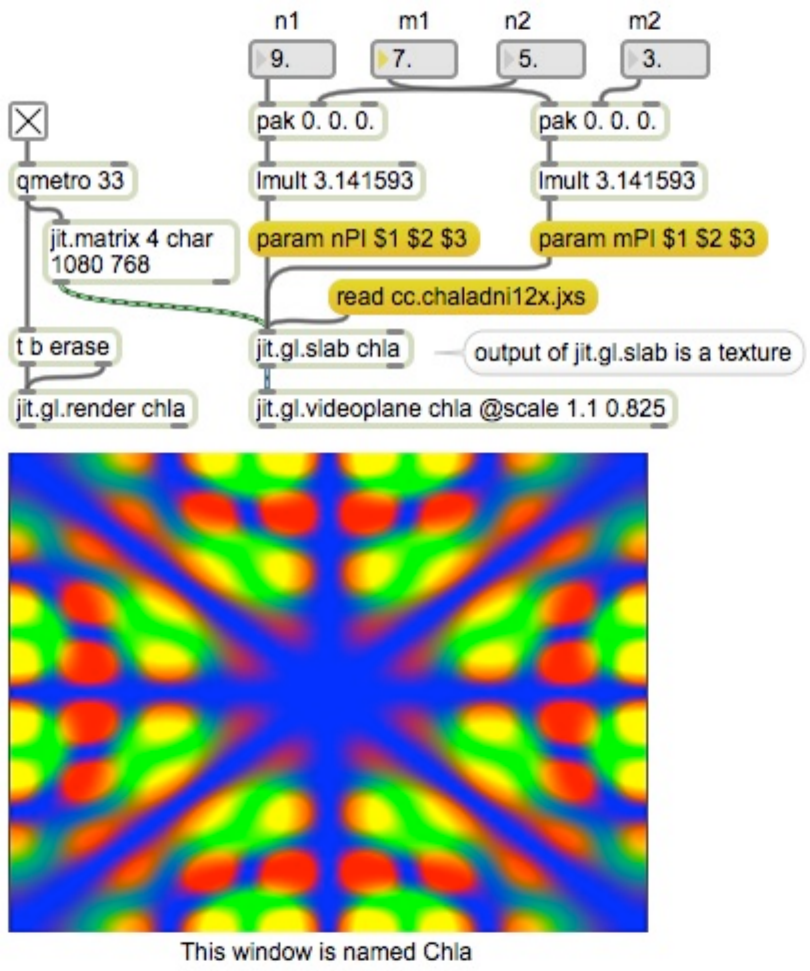


Figure 15.

The param message takes the name of the parameter as its first argument, followed by the parameter values. In this case I named n nPI to remind me that it has already been multiplied by π. nPI contains both n values as a vector.

The code for jit.gl.slab can be written in any text editor. I like to use Texwrangler. The formatting and binding XML code is shown as listing 1.

```
<jittershader name="chladni">
      <description>
      chladni pattern generator
      </description>
      <param name="nPI" type="vec3" default="2.0 2.0 2.0">
            <description>mode n</description>
      </param>
      <param name="mPI" type="vec3" default="1.0 1.0 1.0">
            <description>mode m</description>
      </param>
      <language name="glsl" version="1.0">
            <bind param="nPI" program="fp" />
            <bind param="mPI" program="fp" />
            <bind param="tex0" program="fp" />
            <bind param="tex1" program="fp" />
            <program name="vp" type="vertex"
      source="sh.passthrudim.vp.glsl" />
```

Listing 1.
Note the use of tags in listing one. It is the <*param*> tags that set up parameters. Here there are two, nPI and mPI, which are vectors with 3 values (although I only wind up using 2[2].) The nPI vector contains n values for all three planes, and the mPI vector is similar. The *bind* tags connect parameters to variables as they are listed in the code.

Shaders actually must have two programs in them, one for vertex manipulation and one for fragment (pixel) processing. The *program name* tag with the vertex type loads in a standard vertex routine that passes through the pixel coordinates and texture dimensions. The fragment routine is where the Chladni patterns are generated. It is shown in listing 2. Note the tag ends that follow the code. Leaving these out will produce errors.

[2] I tried layering 3 different Chladni patterns in r g and b, but it was just too messy.

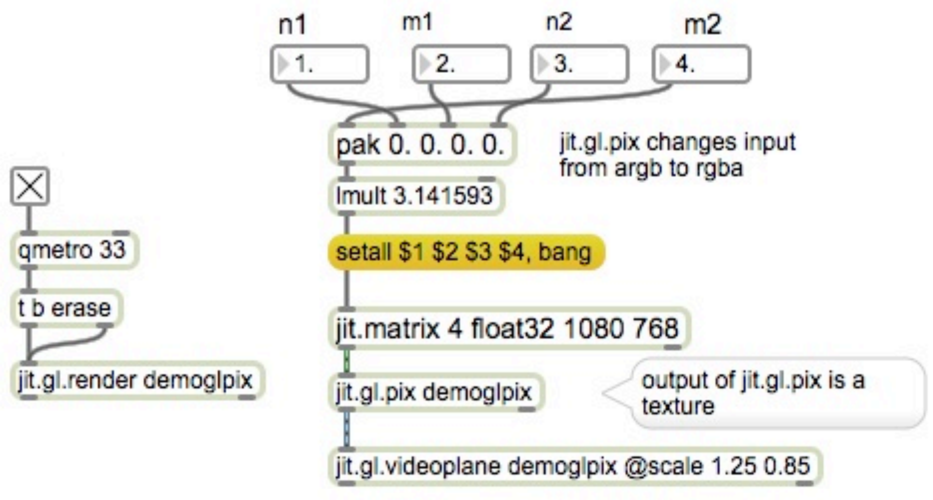
```
            <program name="fp" type="fragment">
<![CDATA[

// Peter Elsea
//UCSC 2013
//Chaladni pattern generator
//setup for 2 textures
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texdim0;

//modes and size
uniform vec3 nPI;
uniform vec3 mPI;

void main()
{

      float normx = texcoord0.x/texdim0.x;
      float normy = texcoord0.y/texdim0.y;
      float rout = cos(nPI.r *normx)*cos(mPI.r*normy);
      rout -= cos(mPI.r *normx)*cos(nPI.r*normy);
      rout = rout * rout;

      float gout = cos(nPI.g *normx)*cos(mPI.g*normy);
      gout -= cos(mPI.g *normx)*cos(nPI.g*normy);
      gout = gout * gout;

      float bout = 1.0 - (rout+gout);
      vec4 mapped = vec4 (rout,gout,bout,1.0);

      // output texture
      gl_FragColor = mapped;
}
]]>
            </program>
      </language>
</jittershader>
```

Listing 2.
Note that all of this code is contained within a *program* tag and is tagged as CDATA. The code in main() does not look much like the code produced by GEN, but does the same thing. Note that operations on vector types like nPI affect all of the values in the vector, unless they are brought out by swizzling- i.e. nPI.r.

The performance of this patch is too fast to measure, even if I up the resolution to full screen size.

Episode 6: The easy way to shaders

We can use GEN to build a shader in jit.gl.pix. It's a lot like the jit.shader patch, but instead of writing glsl code, we build a gen patch in jit.gl.pix.


n1
m1
n2
m2
1.
2.
3.
4.
pak 0. 0. 0. 0.
jit.gl.pix changes input from argb to rgba
lmult 3.141593
setall $1 $2 $3 $4, bang
qmetro 33
t b erase
jit.gl.render demoglpix
jit.matrix 4 float32 1080 768
jit.gl.pix demoglpix
output of jit.gl.pix is a texture
jit.gl.videoplane demoglpix @scale 1.25 0.85

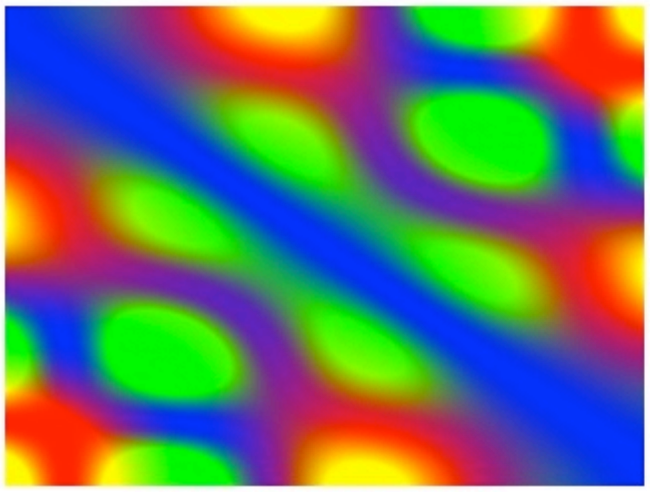


Figure 16.

Figure 16 shows the master patch- this is done in a jit.gl.render context like the jit.gl.slab approach. The n and m values are passed in as a matrix as we did in jit.pix. The patch in jit.gl.pix is in fact the same as the one in jit.pix. (Figure 17.) The difference is in what happens to that patch. Jit.pix compiles the patch into C++ code and runs it on the CPU. Jit.gl.pix compiles the patch into glsl and runs it on the GPU. As far as I can tell the results are identical with my home made shader.

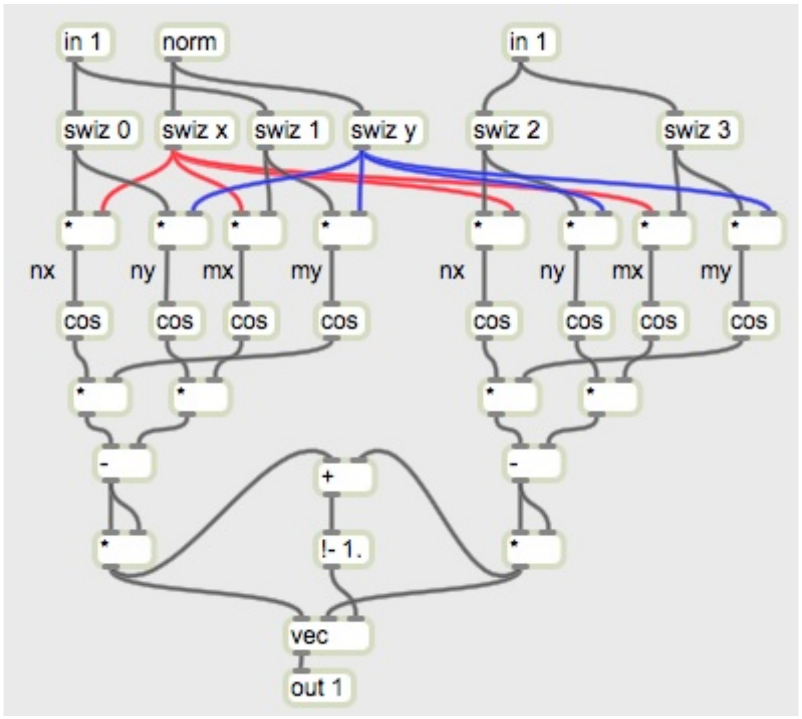


Figure 17.

Conclusion:
My favorite Albert Einstein quote is "Everything should be made as simple as possible, but no simpler." We use Max because it is the most direct way to meet our artistic programming goals. Its main attraction is that simple goals are simple to achieve. When our imagination becomes more ambitious, we ultimately get to the point where the simple functions in Max will no longer do the job. Then we must open one of the "secret closets" in Max to use the more complex and powerful features. At one time, this was a long jump, directly to writing custom objects in C. When Java and JavaScript were added to Max, a lot of power became available with somewhat less intellectual investment. Jitter and jit.expr added a way to manage complex high speed operations within the objects and patchline paradigm. Jit.slab was the first path to the GPU in any programming language short of C++. Finally, Gen has brought the patch paradigm to the highest levels of performance.