The Rip-Off Artist

In the end of 2017, I was running out of material, but I wanted to make a demo to run on a cool SoC: The NVidia Tegra X2. It had 4 useful ARM cores, 2 semi-useful ones, and a 256 Core GPU.

shadertoy.com, an absolutely awesome website, has some of the best GLSL shader coders on the planet, but it seemed that the target GPU at the time was significantly more powerful than the one in the X2. I settled on finding some really good shaders there and tried squeezing them into the 256 measly GPU cores of the X2. It was harder than you'd think. And there was the target to get it done in time for Christmas 2017, leaving around 3 months to do it.

I did my best to abide by the licensing conditions back then. All of the effects, except the Julia quaternions and my raytracer, were based on code from shadertoy.com. The site uses an *Attribution-NonCommercial-ShareAlike 3.0 Unported license*[35]. The original author and link to the ShaderToy page with original source code is listed below. All the modified shaders and control code are still available on my website.

Effect Name	**Link**	**Author**
Galactic Dance	https://www.shadertoy.com/view/XtlGWs	Sinuousity
Glow City	https://www.shadertoy.com/view/XlsyWB	mhnewman
Raytracer		Corneliusen (CC0 1.0 license)
Noise 3D	https://www.shadertoy.com/view/4ddXW4	revers
Julia Quaternions	http://www.cs.cmu.edu/~kmcrane/	Keenan Crane (no license)
More Colorful Than Average	https://www.shadertoy.com/view/ltBcRc	ollj
Seascape	https://www.shadertoy.com/view/Ms2SD1	TDM
Transparent Lattices	https://www.shadertoy.com/view/Xd3SDs	Shane
Twofield	https://www.shadertoy.com/view/Xs2XDV	w23
Torus Thingy	https://www.shadertoy.com/view/llSyWD	bal-khan

Let's find out how little GPU computing power the X2 really has. *GFLOPS* is a synthetic measurement of a situation that never occurs: It assumes every instruction issued is *FMA* (fused multiply add). So, given that the operational units are pipelined, it's starting one *FMA* per clock and completing one *FMA* per clock. *FMA* is 2 basic operations, so somebody had the bright idea to call that 2 operations per clock. Simple to calculate, but not particularly useful.

It's probably more interesting to look at how many, or few, cycles can be given to each pixel per frame and get a grip of how little it was. The GPU was running at 1.3 ghz. The number of cycles available per second would be 1.300.000.000 times 256 = 332.800.000.000. I once had a manager who thought that was an enormous number and cutting-edge. He hadn't been paying attention to PC hardware developments for a long time. Back then I had a graphics card that could do about 5 TFLOPS as defined above. And it had 5 times the ram bandwidth of the X2. Or was it 10? Not important. The difference was gigantic.

Assume the target is to render a 1920x1080 picture 60 times per second. That is a total of 1920*1080*60 = 124.416.000 pixels. Divide the number of cycles by that and the total is 2674 cycles per pixel per frame. That's not a lot, especially if doing really stupid things, like simulating a sea. Let's put that in perspective: A float operation like *FMA* takes about 8 cycles but is fully pipelined. Assuming instruction stalls can be avoided by not using library math functions and unrolling enough, the next hurdle is *if-else* constructs, where everybody takes the cost of both branches, as explained earlier. Then there's the looming presence of running out of registers or the compiler getting confused. Any misses will make the cycle count fall quickly. Very quickly.

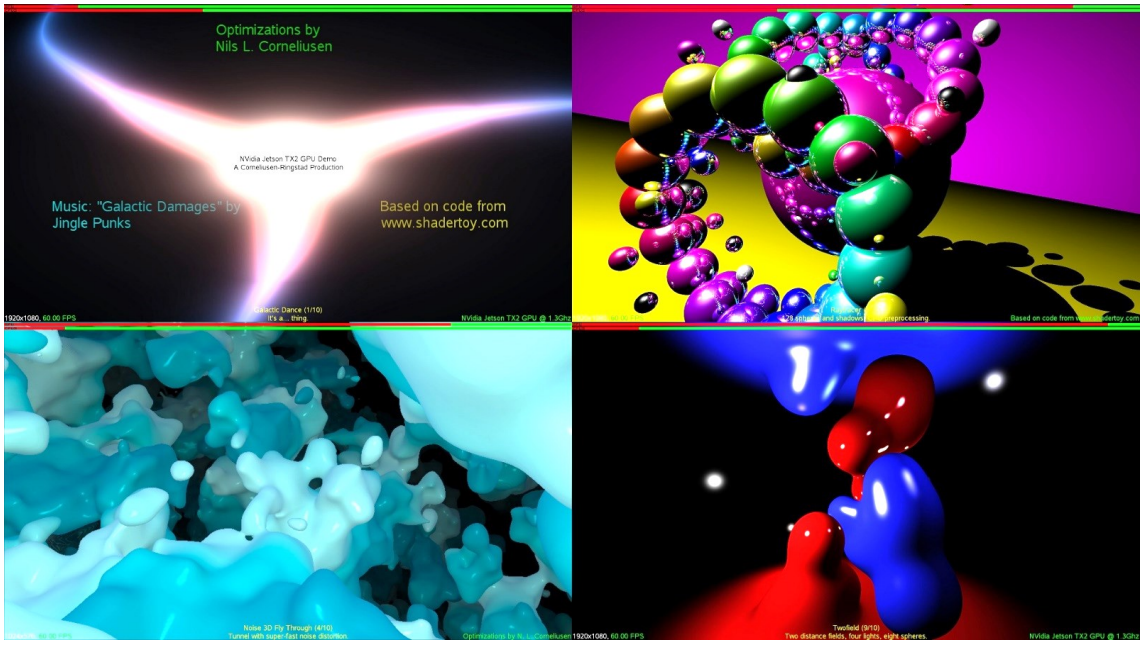
Luckily, shader memory operations could be mostly eliminated, since there wasn't much, except for the display: The output, some overlays, and the occasional transition. That doesn't add up to a lot. Still, it meant that making this demo was doomed from the start. Or was it? Let's find out!

The control code ended up at about 2000 lines. The 11 GPU shaders were a total of 2500 lines. No animals were harmed, but more importantly: No textures were used as part of the effects. It's guaranteed to output 1080p60 with

[35] https://creativecommons.org/licenses/by-nc-sa/3.0/

synchronized music. The music playback had to be removed due to copyright issues but trust me on that one. Total runtime after setup is precisely 5 minutes, so there's 10 effects and each effect runs for 30 seconds. (The 11th shader is a scaler. A low budget bilinear scaler tweaked slightly so the zooms look spiffy. It was the usual crap: The NVidia standard shader is cool unless it's being told to scale from the center on an image with constantly shifting coordinates. I'll get back to that.) It will run and look correct in 1440p or 2160p, but the framerate will drop below 60 now and then. More often now than then on the X2.

One version was rushed out in time for Xmas 2017. It didn't have 60 fps in all the effects. A fixed version was released a bit later, *Nvidia Jetson TX2 Xmas Demo 2017 Remastered*, in April 2018. Ok, so it took a while to fix the 60-fps issue all the way.

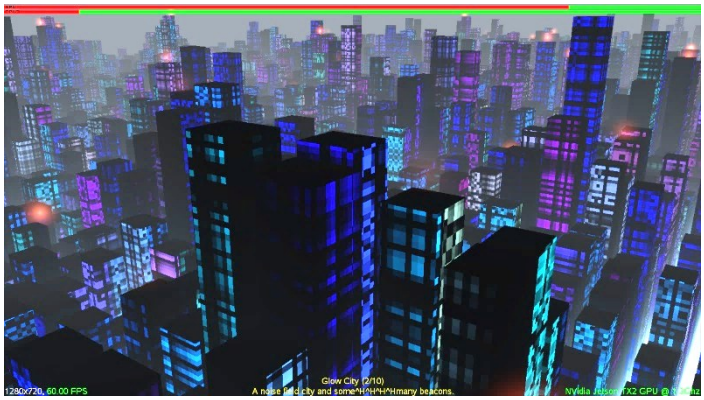


The first tune used in the video is called *Galactic Damages* by Jingle Punks. It was found in the YouTube Audio Library. I'd think such sites were picky about licenses, but it just says, "You're free to use this song in any of your videos.". Yeah, right. Not doing that mistake again. The music was pretty cool, in spite of the license. The second tune is called *Cephalopod* by Kevin MacLeod. A rather hard 90s style techno tune that keeps the blood flowing. I love it! Your mileage may differ.

It's labelled as a Corneliusen-Ringstad production. Tor Ringstad provided invaluable technical information about the X2 and sorted out assorted Android and *Linux* technicalities regarding the output display. 59.94 hz refresh, anybody? Just say no! It's 60 hz or the highway!

As mentioned, the original code for most of these shaders were supposed to run on much more powerful GPUs than the X2. I'd say 6 times the shaders and 10 times the memory bandwidth, and I'm probably being conservative. As such, I guess the authors hadn't bothered maxing them out. And maxing out was needed to make them run in 60 fps on the X2.

Let's review some of the more interesting changes I did back then. The methods should still be applicable to similar problems today. It's strongly recommended to refer to the original source on ShaderToy while reading this chapter. The website is absolutely amazing: Just add some changes and run the shader in the browser. Quite fun!



Glow City - based on code by mhnewman
It's a cityscape with flashing lights on top. The ground is covered in fog. Very eerie.

The unchanged Glow City produced roughly 16 fps on the Jetson TX2, mainly because there's too many buildings and they're too low for the poor X2. So, reduce the building count to 100 and do one iteration of the *j*-loop in *main*. Increase the base height of the buildings and reduce the randomness slightly to cover up the background. And get rid of the fog. Eerie mode slightly reduced. Damn.

The *noise/hash* functions were pretty advanced. The original code looked like this:

```
const float tau = 6.283185;

float hash1(vec2 p2) {
    p2 = fract(p2 * vec2(5.3983, 5.4427));
    p2 += dot(p2.yx, p2.xy + vec2(21.5351, 14.3137));
    return fract(p2.x * p2.y * 95.4337);
}

float hash1(vec2 p2, float p) {
    vec3 p3 = fract(vec3(5.3983 * p2.x, 5.4427 * p2.y, 6.9371 * p));
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.x + p3.y) * p3.z);
}

vec2 hash2(vec2 p2) {
    vec3 p3 = fract(vec3(5.3983 * p2.x, 5.4427 * p2.y, 6.9371 * p2.x));
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.xx + p3.yz) * p3.zy);
}

vec2 hash2(vec2 p2, float p) {
    vec3 p3 = fract(vec3(5.3983 * p2.x, 5.4427 * p2.y, 6.9371 * p));
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.xx + p3.yz) * p3.zy);
}

vec3 hash3(vec2 p2) {
    vec3 p3 = fract(vec3(p2.xyx) * vec3(5.3983, 5.4427, 6.9371));
    p3 += dot(p3, p3.yxz + 19.19);
    return fract((p3.xxy + p3.yzz) * p3.zyx);
}

float noise1(vec2 p) {
    vec2 i = floor(p);
    vec2 f = fract(p);
    vec2 u = f * f * (3.0 - 2.0 * f);
    return mix(mix(hash1(i + vec2(0.0, 0.0)),
                   hash1(i + vec2(1.0, 0.0)), u.x),
               mix(hash1(i + vec2(0.0, 1.0)),
                   hash1(i + vec2(1.0, 1.0)), u.x), u.y);
}
```

It's a noise function where enough randomness is needed, but not total, and not just a simple sine pattern. It can be reduced by using a set of sines with some factors and just tweaking it until it looks ok:

```
vec2 hash2( vec2 p2 )
{
    return vec2( ( sin( p2.x * p2.y ) + 1.0f ) / 2.0f, cos( p2.x * p2.y ) + 1.0f );
}

vec2 hash2( vec2 p2, float p )
{
    return vec2( sin( p2.x ) + 1.0f, cos( p2.y ) * p );
}

float noise1( vec2 p )
{
    return ( ( sin( p.x * 5.78f ) * cos( p.y * 4.23f ) ) + 1.0f ) / 2.0f;
}
```

Increase the beacon probability and change the window weighting. A side effect is that there'll now be colored windows without doing any work. Free stuff for everybody!

```
const float windowSize  = 0.075;
const float windowSpead = 1.0;
(...)
const float beaconProb  = 0.0006;
```

And then it ran at a comfortable 73 fps. It didn't have the gloomy feeling of the original, but it's flashier. I still like flashy stuff.

Noise 3D - based on code by revers

Doesn't everybody love floating blobs? Many floating blobs! The unchanged version delivered 13 fps, again primarily because of heavy noise calculations. A bit too early to call it a trend. The original was:

```
float hash( float h )
{
    return fract( sin(h) * 43758.5453123 );
}

float noise( vec3 x )
{
    vec3 p = floor(x);
    vec3 f = fract(x);

    f = f * f * (3.0 - 2.0 * f);

    float n = p.x + p.y * 157.0 + 113.0 * p.z;

    return mix( mix( mix( hash( n +   0.0), hash( n + 1.0),   f.x ),
                     mix( hash( n + 157.0), hash( n + 158.0), f.x ), f.y ),
                mix( mix( hash( n + 113.0), hash( n + 114.0), f.x ),
                     mix( hash( n + 270.0), hash( n + 271.0), f.x ), f.y ), f.z );
}
```

The only point of interest is that the relation between the hash values remain the same. So, hash value *n+157* has to reappear when *n+0* gets there, otherwise it's just an expensive random function. We're trying to make a tunnel with floating blobs, not random... err, crap.

So: On the CPU, precalculate a table of *vec4* values that's a power of two in size so it wraps nicely. The formula used is the same as in the original, but on a significantly smaller scale. It has just 256 entries, which seems to be enough to generate an interesting result. 128 looked crap.

```
#define TABLELEN  256
#define TABLEMASK (TABLELEN-1)
(...)
    for( int i = 0; i < TABLELEN; i++ ) {
        v4 v;

        v.x = fract( sinf( ((i +   0)&TABLEMASK) * 43758.5453123f ) );
        v.y = fract( sinf( ((i + 157)&TABLEMASK) * 43758.5453123f ) );
        v.z = fract( sinf( ((i + 113)&TABLEMASK) * 43758.5453123f ) );
        v.w = fract( sinf( ((i + 270)&TABLEMASK) * 43758.5453123f ) );
        gpuj->table[i] = v;
    }
(...)
```

Make a new noise function that reads from the table using *LDC.F32X4* before *mix*, which just costs a cool 2 cycles. Take care to handle negative numbers correctly:

```
float noise( vec3 x00 )
{
    ivec3 p00 = ivec3( floor( x00 ) );

    vec3 f00 = fract( x00 );
    f00 = f00 * f00 * (3.0 - 2.0 * f00);

    uint n00 = uint(abs(p00.x + p00.y * 157 + 113 * p00.z));
    vec4 h00 = table[(n00+0u)&uint(TABLEMASK)];
    vec4 h01 = table[(n00+1u)&uint(TABLEMASK)];

    h00    = mix( h00,    h01,    f00.x );
    h00.xy = mix( h00.xz, h00.yw, f00.y );
    h00.x  = mix( h00.x,  h00.y,  f00.z );

    return h00.x;
}
```

Unfortunately, the compiler refused to generate *LRP* (linear interpolate) instructions, unknown for which reason. Interleaving multiple *noise* calls to avoid stalls didn't work very well, since it would deinterleave them to save registers. There's that thing again. Should probably have looked into it. Didn't. No solution this time, either.

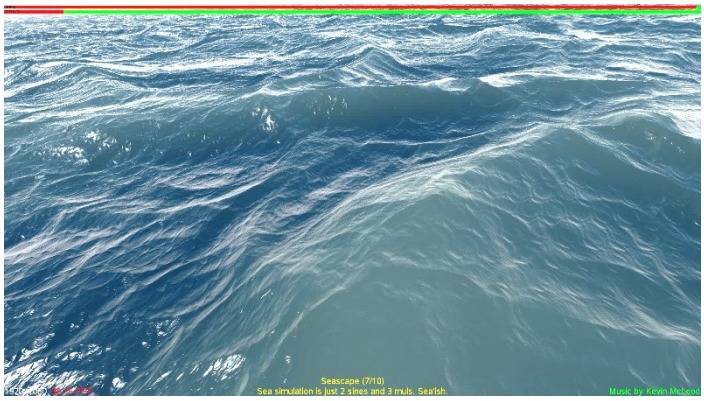
Adding the table-based *noise* function, killing off soft shadows and occlusion, and reducing *depth* to 96 gave 34 fps. A good start!

Unfortunately, the result of all these adjustments was serious visual artifacts. But couldn't just *md* be adjusted on the fly so there's more detail when it's close and less detail further in? Start *MD* at 0.9 and multiply it by 1.025 for each step after the first 24 steps are done. This was the case in several of the shaders. Let's review the idea once. It's quite simple. Or not, the ordering of the *if* statements is also critical. It was imperative back then that the *md* multiplication happened before the *Far* test, for some obscure reason.

```
float castRay( vec3 ro, vec3 rd )
{
    float t = 0.0f;

    float md = MarchDumping;

    for( int i = 0; i < MAX_STEPS; i++ ) {

        float res = map( ro + rd * t );
        if( res < precis ) break;
```

```
        t += res * md;

        if( i > 24 ) md *= 1.025f;

        if( t > Far ) break;

    }

    return t;
}
```

This gave the needed 60 fps and looks almost like the original. Yeah, there's still some artifacts, but it's running in 720p60 on a small SoC, not a discrete GPU.



Seascape - based on code by TDM

Ah, the joy of simulating a sea when unlimited processing power is not an option. The sea is a plane that's distorted by a complex noise function (I'm simplifying a lot here):

```
float hash( vec2 p ) {
    float h = dot(p,vec2(127.1,311.7));
    return fract(sin(h)*43758.5453123);
}

float noise( in vec2 p ) {
    vec2 i = floor( p );
    vec2 f = fract( p );
    vec2 u = f*f*(3.0-2.0*f);
    return -1.0+2.0*mix( mix( hash( i + vec2(0.0,0.0) ),
                              hash( i + vec2(1.0,0.0) ), u.x),
                         mix( hash( i + vec2(0.0,1.0) ),
                              hash( i + vec2(1.0,1.0) ), u.x), u.y);
}
```

What looks almost like that? It's again a complex noise function! Let's call it a trend now. This time it's a collection of sines, so I tried out assorted random crap. It mostly looked like crap with very visible sine patterns. Crap. But I stumbled upon this combination that sort of worked:

```
float noise( in vec2 p )
{
    return ( sin( p.x * 1.91f ) + cos( p.y * 1.71f ) ) * 0.75f;
}
```

That's super cheap and gives it a "sea'ish" look

Next is the *map* and *map_detailed* functions. It's enough to remove one *sea_octave* call to stay marginally above 60 fps with some skyline, so remove the last one in *map* and multiply the first result by 1.4.

I like a straight skyline, so remove the "dir.z += length(uv) * 0.15;" in *main*.

To get constant 60 fps, let's try reducing some more stuff. Move *ang*, *orig*, and *m* calculation to the CPU per frame:

```
        if( d->type == DEMO_SEASCAPE ) {
            float time = gpucommon->frame * 0.3;
            gpusea.ang = v4_set( sinf(time*3.0f)*0.1f, sin(time)*0.2f+0.3f, time, 0.0f );
            gpusea.ori = v4_set( 0.0f, 3.5f , time*5.0f, 0.0f );
```

```
            v2 a1 = v2_set( sinf( gpusea.ang.x ), cosf( gpusea.ang.x ) );
            v2 a2 = v2_set( sinf( gpusea.ang.y ), cosf( gpusea.ang.y ) );
            v2 a3 = v2_set( sinf( gpusea.ang.z ), cosf( gpusea.ang.z ) );

            gpusea.m0 = v4_set(  a1.y*a3.y+a1.x*a2.x*a3.x, a1.y*a2.x*a3.x+a3.y*a1.x, -a2.y*a3.x, 0.0f );
            gpusea.m1 = v4_set( -a2.y*a1.x, a1.y*a2.y, a2.x, 0.0f );
            gpusea.m2 = v4_set(  a3.y*a1.x*a2.x+a1.y*a3.x, a1.x*a3.x-a1.y*a3.y*a2.x, a2.y*a3.y, 0.0f );
        }
```

Untangle *heightMapTracing*. The unchanged version was a bit too complicated:

```
float heightMapTracing(vec3 ori, vec3 dir, out vec3 p) {
    float tm = 0.0;
    float tx = 1000.0;
    float hx = map(ori + dir * tx);
    if(hx > 0.0) return tx;
    float hm = map(ori + dir * tm);
    float tmid = 0.0;
    for(int i = 0; i < NUM_STEPS; i++) {
        tmid = mix(tm,tx, hm/(hm-hx));
        p = ori + dir * tmid;
        float hmid = map(p);
        if(hmid < 0.0) {
            tx = tmid;
            hx = hmid;
        } else {
            tm = tmid;
            hm = hmid;
        }
    }
    return tmid;
}
```

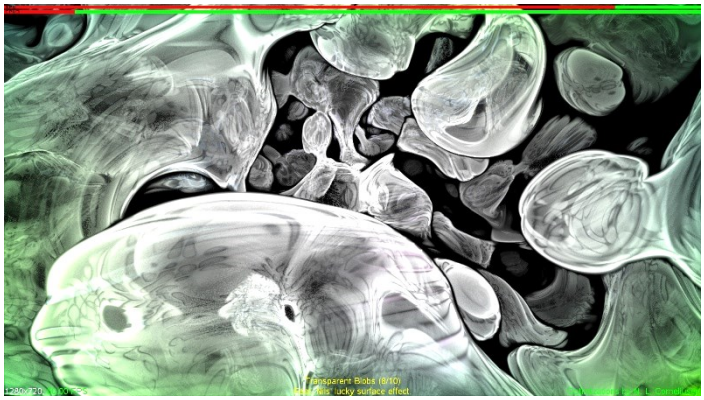
The first *map* call is always the same, so it becomes the much simpler

```
vec3 heightMapTracing( vec3 ori, vec3 dir, out float hx )
{
    vec3 p;
    float tm = 0.0;
    float tx = 1000.0;
    float hm = 1.0f; // ori fixed per frame, so close enough
    float tmid = 0.0;

    for( int i = 0; i < NUM_STEPS; i++ ) {
        tmid = mix( tm, tx, hm / (hm-hx) );
        p = ori + dir * tmid;
        float hmid = map( p );
        if( hmid < 0.0 ) {
            tx = tmid;
            hx = hmid;
        } else {
            tm = tmid;
            hm = hmid;
        }
    }
    return p;
}
```

getPixel can be folded into *main*, the *for* loops stripped out, *hx* moved. The remaining important stuff is then:

```
(...)
    float hx = map( ori.xyz + dir * 1000.0f );
    if( hx > 0.0f ) {
        outColor = vec4( getSkyColor(dir), colscale );
        return;
    }

    vec3 p = heightMapTracing(ori.xyz,dir, hx );
(...)
```

The result is something resembling the original. The sea is much more synthetic, but the patterns look kind of neat. And it delivers 60 fps. A side effect of working with sea simulations is that it'll ruin sea-based movies forever. Is the sea real, or is it simulated? It's annoying having to walk up to the TV and look for patterns all the time.



Transparent ~~Lattices~~ Blobs - based on code by Shane

This one was originally called Transparent Lattices. Unfortunately, the transparent lattices were kind of boring.

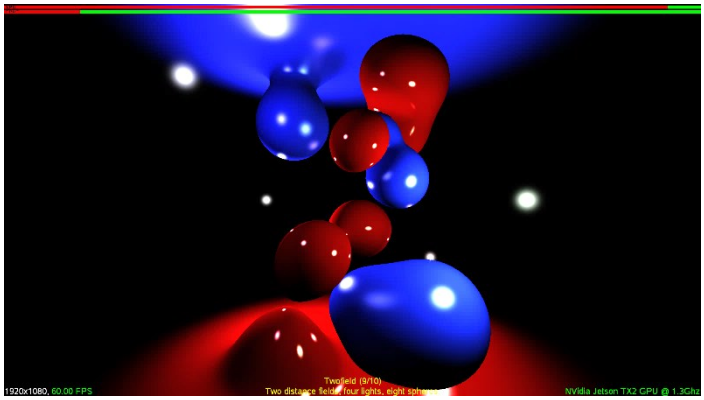
Luckily, the blob code was still in the shader but commented out. Unfortunately, the blobs were also rather boring. They needed something on their surfaces. Let's start with the unchanged *calcNormal* code, except the comment, which is new:

```
// Boring blobs
float map( vec3 p )
{
    p = (cos(p*.315*2.5 + sin(p.zxy*.875*2.5)));

    float n = length(p);

    p = sin(p*6. + cos(p.yzx*6.));

    return n - 1. - abs(p.x*p.y*p.z)*.05;
}

vec3 calcNormal( vec3 p, float d )
{
    const vec2 e = vec2(0.01, 0);
    return normalize(vec3(d - map(p - e.xyy), d - map(p - e.yxy), d - map(p - e.yyx)));
}
```

The *map* calls were unwieldy, so I tried untangling them. It's very easy to do that incorrectly, so I short circuited a couple of them, ran it... and welcome funky blobs with awesome looking surfaces:

```
// Funky blobs
vec3 calcNormal( vec3 p, float d )
{
    vec3 r0 = cos( p * .315 * 2.5 + sin( p.zxy * .875 * 2.5 ) ); p -= 0.01f;
    vec3 r1 = cos( p * .315 * 2.5 + sin( p.zxy * .875 * 2.5 ) );

    vec3 v0 = vec3( r1.x, r0.y, r0.z );
    vec3 v1 = vec3( r0.x, r1.y, r0.z );
    vec3 v2 = vec3( r0.x, r0.y, r1.z );

    float n0 = length( v0 );
    float n1 = length( v1 );
    float n2 = length( v2 );

    v0 = sin( v0 * 6. + cos( v0.yzx * 6. ) );
    v1 = sin( v0 * 6. + cos( v0.yzx * 6. ) ); // oops

    float f = - 1.0f - abs( v1.x * v1.y * v1.z ) * .05;

    n0 = n0 - 1.0f - abs( v0.x * v0.y * v0.z ) * .05;
    n1 = n1 + f;
    n2 = n2 + f;

    return normalize( vec3( d - n0, d - n1, d - n2 ) );
}
```

Never underestimate the power of luck when programming!



Twofield - based on code by w23

It's basically two sets of spheres following a static pattern, being affected by two fields, causing visually pleasing distortions.

The original code produced 26 fps on the NVidia Jetson TX2. The compiler had a rough time following the structure of the *world/w1/w2* etc. calls:

```
// Distance function approximation for the first field only
float t1(vec3 p) {
    float v = 0.;
    for (int i = 0; i < N; ++i) {
        vec3 b = p - b1[i];
        // metaball field used here is a simple sum of inverse-square distances to metaballs centers
        // all numeric constants are empirically tuned
        v += 5. / dot(b, b);
    }
    // add top y=12 (red) plane
    float d = 12. - p.y; v += 3. / (d*d);
    return v;
}

// Second field distance function is basically the same, but uses b2[] metaballs centers and y=-12 plane
float t2(vec3 p) {
    float v = 0.;
         for (int i = 0; i < N; ++i) {
        vec3 b = p - b2[i];
        v += 5. / dot(b, b);
    }
    float d = 12. + p.y; v += 3. / (d*d);
    return v;
}

// "Repulsive" distance functions which account for both fields
float w1(vec3 p) { return 1. - t1(p) + t2(p); }
float w2(vec3 p) { return 1. + t1(p) - t2(p); }

// Combined world function that picks whichever field is the closest one
float world(vec3 p) {
    return min(w1(p), w2(p));
}

vec3 normal(vec3 p) {
    vec2 e = vec2(.001,0.);
    return normalize(vec3(
        world(p+e.xyy) - world(p-e.xyy),
        world(p+e.yxy) - world(p-e.yxy),
        world(p+e.yyx) - world(p-e.yyx)));
}
```

I might be a bit slow, but what's going on there? With a bit of reshuffling and joining and duct-taping it'll look much cleaner. It'll also make the compiler able to produce reasonable code. Again.

```
vec2 t1t2( vec3 p )
{
    float v1 = 0.0f;
    float v2 = 0.0f;

    for( int i = 0; i < N; ++i ) {
        vec3 b1 = p - b1[i].xyz;
        vec3 b2 = p - b2[i].xyz;

        v1 += 5. / dot(b1, b1);
        v2 += 5. / dot(b2, b2);

    }

    float d1 = 12. - p.y; v1 += 3. / (d1*d1);
    float d2 = 12. + p.y; v2 += 3. / (d2*d2);

    return vec2( v1, v2 );
}

float world( vec3 p )
{
    vec2 t12 = t1t2( p );

    float w1 = 1.0f - t12.x + t12.y;
    float w2 = 1.0f + t12.x - t12.y;

    return min( w1, w2 );
}

vec3 normal( vec3 p )
{
    vec2 e = vec2(.001,0.);

    return normalize( vec3( world( p + e.xyy ) - world( p - e.xyy ),
                            world( p + e.yxy ) - world( p - e.yxy ),
                            world( p + e.yyx ) - world( p - e.yyx ) ) );
}
```

Move the trajectory and *O*-calculation to the CPU side for each frame. Also move the color and lightpos tables for later use. The CPU can adjust those for free while rendering a frame and costs roughly 0.5 fps:

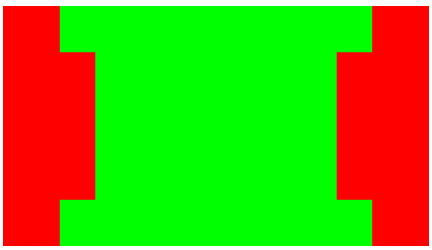
```
        if( d->type == DEMO_TWOFIELD ) {

            // Calculate metaball trajectories
            for( int i = 0; i < NUM; ++i ) {
                float t;

                t = (actual_frame-180)/60.0f * 0.3f;

                float fi = i*0.7f;
                gpu2f.b1[i] = v4_set( 3.7f*sinf(t   +fi), 1.+10.*cos(t*1.1+fi),  2.3*sin(t*2.3+fi), 0.0f );
                fi = i*1.2f;
                gpu2f.b2[i] = v4_set( 4.4f*cosf(t*.4+fi),-1.-10.*cos(t*0.7+fi), -2.1*sin(t*1.3+fi), 0.0f );
            }

            // Move lights
            float fw = gpucommon->frame*0.125f;
            float f0 = fw + 0.25f; f0 = f0 - floor( f0 );
            float f1 = fw + 0.75f; f1 = f1 - floor( f1 );
            float f2 = fw + 0.5f;  f2 = f2 - floor( f2 );
            float f3 = fw + 0.0f;  f3 = f3 - floor( f3 );
            gpu2f.l1pos = rotate( f0, 0.0f, 10.0f ); gpu2f.l1pos.y =  5.0f;
            gpu2f.l2pos = rotate( f1, 0.0f, 10.0f ); gpu2f.l2pos.y = -5.0f;
            gpu2f.l3pos = rotate( f2, 0.0f, 10.0f ); gpu2f.l3pos.y =  0.0f;
            gpu2f.l4pos = rotate( f3, 0.0f, 10.0f ); gpu2f.l4pos.y =  0.0f;

            // Rotation (disabled)
            float tframe = 0.0f;
            float mousex = tframe;
            float mousey = 540.0f;
```

```
            float mx = mousex / gpucommon->width *2.0f-1.0f;
            float my = mousey / gpucommon->height*2.0f-1.0f;
            float a = - mx * 2.0f * 3.1415926535f;
            float s = sinf(a), c = cosf(a);

            gpu2f.o = v4_set( s*20.0f, -my*10.0f, c*20.0f, 0.0f );

        }
```

I never liked the shadows; removing them increases the fps to 43. Adjust the fog-like attenuation from 10-20 to 12-18 and remove the left and right 256 columns for a boost to 58 fps. In *trace*, start *L* at 12.5 instead of 0 for 78 fps, so it's way over target. That meant the number of spheres (*N*) could be increased to 8 and the result is 64 fps. Any more spheres would be too much without changing the paths.

I did some other minor tweaks since I like flashy colors and action: Rotate the light sources around the spheres, make the lights bigger/smaller depending on Z, remove gamma correction, add fade in/fade out, adjust the specular values. Average fps ends up at around 62 fps and the load at 93%.

So, it's running in 60 fps except during the transitions. Let's save a bit more by chopping off even more columns in the far left and right sections. Nothing happens there, anyway. The result is that 37% of the pixels can be eliminated and lots of time saved. Probably simpler to cover the area with high-level OpenGL triangles, but I just stuffed it in the shader:



```
void main( void )
{
    float xedge  = 256.0f/1920.0f;
    float yedge2 = 208.0f/1080.0f;
    float xedge2 = 416.0f/1920.0f;

    if(  (vt.x < xedge  || vt.x > 1.0f - xedge) ||
        ((vt.x < xedge2 || vt.x > 1.0f - xedge2) && vt.y > yedge2 && vt.y < 1.0f - yedge2 ) ) {
         outColor = vec4( 0.0f, 0.0f, 0.0f, colscale );
         return;
    }
    (...)
```

There's a trivial reduction in *lightball* that the compiler wouldn't do. That one was weird: *length* is rather heavy. Reducing it to *dot > L*L* should be trivial for the compiler, but it wouldn't. Change this:

```
vec3 lightball(vec3 lpos, vec3 lcolor, vec3 O, vec3 D, float L) {
    vec3 ldir = lpos-O;
    float ldist = length(ldir);
    if (ldist > L) return vec3(0.);
    float pw = pow(max(0.,dot(normalize(ldir),D)), 20000.);
    return (normalize(lcolor)+vec3(1.)) * pw;
}
```

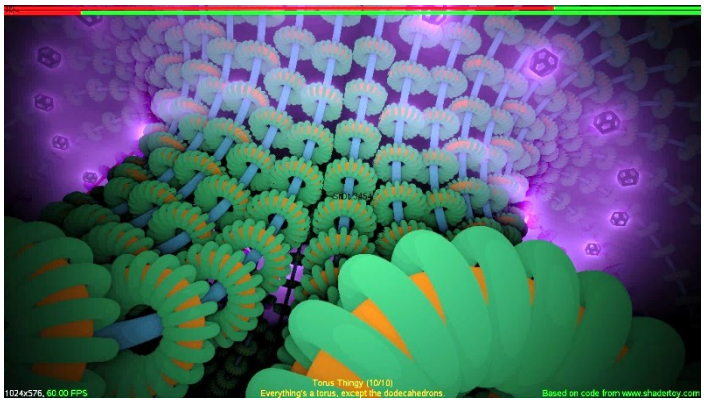
To this:

```
vec3 lightball( vec3 lpos, vec3 lcolor, vec3 O, vec3 D, float L )
{
    vec3 ldir = lpos - O;

    if( dot( ldir, ldir ) > L*L ) return vec3( 0.0f );
```

And adjust the size with this extra *lv* calculation:

```
    float lv = 2.07f - ( ( ( lpos.z / 10.0f + 1.0f ) / 2.0f ) + 1.0f );

    float pw = pow( max( 0.0f, dot( normalize( ldir ), D ) ), 20000.0f * lv );

    return ( normalize( lcolor ) + 1.0f ) * pw;
}
```

And it's 60 fps on the X2. And it's flashy. Cool.



Torus Thingy - based on code by bal-khan
It's a torus that rotates. The torus is built up of tori that's built up of tori... the point should be as clear as tap water. Many tori. And some dodecahedrons (not dodecahedri) floating around. Of course, it wouldn't be the same without.

Start off by removing unused code. Replace *mylength* with standard *length*. No big difference and cheaper. Replace *marc*h with a standard one from noise3d and unroll once:

```
vec2 march(vec3 pos, vec3 dir)
{
    vec2 dist = vec2(0.0, 0.0);

    float esc = 0.01f;

    int i;
    for( i = 0; i < I_MAX; i += 2 ) {
        dist.x = scene( pos + dir * dist.y ); if( dist.x < esc ) break; dist.y += dist.x;
        if( dist.y > FAR ) break;
        dist.x = scene( pos + dir * dist.y ); if( dist.x < esc ) break; dist.y += dist.x;
        if( dist.y > FAR ) break;
    }

    return vec2( float(i), dist.y );
}
```

Yeah, it'll be noisy at distance, but it moves quickly, so it's not particularly important. The dodecahedrons will be a bit fatter too.

Replace *mod* calculations in *scene*:

```
(...)
    p.xy = rotate( p.xy, iTime * 0.25f * ((int(var-0.5f)&1) == 1 ? -1. : 1.) );
(...)
    p.xz = rotate( p.xz, iTime * 1.5f * ((int(var)&1) == 1 ? -1. : 1.) * ((int(vir)&1) == 1 ? -1. : 1.) );
(...)
```

As mentioned earlier, using library math functions in GLSL was horrendously expensive back then. One of the most expensive ones, *atan2*, called *atan* in GLSL, was around 28 instructions that wouldn't get reordered, leading to a never-ending series of stalls. The shader needed many of them, making it very important that it was done quickly.

NVidia's library function was pretty ok, and one possible solution was to convert the assembler version back to GLSL just to get it reordered. But that wouldn't be any fun. It was possible to do better. The guys at IEEE nailed it in the article called *Full Quadrant Approximations for the Arctangent Function*[36]. I converted the example labelled "code 2" to GLSL:

```
// IEEE Signal Processing Magazine ( Volume: 30, Issue: 1, Jan. 2013 )
// Full Quadrant Approximations for the Arctangent Function
// http://ieeexplore.ieee.org/document/6375931/

#define f2u( x ) floatBitsToUint( x )
#define u2f( x ) uintBitsToFloat( x )

float atan2( float y, float x )
{
    uint sign_mask = 0x80000000u;
    float b = 0.596227f;

    // Extract the sign bits
    uint ux_s  = (sign_mask & f2u(x) )^sign_mask;
    uint uy_s  = (sign_mask & f2u(y) )^sign_mask;

    // Determine the quadrant offset
    float q = float( ( ~ux_s & uy_s ) >> 29 | ux_s >> 30 );

    // Calculate the arctangent in the first quadrant
    float bxy_a = abs( b * x * y );
    float num = bxy_a + y * y;
    float atan_1q =  num / ( x * x + bxy_a + num );

    // Translate it to the proper quadrant
    uint uatan_2q = ((ux_s ^ uy_s) | f2u(atan_1q));
    return (q + u2f(uatan_2q)) * PI/2.0f - PI;
}
```

Like most things in life, *atan2* can be done even faster, depending on how much precision is needed. Knut Inge Hvidsten has published some pioneering work about that in the article *Angles, atan2 and taxicabs*[37]:

> *Taxicab angles could be interesting for some applications needing to produce a large number of angles per second, if precision is secondary or if the errors could somehow be absorbed into subsequent use of the angle (table lookup?).*

That solution was a bit too imprecise for this particular problem, but it's an interesting read, nonetheless.

The IEEE guys claimed they did a 4-way SSE2 version of it but did not present any code in the article. I'm sensing a trend in those scientific papers. Weird how often that happens. Trends, I mean. They claimed

> *Therefore, to obtain a fair comparison, a SSE2 version of the functions shown in "Code 1" and "Code 2" has been implemented and benchmarked against Intel IPP, with satisfactory results. It can be noticed that, with the SSE2 implementation, the approximation is nearly three times faster than the Intel IPP proposal*

That only raises more questions than it answers. Code 2 is the 4-quadrant version, so let's go with that. Picking out their performance numbers from the slightly confusing table, I get the following metrics, which is *atan2* per microsecond, for some bewildering reason:

Code 2	140.588
Intel IPP 7.0	218.195
SSE2	741.25

[36] https://ieeexplore.ieee.org/document/6375931

[37] https://www.linkedin.com/pulse/angles-atan2-taxicabs-knut-inge-hvidsten/

It's pretty easy making such statements when no code is shown. To quote a former *Transmeta* employee: "Talk is cheap. Show me the code." If the code were just a raw expansion of "Code 2" to SSE2, it would probably stall a lot. I have no clue what the Intel IPP library did. But ok, that's a factor 5.2, which is pretty neat.

Is it possible to do better? Probably, by just doing even more of them. If the next SIMD instruction depends on the previous one, the going's gonna get slow. To get some throughput, do 2x4 for 8 *atan2* each call, and it'll stall much less. But let's not go down that route. It has been covered already. Let's instead try doing it on the classic Neon. A straightforward 2x4 version of the code above would look like this:

```
#define B    0.596227f
#define PI   3.1415926535f
#define MASK 0x80000000

#define u32_f32 vreinterpretq_u32_f32
#define f32_u32 vreinterpretq_f32_u32

static inline float32x4x2_t satan2_8( float32x4_t y0, float32x4_t x0, float32x4_t y1, float32x4_t x1 )
{
    const float32x2_t v1        = { B,    PI/2  };
    const float32x4_t nan4      = { NAN,  NAN,  NAN,  NAN  };
    const float32x4_t pi        = { PI,   PI,   PI,   PI   };
    const uint32x4_t  sign_mask = { MASK, MASK, MASK, MASK };

    // Extract the sign bits
    uint32x4_t ux_s0 = vbicq_u32( sign_mask, u32_f32( x0 ) );
    uint32x4_t uy_s0 = vbicq_u32( sign_mask, u32_f32( y0 ) );
    uint32x4_t ux_s1 = vbicq_u32( sign_mask, u32_f32( x1 ) );
    uint32x4_t uy_s1 = vbicq_u32( sign_mask, u32_f32( y1 ) );

    // Determine the quadrant offset
    float32x4_t q0 = vcvtq_f32_u32( vorrq_u32( vshrq_n_u32( vbicq_u32( uy_s0, ux_s0 ), 29 ),
                                               vshrq_n_u32(                    ux_s0,   30 ) ) );
    float32x4_t q1 = vcvtq_f32_u32( vorrq_u32( vshrq_n_u32( vbicq_u32( uy_s1, ux_s1 ), 29 ),
                                               vshrq_n_u32(                    ux_s1,   30 ) ) );

    // Calculate the arctangent in the first quadrant
    float32x4_t bxy_a0   = vabsq_f32( vmulq_lane_f32( vmulq_f32( x0, y0 ), v1, 0 ) );
    float32x4_t bxy_a1   = vabsq_f32( vmulq_lane_f32( vmulq_f32( x1, y1 ), v1, 0 ) );
    float32x4_t num0     = vaddq_f32( bxy_a0, vmulq_f32( y0, y0 ) );
    float32x4_t num1     = vaddq_f32( bxy_a1, vmulq_f32( y1, y1 ) );
    float32x4_t atan_1q0 = vmulq_f32(vrecpeq_f32(vaddq_f32(vaddq_f32(vmulq_f32(x0,x0),bxy_a0),num0)),num0);
    float32x4_t atan_1q1 = vmulq_f32(vrecpeq_f32(vaddq_f32(vaddq_f32(vmulq_f32(x1,x1),bxy_a1),num1)),num1);

    // Translate it to the proper quadrant
    uint32x4_t uatan_2q0 = vorrq_u32( veorq_u32( ux_s0, uy_s0 ), u32_f32( atan_1q0 ) );
    uint32x4_t uatan_2q1 = vorrq_u32( veorq_u32( ux_s1, uy_s1 ), u32_f32( atan_1q1 ) );

    float32x4x2_t r01;
    r01.val[0] = vsubq_f32( vmulq_lane_f32( vaddq_f32( q0, f32_u32( uatan_2q0 ) ), v1, 1 ), pi );
    r01.val[1] = vsubq_f32( vmulq_lane_f32( vaddq_f32( q1, f32_u32( uatan_2q1 ) ), v1, 1 ), pi );
```

A bonus: NaN to 0.0 conversion. Might be useful, but usually not. Use it if needed:

```
    // Convert NaN to 0.0f
    uint32x4_t mask0 = vceqq_u32( u32_f32( r01.val[0] ), u32_f32( nan4 ) );
    uint32x4_t mask1 = vceqq_u32( u32_f32( r01.val[1] ), u32_f32( nan4 ) );

    r01.val[0] = f32_u32( vbicq_u32( u32_f32( r01.val[0] ), mask0 ) );
    r01.val[1] = f32_u32( vbicq_u32( u32_f32( r01.val[1] ), mask1 ) );

    return r01;
}
```

How often is it possible to name a function *satan* without insulting anyone? It was hilarious. Right up there with calling a video demuxer *dux*[38], or a buffer holding NAL units *nalle*[39], or a muxer *Mux-O-Matic*[40]. Have been doing too much H.264 video processing lately. Sorry, I'll turn on the air moisturizer. The jokes are too dry.

Anyway, there's also the question of why Intel's IPP implementation sucked, but that ship slid off the edge of the map. Didn't want to spend days figuring out how it worked. And there might be a question about the accuracy. I really don't care about that either, but somebody else might. Heck, if a couple of million *atan2* per second is needed, it probably doesn't matter if answers are a bit, but not too much, off. It's going to be needed very soon.

So, let's get back to the shader under discussion. It needs many *atan2* calls, so the best course of action would be to remove as many as possible. In some places they appear in pairs where the input of the second depends on the result of the first. This optimization is not that easy to spot and approximately correct. A bonus is that it's brutally fast. It all goes to hell when it's not given which side of the torus is displayed: Look closely when it's running. Sometimes the back tori split. Oops. It's close enough most of the time.

The unchanged code:

```
#define TAU PI*2.0f

vec2 modA( vec2 p, float count )
{
    float an = TAU / count;
    float a = atan( p.y, p.x ) + an * 0.5f;
    a = mod( a, an ) - an * 0.5f;

    return vec2( cos( a ), sin( a ) ) * length( p );
}
(...)
    // center spikes
    float var = ( atan( p.x, p.z ) + PI ) / TAU;
    var = var * 40.0f;
    p.xz = modA( p.xz, 40.0f );
```

For almost all, a.k.a. “many”, known values of *var*, this reduces to *PI/4*, so make a simpler *modA* for the second call and use the first result as input:

```
vec2 modAs2( vec2 p, float count, float s2 )
{
    float an = TAU / count;
    float a = s2 + an * 0.5f + PI / 4.0f;
    a = mod( a, an ) - an * 0.5f;
    return vec2( cos( a ), sin( a ) ) * length( p );
}
(...)
    // center spikes
    float s2 = satan2( p.x, p.z );
    float var = ( s2 + PI ) / TAU * 40.0f;
    p.xz = modAs2( p.xz, 40.0f, s2 );
```

And ditto for the orange filler. This...

```
    float vir = (atan(p.x, p.y)+ PI)/(TAU);
    var = vir*30.;
    p.xy = modA(p.xy, 30.)-vec2(.0,.0);
    p.xz -= vec2(4., .0);
    q = vec2(length(p.zx)-0.25, p.y-.0);
    ming = mylength(q)-.05;
    mind = min(mind, ming);
```

[38] From the movie Bloodsport, which is based on the life of martial artist Frank Dux, played by Jean-Claude Van Damme.
[39] Swedish for toy bear.
[40] Ice-O-Matic is an ice machine. Cube-O-Matic a legendary Amiga demo.

...becomes the much simpler:

```
    s2 = satan2( p.x, p.y );
    float vir = ( s2 + PI ) / TAU;
    var = vir * 30.0f;
    p.xy = modAs2( p.xy, 30.0f, s2 );
    p.x -= 4.0f;
    q = vec2( length( p.zx ) - 0.25f, p.y );
    float ming = length( q ) - 0.05f;
    mind = min( mind, ming );
```

Let's follow up with some more reductions. D'oh of the week: *I_MAX* is 100. Sounds rather arbitrary, it never executed 100 iterations anyway. But it will seriously affect the unrolling. The compiler is going to stumble, since there's an *if-break* in there, so a precise estimate is important. Otherwise, the generated code will end up too heavy and sink faster than a lead weight in a swimming pool.

Let's slightly redo the *march* function so it's quicker at a cost of some extra noise. The *log* call isn't really needed. So, we go from this:

```
#define I_MAX 100
(...)
vec2 march(vec3 pos, vec3 dir)
{
    vec2 dist = vec2(0.0, 0.0);
    int i;
    for( i = 0; i < I_MAX+1; i++ ) {
        vec3 p = pos + dir * dist.y;
        dist.x = scene(p);
        dist.y += dist.x;
        float dinamyceps = -dist.x+(dist.y)*(1.0f/1500.0f);
        if( dist.y > FAR || dist.x < dinamyceps || log(dist.y*dist.y/dist.x*(1.0f/100000.0f)) > 0.0f )
break;
    }
    return vec2( float(i), dist.y );
}
```

To this, which is almost the same. Awfully close:

```
#define I_MAX 52
(...)
vec2 march(vec3 pos, vec3 dir)
{
    vec2 dist = vec2(0.0, 0.0);

    float esc = 0.01f;

    int i;
    for( i = 0; i < I_MAX; i += 2 ) {
        dist.x = scene( pos + dir * dist.y ); if( dist.x < esc ) break;
        dist.y += dist.x;                     if( dist.y > FAR ) break;
        dist.x = scene( pos + dir * dist.y ); if( dist.x < esc ) break;
        dist.y += dist.x;                     if( dist.y > FAR ) break;
    }

    return vec2( float(i), dist.y );
}
```

As mentioned, it will fail now and then, and it really needs good, but not great, *atan2* precision. I suggest inserting Mr Hvidsten's super-ultra-fast version mentioned earlier for a trippy experience!

The Control Code and a Trip to Denver

The control code grew into an unwieldy lump of about 2000 lines. The raytracer control code ended up in there, along with, well, everything else. My knowledge of host side OpenGL was not fantastic then, and not now either, so there were lots of trying and failing and finding out that all errors were caused by missing *glBindTexture* calls. I still have no idea what it does, but it seemed to be important. Let's review the structure of it.

The demo was supposed to run on Android, which behaved like a three-legged dog named Lucky. All the useful stuff from Linux had been removed, so I had to resort to the old *syscall* interface. Good going, Google! I had that cool 90s Linux feeling again.

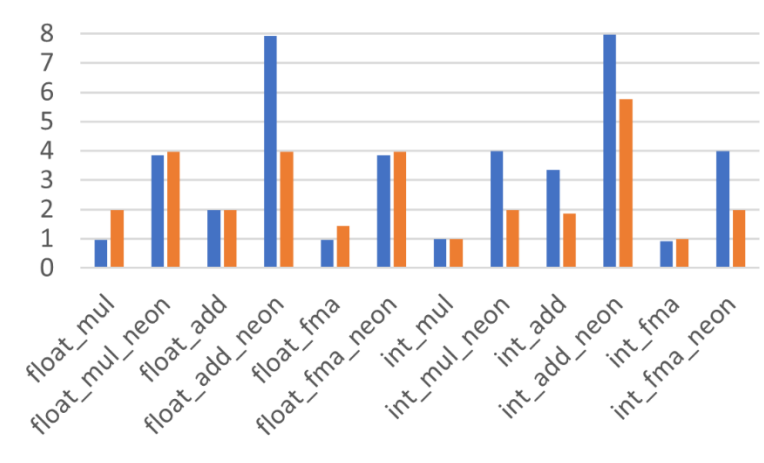
Anyway, a simple operation like *setcore* turned into a pile of rubbish:

```
static bool setcore( int coreid )
{
    int rc;

    uint32_t srcmask = 1 << coreid;
    rc = syscall( __NR_sched_setaffinity, gettid(), sizeof(srcmask), &srcmask );
    if( rc ) {
        printf( "Error syscall setaffinity: syscallres=%d, errno=%d\n", rc, errno );
        return false;
    }

    uint32_t dstmask;
    rc = syscall( __NR_sched_getaffinity, gettid(), sizeof(dstmask), &dstmask );
    if( rc != sizeof(dstmask) ) {
        printf( "Error syscall getaffinity: syscallres=%d, errno=%d\n", rc, errno );
        return false;
    }

    if( srcmask != dstmask ) {
        printf( "setcore error: %d %08x %08x\n", coreid, srcmask, dstmask );
        return false;
    }

    return true;
}
```

setcore was life critical on the X2. If the code ended up running on one of the crap cores, it would have been a total disaster, so I locked it on a normal core.

In case you didn't know, the wise heads at NVidia decided to equip the X2 with 4 normal ARM *A57* cores, and 2 that used super-advanced dynamic recompilation of the running code, known as Denver 2. They had tried that earlier, in the *Tegra K1*, which had Denver 1. It performed way below average.

I tried figuring out what the hell was going on by doing some ultra-synthetic tests in assembler, and the results were crazy. Let's look at instruction issue rate (not to be confused with completion rate, obviously):



Blueish is a regular A57, brownish/orangey is Denver 2. It's Excel default colors. I have no idea why everything has to look like puke by default. Same problem in Visual Studio 2019: Default dark theme has only puke colored elements,

and the background is not even totally black. Seriously, guys! Must be a Microsoft issue, and sometimes Linux: Nobody has forgotten the brown Gnome desktop from the early 2000s, right? The puke color scheme was really popular in Windows 8. When everything has drab colors, the day is gonna be dull. Or maybe the other way around.

Anyway, results were all over the place. **Denver 2 was very fast on bit and integer operations but gave completely anomalous results regarding two very important instructions: fp32 fma and mul** with issue rates of just 1 per cycle. And that's not all: Generic fp32 workloads seemed to trigger a bug in the code translation, which made measuring time completely nondeterministic, and it was even worse if fp32 and Neon instructions were mixed. To make it even more confusing: **Pure 64-bit double workloads were roughly 18% faster than 32-bit float on Denver 2.** Err, what? To quote the game Zero Wing again: "Somebody set up us the bomb."

I had a feeling it was a remnant of TransMeta's ideas: They made some chips back in the early 2000s that tried to emulate x86 chips in hardware. About as stupid as it sounds. Anyway, NVidia had licensed their patents and wanted to put them to use, I guess. Wild guess. Not about the licensing, but about the use.

And, as usual, the question has to be asked: Would it possible to get around these issues and get the Denver 2 cores to behave normally? Signs pointed to yes: The trick was to order the instructions in a particular manner and not mix float and Neon, for example. A fair bit of work was put into that, but it was never completed. It was a time where average managers wanted more average code per hour, and none of this pointless research.

It was lucky that NVidia didn't use that technology again, then! Right. The NVidia *Xavier*, released in 2019, uses a similar system on all 8 cores! They updated the name of the fancy cores to *Carmel*, given that the name Denver had received a few shots across the bow, but was it still crap? Here's a couple of quotes from AnandTech's article *Investigating NVIDIA's Jetson AGX: A Look at Xavier and Its Carmel Cores*[41]:

> *Here NVIDIA's results landed in relatively modest territories, with Carmel landing at around, or slightly higher performance levels of an Arm Cortex-A75.*
>
> *(...)again the rather odd CPU cluster configuration can result in scenarios where not all eight cores are able to perform at their peak performance under some circumstances.*

I rest my case. But if everything evens out and I win the lottery, I might order an Xavier soon. For all I know, they might actually have fixed the code translation, though the AnandTech quotes are not promising.

Let's get back on track. There are some averaging scalers in the control code for rendering it in higher resolution and downscaling it for a high-quality video. One generic, and one ARM specific. The ARM specific one is completely straightforward and simple. Almost. Remarkably close, trust me. Let's have a look at it.

Get rid of *vreinterpret* and *vget* crap:

```
#define qu64u8  vreinterpretq_u64_u8
#define qu64u16 vreinterpretq_u64_u16
#define u8u64   vreinterpret_u8_u64
#define u16u64  vreinterpret_u16_u64
#define u8u16   vreinterpret_u8_u16

#define getq_u64 vgetq_lane_u64
```

[41] https://www.anandtech.com/show/13584/nvidia-xavier-agx-hands-on-carmel-and-more/5

Make a scale function, assume stuff is aligned etc. 4 rows are read at a time since it's a factor 4 average downscaler:

```
void scale_factor_4( uint32_t *src, int srcw, int srch, uint32_t *dst )
{
    int outwidth  = srcw >> 2;
    int outheight = srch >> 2;

    for( int y = 0; y < outheight; y++ ) {

        uint32_t *srcrow0 = __builtin_assume_aligned( src + (y+0) * 4 * srcw, 8 );
        uint32_t *srcrow1 = __builtin_assume_aligned( srcrow0 + srcw, 8 );
        uint32_t *srcrow2 = __builtin_assume_aligned( srcrow1 + srcw, 8 );
        uint32_t *srcrow3 = __builtin_assume_aligned( srcrow2 + srcw, 8 );

        uint32_t *dstrow = __builtin_assume_aligned( dst + y * outwidth, 8 );

        for( int x = 0; x < outwidth; x += 2 ) {

            uint8x16_t r00, r01, r02, r03;
            uint16x8_t s01h, s01l, s23h, s23l;
            uint16x8_t shl;
```

Since it's 32-bit pixels, 4 pixels*4=16 bytes are read from each row:

```
            r00 = vld1q_u8( (void *)srcrow0 ); srcrow0 += 4;
            r01 = vld1q_u8( (void *)srcrow1 ); srcrow1 += 4;
            r02 = vld1q_u8( (void *)srcrow2 ); srcrow2 += 4;
            r03 = vld1q_u8( (void *)srcrow3 ); srcrow3 += 4;
```

Add up everything with an old friend: *vaddl*. And an army of reinterprets and crap, since it's, well, ARM. Picture this code without the macros and strict code formatting rules.

```
            s01l = vaddl_u8( u8u64( getq_u64( qu64u8( r00 ), 0 ) ), u8u64( getq_u64( qu64u8( r01 ), 0 ) ) );
            s01h = vaddl_u8( u8u64( getq_u64( qu64u8( r00 ), 1 ) ), u8u64( getq_u64( qu64u8( r01 ), 1 ) ) );
            s23l = vaddl_u8( u8u64( getq_u64( qu64u8( r02 ), 0 ) ), u8u64( getq_u64( qu64u8( r03 ), 0 ) ) );
            s23h = vaddl_u8( u8u64( getq_u64( qu64u8( r02 ), 1 ) ), u8u64( getq_u64( qu64u8( r03 ), 1 ) ) );
```

That gives a set of high/low 16-bit results. Add them up:

```
            shl = vaddq_u16( vaddq_u16( s01l, s23l ), vaddq_u16( s01h, s23h ) );
```

Shift the answers down. What do we know now? The high byte in each 16-bit unit is empty. Useless, really. Remember that.

```
            uint16x4_t p00 = vshr_n_u16( vadd_u16( u16u64( getq_u64( qu64u16( shl ), 0 ) ),
                                                   u16u64( getq_u64( qu64u16( shl ), 1 ) ) ), 4 );
```

Do the same for the next set of 4 pixels:

```
            r00 = vld1q_u8( (void *)srcrow0 ); srcrow0 += 4;
            r01 = vld1q_u8( (void *)srcrow1 ); srcrow1 += 4;
            r02 = vld1q_u8( (void *)srcrow2 ); srcrow2 += 4;
            r03 = vld1q_u8( (void *)srcrow3 ); srcrow3 += 4;

            s01l = vaddl_u8( u8u64( getq_u64( qu64u8( r00 ), 0 ) ), u8u64( getq_u64( qu64u8( r01 ), 0 ) ) );
            s01h = vaddl_u8( u8u64( getq_u64( qu64u8( r00 ), 1 ) ), u8u64( getq_u64( qu64u8( r01 ), 1 ) ) );
            s23l = vaddl_u8( u8u64( getq_u64( qu64u8( r02 ), 0 ) ), u8u64( getq_u64( qu64u8( r03 ), 0 ) ) );
            s23h = vaddl_u8( u8u64( getq_u64( qu64u8( r02 ), 1 ) ), u8u64( getq_u64( qu64u8( r03 ), 1 ) ) );

            shl = vaddq_u16( vaddq_u16( s01l, s23l ), vaddq_u16( s01h, s23h ) );

            uint16x4_t p01 = vshr_n_u16( vadd_u16( u16_u64( getq_u64( qu64_u16( shl ), 0 ) ),
                                                   u16_u64( getq_u64( qu64_u16( shl ), 1 ) ) ), 4 );
```

vuzp de-interleaves bytes in each half, so one half will contain interesting stuff, and the other crap. Lucky the high bytes were useless. Naah, not really. Not much is based on luck in this business.

```
            uint8x8x2_t pout = vuzp_u8( u8_u16( p00 ), u8_u16( p01 ) );

            vst1_u8( (void *)dstrow, pout.val[0] ); dstrow += 2;

        }
    }
}
```

Wouldn't it have been wise to process even more bytes? Yes, probably, but there was the register drain, and there was a perfectly good reason the rows were 64-bit aligned only: Earlier versions could be run in even lower resolutions, and the frames returned from Android were always 64-bit aligned.

vuzp (de-interleave) and *vzip* (interleave) deserve a mention. The assembler instructions take two parameters only, so they'll need to read from the registers, shuffle them, and write back to the same registers. I wonder how they handled that internally because the latency is rather high. If the same can be accomplished with a couple of *vtrn* - transpose - instead, it's always cheaper. AArch64 Neon fixed that problem by removing them and getting Tilera-style versions instead. But they were fun instructions!

That leads to the obvious question: Why didn't I use AArch64 Neon intrinsics there? Well, there were two compilers available for Android back then. One could do AArch64 Neon but wouldn't compile the display stuff and vice versa. I tried compiling parts with different compilers and linking it and spent a day debugging the linker. Gave up. Wasn't critical.

Let's get back to the control code. Didn't find any straightforward way to represent the shader common data on the host side, so I made one common struct for those that don't have anything and just replicated it.

```
typedef struct {
    float width, height;
    float frame;
    float colscale;
} gpu_default;

typedef struct {
    float width, height;
    float frame;
    float colscale;

    v4 noisetable[TABLELEN];
} gpu_noise3d;

    // ... etc ...
```

The host side control blocks. I didn't bother much with the amount of data. It's unimportant, just had to find the biggest one. Guess who won? It's always the raytracer, so just copy that much data every frame. Doesn't matter much in terms of execution speed.

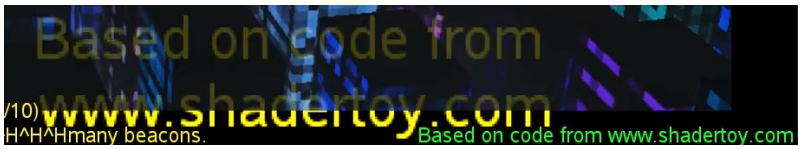
```
    // Gpu data blobs
    void *gpudata = NULL;
    int gpudatalen = sizeof(gpu_tracer); // largest

    gpu_default    gpudef;
    gpu_noise3d    gpu3d;
    gpu_twofield   gpu2f;
    gpu_quat       gpuquat;
    gpu_tracer     gputracer;
    gpu_seascape   gpusea;

    gpu_default *gpucommon = NULL;
```

The loading of shaders and their compilation and linking is trivial. Probably found it somewhere. I'm guessing on the internet. Not completely trivial, just hundreds of lines of terminally boring code.

Finally! The main loop, but first setcore has to be called, with an appropriate comment:

```
    setcore( 3 ); // stay off Denver2, sucky floating point
(...)
    while( true ) {
```

Next came all the fiddly stuff with keeping track of which shader to display etc. And control the zooming, and make sure the text between Galactic Dance and Glow City really moves out of the buffer. Yeah, it was supposed to be that way.




```
        if( d->type != DEMO_GALDANCE && demoframe == demo_fadetime ) {

            // new bg texts after fade done
            gen_bg( bgbuf, bgw, bgh );

        } else if( d->type == DEMO_GALDANCE ||(d->type == DEMO_GLOWCITY && demoframe <= demo_fadetime-1) ) {

            // generate credits in galdance and the fade to glow city every frame
            if( actual_frame > 84 ) { // really
                gen_credits( bgbuf, bgw, bgh, actual_frame );
            }

        }
```

A lot of effort was put into making the transitions look good, hence the earlier mentioned scaler. If it's a transition, first draw the background text into a full-size buffer, then the shader in the middle with correct alpha, and finally the overlapping text. Then cut from the center of that and scale it in place. A lot of work, as mentioned earlier, but personally I think it turned out great.

It was sort of fiddly calculating the zoom values. The comment from back then was spot-on:

```
        // Zoom in/out
        // Seriously, who writes this crap? Please go stand in the corner.
        int ww = winwidth, wh = winheight;
        if( demoframe <= demo_zoomtime ) {
            // zoom in width..winwidth
            // width..winwidth
            ww = winwidth  + (bgw - winwidth )*(demo_zoomtime-demoframe)/demo_zoomtime;
            wh = winheight + (bgh - winheight)*(demo_zoomtime-demoframe)/demo_zoomtime;
        } else if( demoframe >= demo_runtime-demo_zoomtime ) {
            // zoom out winwidth..width
            int f0 = demoframe - (demo_runtime - demo_zoomtime);
            ww = bgw - (bgw - winwidth )*(demo_zoomtime-f0)/demo_zoomtime;
            wh = bgh - (bgh - winheight)*(demo_zoomtime-f0)/demo_zoomtime;
        } else {
            ww = winwidth;
            wh = winheight;
        }

        // shrink window a bit, avoid edge artifacts
        if( ww != width  ) ww -= 2;
        if( wh != height ) wh -= 2;
```

That's how code ends up looking when you're running out of time. And if it works, nobody's going to bother fixing it later. It worked. Nobody fixed it. What a surprise.

Next up was the shader precalc per frame bits. It has been covered earlier. The entire display was drawn in a *Frame Buffer Object (FBO)*. I know it wasn't recommended, but it also worked, so nobody fixed it. It would probably have been wiser to render it directly to a texture.

The control code was single-threaded and locked on core 3, and the CPU load bar only shows that core. It's interesting to note that the load increases when running the raytracer, since it does the probing, as explained earlier, but not by a lot. Plenty of CPU headroom.

An interesting observation:

```
if( display )
    eglSwapBuffers( gl.display, gl.surface );
else
    glReadPixels( 0, 0, 1, 1, GL_RGBA, GL_UNSIGNED_BYTE, debugbuf ); // ! glFinish() too smart
```

If the code was run with the display turned off and no saving of images, *glFinish* would do nothing. Yeah, ok, but it ruined test runs. Sort of like when the optimizer removes all the code because the answers aren't used. So, I had to force it by actually reading back a pixel.

One part that's missing is the audio playback code. The two music pieces were just glued together and played back with some code that had a stingy license, so it had to be removed. Sorry. Just play them with a random player. It wasn't more advanced than that.

It would have been nice to try to get it running on an Xavier... in 4K. Just have to win the lottery first.

Unfortunately, I didn't have a HDMI recorder for recording the video, so it could be run in a mode where it saved the CPU and GPU load to two text files. These could be read back on the next pass and inserted into the load bars while saving each frame separately. Took a long time, but the video was at least a faithful reproduction of the original. Which was a rip-off. Time to do some new stuff.