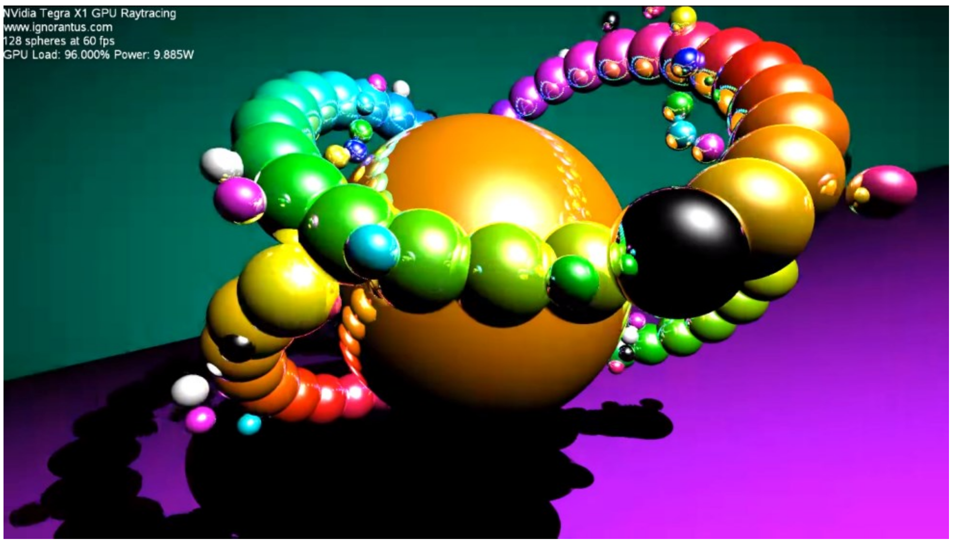
The GPU Raytracer

Two great movies were released in 2017: *Blade Runner 2049* and *Alien: Covenant*. Both were sequels, of sorts. Covenant of *Prometheus* from 2012, supposed to be a new direction in the Alien universe, and '2049 of the original *Blade Runner* from 1982, one of the greatest movies ever made. They have to be evaluated on their own: Comparisons to the originals can't be justified. I liked the eerie feeling in Covenant, and the filming in '2049 is simply great.

2017 was also the year I decided to dig deeper into GPUs. Sequels were out of the question, so no more Julia quaternions or regular fractals for the time being. I had worked on a GPU raytracer in the background for a long time, trying out stuff, unrolling code, formatting *if* statements, getting rid of *if* statements, playing with floats as integers, laughing at how *bool* worked in GLSL, the usual. So, it was time to set a target: 128 spheres with reflections and a plane with shadows at 60 fps in 1080p resolution. And a sky. On an NVidia Tegra X1.

As mentioned, I didn't have the proper development tools and such available. Much of what I wrote for GPUs back then was guesswork, meaning trying random crap until it runs faster. The generated GPU code was a bit too high-level, so it was sometimes hard to guess what was going on. I didn't have any profiler either. I had an editor, a compiler, a way to extract generated assembler code from the GPU, and that's all that's needed. Or so the saying goes. Not sure which saying that might be, but there you go.

I started out with a raytracer for the *Epiphany* 16-core CPU, usually sold on a *Parallella* board, written by Shodruky Rhyammer called *Blobubska*[27]. It was pretty simple and nicely structured. The last version of my raytracer presented here has no remnants of it, but I still recommend looking at his work. And on the Parallella board. Awesomely cool CPU.


NVidia Tegra X1 GPU Raytracing
www.ignorantus.com
128 spheres at 60 fps
GPU Load: 96.000% Power: 9.885W

Raytracers are deceptively simple: Place the viewer somewhere, draw a screen in front, draw lines from the viewer through points on the screen, see if it hits anything. If it does, reflect and check for hit again. There's a certain logic to it, neatly explained in the article *Ray Tracing: Graphics for the Masses*[28] by Paul Rademacher which I highly recommend.

That approach didn't work very well on the X1. Around 80 spheres seemed to be the theoretical maximum with such a simple approach. I had the fantastic (or stupid, I forget which) idea of outsourcing some of the work to the ARM cores, thus making the GPU job simpler.

Let's start off by making some useful definitions. These are the structures used by the CPU control code, and it should be pretty obvious what it is: Some colored spheres, a world with spheres and a plane, and that's all that's

[27] https://github.com/shodruky-rhyammer/blobubska
[28] https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

needed. Would it be possible to do that without virtual functions, subclasses, concepts, modules, and lambda functions? Err... sorry, was sidetracked by some crap toy language there. Let's keep it simple, like always.

Define a sphere:

```
typedef struct {

    v4    obj_pos;
    v4    obj_col;
    v3    poseye;
    float radsq;
    float rv;
```

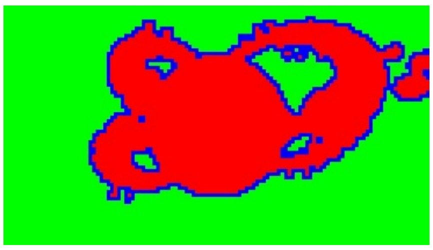
Movement information. Will be covered later.

```
    int   orbit_center; // id of object to rotate around
    float orbit_radius;
    float orbit_tilt;   // xz..yz plane tilt, -PI..0..PI
    int   orbit_speed;  // frames for one full rotation
    int   orbit_offset;
} sphere;
```

worldinfo contains the spheres and plane and such:

```
typedef struct {
    int largecnt;
    int smallcnt;

    v3 light_pos;
    v3 eye;

    sphere spheres[OBJNUM];

    v4 plane_norm;
    v4 plane_col;

    int frame;

    float ax;
    float ay;

    int width, height;
    int stride;
    float swidth, sheight;
} worldinfo;
```

Instead of going straight to the nitty-gritty details of the raytracer, let's consider the CPU simplification first. It will make understanding the raytracer later easier.



The spheres have a defined minimum size, so it's enough to check the corners of each 16x16 block on the CPU to see if something is there. The GPU can then eliminate large swathes of code by knowing if the area is covered only by a sphere or the background. In the illustration above, green is background only, red is spheres only, and blue is a combination.

Fair enough. Let's define a function, *world_probe*, which does just that. First, let's try sorting the spheres, so early cut-off can be done more efficiently. Or that's the idea. Seemed to work ok. The only interesting details to be sorted are the positions and colors:

```
typedef struct {
    v4 obj_pos;
    v4 obj_col;
} poscol;
```

The sorting function just compares the depth:

```
static int sortfunc( const void *a, const void *b )
{
    poscol *va = (poscol *)a;
    poscol *vb = (poscol *)b;

    return va->obj_pos.z - vb->obj_pos.z > 0.0f ? 1 : -1;
}
```

world_probe does the work:

```
static void world_probe( worldinfo *w )
{
    uint8_t r0[BLOCKSX+1];
    uint8_t r1[BLOCKSX+1];
    uint8_t *row0, *row1;
```

Copy position and color to the *poscol* table:

```
    poscol pc[OBJNUM];

    for( int i = 0; i < OBJNUM; i++ ) {
        pc[i].obj_pos = w->spheres[i].obj_pos;
        pc[i].obj_col = w->spheres[i].obj_col;

    }
```

The world has a huge sphere in the center, and it's highly likely that it'll be hit. Always put it first in the sphere list. The others are sorted according to depth, the theory being that they are usually behind the big one if further away.

So, sort all spheres except the first:

```
    qsort( &pc[1], OBJNUM-1, sizeof(poscol), sortfunc );
```

Copy them back to the world:

```
    for( int i = 0; i < OBJNUM; i++ ) {
        w->gt.obj_pos[i] = pc[i].obj_pos;
        w->gt.obj_col[i] = pc[i].obj_col;
    }
```

Next up is the actual probing. Each corner of a 16x16 block is evaluated, and if all four hit the same type of target, it's either background only or sphere only. This means that the entire sphere evaluation can be eliminated on the GPU if the block only has background in it.

There's no need to go overboard with the optimizations here: The entire function runs in about 5 ms per frame on a single ARM core, which leaves a good margin to 16.67 ms for 60 fps. This version just checks two rows and swaps them for the next set.

```
#define BLOCK_ANY 0
#define BLOCK_BG  1
#define BLOCK_SPH 2

    // Check corners of 16x16 blocks
    row0 = r0;
    row1 = NULL;

    v3 ray_pos = v4_get3( w->gt.eye );

    int widthr  = (w->width +BLOCKW-1)&~(BLOCKW-1);
    int heightr = (w->height+BLOCKH-1)&~(BLOCKH-1);
    for( int y = 0; y <= heightr; y += BLOCKH ) {

        float fy = y/(float)w->height;

        uint8_t *dst = row0;

        for( int x = 0; x <= widthr; x += BLOCKW ) {

            float fx = x/(float)w->width;

            v3 startpos = v3_normalize( v3_set( w->gt.startpos.x + w->gt.ax * fx,
                                                w->gt.startpos.y + w->gt.ay * fy,
                                                w->gt.startpos.z ) );

            int i;
            for( i = 0; i < OBJNUM; i += 4 ) {
```

Ultra-simple intersection check. Skip the backside, since the viewer is never placed inside a sphere:

```
                v3 vv0 = v3_sub( ray_pos, v4_get3( w->gt.obj_pos[i+0] ) );
                v3 vv1 = v3_sub( ray_pos, v4_get3( w->gt.obj_pos[i+1] ) );
                v3 vv2 = v3_sub( ray_pos, v4_get3( w->gt.obj_pos[i+2] ) );
                v3 vv3 = v3_sub( ray_pos, v4_get3( w->gt.obj_pos[i+3] ) );

                float rv0 = -v3_dot( vv0, vv0 ) + w->gt.obj_pos[i+0].w;
                float rv1 = -v3_dot( vv1, vv1 ) + w->gt.obj_pos[i+1].w;
                float rv2 = -v3_dot( vv2, vv2 ) + w->gt.obj_pos[i+2].w;
                float rv3 = -v3_dot( vv3, vv3 ) + w->gt.obj_pos[i+3].w;

                if( v3_dotsq( vv0, startpos ) + rv0 > 0.0f ||
                    v3_dotsq( vv1, startpos ) + rv1 > 0.0f ||
                    v3_dotsq( vv2, startpos ) + rv2 > 0.0f ||
                    v3_dotsq( vv3, startpos ) + rv3 > 0.0f ) break;

            }
```

It's reasonably simple to separate sky/plane here too, but I couldn't find a straightforward way to exploit it back then, so it's either background or sphere:

```
            *dst++ = i == OBJNUM ? BLOCK_BG : BLOCK_SPH;

        }
```

If only one row done, calculate next one and restart:

```
        if( row1 == NULL ) {
            row0 = r1;
            row1 = r0;
            continue;
        }
```

Otherwise, swap them. Doesn't matter when it's done:

```
        uint8_t *tmp;
        tmp  = row0;
        row0 = row1;
        row1 = tmp;
```

Then find the correct place to store the probe values and iterate over the two row results:

```
        uint8_t *proberow = w->gt.probe + ((y-BLOCKH)/BLOCKH)*BLOCKSX;

        for( int x = 0; x < widthr/BLOCKW; x++ ) {
```

If all four is the same, it's either sphere or background only. Isn't this kind of code fun? Some people find it confusing. They're usually ~~bad~~ average programmers:

```
            *proberow++ = row0[x] & row0[x+1] & row1[x] & row1[x+1];
        }

    }

}
```

The shared buffer holding the data for the GPU is as follows with the probe block:

```
layout(binding = 0) uniform stuff
{
    vec4 obj_pos[OBJNUM];
    vec4 obj_col[OBJNUM];
    vec4 plane_norm;
    vec4 plane_col;
    vec4 eye;
    vec4 light_pos;
    vec4 startpos;
    float ax;
    float ay;
    float frame;
    float width;
    float height;
    uint8_t probe[BLOCKSX*BLOCKSY];
};
```

uint8_t is overkill but doesn't really matter. Plenty of space in the shared buffer.

Next up is dismantling the GPU code. The balancing act that needs to be done is finding out how much to unroll without risking running out of registers. Many variations were tested over a period of several years.

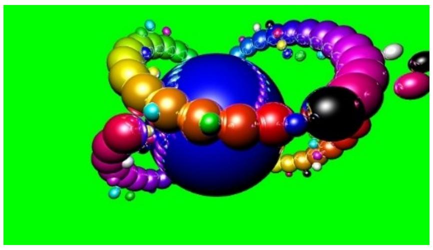
The internet is occasionally useful: After some searching, I found an article called *NV_GPU_program5*[29] that contained a lot of useful details. It lists the instructions, their register specification and such, but it said nothing about the instruction encoding. Damn. So close!

Never mind that, then. Let's use the texture coordinate to determine where the correct probe byte is and fetch it:

```
void main()
{
    (...)

    // Find probe result...
    int blx = int(floor(v_texCoord.x*width /float(BLOCKW)));
    int bly = int(floor(v_texCoord.y*height/float(BLOCKH)));
    int prval = int(probe[bly*BLOCKSX+blx]);
```

And use it to skip the entire sphere calculation if only the background needs to be rendered:

```
    // ...and only do sphere calculation if needed
    if( prval == BLOCK_SPH || prval == BLOCK_ANY ) {
```



It's probably a sphere, so next up is the reflection loop. Preloading data seemed to work well: There's so little code in each calculate and test block that the loads would actually stall. The horror! And there's no point in checking hits on the backside of the spheres, unless the viewer is placed inside one, and that never happened. But what if it did? Try it out! It'll look like the frontside, but in the back, so not that great. Pretty crap, actually. Tried making some stuff with it. Didn't work well.

```
        int refl;
        for( refl = 0; refl < REFLECTIONS; refl++ ) {

            float rt_dist = 0.0f;
            int rt_i = -1;
            vec3 rt_t0;
```

[29] http://developer.download.nvidia.com/opengl/specs/GL_NV_gpu_program5.txt

Start preloading data:

```
            vec4 v0, v1, v2, v3;
            v0 = vec4( ray_pos - obj_pos[0].xyz, obj_pos[0].w );
            v1 = vec4( ray_pos - obj_pos[1].xyz, obj_pos[1].w );
            v2 = vec4( ray_pos - obj_pos[2].xyz, obj_pos[2].w );
            v3 = vec4( ray_pos - obj_pos[3].xyz, obj_pos[3].w );
```

Iterate through the 128 spheres in units of 32. It's going to be a hard day's night. Of unrolling:

```
            for( int i = 0; i < OBJNUM; i += 32 ) {
                vec4 v4, v5, v6, v7;
                vec4 b0, d0;
```

Preload next set:

```
                v4 = vec4( ray_pos - obj_pos[i+4].xyz, obj_pos[i+4].w );
                v5 = vec4( ray_pos - obj_pos[i+5].xyz, obj_pos[i+5].w );
                v6 = vec4( ray_pos - obj_pos[i+6].xyz, obj_pos[i+6].w );
                v7 = vec4( ray_pos - obj_pos[i+7].xyz, obj_pos[i+7].w );
```

Do hit calculations for first set:

```
                b0 = vec4( dot( v0.xyz, ray_dir ),
                           dot( v1.xyz, ray_dir ),
                           dot( v2.xyz, ray_dir ),
                           dot( v3.xyz, ray_dir ) );

                d0 = vec4( b0.x * b0.x - dot( v0.xyz, v0.xyz ) + v0.w,
                           b0.y * b0.y - dot( v1.xyz, v1.xyz ) + v1.w,
                           b0.z * b0.z - dot( v2.xyz, v2.xyz ) + v2.w,
                           b0.w * b0.w - dot( v3.xyz, v3.xyz ) + v3.w );
```

Check for hit in first set:

```
                if( any( greaterThan( d0, vec4( 0.0f ) ) ) ) {
```

It's a hit. But it's not yet known which ones, only at least one in this set. Any negative values are going to make the calculations a lot harder (think quadruple *if*), so a quick way to eliminate them is needed. There's a trick for that:

```
                    vec4 t0  = -b0 - sqrt( d0 );
                         t0 += intBitsToFloat(floatBitsToInt( t0 )>>31);
                    float mv = min( min( t0.x, t0.y ), min( t0.z, t0.w ) );
```

Float adding the sign bit shifted into every slot will yield *NaN* if it's negative and 0 if it's positive, taking care of that problem. Finding the lowest positive value can then be done using *min*. But that would have been too easy. The compiler ended up generating redundant *MOV* instructions, unknown for which reason. Crap. As the android Walter in the movie Alien: Covenant said: "When one note is off, it eventually destroys the whole symphony."

Another way to get rid of them is by using another trick: Inverting the values and the tests. There will be more code generated, but fewer effective operations. Trust me on that one. I tried out many variations. Try it yourself! There's an astounding amount of such tricks that can be done when working with floats. So, do this instead:

```
                    vec4  t0 = 1.0f / (-b0 - sqrt( d0 ));
                    float mv = max( max( t0.x, t0.y ), max( t0.z, t0.w ) );
```

One of them is closer, so save it. But it has to be possible to find out which one later. Just saving all the values won't work, since there's no registers left, and it'll end up with single 32-bit register recycling and *swizzles* getting ruined. But just saving 3 of them and the base position keeps the register bank happy for the time being, making it slightly harder to figure out which one outside the loop:

```
                    if( mv < rt_dist ) {
                        rt_dist = mv;
                        rt_t0   = t0.yzw;
                        rt_i    = i + 0;
                    }

                }
```

The compiler will output a "might be uninitialized" warning for *rt_t0*. If it's initialized, that's 2 cycles down the hatch each iteration per core. Again. It's sums up to a lot. A big lot. Can somebody move that test in the compiler, please? Every compiler did it wrong back then. Thanks.

0(76) : warning C7050: "rt_t0" might be used before being initialized

The next step was manually unrolling all this. So, repeat the stuff above 7 times more here. Adjust the offsets, of course. The code was horrendously long, but that doesn't matter much. All the cores are doing the same anyway, so nobody's messing up anything:

```
                // ...
                // ditto for preload 8-11, calc 4-7 etc.
                // ...
```

Less than 3.0 in the inverted dist was usually close enough. Saved time. Didn't fail that often:

```
                if( rt_dist < 3.0f ) break;
            }
```

If nothing was hit, just jump out:

```
            if( rt_i == -1 ) break;
```

Something was hit, so find the id. It's semi-simple going from the split registers to the actual id. Just see if anything in *rt_t0.xyz* (which is *yzw*, obviously) matches precisely and add up the values in the test vector. Nifty! The things I do to save registers. I'm sure there are other, more important things that need saving, like whales, or maybe some kind of bird:

```
            // Find id of sphere hit
            ivec3 iv = ivec3( equal( rt_t0, vec3( rt_dist ) ) );
            rt_i += iv.x + iv.y + iv.y + iv.z + iv.z + iv.z;
```

Move the position and update it, with some diffusion and specularity so it looks cooler:

```
            ray_pos += ray_dir * (1.0f / rt_dist);
            vec3 n   = normalize( ray_pos - obj_pos[rt_i].xyz );
            vec3 l   = normalize( light_pos.xyz - ray_pos );

            float diffuse  = max( dot( n, l ), 0.0f );
            float specular = pow( max( dot( ray_dir, l - n * 2.0f * diffuse ), 0.0f ), 8.0f );

            col += vec3( specular ) + obj_col[rt_i].xyz * diffuse;
```

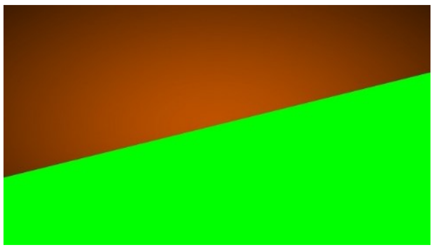
Use GLSL *reflect* to reflect the ray. Probably shouldn't have, since, as discussed in the other GPU chapter, the compiler wouldn't reorder it properly. It's outside the main loop, so not particularly important:

```
            ray_dir = reflect( ray_dir, n );
        }
```

Now it'll either keep reflecting until max reflections is reached or it's a miss. If anything was hit/reflected, the job's done:

```
        if( refl != 0 ) {
            outColor = vec4( col, 1.0f );
            return;
        }

    }
```

With all the sphere calculations out of the way, the rest of the code dealt with the shadow plane and sky.



First, check if the plane is hit:

```
    float rt_dist = -(dot( ray_pos, plane_norm.xyz ) + plane_norm.w) / dot( ray_dir, plane_norm.xyz );

    if( rt_dist <= 0.0f ) {
```

It wasn't, so draw a pulsating sky and exit:

```
        vec2 dxy = v_texCoord - 0.5f;
        float dv = 1.0f - sqrt( dot( dxy, dxy ) + (sin(fract(frame/120.0f)*2.0f*3.14f)+1.0f)*0.5f );
        outColor = vec4( plane_col.yzx * dv, 1.0f );
        return;
    }
```

It's certain that the plane is hit. Move *ray_pos* to the plane and make *ray_dir* point at the light source:

```
    // Plane hit
    ray_pos += ray_dir * rt_dist;
    ray_dir  = normalize( light_pos.xyz - ray_pos );
```

Checking for shadow was a bit simpler since it's only interesting if anything at all is hit, not what it is or how far away. It follows the same structure as the sphere calculation, except it's unrolled slightly differently:

```
    // Preload 0-3
    vec4 v0 = vec4( ray_pos - obj_pos[0].xyz, obj_pos[0].w );
    vec4 v1 = vec4( ray_pos - obj_pos[1].xyz, obj_pos[1].w );
    vec4 v2 = vec4( ray_pos - obj_pos[2].xyz, obj_pos[2].w );
    vec4 v3 = vec4( ray_pos - obj_pos[3].xyz, obj_pos[3].w );

    int i;
    for( i = 0; i < OBJNUM; i += 16 ) {
        vec4 v4, v5, v6, v7;
        vec4 d0, d1, d2, d3;
        vec4 b0;

        // Preload 4-7
        v4 = vec4( ray_pos - obj_pos[i+4].xyz, obj_pos[i+4].w );
        v5 = vec4( ray_pos - obj_pos[i+5].xyz, obj_pos[i+5].w );
        v6 = vec4( ray_pos - obj_pos[i+6].xyz, obj_pos[i+6].w );
        v7 = vec4( ray_pos - obj_pos[i+7].xyz, obj_pos[i+7].w );

        // Calc 0-3
        b0 = vec4( dot( v0.xyz, ray_dir ),
                   dot( v1.xyz, ray_dir ),
                   dot( v2.xyz, ray_dir ),
                   dot( v3.xyz, ray_dir ) );

        d0 = vec4( b0.x * b0.x - dot( v0.xyz, v0.xyz ) + v0.w,
                   b0.y * b0.y - dot( v1.xyz, v1.xyz ) + v1.w,
                   b0.z * b0.z - dot( v2.xyz, v2.xyz ) + v2.w,
                   b0.w * b0.w - dot( v3.xyz, v3.xyz ) + v3.w );
```

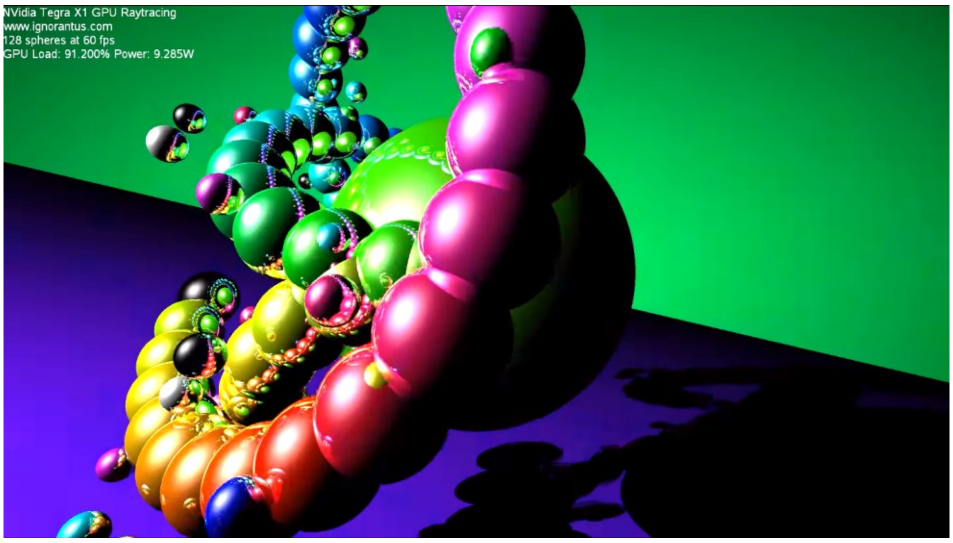
Insert the same 3 more times here, with updated refs. Then check if anything was hit. I like this programming style: Do calculations first, check results later. Simple and to the point:

```
        if( any( greaterThan( d0, vec4( 0.0f ) ) ) ||
            any( greaterThan( d1, vec4( 0.0f ) ) ) ||
            any( greaterThan( d2, vec4( 0.0f ) ) ) ||
            any( greaterThan( d3, vec4( 0.0f ) ) ) ) break;

    }
```

Then determine if it's shadow or plane and calculate the final output color. That's simple: If max iterations are reached, nothing was hit so it's the plane and vice versa. The plane had a semi-cool shady effect if it was a total miss. Well, it wasn't that great, but it was cheap. Just some random crap from the recycle bin:

```
    // Black shadow if anything was hit
    float diffuse = i == OBJNUM ? dot( plane_norm.xyz, ray_dir ) * 1.2f : 0.0f;
    outColor = vec4( plane_col.xyz * diffuse, 1.0f );

}
```

That'll raytrace 128 spheres on the NVidia Tegra X1. Easy peasy.


NVidia Tegra X1 GPU Raytracing
www.ignorantus.com
128 spheres at 60 fps
GPU Load: 91.200% Power: 9.285W

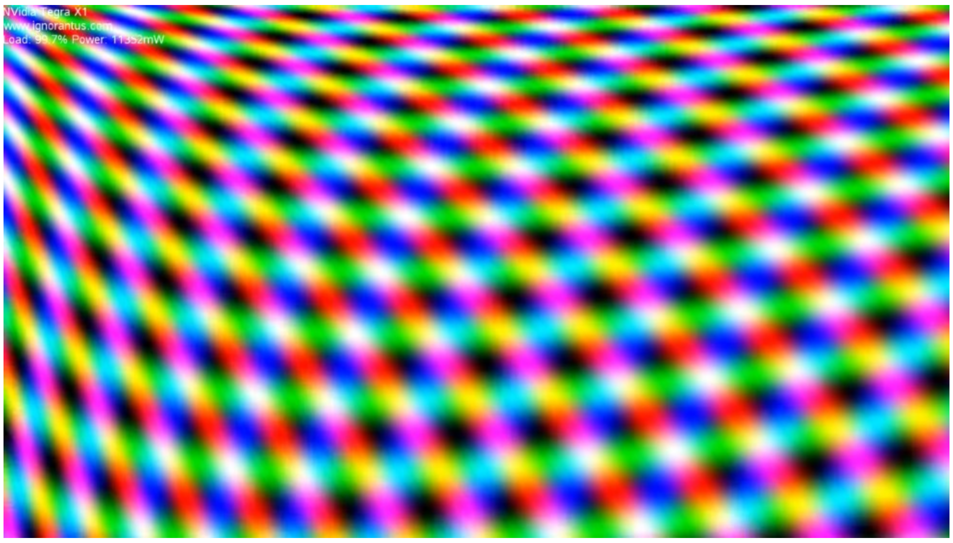
The video called *GPU Raytracing on NVidia Tegra X1 v2.0* shows it off. Again, I forgot to specify who made the music. But it's a pretty catchy tune that I used in several test videos. Sorry, musicians! All newer videos I made have this issue sorted.

What's missing here, obviously, is the CPU control code. That will be covered in the next chapter.

A fun test was trying to force the X1 into thermal shutdown by maximizing the computational load. The raytracer provides a good supply of semi-random data, so I just tried some random stuff until the load was as close to 100% as possible, and the power used as high as possible. Both the X1 and X2 have simple methods to read those values.

```
void main()
{
    float x01 = gl_FragCoord.x/1920.0f;
    float y01 = gl_FragCoord.y/1080.0f;
    vec3 col = vec3( 1.0f );
    vec3 pv = vec3( x01, y01, x01*y01 );
    for( int j = 0; j < 2; j++ ) {
        col = normalize( col );
        for( int i = 0; i < OBJNUM; i++ ) {
            col += obj_col[i].xyz * y01;
            col += obj_pos[i].xyz * x01;
            vec3 pobj = vec3( max( obj_pos[i].x, 0.1f ),
                              max( obj_pos[i].y, 0.1f ),
                              max( obj_pos[i].z, 0.1f ) );
            vec3 pn = normalize( pobj );
            col += 1.0f / sqrt( (pv * dot( pv, pn )) );
            col += reflect( col, pn );
        }
    }
    outColor = vec4( sin01( col ), 1.0f );
}
```

It should be noted that my notes about this were... thin. The function *sin01* probably does what it sounds like. To make the code work in the 2017 version of the raytracer, the sorting of the spheres should be disabled. And it's considered a good idea to reduce OBJNUM a bit and move the normalization further down.



It actually looked pretty nice, with bars bouncing around. Check out the video called *NVidia Tegra X1 Load and Power Test*. The music is called *Enemy Ships* and was made by Jason Shaw, owner of the audionautix.com website. The tune is heavy, but snappy since the video is only 30 seconds long.

GPU power peaked out at around 11.5 W. I guess the X1 would have caught fire with some memory and CPU load. NVidia claimed 15 W at full load for the Jetson TX1 module. That's conservative at best.