Utdrag fra boken *Ekte Programmering* (2021) av Corneliusen & Julin.
Mer informasjon: www.ignorantus.com/ekte/

En kopi og litt trigonometri

På slutten av 2017 hadde jeg gått tom for materiale, men ønsket å lage en demo for å kjøre på en kul SoC: NVidia Tegra X2. Den hadde fire nyttige ARM-kjerner, to delvis nyttige og en 256-kjerners GPU.

shadertoy.com, et helt fantastisk nettsted, har noen av de beste GLSL-shaderprogrammererne på planeten, men det så ut til at mål-GPU-en de benyttet var betydelig kraftigere enn den i X2. Nåvel. Problemer er til for å løses. Jeg fant noen virkelig gode shadere der, og satt som mål å skvise dem inn i de 256 kjernene på den stakkars X2-en. Det var vanskeligere enn man skulle tro. Og målet var å få denne demoen på internett før jul 2017, som ga rundt tre måneder å gjøre det på.

Jeg gjorde mitt beste for å overholde lisensvilkårene den gang. Alle effektene, bortsett fra *Julia Quaternions* og raytraceren min, var basert på kode fra shadertoy.com. Nettstedet bruker en *Attribution-NonCommercial-ShareAlike 3.0 Unported license*[35]. Opprinnelig utgiver og lenke til ShaderToy-siden med original kode er oppført nedenfor. Alle de modifiserte shaderne og kontrollkoden fra den gang er fremdeles tilgjengelige på nettstedet mitt.

Effect Name	**Link**	**Author**
Galactic Dance	https://www.shadertoy.com/view/XtlGWs	Sinuousity
Glow City	https://www.shadertoy.com/view/XlsyWB	mhnewman
Raytracer	https:///www.ignorantus.com/ekte	Corneliusen (CC0 1.0-lisens)
Noise 3D	https://www.shadertoy.com/view/4ddXW4	revers
Julia Quaternions	https://www.cs.cmu.edu/~kmcrane/	Keenan Crane (ingen lisens)
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

Først kan det være greit å finne ut hvor lite regnekraft GPU-en i X2 egentlig har. GFLOPS er en syntetisk måling av en situasjon som aldri oppstår: Den forutsetter at hver eneste instruksjon som utføres er FMA (fused multiply add). Gitt at operasjonsenhetene har en pipeline, starter de en FMA per klokke og fullfører en FMA per klokke. FMA er to basisoperasjoner, multiplikasjon og addisjon, så noen hadde den lyse ideen å kalle det to operasjoner per klokke. Enkelt å beregne, men ikke veldig nyttig.

Det er antageligvis mer interessant å beregne hvor mange, eller få, sykluser som kan gis til hver piksel per bilde. GPU-en kjørte på 1.3 GHz. Antall tilgjengelige sykluser per sekund vil være 1 300 000 000 x 256 = 332 800 000 000. Jeg hadde en gang en sjef som trodde det var et stort tall og banebrytende. Han hadde ikke fulgt utviklingen på PC-markedet de ti siste årene. På den tiden hadde jeg et grafikkort som kunne utføre omtrent 5 TFLOPS etter definisjonen over. Og det hadde fem ganger RAM-båndbredden til X2. Eller var det ti? Uviktig. Forskjellen var gigantisk.

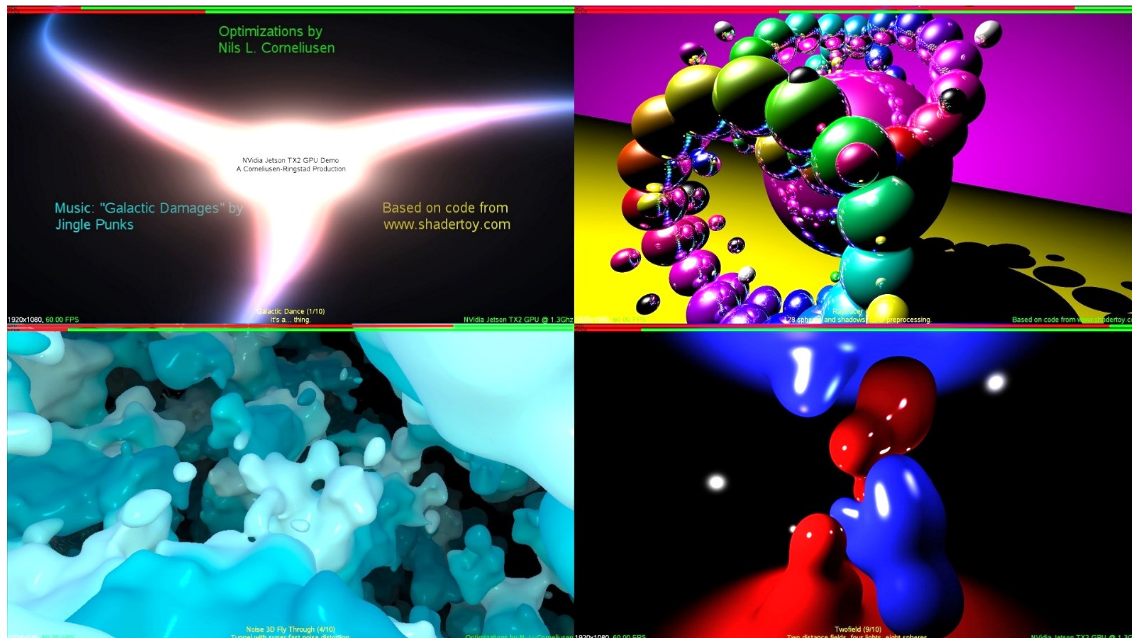
Anta nå at det skal tegnes et 1920x1080-bilde 60 ganger i sekundet. Det er totalt 1920 x 1080 x 60 = 124 416 000 piksler. Del antall sykluser på det, og resultatet blir 2674 sykluser per piksel per bilde. Det er ikke mye, spesielt hvis veldig tidkrevende ting skal gjøres, som å simulere et hav. Det må settes i perspektiv: En flyttallsoperasjon som FMA tar omtrent åtte sykluser, men er helt pipelinet. Forutsatt at instruksjonsstans kan unngås ved ikke å bruke mattebiblioteksfunksjoner, samt å rulle ut nok, er neste hindring *if()-else* konstruksjoner - der alle tar kostnadene for begge grenene, som forklart tidligere. Så er det faren for å gå tom for registre eller at kompilatoren kløner det til. Enhver miss vil få tallet 2674 til å falle raskt. Veldig raskt.

Heldigvis kunne minneoperasjoner i shaderne stort sett elimineres, siden det ikke var mye, utenom skriving til skjerm: Output, noen overlays og en og annen overgangseffekt. Ikke så mye til sammen. Likevel, prosjektet var dømt til å feile fra start. Eller var det? På tide å finne det ut!

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

Kontrollkoden endte opp på rundt 2000 linjer. De 11 GPU-shaderne var totalt 2500 linjer. Ingen dyr ble skadet, men enda viktigere: Ingen teksturer ble brukt som del av effektene. De kommer garantert ut i full oppløsning og i 60 bilder i sekundet med synkronisert musikk. Jeg måtte dessverre fjerne musikkavspillingen pga. problemer med opphavsrett, men stol på meg. Total kjøretid etter oppstart er nøyaktig fem minutter, så det er 10 effekter som kjører i 30 sekunder hver. (Den 11. shaderen er en skalerer. En veldig lavbudsjett bilineær skalerer, modifisert litt så overgangene ser bra ut. Det var den vanlige greia: Standardskalereren til NVidia er kul, så lenge man ikke klipper fra midten av et bilde hvor kildekoordinatene hele tiden endrer seg. Kommer tilbake til det.) Den vil kjøre og se riktig ut i 1440p eller 2160p, men rammeraten vil falle under 60 nå og da. Oftere nå enn da på X2.

En hasteversjon ble gitt ut i tide til jul 2017. Den hadde ikke full 60 fps i alle effektene. En fikset versjon ble gitt ut litt senere, *Nvidia Jetson TX2 Xmas Demo 2017 Remastered*, i april 2018. Ok, så tok det litt tid å fikse alle problemene med 60 fps.

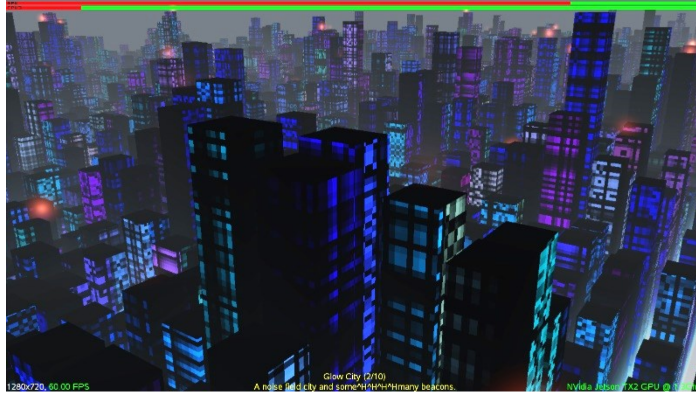


Den første musikken brukt i videoen heter *Galactic Damages* og er laget av Jingle Punks. Den ble funnet i *YouTube Audio Library*. Jeg trodde sånne nettsteder var sære på lisenser, men det står bare: "You're free to use this song in any of your videos." Javel, sikkert. Gjør ikke den feilen igjen. På tross av lisensen var musikken kul. Den andre musikken heter *Cephalopod* og er laget av Kevin MacLeod. En ganske hard 90-talls techno-låt som får blodet til å koke. Jeg elsker den! Det forekommer meg at ikke alle vil være enige i det.

Den er merket som en Corneliusen-Ringstad produksjon. Tor Ringstad ga uvurderlig teknisk informasjon om X2 og ordnet opp i diverse Android- og Linux-teknikaliteter angående utgangsdisplayet. Noen som er interessert i 59.94 bilder i sekundet? Bare si nei! Det er 60 bilder i sekundet eller ingenting!

Som nevnt, den originale koden for de fleste av disse shaderne forutsatte mye kraftigere GPU-er enn X2. Jeg vil si seks ganger antall shaderkjerner og ti ganger minnebåndbredde, og det er et konservativt anslag. Jeg antok derfor, ganske riktig, at de som hadde skrevet dem ikke hadde lagt sjela i få dem til å gå superfort. Og det var nødvendig på X2.

La oss se på noen av de mer interessante endringene jeg gjorde for å få til dette. Metodene skal fremdeles kunne brukes på lignende problemer i dag. Det anbefales sterkt å referere til den originale koden på ShaderToy mens du leser dette kapitlet. Nettstedet er helt fantastisk: Bare legg til noen endringer og kjør shaderen i nettleseren. Ganske gøy!



Glow City - basert på kode av mhnewman
Det er et bylandskap med blinkende lys på toppen. Bakken er dekket av tåke. Veldig uhyggelig.

Den uendrede Glow City leverte omtrent 16 fps på X2, hovedsakelig fordi det er for mange bygninger og de er for lave. Reduser antall bygninger til 100 og gjør én iterasjon av *j*-loopen i *main()*. Øk minstehøyden på bygningene og reduser tilfeldigheten litt for å dekke over bakgrunnen. Og fjern tåken. Uhyggeligmodus redusert. Fillern.

noise() og *hash()* funksjonene var ganske avanserte. Originalkoden var slik:

```
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);
}
```

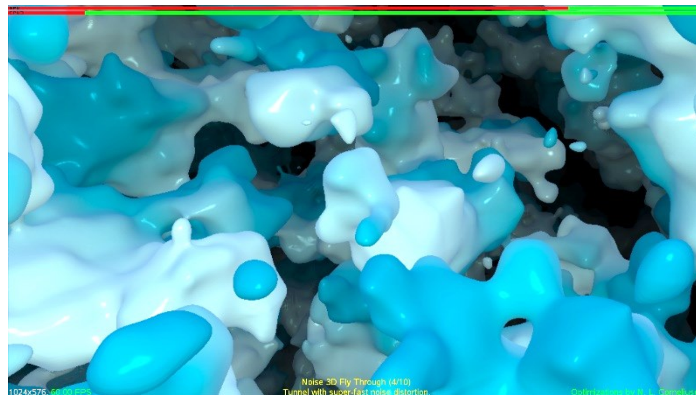
Dette er en støyfunksjon der nok tilfeldighet er nødvendig, men ikke total, og ikke bare et simpelt sinusmønster. Det kan reduseres ved å bruke noen få sinuser med faktorer og bare justere det til det ser ok ut:

```
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;
}
```

Øk sannsynligheten for blinkende lys og endre vekting for vinduer. En bivirkning er at nå blir det fargede vinduer uten ekstra arbeid. Alltid fint når kulere resultater er gratis!

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

Og så kjørte den på komfortable 73 fps. Den hadde ikke den dystre følelsen av originalen, men den er mer prangende. Jeg vokste opp på 80-tallet og liker prangende ting.



Noise 3D - basert på kode av revers

Elsker ikke alle flytende blobber? Mange flytende blobber! Den uendrede versjonen kjørte på 13 fps, igjen først og fremst på grunn av tunge støyberegninger. Litt for tidlig å kalle det en trend. Originalkoden så slik ut:

```
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 );
}
```

Det eneste interessante er at forholdet mellom *hash()*-verdiene forblir det samme: Hash-verdien *n+157* må komme igjen når *n+0* kommer dit, ellers er det bare en veldig dyr random-funksjon. Det skal være en tunnel med flytende blobber, ikke tilfeldig... vel, søppel.

Så: På CPU-en, forhåndsberegn en tabell med vec4-verdier som er en potens av to i størrelse, så den wrapper pent. Formelen som brukes er den samme som i originalen, men i betydelig mindre skala. Den har bare 256 elementer, som var nok til å generere et interessant resultat. 128 så fælt ut.

```
#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;
    }
(...)
```

Lag så en ny støyfunksjon som bare leser fra tabellen. Som nevnt tidligere, så er det billig: To sykluser pr. 128-bit lesing. Og pass på å håndtere negative tall riktig:

```
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;
}
```

Dessverre nektet kompilatoren å generere *LRP*-instruksjoner (lineær interpolering), ukjent av hvilken grunn. Å slå sammen flere *noise()*-kall for å unngå instrukssjonsstans virket heller ikke godt, siden kompilatoren ville fjerne det igjen for å spare registre. Vel, dette har jeg vært ute for før. Fant ingen løsning denne gangen heller.

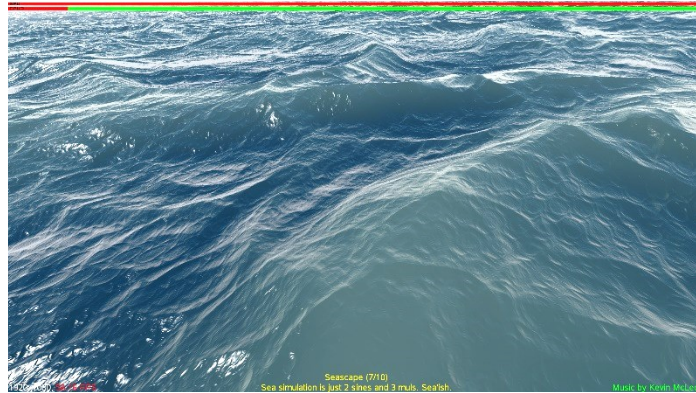
Så det må spares andre steder: Den tabellbaserte *noise()*-funksjonen, fjerning av myke skygger og okklusjon og reduksjon av dybden til 96 ga 34 bilder per sekund. En god start!

Uheldigvis ble resultatet av alle disse endringene alvorlige visuelle artifakter. Men kunne ikke *md* bare justeres fortløpende, så det er flere detaljer nærme i bildet og mindre lenger unna? Start *md* på 0.9 og multipliser med 1.025 for hvert trinn etter at de første 24 trinnene er gjort. Dette var tilfelle i flere av shaderne. Denne idéen måtte testes ut. Det er veldig enkelt. Eller ikke, fordi rekkefølgen av *if()*-uttrykkene er viktig. Det var kritisk at *md*-multiplikasjonen skjedde før *Far*-testen, av en eller annen uklar grunn. Orket ikke sette meg ned og finne ut hvorfor på den tiden, dessverre.

```
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;
}
```

Dette ga de nødvendige 60 bildene i sekundet og ser nesten ut som originalen. Jo, det er fortsatt noen artifakter, men den kjører på en liten SoC, ikke en diskré GPU.



Seascape - basert på kode av TDM

Ah, gleden av å simulere et hav når man ikke har ubegrenset med regnekraft. Sjøen er et plan som er forvrengt av et par funksjoner (jeg forenkler veldig her):

```
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);
}
```

Hva ligner dette på? Det er enda en kompleks støyfunksjon! La oss kalle det en trend nå. Denne gangen er det en samling sinuser, så jeg prøvde ut diverse tilfeldige ting. Det så mest ut som et plan med veldig tydelige sinusmønstre. Fillern. Men jeg snublet over denne kombinasjonen som faktisk fungerte ganske bra:

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

Dette er superbillig og gir den et sjøaktig utseende

Neste er *map()*- og *map_detailed()*-funksjonene. Det er nok å fjerne ett *sea_octave()*-kall for å forbli marginalt over 60 fps med litt horisont, så fjern det siste kallet i *map()* og gang det første resultatet med 1.4.

Jeg liker en rett horisont, så fjern *"dir.z + = length(uv) * 0,15;"* fra *main()*.

For å få konstant 60 fps, så må noen flere ting reduseres. Flytt *ang*-, *orig*- og *m*-beregningene til CPU per bilde:

```
        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 );
    }
```

Vikl ut *heightMapTracing()*. Den uendrede versjonen var litt for komplisert:

```
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;
}
```

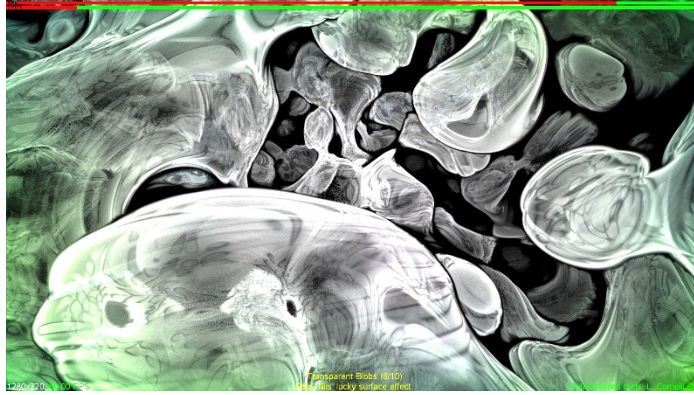
Det første *map()*-kallet er alltid likt, så det blir mye enklere:

```
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() kan dyttes inn i *main()*, *for()*-løkkene strippes ut, *hx* flyttes. De viktige gjenværende greiene er da:

```
(...)
    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 );
(...)
```

Det ender opp med noe som ligner originalen. Sjøen er mer syntetisk, men mønstrene ser ganske pene ut. Og den kjører på 60 fps. En bieffekt av å kødde med sjøsimuleringer er det for alltid vil ødelegge filmer basert på havet. Er havet ekte, eller er det simulert? Kjipt å gå helt frem til TV-en og stirre etter mønstre hele tiden.



Transparent ~~Lattices~~ Blobs - basert på kode av Shane

Denne ble opprinnelig kalt *Transparent Lattices*. Dessverre var de gjennomsiktige overflatene litt kjedelige.

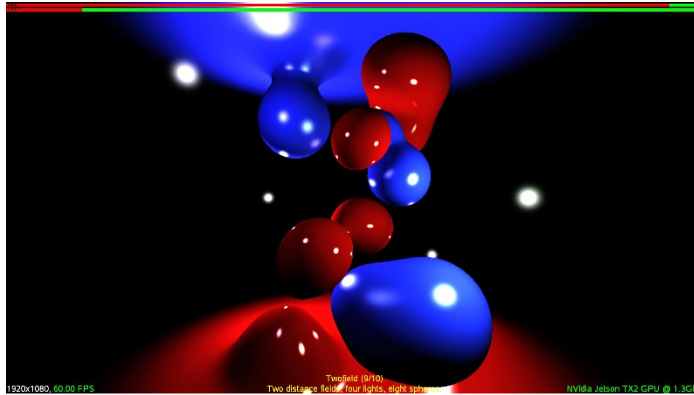
Heldigvis var blobb-koden fortsatt i shaderen, bare kommentert ut. Dessverre var blobbene også ganske kjedelige. De trengte noe på overflaten. Den uendrede *calcNormal()* funksjonen så slik ut, bortsett fra kommentaren som er ny:

```
// 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)));
}
```

map()-kallene var uhåndterbare, så jeg prøvde å flate ut hele greia og finne ut hva som skjer. Dessverre er jeg veldig dårlig til sånt, så jeg kortsluttet et par av dem ved et uhell, kjørte det... og si velkommen til de nye og prangende blobbene:

```
// 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 ) );
}
```

Undervurder aldri kraften av flaks i programmering!



Twofield - basert på kode av w23

I utgangspunktet er det to sett med kuler som følger et statisk mønster, som påvirkes av to felt som forårsaker visuelt behagelige forvrengninger.

Den originale koden leverte 26 fps på NVidia Jetson TX2. Kompilatoren hadde slet med å følge strukturen i *world()/w1()/w2()* osv. Her er originalkoden:

```
// 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)));
}
```

Mulig jeg er treig, men hva er det som skjer i den koden? Med litt omstokking og sammenslåing blir det mye mere oversiktlig. Da klarer også kompilatoren å optimalisere det bedre. Igjen.

```
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 ) ) );
}
```

Fortsett med å flytte bevegelse og O-beregning til CPU-siden for hvert bilde. Flytt også farge og lysposisjontabellene for senere bruk. CPU-en kan justere dem gratis under rendering og det koster omtrent 0.5 bilder per sekund:

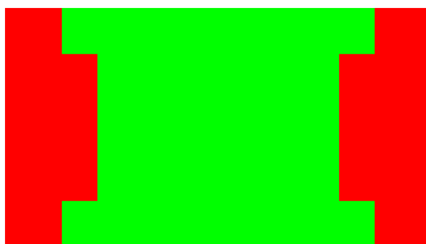
```
        if( d->type == DEMO_TWOFIELD ) {

            // 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 );

    }
```

Jeg likte aldri skyggene, å fjerne dem øker til 43 fps. Juster den tåkelignende attenueringen fra 10-20 til 12-18 og fjern 256 kolonner fra høyre og venstre side i bildet for 58 fps. I *trace()*, start *L* på 12.5 i stedet for 0 for 78 bilder per sekund, så målet er nådd. Dermed kunne antall kuler *N* økes til 8 og det blir 64 bilder per sekund. Enda flere kuler ville bli for mye uten å endre banene.

Jeg gjorde noen andre mindre endringer siden jeg liker skarpe farger og litt action: Roter lyskildene rundt kulene, gjør lysene større/mindre avhengig av Z, fjern gammakorreksjon, legg til fade inn/ut, juster spekulærverdiene. Gjennomsnittlig fps endte opp på rundt 62 fps og belastningen på 93%.

Da kjører den i 60 fps unntatt under overgangene. Det kan spares enda mer ved å kappe av enda flere kolonner i seksjonene lengst til høyre og venstre. Det skjer ikke noe der uansett. Resultatet er at 37% av pikslene kan elimineres og masse tid spart. Se bildet. Sikkert enklere å dekke området med høynivå OpenGL-triangler, men jeg fjernet dem bare i shaderen:



```
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;
    }
    (...)
```

Det er en triviell reduksjon i *lightball()* som kompilatoren ikke ville gjøre. Den var litt rar: *length()* er ganske tung. Å redusere det til *dot > L*L* burde være trivielt for kompilatoren, men den ville ikke samarbeide. Endre dette:

```
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;
}
```

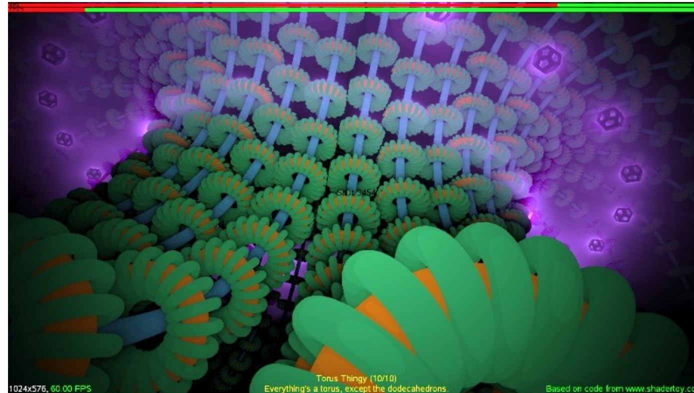
Til dette:

```
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 );
```

Og reguler størrelsen med denne ekstra *lv*-beregningen:

```
    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;
}
```

Og vi er på 60 fps på X2 også i overgangene. Kult.



Torus Thingy - basert på kode av bal-khan
Det er en torus som roterer. Torusen er bygd opp av tori som er bygd opp av tori... poenget bør være klart som kildevann. Mange tori. Og noen dodekaeder som flyter rundt. Selvsagt, det ville ikke være det samme uten.

Begynn med å fjerne ubrukt kode. Erstatt *mylength()* med standard *length()*. Bytt ut *march()* med den som ble kokt sammen i noise3d og rull ut igjen:

```
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 );
}
```

Ja, den blir støyete på avstand, men den beveger seg raskt, så det er ikke veldig viktig. Dodekaedrene blir også litt fetere.

Erstatt *mod()*-kalkulasjoner i *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.) );
(...)
```

Som tidligere nevnt, bruk av mattebiblioteksfunksjoner i GLSL var forferdelig dyrt den gang. En av de dyreste, *atan2(y, x)*, kalt *atan(y, x)* i GLSL, var rundt 28 instruksjoner som ikke ville bli omordnet, noe som førte til en uendelig serie med instruksjonsstans. Shaderen trengte mange av dem, så det var viktig at de ble utført raskt.

Nvidias biblioteksfunksjoner var ganske ok, og en mulig løsning var å konvertere assembler-versjonen tilbake til GLSL bare for å få den omordnet. Men det ville ikke vært noe morsomt. Det er mulig å gjøre dette bedre. Gutta hos IEEE gjorde det riktig i artikkelen som heter *Full Quadrant Approximations for the Arctangent Function*[36]. Jeg konverterte eksemplet merket "code 2" til GLSL:

```
// IEEE Signal Processing Magazine ( Volume: 30, Issue: 1, Jan. 2013 )
// Full Quadrant Approximations for the Arctangent Function
// https://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;
}
```

Som det meste i livet kan *atan2()* gjøres enda raskere, avhengig av hvor mye presisjon som trengs. Knut Inge Hvidsten har publisert noe banebrytende arbeid om det i artikkelen *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?).*

Den løsningen var dessverre litt for upresis for behovet her, men det er interessant lesning og anbefales.

IEEE-gutta hevdet at de laget en 4-veis SSE2-versjon av den, men presenterte ingen kode i artikkelen. Igjen føles det som en trend. Merkelig hvor ofte det skjer. Trender, altså. De hevdet:

> *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.*

Det der reiser bare flere spørsmål enn det besvarer. Kode 2 er fire-kvadrant versjonen, så den er utgangspunktet. Når jeg plukker ut målingene deres fra den litt forvirrende tabellen gir det følgende resultater, som er *atan2()* per mikrosekund, av en eller annen obskur grunn:

Code 2	140.588
Intel IPP 7.0	218.195
SSE2	741.25

Det er ganske enkelt å komme med slike uttalelser når ingen kode vises. For å sitere en tidligere Transmeta-ansatt:

[36] https://ieeexplore.ieee.org/document/6375931
[37] https://www.linkedin.com/pulse/angles-atan2-taxicabs-knut-inge-hvidsten/

"Talk is cheap. Show me the code." Hvis koden bare var en rå ekspansjon av "Code 2" til SSE2 vil den antakeligvis stanse ganske mye. Jeg har ingen anelse om hva Intel IPP-biblioteket gjorde. Men ok, det er en faktor 5.2 forbedring, som er ganske pent.

Kan det gjøres bedre? Sikkert, ved bare å gjøre enda flere av dem. Tung SIMD-kode på arkitekturer med få registre får ofte omorganiseringsproblemer. Hvis den neste SIMD-instruksjonen avhenger av den forrige, så vil det bli tregt. For å få litt gjennomstrømming, gjør 2x4 for 8 *atan2()* hvert kall. Men hva er gøy med å bare gjøre omtrent det samme som beskrevet i artikkelen? Svært lite. Jeg prøvde derfor isteden å gjøre det på klassisk Neon. En rett frem 2x4-versjon av koden ovenfor vil se sånn ut:

```
#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 );
```

En bonus: Konvertering av NaN til 0.0. Kan være nyttig, men stort sett ikke. Ta det med hvis det trengs:

```
    // 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;
}
```

Hvor ofte får man sjansen til å kalle en funksjon satan? Det var hysterisk morsomt. Helt på høyde med å kalle en video demuxer dux[38], eller en buffer som inneholder NAL-enheter nalle[39], eller en muxer Mux-O-Matic[40]. Har drevet for mye med H.264 videobehandling i det siste. På tide å slå på luftfukteren, vitsene ble litt tørre.

Uansett gjenstår spørsmålet om hvorfor Intels IPP-implementering sugde, men det skipet gled over kanten av kartet. Gadd ikke kaste bort dager på å finne ut av det. Og det kan være et spørsmål om nøyaktighet. Jeg bryr meg virkelig ikke om det heller, men kanskje andre gjør det. Hvis det trengs noen millioner *atan2()* per sekund, spiller det trolig ingen rolle om svarene er litt, om ikke for mye, unna et presist svar. Det vil være nødvendig ganske snart.

Tilbake til den aktuelle shaderen. Den trenger mange *atan2()*-kall, så det klokeste ville være å fjerne så mange som mulig. Noen steder opptrer de i par der input til den andre avhenger av resultatet av den første. Denne optimaliseringen er litt vanskelig å se og tilnærmet korrekt. Og brutalt rask. Uansett, det går til helvete visuelt når det ikke er gitt hvilken side av torusen som vises: Se nøye etter, de bakre toriene deles på midten noen ganger. Oops. Det er nært nok mesteparten av tiden.

Den originale koden:

```
#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 nesten alle, dvs. mange, kjente verdier av *var*, reduserer dette til PI/4, så lag en enklere *modA()* for det andre kallet og bruk det første resultatet som 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 );
```

Og ditto for det oransje fyllstoffet. Dette:

```
    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] Fra filmen Bloodsport (1988), basert på livet til Frank Dux. Og Jean-Claude Van Damme var fremdeles kul på den tiden.
[39] Svensk for bamse, leketøyet altså.
[40] Ice-O-Matic er en ismaskin, Cube-O-Matic en legendarisk Amiga demo.

Blir det mye enklere:

```
    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 );
```

Det følges opp med noen flere reduksjoner. Ukens "jeg kan ikke tro at jeg ikke så dette tidligere": *I_MAX* er 100. Høres ganske vilkårlig ut, den kom aldri til 100 iterasjoner uansett. Men det vil påvirke utrullingen seriøst. Kompilatoren klarer det ikke selv pga. *if()-break*, så et presist anslag er viktig. Ellers vil den genererte koden bli altfor tung og synke fortere enn en elektrisk sparkesykkel i Akerselva.

Uansett, *march()*-funksjonen må gjøres om litt, så den blir raskere, mot litt ekstra støy. *log()*-kallet er egentlig ikke nødvendig. Originalkoden er:

```
#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 );
}
```

Som blir dette, som er nesten det samme. Veldig nær:

```
#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 );
}
```

Som nevnt vil den feile innimellom, og den trenger god, men ikke super *atan2()*-presisjon. Prøv selv å sette inn Hvidstens super-ultra-raske versjon for en unik opplevelse!

Kontrollkoden og en tur til Denver

Kontrollkoden vokste til en uhåndterlig klump på rundt 2000 linjer. Raytracer-kontrollkoden ble stappet inn der sammen med, vel, alt annet. Min kunnskap om høynivå OpenGL var ikke fantastisk da, og heller ikke nå, så det ble mye prøving og feiling. Og så finner man til slutt ut at alle feilene kom av manglende *glBindTexture()*-kall. Jeg aner fortsatt ikke hva det kallet gjør, men det så ut til å være viktig.

Demoen ble kjørt på Android, som oppførte seg som en trebent hund kalt Lucky. Alle nyttige ting fra Linux hadde blitt fjernet, så støvet måtte blåses av det gamle *syscall*-grensesnittet. Godt gjort, Google! Jeg fikk den kule 90-talls Linux-følelsen igjen.

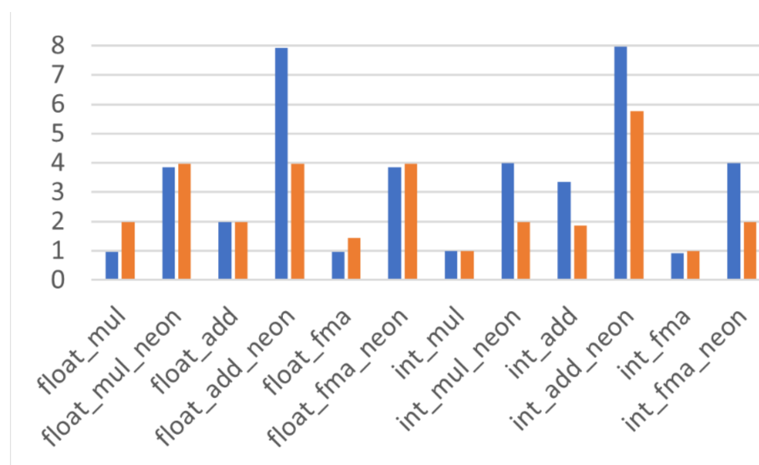
Uansett, en så enkel operasjon som *setcore()* ble til dette:

```
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() var livsviktig på X2. Hvis koden endte opp med å kjøre på en av de tvilsomme kjernene, ville det vært en total katastrofe, så jeg låste den på en normal A57-kjerne.

Om du ikke visste det, de kloke hodene hos Nvidia bestemte seg for å utstyre X2 med fire normale ARM A57-kjerner, og to som brukte super-avansert dynamisk rekompilering av koden, kjent som *Denver 2*. De hadde prøvd dette tidligere, i *Tegra K1*, som hadde *Denver 1*. Det virket langt under middels.

Jeg prøvde å finne ut hva i helvete som foregikk ved å gjøre noen ultra-syntetiske tester i assembler, og resultatene var... dubiøse. Her er utstedelsesfrekvensen for instruksjoner:



Blåaktig er vanlig A57, brunaktig/oransje er Denver 2. Det er Excel-standardfarger. Jeg aner ikke hvorfor alt må se ut som spy. Samme i Visual Studio 2019: Standard mørkt tema har bare brun/pastellfargede ting, og bakgrunnen er ikke

engang helt svart. Seriøst, gutter! Må være en Microsoft-greie, og innimellom en Linux-greie: Ingen har vel glemt den brune perioden til Gnome desktop på det tidlige 2000-tallet? Og spyfargevalget var veldig populært i Windows 8. Altså, når startsiden har duse, kjipe farger, så blir dagen bare trist.

Uansett sprikte resultatene i alle retninger. **Denver 2 var veldig rask på bit- og heltallsoperasjoner, men ga helt avvikende resultater for to veldig viktige instruksjoner: flyttalls *fma* og *mul*** med utstedelsesrater på bare 1 per syklus. Det ble verre: Generiske flyttalls arbeidsbelastninger så ut til å utløse en feil i kodeoversettingen, som gjorde tidsmålinger helt udeterministiske, og det ble skikkelig ille hvis flyttall- og Neon-instruksjoner var blandet. For å gjøre ting enda mer forvirrende: **Rene 64-bits *double* arbeidsbelastninger var omtrent 18% raskere enn 32-bit *float* på Denver 2**. Vent, hva? For å sitere spillet *Zero Wing* igjen: "Somebody set up us the bomb."

Jeg fikk en følelse av at det var en levning av Transmetas idéer: De laget noen prosessorer på begynnelsen av 2000-tallet som prøvde å emulere x86 CPU-er. Ja, det er omtrent så dumt som det høres ut. Uansett, Nvidia hadde lisensiert Transmeta sine patenter og ønsket å ta dem i bruk, antar jeg. Vill gjetting. Ikke om lisensieringen, men om bruken.

Og, som vanlig, må spørsmålet stilles: Var det mulig å komme rundt disse problemene og få Denver 2-kjernene til å oppføre seg noenlunde fornuftig? Tegn pekte i den retningen: Trikset var å stokke instruksjonene på en spesiell måte og ikke blande f.eks. float og Neon. En god del arbeid ble gjort, men dessverre ble det aldri fullført. Det var en tid hvor gjennomsnittlige sjefer ville ha mer gjennomsnittlig kode per time, og ikke noe av dette meningsløse nybrottsarbeidet.

Bra NVidia ikke brukte den teknologien igjen, da! Feil, uflaks igjen. *NVidia Xavier*, som kom i 2019, brukte et lignende system på alle åtte kjerner! De endret navnet på de fancy kjernene til *Carmel*, gitt at navnet Denver hadde fått noen riper i lakken, men var de fortsatt like lurvete? Her er et par sitater fra AnandTechs artikkel *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.*

Jeg sier ikke mer. Men om ting jevner seg ut, og Lotto-tallene går inn, så burde en Xavier snike seg inn i hulen her snart. For alt jeg vet kan de faktisk ha løst problemene med kodetranslasjonen, selv om AnandTech-sitatene ikke er lovende.

På tide å komme tilbake til saken igjen. Det er noen skaleringsrutiner i kontrollkoden i tilfelle noen ville rendre i høyere oppløsning og nedskalere til video eller noe sånt. En generisk, og en ARM-spesifikk. Den ARM-spesifikke er helt enkel og likefrem. Nesten. Veldig nær, stol på meg.

Først, vekk med vreinterpret- og vget-tøys:

```
#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

Lag en skaleringsfunksjon, anta at ting er justert osv. Det leses fra fire rader siden det er en faktor fire nedskalering:

```
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;
```

Siden det er 32-bit piksler leses 4 piksler * 4 = 16 bytes for hver rad:

```
            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;
```

Summér alt sammen med vår gamle venn *vaddl()*. Og en skare av reinterprets og sånt, siden det er, vel, ARM. Se for deg hvordan dette ville sett ut uten makroene. Eller med strikte kodeformateringsregler.

```
            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 ) ) );
```

Det gir et sett med høye/lave 16-bits resultater. Legg sammen:

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

Skyv svarene nedover. Hva burde være kjent nå? Den høye byten i hver 16-bits enhet er tom. Ubrukelig, egentlig. Husk det.

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

Gjør det samme for neste sett med 4 piksler:

```
            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() deler opp bytes i hver halvdel, så en halvdel vil inneholde interessante ting, og den andre tull. Heldig at de høye bytene var ubrukelige. Vel, egentlig ikke. Svært lite er basert på flaks i denne bransjen.

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

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

        }
    }
}
```

Hadde det ikke vært lurt å prosessere enda mer av gangen? Jo, sikkert, men koden bruker mange registre, og det var en god grunn til at alle radene var 64-bit justert: Tidligere versjoner kunne kjøre i enda lavere oppløsning, og bildene fra Android var alltid 64-bit justert.

vuzp (utpakking) og *vzip* (pakking) fortjener en omtale. Assembler-instruksjonen tar bare to parametere, så den må lese fra registrene, modifisere dem og skrive tilbake til de samme registrene. Jeg lurer på hvordan det håndteres internt, for latency på dem er ganske høy. Hvis det samme kan oppnås med et par *vtrn()* - transponering - i stedet, er det vanligvis billigere. AArch64 Neon løste problemet ved å fjerne dem og få Tilera-lignende versjoner i stedet. Men det var morsomme instruksjoner!

Og det åpenbare spørsmålet må stilles: Hvorfor brukte jeg ikke AArch64 Neon intrinsics der? Vel, det var to kompilatorer tilgjengelig for Android den gang. Én kunne gjøre AArch64 Neon, men ville ikke kompilere skjermkoden, og omvendt. Jeg prøvde å kompilere delene med forskjellige kompilatorer og linke dem sammen, men ble sittende og debugge linkeren en hel dag. Ga opp. Var ikke kritisk.

På tide med et nytt forsøk å komme tilbake til kontrollkoden. Jeg fant ikke noen god måte å representere felles data for shaderne på verten, så jeg laget en enkel struct for de som ikke hadde egne data og bare replikerte den.

```
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 ...
```

Kontrollblokkene på verten. Jeg brydde meg ikke mye med datamengden. Den er uviktig, måtte bare finne den største. Gjett hvem som vant? Det var alltid raytraceren, så bare kopier så mye data for hver ramme. Det betyr ingenting for hastigheten.

```
    // 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;
```

Lasting av shadere og kompilering og linking er trivielt, så det hoppes over. Fant sannsynligvis koden et eller annet sted på nettet. Eller, ikke helt trivielt, bare hundrevis av linjer med dødskjedelig kode.

Endelig: Hovedløkken, men først *setcore()* med passende kommentar fra den gang:

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

Så kom alt småplukket med å holde oversikt over hvilken shader som skal vises osv. Og kontrollere zoomingen, og sørge for at teksten mellom Galactic Dance og Glow City virkelig flytter seg ut av bufferen. Ja, det er meningen at det skal være sånn.

```
        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 );
            }

        }
```

Mye arbeid ble lagt i å få overgangene til å bli pene, derav den tidligere nevnte skalereren: Hvis det er en overgang, så tegnes først bakgrunnsteksten inn i et buffer av full størrelse, deretter shader i midten som blendes med alpha, og så til slutt teksten som overlapper. Så klippes det ut fra midten og skaleres på plass. Mye arbeid, men personlig synes jeg det ble kult.

Å kalkulere zoom ble derfor litt knotete. Kommentaren fra den tiden traff definitivt målet:

```
        // 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;
```

Det er sånn det ender når man har lite tid igjen. Og hvis det funker vil ingen bry seg om å fikse det senere. Det funket. Ingen fikset det. For en overraskelse.

Neste ut var shader prekalkulasjon per bilde. De viktigste har blitt dekket tidligere. Hele saken ble tegnet i et *Frame Buffer Object (FBO)*. Jeg vet at det ikke er anbefalt, men det fungerte også, så det ble heller aldri fikset. Ville nok vært bedre å rendre direkte til en tekstur.

CPU-koden var en-trådet og låst på kjerne 3, og CPU-belastningslinjen viser bare den kjernen. Interessant å merke seg at belastningen øker litt når raytraceren kjører, siden den gjør probing som forklart tidligere, men ikke mye. Rikelig med CPU-takhøyde.

En interessant observasjon:

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

Hvis koden ble kjørt med avslått display og ingen lagring av bilder, ville *glFinish()* ikke gjøre noe. Ja, ok, men det ødela testkjøringer. Litt som når optimalisereren fjerner all koden fordi svarene ikke brukes. Så den måtte tvinges til det ved å faktisk lese tilbake en piksel.

En ting som mangler er lydavspillingskoden. De to musikkbitene ble bare limt sammen og spilt av med kode som hadde en tvilsom lisens, så det måtte fjernes. Dessverre. Bare lim sammen de to nevnte musikkstykkene og spill dem av med en eller annen avspiller. Det var ikke mer avansert.

En artig greie ville vært å prøve å kjøre den på Xavier... i 4K. Må bare finne en ekstra Xavier et eller annet sted først. Får se om det er noen muligheter i neste ukes Lotto-trekning.

Dessverre hadde jeg ingen løsning for å ta opp videoen i sanntid, så den kunne kjøres i en modus der den lagret CPU- og GPU-belastningen i to tekstfiler. Ved neste kjøring kunne disse leses tilbake og settes inn i belastnings-indikatorene, og deretter lagret den det hele bilde for bilde. Tok lang tid, men videoen var i det minste en troverdig gjengivelse av originalen. Som stort sett var en kopi av andres arbeid. På tide å gjøre noe helt nytt.