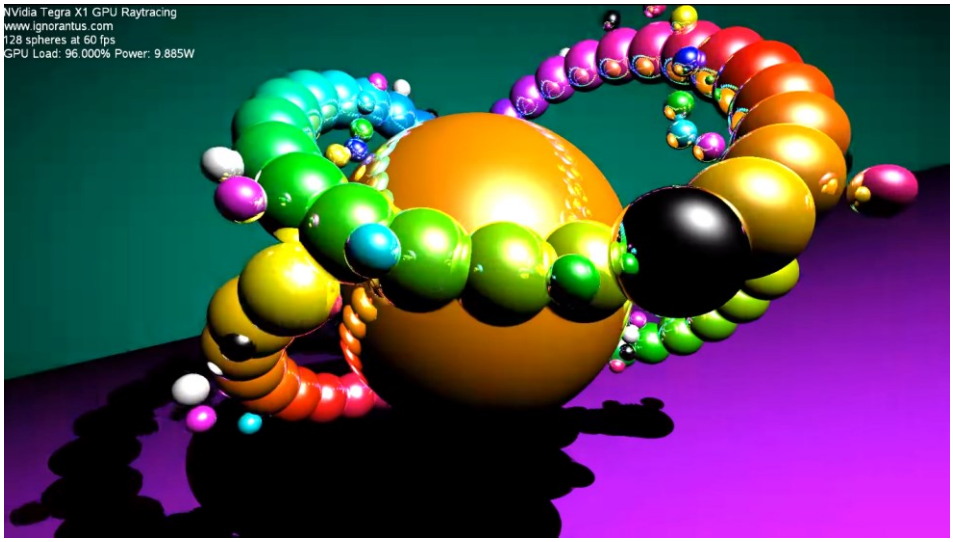
En GPU-raytracer

To storfilmer hadde premiere i 2017: *Blade Runner 2049* og *Alien: Covenant*. Begge var oppfølgere: Covenant til *Prometheus* fra 2012, antatt å være en ny retning i *Alien*-universet, og '2049 til den originale *Blade Runner* fra 1982, en av de beste filmene som noensinne er laget. Jeg synes de må vurderes uavhengig: Sammenligning med originalene kan ikke rettferdiggjøres. Jeg likte den uhyggelige følelsen i Covenant, og filmingen i '2049 er bare fantastisk.

2017 var også året jeg bestemte meg for å grave dypere i GPU-er. Oppfølgere var uaktuelt, så foreløpig ikke flere Julia-kvaternioner eller vanlige fraktaler. Jeg hadde jobbet med en GPU-raytracer i bakgrunnen en god stund, prøvd ut ting, rullet ut kode, formatert *if()*-uttrykk, kvittet meg med *if()*-uttrykk, lekt med flyttall som heltall, ledd av hvordan *bool* fungerte i GLSL, det vanlige. Så det var på tide å sette et mål for det: 128 kuler med refleksjoner og et plan med skygger i 60 bilder i sekundet i 1080p oppløsning. Og en himmel. På en NVidia Tegra X1.

Som nevnt hadde jeg ikke de riktige utviklingsverktøyene og slikt tilgjengelig. Mye av det jeg skrev for GPU-er den gang var ren gjetning, dvs. prøv tilfeldige ting til det går raskere. Den genererte GPU-koden var litt for høynivå, så det var noen ganger vanskelig å gjette hva som foregikk. Jeg hadde en teksteditor, en kompilator, en måte å trekke generert assembler-kode ut fra GPU-en, og det er egentlig alt som trengs. Som ordtaket sier. Ikke sikker på hvilket ordtak det kan være, men sånn er det.

Jeg startet med en raytracer for en *Epiphany* 16 kjerners CPU, vanligvis solgt på et *Parallella*-kort, skrevet av Shodruky Rhyammer kalt *Blobubska*[27]. Den var ganske enkel og pent strukturert. Den endelige versjonen av raytraceren min som presenteres her har ingen rester igjen av den, men jeg anbefaler likevel å se på koden hans. Og på Parallella-kortet. Fantastisk kul ting.



Raytracere er overraskende enkle: Plasser betrakteren et sted, tegn en skjerm foran, trekk linjer fra betrakteren gjennom punkter på skjermen, se om de treffer noe. Hvis de gjør det, reflekter og sjekk for treff igjen. Det er en logikk i det, pent forklart i artikkelen *Ray Tracing: Graphics for the Masses*[28] av Paul Rademacher, som jeg anbefaler som bakgrunnsmateriale.

Nå fungerte den tilnærmingen ikke veldig bra på X1. Rundt 80 kuler så ut til å være det teoretiske maksimum med en slik enkel tilnærming. Jeg hadde den fantastiske (eller dumme, jeg husker ikke hvilken) ideen om å flytte noe av jobben til ARM-kjernene, og dermed gjøre GPU-jobben enklere.

Først noen nyttige definisjoner. Dette er strukturene som brukes av CPU-kontrollkoden, og det skal være ganske tydelig hva som er hva. Det trengs noen kuler med farger, en verden med kuler, og det er stort sett alt. Kan det være

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

mulig å gjøre det uten virtuelle funksjoner, subklasser, konsepter, moduler og lambda-funksjoner? Beklager, ble kjørt ut på sidelinjen av et nytt lekespråk der. Jeg gjør det enkelt, som alltid.

Definer en kule:

```
typedef struct {

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

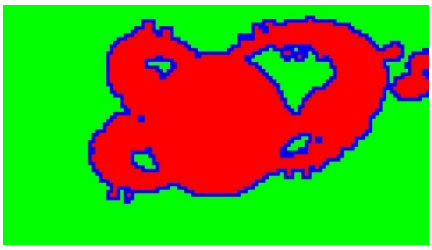
Bevegelsesinformasjon. Blir omtalt senere.

```
    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 inneholder informasjon om verdenen, dvs. alle kulene, bakken, lyset og slikt:

```
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;
```

I stedet for å gå rett til de harde detaljene i raytraceren, vil det være lurt å se på CPU-forenklingen først. Det vil gjøre det lettere å forstå raytraceren senere.



Kulene har en definert minimumsstørrelse, så det er nok å sjekke hjørnene på hver 16x16-blokk på CPU-en for å se om noe er der. GPU-en kan deretter eliminere store deler av koden fordi den da vet om området er dekket av en kule eller bakgrunnen eller en kombinasjon av det. I illustrasjonen over er grønt bare bakgrunn, rødt er bare kuler og blått er en kombinasjon.

Greit nok. Definer en funksjon, *world_probe(),* som gjør nettopp det. Først må kulene sorteres, slik at tidlig avskjæring kan gjøres raskere. Det er ihvertfall idéen. Så ut til å fungere ok. Det eneste interessante med dem er posisjonen og fargen:

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

Sorteringsfunksjonen sammenligner bare dybden:

```
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() gjør arbeidet:

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

Kopier posisjon og farge til *poscol*-tabellen:

```
    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;

    }
```

Verdenen har en stor kule i sentrum, og det er veldig sannsynlig at den blir truffet. Sett den alltid først i kulelisten. De andre er sortert etter dybde, teorien er at de vanligvis er bak den store hvis de er lenger borte.

Sorter alle kulene unntatt den første:

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

Og kopier dem tilbake til verdenen:

```
    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;
    }
```

Neste punkt er den faktiske hjørnetesten. Hvert hjørne av en 16x16 blokk blir evaluert, og hvis alle fire samsvarer, er det enten bare bakgrunn eller bare kuler. Det betyr at hele kuleberegningen kan elimineres på GPU-en hvis blokken bare har bakgrunn i seg.

Det er ikke nødvendig å gå over bord med optimaliseringene her: Hele funksjonen kjører på omtrent 5 ms per bilde på en enkelt ARM-kjerne, noe som gir god margin til 16.67 ms. Denne versjonen sjekker bare to rader og bytter dem om for neste sett.

```
#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-simpel interseksjonssjekk. Hopp over baksiden:

```
                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;

            }
```

Det er rimelig enkelt å skille himmel/plan også her, men jeg kunne ikke finne en enkel måte å utnytte det på da, så det er enten bakgrunn eller kule:

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

        }
```

Hvis bare én rad er ferdig, beregn den neste og start på nytt:

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

Ellers, bytt om. Spiller ingen rolle når det blir gjort:

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

Finn deretter riktig sted å lagre probe-verdiene og gjenta det over de to radresultatene:

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

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

Hvis alle fire har samme bit satt, er det enten kule eller bakgrunn. Er ikke dette artig kode? Noen mennesker finner det forvirrende. De er vanligvis ~~dårlige~~ gjennomsnittlige programmerere:

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

    }

}
```

Den delte bufferen som holder dataene til GPU-en er som følger, med probe-blokken:

```
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 er overkill, men det spiller ingen rolle. Det er god plass i den delte bufferen.

Så er det på tide å demontere GPU-koden. Balansekunsten her er å finne ut hvor mye den kan rulles ut uten å risikere å gå tom for registre. Mange variasjoner ble testet over en lang periode.

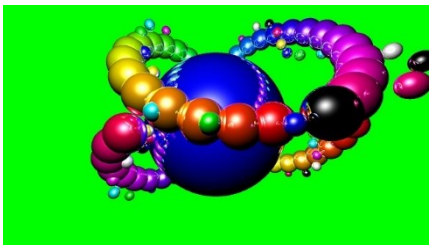
Jeg fant faktisk en artikkel som het *NV_GPU_program5*[29] som oppklarte en hel del: Den lister opp instruksjonene med registerspesifikasjoner og sånt, men den sa ingenting om instruksjonsenkodingen. Fillern. Så nær!

Bruk teksturkoordinaten til å bestemme hvor riktig probe-byte er og hent den:

```
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]);
```

Og bruk den til å hoppe over hele kuleberegningen hvis bare bakgrunn skal tegnes:

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



Det er sannsynligvis en kule, så det neste blir refleksjonsløkken. Forhåndslasting av data så ut til å fungere bra: Det er så lite kode i hver beregnings- og testblokk at lastingene faktisk ville føre til stans i pipelinen. Skrekk og gru! Og det er ikke noe poeng å sjekke treff på baksiden av kulene. Det er kun aktuelt hvis kulen er plassert rundt øyet, og det er de aldri. Men hva om det var tilfellet? Prøv det! Det vil se ut som fronten, men på baksiden, så ikke så bra. Ganske ubrukelig, faktisk. Det er selvsagt mulig å gjøre beregningen på den korrekte måten, men da vil det koste veldig mange GPU-sykluser.

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

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

            vec4 v0, v1, v2, v3;
```

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

Les inn data tidlig:

```
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 );
```

Iterer gjennom de 128 kulene i enheter på 32. Det blir en hard dags natt. Med utrulling:

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

Les inn neste sett:

```
    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 );
```

Gjør treffberegninger for første sett:

```
    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 );
```

Se etter treff i denne blokken av fire:

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

Treff. Hvilken av de fire er ukjent, men det er minst én i dette settet. Negative verdier vil gjøre ting mye vanskeligere (fire *if()-er*), så det trengs en rask måte å eliminere dem på. Det er et triks for det:

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

Tegnbiten til en *float* skjøvet inn i hver bit gir NaN ved negativt og 0 ved positivt tall, og det tar seg av problemet. Den laveste positive verdien kan deretter finnes med *min()*. Men det hadde vært for enkelt. Kompilatoren sleit og endte opp med å generere mange overflødige MOV-instruksjoner, uvisst av hvilken grunn. Uflaks. Som androiden Walter i filmen *Alien: Covenant* sa: "When one note is off, it eventually destroys the whole symphony."

Men: Det er en annet triks som kan brukes: Inverter verdiene og testene. Mer kode vil genereres, men færre effektive operasjoner. Stol på meg. Jeg prøvde ut mange varianter. Prøv det selv! Det er forbløffende mange sånne triks som er mulige. Så gjør dette i stedet:

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

Sjekk om dette treffet er nærmere, og hvis det er nærmes, så lagre den. Men det må være mulig å finne ut hvilken senere. Å bare lagre alle verdiene vil ikke fungere, siden det ikke er nok registre igjen og det vil ende opp med resirkulering av enkle 32-bits registre her og der. Men ved å lagre tre av dem pluss baseposisjonen blir registerbanken fornøyd, noe som kompliserer ting litt, men ikke mye, utenfor løkken:

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

    }
```

Kompilatoren spyr ut en "might be uninitialized" advarsel for *rt_t0*. Hvis den blir initialisert går to sykluser på dynga for hver iterasjon per kjerne. Det blir mange til sammen. En hel haug, faktisk. Kan noen være så snill og flytte testen i kompilatoren? Alle kompilatorer gjorde det galt den gangen, og ganske sikkert nå også. Takk.

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

Neste trinn er å manuelt rulle ut alt dette. Så gjenta greiene over syv ganger til her. Offsetene ol. må selvsagt justeres. Koden ble skrekkelig lang, men det betyr ikke all verden. Alle kjernene gjør det samme likevel, så alt er under kontroll. Tror jeg. Ganske sikker.

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

Invers distanse mindre enn 3.0 var vanligvis nær nok. Sparte tid. Det feiler ikke veldig ofte:

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

Om ingenting ble truffet, er det bare å hoppe ut:

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

Noe er truffet, så finn ID-en. Det er ganske enkelt å gå fra delt register til faktisk ID. Bare se om noe i *rt_t0.xyz* (som åpenbart er *yzw*) samsvarer nøyaktig og legg sammen verdiene i testvektoren. Enkelt! Så mye dumt man gjør for å redde registre! Jeg er sikker på at det finnes viktigere ting å redde, som hvalbestanden eller myrsnipas våtmarksområder:

```
        // 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;
```

Flytt posisjon og oppdater fargen med litt diffusjon og spekularitet så den ser kulere ut:

```
        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;
```

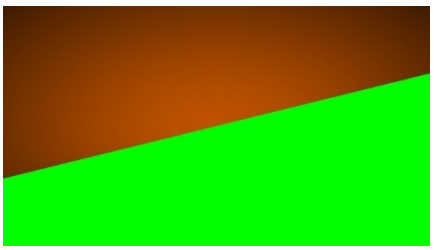
Bruk GLSL *reflect()* for å reflektere strålen. Burde vel ikke bruke den, siden den, som diskutert i det andre GPU-kapittelet, ikke vil reorganiseres skikkelig. Det er utenfor hovedløkken, så det er ingen krise:

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

Nå vil den enten fortsette å reflektere til maksimalt antall refleksjoner er nådd, eller det er bom. Om noe ble truffet/reflektert, så er jobben gjort:

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

}
```

Med alle kuleberegningene ute av veien, handlet resten av koden om himmelen og skyggeplanet.



Først, sjekk om planet er truffet:

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

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

Det var det ikke, så tegn en pulserende himmel og avslutt:

```
    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;
}
```

Planet er truffet. Flytt *ray_pos* til planet og få *ray_dir* til å peke på lyskilden:

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

Å sjekke skygge var litt enklere, siden det bare er interessant om noe i det hele tatt treffes, ikke hva det er eller hvor langt unna. Her følges samme struktur som i kuleberegningen, bortsett fra at det rulles ut litt annerledes:

```
// 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 );
```

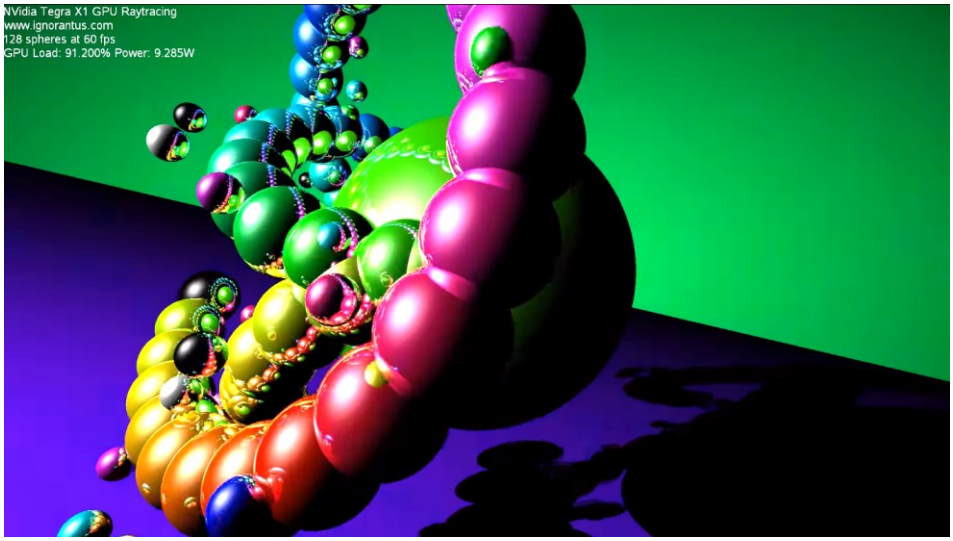
Sett inn det samme tre ganger til her, med oppdaterte referanser. Så sjekk om noe ble truffet. Jeg liker denne programmeringsstilen: Gjør beregninger først, sjekk resultatene senere. Enkelt og rett på sak:

```
    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;

}
```

Avgjør så om det er skygge eller plan, og beregn den endelige fargen. Det er enkelt: Hvis maksimalt antall iterasjoner ble nådd er ingenting truffet, så tegn bakgrunnsplanet, eller vice versa. Planet hadde en litt ullen farge hvis det var total bom. Ikke superfint, men det var billig. Bare slang noe sammen:

```
    // 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 );

}
```

Dette vil raytrace 128 kuler på NVidia Tegra X1. Lett som en plett!



Videoen med tittelen *GPU Raytracing on NVidia Tegra X1 v2.0* viser frem raytraceren. Igjen glemte jeg å notere hvem som laget musikken. Men det er en ganske fengende melodi som jeg brukte i flere testvideoer. Beklager, musikere! Alle nyere videoer jeg har laget har dette i orden.

Det som mangler her, er åpenbart CPU-kontrollkoden. Den omtales i neste kapittel.

En kul test var å prøve å tvinge X1 til termisk shutdown ved å maksimere beregningsbelastningen. Raytraceren produserer ganske brukbare delvis tilfeldige data, så jeg slang sammen noen beregninger på dem til lasten var så nær 100 prosent som mulig, og strømforbruket så høyt som mulig. Både X1 og X2 har enkle metoder for å lese ut de verdiene.

```
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 );
}
```

Det er verdt å merke seg at notatene var litt tynne på dette punktet. Funksjonen sin01() gjør antageligvis det det høres ut som. For å få koden til å virke i 2017-versjonen av raytraceren, bør sorteringen av kulene slås av i kontrollkoden. Og kanskje en god ide å dele ned OBJNUM litt. Og flytte på normaliseringen.

I den gamle 2016-versjonen så det faktisk ganske bra ut. Sjekk ut videoen *NVidia Tegra X1 Load and Power Test*. Musikken heter *Enemy Ships* og er laget av Jason Shaw, eier av nettstedet audionautix.com. Melodien er tung, men rask siden videoen bare er 30 sekunder lang.

GPU-kraften nådde toppen på rundt 11.5 watt. Jeg antar at X1-en ville ha tatt fyr med litt minne- og CPU-belastning. NVidia hevder selv 15 watt strømtrekk ved full belastning for Jetson TX1-modulen. Det er i beste fall konservativt.