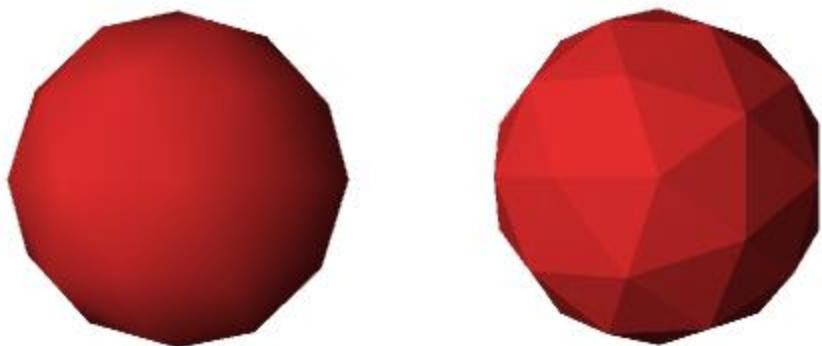
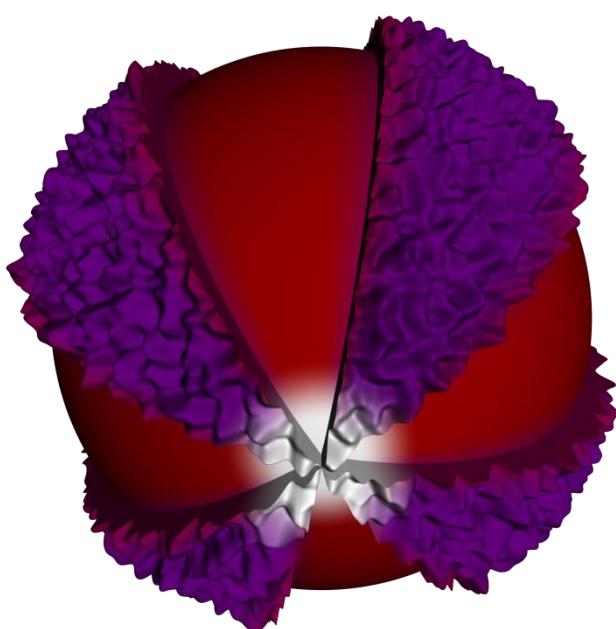


Rapport i Avancerad Datorgrafik



Alexander Milton
19940510-8136
B13alemi
Dataspelutveckling – Programmering
Avancerad Datorgrafik
VT 2014

Uppgift 1



Kod

enkelshader.sl

```
surface enkelshader(
    color purple      = (0.5, 0.0, 0.6); // Purple color
    color red         = (0.6, 0.2, 0.0); // Red color
    color white        = (1.0, 1.0, 1.0); // White color
    float frequency   = (4.0);           // Frequency of meridians (stripes) in the
sphere
    float diffuseStrength = (0.9);       // Value defining the intensity of the light
being shed on the object
    float roughness   = (0.1);           // The level of irregularity
    float bumpheight  = (0.8);           // The depth of the texture bumps
    float f            = (0.1);           // The interval (size) of the white area
)
{
    float segments     = mod((s * frequency), 1.0); // Iterate frequencies through the
meridians (s) to create sphere segments
    float alignedMeridian = mod((s * frequency + 0.125), 1.0); // Additional frequency
used to correctly align with the meridians

    // Smoothly transition from 0.25 units until 0.5 units at each frequency
    // First value indicates the end of solid color (from 0.0)
    // Second value indicates end of smoothed color, i.e. when that color ceases
    float f1 = smoothstep(0.25, 0.5, segments);
    float f2 = smoothstep(0.75, 1.0, segments);
    float f3 = smoothstep((f - 0.065), f, v);

    // Calculate a new normal by altering the global view point and create a noise effect
    // coordinated using bumpheight
    P += step(0.5, alignedMeridian) * (noise(P * frequency) * N * bumpheight);
    N = calculatenormal(P);

    // Calculate the normal of the shape
    normal normalValue = normalize(N);

    // Create a base color mix consisting of red and purple, smoothly blending them using
    // the f1 and f2 smoothstep intervals
    // Create a new color mix, adding white to the base mix and switching/blending between
    // the interval of the third parameter f3
    color baseColor = mix(red, purple, f1-f2);
    color whitedBaseColor = mix (white, baseColor, f3);

    // Diffuse the color to a defined intensity by multiplying it with a diffuse function
    color diffusedColor = mix (whitedBaseColor, whitedBaseColor * diffuse(normalValue),
diffuseStrength);

    // Rendering instruction
    Ci = diffusedColor;
    Oi = Os;
}
```

Reflektion

En stor del av min ansats till att strukturera detta, för mig, något förvirrande och stundvis oregelbundna programmeringsspråk finner sitt ursprung i formella och informella C-konventioner. Det förefaller mig ovant att med en spontan attityd arbiträrt deklarera konkreta värden i min kod, snarare än att samla dem i en gemensam yta eller konstruktor. Av den anledningen har jag strävat efter en välfyllt parameterlista i början av min shader. Detta har gjort det enkelt för mig att översiktligt hantera och kontrollera mina variabler.

Högst upp i min parameterlista finner vi färdigdefinierade färger med logiskt representerade variabelnamn; lila, orange och vitt. Detta för att underlätta framtid färghantering. Att tilldela, blanda eller använda en färg blir betydligt enklare och tydligare när färgens RGB-kod representeras av ett variabelnamn.

Vi finner även en rad flyttalsvärden: *frequency*, *roughness*, *bumpHeight*, *diffuseStrength* och det något intetsägande *f*. Följande parameterlistan deklarerar *segments* och *alignedMeridian*.

Variablernas betydelse lyder som följer:

- Värdet representerat av *frequency* påverkar hur många segment (meridianer) sfären ska delas upp i. 4.0 innebär att vi får fyra linjer som delar sfären i åtta segment.
- Värdet representerat av *roughness* påverkar hur kraftig faktor av oregelbundenhet som bör tillämpas på de ytor som tilldelas en ojämnn yta.
- Värdet representerat av *bumpHeight* påverkar hur djupa försjunkningar den ojämna ytan skall ha. Ett högre värde innebär djupare ojämnheter.
- Värdet representerat av *diffuseStrength* påverkar intensiteten av det diffusa ljuset som kastas på det renderade objektet.
- Värdet representerat av *f* påverkar hur stor omfattningen intervallet på en smoothstep som renderar en vit "pol" på sfären ska ha. Ett högre värde innebär i praktiken att den vita ytan blir större/sträcker sig längre från sin egen origo.
- Värdet representerat av *segments* itererar genom meridianerna för att skapa segment ur sfären.
- Värdet representerat av *alignedMeridian* är en extra frekvens som "stöd" för den ursprungliga *frequency*-variabeln. Den används för att ge en korrekt justerad frekvens mot meridianerna.

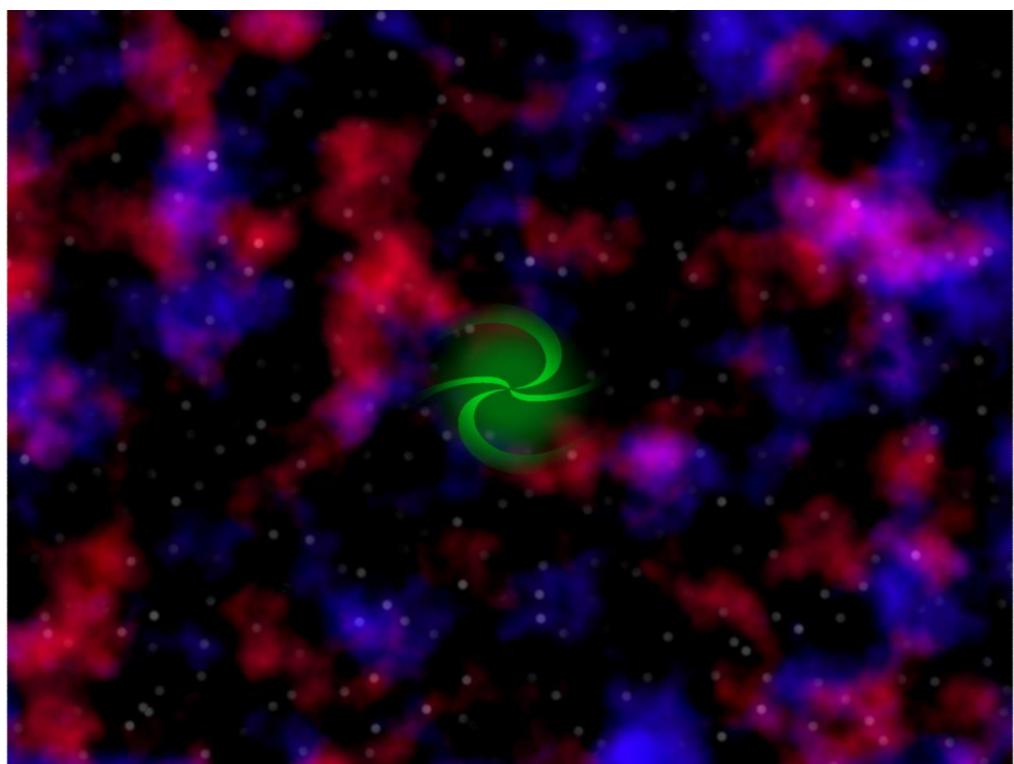
Procedurella Shaders kontra Färdiga Texturer

Det finns en rad olika fördelar med procedurellt beräknade texturer framför målade/ritade/skannade/fotograferade texturer. Flera av dessa anspelar på behovet av dynamisk förändring. En statisk bild är absolut i sitt utseende, något som inte alltid är önskvärt.

En procedurellt skapad textur kan förstoras, förminskas, förvrängas, lappas sömlöst och ökas respektive minskas i sin upplösning på vis som vore omöjliga för en statisk bild. Man kan även med fördel justera enstaka parametrar för att få varierande och multilaterala representationer av sin textur, samtidigt som bilderna tar minimal fysisk plats på hårddisken i kontrast till en mängd statiska bilder. Allt detta är givetvis endast fördelaktigt om det är värt besväret att skapa en helt egen shader för en enda textur. Det krävs med andra ord en förutsättning om att texturen behöver nå krav på flexibilitet och dynamik som berättigar det faktiska arbetet.

Det är inte alltid smidigare att skapa en specifik bild än att ”programmera” en textur. En statisk bild behöver eventuellt en konstnär, utrustning, material och tid för att utföra en rad olika processer. Dessutom kommer resultatet inte kunna ta hänsyn till saker som vilken ljussättning bilden bör förhålla sig till om detta inte är specificerat till extrem detaljnivå. De olika typerna av ljus och deras egenskaper är en handfull vetenskaper av sig själva, och inte ens de mest avancerade (eller tidskrävande) post-produktionseffekterna kan konkurrera med förmågan att anpassa samtliga faktorer på plats. Däremot saknar procedurella texturer förmågan att på ett tidseffektivt vis ändra och skapa övertygande och precisa detaljer. När en konstnär lätt kan lägga till, ändra eller ta bort exempelvis en fläck kan det vara mycket svårare att programmera en så pass skarp specifikation.

Uppgift 2



Kod

stars.sl

```
#define snoise(p) (2 * (float noise(p)) - 1)

float fBm (point p; uniform float octaves, lacunarity, gain)
{
    uniform float amp = 1;
    varying point pp = p;
    varying float sum = 0;
    uniform float i;

    for (i = 0; i < octaves; i += 1) {
        sum += amp * snoise (pp);
        amp *= gain;
        pp *= lacunarity;
    }
    return sum;
}

// The Worley cell noise function creates cells and measures the distance between them,
// offsetting them by random values. This creates an irregular "noisy" cell pattern.
// We utilize these scattered cells by inverting and narrowing their color values, creating a
// realistic
// and spectacular night sky texture with distant, glowing stars.

void voronoi_f1f2_2d (float ss, tt; output float f1; output float spos1, tpos1; output float
f2; output float spos2, tpos2;)
{
    float jitter=1.0;
    float sthiscell = floor(ss)+0.5;
    float tthiscell = floor(tt)+0.5;
    f1 = f2 = 100;
    uniform float i, j;

    for (i = -1; i <= 1; i += 1) {
        float stestcell = sthiscell + i;
        for (j = -1; j <= 1; j += 1) {
            float ttestcell = tthiscell + j;
            float spos = stestcell + jitter * (cellnoise(stestcell, ttestcell) - 0.5);
            float tpos = ttestcell + jitter * (cellnoise(stestcell+23, ttestcell-87) -
0.5);
            float soffset = spos - ss;
            float toffset = tpos - tt;
            float dist = soffset*soffset + toffset*toffset;

            if (dist < f1) {
                f2 = f1;
                spos2 = spos1;
                tpos2 = tpos1;
                f1 = dist;
                spos1 = spos;
                tpos1 = tpos;
            }
        }
    }
}
```

```

        }

        else if (dist < f2) {
            f2 = dist;
            spos2 = spos;
            tpos2 = tpos;
        }
    }

f1 = sqrt(f1);  f2 = sqrt(f2);
}

surface stars(
    color white          = (1.0, 1.0, 1.0);      // White color
    color grey           = (0.5, 0.5, 0.5);      // Grey color
    color black          = (0.0, 0.0, 0.0);      // Black color
    color red            = (1.0, 0.0, 0.2);      // Red color
    color blue           = (0.2, 0.0, 1.0);      // Blue color
    float starlightIntensity = (1.0);           // Modifies the intensity of the
starlight
    float lacunarity     = (6.0);              // Level of "gappiness" or inhomogeneity
    float octaves         = (2.0);              // Number of noise layers to be summed
    float gain            = (0.4);              // The level of effective contribution by
each layer
)
{
    // Initiate variables to be used as parameters for the Worley Noise Cell algorithm
    float f1;
    float spos1, tpos1;
    float f2;
    float spos2, tpos2;

    // Execute the Worley Cell Noise algorithm
    voronoi_f1f2_2d(s*0.1, t*0.1, f1, spos1, tpos1, f2, spos2, tpos2);

    // Create two smoothsteps to manage the strength and fade of the starlight
    // The second smoothstep will take an in-parameter to control the total light strength
    float fla;
    float flb;
    fla = smoothstep(0.05 * starlightIntensity, 0.15, f1);
    flb = smoothstep(0.05 * starlightIntensity, 0.8, f1) * 0.5;

    // Sum and invert the two smoothsteps to gain a potent center light for effect
    f1 = 1 - (fla+flb);

    // Darken the color range by applying a gray color
    color newColor = f1 * grey;

    // Mesh the stars to give the scene more realism
    float fBm1 = clamp( abs(fBm(P*0.05, octaves, lacunarity, gain) ), 0, 1);
    newColor = (newColor * fBm1);
}

```

```
// Add red and blue nebula clouds and neatly layer them together
// Mesh the clouds to give them a naturally irregular appearance
// Remember to move the point P in different directions to spread the mesh layers
color fBm2a = clamp(fBm((P+231)*0.05, 5, lacunarity, gain), 0, 1)*red;
color fBm2b = clamp(fBm((P-231)*0.05, 5, lacunarity, gain), 0, 1)*blue;
newColor = (newColor + fBm2a + fBm2b);

// Finalize the output by adding the color
Ci = Cs * newColor;
Oi = Os;
}
```

galaxy.sl

```
surface galaxy(
    float reachFrequency      =      (12.0);           // The diameter/"reach" of the spirals
    float ovalFactor          =      (3.5);           // How intensely the spirals are twisted
    float galaxySize          =      (0.5);           // The size/diameter of the final galaxy
    color galaxyColor         =      (0.0, 0.4, 0.045); // The color of the galaxy
)
{
    // Create alternative coordinates depending on u and v.
    float uu = (u - 0.5) * reachFrequency;
    float vv = (v - 0.5) * reachFrequency;

    // Create two points, one center (origo) and one at the alternative coordinates uu and vv.
    point p1 = point(uu, vv, 0);
    point p2 = point(0, 0, 0);

    // Calculate the distance between the origo point (p1) and the new point (p2).
    float dist = distance(p1,p2);

    // Create spiraling stripes from the center towards the edges of the texture.
    // atan(uu, vv) is utilized to create the curved lines.
    float spirals = step(1.2, mod(atan(uu, vv * ovalFactor) + dist, 1.57));

    // Smoothstep the spirals to fade out the stripes toward the edges
    spirals = spirals * (clamp(1 - (smoothstep(1.0 * galaxySize, 4.0 * galaxySize, dist)), 0, 1));

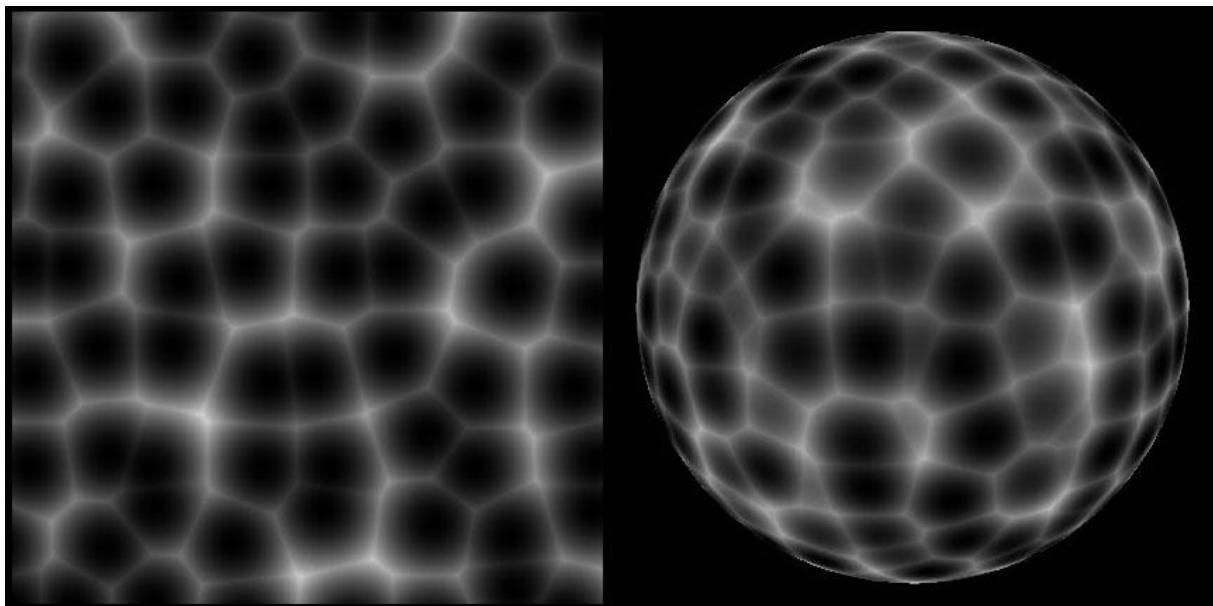
    // Add a glowing "haze", filling the galaxy with a lit substance
    spirals = spirals + clamp(1 - (smoothstep(0.8 * galaxySize, 3.5 * galaxySize, dist)), 0, 1);

    // Finalize the output by adding color
    Ci = spirals * galaxyColor;
    Oi = spirals;
}
```

Väsentliga Saker i Shaderkoden

```
// Execute the Worley Cell Noise algorithm  
voronoi_f1f2_2d(s*0.1, t*0.1, f1, spos1, tpos1, f2, spos2, tpos2);
```

Ur flera aspekter är denna rad kod en av de absolut mest fundamentala för shadern. Koden är ett funktionsanrop till Worley Cell Noise-algoritmen och i sig inte skapad av mig, men kritisk för shaderns funktion. I och med att shaderns syfte är att skapa en trovärdig och effektfull scen med rymden som motiv finns ett starkt grundläggande behov av substans. Algoritmen agerar i denna kontext som en slags slumpgenerator och utgör grunden till det mönster scenen kommer nyttja. Den visuella slutprodukten av Cell Noise-algoritmen ser ut ungefärligt exempelbilden nedan (tagen från nätet, ej från just denna shader).



```
float lacunarity = (6.0);      // Level of "gappiness" or inhomogeneity  
float octaves     = (2.0);      // Number of noise layers to be summed  
float gain         = (0.4);      // The level of effective contribution by each layer  
  
// Mesh the stars to give the scene more realism  
float fBm1 = clamp( abs(fBm(P*0.05, octaves, lacunarity) ), 0, 1);
```

Med ett brusigt mönster som bas behöver vi bearbeta det för att skapa en textur som liknar en stjärnhimmel, det vill säga jämt och oregelbundet utspridda stjärnor av rimligt varierande ljusstyrka och någorlunda rund form. Vi gör detta genom att maska vår nuvarande textur genom flera lager av bruslager, så kallat Fractional Brownian Motion. En av funktionens faktorer, ”lacunarity”, utgör ett mönsters ojämnhet/oregelbundenhet eller ”luckighet”. Faktoren ”octaves” är ett heltal som representerar antalet konkreta bruslager som ska summeras i funktionen, medan ”gain” syftar till hur pass effektiv hänsyn varje lager ska tas. Högre gain innebär att lagrens egenskaper har en större effekt på slutresultatet.

En likande process utförs även för att skapa scenens nebulosamoln.

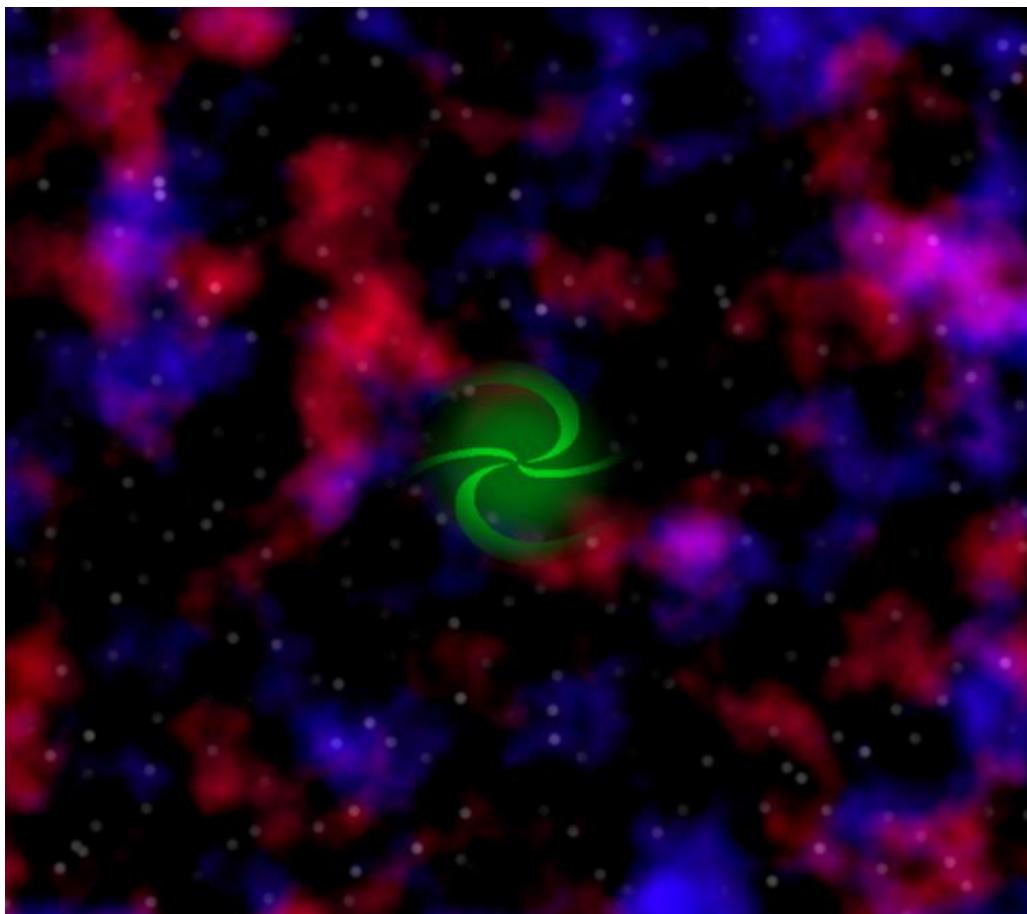
```
// Create spiraling stripes from the center towards the edges of the texture.  
// atan(uu, vv) is utilized to create the curved lines.  
float spirals = step(1.2, mod(atan(uu, vv * ovalFactor) + dist, 1.57));
```

Funktionen i sig är, bortsett från den namngivna variablerna, väldigt intetsägande. Vad den gör är att skapa de spiralerande armar som utgör galaxens struktur och karaktäristiska utseende. Funktionen använder step-anropet för att itererar igenom den geometri som ska skiftas mellan att vara färgad/synlig och vilken som i slutändan kommer vara transparent, alltså anger vilka delar av ytan som ska utgöra en arm och vilka som inte ska det. Atan-anropet syftar matematiskt till trigonometrifunktionen arc-tangent, vilken används för att beräkna de böjande linjer som avgränsar armsegmenten. Tillsammans med en faktor för att låta användare i viss mån kontrollera armarnas omfattning lyckas kodraden skapa ett primitivt men dugligt mönster att illustrera galaxen genom.

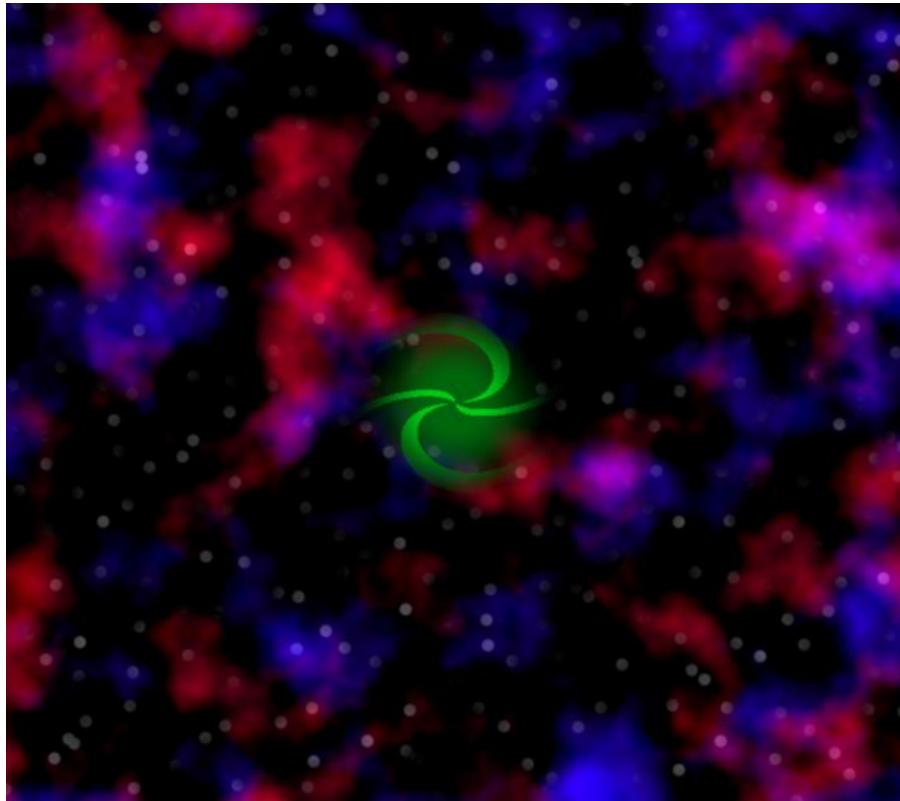
Låt Oss Leka med Lite Värden

Följ länken nedan för att se en pedagogisk .gif-animation över de olika versionerna av den parametermanipulerade texturen:

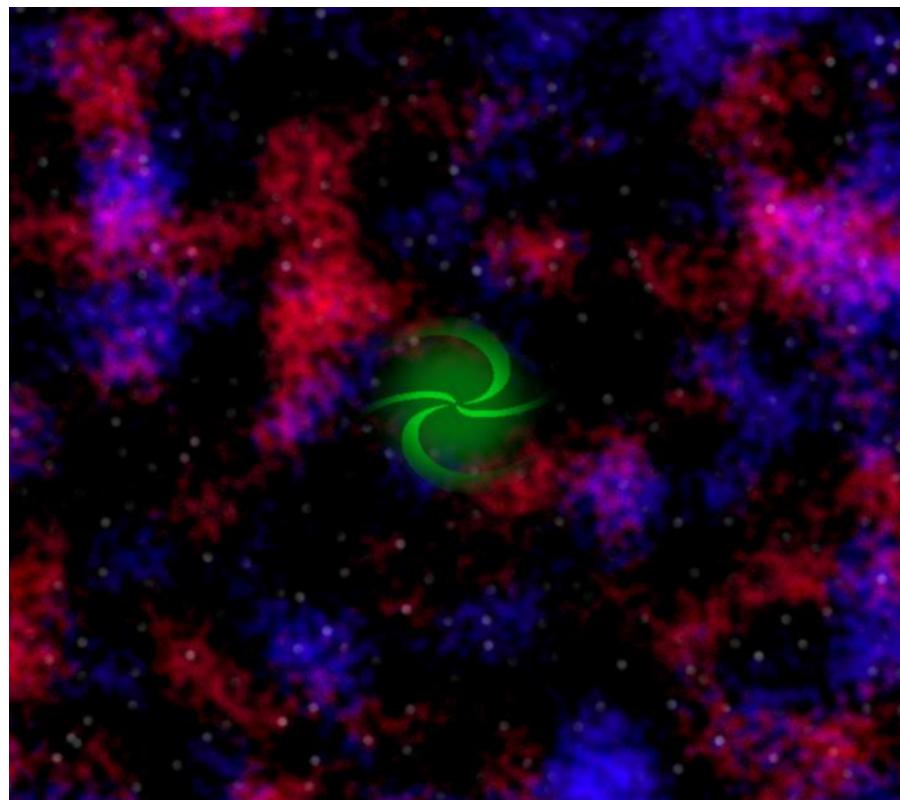
<http://imgur.com/K05WSUs>



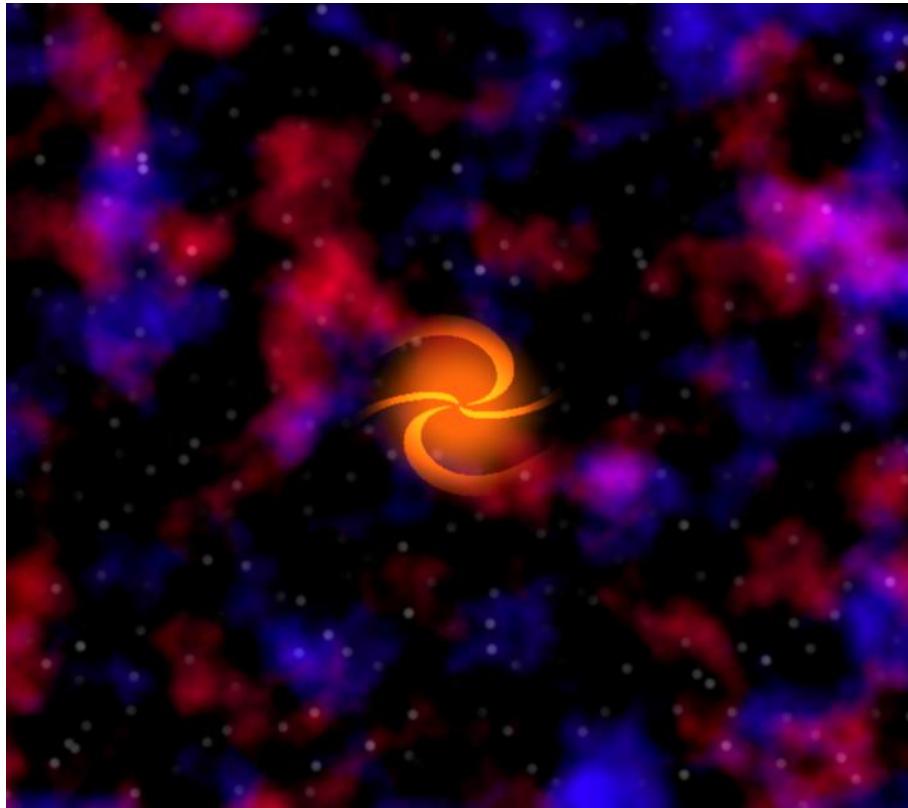
Detta är en oförändrad version av rymdlandskapet. Galaxen är grön och relativt liten. Stjärnorna är diskreta i sin storlek och nebulosorna är konsistenta och molnliknande.



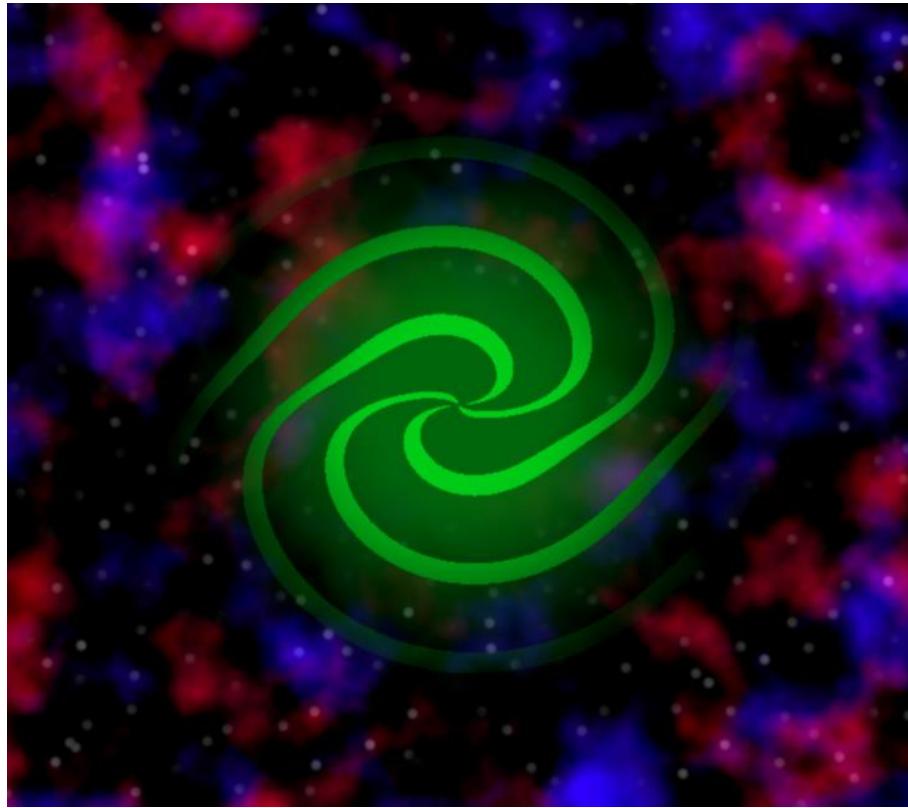
I denna version är parametern `starlightIntensity` ökad från 1.0 till 2.0 vilket innebär att stjärnljuset, som kontrolleras av en smoothstep, får ett högre tröskelvärde. Detta gör att stjärnorna förefaller lysa starkare eller vara större än i ursprungsbilden.



Här är parametern `lacunarity` ökad från 2.0 till 6.0. Detta gör att frekvensen i mönstrets "luckor" som maskar nebulosamolnen blir tätare. Resultatet blir att molnen förlorar sin oslipade mjukhet och istället får en tydligare definierad grynhet.



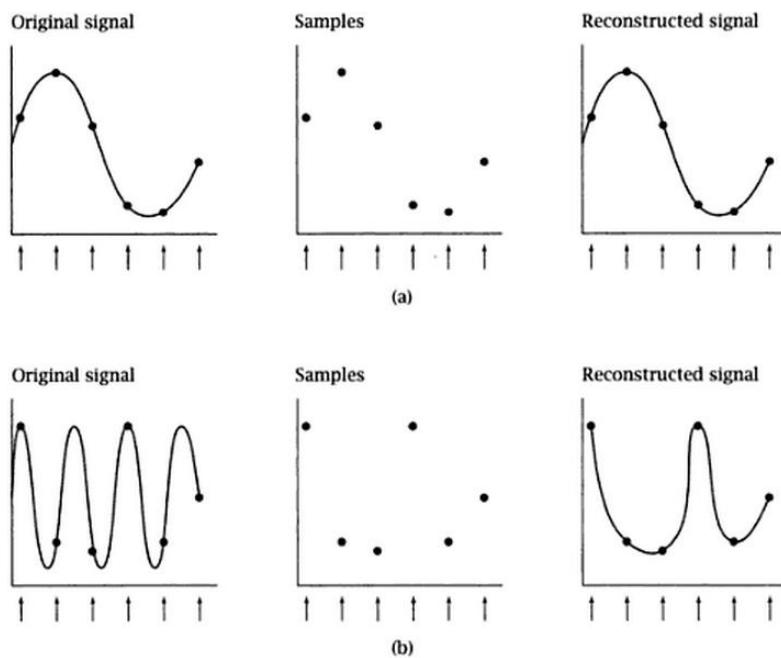
Här har `galaxyColor` som styr galaxens färg ändrats från `(0.0, 0.4, 0.045)` till att inkludera ett fullt rödvärde: `(1.0, 0.4, 0.045)`. Resultatet blir helt enkelt att galaxen blir orangefärgad i sitt utseende.



Här har parametern `galaxySize` ökats från `0.5` till `1.5`. Parametern agerar som en faktor i den `smoothstep` som används för att skapa galaxens spiraler och "dimma". Genom att öka värdet på faktorn ökar vi även den konkreta storleken på galaxen.

Problem som kan Uppstå med Aliasing i Procedurella Texturer

När en signal av ljud, mönster, bilder eller i vårt fall, pixlar, ska rekonstrueras i ett illustrativt syfte kan problem uppstå. Signalen kan inte alltid läsas av i sin helhet, utan identifieras av sina signalvågor genom stickprover, så kallas "samples". När en signal sampelas kan en bild av vågens egenskaper återskapas, men om antalet samples understiger det dubbla antalet frekvenser i signalen (den så kallade Nyquistfrekvensen efter matematikern Henry Nyquist) kommer så kallad vikning, eller "aliasing" att uppstå. Detta innebär att en signal kan förlora sina korrekta egenskaper på grund av att för få sampelas hämtas till stöd för rekonstruktionen. Se figuren nedan (tagen ur boken *Advanced Renderman*).



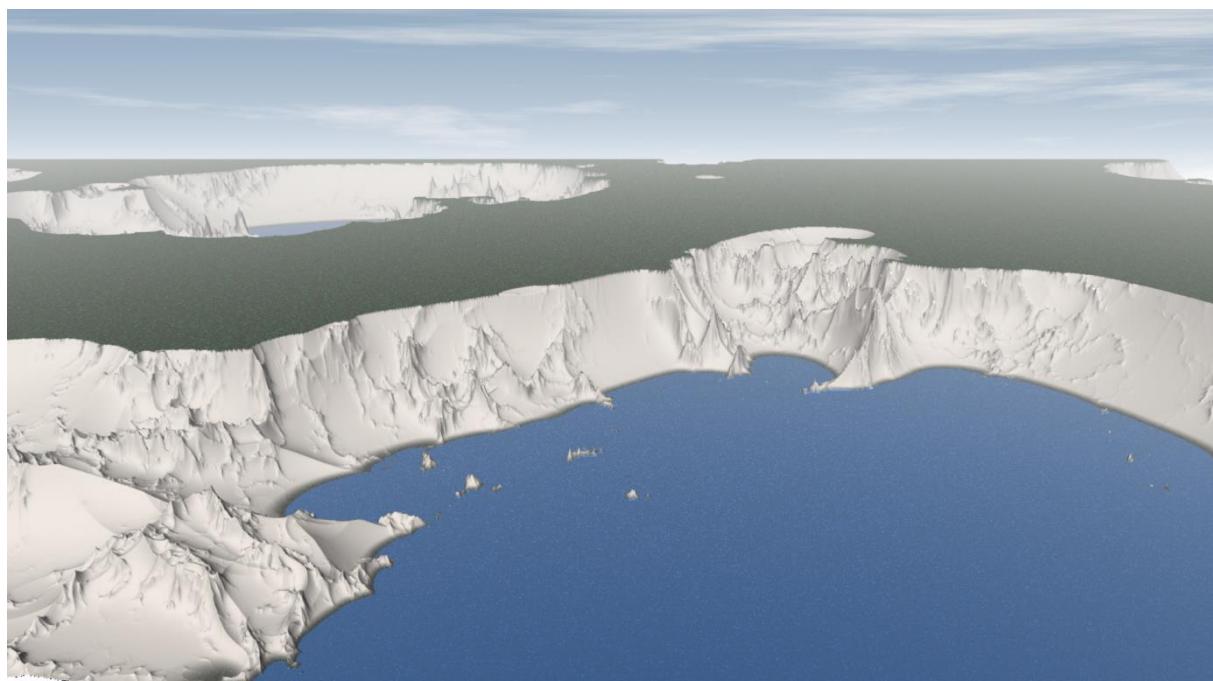
Våra procedurella texturer utsätta för detta problem, till stor del på grund av att sampling av shaders i Renderman sker i ett regelbundet rutnätsutförande, medan skärmrymden sampelas i en stokastisk och oregelbunden ordning vilket orsakar en konflikt i signalregistreringen.

Det finns ett antal sätt att undvika aliasing i grafik. De flesta grundar sig i praktiker för att manipulera antingen signalerna eller samplingsprocessen. Att vrinda upp den råa signalen, exempelvis att höja upplösningen på en grafisk representation, är ett sätt att motverka aliasing med viss hänsynslöshet mot effektivitetsaspekter. Oftast krävs mer rationella och sofistikerade metoder för att lösa problemet.

En mer subtil metod är att utkräva fler samples ur signalen (exempelvis genom supersampling eller multisampling) för att nå en mer precis avbildning, på bekostnad av beräkningskraft. En annan metod med diskuterbar effekt är att ändra just hur samplingen sker, det vill säga var proverna tas i den avsedda rymden, för vilken det finns ett antal definierade algoritmer.

Signalfilter och *motion blur* är andra exempel på tekniker med syfte att åtgärda problem med aliasing och signalmottagning med otillfredsställande resultat.

Uppgift 3



Kod

mountain.sl

```
#define snoise(p) (2 * (float noise(p)) - 1)

float RidgedMultifractal(point p; uniform float octaves, lacunarity, gain, H, sharpness,
threshold)
{
    float result, signal, weight, i, exponent;
    varying point PP=p;

    for( i = 0; i < octaves; i += 1 ) {
        if ( i == 0) {
            signal = snoise( PP );
            if ( signal < 0.0 ) signal = -signal;
            signal = gain - signal;
            signal = pow( signal, sharpness );
            result = signal;
            weight = 1.0;
        }
        else{
            exponent = pow( lacunarity, (-i*H) );
            PP = PP * lacunarity;
            weight = signal * threshold;
            weight = clamp(weight,0,1);
            signal = snoise( PP );
            signal = abs(signal);
            signal = gain - signal;
            signal = pow( signal, sharpness );
            signal *= weight;
            result += signal * exponent;
        }
    }
    return(result);
}

float fBm (point p; uniform float octaves, lacunarity, gain)
{
    uniform float amp = 1;
    varying point pp = p;
    varying float sum = 0;
    uniform float i;

    for (i = 0; i < octaves; i += 1) {
        sum += amp * snoise (pp);
        amp *= gain;
        pp *= lacunarity;
    }
    return sum;
}

float turbulence (point p; uniform float octaves, lacunarity, gain)
```

```

{

uniform float amp = 1;
varying point pp = p;
varying float sum = 0;
uniform float i;

for (i = 0; i < octaves; i += 1) {
    sum += amp * snoise (pp);
    amp *= gain;
    pp *= lacunarity;
}
abs(sum);
return sum;
}

// The Worley cell noise function creates cells and measures the distance between them,
// offsetting them by random values. This creates an irregular "noisy" cell pattern.
/* Voronoi cell noise (a.k.a. Worley noise) -- 2-D, 2-feature version. */
void voronoi_gravelNoise (float ss, tt; output float gravelNoise; output float spos1, tpos1;
output float f2; output float spos2, tpos2;)

{
    float jitter=1.0;
    float sthiscell = floor(ss)+0.5;
    float tthiscell = floor(tt)+0.5;
    gravelNoise = f2 = 1000;
    uniform float i, j;

    for (i = -1; i <= 1; i += 1) {
        float stestcell = sthiscell + i;

        for (j = -1; j <= 1; j += 1) {
            float ttestcell = tthiscell + j;
            float spos = stestcell + jitter * (cellnoise(stestcell, ttestcell) - 0.5);
            float tpos = ttestcell + jitter * (cellnoise(stestcell+23, ttestcell-87) -
0.5);
            float soffset = spos - ss;
            float toffset = tpos - tt;
            float dist = soffset*soffset + toffset*toffset;

            if (dist < gravelNoise) {
                f2 = gravelNoise;
                spos2 = spos1;
                tpos2 = tpos1;
                gravelNoise = dist;
                spos1 = spos;
                tpos1 = tpos;
            }
            else if (dist < f2) {
                f2 = dist;
                spos2 = spos;
                tpos2 = tpos;
            }
        }
    }
}

```

```

        }

gravelNoise = sqrt(gravelNoise);  f2 = sqrt(f2);
}

surface mountain (
    float worldTop          = (50);
    float worldBottom         = (10);
    float beachEdge           = (worldBottom + 0.01);
    float gravelReach         = (worldBottom + 1.6);
    float cliffReach          = (worldTop - 0.001);

    color water              = (0.3,          0.6,   1.0      );
    color gravel              = (0.35,         0.4,   0.35    );
    color grass               = (0.0,          0.55,  0.0      );
    color river               = (0.2,          0.8,   1.0      );
    color cliff               = (0.96,         0.92,  0.88    );
    color rock                = (0.96,         0.96,  0.82    );
    color haze                = (0.9,          0.9,   0.9      );

    float waterOctaves        = (2.5);           // Number of noise layers to be
summed

    float waterLacunarity     = (3.25);          // Level of "gappiness" or inhomogeneity
    float waterGain            = (0.55);          // The level of effective
contribution by each layer

    float grassOctaves        = (5.0);           // Number of noise layers to be
summed

    float grassLacunarity     = (1.4);           // Level of "gappiness" or inhomogeneity
    float grassGain             = (0.8);           // The level of effective
contribution by each layer

    float bumpFrequency        = (40.0);          // The frequency of the bump mapped
irregularity

    float bumpHeight           = (5.8);           // The depth of the texture bumps
    float roughness             = (0.5);           // The roughness of the bump map
    color diffuseColor          = (0.6, 0.8, 0.9); // Diffuse color
    color specularColor         = (0.8, 0.8, 0.9); // Specular color

    float fogDistance           = (550);           // Fog effective distance
    color fogColor              = (0.9, 0.9, 0.9); // Fog color
)

{

// (point p; uniform float octaves, lacunarity, gain, H, sharpness, threshold)
float magnitude = 2.1*(RidgedMultifractal((P)*0.002, 8, 2.2, 0.94, 0.8, 6.4, 5)*120.00001);
float mountainAtt = distance(zcomp(P), 55)/1000.0;

// Set the height of the landscape
float height = magnitude * mountainAtt;

// Clamp the world bounds to create a water surface and a maximum cliff height
height=clamp(height, worldBottom, worldTop);

// Calculate and even out normals to avoid bugs
}

```

```

P += height * normalize(N);
N = calculatenormal(P);
normal Nf = faceforward (normalize(N), I);

// Create a copy of the point P, add bump map noise and calculate the new normal
point pTemp = P;
// Execute a bump map noise function to create a water texture
pTemp += noise(pTemp * bumpFrequency) * N * bumpHeight;
N = calculatenormal(pTemp);

// Normalize
normal Nn = normalize(N);
vector V = normalize(-I);

// Create diffuse and specular water colors
color waterTextureA = Cs*(diffuseColor * diffuse(Nf));
color waterTextureB = Cs*(specularColor * specular(Nn, V, roughness));
// Add them together
color waterTexture = waterTextureA + waterTextureB;

// Execute a turbulence noise algorithm to create water noise
// Nota Bene: This has been replaced by bump mapping
// float waterNoise = clamp((turbulence(P, waterOctaves, waterLacunarity, waterGain)), 0, 1);

// Voronoi parameters, gravelNoise is the desired output
float gravelNoise, spos1, tpos1, f2, spos2, tpos2;

// Execute a Worley Cell Noise algorithm to create gravel noise
voronoi_gravelNoise(u*10000.0, v*10000.0, gravelNoise, spos1, tpos1, f2, spos2, tpos2);

// Execute an fBm noise algorithm to create the grass noise
float grassNoise = clamp((fBm(P, grassOctaves, grassLacunarity, grassGain)), 0, 1);

// Mix the water texture with the gravel texture to create a beach edge
color outputColor = mix((water * waterTexture), (gravel * gravelNoise),
smoothstep(worldBottom, beachEdge, height));

// Mix the gravel texture with the cliff texture to create a cliff foot
outputColor = mix(outputColor, cliff, smoothstep(beachEdge, gravelReach, height));

// Mix the cliff texture with the grass texture to create a cliff edge
outputColor = mix(outputColor, (grass * grassNoise), step(cliffReach, height));

// Diffuse the product
outputColor = Cs*(outputColor * diffuse(Nf));

```

```
// Create fog
float d = 1 - exp((-length(I) + 100)/fogDistance);
outputColor = mix (outputColor, fogColor, d);

// Output
Ci = outputColor;
Oi = Os;

}
```

mountainsky.sl

```
#define snoise(p) (2 * (float noise(p)) - 1)

float fBm (point p; uniform float octaves, lacunarity, gain)
{
    uniform float amp = 1;
    varying point pp = p;
    varying float sum = 0;
    uniform float i;

    for (i = 0; i < octaves; i += 1) {
        sum += amp * snoise (pp);
        amp *= gain;
        pp *= lacunarity;
    }
    return sum;
}

surface mountainsky(
    color skyColor          = (0.54, 0.64, 0.74);
    color hazeColor     = (1, 1, 1);
    color cloudColor   = (0.9, 0.9, 0.9);
)
{
    // Create a base for the sky using the haze and sky colors
    float baseColor = (0.6 - v);
    color outputColor = mix(hazeColor, skyColor, baseColor);

    // Create cloud noise using an fBm algorithm, dependent on the y-coordinate of the scene
    float cloudFreq = ((1.0 - v) * 6.0);
    float cloudNoise = clamp(fBm(P * cloudFreq * 0.001, 5, 2.75, 0.6), 0, 1);

    // Mix the sky and the clouds to the cloud noise
    outputColor = mix(skyColor, cloudColor, cloudNoise);

    // Add a haze effect to fade the horizon
    float haze = pow(1.4-v, 3);
    outputColor = mix(outputColor, hazeColor, haze);

    // Output
    Ci = outputColor;
    Oi = Os;
}
```

Tjena vilken Fin Kod du har

Idén med denna uppgift var att skapa ett inlevelserikt och detaljerat bergslandskap i enlighet med en handfull krav, men i övrigt fritt för artistiska tolkningar. Bergslandskapet behövde ha ett varierat beteende beroende på faktorer lutning och höjd. Även en himmel med någon form av detalj som bakgrund krävdes. Jag valde att tolka Dovers vita klippor, ett landskap med ett väldigt karaktäristiskt utseende. Nedan följer en övergriplig sammanfattning av processen som formade min textur.



Dovers vita klippor.

Bergen

Först av allt måste en grund för landskapet skapas. I praktiken ska alla berg och ojämnheter beräknas fram innan vi kan börja forma och justera landskapet till det önskvärda resultatet. Bergen, som egentligen innebär en samling slumpmässiga höjningar och sänkningar på en yta, beräknas genom en algoritm. Denna algoritm kallas Ridged Multifractal och tar bland annat parametrar likt de vi tidigare använt i maskningsalgoritmer som fBm, vilka hjälper oss specificera ojämnheternas egenskaper. Dovers klippor är väldigt branta, näst intill rätvinkliga, och jag skapade därför väldigt höga och skarpt lutande berg. För att inte bergen ska befina sig för nära den virtuella kameran skalades bergens höjd mot positionen P i världen.

Dovers klippor är givetvis inga berg, utan snarare ett kpat landskap. För att skapa en platt yta i nivå med landet och en annan platt yta i jämn höjd med vattenståndet använde jag en clamp som kapade landskapets högsta och lägsta höjdsvälvningar lämpligt för att simulera ett hav och landet bortom klipporna. Med hjälp av bump mapping gav jag den längsta, platta nivån i landskapet en ojämnn, knotrig yta för att simulera lekfulla små vågor.

Det finns totalt fyra olika fysiska ytor med olika utseende. Vatten, grusstrand, klipporna samt mark/gräs. Dessa tillämpades ganska enkelt på landskapet beroende på dess höjd. I botten, alltså på havet, tillämpades vattenfärgen i samband med dess bump map. Straxt ovanför simulerade jag en grusig/stenig sand med en brusig, grå yta. Bruset genererade jag med hjälp av en Worley Cell Noise-algoritm med en mycket hög frekvens (då grus är oskiljaktigt från kamerans stora avstånd) vilket jag blandade med grusfärgen.

Klipporna har en ljus, nästan vit/ägg färg. I och med att den första multifraktalen skapade en övertygande bergsyta gjorde scenens ljussättning att de slumpmässiga utspridda skuggorna betonade klippornas ojämnhet realistiskt. Därför har ingen vidare åtgärd tagits för klippornas branter.

På marknivå har en smidig och enkel fBm-algoritm använts för att skapa en varierad, gräsliknande yta. Ett av kraven för uppgiften var att använda en rad olika brusalgoritmer, bland annat fBm. Hade jag gjort om uppgiften utan dessa krav hade jag med största sannolikhet använt ännu en bump map, då vattenytan i min åsikt är mycket mer tilltalande och övertygande än en simpel fBm. Resultatet känns trots det helt godtagbart.

Med landskapet färdigt tillämpas diffus ljussättning och till sist dimma. Grunden till dim-algoritmen är tagen från Pixars egna exempelkod. En mycket ljus grå färg blandas in i landskapet med en täthet som beror på scenens djup (z-koordinat). Detta gör att dimman inte förstör sikten framför kameran samtidigt som den täcker scenens horisont så som en lång utsikt faktiskt gör.

Himlen

Bakom landskapet finns en yta som jag modifierar till att efterlikna en himmel. Som bas tilldelades vi exempelkod som skapade en blå himmel som tonade mot vitt längre ned. Jag justerade denna kod för att bättre passa mitt landskap och mina färgpreferenser samt tonade den vita övergången för att låta den smälta ihop med bergslandskapets tät horisontdimma. Jag ska hädanefter komma ihåg att RGB-blandningen (0.54, 0.64, 0.74) lämpar sig perfekt som en färg för en klar himmel, särskilt när den tonar mot vitt.

Därefter skapade jag ett fBm-brus att använda för molnen. Just moln anser jag var extra svårt att simulera realistiskt, men den väldigt lågfrekventa fBm-algoritmen jag skapade i kombination med en multiplicerad hänsyn för scenens höjd (vilket i bakgrunden visuellt tolkas som scenens djup, då längre ned på bilden närmar sig horisonten) skapade en förvånansvärt trovärdig effekt. Molnens utseende påminner väldigt mycket om naturligt förekommande cirrostratosmoln, en väldigt tunn molntyp som uppstår på mycket höga höjder och oftast dras ut till långa slingor på grund av altitudens kraftiga vindar. Illusionen av höjd förstärks dessutom utmärkt av deras skalning mot det upplevda skendjupet.

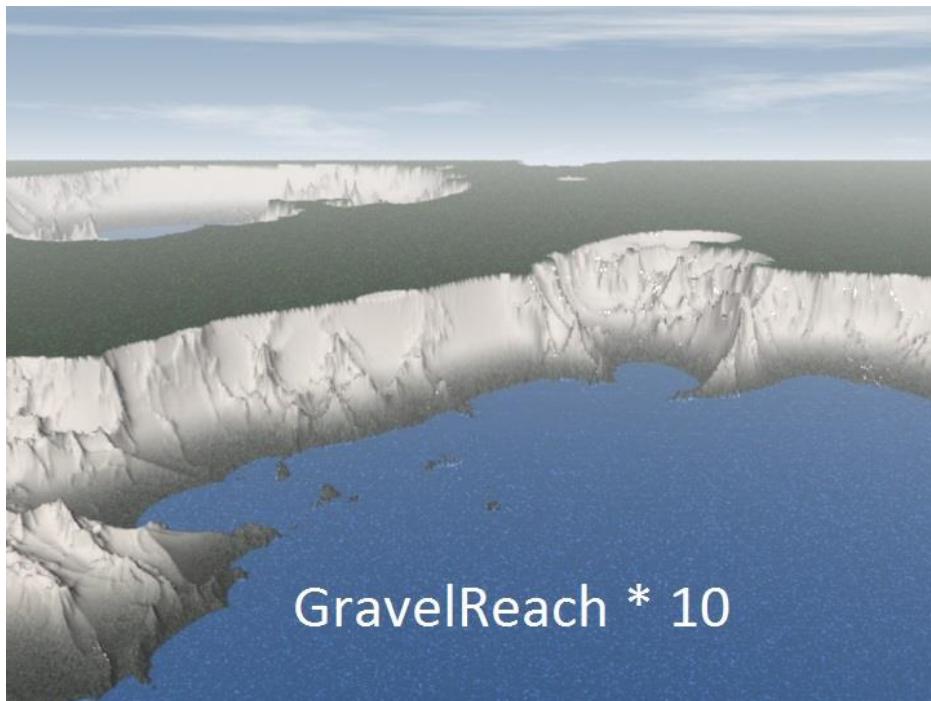
Låt Oss Leka med Lite Värden

Följ länken nedan för att se en pedagogisk .gif-animation över de olika versionerna av den parametermanipulerade texturen:

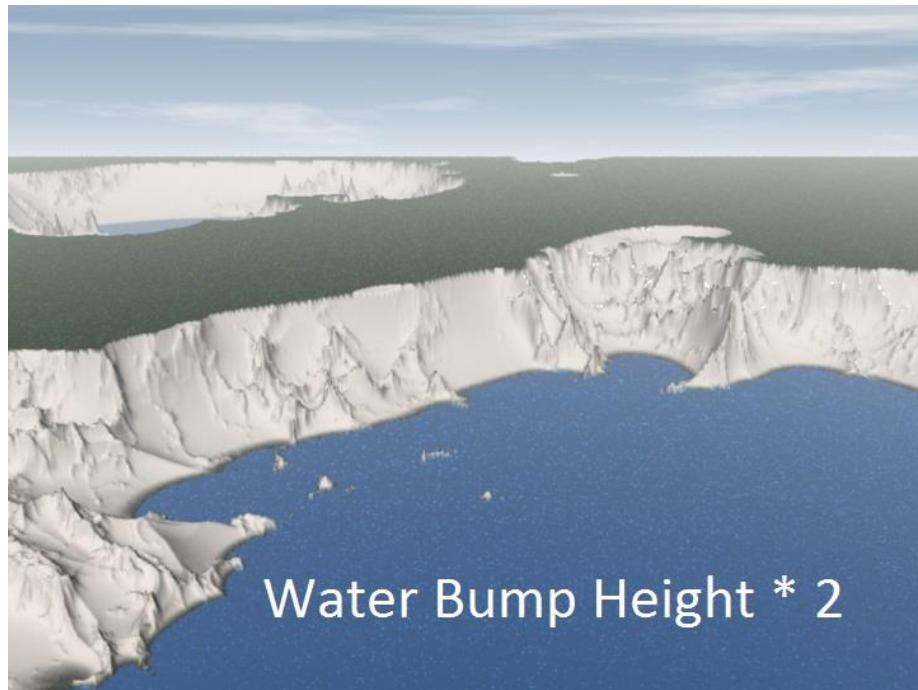
<http://imgur.com/FnECvJD>



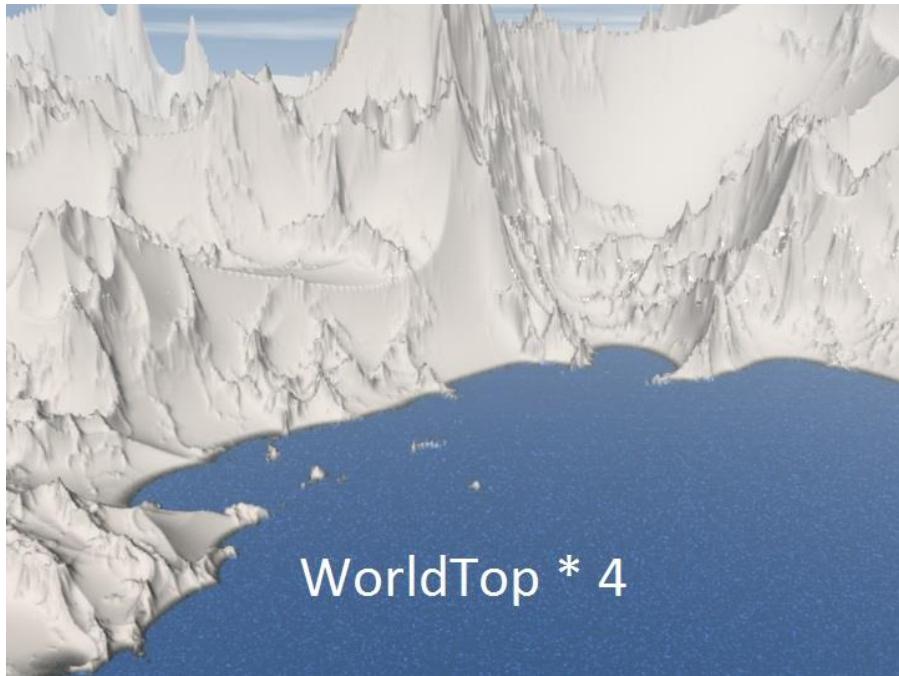
Detta är en oförändrad version av bergslandskapet. Bergen är förhållandevis låga, vattnet högfrekvent brusigt, himlen naturligt blå och gruset är knappt synligt i och med att det ligger så nära vattnet.



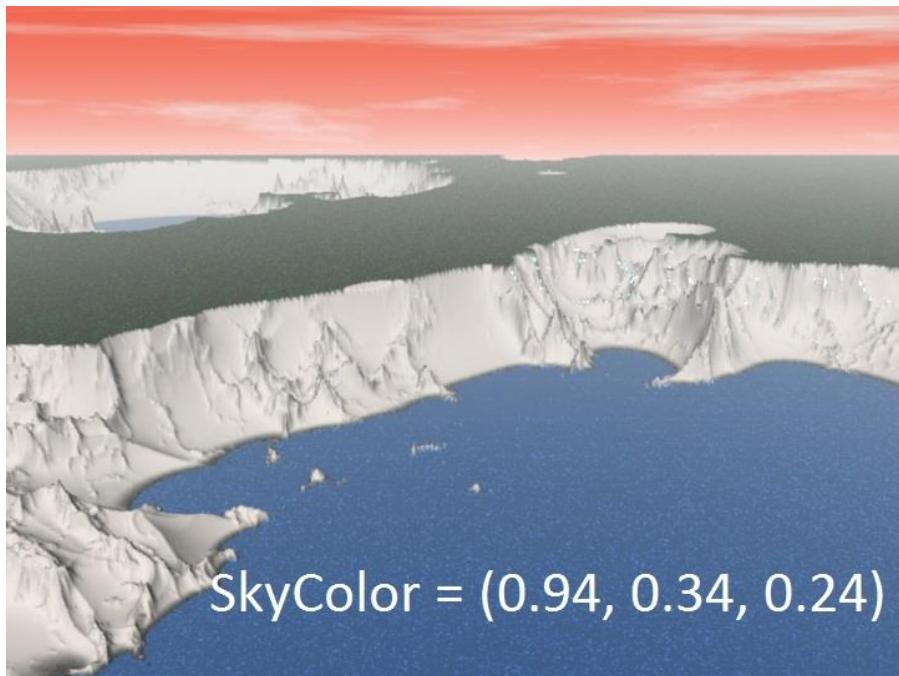
Här ökar har parametern gravelReach tiofaldigt ökats. Gruset, som utgör gränsen mellan klipptyta och vattenyta, tar upp betydligt mer plats. Det syns även mycket tydligare att grusstranden använder sig av en smoothstep för att ge en len övergång utan skarpa kanter.



Här har parametern som kontrollerar vågornas omfattning ändrats. Vågorna använder en bump map som bland annat tar en parameter för ojämnheternas individuella maxhöjd. Den höga frekvensen orsakar i originalbildens samplinginställningar och upplösning en bugg som lyckligtvis ger intrycket av vågor som rör sig mot den bortersta stranden. Genom att öka parametern BumpHeight till det dubbla försvinner buggen och vattenytan ser marginellt mindre glimmande ut.



Precis som vi ändrade sträckningen för grusytan har vi här fyrdubblat den maximalt tillåtna bergshöjden. Skillnanden är att parametern används med en clamp som gränsar bergens högst och lägsta tillåtna altituder, till skillnad från grusräckvidden som påverkar hur stor yta som behandlas med grustexturen. Bilden illustrerar klippornas "sanna" utseende som de hade sett ut om de inte kapats. Inget gräs syns eftersom att topparna är spetsiga snarare än plana ytor.



I denna bild har inget ändrats i landskapet, utan istället i filen MountainSky.sl som styr utseendet på bakgrunden. I en mycket enkel ordning har vi ändrat den primära bakgrundsfärgen (parametern SkyColor) från himmelsblå till en persika-/laxfärg. Notera att himlen trots sin nya färg fortfarande tonar mot vitt i horisonten. Det innebär att parametern enkelt kan användas för att manipulera himlens färg utan att störa himlens förhållandevis realistiska utformning.

Procedurella Shaders kontra Färdiga Texturer

En statisk, handmålad bild eller textur kan skapas med en näst intill oändlig hänsyn för detalj och specifikation om resurser i form av tid och utrustning försummas. Men i projekt av alla typer av skalor är resurser väldigt begränsade, och att skapa en bild får inte ta hur lång tid som helst eller kosta hur mycket som helst. Skulle dessutom arbetets specifikationer plötsligt ändras riskerar hela arbetsprestationen att bli värdelös.

En procedurellt skapad textur kan mycket snabbt ge konkret resultat och mycket effektivt och dynamiskt ändras efter krav och feedback. Samma mängd tid och energi fördelad mellan en programmerare och en grafiker kan mycket väl ge samma resultat eller till och med resultat i den ena eller andres fördel, men programmeraren har betydligt större frihet efter planeringssteget att ändra spår och stil, medan grafikern sitter fast vid sin planering och enbart har rum för slutgiltiga detaljer att ändra sitt arbete.

Anställda hos Pixar diskuterar i Renderman-boken hur en kombination av handmålat detaljarbete applicerat mot procedurella texturer i slutändan ofta var den enda verkligt effektiva processen för att skapa olika scener. I fallet med ett bergslandskap ser jag inte procedurella texturer som definitivt överlägsen handmålade verk, men jag ser en enorm fördel i att så pass fritt kunna manipulera min scen och samtidigt producera stora mängder material ur den. Genom att förflytta den virtuella kameran kan jag välja och plocka ur ett rent teoretiskt oändligt antal varianter av klipplandskap med samma specifikation. Jag skulle även utifrån min nuvarande kunna skapa ett fullfjädrat bergslandslandskap enbart genom att ändra några enstaka kodrader. Däremot saknas jag de detaljer och möjligheter till justeringar som hade fulländat landskapets utseende.

Uppgift 4



Kod

Fragment/pixel program

```
uniform vec4 uTeapotColor;
uniform vec4 uLight1Color;
uniform vec4 uLight2Color;

uniform vec3 uLight1Position;
uniform vec3 uLight2Position;

uniform float uDiffuseIntensity;
uniform float uSpecularIntensity;
uniform float uRimIntensity;
uniform float uTexture2DIntensity;
uniform float uTexture3DIntensity;

uniform vec4 uDiffuseColor;
uniform vec4 uSpecularColor;
uniform vec4 uRimColor;
uniform sampler2D baseMap;
uniform sampler3D baseMap3D;

varying vec3 vNormal1;
varying vec3 vNormal2;
varying vec3 vLightDir1;
varying vec3 vLightDir2;
varying vec3 vEyeVec;
varying vec2 vTexCoord;
varying vec3 P;

void main(void)
{
    // Normalize vectors
    vec3 E    = normalize(vEyeVec);

    vec3 N1   = normalize(vNormal1);
    vec3 N2   = normalize(vNormal2);

    vec3 L1   = normalize(vLightDir1);
    vec3 L2   = normalize(vLightDir2);

    vec3 R1 = reflect(L1, N1);
    vec3 R2 = reflect(L2, N2);

    // Create a 2D texture and mix it to the output
    vec3 texture2 = texture2D(baseMap, (vTexCoord * 0.5)).xyz;
    texture2 = mix(texture2, vec4(1.0, 1.0, 1.0, 1.0), uTexture2DIntensity);
```

```

// Create a 3D texture to control the intensity of the specular light
float texture3 = texture3D(baseMap3D, P * 0.3).x; fextexture3mixm
texture3 = mix(texture3, vec4(1.0, 1.0, 1.0, 1.0), uTexture3DIntensity);

// Lambert Diffuse Lighting
float light1Distance = 1.0 - (abs(uLight1Position.x + uLight1Position.y +
uLight1Position.z) / 300);
float light2Distance = 1.0 - (abs(uLight2Position.x + uLight2Position.y +
uLight2Position.z) / 300);

float diffuseIntensity1 = light1Distance * uDiffuseIntensity;
float diffuseIntensity2 = light2Distance * uDiffuseIntensity;

float diffuse1 = clamp(dot(N1, -L1), 0.0, 1.0) * diffuseIntensity1;
float diffuse2 = clamp(dot(N2, -L2), 0.0, 1.0) * diffuseIntensity2;

// Phong Specular Lighting
float specular1 = pow(clamp(dot(R1, E), 0.0, 1.0), 80.0) * uSpecularIntensity;
float specular2 = pow(clamp(dot(R2, E), 0.0, 1.0), 80.0) * uSpecularIntensity;

specular1 *= clamp(texture3, 0.0, 1.0);
specular2 *= clamp(texture3, 0.0, 1.0);

// Rim Lighting
float rimLight = smoothstep(0.5, 0.05, dot((N1 + N2), E)) * uRimIntensity * uRimColor;
specular1 += rimLight;
specular2 += rimLight;

// Diffuse Attenuation
specular1 *= diffuse1;
specular2 *= diffuse2;

// Assemble components to a common output vector
vec4 light1Output = uLight1Color * diffuse1 + specular1;
vec4 light2Output = uLight2Color * diffuse2 + specular2;
vec4 outputColor = light1Output + light2Output;

// Output
gl_FragColor = outputColor * vec4(texture2, 1.0);

}

```

Vertex program

```
uniform vec3 uLight1Position;
uniform vec3 uLight2Position;

varying vec3 vNormal1;
varying vec3 vNormal2;
varying vec3 vLightDir1;
varying vec3 vLightDir2;
varying vec3 vEyeVec;
varying vec2 vTexCoord;
varying vec3 P;

void main( void )
{

    // In OpenGL we can use ftransform function or matrices explicitly
    vec4 vertex = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position = vertex;

    // First light source
    vLightDir1 = -vec3(gl_ModelViewMatrix * vec4(uLight1Position, 0.0));
    vNormal1 = gl_NormalMatrix * gl_Normal;

    // Second light source
    vLightDir2 = -vec3(gl_ModelViewMatrix * vec4(uLight2Position, 0.0));
    vNormal2 = gl_NormalMatrix * gl_Normal;

    // Eye vector
    vEyeVec = -vec3(gl_ModelViewMatrix * gl_Vertex);
    vTexCoord = vec2(gl_MultiTexCoord0);
    P = gl_Vertex.xyz;

}
```

Låt Oss Leka med Lite Värden

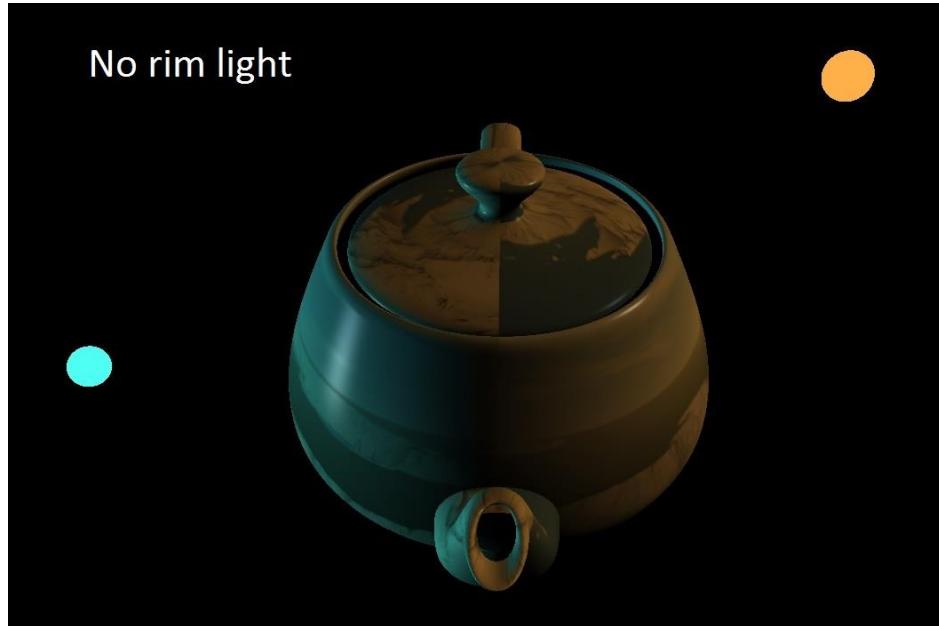
Följ länken nedan för att se en pedagogisk .gif-animation över de olika versionerna av den parametermanipulerade scenen:

<http://imgur.com/m3wY0pv>



Detta är en oförändrad version av scenen. Tre objekt finns närvarande, en vit tekanna, en ljuscyan ljuskälla (t.v. i bild) och en aprikosfärgad ljuskälla (t.h. i bild). Tekannan ser ut att anta ljuskällornas färg eftersom att de kastar ljus med samma färg som källorna. Under tekannans "färg" syns en påklistrad tvådimensionell textur (föreställande bergslandskapet från Uppgift 3). Det finns även en 3D-texture som ger tekannan en aning mer ojämn yta, vilket ger ett mer realistiskt intryck än en perfekt jämn, blank yta. Tekannan belyses av ljuskällorna diffust, spekulärt och med kantbelysning.

No rim light



I denna bild har kantljusberäkningsalgoritmen helt utelämnats. Det övergripande intrycket av realism är inte väldigt utmanat av dess frånvaro, i och med att ljuset påverkar en så pass liten yta. Däremot med kantljuset tillämpat är skillnaden ganska påtaglig. Se graphics interchange format-bilden länkad i kapitlets begynnelse för en tydligare kontrast.

Changed second light source color



Här har den högra ljuskällans färg ändrats från aprikos till cerise, vilket får tekannan att anta samma färg på de ytor som träffas av ljuset.

Moved second light source position



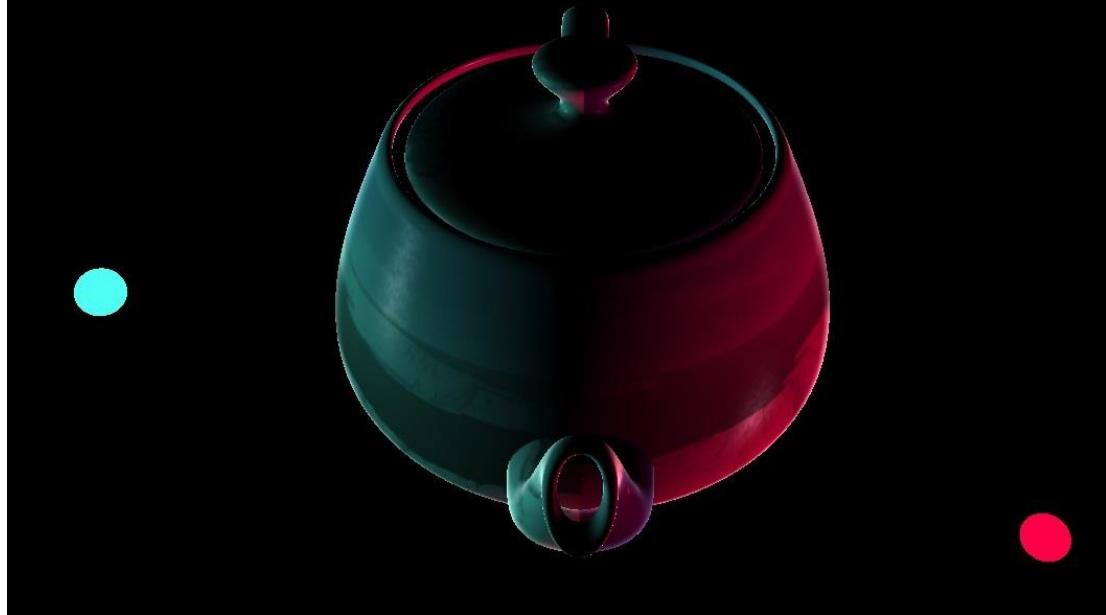
Här har den högra ljuskällans position förflyttats, i huvudsak med Y-axeln. Alla objekt på scenen har en tredimensionell position i världsrymden. Z-axeln i denna scen syftar till djupet i tekannans sidor. Den cyanfärgade ljuskällan står i en negativ koordinat på Z-axeln, alltså "bakom" tekannan, medan den cerisefärgade sfären står "framför" den, i en positiv Z-kordinat. Y-axeln syftar till positionen i höjdled, det vill säga närmare tekannans botten eller lock. Den förflyttade ljuskällan har fått ett sänkt Y-värde, alltså sänkts ned närmare scenens botten.

No 2D texture



I denna bild har 2D-texturen (bergslandskapet) uteslutits från shadern. Tekannan har alltså en helt neutral yta bortsett från 3D-texturen. Endast ljuskällorna spelar in i tekannans estetiska utseende.

Maximum 3D texture intensity



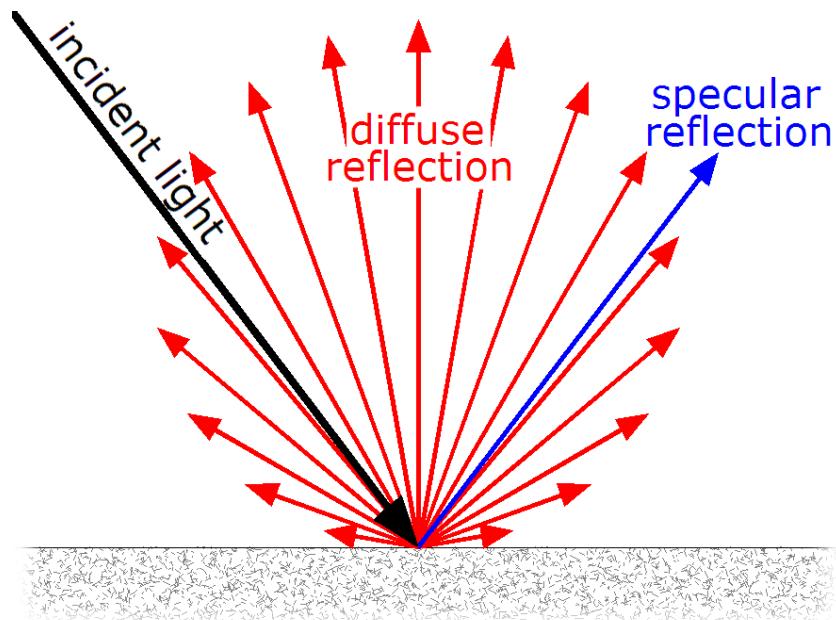
I den sista bilden har den tidigare nämnda 3D-texturen tilldelats en högre intensitet, alltså väger in med större magnitud i ytans utseende. Utan 3D-texturen skulle tekannan se helt blank och jämn ut till ytan, men med 3D-texturen tillämpad ser tekannan skrovlig och mer realistiskt ojämnn ut. För att förtydliga detta fenomen har parametern för texturens influens på tekannan maximerats. En av de mest synliga effekterna är att det spekulära ljuset stryps av ytans ojämnheter och mindre spekulärt ljus kastas. Detta av precis samma anledning som en blank kaffekopp blänker tydligt i medljus medan en skrovlig plastmugg knappt reflekterar något direkt (spekulärt) ljus alls.

Styrkor/Svagheter med Diffus-, Spekulär- och Kantbelysning

Ljus och dess beteende är väldigt komplext i det avseende att många faktorer väger in i ljusets egenskaper. Det är inte så enkelt som att påstå att fotoner beter sig som en kastad tennisboll som studsar omkring mot ytor tills energin tar slut. Det finns mycket att elaborera från en så pass oraffinerad perception. En av många viktiga kategoriska egenskaper hos ljus innefattar reflektion. Tack vare reflektioner, det vill säga hur ljus studsar, kan vi visuellt uppfatta vår omgivning, liv kan existera och lampskärmar fyller ett syfte.

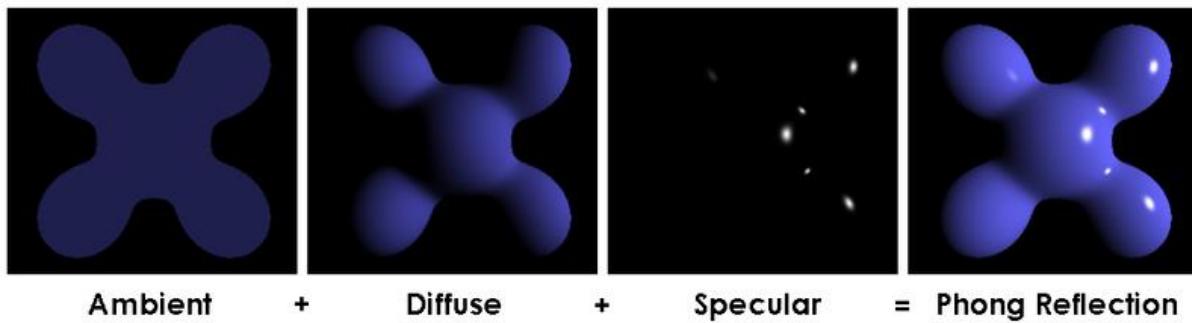
Det finns tre enkla modeller att beskriva några typer av naturligt förekommande reflektioner; diffusa, spekulära och kantreflektioner. Likt hur påståendet att ambient ljus existerar vart än ljus existerar stämmer kan påstås att diffust ljus uppstår vart än en definierad ljuskälla är närvarande. Det diffusa ljuset innefattar i teorin allt ljus som reflekteras av en yta i splittrade riktningar. Alla matta eller ojämna ytor reflekterar ljus diffust, och det är tack vare detta som vi faktiskt kan se vår omgivning, då detta reflekterade ljus är vad som i huvudsak når våra näthinnor. Det kan påstås att en helt "matt" yta enbart reflekterar diffust ljus.

En annan typ av reflektion är spekulärt ljus. Spekulära reflektioner uppstår, till motsats från diffusa, i huvudsak mot material med någon slags blankhet. Det spekulära ljuset står närmare beskrivningen av tennisbollen, då det reflekteras bort från en yta i samma vinkel om infallsvinkeln. Diffust ljus, å andra sidan, utgör de reflektioner som kaotiskt sprider sig i alla andra riktningar. Se figuren nedan.



Kantljus, eller *rim light*, syftar till de reflektioner som uppstår runt kanterna hos ett objekt som befinner sig framför en ljuskälla. Kantljus ger objekt ett väldigt levande utseende och skapar en väldigt tydlig kontrast mot bakgrunden i och med att en gräns av reflekterat ljus avgränsar objekts yta.

Samtliga av dessa tre ljuseffekter är ganska primitiva i förhållande till den enorma komplexiteten som ljusets egenskaper innebär och de många olika algoritmer som med betydligt mer komplexa algoritmer försöker simulera det. Mina personliga iakttagelser är att diffust, spekulärt och kantat ljus är så pass grundläggande i sin algoritmer och principer att de kräver minimal ansträngning såväl för programmeraren att utföra som för en grafikprocessor att beräkna. Med detta i beaktning kan det understrykas att de tre ljuseffekternas tillsammans tar hänsyn till bland annat ett objekts form, vinklar, material, matthet, blankhet och en rad olika positionsförhållanden i relation till olika ljuskällor. Byter vi ut kantbelysningen mot vanligt ambient ljus får vi en kombination av effekter som tillsammans skapar den så kallade Phongreflektionen, en välkänd och populär grafisk komplikation av ljuseffekter. Det bevisar att även simpla algoritmer kan åstadkomma imponerande effekter när de kombineras.



Detta innebär att vi med förhållandevis lite arbete kan skapa en grundläggande belysning som i många sammanhang är mer än tillräcklig för att återskapa en trovärdig och effektfull ljussättning. När vår scen inte fordrar en väldigt sofistikerad eller avancerad grafisk trovärdighet eller resurser i form av arbetskraft, tid, pengar eller beräkningsprestanda är begränsade kan enklare algoritmer likt de ovan nämnda vara betydligt mer önskvärda att använda. Det är i mina ögon de alla främsta exemplen på egenskaper som i vissa sammanhang gör mindre komplexa algoritmer mer önskvärda än avancerade ljusapproximationer.

Problem och Lösningar för att Implementera flera Ljuskällor

Att en scen har fler än en ljuskälla är en fullständigt logisk specifikation. Om vi så mycket som kikar oss omkring hittar vi garanterat en rad olika lysande element. TV-apparater, mobiler, lampor, ljusdioder, varningslampor, vår egen sol, gatlyktor och allt i världen som avger energi i form av ljus. Kort sagt kan påstås att många olika tänkbara scenarion innefattar fler än en enda ljuskälla närvarande.

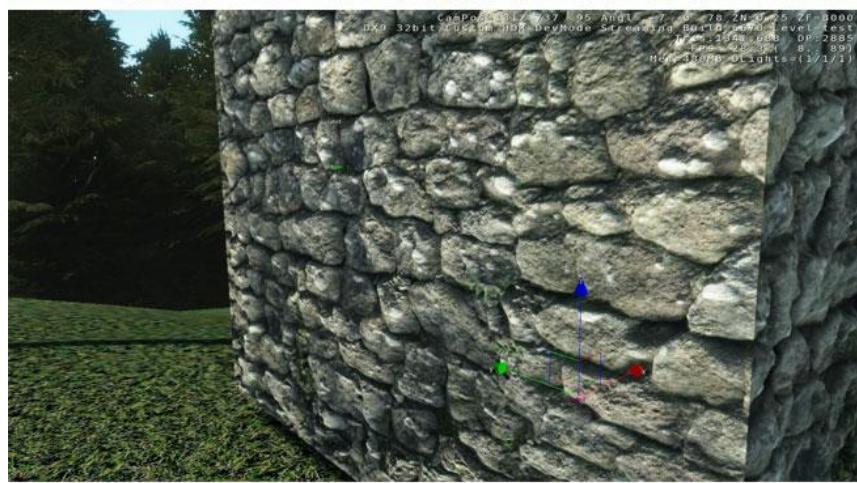
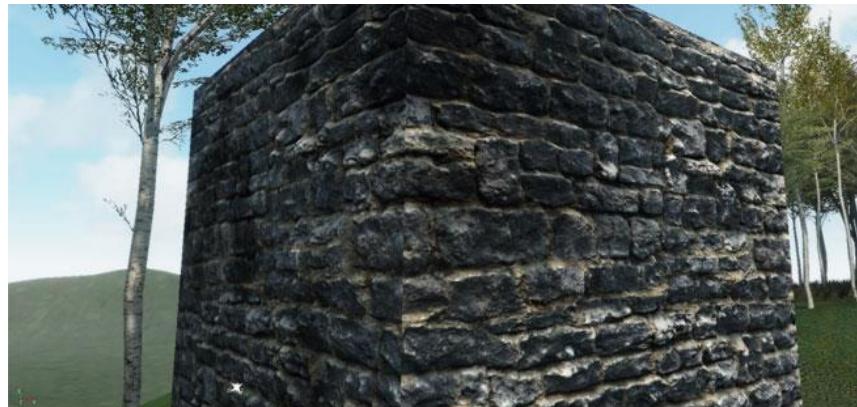
Men vad innebär detta i praktiken för oss programmerare? Det innebär att vi måste fatta rationella beslut om hur ljus bör bete sig i vår nuvarande scen, men även vilken effekt det är vi försöker åstadkomma. Kanske har vi valt att skapa en scen där några personer sitter samlade framför en TV i ett kolsvart rum. Målet är att skapa en verklighetstrogen uppställning som inger en känsla av att personerna är fullständigt fokuserade på TV-apparaten. Vi lägger enorm vördnad i att skapa en oerhört detaljerad och sofistikerad ljussättning med TV-apparaten som enda ljuskälla. Men när scenen är färdigställd ändras kraven, och vi behöver plötsligt även inkludera fler ljuskällor. En av personerna på bilden håller i en påslag mobiltelefon, och utanför fönstret skiner en gatlampa in i rummet.

Vi har alltså en scen där en ljuskälla plötsligt måste multipliceras, men med nya egenskaper. Det finns flera sätt att lösa detta problem som är mer eller mindre tids- och resurseffektiva. Givetvis kan programmeraren välja att procedurellt arbeta vidare och lägga till samtliga ljuskällor ett i taget med sin egna unika egenskaper. Det är en enkel lösning, men allt annat än effektiv eller strukturerad. Vi får en rörig och svårtydd hög med kod och en väldigt svårkontrollerad scen. En aningen bättre lösning vore att skapa en funktion för infogning av ljuskällor. På så vis kan vi mer dynamiskt lägga till och ta bort ljus från scenen.

Problemet med en ljusrenderingsfunktion är att samtliga ljuskällor i största allmänhet måste följa liknande algoritmer då de måste ta emot samma parametrar. Vi förlorar på så vis flexibiliteten att styra ljuskällornas egenskaper separat från varandra. Ponera det tidigare scenariot där vi ville lägga stort fokus på en scen med realistiskt och avancerat ljus från TV-apparaten, men samtidigt inkludera ett antal mindre ljuskällor som kastar mindre ljus omkring scenen. Det är högst sannolikt att de mindre signifika ljuskällorna inte kräver mer än simpla approximationer och enkla effektrealiseringar. Det är i sådant fall onödigt att tillämpa TV-apparatljusets sofistikerade algoritmer flera gånger och slentrianmässigt förkasta beräkningsresurser som i praktiken har en näst intill obetydlig effekt.

Ett förslag på en lösning kan vara att bygga en hierarkisk funktionsmodell som låter programmeraren steg för steg iterera genom olika komplexitetsnivåer av möjliga tillämpningar. Beroende på situationen och de önskade egenskaperna hos ljuskällan som skall infogas kan programmeraren välja vilka algoritmer som bör användas och därigenom endast behandla de parametrar som är relevanta för just den ljuskällan. För små, mindre betydliga ljuskällor kan simplare approximationer nyttjas och i mer detaljkravande och omfattande ljussystem kan mer komplexa algoritmer användas. Det vore förhållandeviseffektivt i sin användning men omfattande i sin implementation. Detta system kräver uppenbarligen en omfattande mängd förarbete, men kan eventuellt användas effektivt över en längre tid och i större projekt när hänsyn till dynamiska specifikationer på ljuskällor kan bli aktuellt.

Uppgift 5



Låt Oss Leka med Lite Mer Värden

Nämen, vad kul! Det finns ju en massa fräna parametrar man kan spaka och vrida på. Det tycker jag att vi provar för att se vad som händer. Jag slänger upp några bilder så kan vi diskutera vad vi ser.

Vi börjar med en enkel:



I oförändrat tillstånd ser vår tekanna ut såhär. Den kommer agera referens för övriga bilder. En mjuk skugga ligger motsatt den sfäriska ljuskällan. Texturen är förhållandevis lågfrekvent och skårorna i ytan ser ut att ha ett förhållandevis tydligt definierat djup.



I denna bild har vi dragit upp parametern som styr ljusets relief. Resultatet av detta blir att skuggan som omlindar tekannans vänstra sida (vid pipen) får en större täckning och en skarpare kant. Samma effekt sker på pipen, men från ögats vinkel i bilden syns det mycket svagt.



I denna bild har parametern NormalDiffuseIntensity sänkts, vilket innebär att texturen som används för att skapa en normalmap har en näst intill obefintlig inverkan på tekannan den tillämpades på. Resultatet blir att den ojämna yttexturen försvisser och lämnar den rutmönstrade tekannan fullständigt blank och jämn till ytan.



Här har vi försiktigt ökat parametern ParallaxBias, vilken manipulerar vår parallaxmappning. Parallaxmappens uppgift är att skapa illusionen av djup i texturen. Genom att öka och minska hur intensivt mappen påverkar texturen förskjuts texturens koordinater mot dess tangentvektorer. Texturen får då tekannans mönster att förefalla utstickande eller insunken.



Till sist har vi sänkt tekannans relief. Reliefen använder en map som motsvarar "tegelstensmönstret" på tekannan, men endast dess konturer. Genom att invertera och infoga mappen betonas skårorna mellan "stenarna" i texturen, vilket ger ett betydligt förstärkt intryck av djup. Jämfört med originalet ser tekannans textur ut att här vara "grundare" och mindre definierad.

Kubmap-reflektioner kontra Spekulär belysning

En fördel med spekulär belysning är att den är väldigt förutsägbar. Algoritmen gör precis vad som förväntas och sällan med överraskande resultat. Det är dock inte en billig process, utan kan kräva ganska stor beräkningskraft, något som är oönskat när resurser är begränsade eller prestanda måste optimeras. Det kanske absolut största problemet med spekulär belysning kan dock vara att den är så primitiv i sin princip. Spekulär belysning reflekterar enbart ljuset från en ljuskälla, och fungerar därför dåligt om flera ljuskällor används i närheten av varandra. De måste dessutom skapas och hanteras separat, till skillnad från reflektionsmappar som kan hantera mer ljus och flera ljuskällor i samma funktion.



Kubmap-reflektioner är många sammanhang lättillgängliga då de i allmänhet är smidiga att använda samtidigt som flera stora plattformar stödjer det rent tekniskt. Ett antal motorer tillåter för kubmappar att genereras dynamiskt i världsrymden vilket är till enomrätta förtunat för beräkning av ljus och ger dessutom möjlighet till betydligt mer realistiska reflektioner.

I stora, energiska scener med många objekt och/eller ljuskällor som ska renderas i realtid kan kubmappreflektioner vara en god idé. I mindre scener med något enstaka objekt kan dock spekulär belysning vara fördelaktigt för att producera ett precist resultat som enklare kan justeras eller användas vidare med andra belysningar. Det spekulära ljuset är enkelt att använda, förståeligt och förutsägbart i sin princip.

Parallaxmappning och Varför den Äger

Parallaxmappning är en metod som kan användas för att skapa en mer effektiv illusion av djup hos ett eller flera objekt i en scen. Genom att förskjuta ett objekts texturkoordinater med hjälp av dess tangentrymd ser texturen ut att ”dras ut” från sina kanter vilket skapar en 3D-effekt.



I och med att objektets yta fortfarande utgörs av en tvådimensionell textur tillför vi inget arbete i den mer beräkningskrävande 3D-rymden. Vi jobbar enbart mot den redan existerande ytan utan att efterfråga något extra arbete från vertexprogram i den grafiska pipelinen, trots att resultatet lätt kan tolkas som ett objekt med ökat Z-djup. Effekten är inkapabel att mäta sig mot äkta 3D när kravet på djup når ansenliga nivåer. Parallaxmappning kan med andra ord inte användas för att ersätta exempelvis hela byggnader eller träd i en scen.

Ett alternativ till parallaxmappning är tessellation, alltså processen att fylla ut lågfrekventa ytor med ytterligare polygoner. Om det önskas kan en yta som saknar djup tesselleras för den konkreta effekten att djup tilldelas geometriskt. Stenar, pinnar och andra objekt i en scen matas på med trianglar, något som ger ett strålande resultat. Precis som att höja en texturs upplösning blir saker och ting mer detaljerat. Eventuella kanter, aliasingeffekter från platta ytor och skarpa ändringar elimineras med ”rå styrka”. Dock så måste givetvis denna styrka ha ett ursprung, och saknas resurser för att utföra detta (exempelvis om en scen är renderad i realtid på en dator med för svag hårdvara) kan det vara ett oönskat alternativ.

I sådant fall kan parallaxmappning vara ett utmärkt alternativ att tillämpa, då effekt som i flera avseenden kan anses fullt duglig (exempelvis mot stora ytor med låga krav på djup, som kullersten eller ojämna väggar) utan att tiqdubbla en redan ansträngd triangelsumma. Att göra en 3D-modell av separata detaljer på en lång kullerstensväg som endast saknar några centimeter av djup är överflödigt i förhållande till parallaxmappens styrka.

Berättelsen om den Alternativa Normalmapstexturen

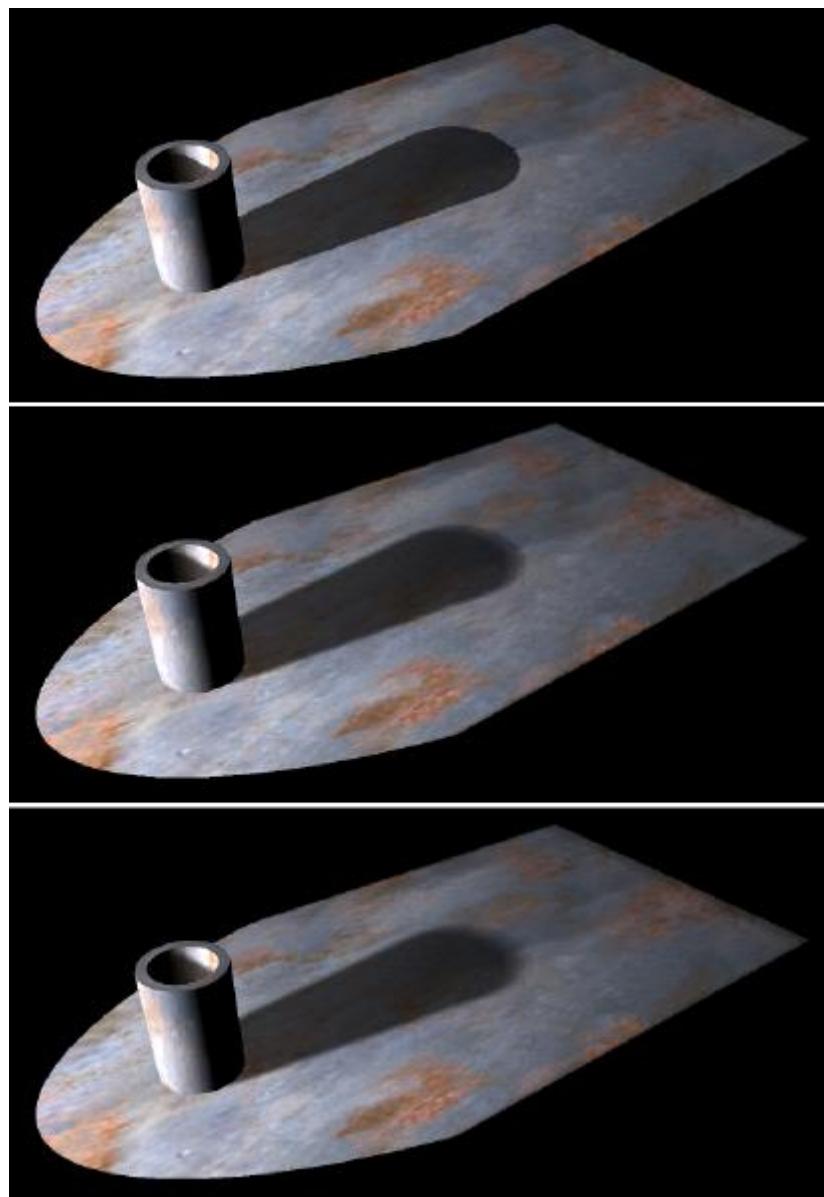


Genom att byta ut normalmapstexturen mot en alternativ variant med tydligt definierade kanter runt tegelstenarna blir resultatet drastiskt annorlunda, som bilden ovan illustrerar. Bägge maps har för- och nackdelar gällande realismen i deras utseende.

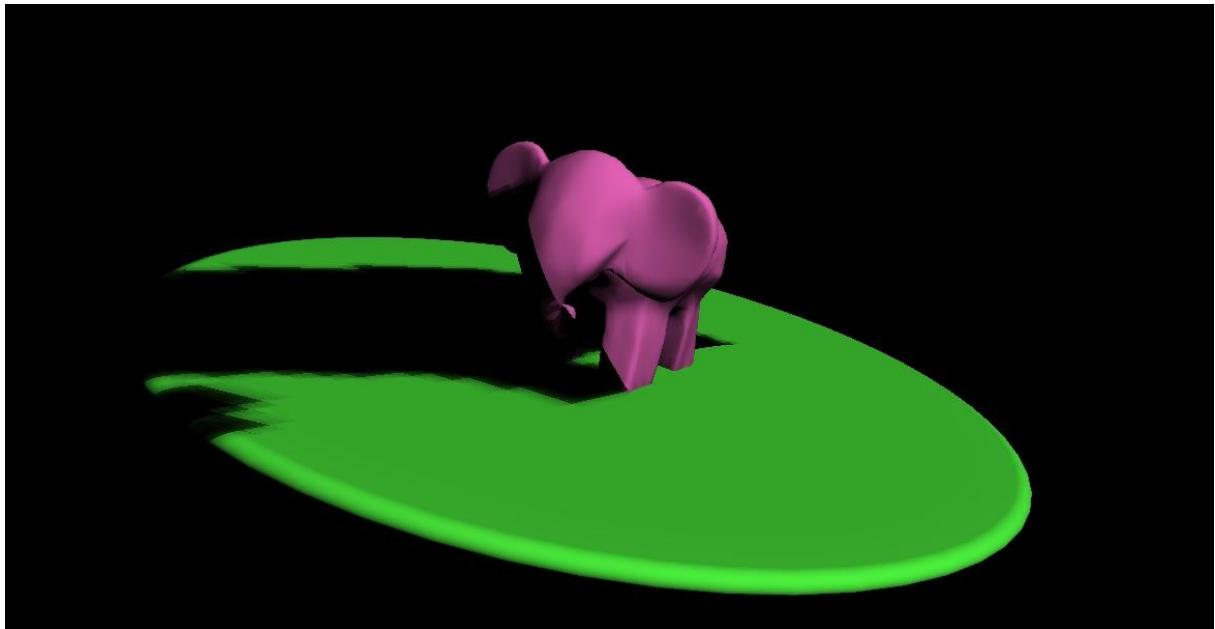
Den alternativa mappen (längst ned i bilden) ersätter på sätt och vis parallax-effekten och simulerar djup ganska effektivt, särskilt på ett avstånd. Däremot förvränger den ljuset runt kanterna som ska reflektera ljuset in mot tekannans skåror på ett sätt som ifrågasätter det övergripande utseendet och ljussättningen samtidigt. Samma sak sker givetvis även med den ordinarie normalmapen, men då uppstår förvrängningen betydligt mer diskret.

Däremot ser den ordinarie normalmapstexturen betydligt plattare och tråkigare ut än den alternativa mapen när parallax-effekten inte används. Med parallaxeffekten är resultatet i mina ögon mer realistiskt, men inte riktigt lika tilltalande. Om realism är målet hade jag personligen valt den ordinarie mapen, men om min agenda var estetisk hade jag valt den alternativa.

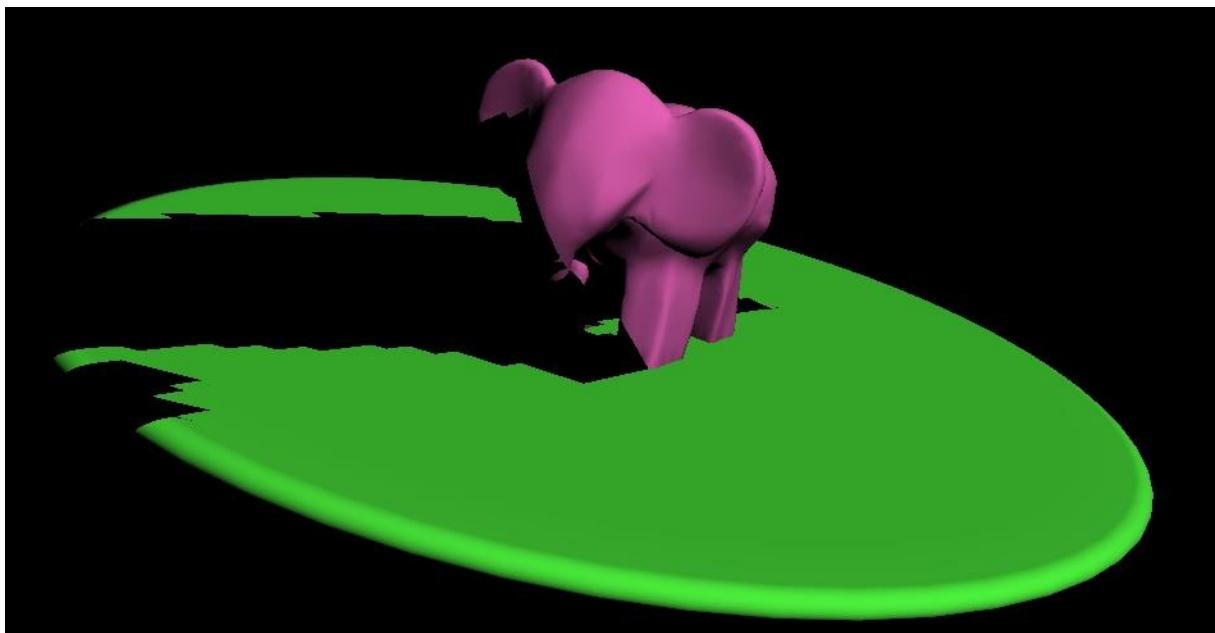
Uppgift 6



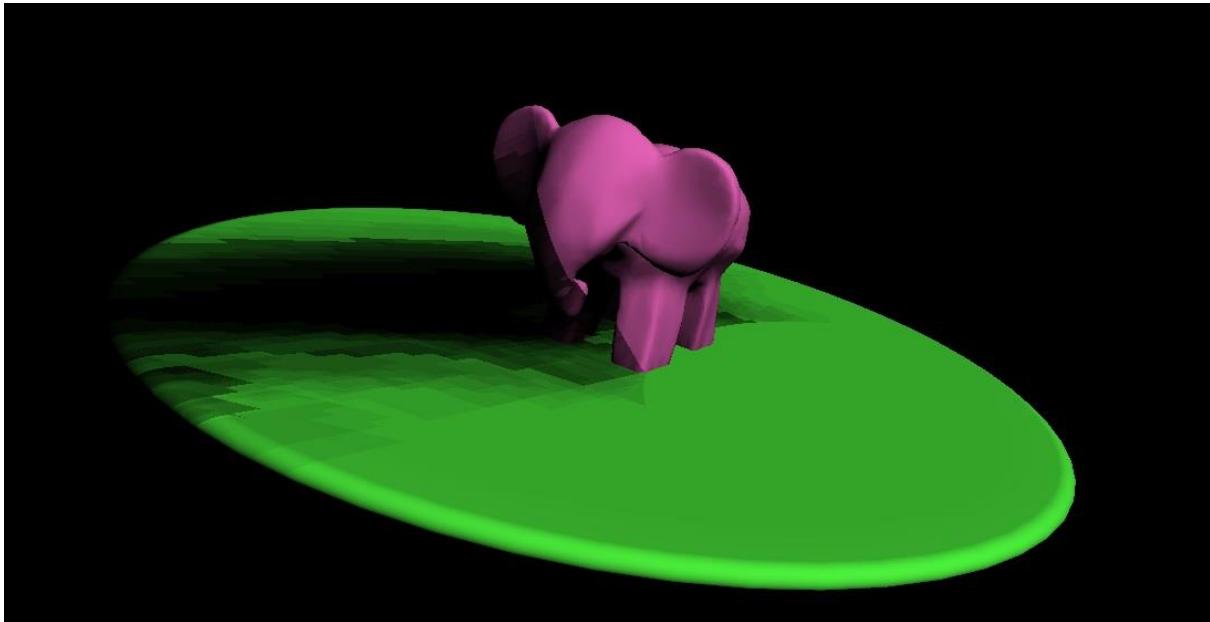
Låt Oss Leka med Lite Värden



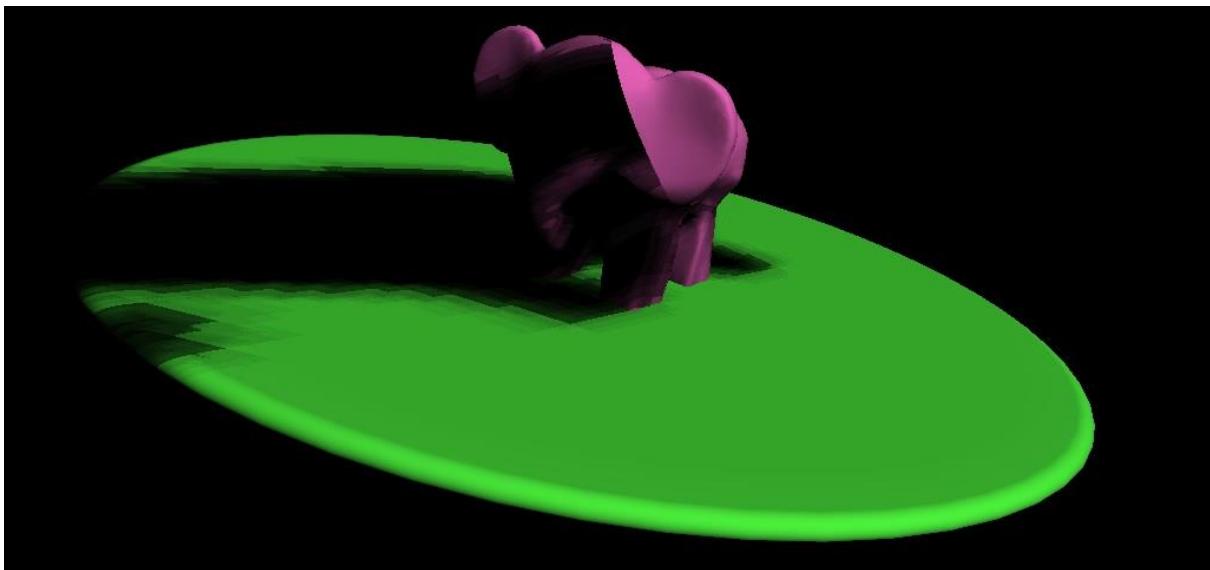
Detta är vår scen. En rosa elefant pryder en grön disk. Till höger utanför bilden finns en ljuskälla som orsakar att elefanten kastar en skugga till vänster om elefanten. Skuggan har samplats för att se någorlunda jämn ut och justerats med en parameter för att tacka elefanten med rimlig omfattning.



Om skuggorna inte samplats och multiplicerats ihop likt i den första bilden skulle skuggorna bli kantiga och otrevliga likt i denna bild. Här har samplingen stängts av genom att ställa samplingsytans offset till 0. Notera det kantiga utseendet. Det ser inte så jäkla kul ut, eller hur?



Men att bara ösa offset-area på samplingen löser inte så mycket. Det krävs mer arbete och fler samplingspunkter med en *balanserad* offset för att skapa jämn antialiasing i skuggorna. Med endast samplearean uppvriden återuppstår de otrevliga kanterna, då de inmultiplicerade samplingarna bara blir förskjutna kopior av originalet, inte ett "blurr" som det bör vara.



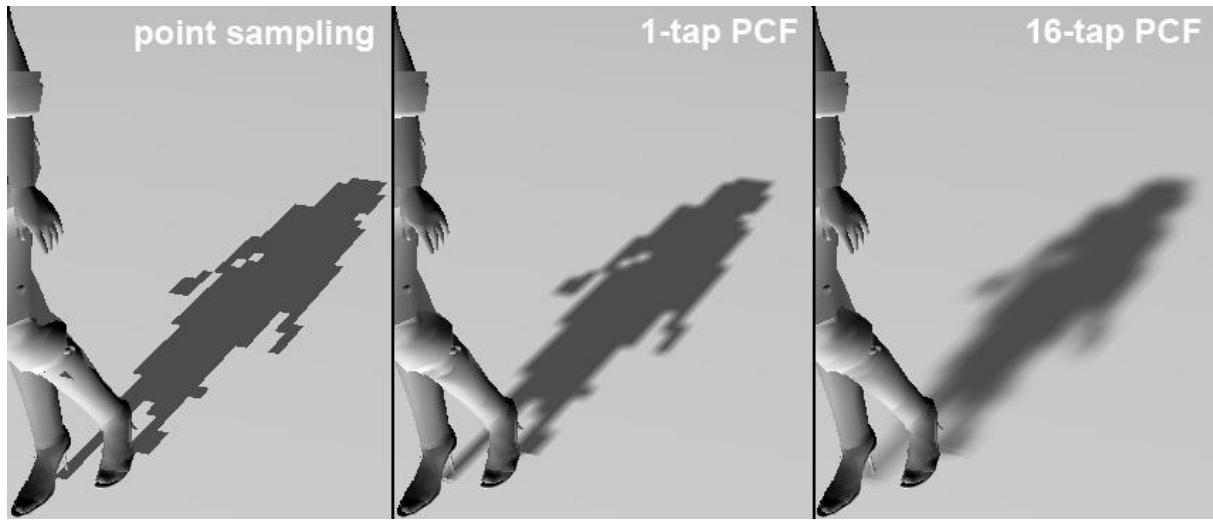
Vi lämnar samplingen och kikar istället på parametern som styr skuggans inverkan på elefantobjektet. När vi vrider ned den "klättrar" skuggan ned mot disken, och ökar vi den likt på bilden ovan täcks en större portion av elefanten av skuggan. Simple as that!

Fördelar och Nackdelar med Shadow Maps och Percentage Closer Shadows

Skuggor representerade av shadow maps är ofta mer diskreta än andra typer av algoritmer, då datorkalkylerade approximationer inte är oerhört effektiva som bas för skuggning i förhållande till andra metoder. Resultaten av skuggmappning är därför i vissa avseenden under kritik.

Skuggmappar som utgår från en konkret ljuskälla har problem med att skugga sitt eget källobjekt, vilket resulterar i en potentiellt inkonsekvent belyst scen och bruten realism.

Skuggmappning är dessutom begränsat till rymdens frustum, vilket riskerar att sabotera för vissa objekt i scenen.



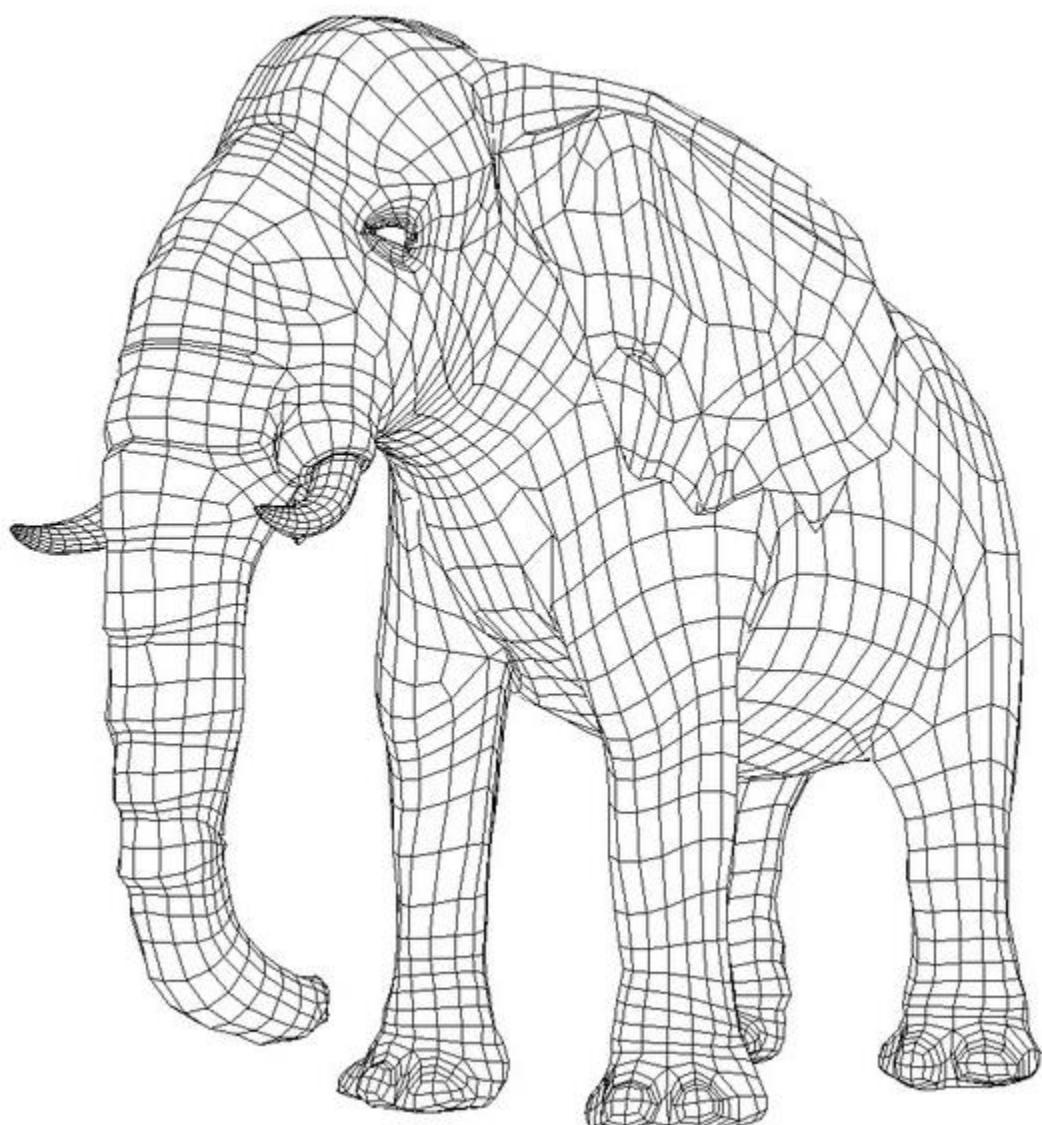
En fördel med skuggmappning är att det kan bevaras och återanvändas i en scen, förutsatt att dess geometri och ljuskällor är statiska, vilket är till enorm hjälp i flera situationer.

Det är även en smidig metod att tillämpa för att tillåta för skuggor att kunna kastas och träffas av flera olika objekt oberoende av varandra.

Med dessa aspekter i åtanke är skuggmappning ett väldigt effektivt sätt att skugga en scen, och med hjälp av percentage closer-filtrering kan resultatet av väldigt hög kvalitet skapas.

I och med att vi själva kan välja hur många samples vi hämtar för att skapa vår percentage closer-skuggning kan vi relativt enkelt skala skuggningens effektivitet mot den prestandamässiga kapaciteten som står till vårt förfogande.

Uppgift 7



Vad gör Parametrarna för Våra Skattepengar?

I denna uppgift arbetar vi huvudsakligen i screen space med en elefantmodell. Vi har till vårt ödmjuka förfogande inte mindre än **tre** fantastiskt spännande parametrar relevanta för oss att kika på.

Jag insisterar:

- Area
- uNoiseFrequency
- uBlendMode

Area fungerar mycket likt parametern i föregående uppgift som styrparameter för sampling. I detta avseende samplas elefanten för att kunna skapa gaussisk oskärpa (gaussian blur) genom att placera "spökkopior" över, under och bredvid originalobjektet. Högre area innebär större samplingsyta, alltså att "spökkopierna" förflyttas längre bort från originalet.

Parametern uNoiseFrequency ger ett uns av kontroll över brustexturen som kryper över hela skärmen. Genom att öka parametern ökas den tidsstyrda förskjutningen av texturens y-koordinater vilket resulterar i grövre brus, till den nivån att skärmen förefaller gravt pixelerad.

uBlendMode är en boolesk variabel som låter användaren byta mellan två olika blend modes (sammanslagningsoperatorer) i elefantens sista pass. Om uBlendMode är 1 (true) används en enkel Multiply blend. Är den 0 (false) används Exclusion. Mer om dessa fascinerande operatorer straxt.

Nej! Vi vill Höra mer om Operatorerna Nu, inte Straxt!

Okej, då går vi igenom dem lite snabbt. Det är alltså passen *colorBlurX* och *colorBlurY* vi ska mixa ihop med hjälp av en sammanslagningsoperator.

Den första sammanslagningsoperatorn är som sagt en mycket enkel **Multiply blend**. Den tar emot två element och slår samman dem genom multiplikation.

```
colorBlurX * colorBlurY;
```

Resultatet blir en något utjämnad blandning mellan passen. Det lägsta av varje värde kommer sänka det andra elementets motsvarande värde och vice versa. Busenkelt.



Den andra sammanslagningsoperatorn är en så kallad **Exclusion blend**. Exclusion fungerar likt *difference blend* genom att låta det ena elementet subtraheras från det andra på ett vis som alltid genererar ett positivt värde.

```
colorBlurX + colorBlurY - (2.0 * colorBlurX * colorBlurY);
```

Slutresultatet ser ut att likna en ungefärlig invertering av elefantens färger i en relativt låg kontrast.



Skillnaden Mellan att i Y-passet sampla från X-passet och att Direkt Sampla från Belysningstexturen



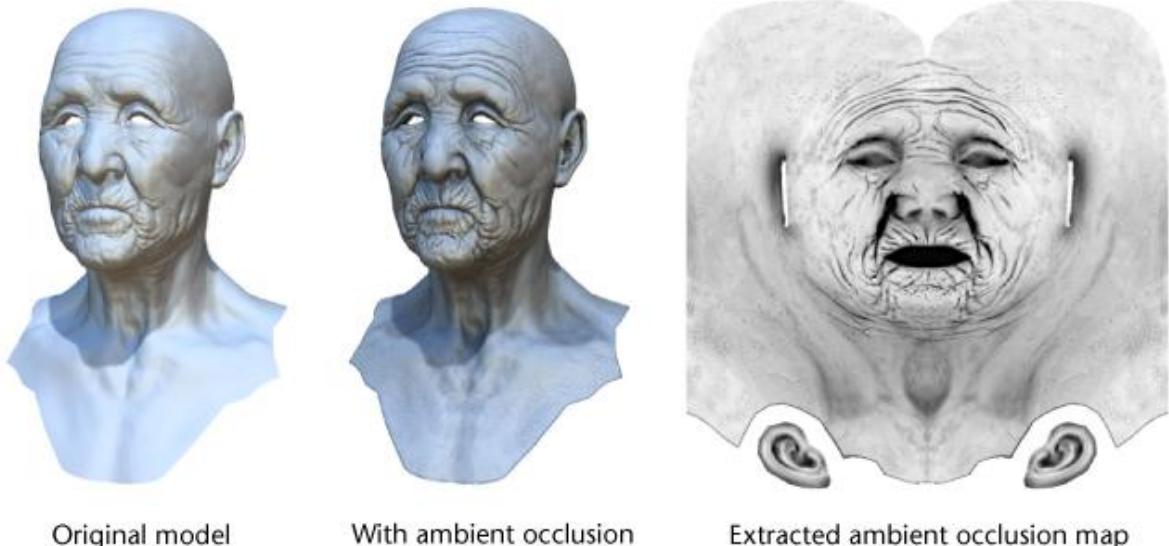
I denna sekvens av pass sker sampling för gaussisk oskärpa oberoende av tidigare pass. Samplingen i pass 2 sker utifrån elefanten och förskjuter resultatet i x-led. Samplingen i pass 3 sker precis som pass 2 utifrån originalelefanten och förskjuter resultatet i y-led. Förskjutningen sker på så vis i kors bort från originalet i mitten.

Hade passen jobbat mot varandra snarare än mot elefanten hade resultatet blivit annorlunda. Nästa sekvens av samplingar skulle då hamnat runt om hela objektet, snarare än endast längst respektive axel. Det hade genererat ett betydligt mer realistiskt resultat då blurren i hög frekvens runt elefanten skulle täckt mer eller mindre jämnt runt hela elefanten, lika mycket i varje riktning så långt samplingen tillåter. I låg area-utsträckning fungerar de oberoende samplingsaxlarna för ett dugligt resultat, men i mer omfattande effekter, som exempelvis motion blur som simulering av höga hastigheter i ett högupplöst racingspel kommer inte två axlar duga på långa vägar (pun intended). En mer associativ lösning krävs för att resultatet ska heta duga.

Fördelar med att Arbeta i Screen Space

När vi arbetar i skärmrymden har vi möjlighet att skapa shaders som utför samma arbete som i objektsrymd, men med mindre färre beräkningar. Vill vi exempelvis skapa en enkel phong-shader kan vi istället för att beräkna och bearbeta hörnnormaler, den virtuella kamerans position och normalisering i fragmentshadern direkt utifrån skrämräymden åstadkomma ett liknande resultat. Ett vanligt problem med denna typ av parering kan dock orsaka problem i form av exempelvis svarta kanter runt objekten.

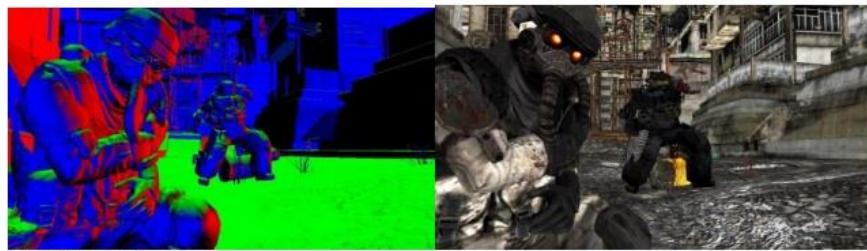
En annan teknik som kan utnyttjas genom shading i skärmrymden är ambient occlusion som samplar ytnormaler och blockerar det ambienta ljusets tillgång ytor med djupa z-värden. Det är en spännande och effektfylld metod för att implementera ljussättning.



Shading mot skärmrymden innebär i praktiken att vi kan utgå direkt från användarens vypunkt (det vi ser på skärmen) för att utföra de nödvändiga approximationerna som gör det möjligt att komponera en rad olika shadingtekniker.

Fördelar med att Jobba med Många Pass

Att arbeta i flera pass öppnar upp för en mängd olika shadingmöjligheter, bland annat så kallad *deferred shading* som skjuter upp den faktiska shadingen bortom de absolut första passen. Att arbeta i multipla pass låter programmeraren utföra olika beräkningar och operationer i separata steg, låta passen arbeta mot varandra och kompilerera ihop alla pass till ett slutgiltigt resultat. Detta gör alla variabler och egenskaper så som djup, ljusvinklar, ojämnheter, normaler och dylikt väldigt lättöverskådligt då varje approximation kan överblickas och granskas i sitt eget pass tills resultatet är i gångbart skick för slutmålet.



G-Buffer – view space normals and albedo



G-Buffer – roughness and specular intensity



G-Buffer – motion vectors and sunlight occlusion



G-Buffer light accumulation (including indirect lighting) and depth buffer

Hade vi använt samma algoritmer, tekniker och framför allt variabler som exemplet ovan i ett och samma pass hade det blivit betydligt svårare att kontrollera saker som ljussättning och vinklar på ett så pass översiktligt vis.

Endast genom användandet av pass tillåts programmeraren dessutom att skapa ett hierarkiskt målsystem som specificerar vilken kod som arbetar mot respektive pass. Vi kan enkelt skapa procedurellt exekverade pass eller, om vi så önskar, lagra effekter i pass som jobbar mot gemensamma ytor och grena ut mot en dynamisk och överskådlig arbetsyta av passlager.

Uppgift 8



Låt Oss Leka med Lite Värden

I denna ståtliga kryptshader har vi följt föregående raports visdomsord och tagit väl hand om vår kunskap om pass. Inte färre än 13 mer eller mindre unika pass lyser (eller mörklägger) med sin närvaro. Mycket kod, tveklöst. Knappast tomt på parametrar här. Ska vi inte ta och kika på dem?

BlurArea

Denna har vi sett förut. Mycket enkel i sin princip. BlurArea anger ett värde 0.0 – 1.0 för den offset i samplingen som låter blurra kryptans lightmap (fönstret). Om BlurArea = 0.0 sker ingen sample offset och resulterar i att ingen blureffekt uppstår. Högre värden förskjuter alla samples bort från originalet.

En omfattande skillnad i denna gaussiska oskärpa i förhållande till tidigare tillämpningar av tekniken är att vi istället för att låta två förskjutningar arbeta i var sin axel oberoende av varandra så arbetar passet för y-axelsförskjutningen direkt mot x-axelsförskjutningen, vilket producerar effekten nämnd i kapitel "*Skillnaden Mellan att i Y-passet sampla från X-passet och att Sampla från Belysningstexturen Direkt*" ur rapporten för uppgift 7. Fönstret sampas alltså i alla riktningar från origo, vilket ger en jämnare och snyggare oskärpa. Fantastiskt.

DepthOfField och FocalPoint

Nu doppar vi tårna i lite djupare vatten. Tekniker för realistiska linsegenskaper är vanliga i spel som förstapersonsskjutare där skärpedjup simuleras i kikarsikten och i spel som eftersträvar grafisk realism. Med hjälp av lite grafisk magi har vi lyckats återskapa två typiska egenskaper hos linser, vilka vi nu kan styra med hjälp av parametrarna DepthOfField och FocalPoint. Focal Point, eller brännpunkt, innebär det exakta avståndet från en lins som en yta är synlig skarpt och tydligt. Allt framför och bortom brännpunkt en kommer förefalla suddigt. Effekten uppstår i kameror som fokuserat på en punkt på ett visst avstånd, eller mer naturligt förekommande, i våra egna ögon när vi fokuserar på något på ett särskilt avstånd. FocalPoint-parametern styr alltså hur långt fokusintervallet, eller fokuspunkten, befinner sig. Ett högre värde betyder att fokus ligger längre bort från den virtuella kameran.



Depth of Field, eller skärpedjup, styr det faktiska fokusintervallets storlek. Om vårt skärpedjup är litet ser vi en mindre yta (field) som skarp än om skärpedjupet var större. Vänligen kika på bilden ovan. Bilden demonstrerar två bilder tagna med samma motiv från samma position, men med olika skärpedjup. Lågt skärpedjup (vänstra fotografi) tillåter för ett betydligt kortare intervall med fokus. Högt skärpedjup (högra fotografi) låter en större portion av en rymds fält vara fokuserad samtidigt.

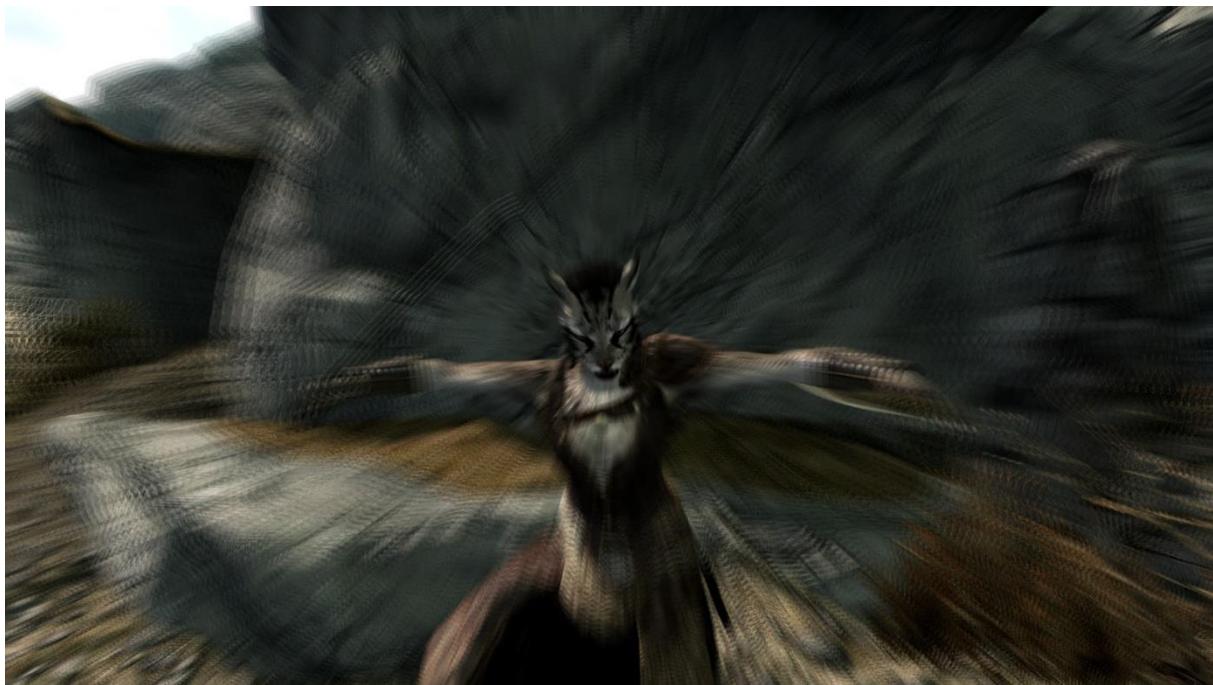
SepiaIntensity och TextureIntensity

Busenkla. SepiaIntensity justerar har kraftigt det sepiafärgade filtret påverkar det resulterande utseendet på shadern. TextureIntensity gör det samma med 3D-texturen som skapar en regnbågseffekt utanför fönstret. 0.0 dödar effekterna, 1.0 ger full effekt.

RadialBlurIntensity och RadialSampleArea

Till sist har vi ännu en spännande linseeffekt, radial blur. Radial blur samplar och förskjuter kopior av shaderns output bort från en specifik punkt i rymden. I vår shader sker blureffekten under hela tiden som den är påslagen, medan effekten i vanliga fall snarare används under korta eller intensiva perioder. Ett vanligt exempel på användande av radial blur är racingspel, där effekten används runt skärmens mittpunkt eller fordonets färdriktning för att simulera känslan av mycket höga hastigheter. Ett annat exempel på en ganska intressant användning av radial blur-effekten är *The Elder Scrolls: Skyrim*, vars egenskaper enkelt kan parallelliseras till min egen shader. De har valt att implementera radial blur som effekten av den fysiska responsen på plötslig och kraftig smärta. När spelaren tar skada uppstår en mycket hastig radial blur-effekt. Det ser ut som att spelaren tappar visuell fokusförmåga, som på grund av chock, i upp till en sekund.

Det går utmärkt att ta en titt på skärmdumpen från spelet nedan. Den första egenskapen av effekten som är värd att notera är effektens utgångspunkt, det vill säga var origo för samplingen befinner sig på skärmen. I *Skyrim* sker detta intressant nog från den riktning som skadan sker. Träffas jag av en pil i min högra sida kommer radial blur att ske bort mot vänster på skärmen. I detta exempel kommer skadan rakt framifrån, och därför bluras skärmen från mitten och ut mot alla skärmens kanter. I kyptan är linseeffektens origo alltid i mitten.



Den andra egenskapen vi borde kika på är hur många samples som används. I min egen shader sampelas målet nio gånger, något som gav en ganska jämnt fördelad och lagom frekvent oskärpa. I *Skyrim*, som tvingats optimeras ganska hårt på grund av den sjunde generationens spelkonsolers begränsande hårdvara, hämtas fem samples. Titta riktigt noga på khajiitens svärd i bilden; exakt fem svärdseggar kan nätt och jämt uttydas. Det innersta, närmast skärmens mitt är med största sannolikhet originalet, och de yttersta fyra eggarna är ett resultat av radial blur-effekten. Coolt.

En tredje faktor som väger in i effektens utseende är sample arean. I kryptan styrs det med hjälp av parametern RadialSampleArea. Ett högre värde innebär då att bilden förskjuts längre bort från origo. Även här ser det ut som att Bethesda Softworks har valt en intressant lösning. SampleAreaen förefaller bero till viss del på hur mycket skada spelaren tar. En smäll på käften resulterar i en något mindre synskada än om en eldsprutande drake sätter sig i knät på spelare. Måhända att jag ser fel, men oavsett så är det en ganska frän lösning. Om det inte är den avsedda implementationen bör de borde definitivt anställa mig.

Till sist har vi en intensitetsparameter RadialBlurIntensity, ett värde 1.0 – 10.0. Den styr hur pass kraftigt blur-effekten viktas mot avståndet från origo. Ett högre värde ger snabbt en kraftigt förvrängd bild, medan lägre värden skapar en enklare effekt. Notera att den generella strukturen kring radialeffekten, som kommentarer i koden anger, är tagen/kopierad från en blogg skriven på hemsidan [GameRendering](#) med deras tillstånd.



blaargh

Deferred och Hänförd

Deferred, eller ”förskjuten”, shading innebär kortfattat att det/de första passen enbart används till beräkningar av data nödvändigt för shaderarbetet, utan att faktiskt utföra någon egen shading.

Den uppenbara fördelen med detta på projektnivå är att det blir betydligt enklare att överse och verifiera shaderns data, i och med att hela pass är dedikerade till deras beräkning. Antag att elakartade buggar skulle uppstå i vår krypta. Tack vare vår passuppdelning kan vi enkelt granska och utesluta om det är normaler som spökar, om scenens djup inte beräknas korrekt, om det är färgerna som trasslat ihop sig eller om ljusvektorn inte beräknas korrekt. Genom att stänga av pass ett i taget kan vi precisera exakt var problemen uppstår, något som varit mer än hjälpsamt i detta projekt. Hade samma arbete utförts i endast ett fåtal pass med de grundläggande beräkningarna inbakade hade felsökning inneburit ett katastrofalt ödslande med tid i förhållande till min erfarenhet med grafikprogrammering.

En annan omfattande fördel med tekniken är att ljus- och geometriska beräkningar kan separeras, vilket gör det möjligt att tillämpa ljussättning på de ytor som faktiskt påverkas av ljuset. Tack vare detta kan flera ljuskällor implementeras utan att överväldiga den grafiska beräkningskapaciteten.



En nackdel med att arbeta med deferred shading kan vara rent strukturell i det avseende att ändringar som påverkar alla pass kan kräva mycket arbete för att organisera. Att stycka upp en shader och få dem att arbeta mot varandra innebär många saker att hålla reda på samtidigt. Problemet är dock mer en fråga om multipassshading snarare än just deferred shading.

Det var stundvis under projektets gång svårt att skapa en verklighetsförankrad bild av samtliga pass. Visst finns en tydlig logik i en bild enbart med ljus, färger, mönster, normaler eller djup, men många gånger kunde det vara intetsägande information. Hade vi arbetat direkt mot ett objekt hade det varit enklare att se den konkreta effekten av ett pass, snarare än ett låta visuell data hanteras separat från resten av vårt arbete. Det finns givetvis även en rad tekniska nackdelar med deferred shading, men för min egen del upplevde jag mig påverkas jag väldigt lite av dessa i mitt arbete.

Fördelar och Nackdelar med Implementationen av min Depth of Field

Mitt pass för linseffekter som simulerar skärpa har ett antal kvaliteter, men är långt ifrån exemplarisk i avseendet om realism. För att summera fördelarna kan vi börja med att konstatera att den fungerar. Det är alltid en bra början. Det framgår tydligt vad passet har för effekt på shadern direkt när man börjar spaka i parametrarna. Brännpunkten och skärpedjupet har precis samma effekt som den skulle haft på en analog kamera.

Det är bra. Att imitera en kamera visar ju på förståelse om linser. Men nästa nivå hade uppenbarligen varit att simulera ett öga och dess beteende istället. Med lite mer tid och energi hade det inte varit så svårt att utnyttja djupmappen för att dynamiskt förskjuta brännpunkten till samma djup som kamerans mittpunkt. Fokuset hade då hamnat på precis samma djup som den punkt vi tittar på, det vill säga mitten av skärmen. För extra realism hade vi dessutom använt en tidsvariabel för att skapa en liten fördröjning i brännpunktsjusteringen, likt hur linsen på en kamera justeras för att fokusera. Det hade skapat illusionen av att den virtuella kameran hämtar fokus över en bråkdel av en sekund, precis som det mänskliga ögat.

Samplingen som används för att simulera oskärpa är kass. There, I said it. Vid kraftig oskärpa förskjuts samples så långt att de kan urskiljas från varandra. Min syn är utmärkt, så jag kanske inte bör uttala mig ur någon annans mun, men jag har trots det en stark känsla av en synskada som påverkar skärpan inte genererar artefakter i ögonvrån med tydligt definierade kanter (se bilden nedan). Det behövs fler, lägre frekvent samples för att göra oskärpan mer trovärdig.

Avståndet för skärpedjupet fungerar utmärkt. Det korrekta avståndet beräknas som det bör i en sfär från kamerans position i rymden. Med ett väldigt precist skärpedjup kan man därför enkelt urskilja var fokuset befinner sig och se att det skapas en cirkulär rand runt om kameran (se bilden nedan). Precis som sig bör. Men om målet var att efterlikna hur ett öga fokuserar ser det ärligt talat bara dumt ut. Ögat fokuserar på en punkt och alldelens straxt runt om den, inte i en rand längs en hel yta. Fel, fel, fel. Det måste åtgärdas, trots att vi fick vad vi beställde, om vi vill ha realistiskt fokus.



Fördelar med att Jobba med Radiella Effekter genom Deferred Shading

Radial blur fungerar likt tidigare nämnt i kapitlet "*Låt Oss Leka med Lite Värden*" i denna uppgiftsrapport genom att sampla och förskjuta skärmresultatet bort från en angiven punkt för att skapa ett oskarpt "blurr". Mer detaljer kring hur mitt radial blur-pass fungerar finns att utläsa i delkapitlet om shaderns parametrar. Nu ska vi diskutera varför just deferred shading funkade ovanligt bra till detta.



Just radial blur som jag valt att använda förefaller inte påverkas enormt mycket av det faktum att shadern övergripande utvecklats med hjälp av deferred-tekniken. I fall där andra typer av effekter implementerats är skillnaden kanske större. Det väger skarpt in i resonemanget att olika aspekter av scenens data beräknas i separata pass. Med hjälp av denna struktur kan vi granska vår radialeffekts beteende i förhållande till de olika variablerna. Av valfri fantasifull anledning kan det bli aktuellt att undersöka så att radialeffekten inte har en allt för står inverkan på scenens 2D-texturer. Vi kan då enkelt rikta radialeffekten direkt mot passet som beräknar diffustexturen för att försäkra oss om att effekten inte inkräktar på passets integritet, och därefter sova lugnt om nätterna.

Samma sak gäller givetvis även för ljus- och djupmapparna. En förskjutning i texturkoordinaterna, om än via en skärmnära effekt som denna, kan ge otrevliga resultat om inte grundläggande data tas hänsyn till. Tack vare deferred shading kan vi lättare se till att effekten exekveras korrekt och snyggt.

Färgkorrigering

Kul grej det där med färg. En helt vetenskap, och ändå ser vi bara en löjlig fraktion av det fullständiga ljusspektrumet. Mer paradoxalt blir det när du hör att färgkorrigeringen jag valt för min shader lägger på ett grå-beige sepia-filter över hela bilden. Ett förkastligt slöseri. Men vad gör man inte för konst?



konst

Här är den finurliga lilla koden som färgar allt i en hobbyartistisk instragramanda:

```
float grey = dot(outCol.xyz, vec3(0.299, 0.587, 0.114));
vec4 luminance = vec4(grey * vec3(1.2, 1.0, 0.8), 1.0);
outCol = mix(vec4(vec3(luminance), 1.0), outCol, (1.0 - uSepiaIntensity));
outCol = vec4(outCol.xyz, 1.0);
```

En påtaglig effekt på det slutgiltiga resultatet med ett minimalt och oerhört konfigurerbart stycke kod. Omedelbart går det upp för dig, varför man vill låta en shader ansvara för en så viktig grafisk effekt. Ändå in på flyttalens minsta detaljer kan vi justera exakt vilka värden våra färger ska ha. Det ger oss utmärkt detaljfrihet, särskilt om vi låter parameterisera dessa variabler för finjustering i realtid.

En annan fördel är det faktum att sepia är en effekt som direkt påverkar scenens färger och ljus. Att försöka tillämpa liknande effekter med hjälp av post-processing, externa filter eller med hjälp av andra verktyg än en shader vore komplicerat, eftersom att de med största sannolikhet inte har samma tillgång till scenens ursprungliga färg- och ljusförhållanden. Med hjälp av en shader kan vi direkt se och kontrollera saker som ljuskällor, ljusvektorer, diffusmap och färgtexturer och justera vår färgkorrigering därefter, snarare än att famla i mörkret (läs ljuset) efter ett lämpligt resultat.

Att låta en shader utföra färgkorrigeringen ger programmeraren utmärkt kontroll och dynamik. Effekten kan stängas av fullständigt med ett knapptryck eller finjusteras med alla parametrar man kan tänka sig och önskar implementera. Chansen att få det exakta resultatet man önskar är betydligt högre i detta format än de flesta alternativ. Observera att kodstycket ovan är taget/kopierat från exempelkoden på hemsidan för uppgifter i shaderprogrammering.

Uppgift 9



Jaha, det var väl Bra att du Gjort en Shader, men Vad ska du med Den Till?

En utmärkt fråga. Jag har ett väldigt konkret förslag på vad min skinande, bubblande, metalliska shader hade kunnat användas till. Ytan på ett böljande vatten eller en enkel simulering av kokande eller smält metall. Mina tankar förs delvis till slutscenen i den onödigt våldsamma actionrullen *Terminator 2*, där robotar från framtiden byggs av metaller med någon slags halvmagiskt, associativt platsminne som låter dem smälta ihop och laga sig själva när de går sönder. Bilden nedan demonstrerar hur en robot lagar sig själv efter att ha krossats i tusen bitar, likt svampdjur som låter lösa celler sammanfogas med hjälp av stamceller.

Science fiction och svampar åsido, effekten av flytande metall som harmoniskt flyter omkring, bubblar och reflekterar sin omgivning är inte att klaga på. Jag tror absolut att min egen shader kan träda in i rollen som en liknande effekt. Om inte på en smält mördarrobot från framtiden så kanske i mer realistiska sammanhang som i fabriksmiljöer, metallindustrier eller virtuella sjöar.



Shadern besitter flera egenskaper som är jämförbara med flytande metaller som kvicksilver eller smält silver. Till att börja med använder shadern inga texturer. Istället beräknas en väldigt tydligt definierad reflektion mot hela objektets yta. Det vi faktiskt ser av objektet är endast reflektionerna av scenens kubmap, det vill säga allt som omringar objektet. Det gör att objektet ser ut att vara väldigt blankt och rent, som ett fläckfritt silvermynt eller glasklart vatten mot en solnedgång.

Näst har vi en matematisk algoritm som använder sinusoperatorer för att förskjuta objektets koordinater med systemtiden som iterator. Det är dessa två rader kod nedan som skapar den böljande, nästan bubblande effekten på objektets yta. Det påminner om vattenytan på en lugn sjö som inte är riktigt perfekt vindstilla eller vad som skulle kunna vara en bred ränna med långsamt flytande metall.

```
ntex.x += (sin((texCoord.x + (timeFactor)) * uRippleTimeFrequency) * uRippleFrequency);  
ntex.y += (sin((texCoord.y + (timeFactor)) * uRippleTimeFrequency) * uRippleFrequency);
```

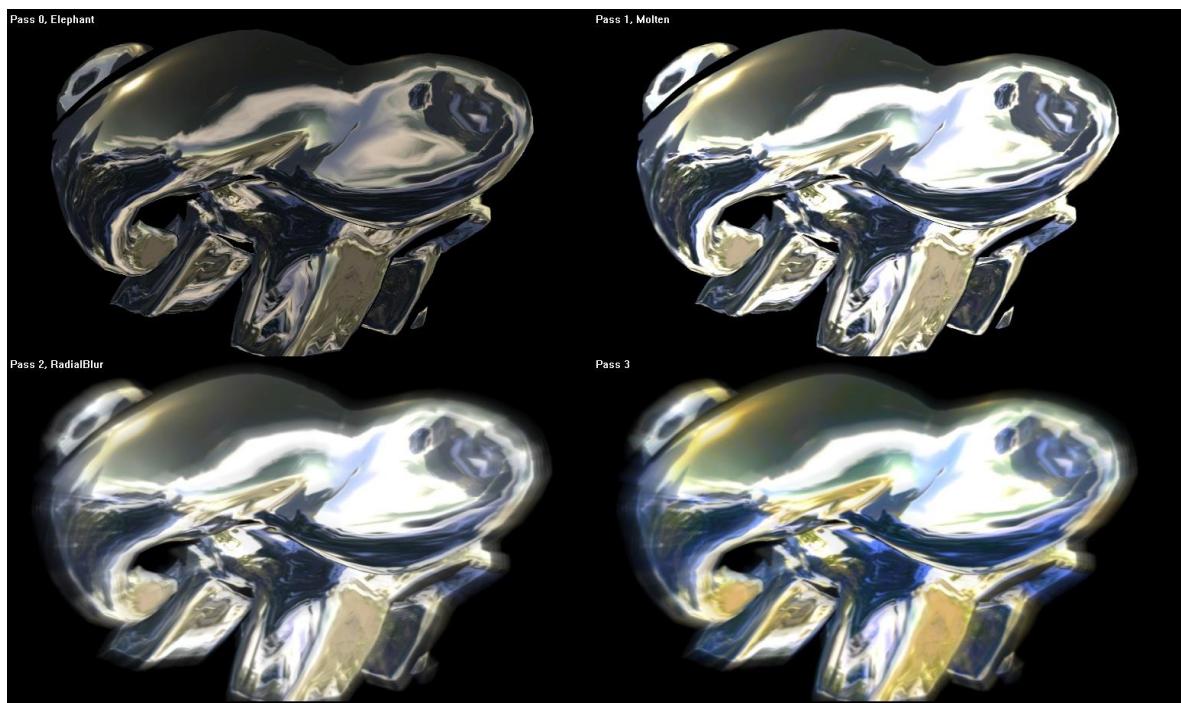
Näst finner vi en linseffekt med diskuterbar betydelse för just den likvida effekten vi hoppas kunna utnyttja. Linseffekten är en något uppdaterad och raffinerad version av den radial blur vi skapade i uppgift 8. Effekten skapar ett oskarpt blurr/oskärpa runt objektet. Huruvida vi vill segrera detta pass uteslutande som en extra effekt får var och en avgöra, men i resonemanget om en flytande metallshader ser jag potential i detta pass som en värmeeffekt. Eftersom att texturkoordinatförskjutningen sker på hela objektet och alla dess omgivande element kommer blur-effekten bölja i samma stil som resten av objektet. Det kan, med lite ytterligare polering kunna användas för att återskapa effekten av en hägringar i form av värmevågor, likt de som uppstår ovan solstekta vägar, skorstenar eller brödrostar som går varma. Lite fantasi och vilja kan ta oss dit.

I shaderns sista pass finner vi en färgkorrigering som låter oss utföra enkla justeringar i utseendet. Som gjort för att kalibrera metallens eller atmosfärens egenskaper. En lite blåare metall kan satureras med blå färg, en lite dunklare miljö kan mötas upp med en lite mörkare färgkorrigering.



Upplägget som inte blir mycket mer Metal än Så här

I min relativt enkla shader använder vi endast fyra pass. Det är tillräckligt för våra ganska grundläggande operationer och jag har av den anledningen valt bort att arbeta genom deferred shading. Vi kikar med fördel på våra pass, ett i taget, för att bilda en uppfattning om hur upplägget ser ut. From the top, maestro.



I shaderns första pass hittar vi basen till shadern som i huvudsak arbetar med ljusegenskaper och bearbetning av grundläggande data. Här normaliseras objektets olika vektorer, output nollställs, eventuella diffusfärger, diffuse attenuation och fresnelreflektioner tillämpas. Till sist implementeras reflektionseffekten mot shaderns kubmap (ett bergslandskap i detta fall) som ger det annars helblanka objektet sitt metalliska utseende. I detta skede är shadern fortfarande statisk och oföränderlig.

I pass två utför vi texturkoordinatsförskjutningsalgoritmen som skapar den böjande effekten. Parametrarna i exemplet ovan är inställda så att denna effekt inte är enormt påtaglig, därfor kan det vara lite svårt att urskilja genom enbart en skärmdump. Men kika i projektfilen också, det ser faktiskt riktigt neat ut. För att hålla god struktur och ett tydligt flöde i operationer genom hela utvecklingsprocessen valde jag att inte låta pass utföra mer än en större operation på egen hand. Det gör dessutom denna redovisning enklare då jag kan hänvisa till det andra passet som den enda implementeraren av noise-effekten.

I det tredje passat implementeras radialeffekten. Det syns som tydligast i objektets kanter, som kopierats och blurrats ut. Som nämnt baserar sig koden på samma exempel från GameRendering-bloggen. Skillnaden på radial blur-effekten i denna shader till skillnad från kryptan är i första hand att en generös mängd samples hämtats, 21 stycken närmare bestämt, för att ge en så ren och jämn blur som möjligt. Resultatet blev jag riktigt nöjd med. Det är en schysst blur.

Till sist färgkorrigeras objektet i det slutgiltiga passet med en saturation-formel. Den är i bilden ovan en aning överdriven just i redovisningssyfte. Notera att gula, blå och gröna toner ser kraftigt accentuerade ut, effekter som kan vara helt avgörande för om en vätska i en virtuell scen är trovärdig eller inte. Det finns inget värre än en besservisser till metallurg som pekar och skrattar åt ditt tappra försök till en metall-shader på något low-life forum någonstans i världen. Det är då du kommer vara tacksam för en färgkorrigeringsparameter som elegant kan polera detaljerna i din shader.

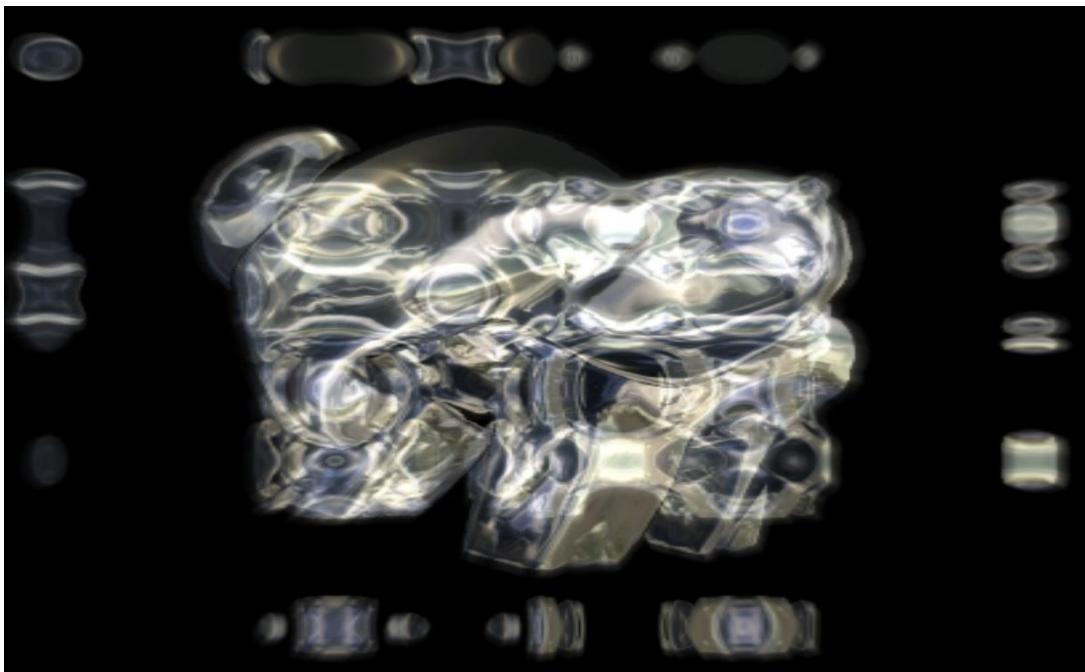
Låt Oss Leka med Lite Värden

All right, nu kan vi den här övningen. Vi drar igång direkt. Vi går loss på alla roliga parametrar.

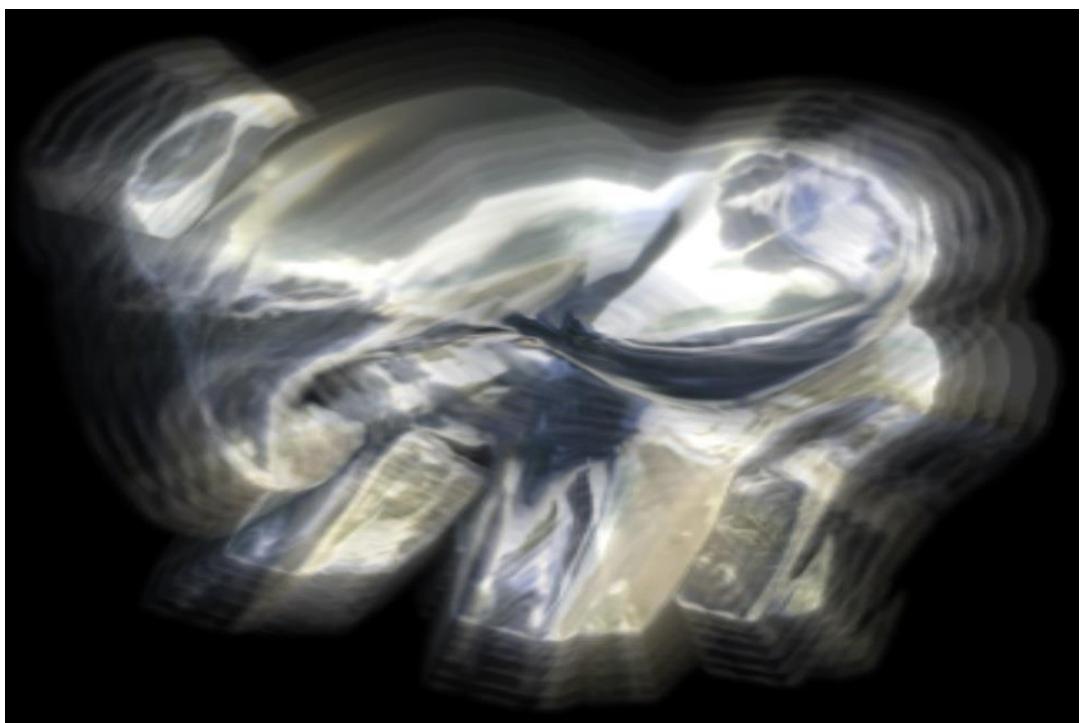


Vi börjar från början. Här har vi shadern i sitt ursprungstillstånd, vilket till viss nivå demonstrerar alla de olika passen. Vi ser kubmappen i reflektionerna, färgmättnaden har sänkts en aning, vi ser att ytan böjlar sig något i förskjutningen och en svag radial blur-effekt kan uttydas runt öronen och tassarna.

Häftigt osv.



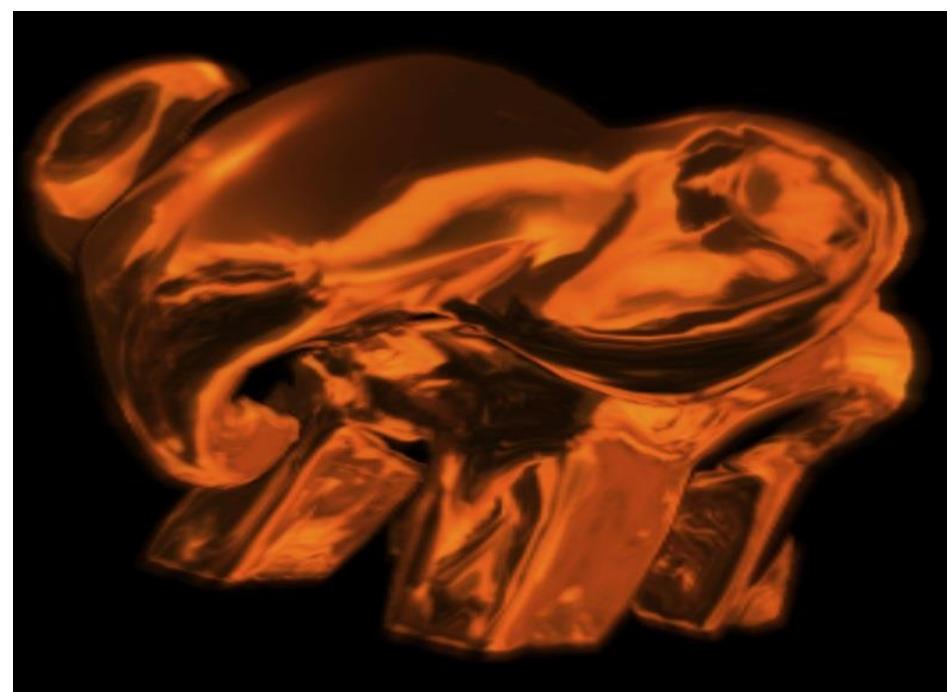
Nu händer det grejer. Om vi drar upp parametern **RippleTimeFrequency** kommer inte mycket mer att hända än att texturkoordinatförskjutningen sker fler gånger per sekund, alltså kan vi styra hur energiskt ytan ser ut att bubbla/bölja. Jättebra, men effekten är nästan omärkbar per skärm dump. Det vi har gjort här är att dra upp **RippleFrequency**, frekvensen på texturförskjutningen. Det betyder en kraftigare förskjutning i alla riktningar, vilket efter en viss nivå inte bara få objektet att se helt urspårat ut, texturartefakter läcker bokstavligt talat ut bortom det faktiska objektet. En häftig hägning, men passar sig snarare bättre till filmer eller spel som handlar om LSD än metallindustrier.



Här har vi marginellt ökat intensiteten på radial blur-effekten (**RadialBlurIntensity**) och markant dragit upp sample arean (**RadialSampleArea**). Resultatet blir att blur-effekten förskjuter objektet så pass långt att de olika sample-elementen kan urskiljas separat. Inte så vacker kanske, men ganska effektfullt. Jämför gärna effekten och antalet synliga samples med skärm dumparna från *Skyrim* i rapporten till uppgift 8.



Här har vi som du kanske redan gissat kastat oss över saturation-parametern, **SaturationIntensity**. Saturation-parametern ökar eller minskar de synliga färgernas mättnad. Om **SaturationIntensity** = 0 kommer alla färger mattas ut och objektet ses i gråskala. Är värdet 1 kommer ingen skillnad från originalet synas. Högre värden kommer skärpa och öka färgerna över sitt ursprung. Ett negativt värde kommer producera färgernas invers, vilket i detta exempel skulle få elefanten att se gulbrun, ljusblå och rosa ut. Härjigt och användbart.



Till sist den busenkla parametern **DiffuseColor** som anger objektets basfärg (i det första passet). Det ger oss möjligheten att producera output som imiterar vätskor i andra färger än bara helblankt. Exemplet i skärmdumpen ovan ser ut att ha någon slags brons- eller kopparaktigt utseende.

De Viktiga delarna av Koden – Citerade och Beskrivna av undertecknad

Reflektion med kubmap

```
// Create cube map reflections using the negated eye vector and the normal
vec3 R = reflect(-E, N);
R = textureCube(cubeMap, R).xyz;
vec4 reflection = mix(vec4(R, 1.0), outputColor, uReflectionIntensity);
```

OpenGL-funktionen *reflect* tar två vektorer som argument, en infallsvektor (*incident*) och en normalvektor. Genom att ange den normaliserade ögonvektorn beräknar vi reflektionen utifrån den virtuella kamerans position. Med den standarddefinierade datatypen *texturecube* anger vi kubmapstexturen (bergslandskapet) och den beräknade reflektionen. Vårt objekt är nu redo att reflektera sin virtuella omgivning.

Förskjutning av koordinater

```
// Add a factor to manipulate Time parameter externally
float timeFactor = Time * 0.1;
ntex.x += (sin((texCoord.x + (timeFactor)) * uRippleTimeFrequency) * uRippleFrequency);
ntex.y += (sin((texCoord.y + (timeFactor)) * uRippleTimeFrequency) * uRippleFrequency);
```

Med den standarddefinierade variabeln *Time* som utgångsläge kan vi dynamiskt rendera vårt objekt baserat på antalet sekunder som passerat. För varje tidsenhet som passerar förskjuter vi objektets koordinater, x- och y-koordinater var för sig, mot värdet av en ständig fluktuerande sinuskurva. Övriga parametrar styr algoritmens omfattning och frekvens, men det visste du säkert redan.

Radial Blur

```
// Create a vector pointing towards the screen center
vec2 direction = 0.5 - vTexCoord;

// Calculate distance to the screen center
float dist = sqrt((direction.x * direction.x) + (direction.y * direction.y));

// Normalize direction using distance
direction /= dist;

// Collect samples towards the center of the screen
for (int i = 0; i < 21; i++){
    // Apply and add textures blurred using the direction and samples
    sampledOutputColor += texture2D(Texture, vTexCoord + direction * samples[i] *
        uRadialSampleArea);
}

// Divide by the number of samples
sampledOutputColor *= 1.0 / 21.0;

// Weighten blur effect against the distance to the center
float t = dist * uRadialBlurIntensity;
t = clamp(t, 0.0, 1.0);
```

Detta är hela kärnan till radial blur-passet. Vid det här laget har jag förklarat ganska många gånger vad det är som sker, men jag skall ta och summera kodens innehörd.

Vi skapar en vektor med riktning mot skärmens center, det är vår referens till effektens origo. Vi beräknar och normaliseras avståndet till mittpunkten och inleder sedan en for-loop. Loopen itererar igenom en fördefinierad lista med totalt 21 sample-värden och samlar samplade kopior av objekten förskjutna i riktning från skärmens mitt. Slutresultatet jämnas ut, clampas och viktas mot ett antal variabler innan det skickas vidare som output.

Saturation

```
// Saturation
vec4 outputColor = texture2D( Texture, vTexCoord);
vec3 LuminanceWeights = vec3(0.299, 0.587, 0.114);
float luminance = dot(outputColor.xyz, LuminanceWeights.xyz);
outputColor = mix(vec4(vec3(luminance), 1.0), outputColor, uSaturationIntensity);
```

Denna algoritm är likt sepia-filtret tagen/kopierad från exempelkoden på hemsidan för uppgifter i shaderprogrammering. Den fungerar genom att skapa en färg baserad på scenens nuvarande textur och mixa in den med skalärprodukten av originalfärgen och en viktande färg med värdena 0.299, 0.587, 0.114. Resultatet, skalat med hjälp av en intensitetsparameter blir att bildens färger mättas, tunnas ut och inverteras beroende på det angivna värdet.

Problem och Svårigheter som uppstått under Projektets Gång

Min shader är relativt enkel i det avseende att det samlar en handfull av de tekniker jag lärt mig som jag uppskattade bäst och anpassade dem för att kunna producera den typ av shader jag ämnade. Således hade jag minst en viss mängd förståelse för varje steg i min utvecklingsprocess, vilket gjorde att jag inte stötte på några omfattande problem. Men för goda mätvärden vill jag peka ut ett antal detaljer som inte stod glasklara för mig.

Bland de allra första effekterna jag implementerade var texturförskjutningsalgoritmen. Idén var min egen men dess implementation fick jag söka hjälp för. Svårigheten låg i att förstå hur jag skulle greppa en referenspunkt (objektet i scenen) till själva manipulationen samt vilken typ av matematisk formel som var mest lämplig för en tidsenlig förskjutning (en enkel sinus- eller cosinuskurva visade det sig). Detta löste sig rappt med lite professionell handledning av allas vår grafiklärare.

Vid ett senare skede beslutade jag mig för att implementera färgkorrigering, men då den enda typ av färgkorrigering jag hade erfarenhet av var sepiatoningen fick jag problem med att välja en färgkorrigering som passade just min typ av shader. Enligt originalidén var texturer, bortsett från kubmapstexturen, fullständigt uteslutna för att inte riskera att bryta illusionen av en helblank vätska. Efter lite experimenterande med olika tekniker fann jag att mättnad (saturation) lämpade sig utmärkt för shadern, eftersom att den fungerade oberoende av eventuella texturer eller grundfärgar i själv objektet. Mättnadskorrigering kan agera som ett filter för reflektionerna från kubmapen och således utgöra en egenskap i objektytans material. En blekare yta simulerar en mattare yta. Så småningom valde jag även att lägga till en basfärg, då det tillförde ett element av dynamik till shadern eftersom att det möjliggjorde imitering av vätskor och material med annat än en helblank yta.

En Lista med Referenser till Källmaterial eller Shaders jag utgått från

- Akenine-Möller, Tomas, m.fl. (2008) *Real-Time Rendering* (3:e utgåvan), CRC Press, Boca Ranton
- Rost, Randi J. m.fl. (2010) *OpenGL Shading Language* (3:e utgåvan), Pearson Education, Boston
- Game Rendering, *Radial Blur Filter*,
<http://www.gamerendering.com/2008/12/20/radial-blur-filter/>, [2014-05-28]
- OpenGL, *Reflect Documentation*,
<https://www.opengl.org/sdk/docs/man/html/reflect.xhtml>, [2014-05-28]
- OpenGL, *Texturing Introduction*,
<http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/texturing.php>, [2014-05-29]