

# Fördefinierade funktioner

En riklig mängd med det mesta man önskar sig!

Trigonometri: sin, cos, atan mm

Andra matematiska funktioner: pow, exp, abs, fract, mod, min, max, clamp...

Geometriska funktioner: length, dot, cross, normalize, reflect...

Texturfunktioner, texture2D mfl

Lokal derivata (för bump mapping): dFdx, dFdy

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## Brusfunktioner

Med brusfunktioner kan många naturliga företeelser modelleras, t.ex. moln, ytor, mönster...

“Perlin noise” introducerade brusfunktioner i datorgrafik. Med “färgat brus”, med olika karakteristik i olika frekvensband, kan önskade beteenden modelleras.

Tre sätt att använda i shaders:

- 1) Förgenererade texturer
- 2) Egen brusfunktion implementerad i shader
- 3) Inbyggd brusfunktion i GLSL!

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Brusfunktioner

noise1, noise2, noise3, noise4 är brusfunktioner i 1-4 dimensioner.

En brusfunktion är en funktion  $f(x)$ , där funktionsvärdet varierar med  $x$ , med ett stokastiskt beteende som liknar slump men inte egentligen är slumpmässigt.

- Returvärde i  $[-1, 1]$ , medelvärde 0
- Givet argument ger alltid samma svar, varierar ej med tiden eller antal anrop
  - Bandbegränsad
- Statistiskt beteende oförändrat av translation och rotation

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Brusfunktioner

En värdefull funktion som inte finns i alla kort än!

- Kostar inget VRAM. Ger procedurella 3D-texturer utan problem!
  - Möjliggör stor variation
  - God skalbarhet

Här hade jag tänkt göra ett litet exempel... men NVidias drivrutiner stödjer inte brusfunktioner än. De returnerar alltid noll!

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Exempel: Mandelbrot

Procedurell textur

Rekursiv funktion, imaginärt argument med beroende av startvärdet

$$f(a) = a^2 + a_{\text{start}}$$

Inspektera antalet iterationer det tar tills den är utanför en given radie, om någonsin

Lätt att implementera!

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Exempel: Mandelbrot

**MEN! GLSL tillåter inte rekursion!**

**Då gör vi det med for-loop då!**

```
for (iter = 0.0; iter < MaxIterations && r2 < 4.0; ++iter)
{
    tempreal = real;
    real = (tempreal * tempreal) - (imag*imag) + Creal;
    imag = 2.0 * tempreal * imag + Cimag;
    r2 = (real * real) - (imag*imag);
}
```

**Toppen! Eller...?**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Exempel: Mandelbrot

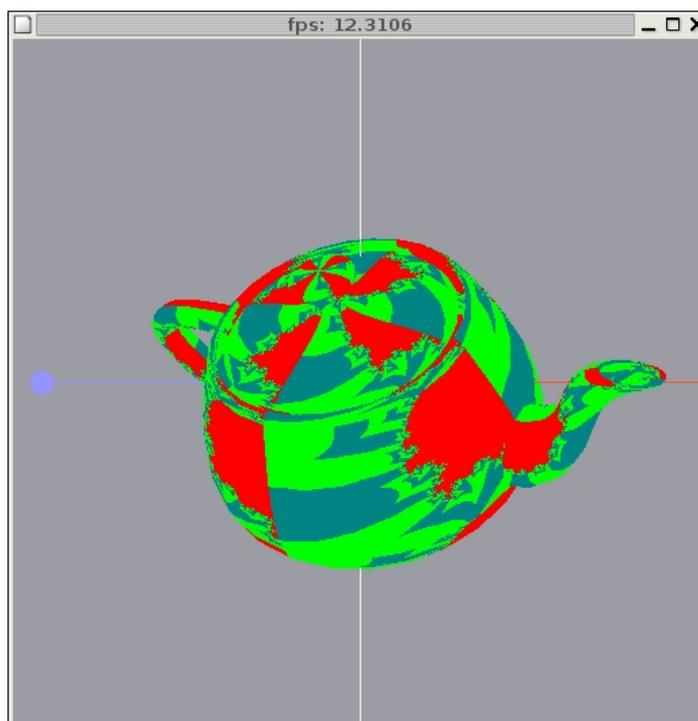
**MEN! GForce 5900 tillåter inte for-loopar!**

**Rulla ut for-loopen!**

```
iter = 0.0;
{
    iter++;
    tempreal = real;
    real = (tempreal * tempreal) - (imag*imag) + Creal;
    imag = 2.0 * tempreal * imag + Cimag;
    r2 = (real * real) - (imag*imag);
}
if (r2 < 4.0)
{
    iter++;
    tempreal = real;
    real = (tempreal * tempreal) - (imag*imag) + Creal;
    imag = 2.0 * tempreal * imag + Cimag;
    r2 = (real * real) - (imag*imag);
}
if (r2 < 4.0)
{
    iter++;
    etc...
```

Ingemar  
Ragnemalm  
ingis@isy.liu.se

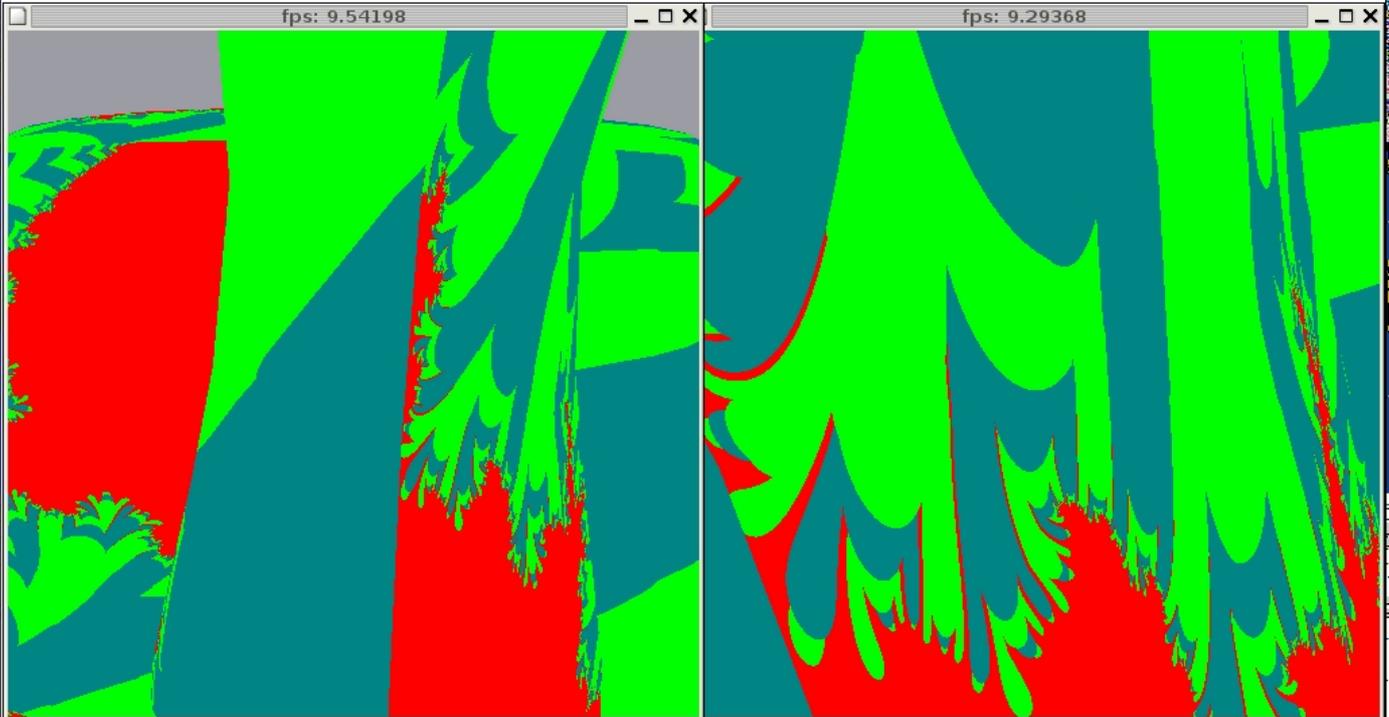
# Exempel: Mandelbrot



Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Exempel: Mandelbrot

Kan zoomas i realtid



## Fördefinierade variabler

Flera olika grupper:

Vertexshader:

Attribut  
Uniform  
Utdata  
Varying

Fragmentshader:

Varying  
Indata  
Utdata  
Konstanter

Inte samma  
som är  
viktigast i  
båda

# Vertexshader, attribute

gl\_Color  
gl\_Normal  
gl\_Vertex  
gl\_FogCoord  
gl\_MultiTexCoord0  
gl\_MultiTexCoord1  
...

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Vertexshader, uniform

(Urval)

gl\_ModelViewMatrix  
gl\_ProjectionMatrix  
gl\_ModelViewProjectionMatrix  
gl\_NormalMatrix  
gl\_TextureMatrix[]  
gl\_NormalScale  
gl\_DepthRange  
gl\_Point  
  
gl\_LightModel  
gl\_TextureEnvColor  
gl\_Fog

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Vertexshader, uniform

Ljuskällor!

`gl_LightSource[gl_MaxLights]`

strukturer som innehåller

```
vec4 ambient;  
vec4 diffuse;  
vec4 specular;  
vec4 position;  
vec4 halfvector;  
(spotlightdata)  
float constantAttenuation;  
float linearAttenuation;  
float quadraticAttenuation;
```

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Vertexshader, utdata

Måste skrivas:

`gl_Position`

Kan skrivas vid behov:

```
gl_PointSize  
gl_ClipVertex
```

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Vertexshader, varying

Inbyggda variabler för att skicka data till  
fragmentshader:

```
gl_FrontColor  
gl_BackColor  
gl_TexCoord[]  
gl_FogFragCoord
```

# Fragmentshader, varying

Inbyggda variabler för data från vertexshader:

```
gl_Color  
gl_TexCoord[]  
gl_FogFragCoord
```

**gl\_color** beräknas från **gl\_FrontColor** eller **gl\_BackColor**, beroende på om man ser fram- eller baksida. Med backface culling bryr vi oss bara om framsidan.

# Fragmentshader, indata

Specialvariabler för vissa indata till  
fragmentshader:

`gl_FragCoord`  
`gl_FrontFacing`

# Fragmentshader, utdata

Specialvariabler för utdata från fragmentshader:

`gl_FragColor`  
`gl_FragDepth`

eller

`discard`

“Discard” är ingen variabel utan en styrinstruktion.  
Betyder att man nekar att skriva fragmentet till  
frame buffern.

# Konstanter

(urval)

gl\_MaxLights (minst 8)  
gl\_MaxClipPlanes (minst 6)  
gl\_MaxTextureUnits (minst 2)  
etc...

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Constructors

## Initiering av variabler samt type casting

“Constructors” precis som i C++ och Java - men det är inte objekt och du kan inte skriva egna konstruktörer!

```
a = int(b);  
b = float(a);  
c = bool(a);
```

Snyggare skrivsätt för type casting än i C! (Tycker jag i alla fall!)

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Constructors för vektorer

```
vec4 color = vec4(1.0, 0.5, 0.0, 1.0);
```

Precis som i C++/Java!

Varianter för att ändra antal komponenter:

```
color = vec4(1.0);
```

sätter alla till 1.0

```
vec3 a = vec3(color);
```

kopierar tre komponenter och skippar den fjärde

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## Vektordata

Komponentvis eller kollektivt

$$a = b + c;$$

eller

$$a.x = b.x + c.x;$$
$$a.y = b.y + c.y;$$
$$a.z = b.z + c.z;$$

Tre synonyma komponentnamnuppsättningar:

x, y, z, w - r, g, b, a - s, t, p, q

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Swizzling

Delar av en vektor kan anges med *swizzling*:

```
v2 = v4.rg;  
v3 = v4.xyz;  
v2 = v4.st;
```

Komponenterna får ändra ordning:

```
v3 = v3.bgr;
```

## Mer vektoroperationer

De flesta vektoroperationer gör det man förvänta sig.

```
vec3 v3 = {1.0, 2.0, 3.0};  
float f = 1.0;
```

```
v3 = v3 + f;
```

```
ger {2.0, 3.0, 4.0}
```

# Mer vektoroperationer

Multiplikation av två vektorer multiplicerar komponentvis (returnerar vektor).

Multiplikation av matriser och vektorer fungerar som väntat.

Skalarprodukt och kryssprodukt är inbyggda funktioner, `dot()` och `cross()`.

`length()` returnerar en vektors norm (längd).

`normalize()` returnerar normerad version av vektor

`reflect()` beräknar speglad vektor