

# GLSL

## OpenGL Shading Language

C-likande språk

- Syntax någonstans mellan C och C++
- Inga klasser. Rak och enkel kod. Anmärkningsvärt begripligt och självklart!
- Undviker de flesta av C/C++ dåliga sidor.

En del av fördelarna kommer från den begränsade miljön. Begränsat problem ger städad lösning!

“Algol”-språk, lätt att lära sig om man kan något av dess många syskon.

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## GLSL

### Exempel

Vertex shader:

```
void main()
{
    gl_Position = gl_ProjectionMatrix *
                  gl_ModelViewMatrix * gl_Vertex;
}
```

“Pass-through shader”, gör bara fixa funktionalitetens minimala operationer

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# GLSL Exempel

Fragment shader:

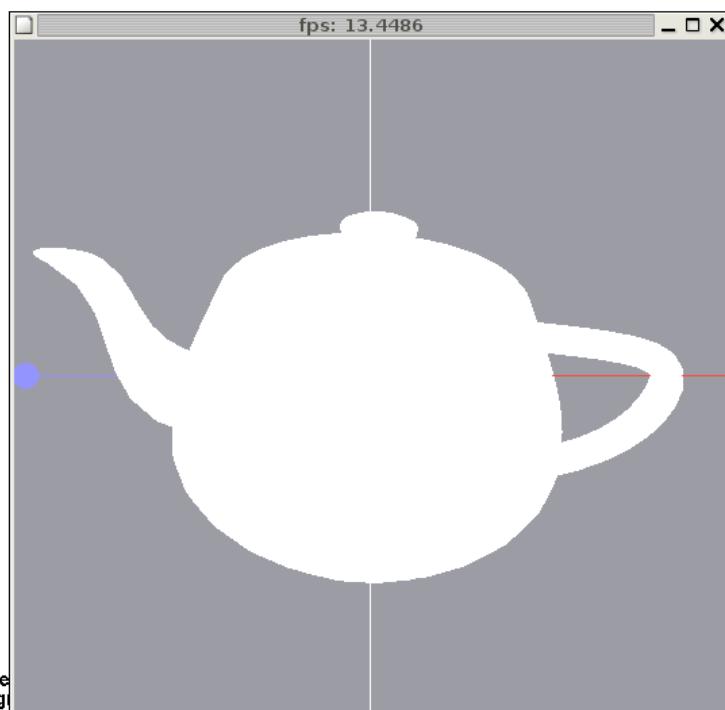
```
void main()
{
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

“Set-to-white shader”

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Exempel

Pass-through vertex shader  
+ set-to-white fragment shader



```
// Vertex shader
void main()
{
    gl_Position = gl_ProjectionMatrix *
                  gl_ModelViewMatrix * gl_Vertex;
}

// Fragment shader
void main()
{
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

# Lägg märke till:

Inbyggda variabler:

<code>gl_Position</code>	transformerad vertex, utdata
<code>gl_ProjectionMatrix</code>	projektionsmatrisen
<code>gl_ModelViewMatrix</code>	modelviewmatrisen
<code>gl_Vertex</code>	vertex i modellkoordinater
<code>gl_FragColor</code>	resulterande fragmentfärg

Inbyggd typ som vi inte sett förr:

`vec4` 4-komponent-vektor

Redan nu börjar vi se vissa möjligheter, eller hur?

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# Lägg också märke till:

Matrismultiplikation med \*-operator

Shaders startar alltid i main()

Kommentar: Följande multiplikation extremt vanlig

```
gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
```

**alias:**

```
gl_ModelViewProjectionMatrix  
samma sak som  
gl_ProjectionMatrix * gl_ModelViewMatrix  
ftransform();
```

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# Grunderna i GLSL

En genomgång av språket (med lite exempel)

- Teckenset
- Preprocessor-direktiv
- Kommentarer
- Identifierare
  - Typer
  - Modifierare
- Constructors
- Operatorer
- Inbyggda funktioner
- Inbyggda variabler
- Aktivera shaders från OpenGL
- Kommunikation med OpenGL

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## Teckenset

Alfanumeriska tecken: a-z, A-Z, \_, 0-9

. + - / \* % < > [ ] { } ^ | & ~ = ! : ; ?

# för preprocessordirektiv (!)

mellanslag, tab, FF, CR, FL

OBS: Tolererar både CR, LF och CRLF! ☺

Case sensitive

MEN

Tecken och strängar finns inte! ‘a’, “Hej” mm

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Preprocessorn

#define #undef #if mm

\_VERSION\_ användbar för versionsskillnader

(Kan nog behövas ibland - tyvärr.)

#include finns inte

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Kommentarer

/\* Detta är en kommentar  
som går över flera rader \*/

// men jag tycker oftast bättre om enradiga

Precis som vanligt! ☺

Så kommentera flitigt!

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Identifierare

Precis som i C: alfanumeriska tecken, ej siffra först.

MEN

Reserverade identifierare, fördefinierade variabler,  
har gl\_ som prefix.

Man får inte själv deklarera variabler med gl\_-  
prefix.

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Typer

Några självklara skalärer:

**void:** returvärde för procedurer

**bool:** Boolesk variabel, flagga

**int:** heltal

**float:** flyttal

**men ingen long eller double.**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Mer typer

Vektorer:

**vec2, vec3, vec4: Flyttal, 2, 3 eller 4 komponenter**

**bvec2, bvec3, bvec4: Booleska vektorer**

**ivec2, ivec3, ivec4: Heltalsvektorer**

Matriser:

**mat2, mat3, mat4: 2x2, 3x3, 4x4 (flyttal)**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

Viktigt!

## Modifierare

Hur variabler skall användas deklarerats med modifierare:

**const**

**attribute**

**uniform**

**varying**

Om dessa saknas så är variabeln “lokal” i sitt scope och kan läsas och skrivas efter behag.

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# **const**

**konstant, sätts vid kompilering och kan inte  
ändras**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# **attribute och uniform**

**attribute är argument från OpenGL, per-vertex-data**

**uniform är argument från OpenGL, per primitiv.  
Kan inte ändras inom ett primitiv**

**Många fördefinierade variabler är “attribute”  
eller “uniform”.**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# **varying**

**Data som skall interpoleras mellan vertexar deklareraras ”varying”, både i fragment och vertex shader.**

**Skrivs i vertex shader.**

**Kan inte skrivas av fragment shader.**

**Exempel: texturkoordinater, normalvektor för Phong shading, vertexfärg, ljusvärde för Gouraud shading**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

## **Exempel: Gouraud shader**

**Nej, det var inte för Gouraud vi ville lära oss shaders, men det är ett enkelt exempel.**

- Transformera normalvektor
- Beräkna shadingvärde per vertex, endast diffuse, genom skalärprodukt med ljusvektor
- Interpolera mellan vertexar

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# Gouraud shader

## Vertex shader

```
varying float shade;  
  
void main()  
{  
    vec3 norm;  
    const vec3 light = {0.58, 0.58, 0.58};  
  
    gl_Position = gl_ProjectionMatrix *  
                 gl_ModelViewMatrix * gl_Vertex;  
    norm = normalize(gl_NormalMatrix * gl_Normal);  
    shade = dot(norm, light);  
    shade = clamp(shade, 0, 1);  
}
```

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Gouraud shader

## Fragment shader

```
varying float shade;  
  
void main()  
{  
    gl_FragColor = vec4(shade);  
}
```

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Gouraud shader

Observera:

Variabeln "shade" är varying, interpoleras mellan vertexar!

Funktionerna dot() och normalize() gör de vektoroperationer man förväntar sig.

Funktionen clamp() håller variabel inom önskat intervall.

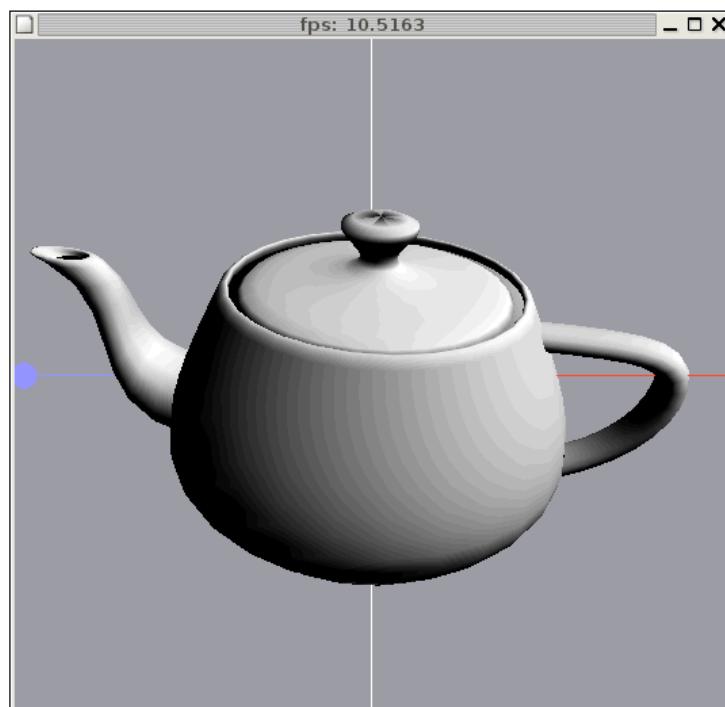
`gl_Normal` är normalvektor i modellkoordinater  
`gl_NormalMatrix` transform för normalvektor

Konstanta vektorn `light()` initieras på känt vis.

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Gouraud shader

Resultat



Notera artefakterna i underkant.

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Exempel: Phong shader

Har jag sagt A får jag säga B...

- Transformera normalvektor
- Interpolera normalvektorer mellan vertexar
- Beräkna shadingvärde per fragment, endast diffuse, genom skalärprodukt med ljusvektor

Så gott som samma operationer, men ljusberäkningen görs i fragment shader

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## Phong shader

Vertex shader

```
varying vec3 norm;  
  
void main()  
{  
    gl_Position = gl_ProjectionMatrix *  
    gl_ModelViewMatrix * gl_Vertex;  
    norm = normalize(gl_NormalMatrix * gl_Normal);  
}
```

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Phong shader

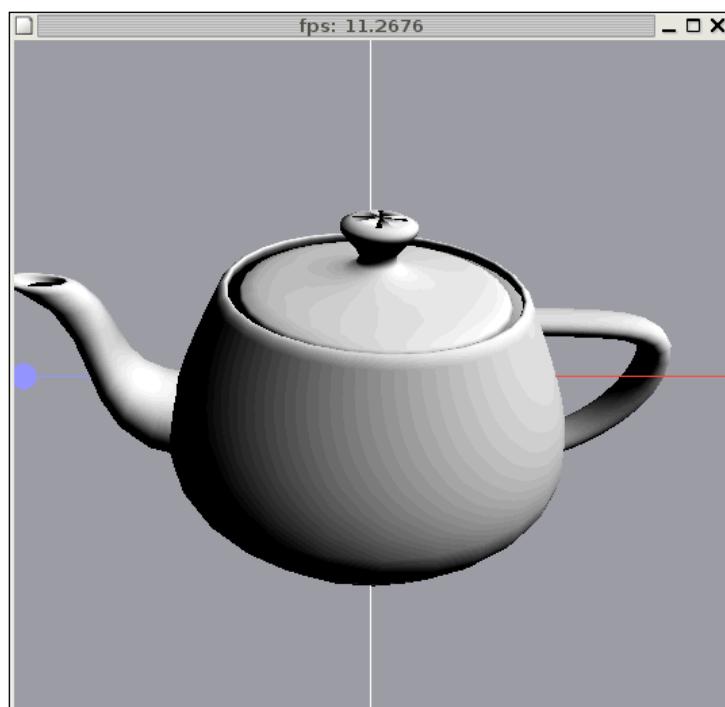
## Fragment shader

```
varying vec3 norm;  
  
void main()  
{  
    float shade;  
    const vec3 light = {0.58, 0.58, 0.58};  
  
    shade = dot(normalize(norm), light);  
    shade = clamp(shade, 0, 1);  
    gl_FragColor = vec4(shade);  
}
```

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Phong shader

## Resultat



Inga artefakter i underkant!

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Texturkoordinater

Inbyggda variabler:

**gl\_MultiTexCoord0** är texturkoordinat till vertex för texturenhet 0.

**gl\_TexCoord[0]** är en inbyggd varying för att interpolera texturkoordinater.

**gl\_TexCoord[0].s** och **gl\_TexCoord[0].t** ger S- och T-komponenten var för sig.

Vi väntar med texturdata så länge...

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## Exempel: Procedurell textur

Textur genererad av fragment shader!

- Vertex shader skickar vidare texturkoordinater
- Texturkoordinater används kreativt i fragment shader

Enklare än man kan tro!

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Procedurell textur

## Vertex shader

```
void main()
{
    gl_Position = gl_ProjectionMatrix *
                  gl_ModelViewMatrix * gl_Vertex;
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

**Bara en “pass-through”-shader plus stöd för texturkoordinater, som också bara skickas vidare!**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Procedurell textur

## Fragment shader

```
void main()
{
    gl_FragColor = vec4(1.0, 1.0, 1.0, 0.0);

    float a = sin(gl_TexCoord[0].s*30)/2+0.5;
    float b = sin(gl_TexCoord[0].t*30)/2+0.5;
    gl_FragColor = vec4(a, b, 1.0, 0.0);
}
```

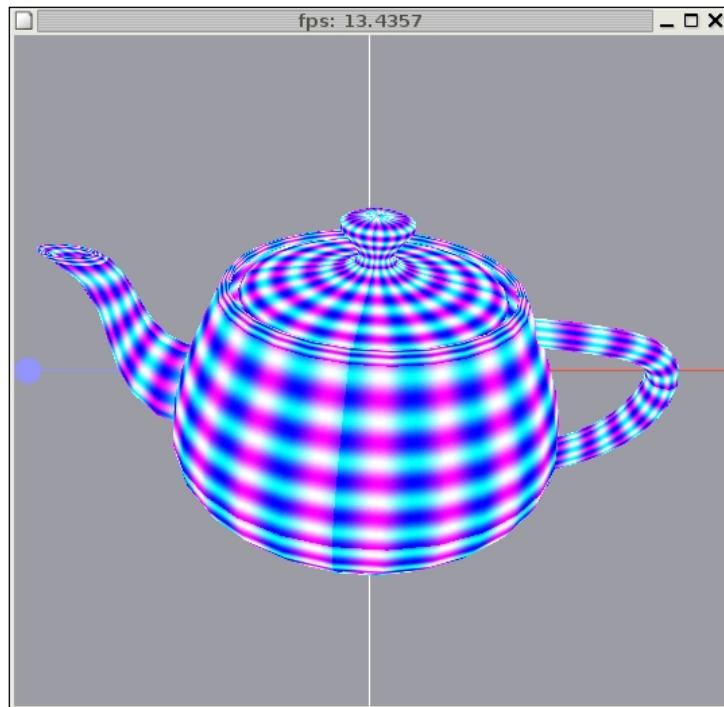
**Behagligt lätt! Fragmentvärdet är en funktion av S och T, i detta fall en enkel sinusvåg på varje.**

**Observera `sin()`, en av många vanliga matematiska funktioner.**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Procedurell textur

## Resultat



Ingemar  
Ragnemalm  
ingis@isy.liu.se

## Texturdata

**För att använda förgenererade  
texturdata måste dessa  
kommuniceras från OpenGL!**

**Detta görs med en “uniform”, en  
variabel som inte kan ändras inom  
ett primitiv.**

**Fördeklarerade typer för  
texturreferens, “samplers”**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Texturdata

## Exempel:

```
uniform sampler2D texture;  
  
void main()  
{  
    gl_FragColor = texture2D(texture,  
                             gl_TexCoord[0].st);  
}
```

**texture2D()** gör texturuppslagning i **texture**.

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Mer typer för texturer

## Texturaccess:

**sampler1D**  
**sampler2D**  
**sampler3D**  
**samplerCube**  
**sampler1DShadow**  
**sampler2DShadow**

**Texturreferenser, ej skrivbara (“uniform”)**

**sampler2D** används för att läsa från vanliga 2D-texturer

**Måste kommuniceras från värdprogrammet.**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Multitexturering

Behagligt enkelt!

Två eller flera texturreferenser sätts upp från värdprogram, deklareras i shaders. Använd efter behag!

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# Mer typer

Pekare finns inte!

Strukturer: “struct” ungefär som i C.

Arrayer: som i C, deklareras med []

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)