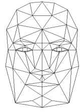


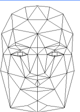
GLSL
OpenGL Shading Language

Language with syntax similar to C

- Syntax somewhere between C och C++
- No classes. Straight ans simple code. Remarkably understandable and obvious!
- Avoids most of the bad things with C/C++.

Some advantages come from the limited environment!

“Algol” descendant, easy to learn if you know any of its followers.

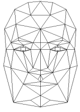


GLSL
Example

Vertex shader:

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

“Pass-through shader”, implements the minimal functionality of the fixed pipeline

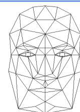


GLSL
Example

Fragment shader:

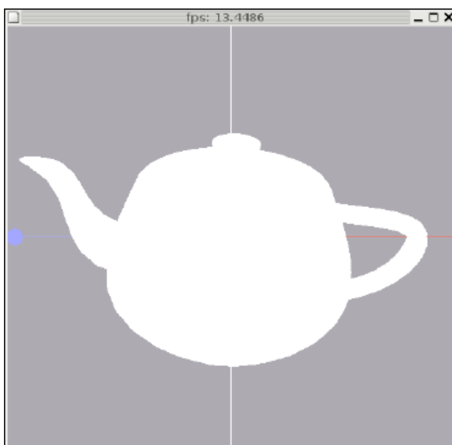
```
void main()
{
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

"Set-to-white shader"



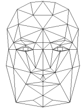
Example

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);
}
```



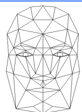
Note:

Built-in variables:

gl_Position	transformed vertex, out data
gl_ProjectionMatrix	projection matrix
gl_ModelViewMatrix	modelview matrix
gl_Vertex	vertex in model coordinates
gl_FragColor	resulting fragment color

Also a new built-in type:
vec4 4 component vektor

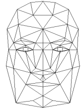
Some possibililties start to show up, right?



GLSL basics

A short tour of the language

- Identifiers
- Types
- Modifiers
- Constructors
- Operators
- Built-in functions and variables
- Activating shaders from OpenGL
- Communication with OpenGL



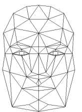
Identifiers

Just like C: alphanumerical characters, first non-digit

BUT

Reserved identifiers, predefined variables, have the prefix gl_!

It is not allowed to declare your own variables with the gl_ prefix!



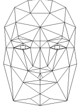
Types

There are some well-known scalar types:

void: return value for procedures
bool: Boolean variable, that is a flag
int: integer value
float: floating-point value

However, no long or double.

Double exists as extension!



More types

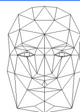
Vector types:

vec2, vec3, vec4: Floating-point vectors with 2, 3 or 4 components

bvec2, bvec3, bvec4: Boolean vectors

ivec2, ivec3, ivec4: Integer vectors

mat2, mat3, mat4: Floating-point matrices of size 2x2, 3x3, 4x4



Important!

Modifiers

Variable usage is declared with modifiers:

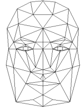
const

attribute

uniform

varying

If none of these are used, the variable is "local" in its scope and can be read and written as you please.

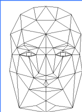


attribute and uniform

attribute is argument from OpenGL, per-vertex-data

uniform is argument from OpenGL, per primitive. Can not be changed within a primitive

Many predefined variables are “attribute” or “uniform”.



varying

data that should be interpolated between vertices

Written in vertex shader

Read (only) by fragment shaders

In both shaders they must be declared “varying”. In the fragment shader, they are read only.

Examples: texture coordinates, normal vectors for Phong shading, vertex color, light value for Gouraud shading

Example:
Gouraud shader

No, we didn’t learn shaders to do Gouraud shading, but it is a simple example

- **Transform normal vectors**
- **Calculate shading value per vertex, (here using diffuse only), by dot product with light direction**
- **Interpolate between vertices**

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);
}
```

Gouraud shader

Fragment shader

```
varying float shade;

void main()
{
    gl_FragColor = vec4(clamp(shade, 0, 1));
}
```

Gouraud shader

Note:

The variable "shade" is varying, interpolated between vertices!

dot() och normalize() do what you expect.

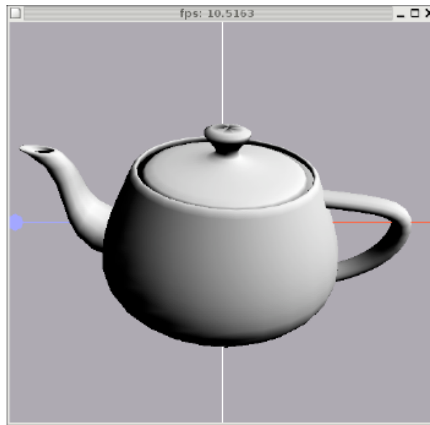
clamp() clamps a variable within a desired interval.

gl_Normal is the normal vector in model coordinates
gl_NormalMatrix transform for normal vectors

The contstant vector light() is here hard coded

Gouraud shader

Result



Very good - for this model

Texture coordinates

Built-in variables:

gl_MultiTexCoord0 is texture coordinate for vertex for texture unit 0.

gl_TexCoord[0] is a built-in varying for interpolating texture coordinates.

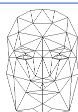
gl_TexCoord[0].s and gl_TexCoord[0].t give the S and T components separately.

Texture data

In order to use predefined texture data, they should be communicated from OpenGL!

This is done by a “uniform”, a variable that can not be changed within a primitive.

“samplers”: pre-defined type for referencing texture data



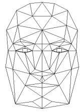
Texture access

Example:

```
uniform sampler2D texture;

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

texture2D() performs texture access

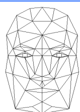


Communication with host

Important! The host must be able to set uniform and attribute variables for GLSL to read.

GLSL can only output information through fragments.

OpenGL sends address and names to GLSL with special calls.



Example: uniform float:

```
float  myFloat;
GLint  loc;

loc  =  glGetUniformLocation(p,  “myFloat”);
glUniform1f(loc,  myFloat);
```

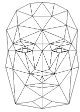
p: Ref to shader program, as installed earlier,

loc: address to variable

Now the variable can be used in GLSL:

```
uniform  float  myFloat;
```

Note that the string passed to glGetUniformLocation specifies the name in GLSL!



Example: texture, uniform sampler:

```
GLuint tex;

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tex);
loc = glGetUniformLocation(PROG, "tex");
glUniform1i(loc, 0);
```

zero to glUniform1i = texture unit number!

Use in shader:

```
uniform sampler2D tex;

vec3 texval = vec3(texture2D(texture, gl_TexCoord[0].st));
```

Information Coding / Computer Graphics, ISY, LiTH

Example: Multitexturing

Bind one texture per texturing unit
Pass GLSL unit number and name
Declare as samplers in GLSL

Many possibilities:

- Combine texture data using arbitrary function.
- Make one texture sensitive to lighting and another not.
- Use texture as bump map

My simple example: Select different texture depending of light level.

Example: Multitexturing

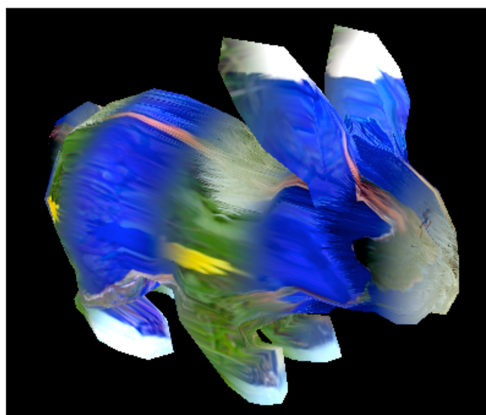
```
uniform sampler2D worldTex;
uniform sampler2D flowerTex;

void main()
{
  gl_FragColor = texture2D(flowerTex, gl_TexCoord[0].st); // Single tex

  gl_FragColor = sin(pixelPos.z*10.0) * texture2D(flowerTex,
          gl_TexCoord[0].st) + (1.0 - sin(pixelPos.z*10.0)) *
          texture2D(worldTex, gl_TexCoord[0].st); // Multi
}
```

Example: Multitexturing

Combines two textures



Compilation and execution

Done in two steps:

1) Initialization, compilation

- **Create a "program object"**
- **Create a "shader object" and pass source code to it**
- **Compile the shader programs**

2) Activation

- **Activate the program object for rendering**

The entire initialization in code

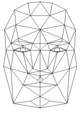
```
PROG = glCreateProgram();

VERT = glCreateShader(GL_VERTEX_SHADER);
text = readTextFile("shader.vert");
glShaderSource(VERT, 1, text, NULL);
glCompileShader(VERT);
```

Same for fragment shader

```
glAttachShader(PROG, VERT);
glAttachShader(PROG, FRAG);

glLinkProgram(PROG);
```



Activate the program for rendering

Givet ett installerat och kompilerat programobjekt:

```
extern GLuint PROG; // Was GLhandleARB
```

activate:

```
glUseProgram(PROG);
```

deactivate:

```
glUseProgram(0);
```