Introduction to shaders using GLSL

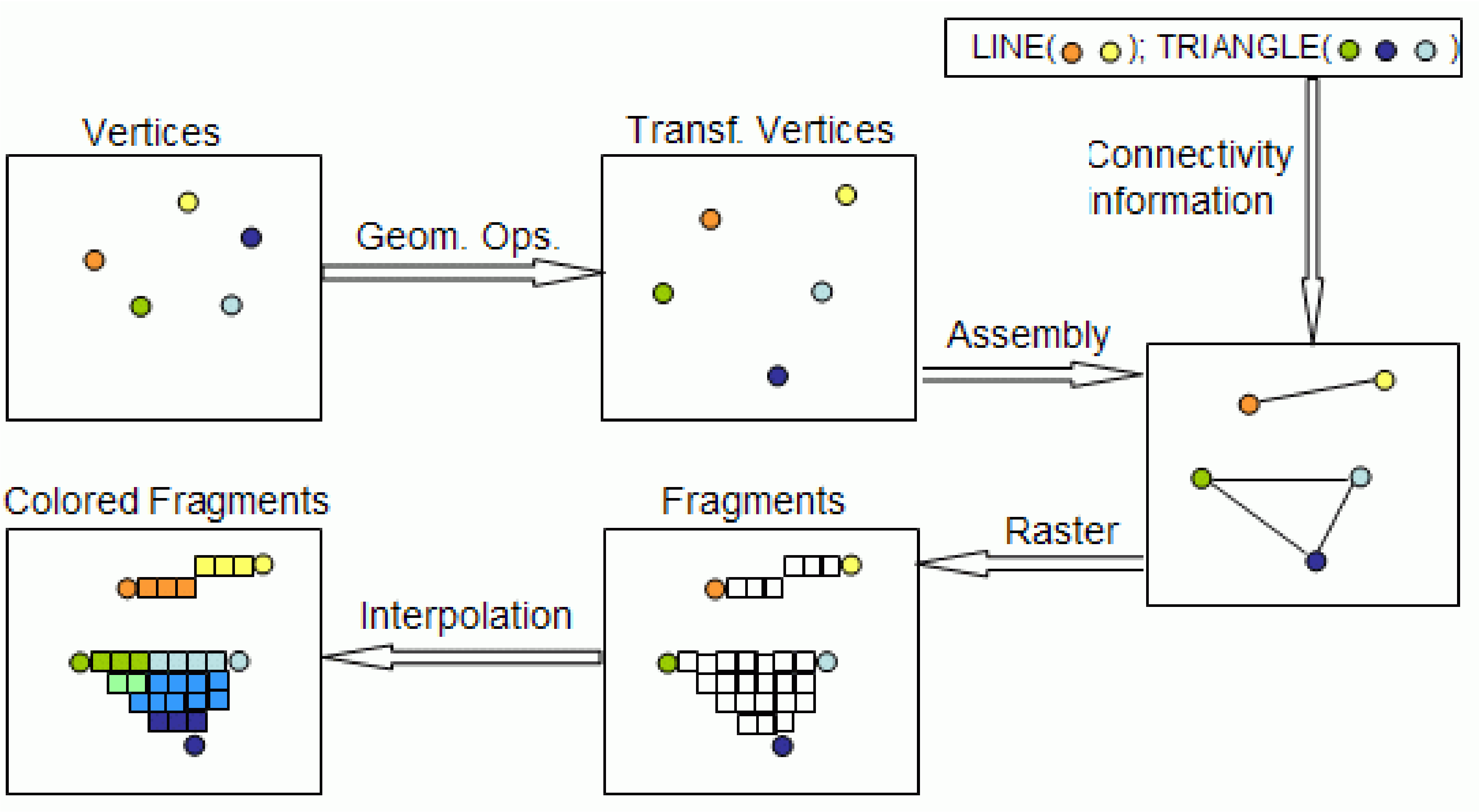
Rui Rodrigues
rui.rodrigues@fe.up.pt

v1.0, 10/2012

Outline

- Graphics pipeline
- Shader types
- Common shading languages
- GLSL details
  - Data types
  - Special variable declarations
  - Swizzling
- Passing values
  - From App to Shaders
  - From Vertex Shader to Fragment Shader
- Working with textures

Graphics pipeline: visual representation



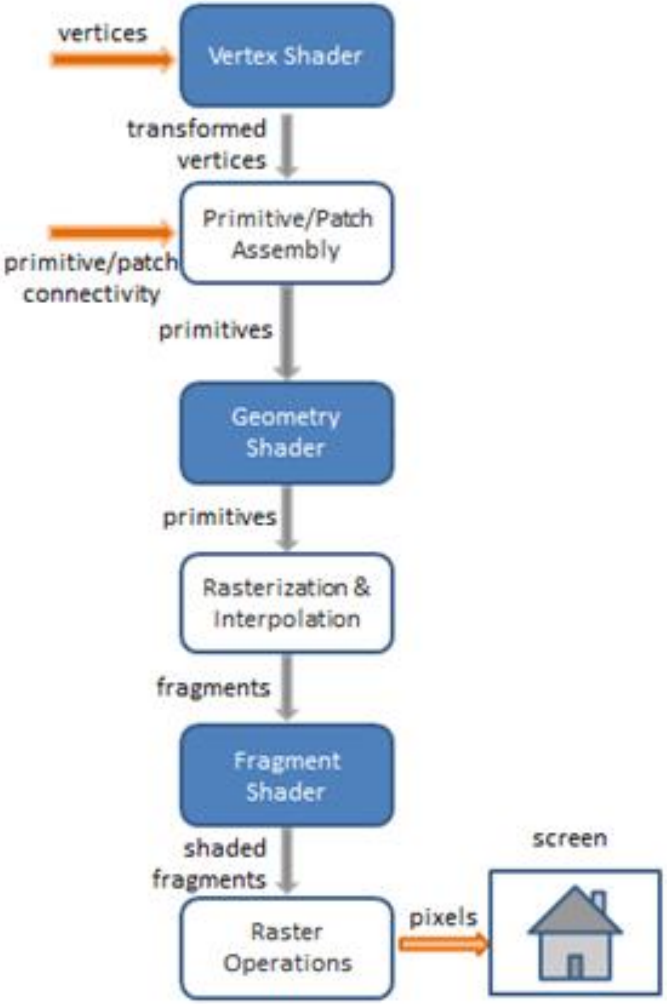
OpenGL pipeline visual representation [GLSL12Tut11]

Graphics pipeline: simplified block diagram

- Inputs

  (vertices, triangles, textures, matrices, etc.)
- **Vertex shading**
- Primitive assembly, culling and clipping
- **Geometry shading** (optional)
- Projection and rasterization
- **Fragment shading**

  (may output to multiple render targets)
- Depth, Stencyl and Alpha-blend (raster) operations
- Output to screen



OpenGL simplified pipeline
(Adapted from [GLSLTut11])

Shaders

- Small programs that replace the fixed functionality of some stages
  - Vertex shaders (VS)
    - Manipulate and define per-vertex properties (coordinates, color, normals)
  - Geometry shaders (GS) (less used)
    - Manipulate and define per-primitive properties (connectivity)
    - May generate new primitives
  - Fragment shaders (FS)
    - Manipulate and define per-fragment (pixel or sample) properties - typically color and transparency
  - Other (e.g. tesselation shaders)

Common shading languages

- OpenGL’s GLSL (our focus)
- Microsoft’s HLSL
- Nvidia’s CG
- Other (earlier)
    - RenderMan
    - OpenGL ISL

GLSL

- C-like language
- Shaders can be loaded as text strings and are compiled in runtime
    - Meaning they can also be changed in runtime
- Tightly coupled with OpenGL
    - Shaders have direct access to most of OpenGL state
- Values/variables can be passed from application to shaders
- Values can be output from the vertex shader and interpolated to the fragment shader
    - (e.g. Vertex's color interpolated over fragment)

First example (1/3): vertex shader

(Vertex shaders will be surrounded by dotted lines)

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

- The basic implementation of the vertex transformation as implemented in the fixed pipeline
- It is applied to every vertex (while this shader is active)
- It outputs a vertex’s position in eye space as the result of multiplying...
    - the vertex coordinates (e.g. Defined by glVertex() calls in code)
    - ...by the OpenGL’s model-view matrix...
    - ...followed by OpenGL’s Projection matrix

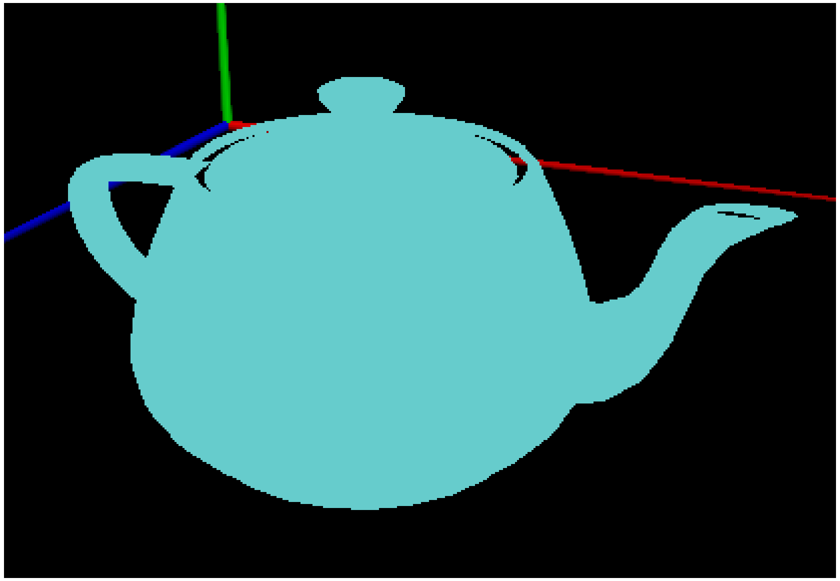
First example (2/3): fragment shader (FS)

(Fragment shaders will be surrounded by dashed lines)

```
void main()
{
        gl_FragColor = vec4(0.5,0.0,0.0, 1.0) * gl_LightSource[0].diffuse;
}
```

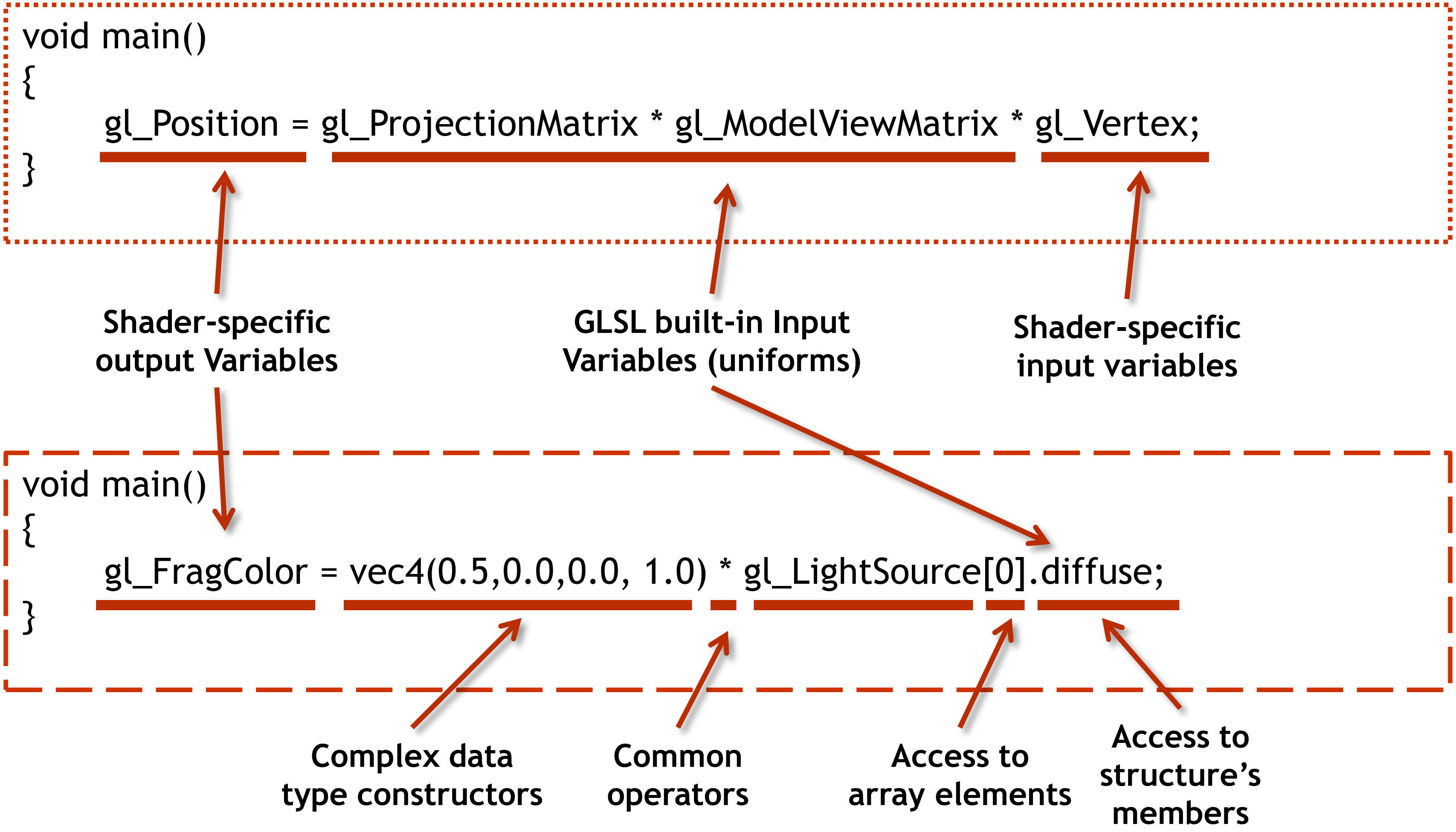
- A simple shader that sets the current fragment’s color based on the diffuse component of a light source

First example (3/3): sample output



- Notice that this gives a solid colored surface, as we set every fragment to the same color
- IMPORTANT: When shaders are active, normal shading is disabled, so if local illumination is desired, it must be computed explicitly in the shader.

Some elements to notice



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

Shader-specific output Variables

GLSL built-in Input Variables (uniforms)

Shader-specific input variables

```
void main()
{
	gl_FragColor = vec4(0.5,0.0,0.0, 1.0) * gl_LightSource[0].diffuse;
}
```

Complex data type constructors

Common operators

Access to array elements

Access to structure’s members

What can be used in shaders?

- OpenGL’s built-in information and data structures such as
  - vertex, normal and color information
  - transformation matrices,
  - light sources and parameters,
  - material parameters, etc.
- Parameters in any of the supported data types
  - passed from the application to the shaders, and between shaders
- A series of built-in functions, including
  - trigonometry and other geometry-related functions,
  - matrix and vector calculus,
  - texture sampling and noise generation
- Multiple textures
  - can be used not only for color modulation, but also for passing information structured as arrays
- User-defined functions and structures, arrays

Data types

- float, vec2, vec3, vec4
  - Individual float values, and vectors of 2, 3 or 4 float components
- int, ivec2, ivec3, ivec4
  - Individual integer values, and vectors of 2, 3 or 4 integer components
- bool, bvec2, bvec3, bvec4
  - Individual boolean values, and vectors of 2, 3 or 4 boolean components
- mat2, mat3, mat4
  - Square matrices of dimensions 2x2, 3x3, or 4x4
- void
  - Used for functions with no return value
- sampler1D, sampler2D, sampler3D
  - Used to sample points on a texture map of 1, 2 or 3 dimensions
- Other samplers

Swizzling

- Accessing one or more vector components in any order

```
myColor.rgb=vec3(1.0,0.0,0.0);
myPos.xz=vec2(10.0,5.0);
myTexCoord.st=myPos.zx;
myVec4=vec4(myPos.xyz,1.0);
```

- Three possible sets (cannot be mixed)

  xyzw (for coordinates)

  rgba (for colors)

  stpq (for texture coordinates)

Global variable declarations

- uniform
  - input to Vertex and Fragment shader from OpenGL or application (RO)
- attribute
  - input per-vertex to Vertex shader from OpenGL or application (RO)
- varying
  - output from Vertex shader (RW), interpolated, then input to Fragment shader (RO)
- const
  - compile-time constant (READ-ONLY)

Function parameter declaration

- In (default)
  - value initialized on entry, not copied on return
- out
  - copied out on return, but not initialized
- inout
  - value initialized on entry, and copied out on return
- const
  - constant function input

Vertex shader input attributes (RO)

- Coming from OpenGL commands
  - vec4 gl_Vertex
  - vec3 gl_Normal
  - vec4 gl_Color
  - vec4 gl_MultiTexCoord0.. gl_MultiTexCoord7
  - ...

Vertex shader output variables

- Special (RW)
  - vec4 gl_Position
    - must be written by VS, it is the vertex position in eye space
  - Other
- Varying (RW)
  - vec4 gl_FrontColor;
  - vec4 gl_BackColor;
  - vec4 gl_FrontSecondaryColor;
  - vec4 gl_BackSecondaryColor;
  - vec4 gl_TexCoord[ ];
  - float gl_FogFragCoord;

Fragment shader inputs

- Special Input Variables (RO)
  - vec4 gl_FragCoord;
  - bool gl_FrontFacing;
- Varying Inputs (RO)
  - varying vec4 gl_Color;
  - varying vec4 gl_SecondaryColor;
  - varying vec4 gl_TexCoord[ ];
  - varying float gl_FogFragCoord;

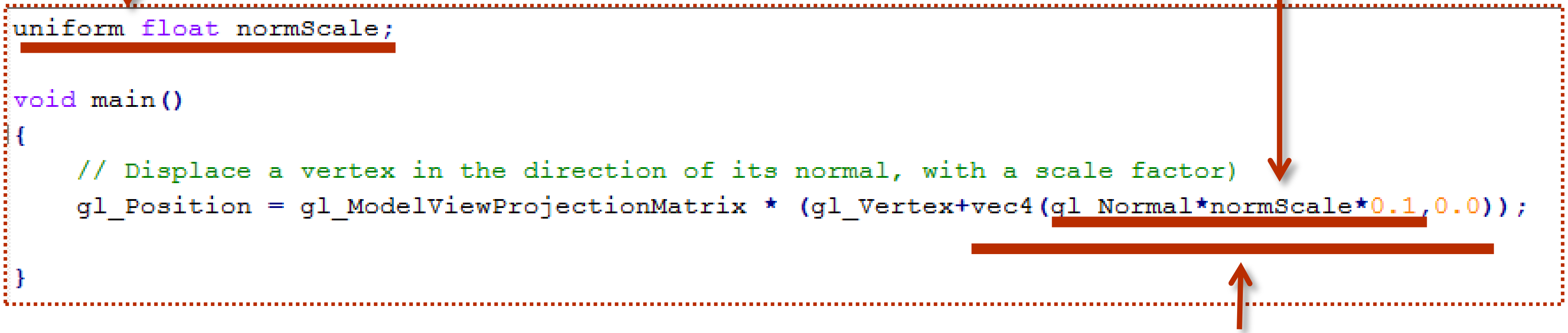
Fragment shader output variables

- Special (RW)
  - vec4 gl_FragColor;
  - vec4 gl_FragData[];
  - float gl_FragDepth;

Passing values: from app to shaders (1/3)

Uniform declaration

Used as a variable



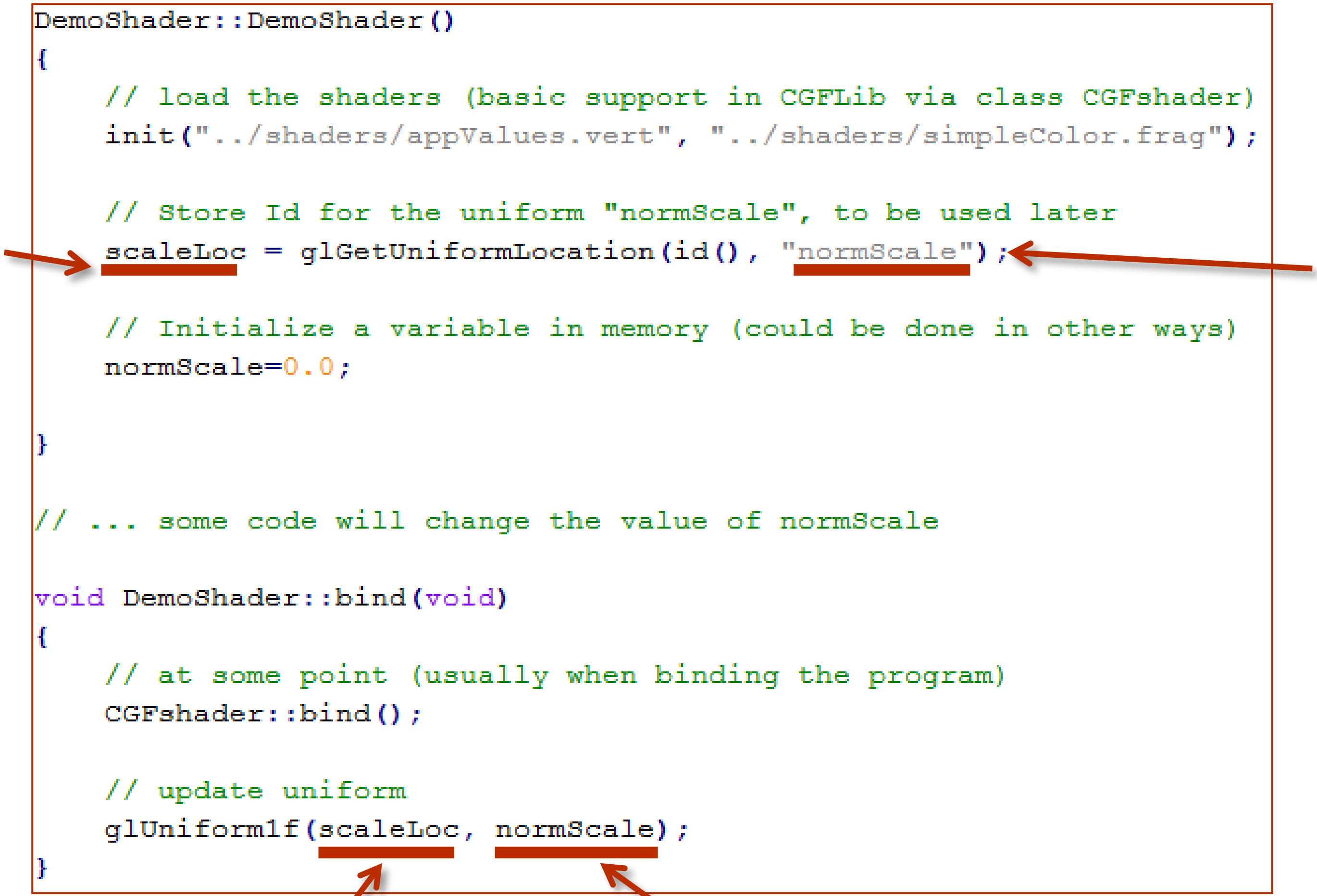
```
uniform float normScale;

void main()
{
    // Displace a vertex in the direction of its normal, with a scale factor)
    gl_Position = gl_ModelViewProjectionMatrix * (gl_Vertex+vec4(gl_Normal*normScale*0.1,0.0));

}
```

Notice building a vec4 using a vec3 plus a fourth component

- This shader displaces a vertex by adding a vector that has the direction of the vertex's normal, and a scale controlled by a parameter, *normScale*
- The parameter value can be controlled in the application

Passing values: from app to shaders (2/3)

Store Reference to the uniform



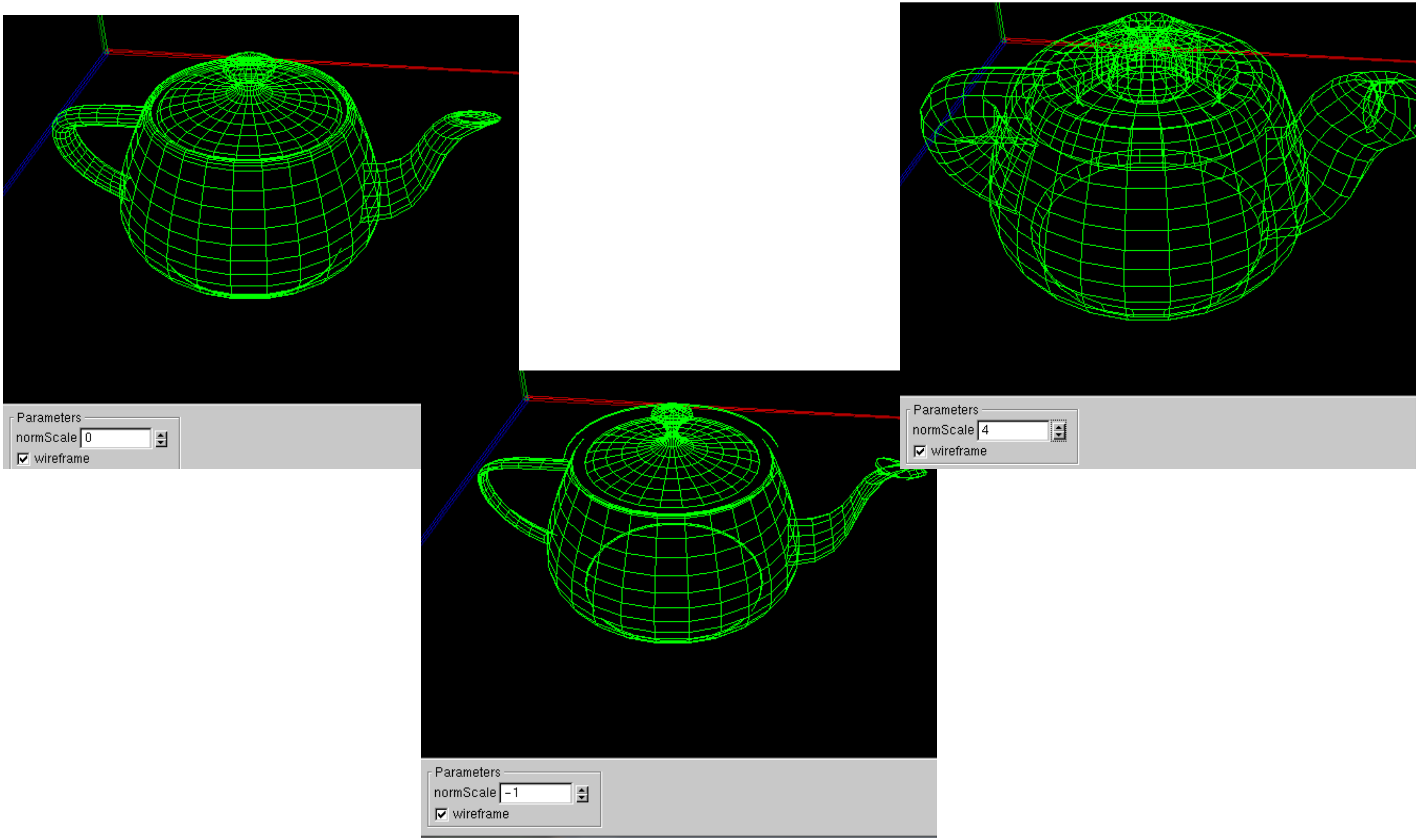
```
DemoShader::DemoShader()
{
    // load the shaders (basic support in CGFLib via class CGFshader)
    init("../shaders/appValues.vert", "../shaders/simpleColor.frag");

    // Store Id for the uniform "normScale", to be used later
    scaleLoc = glGetUniformLocation(id(), "normScale");

    // Initialize a variable in memory (could be done in other ways)
    normScale=0.0;

}

// ... some code will change the value of normScale

void DemoShader::bind(void)
{
    // at some point (usually when binding the program)
    CGFshader::bind();

    // update uniform
    glUniform1f(scaleLoc, normScale);
}
```

String identifying the uniform in the shader

Use stored reference

Provide new value

Passing values: from app to shaders (3/3)



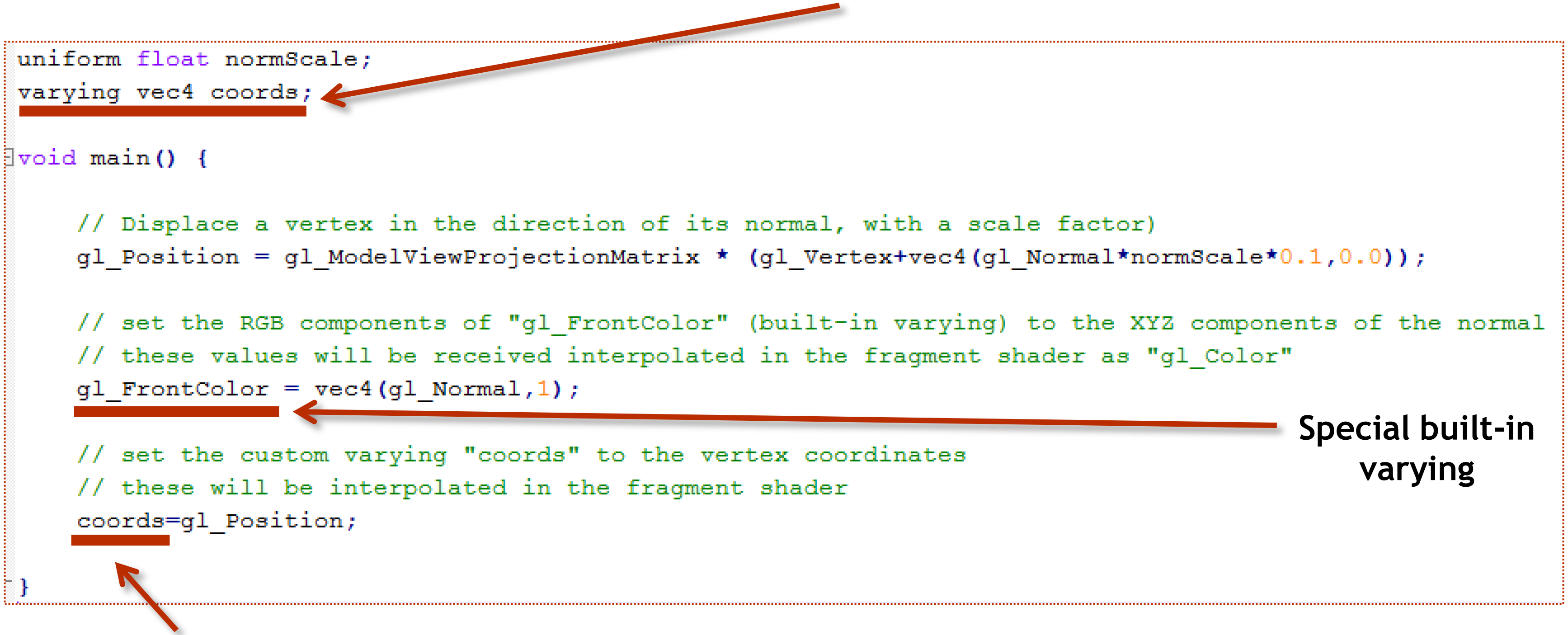
Parameters
normScale 0
☑ wireframe

Parameters
normScale -1
☑ wireframe

Parameters
normScale 4
☑ wireframe

Passing values: from VS to FS (1/3)

Declaration of user-defined varying



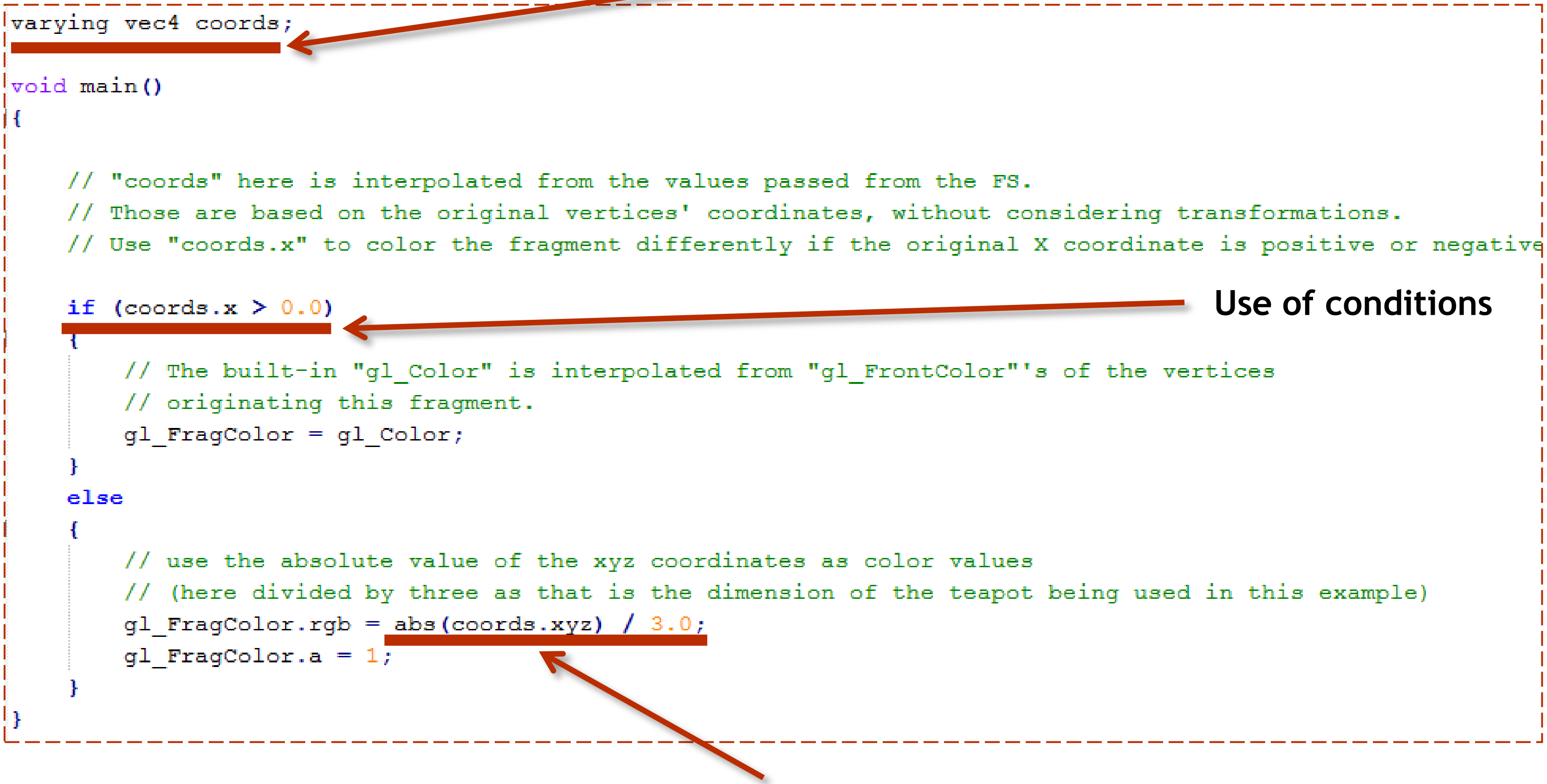
```
uniform float normScale;
varying vec4 coords;

void main() {

    // Displace a vertex in the direction of its normal, with a scale factor)
    gl_Position = gl_ModelViewProjectionMatrix * (gl_Vertex+vec4(gl_Normal*normScale*0.1,0.0));

    // set the RGB components of "gl_FrontColor" (built-in varying) to the XYZ components of the normal
    // these values will be received interpolated in the fragment shader as "gl_Color"
    gl_FrontColor = vec4(gl_Normal,1);

    // set the custom varying "coords" to the vertex coordinates
    // these will be interpolated in the fragment shader
    coords=gl_Position;

}
```

Special built-in varying

Usage of user-defined varying

Passing values: from VS to FS (2/3)

Declaration of user-defined varying

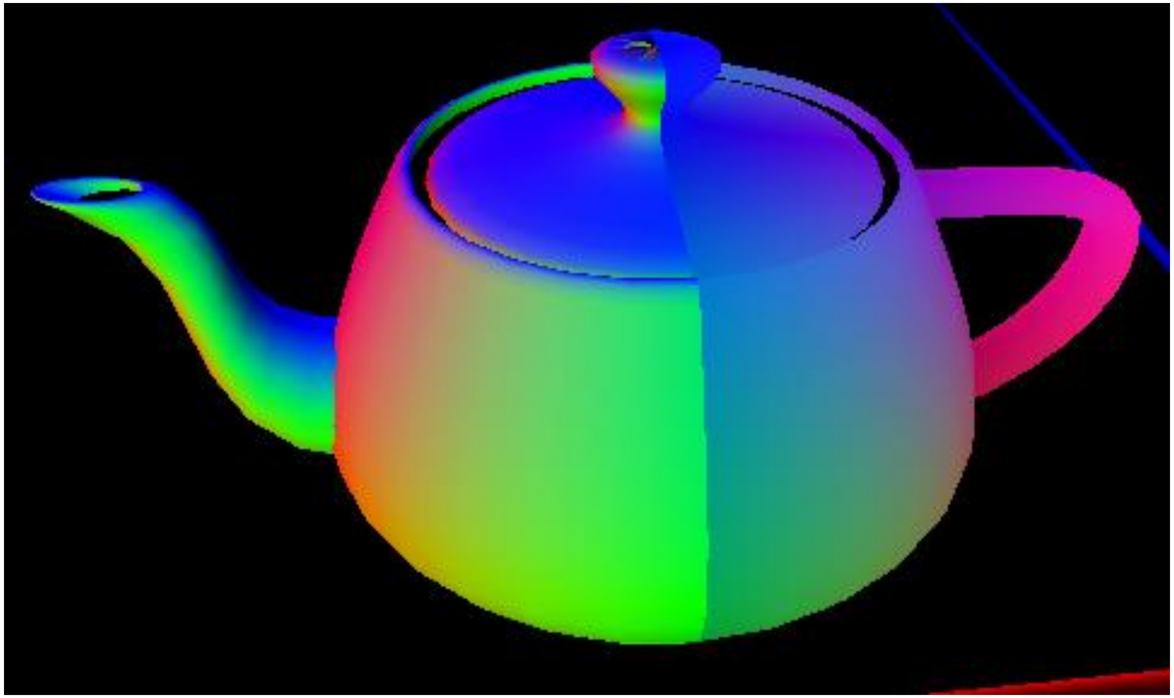


```
varying vec4 coords;

void main()
{

    // "coords" here is interpolated from the values passed from the FS.
    // Those are based on the original vertices' coordinates, without considering transformations.
    // Use "coords.x" to color the fragment differently if the original X coordinate is positive or negative

    if (coords.x > 0.0)
    {
        // The built-in "gl_Color" is interpolated from "gl_FrontColor"'s of the vertices
        // originating this fragment.
        gl_FragColor = gl_Color;
    }
    else
    {
        // use the absolute value of the xyz coordinates as color values
        // (here divided by three as that is the dimension of the teapot being used in this example)
        gl_FragColor.rgb = abs(coords.xyz) / 3.0;
        gl_FragColor.a = 1;
    }
}
```

Use of conditions

Built-in functions and swizzling

Passing values: from VS to FS (3/3)

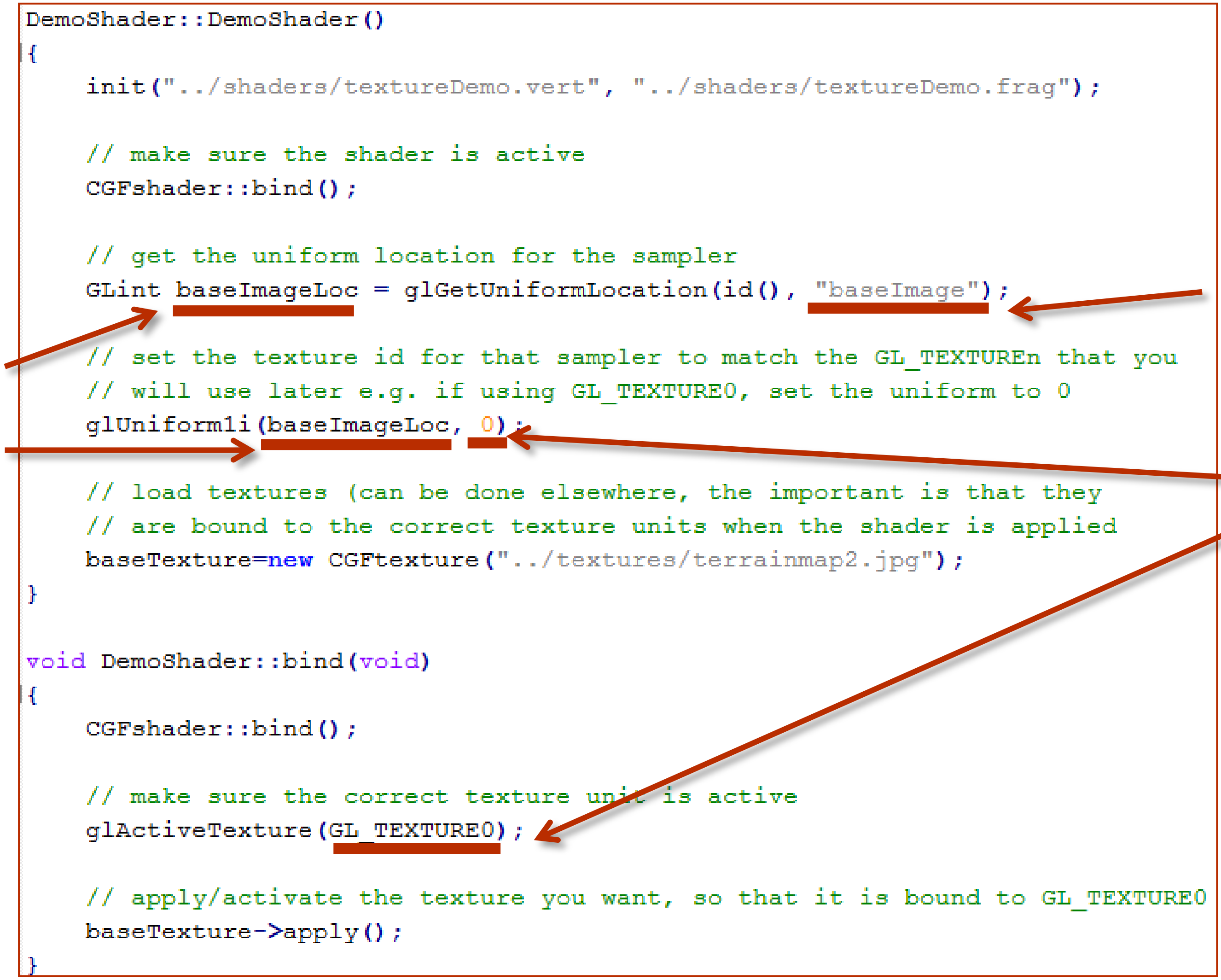


- The left half has color varying depending on the surface orientation (as it is based on the normals)
- The right half has color varying depending on their vertical and horizontal position

Working with textures (1/7)

- Textures are referenced as uniforms of type *int*, in which the uniform's value defines the texture unit to be used
  - A uniform sampler2D assigned with the value 0 gets linked to GL_TEXTURE0
- The steps to work with a texture are
  - Create the uniform sampler in the shader(s)
  - Get the uniform location in the app, and set it to a texture number (typically 0)
  - When binding the shader, make sure that you bind a texture to GL_TEXTURE0

Working with textures (2/7)



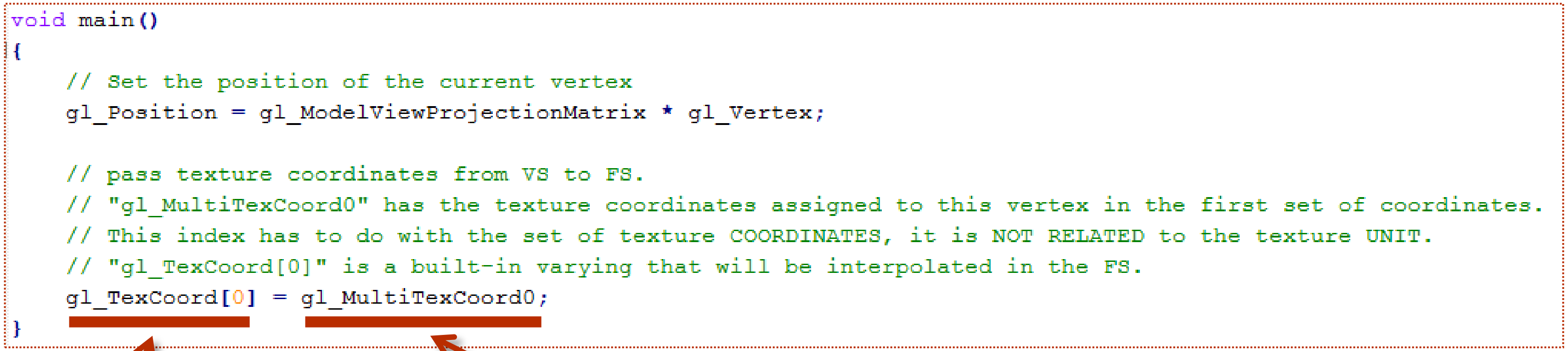
```
DemoShader::DemoShader()
{
    init("../shaders/textureDemo.vert", "../shaders/textureDemo.frag");

    // make sure the shader is active
    CGFshader::bind();

    // get the uniform location for the sampler
    GLint baseImageLoc = glGetUniformLocation(id(), "baseImage");

    // set the texture id for that sampler to match the GL_TEXTUREn that you
    // will use later e.g. if using GL_TEXTURE0, set the uniform to 0
    glUniform1i(baseImageLoc, 0);

    // load textures (can be done elsewhere, the important is that they
    // are bound to the correct texture units when the shader is applied
    baseTexture=new CGFtexture("../textures/terrainmap2.jpg");
}

void DemoShader::bind(void)
{
    CGFshader::bind();

    // make sure the correct texture unit is active
    glActiveTexture(GL_TEXTURE0);

    // apply/activate the texture you want, so that it is bound to GL_TEXTURE0
    baseTexture->apply();
}
```

Sampler name
Used on shaders

Location
of the
uniform

Reference to
Texture unit

Working with textures (3/7)

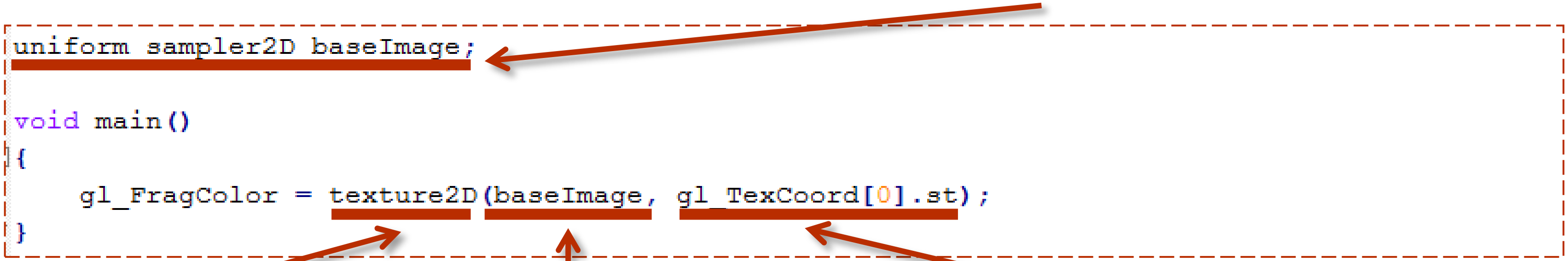


```
void main()
{
    // Set the position of the current vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // pass texture coordinates from VS to FS.
    // "gl_MultiTexCoord0" has the texture coordinates assigned to this vertex in the first set of coordinates.
    // This index has to do with the set of texture COORDINATES, it is NOT RELATED to the texture UNIT.
    // "gl_TexCoord[0]" is a built-in varying that will be interpolated in the FS.
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Tex-coords output from VS to be input to FS

Tex-coords input to VS

Sampler declaration



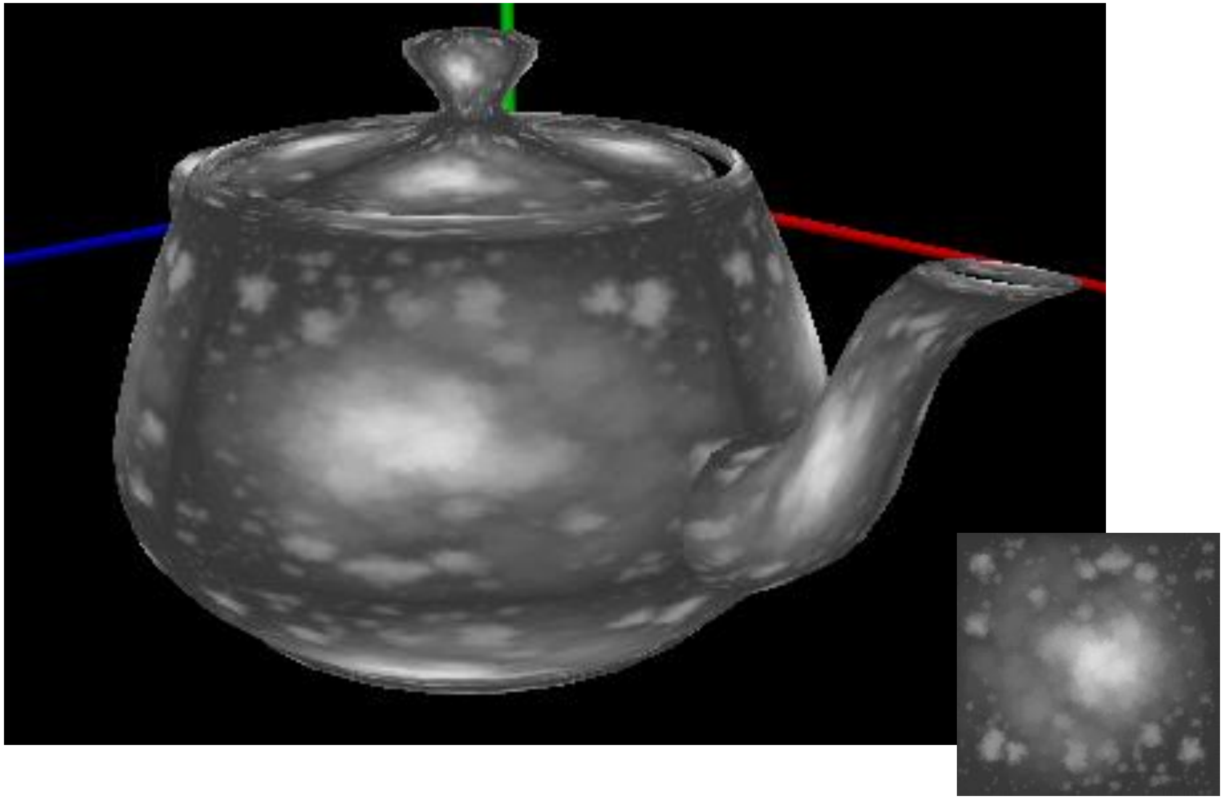
```
uniform sampler2D baseImage;

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

Built-in function returning texel

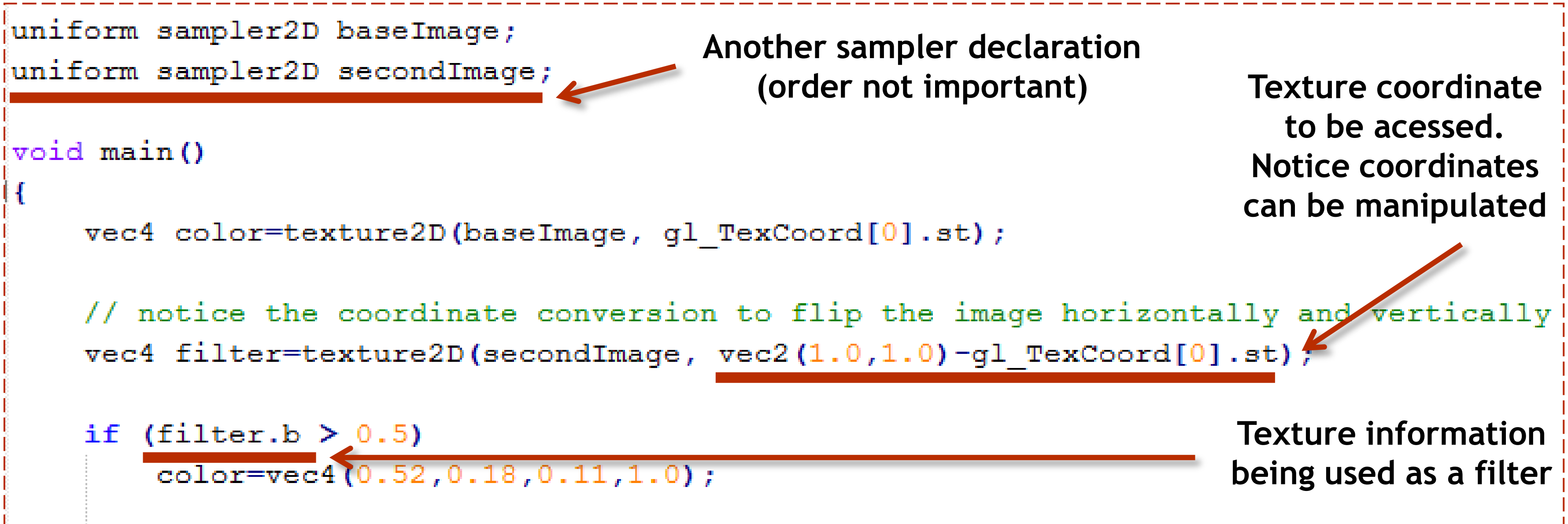
Sampler to be accessed

Texture coordinate to be acessed.
Notice swizzling to use only 2D coords

Working with textures (4/7)



Working with textures (5/7)

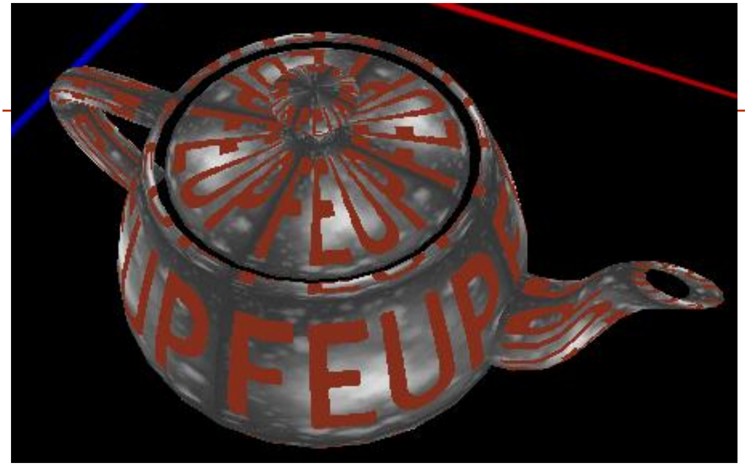


```
uniform sampler2D baseImage;
uniform sampler2D secondImage;

void main()
{
    vec4 color=texture2D(baseImage, gl_TexCoord[0].st);

    // notice the coordinate conversion to flip the image horizontally and vertically
    vec4 filter=texture2D(secondImage, vec2(1.0,1.0)-gl_TexCoord[0].st);

    if (filter.b > 0.5)
        color=vec4(0.52,0.18,0.11,1.0);

    gl_FragColor = color;
}
```

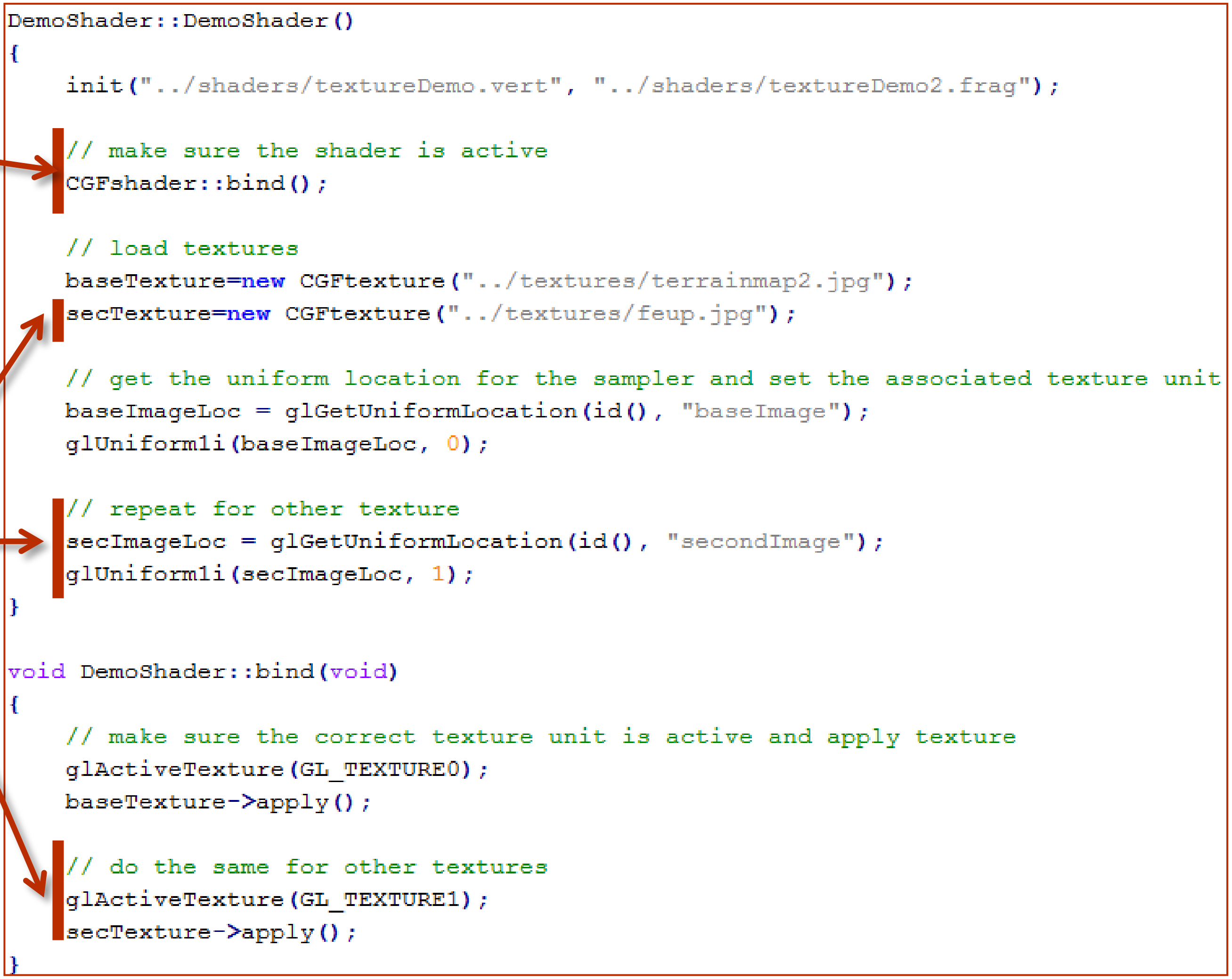
Another sampler declaration (order not important)

Texture coordinate to be acessed. Notice coordinates can be manipulated

Texture information being used as a filter



Working with textures (6/7)

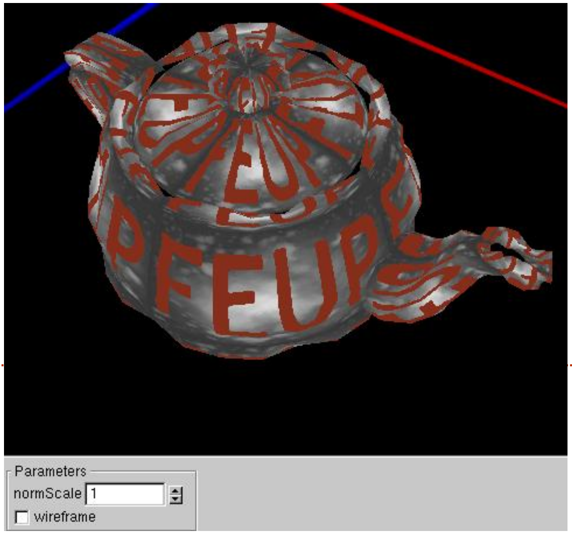


```
DemoShader::DemoShader()
{
    init("../shaders/textureDemo.vert", "../shaders/textureDemo2.frag");

    // make sure the shader is active
    CGFshader::bind();

    // load textures
    baseTexture=new CGFtexture("../textures/terrainmap2.jpg");
    secTexture=new CGFtexture("../textures/feup.jpg");

    // get the uniform location for the sampler and set the associated texture unit
    baseImageLoc = glGetUniformLocation(id(), "baseImage");
    glUniform1i(baseImageLoc, 0);

    // repeat for other texture
    secImageLoc = glGetUniformLocation(id(), "secondImage");
    glUniform1i(secImageLoc, 1);
}

void DemoShader::bind(void)
{
    // make sure the correct texture unit is active and apply texture
    glActiveTexture(GL_TEXTURE0);
    baseTexture->apply();

    // do the same for other textures
    glActiveTexture(GL_TEXTURE1);
    secTexture->apply();
}
```

Important to ensure context

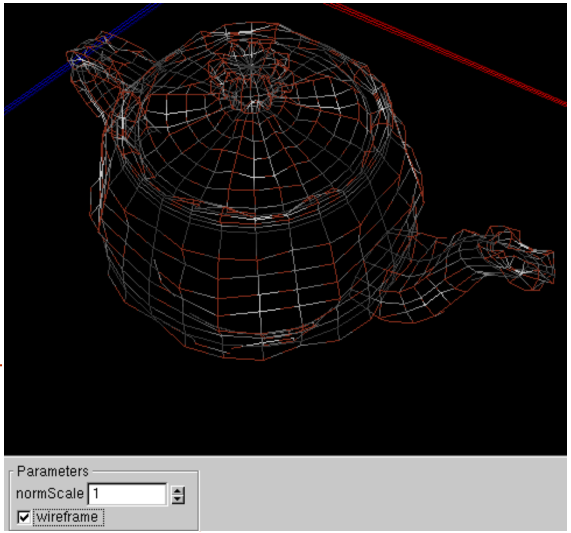
Required additions

Working with textures (7/7)


Parameters

normScale 1

☐ wireframe


Parameters

normScale 1

☑ wireframe

Samplers can also be used in vertex shader

```
uniform float normScale;
uniform sampler2D secondImage;

void main()
{
    vec4 offset=vec4(0.0,0.0,0.0,0.0);

    // change vertex offset based on texture information
    if (texture2D(secondImage, vec2(1.0,1.0)-gl_MultiTexCoord0.st).b > 0.5)
        offset.xyz=gl_Normal*normScale*0.1;

    // Set the position of the current vertex
    gl_Position = gl_ModelViewProjectionMatrix * (gl_Vertex+offset);

    // pass texture coordinates from VS to FS.
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Sampler being used as a filter to change geometry

References

[GLSL12Tut11] GLSL 1.2 Tutorial, António Ramires Fernandes, http://www.lighthouse3d.com/tutorials/glsl-tutorial/ , Lighthouse3D tutorials (accessed October 2012)

[GLSLCTut11] GLSL Core Tutorial, António Ramires Fernandes, http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/, Lighthouse3D tutorials (accessed October 2012)

[GLSLRC05] GLSL Reference Card, Michael E. Weiblen, http://mew.cx/glsl_quickref.pdf (accessed October 2012)

[GLSLSpec12] GLSL Specification, Khronos Group, http://www.opengl.org/documentation/glsl/ (accessed October 2012)