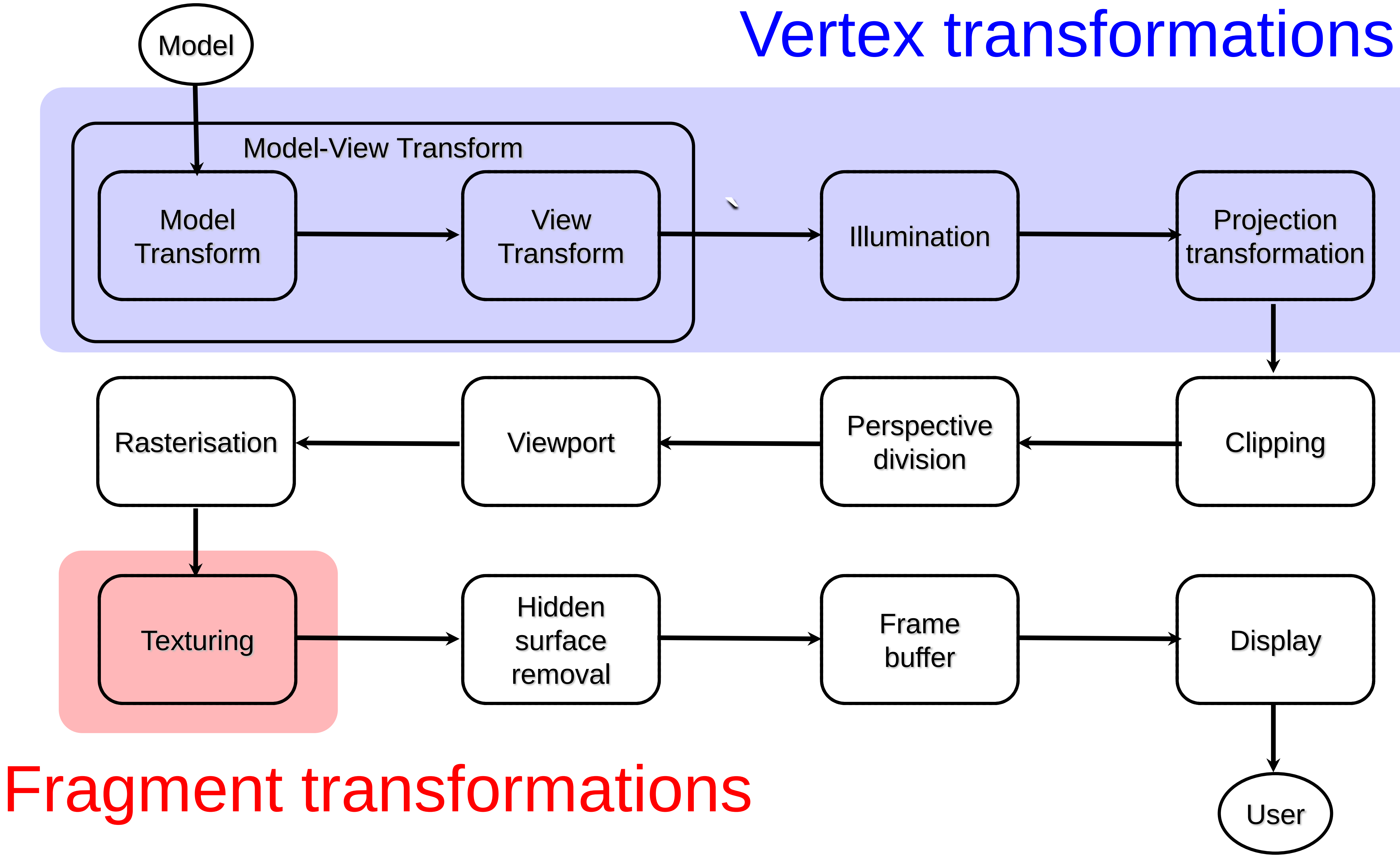
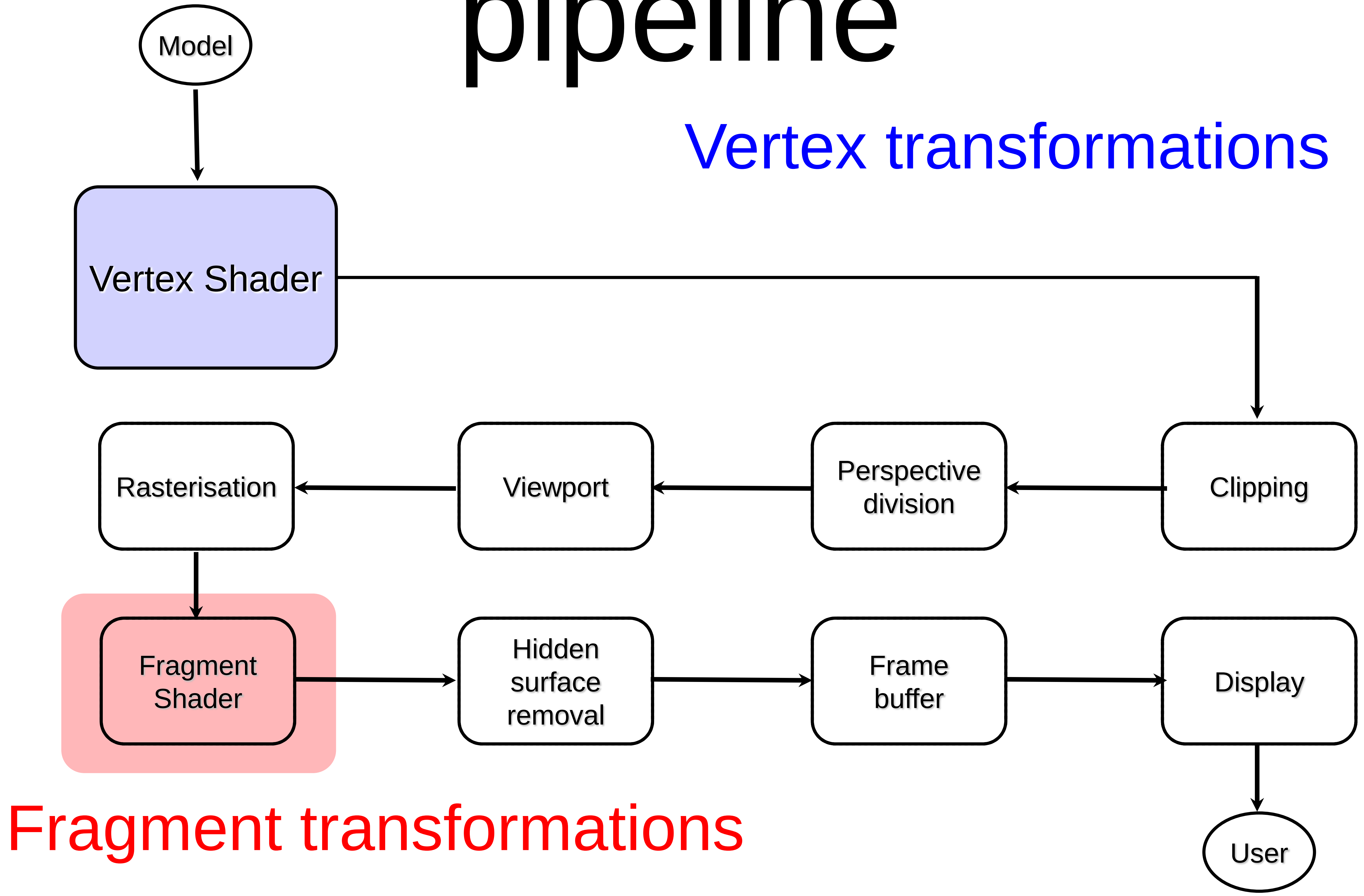
COMP3421

The programmable pipeline and Shaders

Fixed function pipeline



Programmable pipeline



Shaders

The programmable pipeline allows us to write shaders to perform arbitrary vertex and fragment transformations. (There are also optional tessellation and geometry shaders.)

Shaders are written in a special language called GLSL (GL Shader Language) suitable for specifying functions on the GPU.

GPU

The graphics pipeline performs the same transformations to thousands of millions of vertices and fragments.

These transformations are highly parallelisable.

The GPU is a large array of SIMD (single instruction, multiple data) processors.

Vertex Shaders

Replaces fixed function

- vertex transformation
- normal transformation, normalization
- vertex illumination
- Has access to OpenGL states such as model view transforms, colors etc (in later versions these must be explicitly passed in). These variables start with gl_ (eg gl_Color)

Vertex Shaders

Input: individual vertex in model coordinates

Output: individual vertex in clip (cvv) coordinates

They may also be sent other inputs from the application program and output color,lighting and other values for the fragment shader

They operate on individual vertices and results can’t be shared with other vertices.

They can’t create or destroy a vertex

Basic Vertex Shader

```
/** This does the bare minimum. It
transforms the input gl_Vertex and
outputs the gl_Position value in CVV
coordinates **/

void main(void) {

    gl_Position =

    gl_ModelViewProjectionMatrix *

                             gl_Vertex;

}
```

Fragment Shaders

Replaces fixed function

- texturing and colouring the fragment.

Enables lighting to be done at the fragment stage (such as phong shading) which could not be done in fixed function pipeline

Has access to OpenGL states (in later versions this must be explicitly passed in)

Fragment Shaders

Input: individual fragment in window coordinates

Output: individual fragment in window coordinates (with color set)

May receive inputs (that may be interpolated) from the vertex shader and inputs from the application program.

They can’t share results with other fragments.

Can access and apply textures.

Fragment Shaders

The fragment shader does not replace the fixed functionality graphics operations that occur at the back end of the OpenGL pixel processing pipeline such as

- depth testing
- alpha blending.

Basic Fragment Shader

```
/** Make everything red **/
void main (void) {
   gl_FragColor = vec4(1,0,0,1);
}
```

Setting up Shaders

1. Create one or more empty *shader objects* with **glCreateShader**.
2. Load source code, in text, into the shader with **glShaderSource**.
3. Compile the shader with **glCompileShader**.
4. Create an empty *program object* with **glCreateProgram**.
5. Bind your shaders to the program with **glAttachShader**.
6. Link the program with **glLinkProgram**.
7. Register your program for use with **glUseProgram**.

Setting up Shaders in OpenGL code

```
// create shader objects and

// reserve shader IDs

int vertShader =
gl.glCreateShader(GL2.GL_VERTEX_
SHADER);

int fragShader =
gl.glCreateShader(GL2.GL_FRAGMEN
T_SHADER);
```

OpenGL

```
// compile shader which is just
// a string (do once for vertex
// and then for fragment)

gl.glShaderSource(
    vertShader,
    vShaderSrc.length,
    vShaderSrc, vLengths, 0);
gl.glCompileShader(vertShader);
```

OpenGL

```
// check compilation

int[] compiled = new int[1];

gl.glGetShaderiv(vertShader,
       GL2ES2.GL_COMPILE_STATUS,
       compiled, 0);

if (compiled[0] == 0) {
    // compilation error!
}
```

OpenGL

```
// program = vertex shader + frag shader

int shaderprogram =
gl.glCreateProgram();

gl.glAttachShader(shaderprogram,
                           vertShader);
gl.glAttachShader(shaderprogram,
                           fragShader);
gl.glLinkProgram(shaderprogram);
gl.glValidateProgram(shaderprogram);
gl.glUseProgram(shaderprogram);
```

Different Shaders

Can set multiple shaders within a program

```
gl.glUseProgram(shaderProgram1);

//render an object with shaderprogram1

gl.glUseProgram(shaderProgram2);

//render an object with shaderprogram2

Gl.glUseProgram(0);

//render an object with fixed function
pipeline
```

Simple Vertex Shader

```
/* vertex shader */
void main(void) {
    gl_Position =
    gl_ModelViewProjectionMatrix *
                            gl_Vertex;
    gl_FrontColor = gl_Color;
}
```

Interpolation

By default the rasteriser interpolates vertex attributes such as

    positions, colors, normals

across primitives to generate the right interpolated values for fragments.

A variable qualified with **flat** will **not** be interpolated

Simple Fragment Shader

```
/* fragment shader */

void main(void) {

/* gl_Color is set by each vertex in
the vertex shader and the rasteriser
interpolates these values to get the
fragment gl_Color */

    gl_FragColor = gl_Color;

}
```

GLSL Syntax

C like language with

- No long term memory
- Basic types: float int bool
- Other type: sampler (textures)
- C++ Style Constructors
- Standard C/C++ arithmetic and logic operators
- if statements,loops

GLSL Syntax

Has limited support for loops

```
for(i=0; i< n; i++){

/* etc */

}
```

Conditional branching is much more expensive than on CPU!

Do not use too much – especially in fragment shader

GLSL Syntax

Has support for 2D, 3D, 4D vectors of different types

- vec2, vec3, vec4 are float vectors.
- ivec2, ivec3, ivec4 are int vectors.

Has support for float matrix types

- mat2, mat3, mat4

Operators are overloaded for matrix and vector operations

GLSL Syntax

No characters, strings or printf

No pointers

No recursion

No double (limited support in later versions)

Vectors

C++ Style Constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

For vectors can use [ ], xyzw, rgba

```
vec3 v;
```

`v[1], v.y, v.g` all refer to the same element

Swizzling: `vec3 a, b;`

```
a.xy = b.yx;
```

Matrix Components

Matrices are in column major order

M[i][j] is column i row j

```
mat4 m = mat4(1.0); //identity matrix
m = mat4(1.0,2.0,3.0,4.0, //first col
         5.0,6.0,7.0,8.0, //second col
         9.0,10.0,11.0 12.0,   //third col
         13.0,14.0,15.0,16.0); //fourth col
float f = m[0][1]; //Would be 2.0
```

Variable Qualifiers

uniform: read-only input variables to vertex or fragment shader. Can’t be changed for a given primitive.

attribute/in : read-only input variables to vertex shader. May be different for each vertex. Eg materials, normals, positions

varying/in, out : Outputs from the vertex shader that are passed to the fragment shader. They are interpolated for each fragment

Built in Vertex Shader Variables

in: gl_Vertex,
    gl_Normal,
    gl_Color
out: gl_FrontColor,
     gl_BackColor,
     gl_Position (must be written)

Built in Fragment Shader Variables

in: gl_FragCoord

gl_Color - If this is a front facing it will be the interpolated value set by the vertex shader for gl_FrontColor, (if it is back facing it will be gl_BackColor Note: gl_BackColor does not work on my computer).

out: gl_Fragment - fragment colour

in/out: gl_depth

Built-in Uniform Variables

gl_ModelViewMatrix

gl_ModelViewProjectionMatrix,

gl_NormalMatrix

gl_LightSource[0].position (in camera coords)

gl_LightSource[0].diffuse

etc

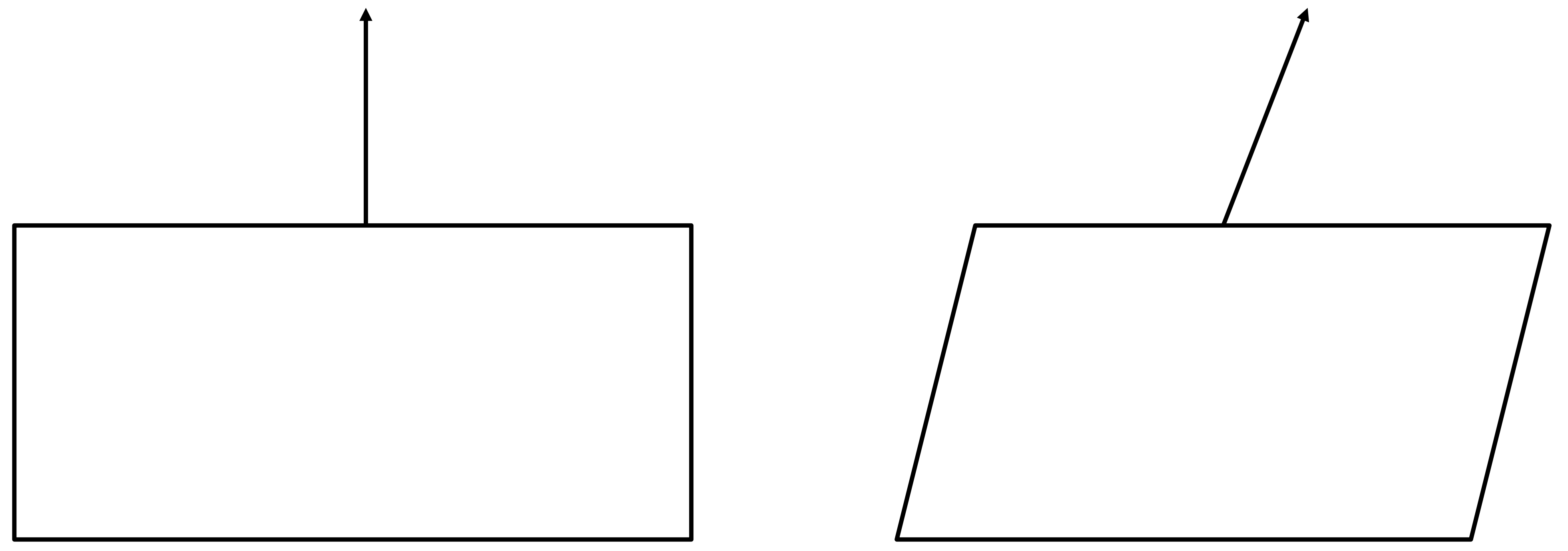
User Defined Variables

To pass in your own uniforms or attribute ‘in’ variables into your shaders from the application program

```
int loc =
gl.glGetUniformLocation(shaderProgram,”myVal”
);
gl.glUniform1f(loc,0.5);
//Or for attributes
int loc =
glGetAttribLocation(shaderProgram,”myVal”);
gl.glVertexAttrib1f(loc, 1, 1, 0, 1);
```

Aside: gl_NormalMatrix

If the *modelview* matrix contains a non-uniform scale then it will not transform normals properly. It is no longer perpendicular!



gl_NormalMatrix

Instead we use the transpose of the inverse of the upper 3*3 corner of the modelview matrix. The fixed function pipeline has been doing this behind the scenes for us.

In our shaders (not in later versions) we can use gl_NormalMatrix.

In later versions we would use a matrix library

Ambient light vertex shader

```
void main() {
 vec4 ambient =
       gl_LightModel.ambient *
       gl_FrontMaterial.ambient +
       gl_LightSource[0].ambient *
       gl_FrontMaterial.ambient;

 gl_FrontColour = ambient;
 gl_Position =  gl_ModelViewProjectionMatrix

                * gl_Vertex;
}
```

Ambient light fragment shader

```
/* Most of the work is done at the
vertex level. Here we just get an
interpolated gl_Color from the vertex
shader setting gl_FrontColor and we
just use that */

void main() {

    gl_FragColor = gl_Color;

}
```

Diffuse vertex shader

```
void main() {
   vec3 v, normal, lightDir;
   float NdotL;
   // transform the normal into eye space and normalize
   normal = normalize(gl_NormalMatrix * gl_Normal);

   //transform co-ords into eye space
    v = vec3(gl_ModelViewMatrix * gl_Vertex);

   // normalize the light's direction
   lightDir = normalize(gl_LightSource[0].position.xyz - v);

   NdotL = max(dot(normal, lightDir), 0.0);

   /* Compute the diffuse term */
   gl_FrontColor = NdotL * gl_FrontMaterial.diffuse *
                       gl_LightSource[0].diffuse;
   gl_Position = gl_ModelViewProjectionMatrix
                    * gl_Vertex;
}
```

Diffuse light fragment shader

```
/* Same as the ambient one

void main() {
    gl_FragColor = gl_Color;
}
```

Specular light vertex shader

```
vec4 specular = vec4(0.0,0.0,0.0,1.0);
vec3 dirToView = normalize(-v);
vec3 H = normalize(LightDir+dirToView);
// compute specular term if NdotL is  larger than  zero
if (NdotL > 0.0) {
   float NdotHV = max(dot(normal, H),0.0);
   specular = gl_FrontMaterial.specular *
              gl_LightSource[0].specular *
              pow(NdotHV,gl_FrontMaterial.shininess);
}
gl_FrontColor = ambient + diffuse + specular;
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Specular light fragment shader

```
void main() {
    gl_FragColor = gl_Color;
}
```

Phong vertex shader

```
/* data associated with each vertex */

out vec3 N;
out vec3 v;

void main(){

   /* send point and the normal in camera coords  to the
fragment shader – these will be interpolated */

   v = vec3(gl_ModelViewMatrix * gl_Vertex);
   N = normalize(gl_NormalMatrix * gl_Normal);

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

}
```

Phong fragment shader

```
in vec3 N;
in vec3 v;
void main (void){
   vec3 L = normalize(gl_LightSource[0].position.xyz - v);
   vec3 dirToView = normalize(-v);
   vec3 normal = normalize(N);

   //etc same calculations as done in the
   // specular vertex shader
   // except we are doing calculations on EVERY fragment
   gl_FragColor = ambient + diffuse + specular;

}
```

Things to try

These shaders have not handled basics such as

- multiple lights
- two sided lighting
- lights being enabled/disabled
- directional lights, spotlights etc
- attenuation...

More...

There are many more shading algorithms designed to implement different lighting techniques with different levels of speed and accuracy.

For example Cook Torrance is a more realistic model than Phong or Blinn-Phong

Check out the Graphics Gems and GPU Gems books for lots of ideas.

Optional Shaders

In later versions on GLSL, there are optional shaders between the vertex shader and the clipping stage.

Tesselation Shaders: can create additional vertices in your geometry

Geometry Shader: can be used to add, modify, or delete geometry,

GLSL Documentation

For the version compatible with class demos and slides:

https://www.opengl.org/registry/doc/GLSLangSpec.Full.1.30.10.pdf