The Programmable Pipeline

GLSL

OpenGL Shading Language

שקפים: ליאור שפירא

Why GLSL?

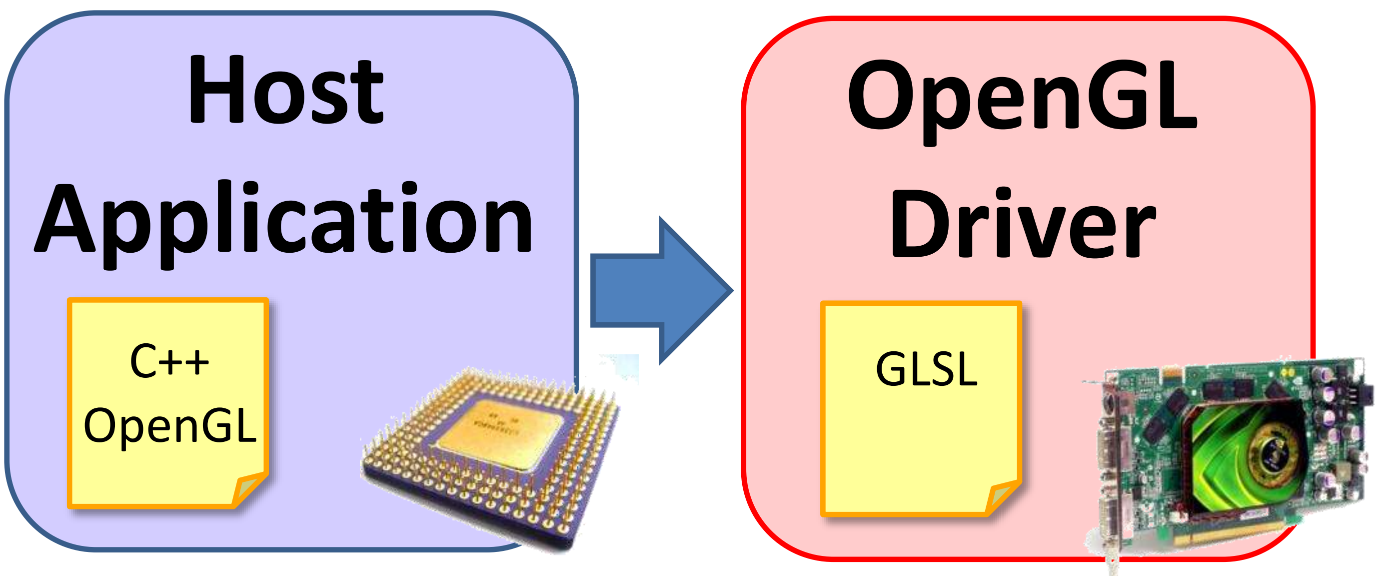
"Fixed functionality" can only get you so far.

- Linear transformations
- Gouraud Shading
- Limited operators multi-texturing

Inventing APIs for advanced features becomes more and more and more complicated

What is GLSL?

- GLSL replaces most of the fixed functionality with custom rendering
- A High level Language for programming graphic hardware



What is GLSL?

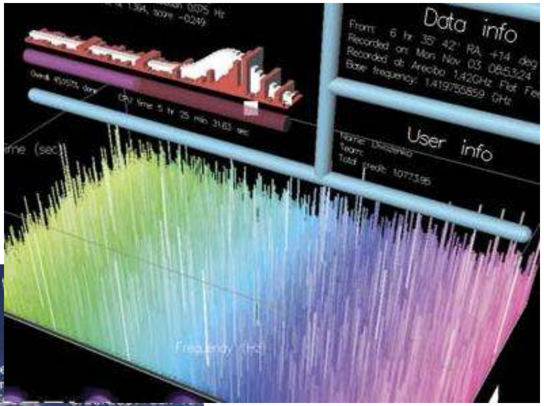
Enables effects which are impossible with the fixed functionality, in real time.

- Multi texturing with arbitrary function
- Advanced lighting and shadows
- Arbitrary deformation of the rendered shape.
- Procedural texture generation.
- Chromatic aberration
- Bump mapping
- Non photo realistic rendering

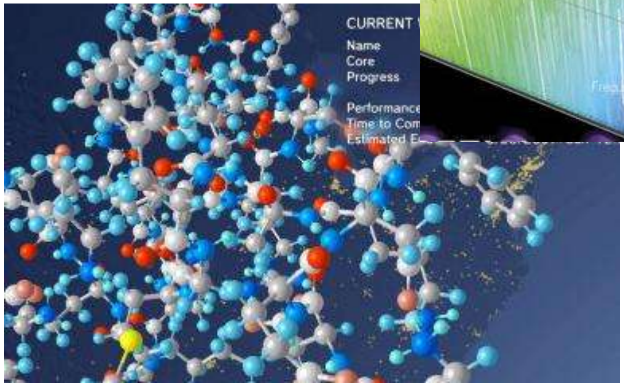
What is GLSL?

Allows General Purpose Computations (GPGPU)

- The GPU is made of many fast stream processors optimized for number crunching.
- May be used for purposes other than graphics

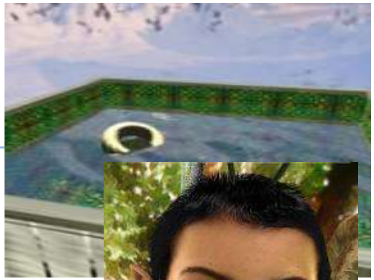


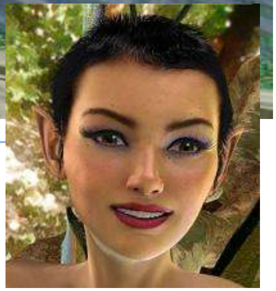
- Digital Signal Processing
  – Audio, Radio signals
- SETI@Home, Folding@Home



History

Riva 128 3D Rage, Voodoo	1997	First cards with Hardware acceleration Only fixed fragment pipeline.
Riva TNT, ATI Rage 128	1998	true color display (32 bit), pixel pipeline, alpha blending, 2 cores
GeForce 256 Voodoo 3	1999	Single chip “GPU”, OpenGL 1.2
GeForce 2 Radeon 7500	2000	2 Texture Units, OpenGL 1.3
GeForce 3 Radeon 8500	2001	First programmable cards. Allow simple, fixed length assembly code.
GeForce 4 Radeon 9500	2002	4 cores, faster, more memory
Radeon 9700 WildCat VP	2003	full flow control - ‘real’ sharers GLSL 1.0, OpenGL 1.5





History

OpenGL 2.0, GLSL 1.1 as part of the OpenGL standard. `glUniformARB()` → `glUniform()`	**2004**	GeForce 6 Radeon x850
12-16 cores Last AGP Cards	**2005**	GeForce 7 SLI Radeon x1650
GLSL 1.2 with OpenGL 2.1, mat2x3, mat4x2 data types	**2006**	GeForce 8 - PhysX Radeon HD2600
	2007	
GLSL 1.3 with OpenGL 3.0 FBOs, VBOs in the standard	**2008**	GeForce 9 Radeon HD 4600
GLSL 1.4 with OpenGL 3.1 minimum 16 Texture Units	**2009**	GeForce 200 Radeon RV 840
Geometry Shaders Tesellation Shaders?	**2010**	
Instanced rendering?		

Other Shading Languages

- **HLSL** - Microsoft **H**igh **L**evel **S**hading **L**anguage – Part of Direct3D

- **Cg** - nVidia's **C** for **G**raphics – Part of nVidia drivers.

Both languages are similar and are analogous in their capabilities.

In nVidia drivers, GLSL and HLSL code is compiled to Cg Code.

OpenGL Pipeline - Fixed Function

Input: Vertices

↓

Per-Vertex Operations — Model-View, Projection Transformations, Lighting, *glTexGen*

↓

Primitive Assembly — Group vertices to primitives

↓

Clip, Viewport, Cull

↓

Rasterization — Scan line conversion of every primitive

↓

Fragment Processing — Color interpolation, Texture mapping, Fog

↓

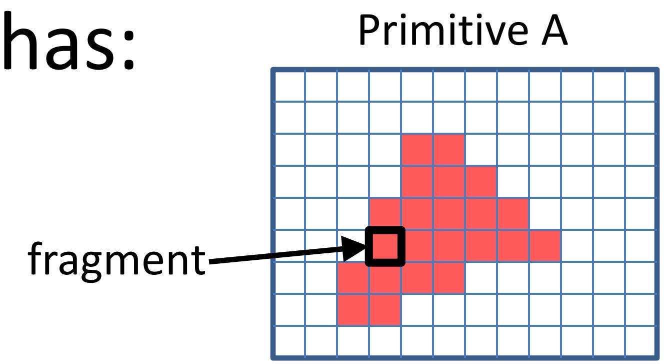
Per-Fragment Operations — Depth test, Stencil test and update, alpha blending

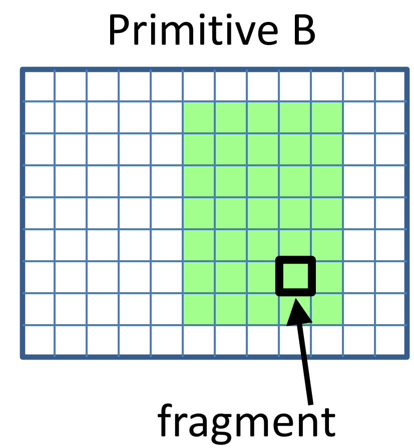
↓

Frame Buffer

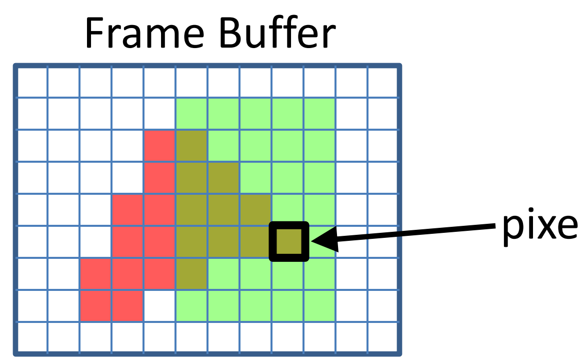
Fragment VS Pixel

- A fragment is a single pixel of a single primitive.
- Rasterizing a primitive generates fragments
- Every fragment has:
  - A position
  - Depth
  - Color





- A pixel in the frame buffer may be composed of one or few different fragments



OpenGL Pipeline - Programmable

Input: Vertices

GLSL Vertex Shader

Primitive Assembly

Clip, Viewport, Cull

Rasterization

GLSL Fragment Shader

Per-Fragment Operations

Frame Buffer

Required Goal: Assign coordinates to the processed vertex

Any per-vertex pre-processing required for the fragment shader.

Required Goal: Assign a color to the processed fragment

Or discard the fragment

Vertex Shader

- The Vertex Shader is invoked for every single vertex sent by the user

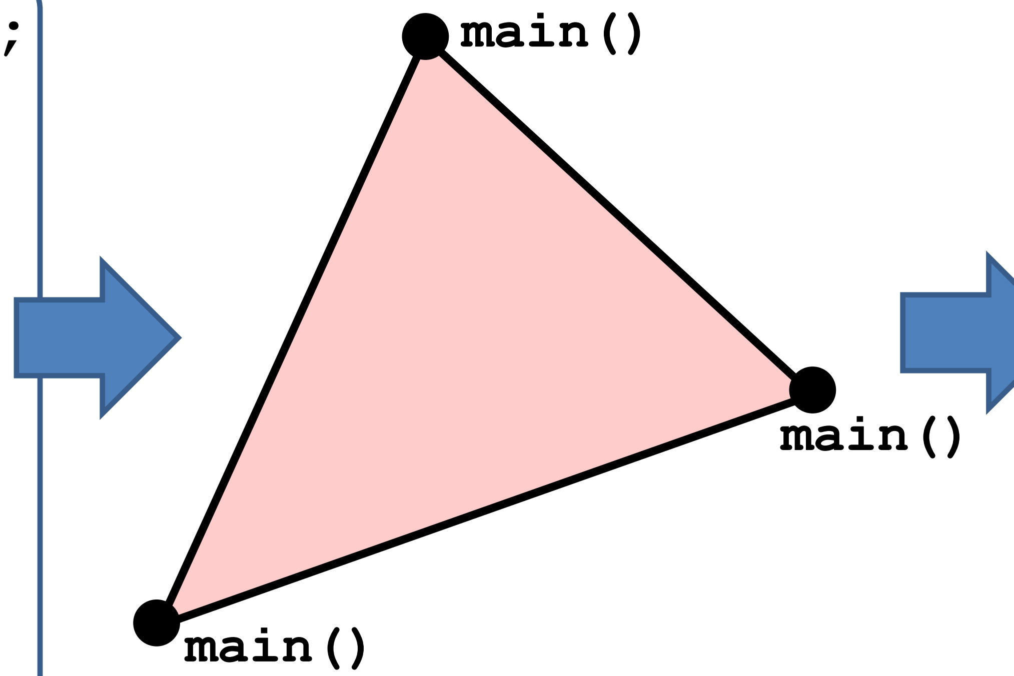
```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

A trivial vertex shader that translates every vertex according to the Model View and Projection matrix, mimicking the fixed function

- The minimal requirement of a vertex shader is to <u>set a position</u> for the vertex by assigning a value to `gl_Position`.

Vertex Shader

```
glBegin(GL_TRIANGLES);
  glVertex3f(0,0,0);
  glVertex3f(1,0,0);
  glVertex3f(1,1,0);
glEnd();
```

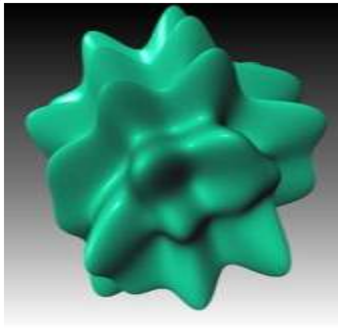


Every call to **`glVertex()`** invokes the vertex shader.
Sets the position of the vertex in clip-space

Vertex Shader

Possible tasks for the vertex shader:

- Transform the vertex by the *modelview* and *projection* matrix
- Custom manipulation of the position.
- Transform and normalize the vertex normal.
- Per-Vertex lighting
- Per Vertex color computation
- Prepare variables for the fragment shader.
- Access textures



Replaces Fixed Functionality!

There is no way to invoke it once replaced. need to re-implement!

Fragment Shader

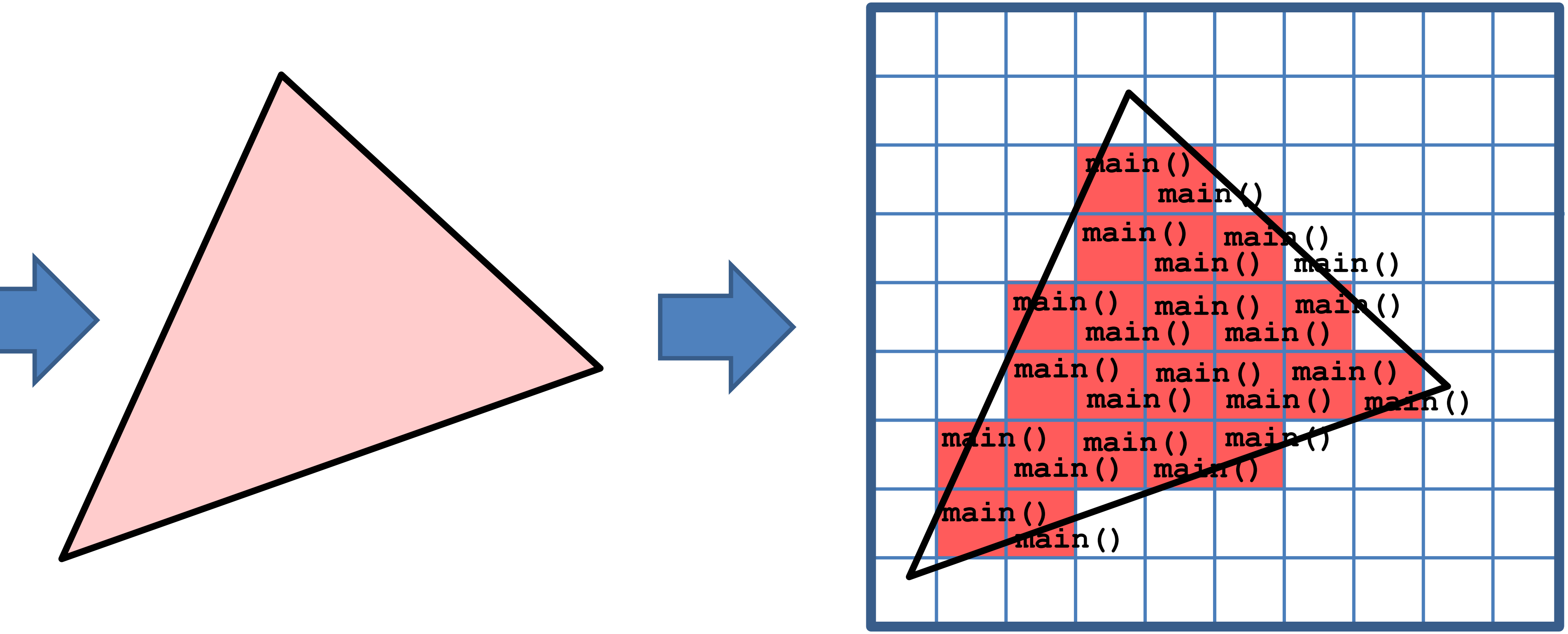
- The fragment shader is invoked for every single fragment of every displayed primitives

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

A trivial shader colors all fragments in a constant color.

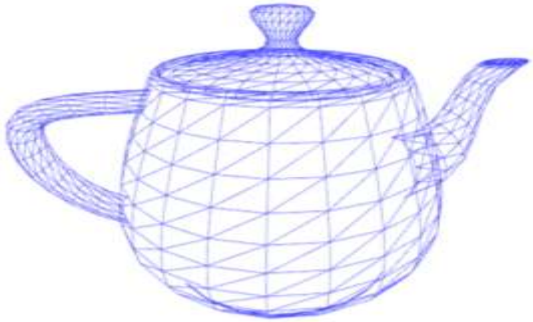
- The minimal requirement of a fragment shader is to assign a color to the fragment by setting `gl_FragColor` or discard it with '`discard`'
- *Optionally, can set the depth of the fragment.*

Fragment Shader



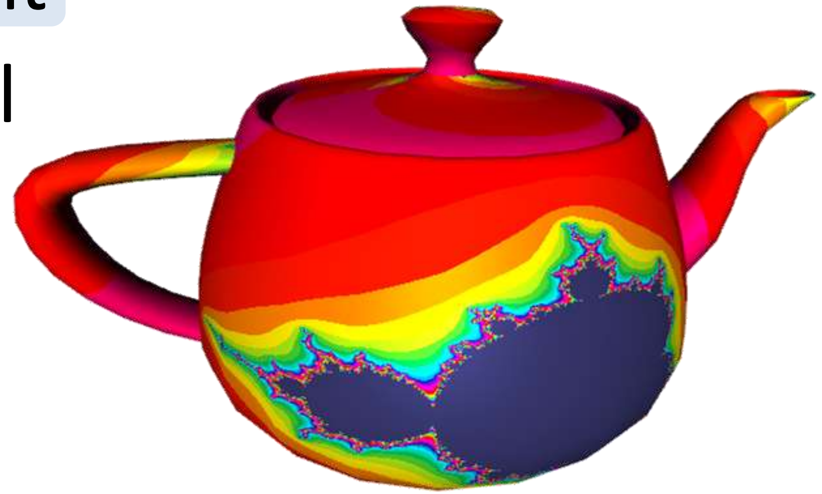
Every rasterized fragment invokes the fragment shader.
Sets the color of the fragment

Fragment Shader



Common tasks in the fragment shader

- Compute Color per-fragment
- Texture coordinate per-pixel
- Normal per-pixel
- Lighting per-pixel
- Apply texture
- Compute Fog or other global effects



Replaces Fixed Functionality!

There is no way to invoke it once replaced. need to re-implement!

GLSL Syntax

GLSL Syntax is a C-like language which borrows features from C++ and some original ideas.

- Has a preprocessor (`#define`, `#ifdef`, `no #includes`)
- Variables can be defined anywhere as in C++
- Most of C's flow control structures
  - `for`, `if`, `while`, `do…while`
- Functions - Allow argument overloading
  - Everything is passed by value.
- Comments – `//`, `/* */`

Trivial Shaders Example

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

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

- Sets a position for every vertex according to the Model View and Projection Matrix. [Trivial Example]
- Color all fragments yellow.
- `ftransform()` performs the vertex transformation in the fixed functionality.

Concurrent Execution

- A modern GPU (GeForce 9800) can have up to 256 cores on a single chip.
- Every core can run an instance of a vertex shader or a fragment shader

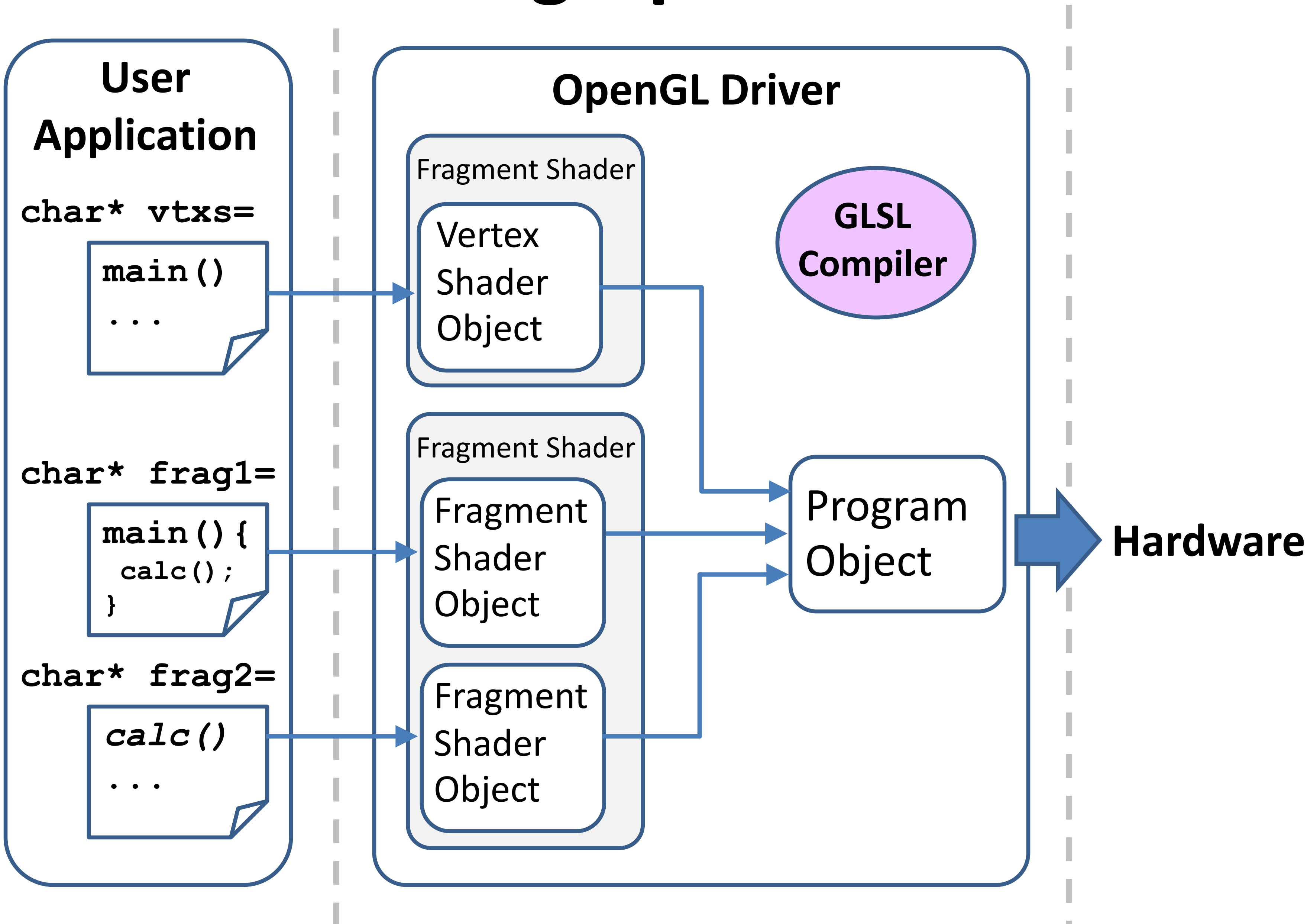
core0	Vtx 0	Vtx 4	Vtx 10	Vtx 11	Frag4	Frag9	Frag4	Frag4	Frag18	Frag22
core1	Vtx 1	Vtx 5	Vtx 9	Vtx 12	Frag3	Frag8	Frag10	Frag4	Frag15	Frag19
core2	Vtx 2	Vtx 6	Frag0	Frag1	Frag6	Frag11	Frag13	Frag16	Frag20	
core3	Vtx 3	Vtx 7	Vtx 8	Frag2	Frag5	Frag7	Frag12	Frag14	Frag17	Frag21

- For this reason, one execution of a shader cannot access the result of another execution

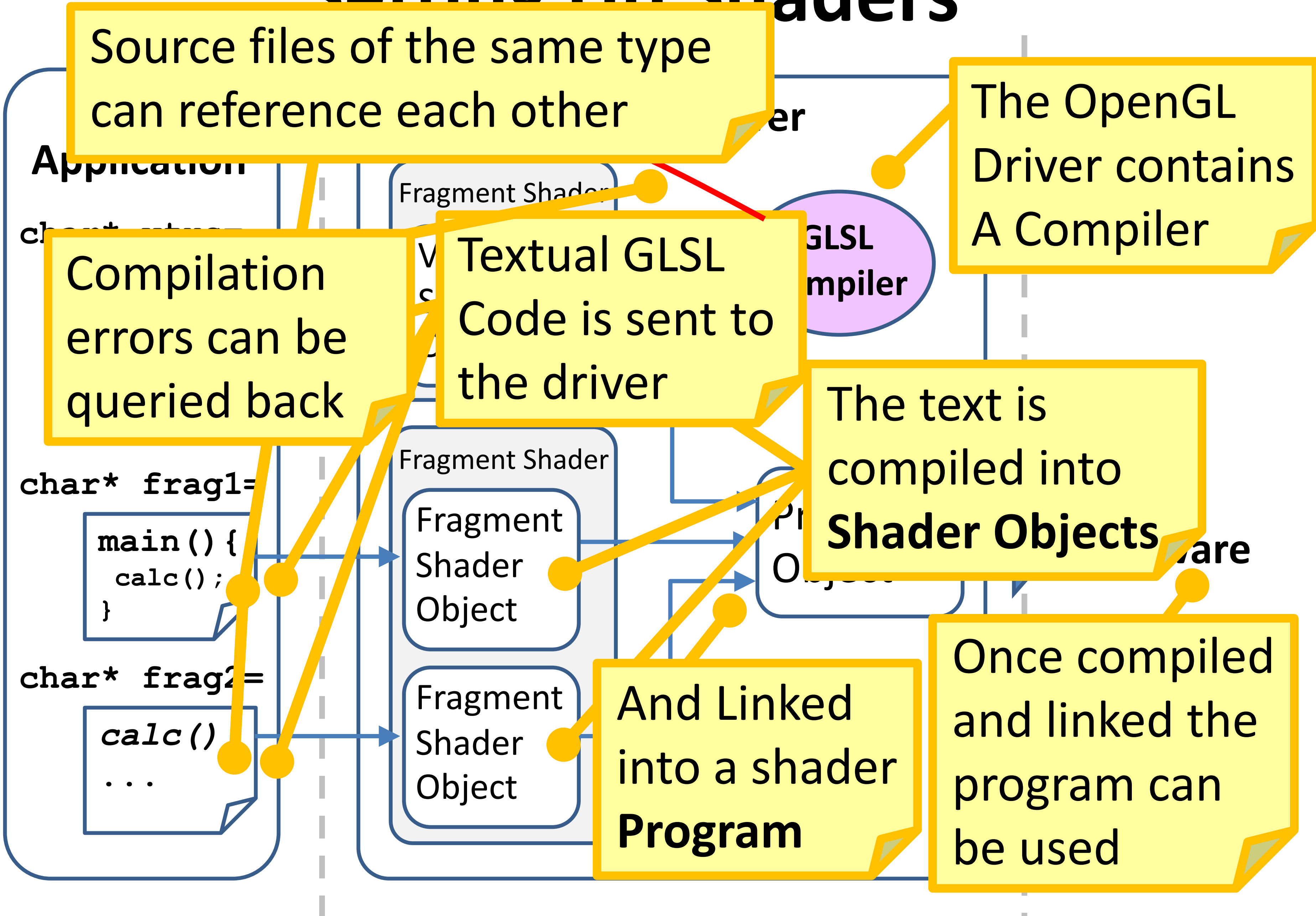
Setting Up Shaders

- To be able to call the shader setup OpenGL functions they first need to be *mapped.*
- Doing this manually (using `wglGetProcAddress()`) is an unpleasant labor.
- There are several libraries who do the dirty work for you

Setting Up Shaders



Setting Up Shaders



Setting Up Shaders

```
Vertex info
-----------
0(7) : warning C7011: implicit cast from "vec2" to "float"
0(5) : err

Fragment info
-------------
0(3) : warning C7555: 'varying' is deprecated, use 'in/out' instead
0(10) : error C0000: syntax error, unexpected floating point
        constant at token "<undefined>"
0(10) : error C0501: type name expected at token "<undefined>"
0(10) : error C1002: the name "c" is already defined at 0(9)
0(11) : error C7011: implicit cast from "float" to "int"
0(12) : warning C7533: global variable gl_FragColor is deprecated
        after version 120
0(12) : error C1115: unable to find compatible overloaded function
        "texelFetch(error, ivec2, int)"
```

Source text index in the call to glShaderSource()

Line number

Setting Up Shaders

Using a shader program:

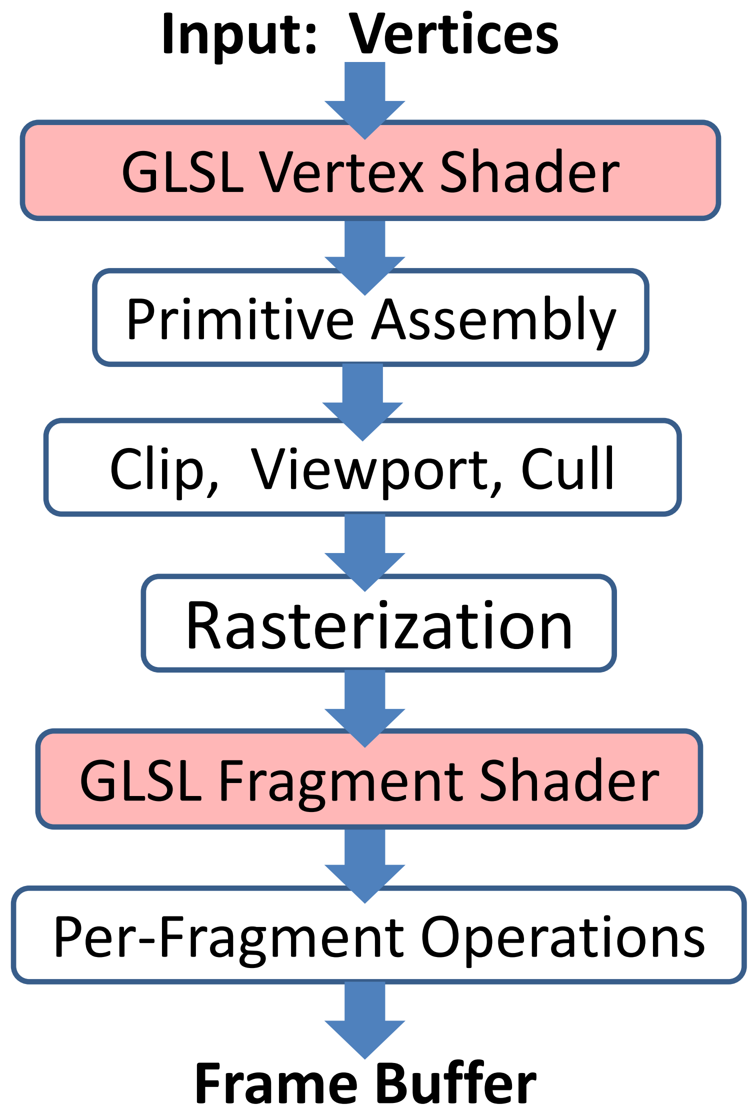
```
glUseProgram(prog);
renderScene(...);
glUseProgram(0);
```

The used program becomes part of the OpenGL state.

All rendering will pass through the shaders.

Using program 0 reverts back to the fixed function

May **not** be called between `glBegin()..  glEnd()`



Data Types

- Data types for GLSL Variables

Vectors	Floating Point	Integer	Boolean
Primitive	float	int	bool
2 elements	vec2	ivec2	bvec2
3 elements	vec3	ivec3	bvec3
4 elements	vec4	ivec4	bvec4

Matrices
mat2
mat3
mat4

Texture Samplers	For shadow maps
sampler1D	sampler1DShadow
sampler2D	sampler2DShadow
sampler3D	
samplerCube	

structs

```
struct Object
{
    vec3 position;
    vec3 color;
};
```

1-D arrays	`vec4 myArray[10];`

No Strings!

Initialization and Constructors

- Variable initialization and assignment is similar to explicit constructors of C++

```
float a, b = 1.0;
int i = 1;
i = floor(a);
bool c = true;
c = (a == b);
```

No qualifiers for float!
(unlike `1.0f` in c++)

```
vec2 v = vec2(1.0, 2.0);     // compose from values
v = vec2(3.0, 4.0);          // doesn't have to be in an initialization
vec4 u = vec4(0.0);          // initialized all elements to 0.
vec2 t = vec2(u);            // take the first two components
vec3 vc = vec3(v, 1.0);      // compose vec2 and float
```

Constructors - Matrices

```
mat2 d = mat2(1.0);  // identity matrix
mat4 t = mat4(2.0);  // diagonal matrix
```

$$\begin{pmatrix} 2.0 & 0 & 0 & 0 \\ 0 & 2.0 & 0 & 0 \\ 0 & 0 & 2.0 & 0 \\ 0 & 0 & 0 & 2.0 \end{pmatrix} \quad \begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$$

```
mat2 m = mat2(1.0, 2.0,     // first column
              3.0, 4.0);    // second column
```

$$\begin{pmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{pmatrix}$$

```
vec2 v(1.0,2.0), u(3.0, 4.0);
m = mat2(v, u);   // compose a matrix of two columns

mat3 t3 = mat3( vec3(v, 0.0),
                vec3(u, 0.1),
                vec3(5.0)     );
```

$$\begin{pmatrix} 1.0 & 3.0 & 5.0 \\ 2.0 & 4.0 & 5.0 \\ 0.0 & 0.1 & 5.0 \end{pmatrix}$$

Constructors - Arrays

```
float a[5] = float[](1.0, 2.0, 3.0, 4.0, 5.0);
float b[5] = float[5](1.0, 2.0, 3.0, 4.0, 5.0);
b[3] = 1.0;

b = float[5](x, y, z, 3.0, x+y);    // assignment of an array

vec2 va[3] = vec2[](vec2(0.0), vec2(1.0), vec2(2.0));
                                    // array of vectors

float c[5][6];     // illegal – no arrays of arrays.
vec4 vs[10] = vec4[10](...);
vs[5][2] =  1.0;  // OK.  access 2nd component of 6th vector in the array.
```

Constructors - Structs

```
struct Object        // composite structure definition
{
  vec4 position;           // a data member
  struct ObjectColor       // an inner struct
  {
    vec3 color;
    float intensity;
  } objectColor;           // a data member of the inner class
} obj1 = Object(u, ObjectColor(c, 0.9));   // instantiation

Object obj2;               // another instance of Object struct.
ObjectColor inner1;        // an insatnce of the inner struct
obj2.objectColor.color = vec3(1.0);
obj1.objectColor = inner1;    // copying values
```

Component Access

- vec2/3/4s can be considered either as vectors, colors or texture coordinates

```
vec4 v(1.0, 2.0, 3.0, 4.0);
float a = v.x, b = v.y;
float a = v.r, b = v.g;
float a = v.s, b = v.t;
float a = v[0], b=v[1];

v.x = 2.0;
float c = v.r;  // c gets the value 2.0.
V[3] = 4.2;
vec2 u;
float d = u.z;  // Error – no z in vec2
```

{x, y, z, w} - treat as a vector
{r, g, b, a} - treat as a color
{s, t, p, q} - treat as texture coordinate
[0],[1],[2],[3] - treat as an array

Swizzling

More than one components can be accessed by appending their names, from the same name set.

```
vec4 v;
vec3 a = v.rgb;
vec3 a = vec3(v.r, v.g, v.b); // same thing as above
vec3 b = v.gbr;
vec4 c = v.wzxy;
vec2 d = v.ra;              // red,alpha
vec4 f = v.xxyy;            // ok to duplicate components
vec3 e = v.rgz;             // Illegal. can't mix component sets

vec2 u(1.0,2.0);
v = u.xyz;                  // illegal. vec2 doesn't have z
```

Swizzling

Swizzling may also occur in the Left side of an assignment.

```
vec4 u(1.0, 2.0, 3.0, 4.0);
u.xw = vec2(5.0, 6.0); // u = (5.0, 2.0, 4.0, 6.0)

vec3 v(1.0, 2.0, 3.0);
v.xyz = v.yxz;         // v = (3.0, 2.0, 1.0)

v.xx = vec2(1.0, 2.0);   // illegal. Can't duplicate in l-value
v.rgb = vec2(1.0);       // illegal. Mismatch vec3,vec2
v.rgxy = vec4(…);        // illegal. mixing sets
```

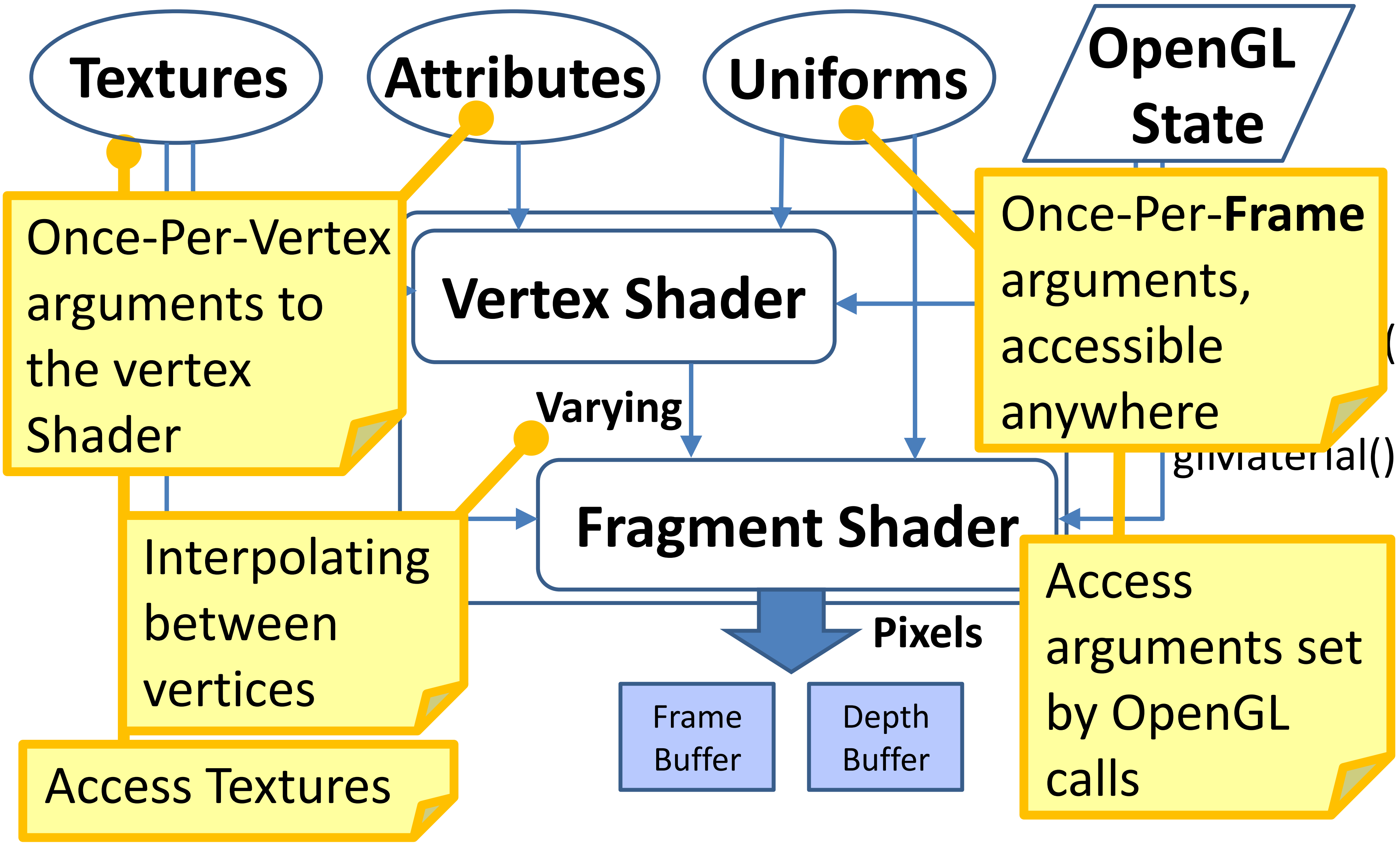
Component Access

- Matrices and arrays are accessed using the index operator. Structs using the '.' operator.

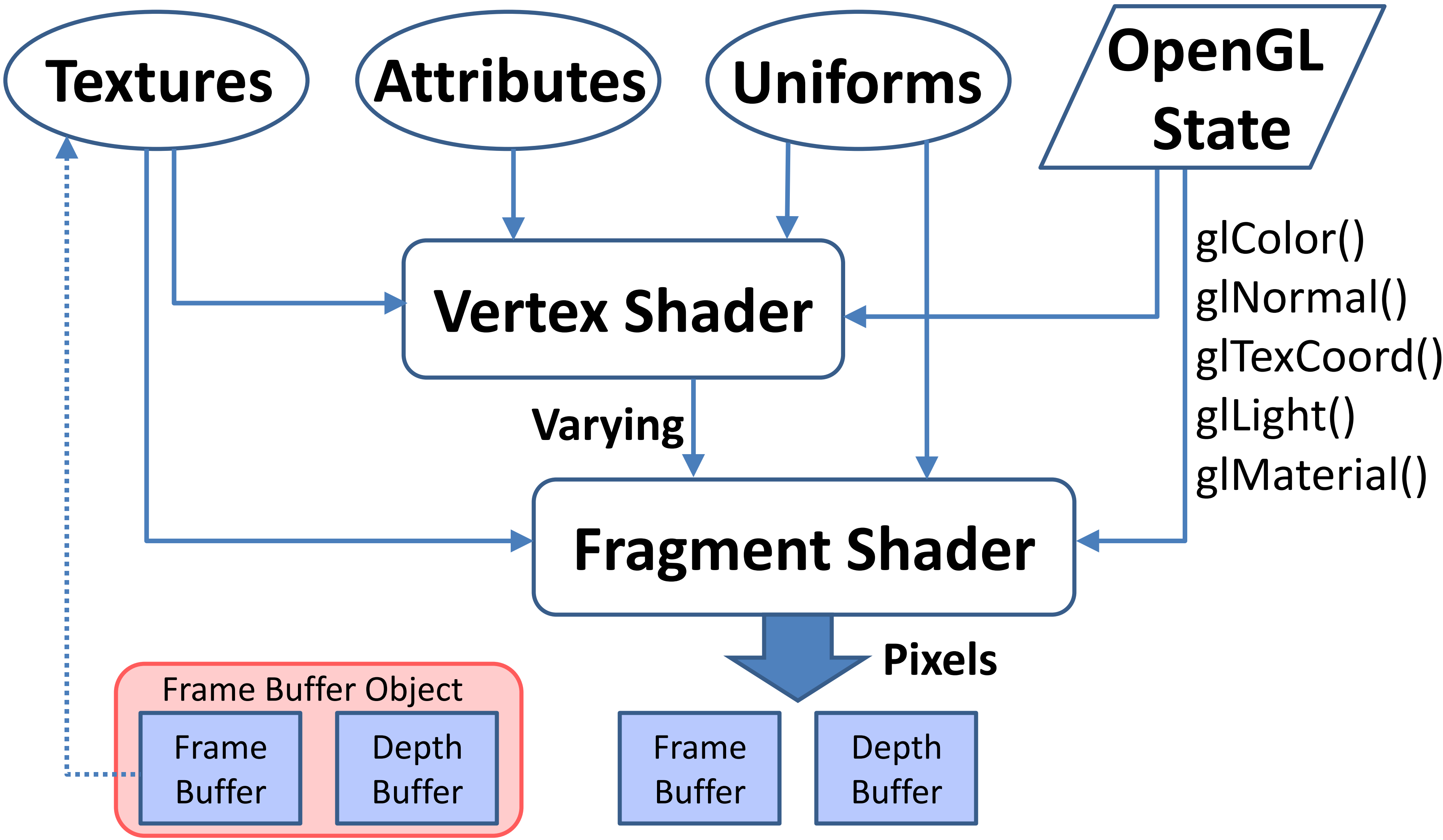
```
mat4 m(2.0);                 // 4x4 diagonal matrix
float a = m[0][0];           // single element access
vec2 v = m[1];               // whole vector access
m[0] = vec2(0.0, 1.0);
m[2][3] = 3.0;               // last element of 3rd column

int a[5] = int[5](1,2,3,4,5);
a[0] = 3;
int len = a.length();

object1.objectColor.color = vec3(0.5, 1.0, 1.0);
vec4 p = object1.position;
```

COMMUNICATION WITH THE HOST OPENGL PROGRAM

Shaders Communication



Shaders Communication



Shaders Communication

- **Uniforms** - Once-Per-Frame arguments, accessible from the vertex and fragment shader
- **Attributes** - Once-Per-Vertex arguments. accessible only from the vertex Shader.
- **Varying** - Communicating between the vertex shader and fragment shader. Interpolated linearly between vertices.
- **OpenGL State** - Vertex and fragment shader can access arguments set by OpenGL calls.
- **Textures** - All texels are accessible from both vertex and fragment shaders.

Uniform Variables

- A uniform variable can have its value changed **only between primitives**.
- Can’t be changed between `glBegin()..glEnd()`

```
uniform vec3 color;
void main()
{
     gl_FragColor = vec4(color);
}
```

- Suitable for parameters that change seldom, say once per frame.
- **Read-Only** in both vertex and fragment shaders

Uniform Variables

- Once the program is compiled and linked the user can get the location of the variable

```
glUseProgram(prog);
uint loc = glGetUniformLocation(prog, "color");
```

- Returns -1 if the name isn't an Active variable

```
uniform vec3 color;
void main()
{
    gl_FragColor = vec4(1.0);
}
```

With this shader

`glGetUniformLocation(prog, "color")` returns -1.

Setting values to Uniforms

Use the appropriate flavor of **`glUniform()`** for setting values to uniform variable from your C/C++ code.

```
glUniform2f(uint loc, float a, float b)
```

Number of Components

Type of Components

```
glUniform2fv(uint loc, uint size, float* ptr)
```

Number of Components

Type of Components

`size` is used for array. Set to 1 for non-arrays.

Setting values to Uniforms

```
glUseProgram(prog);
glUniform2f(loc, 1.0, 2.0);  → vec2
glUniform3i(loc, 1, 2, 3);   → ivec3
float a[4] = {1.0f, 2.0f,
              3.0f, 4.0f};
glUniform4fv(loc, 1, a);     → vec4
glUniform2fv(loc, 2, a);     → vec2[2]
float b[9] = {1.0f, 2.0f, 3.0f,
              4.0f, 5.0f, 6.0f,
              7.0f, 8.0f, 9.0f};
glUniform3fv(loc, 3, b);     → vec3[3];

glUniformMatrix2fv(loc, 1, false, a) → mat2
glUniformMatrix2fv(loc, 1, true, a)  → mat2
```

The program needs to be in use before setting any variables

$$\begin{pmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{pmatrix}$$

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

Transpose

Example – Uniform Variables

- Vertex Shader – Squash/Scale

```
uniform float ratio;

void main()
{
        vec4 pos = gl_Vertex;
        pos.x *= ratio;
        pos.y /= ratio;
        gl_Position =
        gl_ModelViewProjectionMatrix * pos;
}
```

- Fragment Shader - single color / combine with procedural texture generation

Shaders Communication

- Uniforms
- **Attributes** - Once-Per-Vertex arguments. accessible only from the vertex Shader.
- Varying
- OpenGL State
- Textures

Attribute Variables

- An attribute variable can have its value changed at any time
- Can be changed between `glBegin()..glEnd()`

```
attribute float height;
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_Position.y += height;
}
```

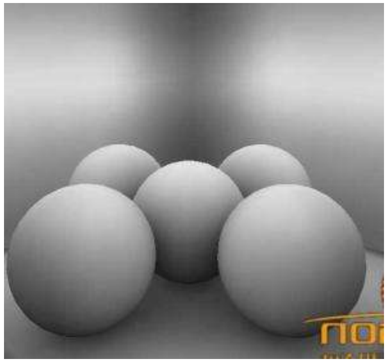
- Suitable for parameters that change frequently, say for every single vertex.
- **Read-Only** and only in the **vertex shader**.

Attribute Variables

- Like uniforms, before an attribute is used its location needs to be retrieved

```
glUseProgram(prog);
uint loc = glGetAttribLocation(prog, "height");
```

- Usage examples
  - Tangent and bi-tangent (bump mapping)
  - Per-point `glPointSize()`
  - Distance to nearest object - for global illumination effects
  - Reflection/refraction parameters for environment mapping



Setting values to Attributes

Use the `glVertexAttrib()` for setting values to uniform variable.

```
glVertexAttrib2f(uint loc, float a, float b)
```

Number of Components

Type of Components

```
glVertexAttrib2fv(uint loc, float* ptr)
```

Number of Components

Type of Components

Matrices are accessed by successive locations attribute arrays are not supported

Setting values to Attributes

```
glUseProgram(prog);
glVertexAttrib2f(loc, 1.0, 2.0);  → vec2
glVertexAttrib3i(loc, 1, 2, 3);   → ivec3
float a[4] = {1.0f, 2.0f, 3.0f, 4.0f};
glVertexAttrib4fv(loc, a);    → vec4

float b[9] = {1.0f, 2.0f, 3.0f,
              4.0f, 5.0f, 6.0f,
              7.0f, 8.0f, 9.0f};
glVertexAttrib2fv(l, b);
glVertexAttrib2fv(loc+1, b+3);
glVertexAttrib2fv(loc+2, b+6); → mat3
```

$$\begin{pmatrix} 1.0 & 4.0 & 7.0 \\ 2.0 & 5.0 & 8.0 \\ 3.0 & 6.0 & 9.0 \end{pmatrix}$$

C Pointer arithmatic

Setting values to Attributes

Application

```
uint loc;
void init() {
  glUseProgram(prog);
  loc = glGetAttribLocation(prog, "height");
}
```

```
void paintEvent()
{
  glUseProgram(prog);
  glBegin(GL_TRIANGLE);
    glVertexAttrib1f(loc, 2.0);
    glVertex2f(-1,1);
    glVertexAttrib1f(loc, 1.5);
    glVertex2f(1,1);
    glVertexAttrib1f(loc, -2.0);
    glVertex2f(-1,-1);
  glEnd();
}
```

Vertex Shader

```
attribute float height;
void main()
{
    gl_Position =
       ftransform();
    gl_Position.y += height;
}
```

Shaders Communication

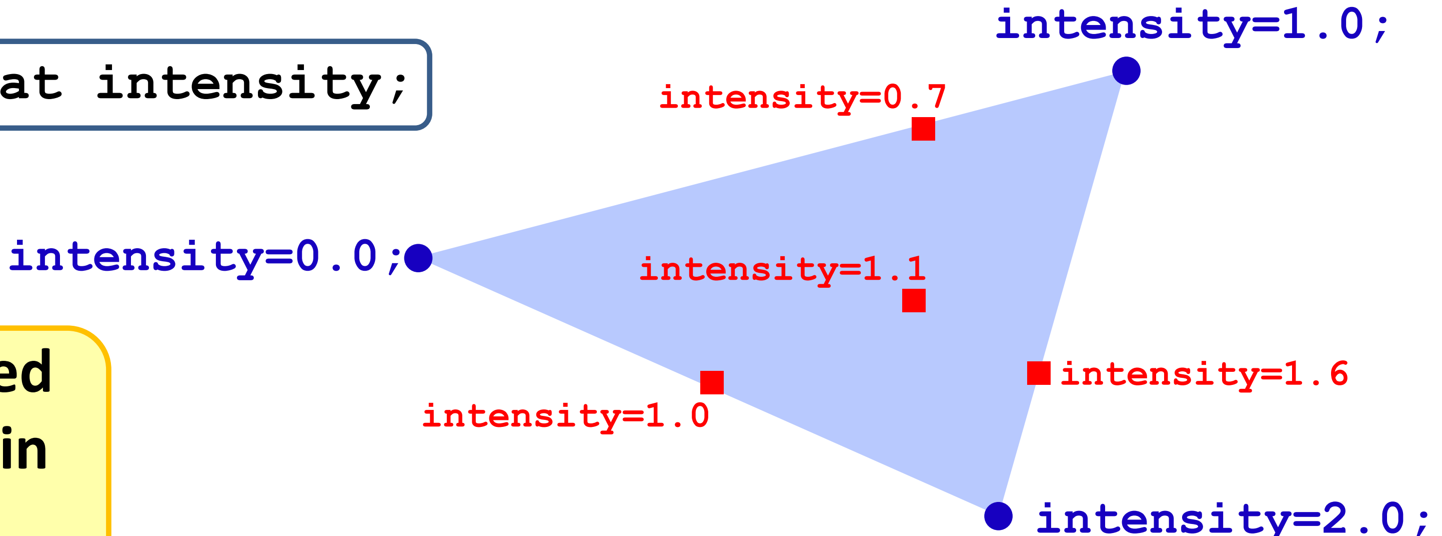
- **Uniforms**
- **Attributes**
- **Varying** - Communicating between the vertex shader and fragment shader. Interpolated linearly between vertices.
- **OpenGL State**
- **Textures**

Varying Variables

- Varying variable allow the vertex shader to communicate with the fragment shader.
- The vertex shader writes values to the variable and the fragment shader reads the <u>linearly interpolated</u> values between vertices.

```
varying float intensity;
```

Must be defined the same way in both shaders



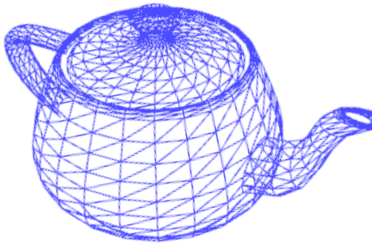
Using Varying Variables

Vertex Shader

```
varying vec3 intensity;
void main()
{
    intensity = gl_Vertex.xyz;
    gl_Position = ftransform();
}
```

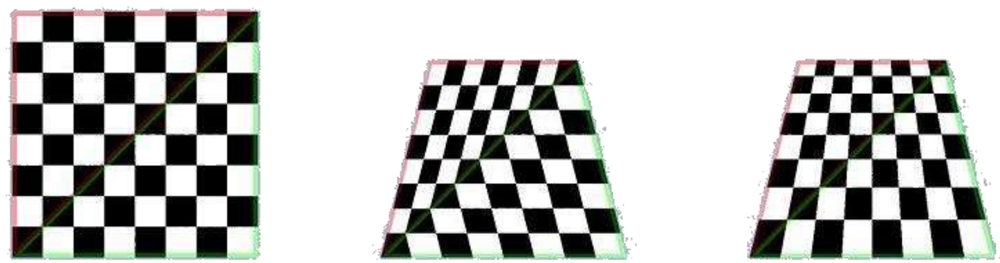
Fragment Shader

```
varying vec3 intensity;
void main()
{
    gl_FragColor = abs(vec4(intensity, 1.0));
}
```



Varying Variables

- Read-only in the fragment shader.
- The vertex shader can write and read back what it wrote.
- Interpolation is always on and always perspective-correct.



Flat **Affine** **Correct**

- More fine tuned interpolation was introduced in GLSL 1.4

Varying Variables

- The host OpenGL application can't directly access varying variables
- If necessary, this is easily achieved by copying an attribute to a varying variable.

Vertex Shader

```
varying float intensity;
attribute float vtxIntensity
void main()
{
    intensity = vtxIntensity;
    gl_Position = ftransform();
}
```

Fragment Shader

```
varying float intensity;
void main()
{
    gl_Position =
        vec4(intensity);
}
```

Example - Varying Variables

- Render model using varying built-in variables (gl_Vertex, gl_Normal)

```
varying vec3 intensity;
void main()
{
   intensity = gl_Vertex.xyz;
   gl_Position = ftransform();
}
```

Vertex Shader

```
varying vec3 intensity;
void main()
{
    gl_FragColor =
        abs(vec4(intensity, 1.0));
}
```

Fragment Shader

Shaders Communication

- **Uniforms**
- **Attributes**
- **Varying**
- ***OpenGL State***
- **Textures** - All texels are accessible from both vertex and fragment shaders.

Texture Variables

- A Textures is represented in a shader by a variable of type "**`sampler1D/2D/3D`**".
- A sampler is an **opaque** data type - It can't be assigned or evaluated directly.
  - It can only be passed around to functions.
  - The GLSL program can't access the actual value.
- Textures in shaders are read-only. They cannot be modified or written to.

Texture Variables

- A sampler can only be a uniform variable or a function argument and can only be initialized by the host.

```
uniform sampler2D tex;
void func() {
    sampler2D mytex;  // illegal. Can't create sample variables
    tex = 3;  // illegal. Samplers cannot be used in expressions
}
void func2(sampler2D argtex) {  // OK. sampler as function argument
  ...
}
void main() {
   func2(tex);  // OK. passing a sampler to function
}
```

Texture Variables

- Using a sampler to get the data of a texture

```
uniform sampler2D tex;
   // Sample using texture coordinates (range: [0,1])
vec4 c = texture2D(tex, vec2(0.5, 0.5));

   // Sample using absolute image coordinates
vec4 c = texelFetch(tex, ivec2(200, 200), 0);

   // Get the absolute size of a the image
ivec2 sz = textureSize(tex, 0);
```

LOD argument
Level **O**f **D**etail
The mipmap index.
0 is the original image

Using Textures

Application

```
uint loc;
void init() {
  uint texobj = initTexture("hello.png");
  glActiveTexture(1);
  glBindTexture(texobj)
  glUseProgram(prog);
  loc = glGetUniformLocation(prog, "tex");
  glUniform1i(loc, 1);
}
```

```
void paintEvent()
{
  glUseProgram(prog);
  drawObject();
}
```

Fragment Shader

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

Standard Library

GLSL provides a wide selection of functions that may be implemented in hardware

Trigonometric	sin, cos, tan, asin, acos, atan, radians, degrees
Exponential	**pow**, exp, log, exp2, log2, sqrt, inversesqrt
Common	**abs**, sign, round, trunc, floor, ceil, fract, mod, min, max, **clamp**, **mix**, step, smoothstep, isnan, isinf
Geometric	**length**, distance, dot, cross, **normalize**, reflect, refract
Texture	texture1D/2D/3D/Cube, texelFetch, textureSize
Component-wise	lessThan, lessThanEqual, equal, any, all, not
Matrix	matrixCompMult, outerProduct, transpose
Other	noise1, noise2… , **ftransform**

noiseX returns 0.0 in nVidia cards…

Standard Library

```
vec3 ← sqrt(vec3)
vec2 ← cos(vec2)
float ← length(vec3)
mat4 ← transpose(mat4)
```

```
vec3 ← normalize(vec3)
float ← sin(float)
float ← distance(vec2, vec2)
```

```
bvec3 ← lessThan(vec3,vec3)   //component-wise result
bvec2 ← equal(mat2, mat2)     //component-wise result
bool ← any(bvec3)             // logical AND
bool ← all(bvec4)             // logical OR
bvec4 ← not(bvec4)
```

```
clamp(x, minVal, maxVal) = min(max(x, minVal), maxVal);
mix(x, y, a) = x·(1-a) + y·a;
```

Shaders Communication

- Uniforms
- Attributes
- Varying
- **OpenGL State** - Vertex and fragment shader can access arguments set by OpenGL calls.
- Textures

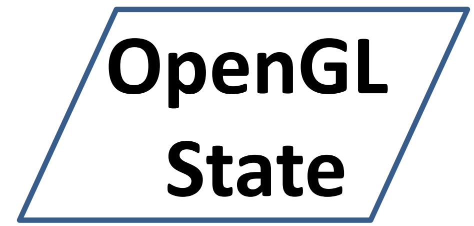
Built-In Variables

- GLSL defines Built-In global variables that are accessible to shaders and used for various purposes
- All built-in variables use the reserved prefix "`gl_`" and are defined in the global scope.
- We've already seen a few of these
  - `gl_Vertex`
  - `gl_ModelViewProjectionMatrix`
  - `gl_FragColor, gl_Position`

Built-In Variables

Built-in variables allow:

- Access to the OpenGL fixed function state
  - Fixed function transformations
  - Lighting, materials, point parametes
- Setting required and optional output
  - pixel position, fragment color
- Standartized communication between vertex and fragment shaders
  - texture coordinates



Built-In Attributes

Like any attributes, can be read **only in the Vertex Shader**.

`attribute vec4 gl_Vertex;` // filled with **`glVertex()`**

A vertex shader would usually use **`ftransform()`** instead of referencing **`gl_Vertex`** directly, unless it wants to change (deform) the position of the vertex

`attribute vec4 gl_Color;` // filled with **`glColor()`**

This is always the value of **`glColor()`**, unrelated to material or lighting state

`attribute vec3 gl_Normal;` // filled with **`glNormal()`**

Used for lighting calculations.
If used it needs to be transformed with **`gl_NormalMatrix`**

Built-In Attributes

```
attribute vec4 gl_MultiTexCoord0;       // glTexCoord()
attribute vec4 gl_MultiTexCoord1..8;   //glMultiTexCoord()
```

In the fixed function:

`glTexCoord()` only relates to the texture in TIU-0

`glMultiTexCoord(GL_TEXTURE`*`i`*`,…)` relates to any TIU, including 0

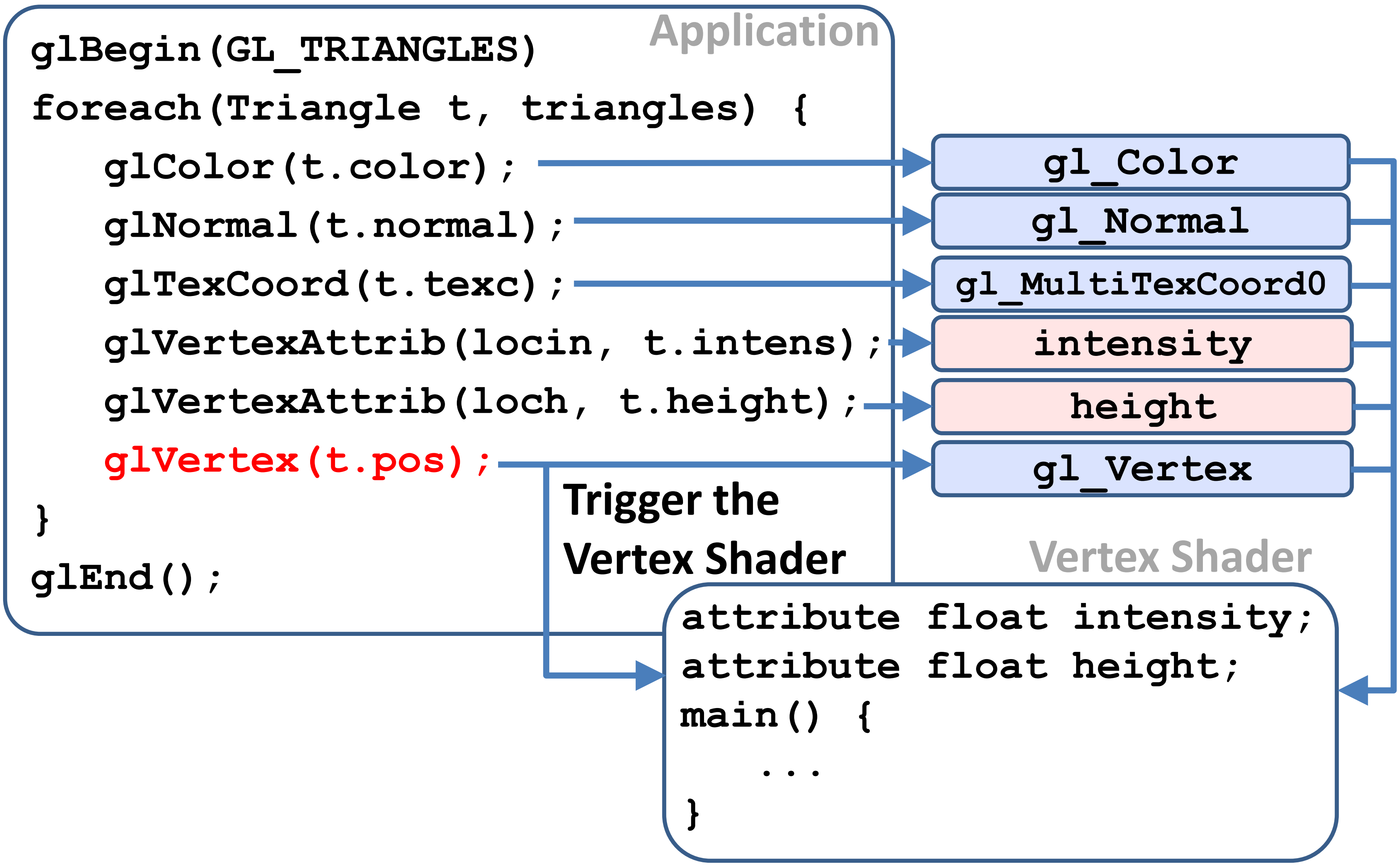
If we want to use TIU-2 we need to write:

```
glMultiTexCoord2f(GL_TEXTURE2, 0.0, 1.0);
```

In the vertex shader however, all texture coordinates are accessible using the attributes:

`gl_MultiTexCoord`*`i`*

Built-In Attributes



Vertex Shader Special Variables

Variables which are written to in the Vertex Shader

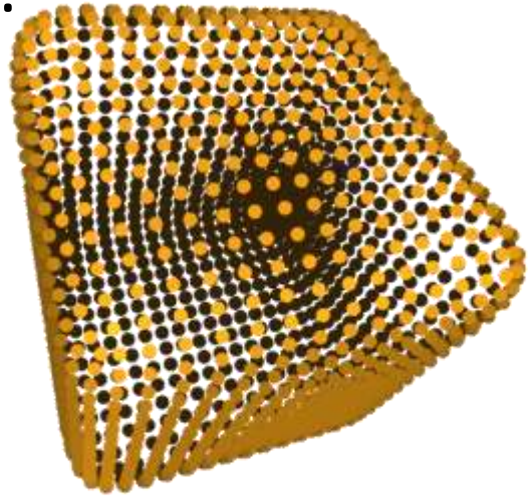
`out vec4 gl_Position`

Write the he homogeneous vertex position after transformations. Must be written to.

`out float gl_PointSize`

Write the size in pixels of the point to be rasterized. relevant with `GL_POINTS`. Need to enable:

`glEnable(GL_VERTEX_PROGRAM_POINT_SIZE)`

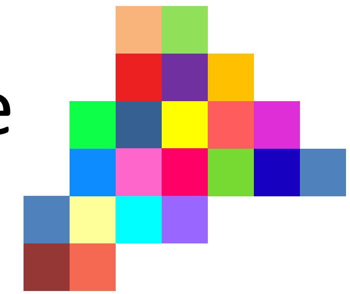


`out vec4 gl_ClipVertex`

Write the reference for user clipping via `glClipPlane()`

Fragment Shader Special Variables

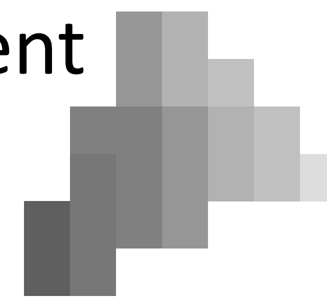
`out vec4 gl_FragColor`



Write the color of the fragment. must be written to unless `discard`-ed

`out float gl_FragDepth`

Optionally write the depth of the fragment



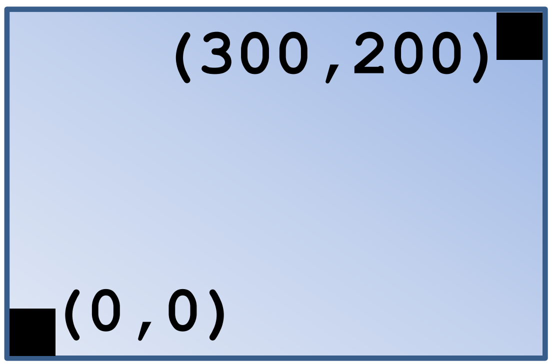
`out vec4 gl_FragData[]`

Alternate output to multiple buffers. Used with `GL_ARB_draw_buffer` extension.

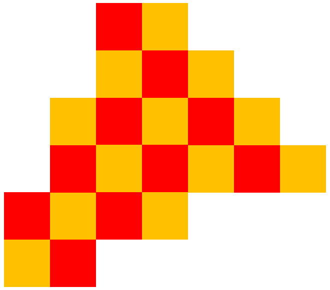
Fragment Shader Special Variables

`in vec4 gl_FragCoord` (read-only)

Contains the screen-coordinates of the fragment. Z contains the fragment depth.



Can be used to create Pixel based effects



`in bool gl_FrontFacing` (read-only)

true if we're in a front-facing polygon. false in back-facing polygons.

OpenGL State Uniforms

Alot of the OpenGL state is accessible to shaders through built-in uniform variables.

```
mat4 gl_ModelViewMatrix;
mat4 gl_ProjectionMatrix;
mat4 gl_ModelViewProjectionMatrix;  // ModelView * Projection
mat3 gl_NormalMatrix;
        // transpose of the inverse of the upper left 3x3 of  ModelView Matrix
        // Used for transforming normals.

mat4 gl_ModelViewMatrixInverse, gl_ModelViewMatrixTranspose,
     gl_ModelViewMatrixInverseTranspose;
mat4 gl_ProjectionMatrixInverse, gl_ProjectionMatrixTranspose,
     gl_ProjectionMatrixInverseTranspose;
mat4 gl_ModelViewProjectionMatrixInverse, gl_ModelViewProjectionMatrixTranspose,
     gl_ModelViewProjectionMatrixInverseTranspose;
```

OpenGL State Uniforms

```
mat4 gl_TextureMatrix[]; // transformations of TIUs

mat4 gl_TextureMatrixInverse[], gl_TextureMatrixTranspose[],
     gl_TextureMatrixInverseTranspose[];

uniform float gl_NormalScale;
                          // related to glEnable(GL_RESCALE_NORMAL)
uniform vec4 gl_ClipPlane[gl_MaxClipPlanes];
                          // user clip planes from glClipPlane()
uniform gl_DepthRangeParameters gl_DepthRange;
                          // from glDepthRange()
uniform gl_PointParameters gl_Point;
                          // from glPointParameter()
```

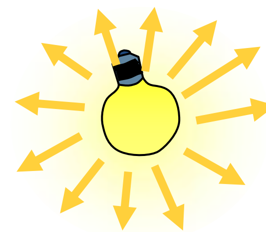
OpenGL State Uniforms - Lighting

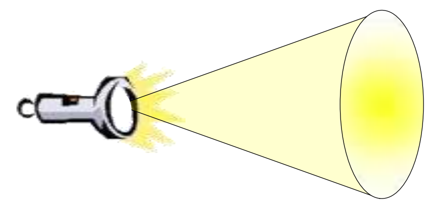
```
struct gl_MaterialParameters {
   vec4 emission;    // Ecm
   vec4 ambient;     // Acm
   vec4 diffuse;     // Dcm
   vec4 specular;    // Scm
   float shininess; // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

These hold the current material set by `glMaterial(…)`

Notice that `GL_COLOR_MATERIAL` doesn't affect these variables since it is in the over-ridden fixed functionality

OpenGL State Uniforms - Lighting





general

spot

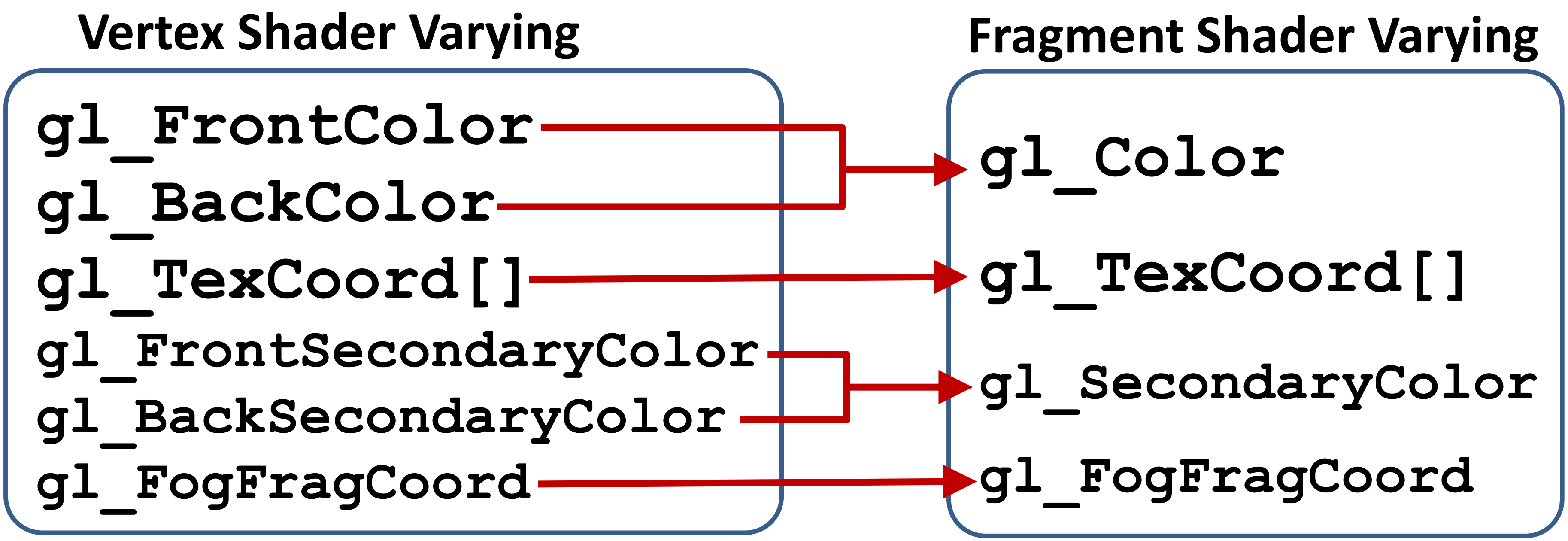
point

```
struct gl_LightSourceParameters {
    vec4 ambient;        // Acli
    vec4 diffuse;        // Dcli
    vec4 specular;       // Scli
    vec4 position;       // Ppli
    vec4 halfVector;  // Derived: Hi
    vec3 spotDirection;  // Sdli
    float spotExponent;  // Srli
    float spotCutoff;    // Crli ([0.0,90.0], 180.0)
    float spotCosCutoff; // cos(Crli)([1.0,0.0],-1.0)
    float constantAttenuation; // K0
    float linearAttenuation;   // K1
    float quadraticAttenuation;// K2
};
uniform gl_LightSourceParameters gl_LightSource[];
```

All Parameters controlled by `glLight(…)`

Built-In Varying

- Built-in varying variables don’t have one-to-one mapping between vertex and fragment shaders



`gl_Color` is an attribute in the vertex shader and a varying variable in the fragment shader.

Built-In Varying

- The built-in varying variables are for the convenience of the programmer.
- Instead of defining a new varying variable, you can use the appropriate built-in one.

Vertex Shader

```
varying vec2 myTexCrd;
void main() {
   gl_Position = ftransform()
   mytexCoord =
       gl_MultiTexCoord0.xy;
}
```

Vertex Shader

```
void main() {
   gl_Position = ftransform()
   gl_TexCoord[0] =
        gl_MultiTexCoord0.xy;
}
```

Equivalent

```
varying vec2 mytexCrd;
uniform sampler2D tex
void main() {
   gl_FragColor =
     texture2D(tex, mytexCrd[0]);
}
```

Fragment Shader

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

Fragment Shader

Fixed Function Interaction

- When only a fragment shader is defined, the Vertex shader remains with the fixed function
- The fixed function assigns values to varying variables for the Fragment shader

Gouraud shading color per vertex → **`gl_Color`**

Texture coordinates (of **`glTexCoord`**) → **`gl_TexCoord[0]`**

Fixed Function Interaction

No Vertex Shader

Fragment Shader:

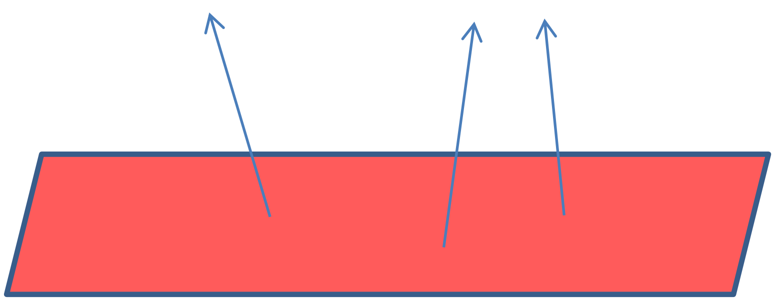
```
void main()
{
   gl_FragColor = vec4(1,1,1,1) - gl_Color;
}
```

Invert the color calculated by Gouraud shading

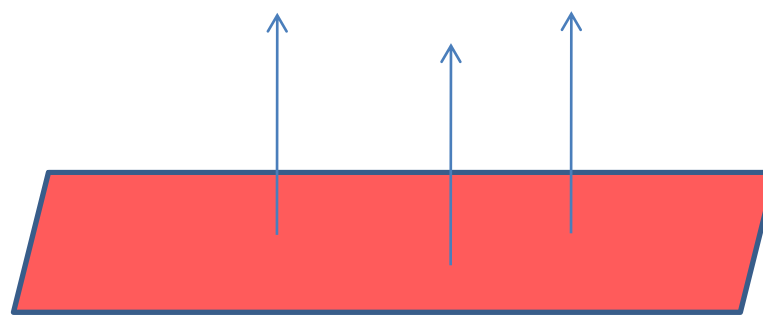
Bump Mapping

- נרצה ליצור אשליה של עומק

ישנה את כיוון הנורמל



ישפיע על הצבע



Bump Mapping

- האלגוריתם פשוט
  - חשב לכל פרגמנט את הכיוון לאור, לצופה ולנורמל
  - שנה את הנורמל בהתאם ל-bump map
  - חשב את התאורה
- באיזה מערכת קואורדינטות נעבוד?
  - Eye space?

Bump Mapping

```
varying vec3lightVec;        varying vec3 eyeVec, halfVec;      varying vec3 normal;          attribute vec3 tangent;
attribute vec3 bitangent;

void main(void)
{
	gl_Position = ftransform();
	gl_FrontColor = gl_Color;
	normal = gl_Normal;

	gl_TexCoord[0] = gl_MultiTexCoord0;

	vec3 vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
	lightVec = gl_LightSource[0].position.xyz - vVertex;
	eyeVec = -vVertex;

	vec3 n = normalize(gl_NormalMatrix * gl_Normal);

	mat3 toVtx = mat3(gl_NormalMatrix * tangent,
				gl_NormalMatrix * bitangent, n);

	lightVec = lightVec * toVtx;
	eyeVec = eyeVec * toVtx;
	halfVec = gl_LightSource[0].halfVector.xyz * toVtx;
}
```

Vertex Shader

Transform to local space (tangent space)

Bump Mapping

```
varying vec3 lightVec;          varying vec3 eyeVec, halfVec;          varying vec3 normal;          uniform sampler2D colorMap;
uniform sampler2D normalMap;

void main (void)
{
	vec3 N = normalize( texture2D(normalMap, gl_TexCoord[0].st).xyz * 2.0 - 1.0);
	vec3 L = normalize(lightVec);
	vec3 H = normalize(halfVec);

	float lambertTerm = max(dot(N,L), 0.0);

	// ** phong
	//vec3 E = normalize(eyeVec);
	//float prod = dot(reflect(L, N), E);
	// ** blinn
	float prod = dot(H, N);

	float specularTerm = pow( max(prod, 0.0), gl_FrontMaterial.shininess );

	vec4 ambient = gl_LightSource[0].ambient;
	vec4 diffuse = gl_LightSource[0].diffuse * lambertTerm;
	vec4 specular = gl_LightSource[0].specular *  gl_FrontMaterial.specular * specularTerm;

	vec4 base = gl_Color * texture2D(colorMap, gl_TexCoord[0].st);

	gl_FragColor = (ambient + diffuse) * base + specular;
}
```

Fragment Shader

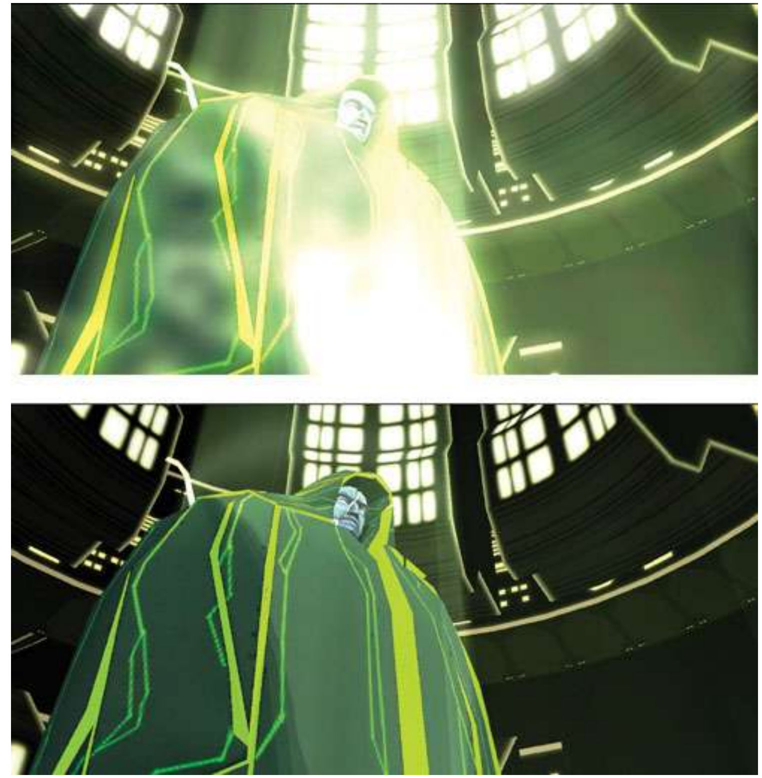


IMAGE PROCESSING IN GLSL

Image Processing via GLSL

Recipe for an Image Processing shader:

- Render a Quad that covers the screen completely.
  - This causes the fragment shader to be called for every single pixel of the screen.
- (optional) Add texture coordinates.
- Write a the Image processing Algorithm in a Fragment Shader
  - Take a texture as an input.

Image Processing via GLSL

Whole screen Quad (One of many options to do this)

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluOrtho2D(-1.0,1.0,-1.0,1.0);

glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);
glDisable(GL_DEPTH_TEST);
glDisable(GL_LIGHTING);

glColor3f(1.0f, 1.0f, 1.0f);
```

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0f, 0.0f
  glVertex2f(-1.0f, -1.0f);
  glTexCoord2f(1.0f, 0.0f);
  glVertex2f(1.0f, -1.0f);
  glTexCoord2f(1.0f, 1.0f);
  glVertex2f(1.0f, 1.0f);
  glTexCoord2f(0.0f, 1.0f);
  glVertex2f(-1.0f, 1.0f);
glEnd();

glEnable(GL_DEPTH_TEST);
glPopMatrix();
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
```

Image Processing Fragment Shader

- Every invocation of the fragment outputs a single pixel of the output image.
- May access the entire input image using a texture sampler
- Where is *my* pixel in the input image?

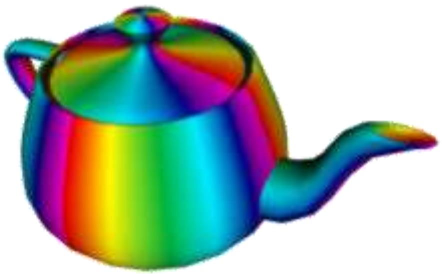
```
texelFetch(input, ivec2(gl_FragCoord.xy),0);
```

```
texture2D(input, gl_TexCoord[0].xy);
```

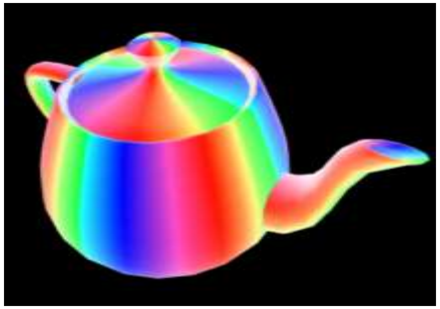
- `gl_FragCoord` is the screen coordinate of the fragment

Image Processing Fragment Shader

A trivial example



```
uniform sampler2D img;
void main()
{
  gl_FragColor = vec4(1,1,1,1) -
    texture2D(img, gl_TexCoord[0].xy);
}
```



```
uniform sampler2D img;
void main()
{
  gl_FragColor = vec4(1,1,1,1) -
    texelFetch(img, ivec2(gl_FragCoord.xy),0);
}
```

SUMMARY

State of the Art

- The origins of programmable shading – Shade Trees (Cook et al) 1984
- Shader Model 3.0 (2004) is widely supported (XBox360, PS3, most PC’s)
- Shader Model 4.0 (2007) available in DirectX 10.0 and OpenGL (via extensions)
  - Geometry Shader
  - Stream Output