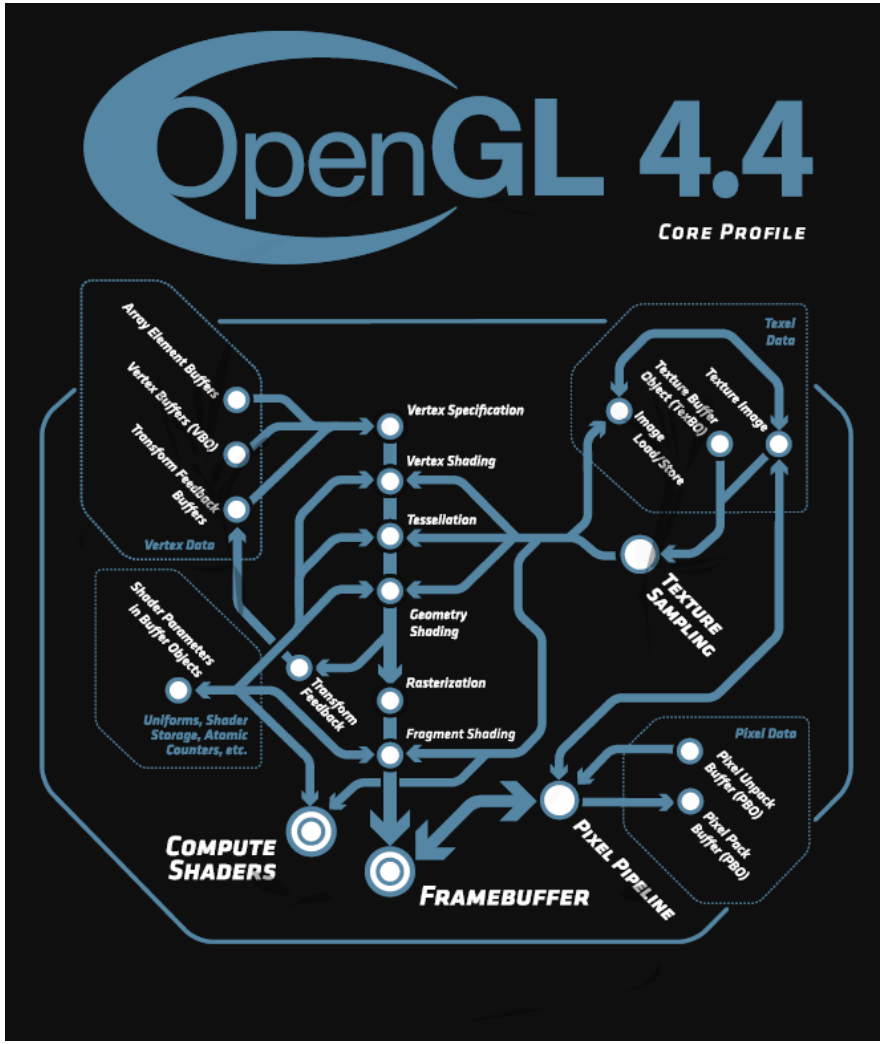
OpenGL Shading Language

2005 James K. Hahn, 2010 Robert Falk, 2013 Wei Li

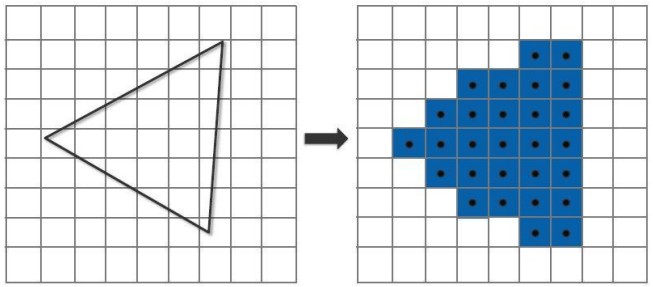
OpenGL Pipeline



Rasterization Pipeline

Attribute :
Position
Color
Normal
Texcoord

Geometry :
Line
Triangle
Polygon



Vertex Specification

Vertex Processor

Primitive Assembly

Clipping and Culling

Rasterization

Fragment Processor

Framebuffer Operation

Framebuffer

Transform :
Model
View
Projection

Per-Fragment Operations :
Scissor
Stencil
Depth Test
Blending

Shader

- Shaders: programs that run on the graphics card
- Two standard shaders
  - Vertex shader - invoked for each vertex
  - Pixel Shader - invoked for each pixel overlapped by each fragment

- Without shaders, OpenGL uses a "fixed functionality" pipeline with default shaders
  - No Phong shading - no per-pixel lighting
  - Special effects difficult or impossible
  - Less efficient

- in → [Fixed Function] → out
- in → [Shader] → [Fixed] → [Shader] →[Fixed] → out

Shader Hardware

- Highly parallel
- Many vertex Shaders
- Even more pixel shaders

Shader Terminology Soup

- Shader Languages
  - GLSL : OpenGL Shading language
  - HLSL : High Level Shading Language - Direct3D
  - CG : Nvidia's first shading language
  - Renderman : The original shading language - for CPUs.
- GPU Graphics Processing Unit
- GPGPU General purpose Graphics Processing Unit
- GPGPU programming languages
  - CUDA: Nvidia's parallel computing architecture
  - OpenCL: Open Computing Language
  - Direct Compute: Microsoft's parallel computing language

OpenGL Shading Language - GLSL

- You can have Vertex Shader and Pixel Shader
  - Even more : Geometry Shader, Tesslelation Shader, Compute Shader, etc.
- C-like syntax (no pointers or strings)
- Each shader has a single main() function
- Language focused on numerical computation
  - Vectors and matrices
  - Math library functions
  - Mechanism to communicate with main program
    - CPU → Shader
  - Can pass data from Vertex to Fragment shader
    - Vertex Shader → Fragment Shader

Vertex Shader

- Input
  - **Vertex Attributes (per vertex data)**
    - Built-in
      - glVertex → gl_Vertex
      - glNormal → gl_Normal
      - glColor → gl_Color
      - …
    - Custom
      - glVertexAttrib → **attribute**
  - **Uniforms (global, per program data)**
    - Built-in
      - glMatrixMode → gl_ModelViewMatrix
      - glLight → gl_LightSource[n]
      - …
    - Custom
      - glUniform → **uniform**
- Output
  - **Varyings**
    - Built-in
      - gl_Position
      - gl_TexCoord[n]
      - …
    - Custom
      - **varying**

Fragment Shader

- Input
  - **Uniforms**
    - Textures → sampler2D
  - **Varying**
    - Built-in
      - gl_FragCoord
    - Custom
      - **varying**
- Output
  - gl_FragColor
  - gl_FragDepth
  - gl_FragData[n]

Scalar

- float // single precision floating point
- int // 32 bit signed integer
- uint // 32 bit unsigned integer
- bool // bool is true/false, not 1/0.

- float a; //uninitialized
- float b = 1.0; //initialized to 1.0
- int c = int(b); //use explicit conversion.

Vector

- vec{2,3,4} floating-point
- ivec{2,3,4} integer vectors
- uvec{2,3,4} unsigned integers
- bvec{2,3,4} boolean vectors

- **Vectors: Constructors**
  - vec3 a = vec3(3.2, 1.5, 2.0);
  - vec3 b = a;
  - ivec3 c = ivec3(1); //sets all elements to 1.
  - vec3 d = vec2(b); // uses first 2 values of b
  - vec4 v = vec4(d,1.0); // mix smaller vectors and floats

- **Vectors: Element access**
  - float r = v[0]; // Array style
  - float g = v.g; // by name: xyzw, rgba, and stpq
  - vec2 s1 = v.gr; // Swizzling: Get any combination
  - vec3 s2 = v.bgr; // can change the order
  - vec4 s3 = v.xxzz; // can repeat elements
  - vec4 s4 = v.xyba // No! Can't mix xyzw/rgba/stpq

Matrix

- mat2; // 2x2 matrix
- mat3; // 3x3 matrix
- mat4; // 4x4 matrix
- mat{mxn} m; // **colsxrows**
  - {mxn,2x3,2x4,3x2,3x4,4x2,4x3}

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

- **Constructors**
  - **Initialize in column-major order:**
  - mat3 m1 = mat3(1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0);
  - mat3 m2 = mat3(1.0); **// set diagonal to 1.0, rest to 0**

  - vec4 a,b,c,d;
  - mat4 m3 = mat4(a,b,c,d); // set columns to vecs abcd.
  - mat4 m4 = mat4(m1); // upper == m1, rest 0
  - mat3x4 = mat3x4(a,b,c); // Only 3 columns

- **Element Access**
  - float a = m1[1][2]; // [column][row] (a==6.0)
  - vec3 v = m1[0]; // v=first column=(1.0,2.0,3.0)

Array

- **Arrays**
  - float a[22]; // array of float
  - vec3 b[];   // define b's size later
  - vec3 b[5]; // b is now size 5 (can't change again)

  - float c[5] = float[5](1.0, 2.0, 3.0, 4.0, 5.0);
  - int len = c.length(); // can get the size of an array

- **Structures**

```
// Much like in "C":
struct LightInfo
{
    vec3 color;
    float intensity;
};
LightInfo l[3];
```

Function

- **functions**
  - can overload by parameters
  - parms from caller must match exactly - no conversions (like int to float)
  - no recursive calls

- **parameter qualifiers**
  - **in**, or no qualifier : Call-by-value (value, if updated, is not returned)
  - **Out** : Undefined when function starts. Copied out
  - **Inout** : Defined at function start, copied out

```
bool diffuse(vec3 norm,   in vec3 light,  out float d,  in float k) {
        d = k*dot(norm, light);
        if (d>0.0) {
                return true;
        }
        return false;
}
```

Built-in Functions

- **Many functions overloaded to take float, vec2, vec3, vec4 (so T here)**
- **math functions**
  - T = radians(T degrees), degrees(T radians)
  - T = sin|cos|tan|asin|acos(T radians) , atan(T y, T x), atan(T y_over_x)
  - T = pow(T x, T y) // x^y
  - T = exp(T x) // e^x
  - T = log(T x)
  - T = sqrt(T x)
  - T = abs(T x) // absolute value of each element in x
  - T = sign(T x) // (x>0)→ 1.0 (x==0)→0.0 (x<0.0)→-1.0
  - T = mix(T x, T y, float a) // x*(1.0-a) + y*a, e.g. linear interpolation
  - T = step(T edge, T x) // (x<edge) →0.0, (otherwise) → 1.0
  - Lots more: (various min, max, floor, ceil, clamp, mod)

Geometry Function

- **vector functions**
  - float length(T x) // sqrt(x[0]*x[0]+..+x[n]*x[n])
  - float distance(T x, T y) // length(x-y)
  - float dot(T x, T y) // x[0]*y[0]+ x[1]*y[1]+..+x[n]*y[n]
  - vec3 cross(vec3 x, vec3 y) // cross product of x and y
  - T normalize(T x) // vector in direction 'x' with length 1.0
  - T reflect(T I, T n) // reflection vector
  - T refract(T I, T n, float eta) // refraction vector

- **matrix functions (T here can be mat2, mat3, or mat4)**
  - T transpose(T m) // transpose of matrix 'm'
  - T inverse(T m) // inverse of 'm'

GLSL Hello World

- **Vertex Shader**

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

- **Fragment Shader**

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

- **API**

```
glVertex3f( … );
```

Use Uniforms

- **Vertex Shader**

```
uniform vec4 color;

void main( void ) {
        gl_Position = ftransform();
}
```

- **Fragment Shader**

```
uniform vec4 color;

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

- **API**

glVertex3f(…);

float g_color[4] = { 1.0, 1.0, 1.0, 1.0 }; ←

location = glGetUniformLocation(program, “color”); ←
glUniform4fv(program, location, 1, g_color); ←

Use Varyings

- **Vertex Shader**

```
varying vec4 color;

void main( void ) {
        color = gl_Color;
        gl_Position = ftransform();
}
```

- **Fragment Shader**

```
varying vec4 color;

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

- **API**

  glColor3f(…); ←

  glVertex3f(…);

Use Texture

- **Vertex Shader**

```
varying vec2 tc;

void main( void ) {
        tc = gl_MultiTexCoord0.xy;
        gl_Position = ftransform();
}
```

- **Fragment Shader**

```
uniform sampler2D tex;

varying vec2 tc;

void main( void ) {
        gl_FragColor = texture2D( tex, tc );
}
```

- **API**

```
loc = glGetUniformLocation( program, “tex” );
glUniform1i( program, loc, 0 );

glActiveTexture( GL_TEXTURE0 );
glEnable( GL_TEXTURE_2D );
glBindTexture( … );

glTexCoord2f( … );
glVertex3f( … );
```

Gouraud Shading : Diffuse

- **Vertex Shader**

```
// per vertex intensity
varying vec4 intensity;

void main( void ) {
        // vertex position in view space
        vec3 vp = ( gl_ModelViewMatrix * gl_Vertex ).xyz;

        // vertex normal in view space
        vec3 vn = gl_NormalMatrix * gl_Normal;

        // light position in view space
        vec3 lp = gl_LightSource[0].position.xyz;

        // direction from vertex to light in view space
        vec3 vl = normalize( lp – vp );

        // diffuse intensity
        float diffuse = dot( vl, vn );
        diffuse = max( diffuse, 0.0 );

        // final intensity
        intensity = diffuse * gl_FrontLightProduct[0].diffuse;

        gl_Position = ftransform();
}
```

- **Fragment Shader**

```
varying vec4 intensity;

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

Phong Shading : Diffuse

- **Vertex Shader**

```
// per vertex intensity
varying vec3 vn;
varying vec3 vl;

void main( void ) {
        // vertex position in view space
        vec3 vp = ( gl_ModelViewMatrix * gl_Vertex ).xyz;

        // light position in view space
        vec3 lp = gl_LightSource[0].position.xyz;

        // direction from vertex to light in view space
        vl = lp – vp;

        // vertex normal in view space
        vn = gl_NormalMatrix * gl_Normal;

        gl_Position = ftransform();
}
```

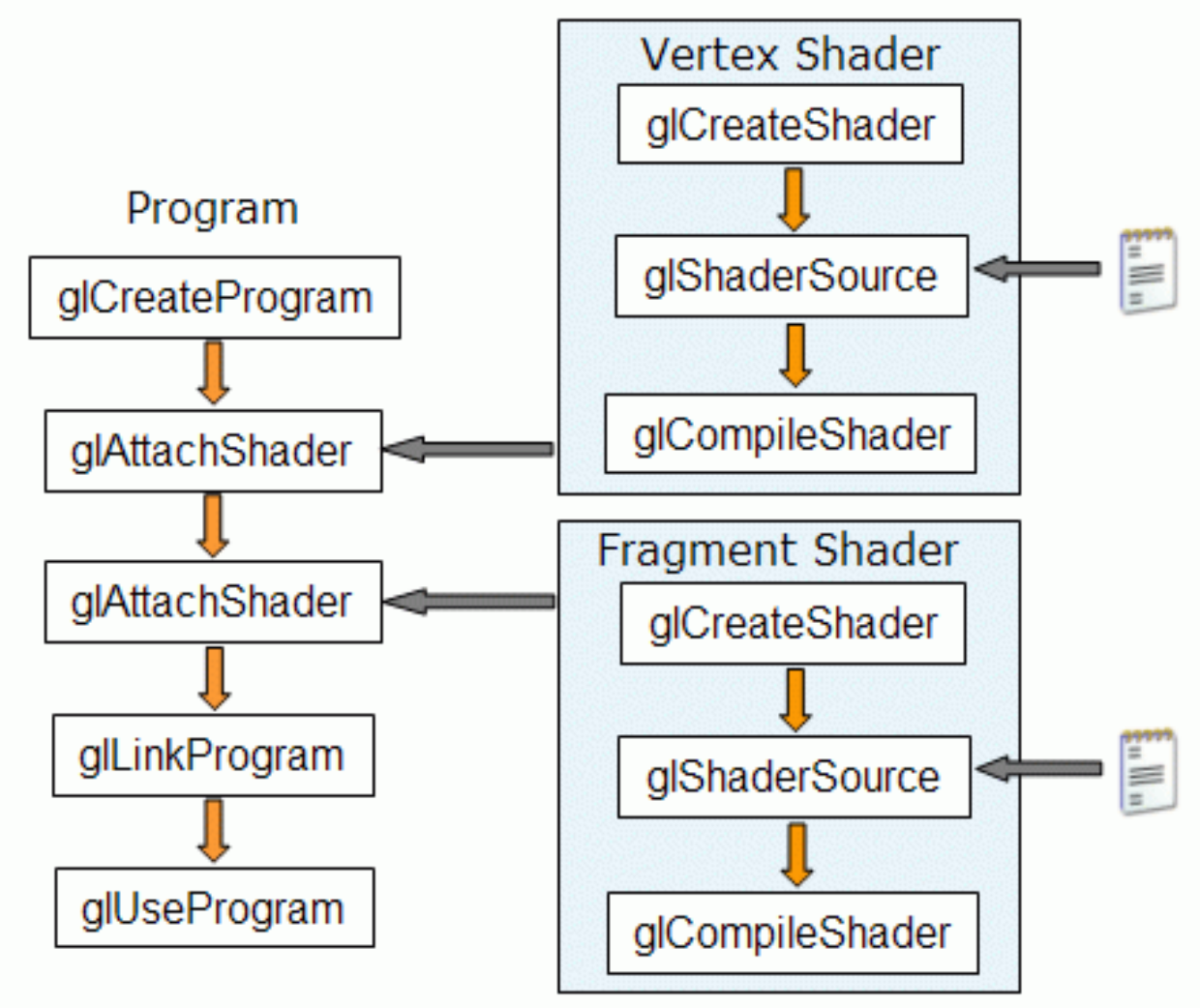
- **Fragment Shader**

```
varying vec3 vn;
varying vec3 vl;

void main( void ) {
  // diffuse intensity
  float diffuse = dot( normalize(vl), normalize(vn) );
  diffuse = max( diffuse, 0.0 );

  // final intensity
  vec4 intensity = diffuse *
  gl_FrontLightProduct[0].diffuse;

  gl_FragColor = intensity;
}
```

OpenGL 2.0 : Extensions

```
#include <GL/glew.h>
#include <GL/glut.h>

void main(int argc, char **argv) {
        glutInit(&argc, argv);
        ...
        glewInit();

        if ( !GLEW_ARB_vertex_shader || !GLEW_ARB_fragment_shader ) {
                printf("Not totally ready for GLSL\n");
                exit(1);
        }

        if ( !glewIsSupported("GL_VERSION_2_0") ) {
                printf("OpenGL 2.0 not supported\n");
                exit(1);
        }

        ... init opengl states, load shaders, textures, and so on ...

        glutMainLoop();
}
```

Load and Compile Shader Program



Creating a Shader

- // Create Shader
- GLuint **shader** = glCreateShader(**shaderType**);
  - // GL_VERTEX_SHADER
  - // GL_FRAGMENT_SHADER

- // Shader Source
- glShaderSource(**shader**, 1, (const char**)&text, NULL);

- // Compile Shader
- glCompileShader(**shader**);

- // Check Status
- int success = 0;
- glGetShaderiv(**shader**, GL_COMPILE_STATUS, &success);

- if (!success) {
  const int MAX_LENGTH = 1024;
  char info[MAX_LENGTH];
  glGetShaderInfoLog(**shader**, MAX_LENGTH, NULL, info);
  }

Creating a Program

```
// Create Program
GLuint program = glCreateProgram();

// Attach Shaders
glAttachShader( program, vert_shader );
glAttachShader( program, frag_shader );

// Link Program
glLinkProgram( program );

// Check Status
int success = 0;
glGetProgramiv( program, GL_COMPILE_STATUS, &success );

if ( !success ) {
        const int MAX_LENGTH = 1024;
        char info[ MAX_LENGTH ];
        glGetProgramInfoLog( program, MAX_LENGTH, NULL, info );
}

// Use Program
glUseProgram( program );
... render something...
glUseProgram( 0 );
```

Resource

- **Lighthouse 3D**
- http://www.lighthouse3d.com/tutorials/glsl-tutorial/

- **Clockworkcoders Tutorials**
- http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/index.php

- **GLEW**
- http://glew.sourceforge.net/index.html

- **GLSL Specification**
- http://www.opengl.org/documentation/glsl/