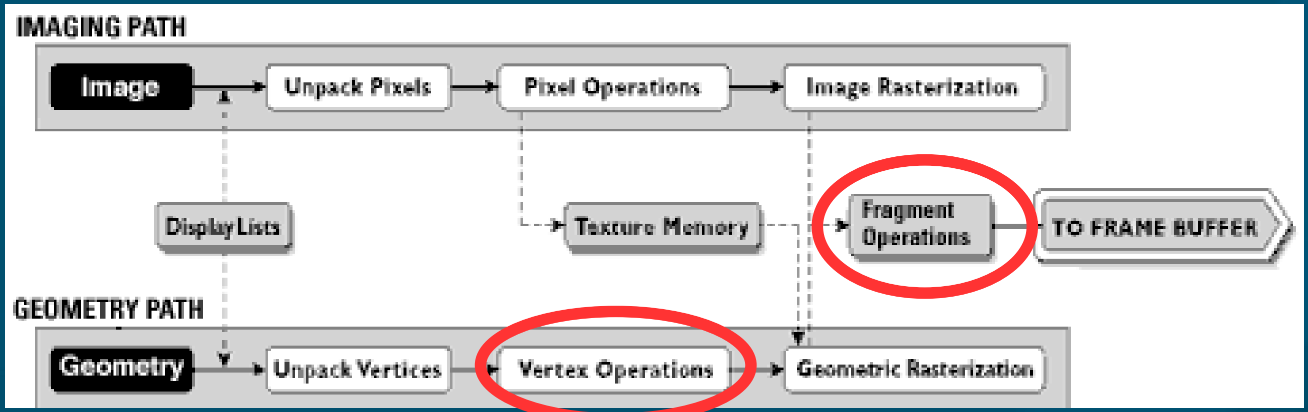
GLSL: OpenGL Shading Language

See: Edward Angel's text, "Interactive Computer Graphics"

24 March 2009
CMPT370
Dr. Sean Ho
Trinity Western University

Review of rendering pipeline

- OpenGL rendering pipeline:



- Vertex operations:
  - Transform points via model-view matrix
  - Normals, other per-vertex data
- Fragment operations:
  - Shading: colour for each pixel of fragment

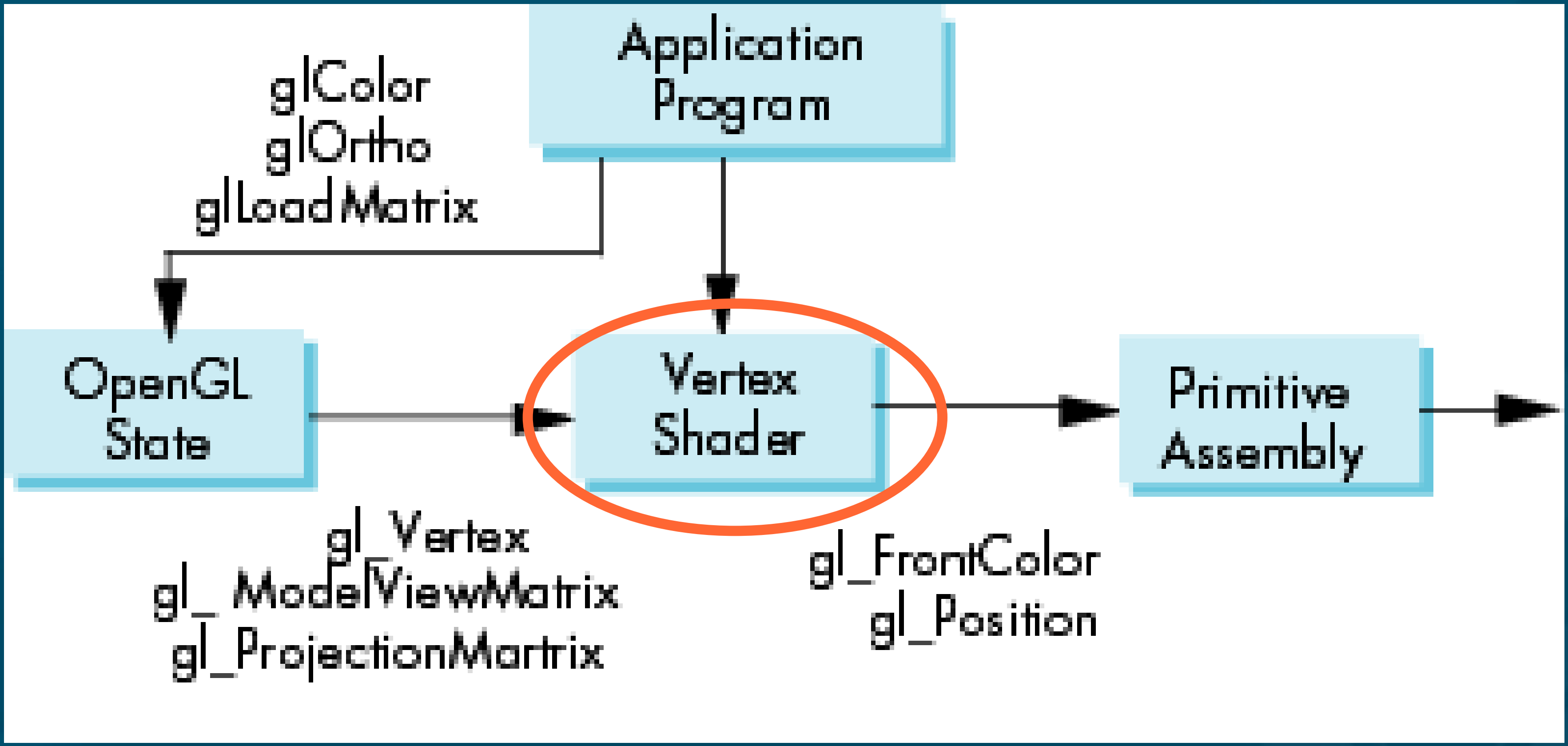
GLSL: shading language

- High-level C-like language for programming vertex shaders and fragment shaders
  - Separate language, separate files (*.glsl)
  - Compile, link, use GLSL shader programs via GL function calls in main program
- Built-in data types: bool, int, float, but also vectors, matrices, and samplers (for textures)
  - Common vec/matrix operations
- OpenGL state available through built-in vars
- Output by writing to specific built-in vars

Simplest vertex shader

- Pass-through vertex shader: vPassthrough.glsl
  - void main() {
    gl_Position =
    gl_ModelViewProjectionMatrix * gl_Vertex;
  - // equiv: = ftransform(gl_Vertex);
  - }
- Input: gl_Vertex
- GL state: gl_ModelViewProjectionMatrix
  (= gl_ProjectionMatrix * gl_ModelViewMatrix)
- Output: gl_Position (screen coordinates)
  - gl_FrontColor (will be interpolated)

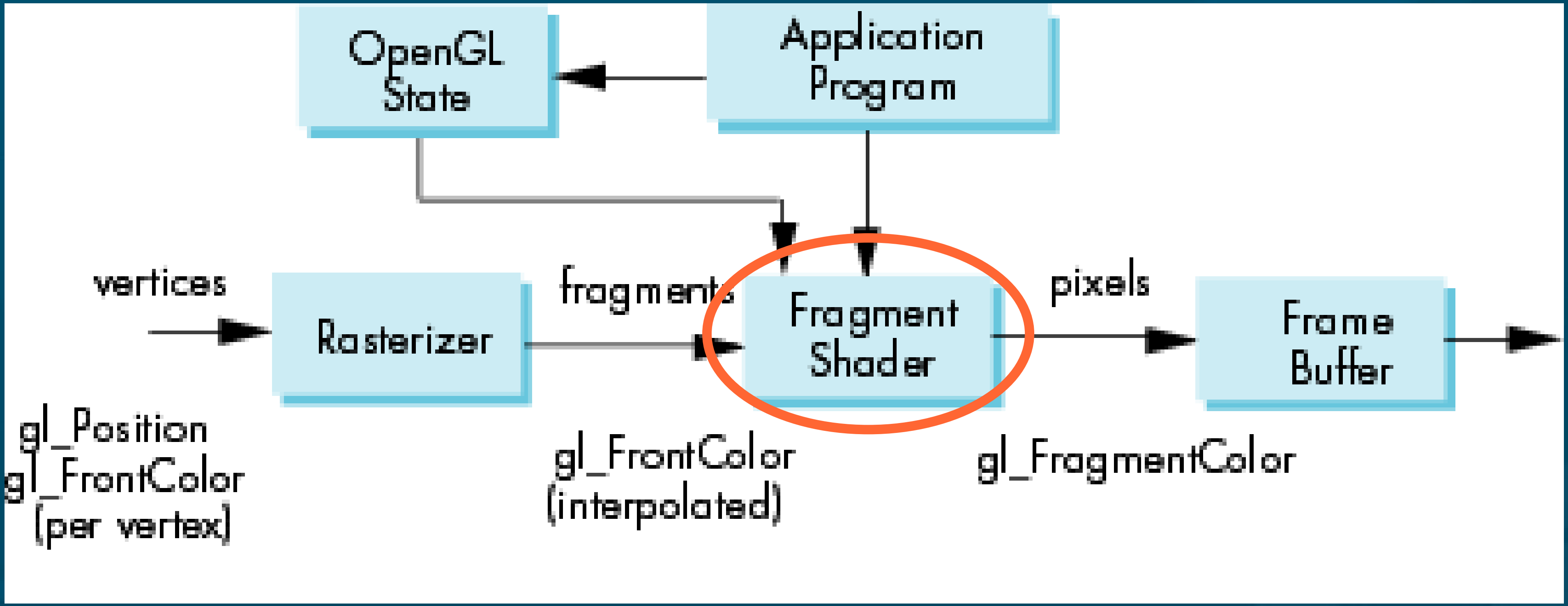
Execution model: vertex shader



Simplest fragment shader

- Pass-through fragment shader: fPassthrough.glsl
  - void main() {
    gl_FragColor = gl_Color;
  - }
- Input: gl_Color (interpolated across the polygon)
- GL state: gl_LightSource[], gl_FrontMaterial, etc.
- Output: gl_FragColor
  - still subject to blending and hidden-surface removal

Exec. model: fragment shader



A note on GLEW

- GLSL is relatively new
- It is a part of OpenGL as of OpenGL 2.1
  - But not many vendors support 2.1
- Prior to being a standard part of OpenGL, it was an OpenGL extension
- GLEW: GL Extension Wrangler library
  - Allows you to use GLSL without worrying about whether it's an extension or not
  - #include “glew.h”, call glewInit() early on, and link with glew.o or -lGLEW

Compiling and using shaders

- Host application program does this (see particle.c):
  - Read *.glsl source into string buffer (null-term)
  - Create shader program object
    - vSh = glCreateShader(GL_VERTEX_SHADER);
    - prog = glCreateProgram();
    - glAttachShader(prog, vSh);
  - Load source code, compile, link:
    - glShaderSource(..., stringBuf, ...);
    - glCompileShader(vSh);
    - glLinkProgram(prog);
    - glUseProgram(prog);

GLSL data types

- Regular scalar types as in C: bool, int, float
- Vectors: float: vec2, vec3, vec4
  - Also int (ivec2, etc.), bool (bvec2, etc.)
- Matrices (float): mat2, mat3, mat4
  - Standard indexing: myMat[row][col]
  - Internally linearized in column-major order
- C++-style constructors:
  - vec3 a = vec3(1.0, 2.0, 3.0);
- No pointers

Vector swizzling

- GLSL has a shorthand for accessing vectors:
  - vec3 velocity;
  - velocity.x = 2.0;
  - Use (x, y, z, w) or (r, g, b, a) or (s, t, p, q)
  - vec.x == vec.r == vec.s == vec[0]
- Swizzling components:
  - velocity.xz = vec2(1.0, 3.0);

GLSL built-in functions

- Math operators are overloaded for vec/mat:
  - myMat4 * yrMat4 * myVec3
- Common math: abs, min/max, pow, exp, sqrt
- Trig: sin/cos/tan, radians, degrees
- Geometric: cross, dot, distance, length, normalize
- Several more; see GLSL quick-ref PDF

Subroutines in GLSL

- Besides the main() in each GLSL shader, you can write helper subroutines called by main()
- Vectors/matrices are first-class data types in GLSL: can pass and return from subroutines
  - mat4 transpose(mat4 m) { … }
- Call by value-return: qualify parameters with:
  - in (read-only, copy in) (default)
  - out (write-only, copy out)
  - inout (copy in, copy out)
    - void transpose(inout mat4 m) { … }

Variables in GLSL

- Variables may change
  - Not at all (const) (compile-time constant)
  - Once per frame (each redraw) (e.g., time)
  - Once per primitive (uniform) (e.g. texture)
  - Once per vertex (attribute) (e.g., normal)
  - Once per fragment (varying) (e.g shading)
- These variables are declared in the global scope of the GLSL shader code
- May also declare local variables inside your GLSL functions

Uniform variables (per-primitive)

- uniform float time;
- void main() { .... }

- Constant across whole primitive
- Host app sets outside of glBegin()/glEnd()
- Read-only for the shader
- Both vertex and fragment shaders can access
- e.g., pass bounding box of primitive
- Some GL state is uniform:
  - gl_ModelViewMatrix, gl_LightModel, etc.

e.g.: wave motion vertex shader

- Modulate vertex position by sinusoid:
  - uniform float time;
  - void main() {
    vec4 t = gl_Vertex;
    t.y = 0.1*sin(0.001*time + 5.0*t.x) + 0.1*sin(0.001*time + 5.0*t.z);
    gl_Position = gl_ModelViewProjectionMatrix * t; }
- Parent program sets the uniform float time
- Modulate y-coord by time, x, z coords

Attribute variables (per-vertex)

- attribute float temperature;
- void main() { … }

- May take on different values for each vertex
  - e.g., temperature differs at each vertex
- Only accessible by vertex shader, not fragment
- Read-only for the shader
- Some GL state is per-vertex attribute:
  - gl_Vertex, gl_Normal, gl_Color

e.g.: gravity particle system

- Input velocity vector per-vertex:
  - attribute vec2 velocity;
  - uniform float time;
  - void main() { … }
- See vparticle.glsl
- Read-only variables:
  - time (global)
  - velocity (per-vertex)

Varying: from vert->frag

- varying vec3 N, L;
- void main() { … }

- Declare, use in both vertex and frag shaders
- Way to pass data from vert to frag shaders
- Vertex shader sets one value per vertex
- Rasterizer interpolates values across primitive
- Fragment shader can read variable per-fragment
- e.g.: Phong shading: calc N, L vectors per-vert, calc lighting model per-fragment

e.g.: Phong lighting

- Let's use vertex+fragment shaders to implement per-pixel Phong shading
  - Default is per-vertex Gouraud lighting
- (Shade) = (Ambient) + (Diffuse) + (Specular)
  - $I = k_a I_a + k_d I_d (l * n) + k_s I_s (v * r)^\alpha$
- Need vectors l (to light) and r (reflection)
- Use vertex shader to calculate l, n
  - Rasterizer will interpolate these vectors
- Use fragment shader to calc r and do Phong shading

Vertex shader program

```
varying vec3 N, L;
void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    N = gl_NormalMatrix * gl_Normal;
    L = gl_LightSource[0].position.xyz;
    gl_FrontColor = vec4(0.5, 0.5, 0.8, 1.0);
}
```

- Input: gl_Vertex, gl_Normal
- Output: gl_Position (eye coords), N, L (send to fragment shader)