Supplement to Lecture 22

Programmable GPUs

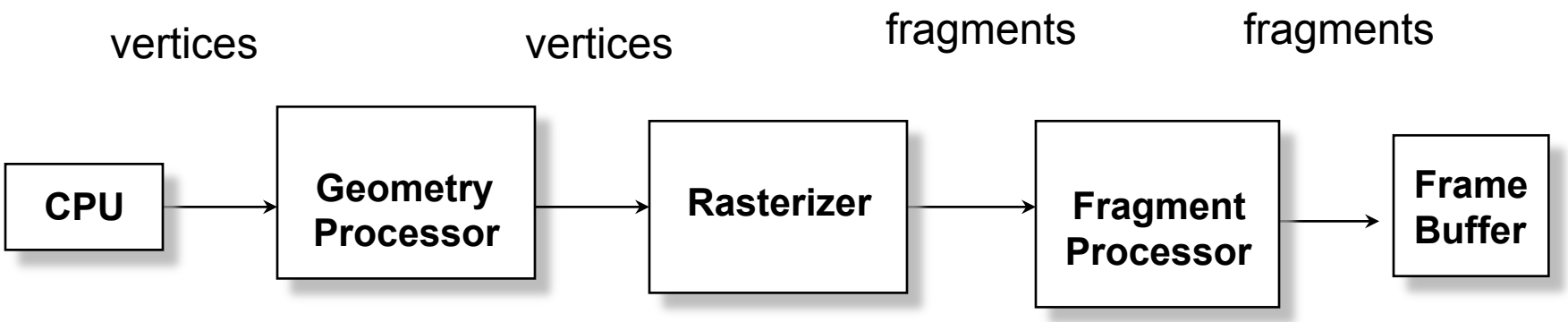
Programmable Pipelines

- Introduce programmable pipelines
  - Vertex shaders
  - Fragment shaders
- Introduce shading languages
  - Needed to describe shaders
  - RenderMan

Preliminaries

- Recent major advance in real time graphics is programmable pipeline
  - First introduced by NVIDIA GForce 3
  - Supported by high-end commodity cards
    - NVIDIA, ATI, 3D Labs
  - Software Support
    - Direct X 8 , 9, 10
    - OpenGL Extensions
    - OpenGL Shading Language (GLSL)
    - Cg
- Two components
  - Vertex programs (shaders)
  - Fragment programs (shaders)
- Requires detailed understanding of two seemingly contradictory apporachs
  - OpenGL pipeline
    - Real time
  - RenderMan ideas
    - offline

Pipeline


vertices
vertices
fragments
fragments
CPU
Geometry Processor
Rasterizer
Fragment Processor
Frame Buffer

Geometric Calculations

- Geometric data: set of vertices + type
  - Can come from program, evaluator, display list
  - type: point, line, polygon
  - Vertex data can be
    - (x,y,z,w) coordinates of a vertex (glVertex)
    - Normal vector
    - Texture Coordinates
    - RGBA color
    - Other data: color indices, edge flags
    - Additional user-defined data in GLSL

Per Vertex Operations

- Vertex locations are transformed by the model-view matrix into eye coordinates
- Normals must be transformed with the inverse transpose of the model-view matrix so that $v \cdot n = v' \cdot n'$ in both spaces
  - Assumes there is no scaling
  - May have to use autonormalization
- Textures coordinates are generated if autotexture enabled and the texture matrix is applied

Vertex Processor

- Takes in vertices
  - Position attribute
  - Possibly color
  - OpenGL state
- Produces
  - Position in clip coordinates
  - Vertex color

Vertex Objects -> Fragments

- Vertices are next assembled into objects
  - Polygons
  - Line Segments
  - Points
- Transformation by projection matrix
- Clipping
  - Against user defined planes
  - View volume, x=±w, y=±w, z=±w
  - Polygon clipping can create new vertices
- Perspective Division
- Viewport mapping

- Geometric objects are rasterized into **fragments**
- Each fragment corresponds to a point on an integer grid: a displayed pixel
- Hence each fragment is a *potential pixel*
- Each fragment has
  - A color
  - Possibly a depth value
  - Texture coordinates

Fragment Processor

- Takes in output of rasterizer (fragments)
  - Vertex values have been interpolated over primitive by rasterizer
- Outputs a fragment
  - Color
  - Texture
- Fragments still go through fragment tests
  - Hidden-surface removal
  - alpha

Fragment Operations

- Texture generation
- Fog
- Antialiasing
- Scissoring
- Alpha test
- Blending
- Dithering
- Logical Operation
- Masking

Programmable Shaders

- Replace fixed function vertex and fragment processing by programmable processors called **shaders**
- Can replace either or both
- If we use a programmable shader we must do *all* required functions of the fixed function processor

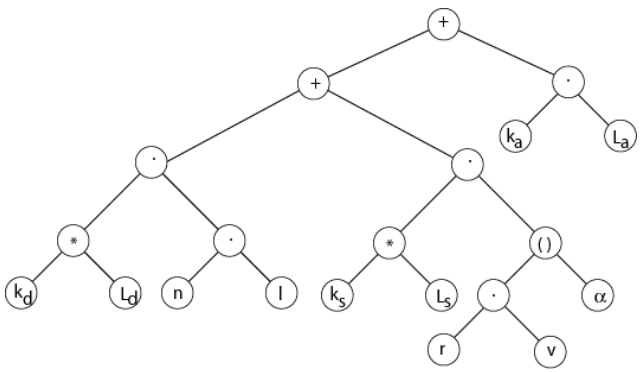
Modeling, Rendering, Shaders

- Modeler outputs geometric model plus information for the renderer
  - Specifications of camera
  - Materials
  - Lights
- May have different kinds of renderers
  - Ray tracer
  - Radiosity
- How do we specify a shader?

- Shaders such as the Phong model can be written as algebraic expressions

$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^s + k_a I_a$$

- But expressions can be described by trees
- Need now operators such as dot and cross products and new data types such as matrices and vectors
- Environmental variables are part of state



Fragment Shader Applications

Per fragment lighting calculations

per vertex lighting

per fragment lighting

Vertex Shader for Per Fragment Lighting

```
varying vec3 N, L, E, H;

void main()
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

   vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
   vec4 eyeLightPos = gl_LightSource[0].position;
   N = normalize(gl_NormalMatrix * gl_Normal);
   L = normalize(eyeLightPos.xyz - eyePosition.xyz);
   E = -normalize(eyePosition.xyz);
   H = normalize(L + E);
}
```

Fragment Shader for Phong Lighting I

```
varying vec3 N;
varying vec3 L;
varying vec3 E;
varying vec3 H;

void main()
{
    vec3 Normal = normalize(N);
    vec3 Light  = normalize(L);
    vec3 Eye    = normalize(E);
    vec3 Half   = normalize(H);
```

Fragment Shader for Phong Lighting II

```
    float Kd = max(dot(Normal, Light), 0.0);
    float Ks = pow(max(dot(Half, Normal), 0.0),
                   gl_FrontMaterial.shininess);
    float Ka = 0.0;

    vec4 diffuse  = Kd * gl_FrontLightProduct[0].diffuse;
    vec4 specular = Ks * gl_FrontLightProduct[0].specular;
    vec4 ambient  = Ka * gl_FrontLightProduct[0].ambient;

    gl_FragColor = ambient + diffuse + specular;
}
```

Writing Shaders

- First programmable shaders were programmed in an assembly-like manner
- OpenGL extensions added for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
- OpenGL Shading Language (GLSL)

GLSL

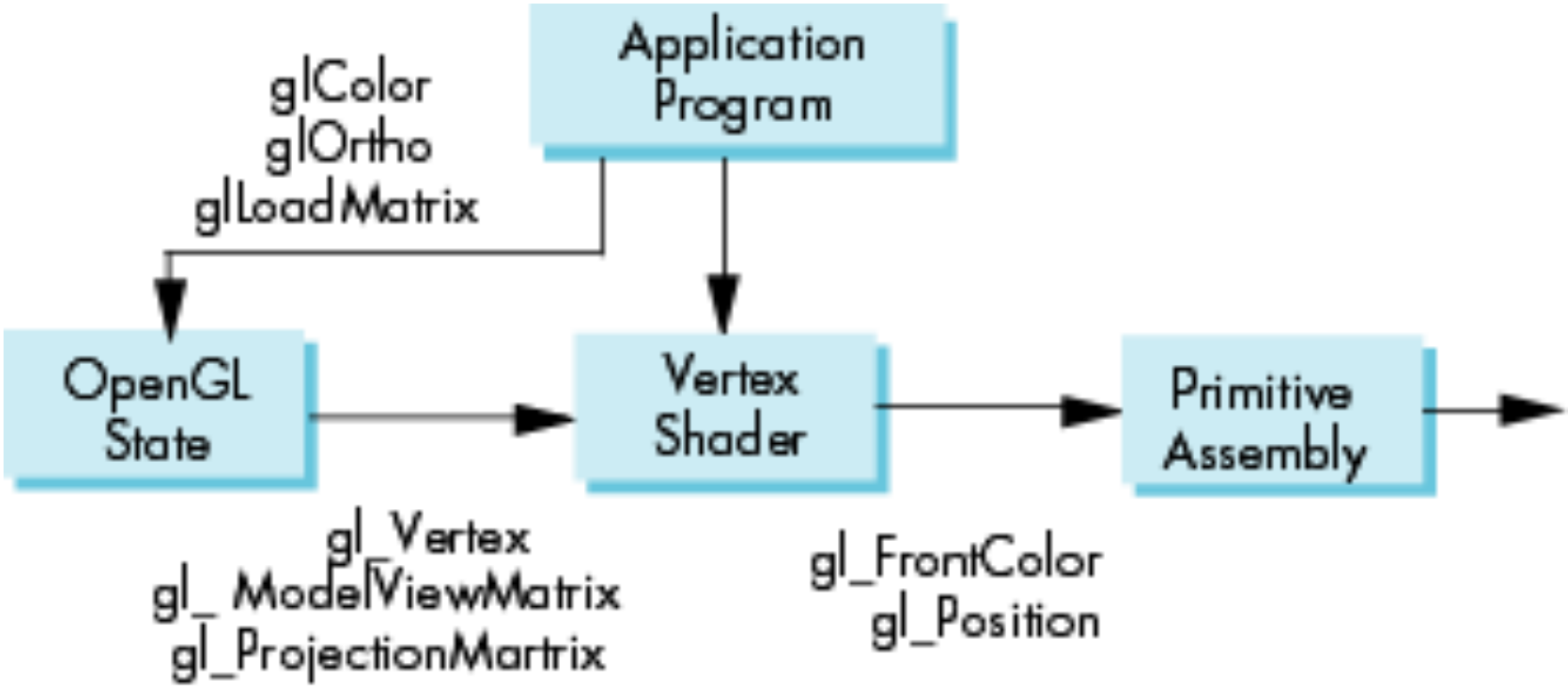
- OpenGL Shading Language
- Part of OpenGL 2.0
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- OpenGL state available through built-in variables

Single Vertex Shader

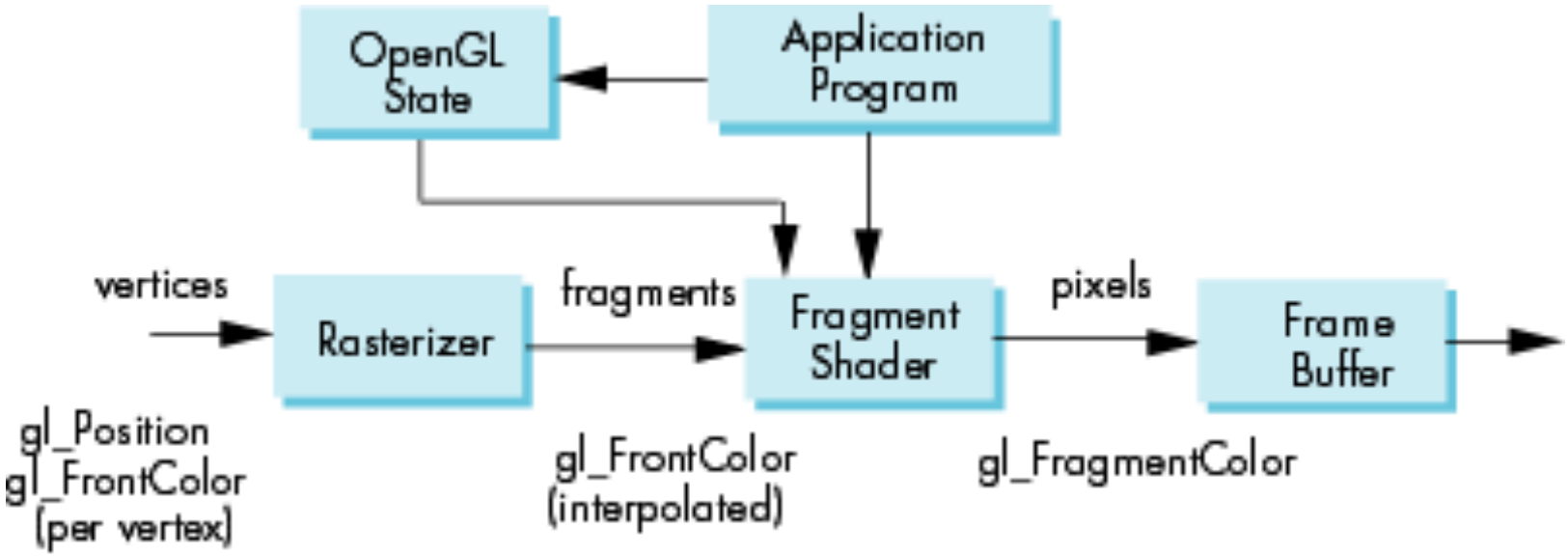
```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main(void)
{
    gl_Position = gl_ProjectionMatrix
        *gl_ModelViewMartrix*gl_Vertex;

    gl_FrontColor = red;
}
```

Execution Model



Simple Fragment Program



Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec 3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a =vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

Pointers

- There are no pointers in GLSL
- We can use C structs which

  can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

  matrix3 func(matrix3 a)

Qualifiers

- GLSL has many of the same qualifiers such as **`const`** as C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
  - Cannot be used in fragment shaders
- Built in (OpenGL state variables)
  - `gl_Color`
  - `gl_ModelViewMatrix`
- User defined (in application program)
  - `attribute float temperature`
  - `attribute vec3 velocity`

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application outside scope of **`glBegin`** and **`glEnd`**
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- Built in
  - Vertex colors
  - Texture coordinates
- User defined
  - Requires a user defined fragment shader

Required Fragment Shader

```
varying vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
```

Passing Values

- call by **value-return**
- Variables are copied in
- Returned values are copied back
- Three possibilities
  - **in**
  - **out**
  - **inout**

Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
varying vec3 color_out;
void main(void)
{
  gl_Position =
   gl_ModelViewProjectionMatrix*gl_Vertex;
  color_out = red;
}
```

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

Swizzling & Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - `a[2], a.b, a.z, a.p` are the same
- **Swizzling** operator lets us manipulate components

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

Linking Shaders to OpenGL

- OpenGL Extensions
  - ARB_shader_objects
  - ARB_vertex_shader
  - ARB_fragment_shader
- OpenGL 2.0
  - Almost identical to using extensions
  - Avoids extension suffixes on function names

Program Object

- Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```
GLuint myProgObj;
myProgObj = glCreateProgram();
 /* define shader objects here */
glUseProgram(myProgObj);
glLinkProgram(myProgObj);
```

Reading a Shader

- Shader are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function **`glShaderSource`**
- If the shader is in a file, we can write a reader to convert the file to a string

Adding a Vertex Shader

```
GLint vShader;
GLunit myVertexObj;
GLchar vShaderfile[] = "my_vertex_shader";
GLchar* vSource =
        readShaderSource(vShaderFile);
glShaderSource(myVertexObj,
          1, &vertexShaderFile, NULL);
myVertexObj =
          glCreateShader(GL_VERTEX_SHADER);
glCompileShader(myVertexObj);
glAttachObject(myProgObj, myVertexObj);
```