CS 428: Fall 2009

Introduction to Computer Graphics

Programmable pipelines
GLSL: vertex and fragment shaders

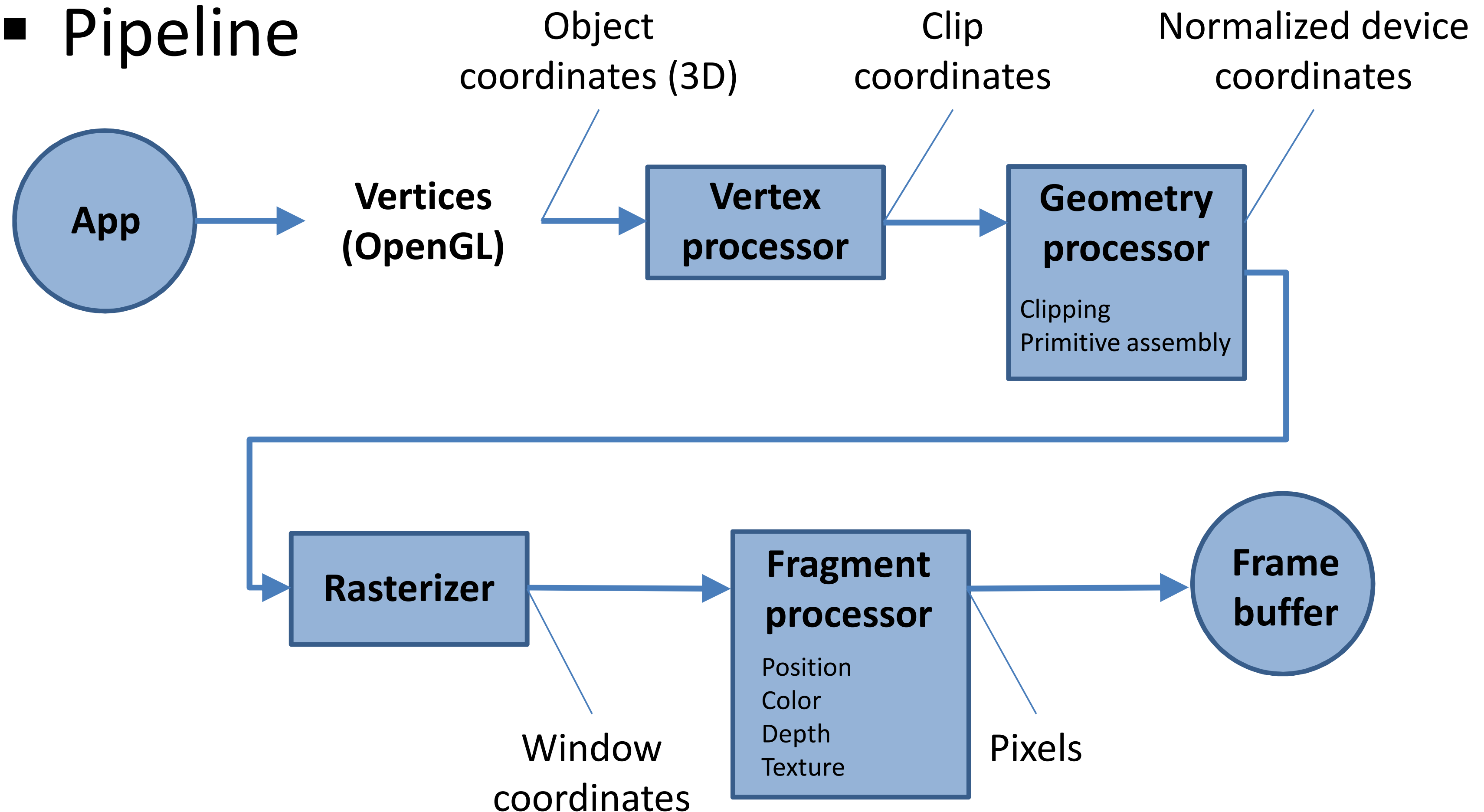
Graphics hardware

- Programmable {vertex, geometry, pixel} shaders
- Powerful

GeForce 8800 Ultra	GeForce 280 GTX	Intel Core2 Quad 3GHz
520 GFlops	933 GFlops	96 GFlops
86 GB/s	142 GB/s	21 GB/s
~250$ (board)	~400$ (board)	~400$ (chip)

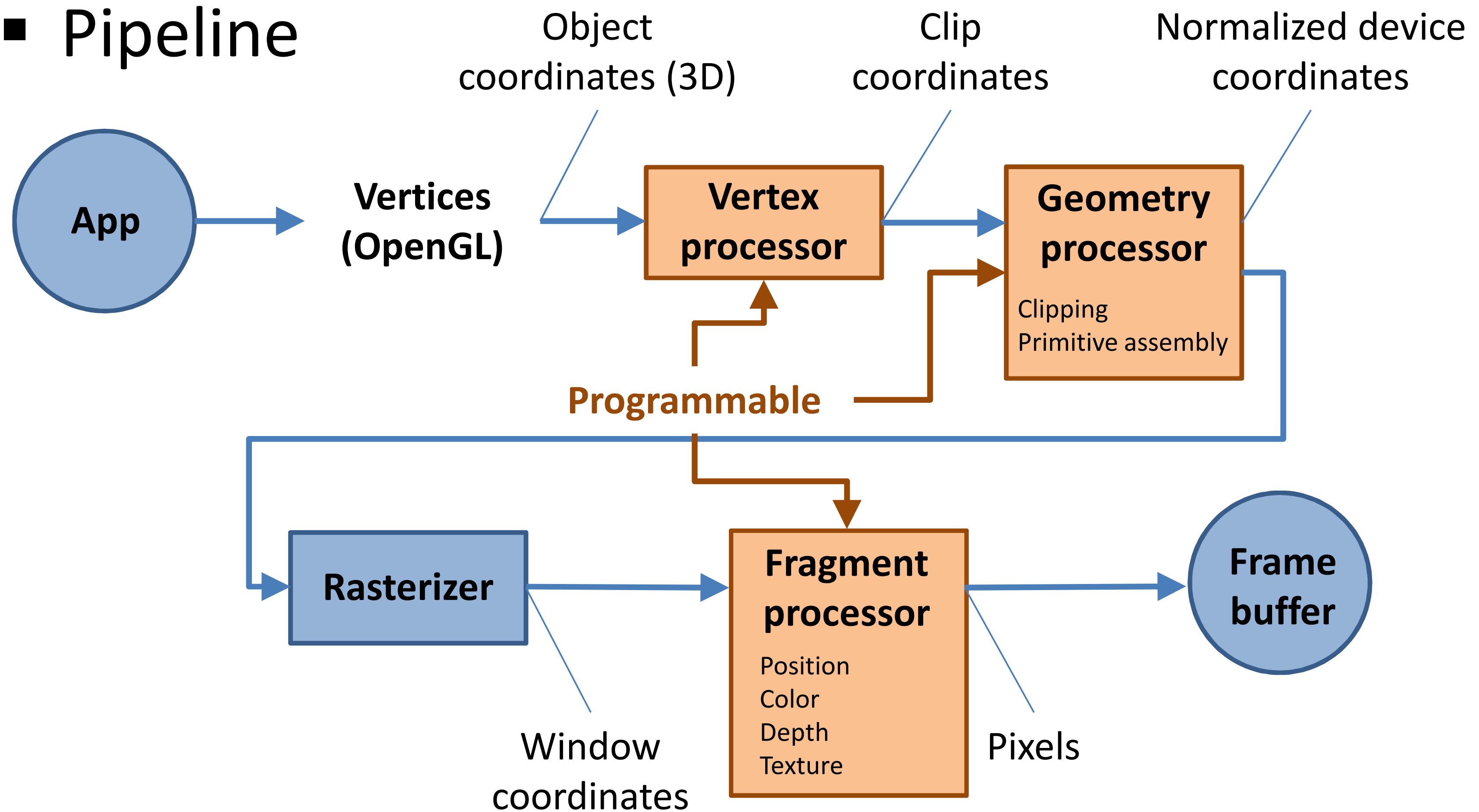
Hardware rendering

- Pipeline

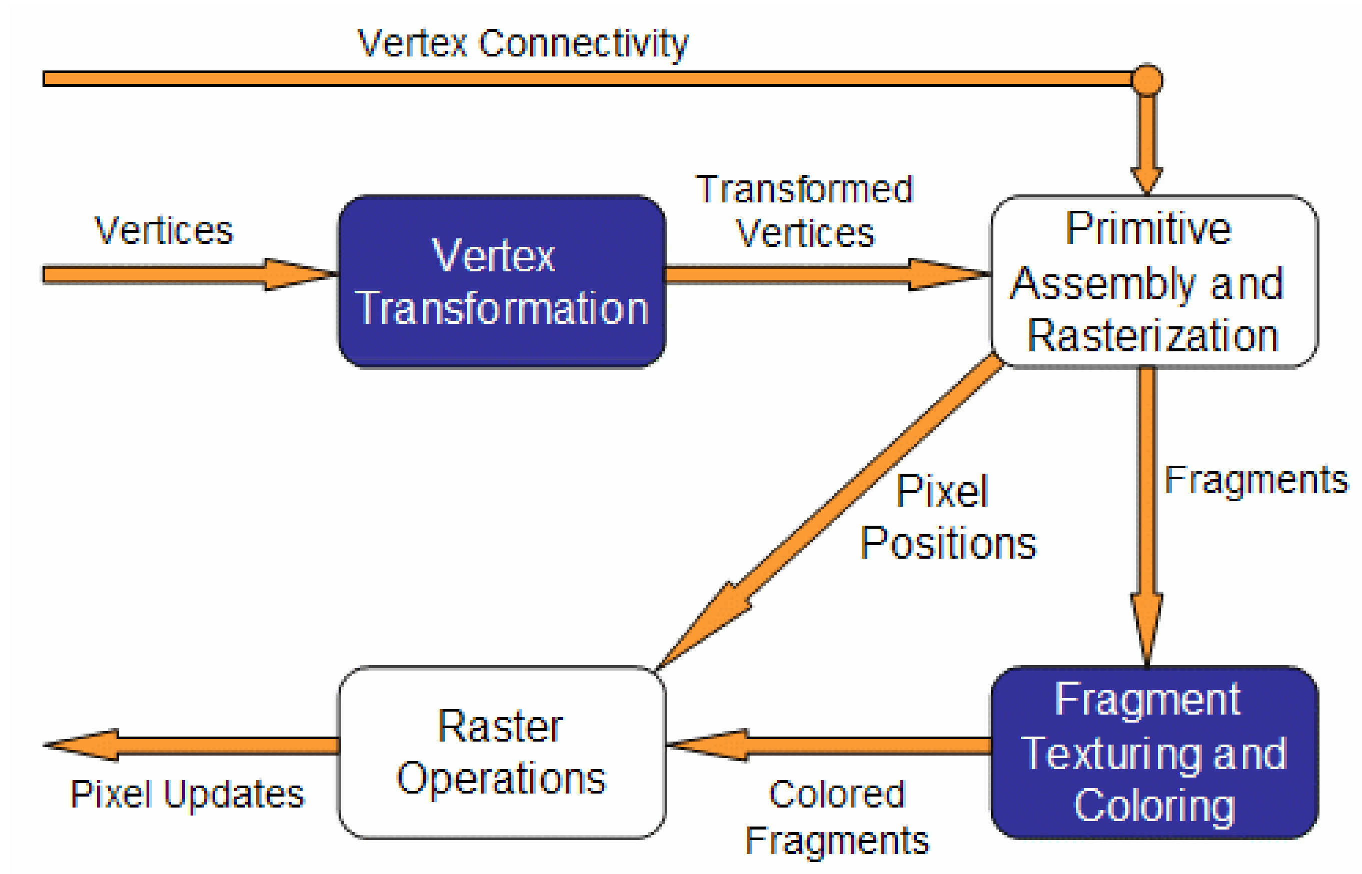


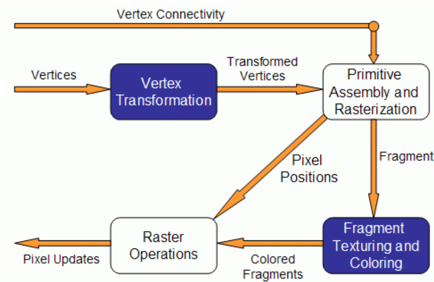
Hardware rendering

- Pipeline



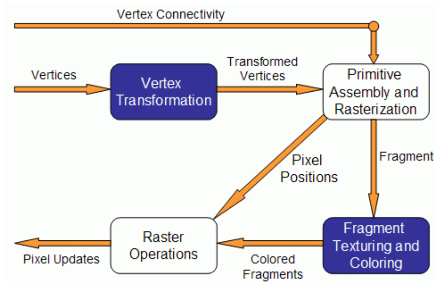
Pipeline overview





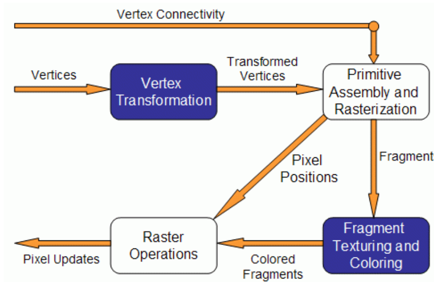
Pipeline

- **Vertex transformation**
  - Vertex = set of attributes such as location in space, as well as color, normal, texture coordinates, etc.
- Some of the operations performed by the fixed functionality at this stage are
  - Vertex position transformation
  - Lighting computations per vertex
  - Generation and transformation of texture coordinates



Pipeline

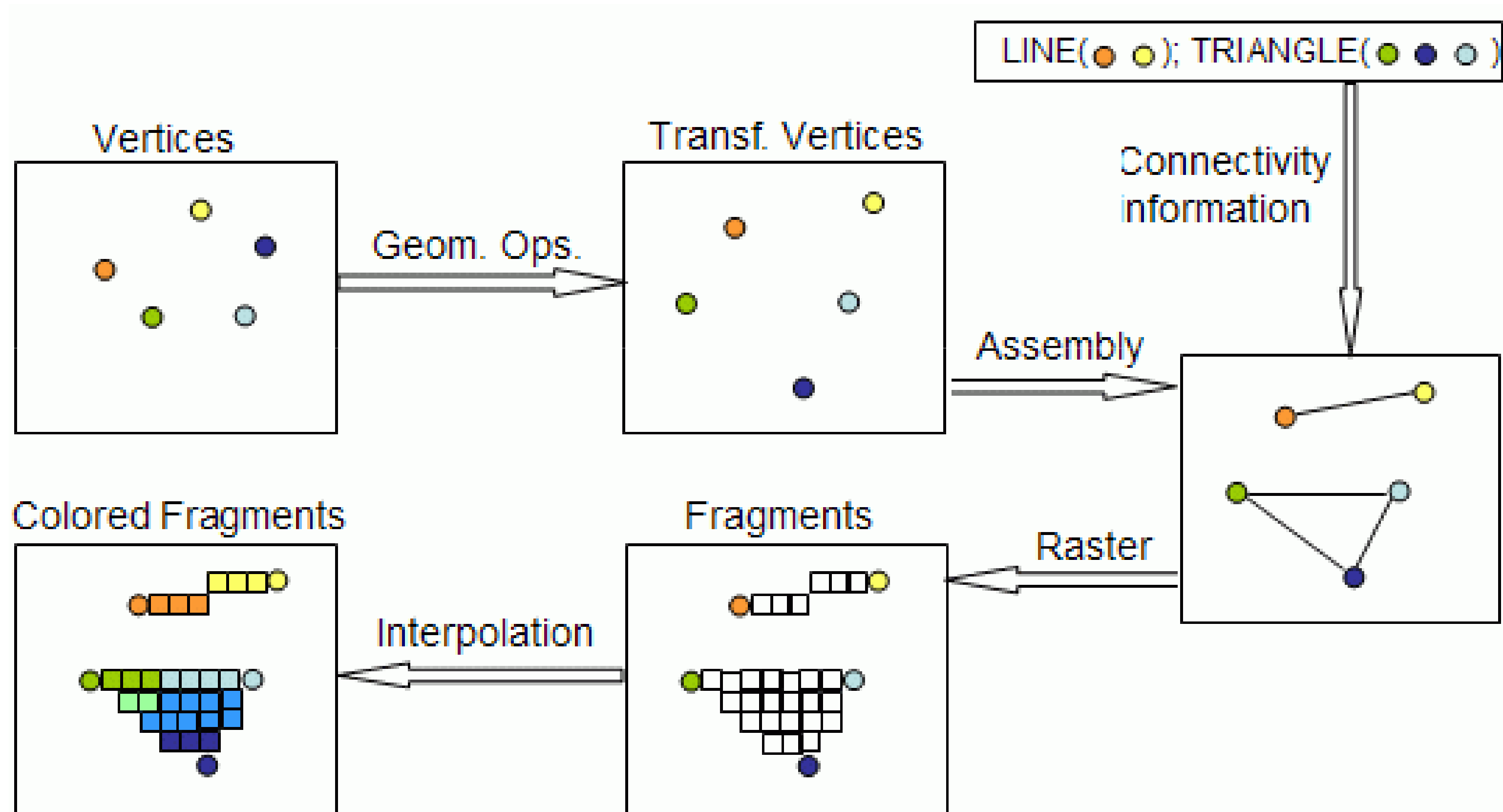
- **Primitive assembly and rasterization**
  - The inputs for this stage are the transformed vertices, as well as connectivity information
- Rasterization
  - Determines the fragments, and pixel positions of the primitive (Fragment = pos, color, normal, etc.)
- The output of this stage is twofold
  - The position of the fragments in the frame buffer
  - The interpolated values for each fragment of the attributes computed in the vertex shader



Pipeline

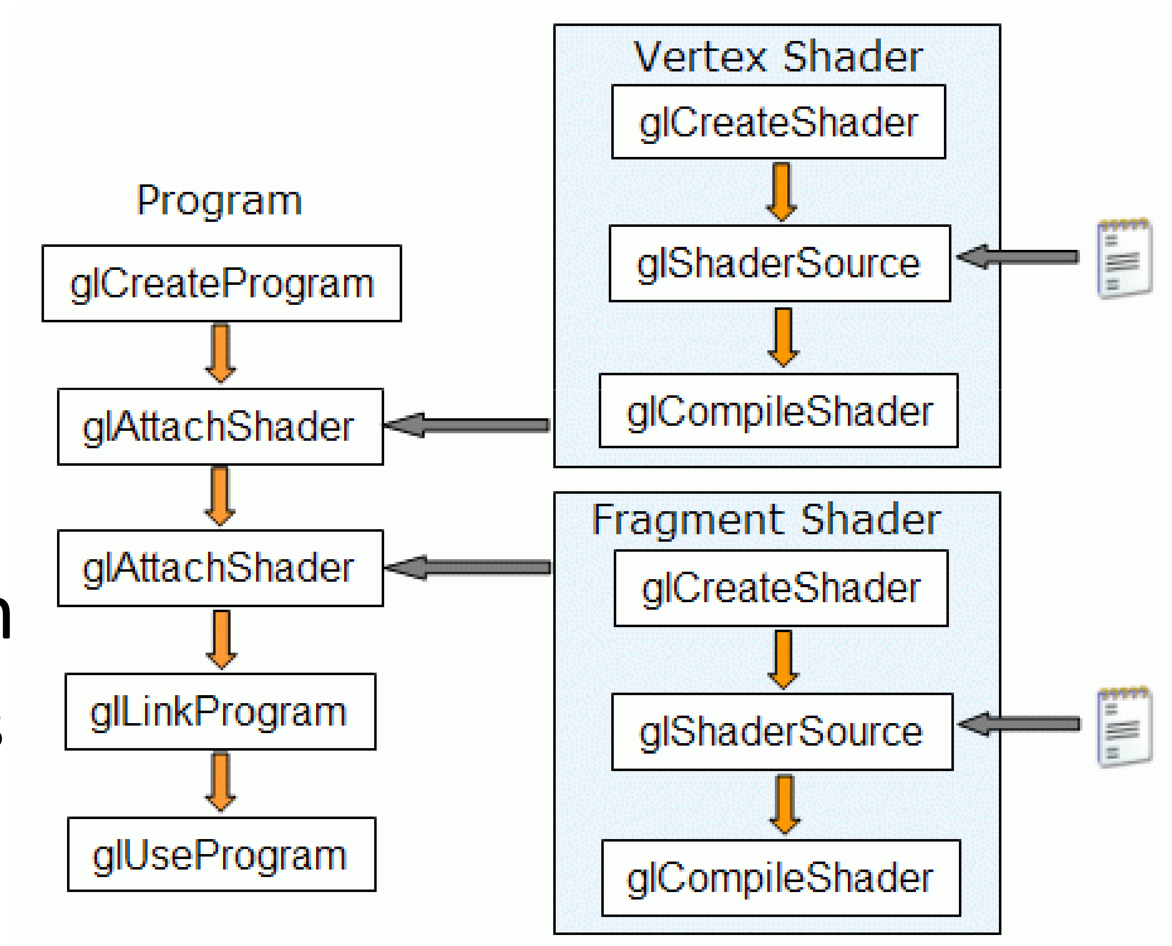
- **Fragment texturing and coloring**
  - Interpolated fragment information is the input of this stage
  - The common end result of this stage per fragment is a color value and a depth for the fragment
- **Raster operations**
  - Inputs: pixel location + depth and color values
  - Series of tests on the fragment: Scissor test, Alpha test, Stencil test, Depth test

Pipeline overview



Setting up GLSL shaders

- Create vertex and fragment shaders
  - Read source
  - Compile
- Create program
  - Attach shaders
  - Link
  - Use (enable)



Vertex shader

- A vertex shader operates on every vertex
  - Parallel processing (SIMD)
  - If you activate a vertex shader, **all** per-vertex operations of the fixed function OpenGL pipeline are replaced
- Can change the vertex color, position, etc.,
  - And also add extra info (***varying*** type) that is sent to the fragment shader (linearly Interpolated)
  - Can use vars from program (***uniform*** type)

Fragment shader

- A fragment shader operates on every fragment which is produced by rasterization
  - Parallel processing (SIMD)
  - Just like a vertex shader, a fragment shader replaces **all** per-fragment operations of the fixed function OpenGL pipeline.
- Attributes from vertices are interpolated across primitive per pixel

Types

- **`float`**, **`int`**, **`bool`** (also as arrays)
- **`vec2`**, **`vec3`**, **`vec4`** (float vectors)
- **`mat2`**, **`mat3`**, **`mat4`** (float matrices)

- Use construction for initialization

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

Uniforms, Attributes and Varyings

- **Uniforms**
  - Values which do not change during a rendering (light position or the light color). Available in vertex and fragment shaders (read-only).
- **Attributes**
  - Only available in vertex shader. Values which change for every vertex (vertex position or normals).
- **Varyings**
  - used for passing data from a vertex shader to a fragment shader. Read-only in fragment shader, read and writeable in vertex shader.

Built in attributes

- **`gl_Vertex`**
  - 4D vector representing the vertex position
- **`gl_Normal`**
  - 3D vector representing the vertex normal
- **`gl_Color`**
  - 4D vector representing the vertex color

Built in uniforms

- **`gl_ModelViewMatrix`**
  - 4x4 Matrix representing the model-view matrix
- **`gl_ModelViewProjectionMatrix`**
  - 4x4 Matrix representing the model-view-projection matrix
- **`gl_NormalMatrix`**
  - This matrix is used for normal transformation

Built in types for shader output

- **`gl_Position`**
  - 4D vector representing the final processed vertex position. Only available in vertex shader.
- **`gl_FragColor`**
  - 4D vector representing the final color which is written in the frame buffer. Only available in fragment shader.
- **`gl_FragDepth`**
  - Float representing the depth which is written in the depth buffer. Only in fragment shader.

Built in functions

- **`dot(a,b)`**
  - a simple dot product
- **`cross(a,b)`**
  - a simple cross product
- **`normalize(v)`**
  - normalize a vector
- **`clamp(v)`**
  - clamping a vector to a minimum and a maximum
- **`sin() cos() min() max() reflect()`**

Vertex shader

- Basic vertex shader (pass thru)

```
void main() {
    gl_Position = gl_ProjectionMatrix *
    gl_ModelViewMatrix * gl_Vertex;

    gl_FrontColor = gl_Color;
}
```

Vertex shader

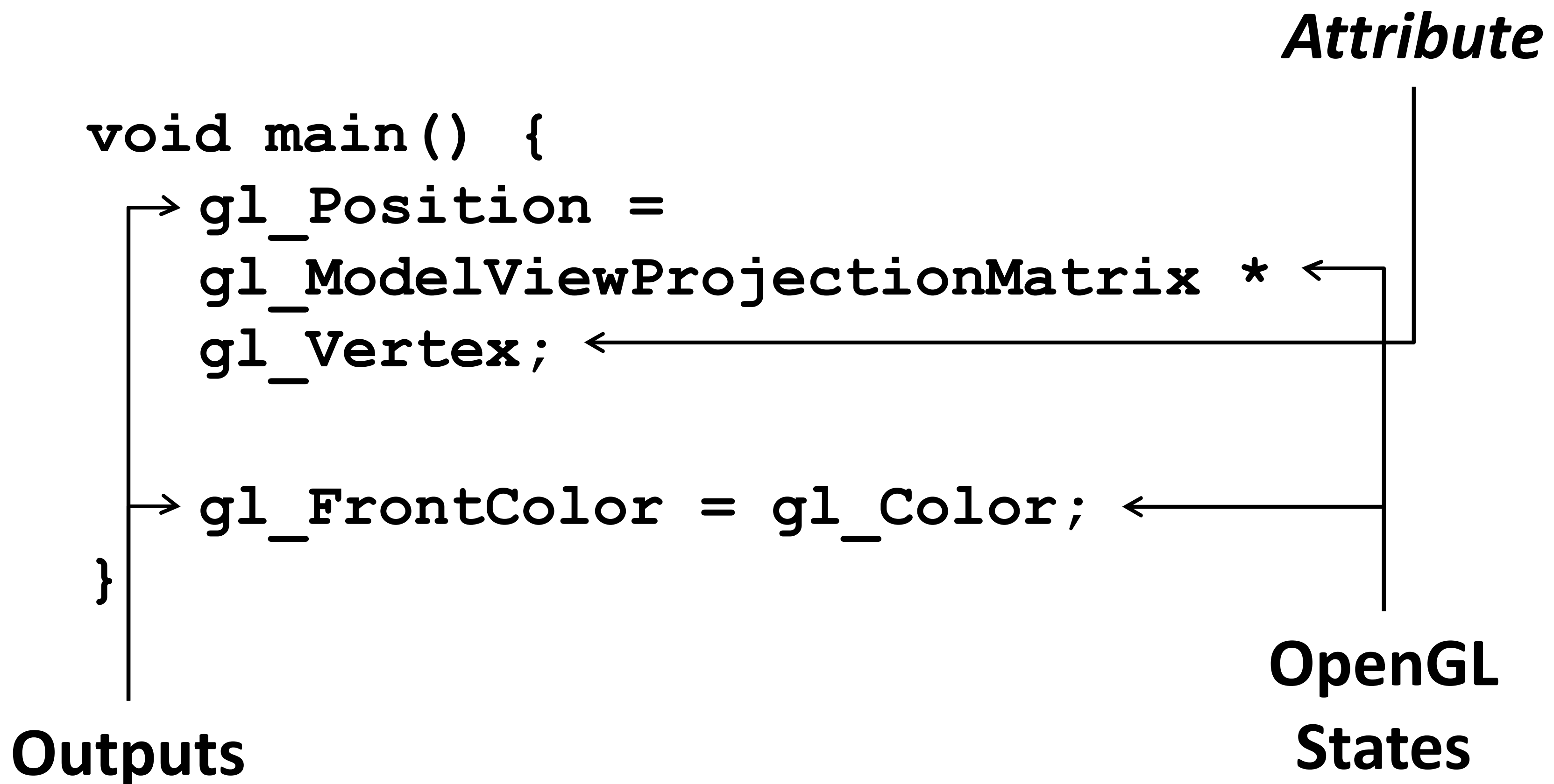
- Basic vertex shader (pass thru)

```
void main() {
    gl_Position = gl_ProjectionMatrix *
    gl_ModelViewMatrix * gl_Vertex;

    gl_FrontColor = vec4(1.0,0.0,0.0,1.0);
}
```

Vertex shader

- Basic vertex shader (pass thru)



Vertex shader

- ***Varying*** types

```
varying pos; // both in eye coordinates
varying norm;
void main() {
    gl_Position = ftransform();
    pos = gl_ModelViewMatrix * gl_Vertex
    norm = gl_NormalMatrix * gl_Normal
    gl_FrontColor = gl_Color;
}
```

Fragment shader

- Basic fragment shader (pass thru)

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

Fragment shader

- ***Varying*** types

```
varying pos; // both in eye coordinates
varying norm;
void main() {
    // do stuff with pos and norm
    vec3 eye = normalize(-vec3(pos));
    vec3 n = normalize(norm);
    float a = dot(eye,n);
    gl_FragColor = vec4(a,a,a,1.0);
}
```

JOGL/GLSL demo