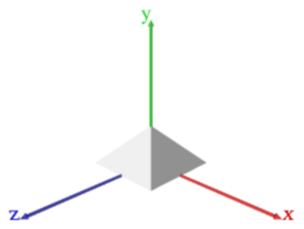
GRK 3

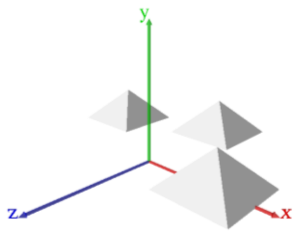
Dr W Palubicki

Simplified Rendering Pipeline

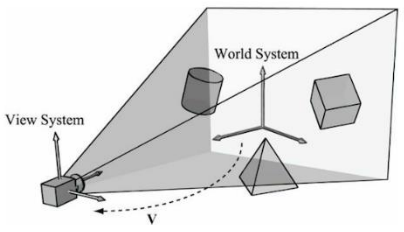
Model Matrix → View/Camera Matrix → Projection Matrix → Viewport Transform →



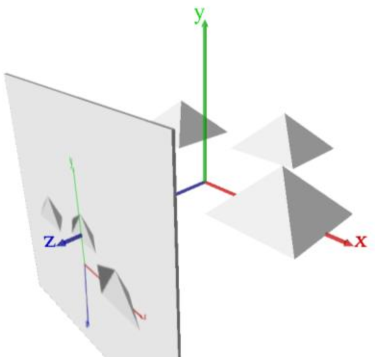
Object Space



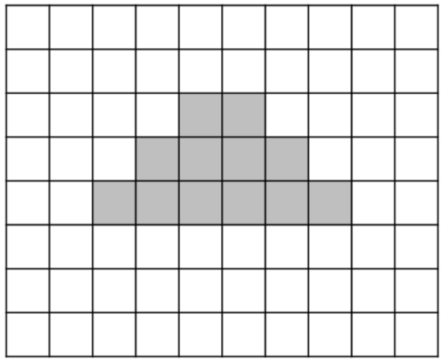
World Space



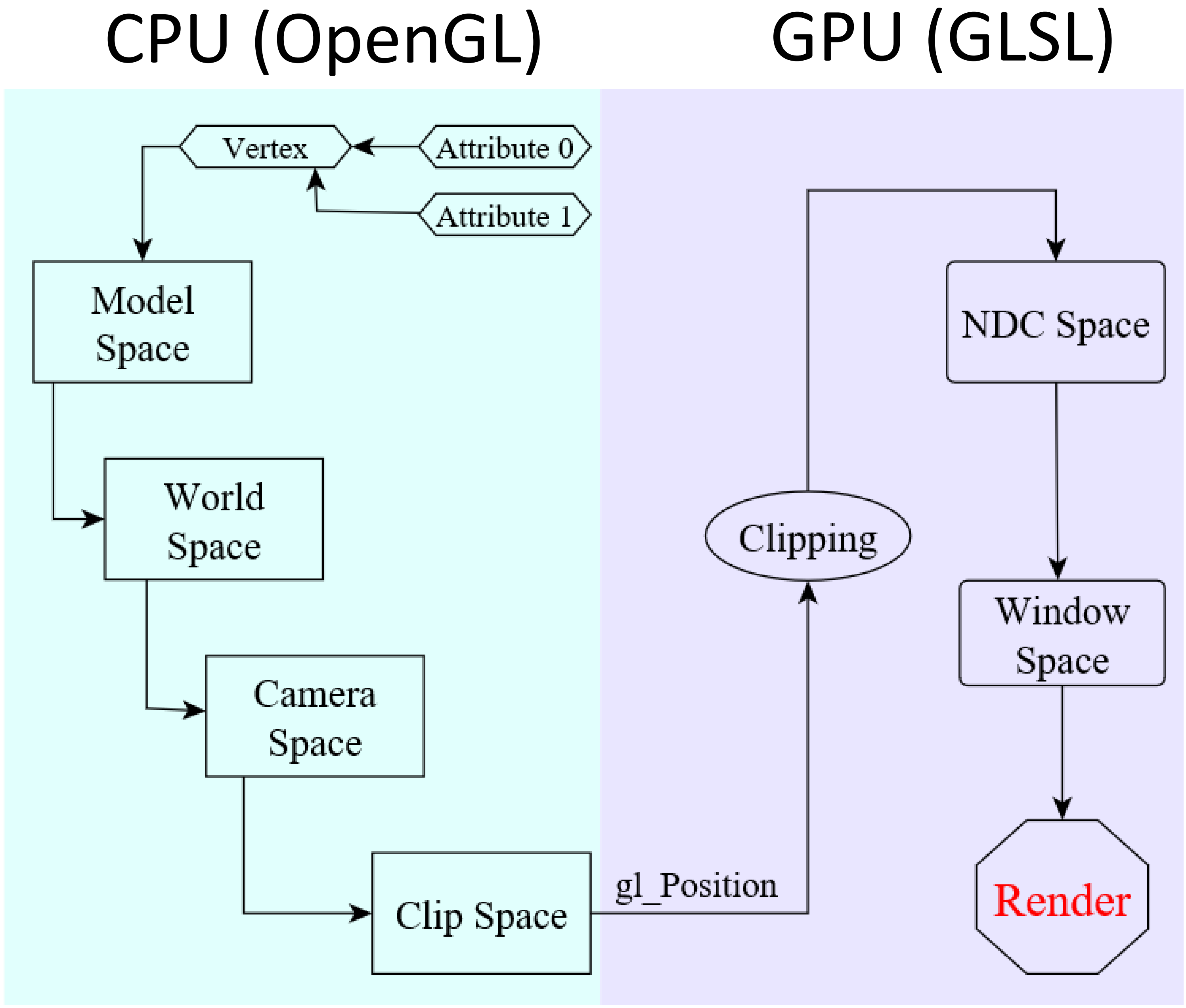
View/Camera Space



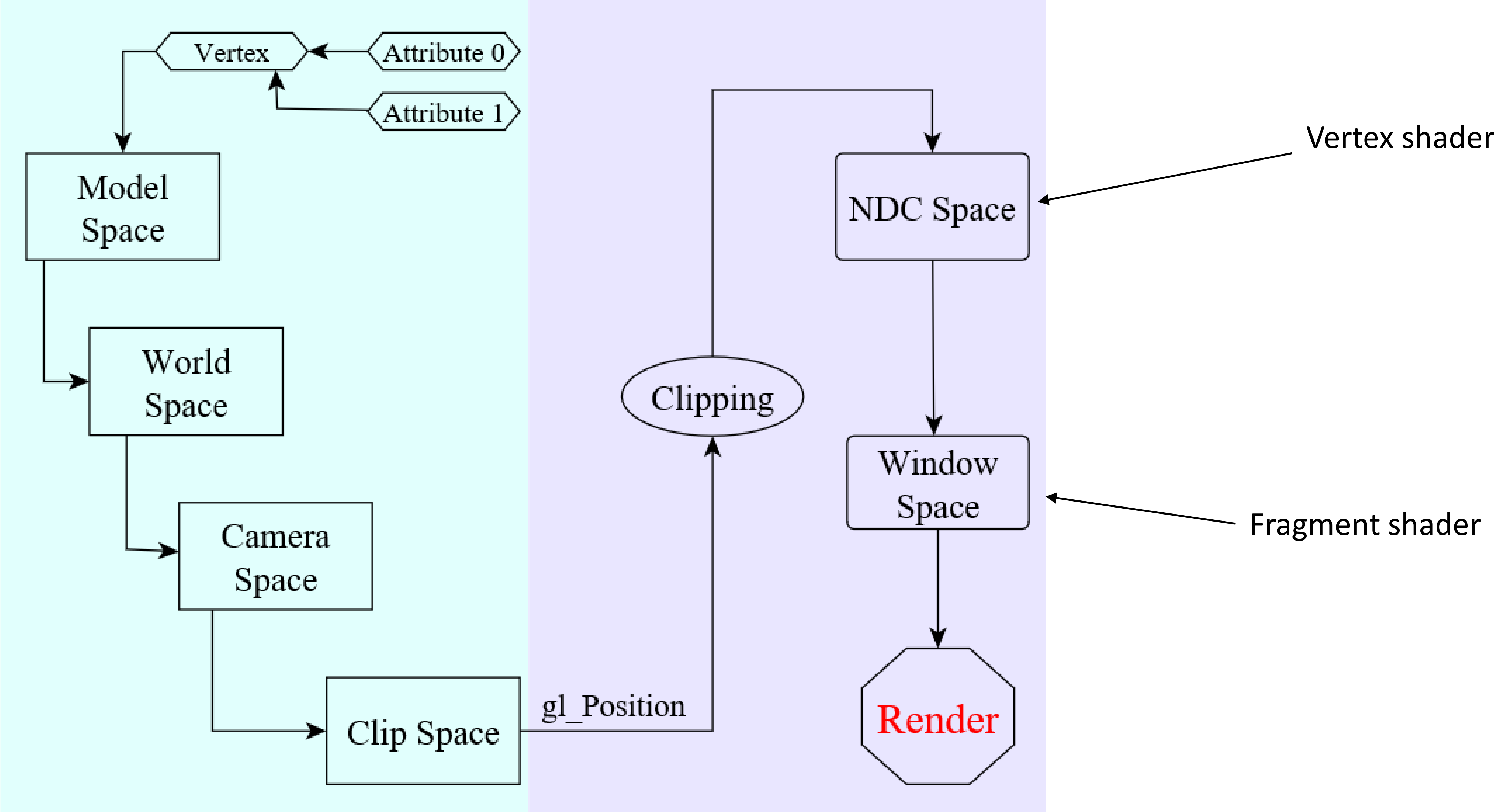
Clip Space



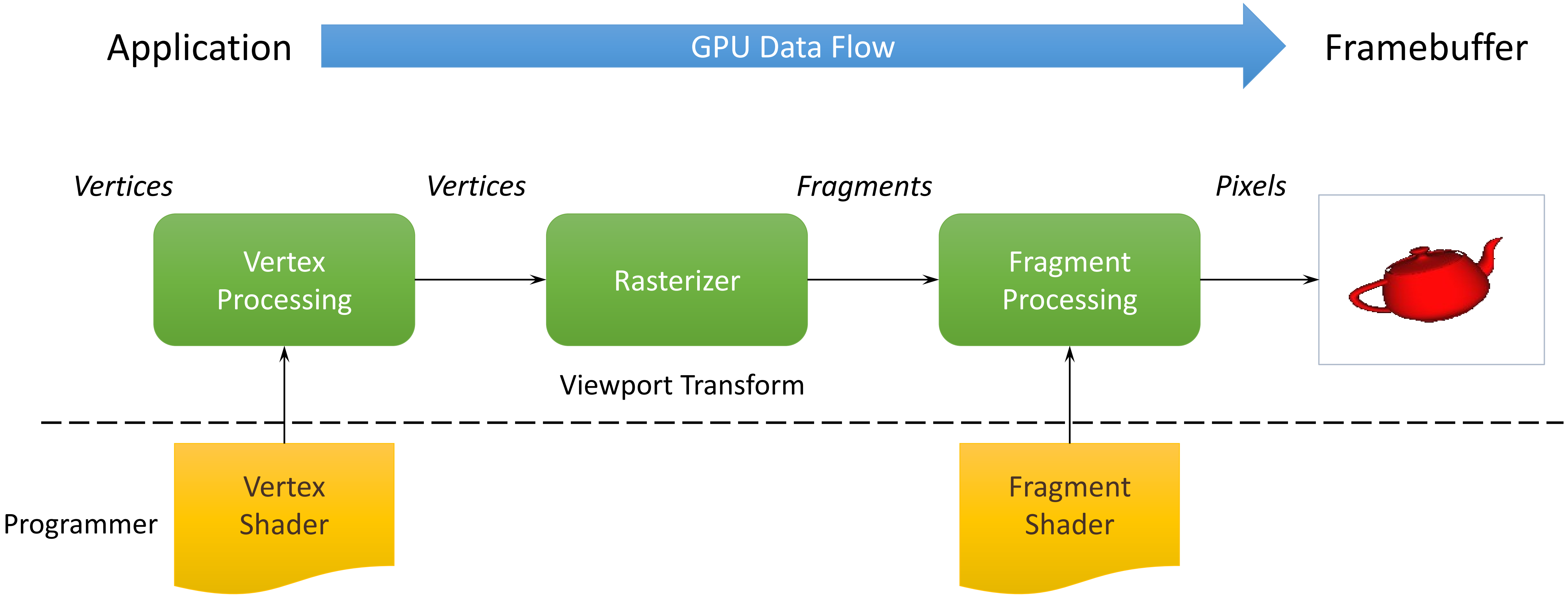
Screen/Window Space


CPU (OpenGL)
GPU (GLSL)
Vertex
Attribute 0
Attribute 1
Model Space
World Space
Camera Space
Clip Space
gl_Position
Clipping
NDC Space
Window Space
Render

CPU (OpenGL) GPU (GLSL)



A Simplified GPU Data Flow

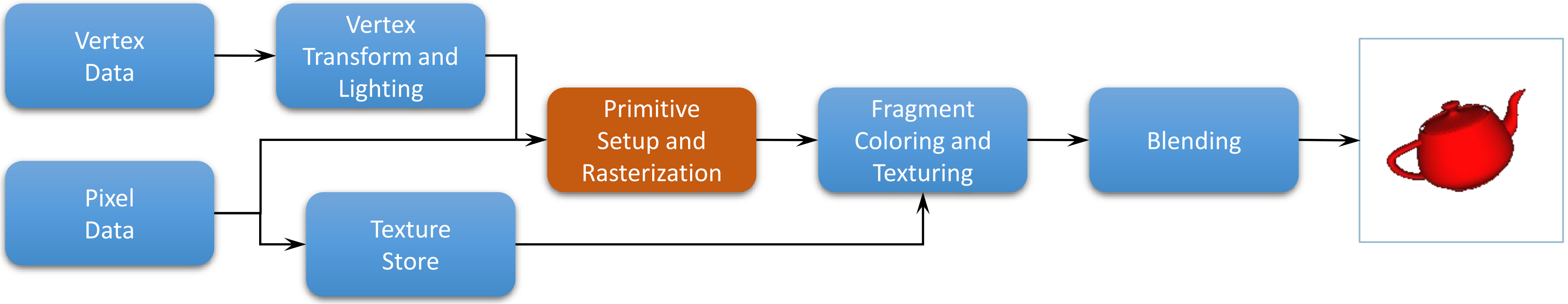


Overview

- History of the OpenGL Pipeline
- OpenGL Shading Language (GLSL)
- Vertex Shaders
- Fragment Shaders

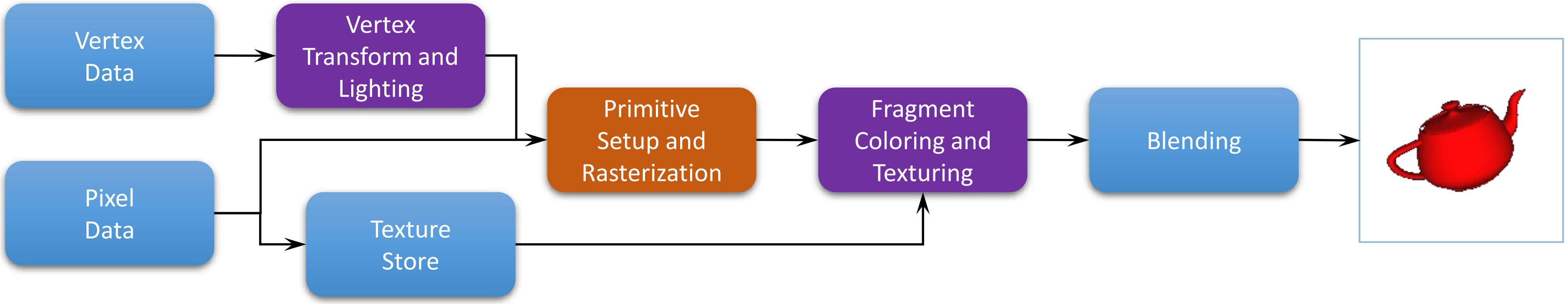
Beginnings of the Pipeline (1994)

- OpenGL 1.0 was a *fixed-function* pipeline



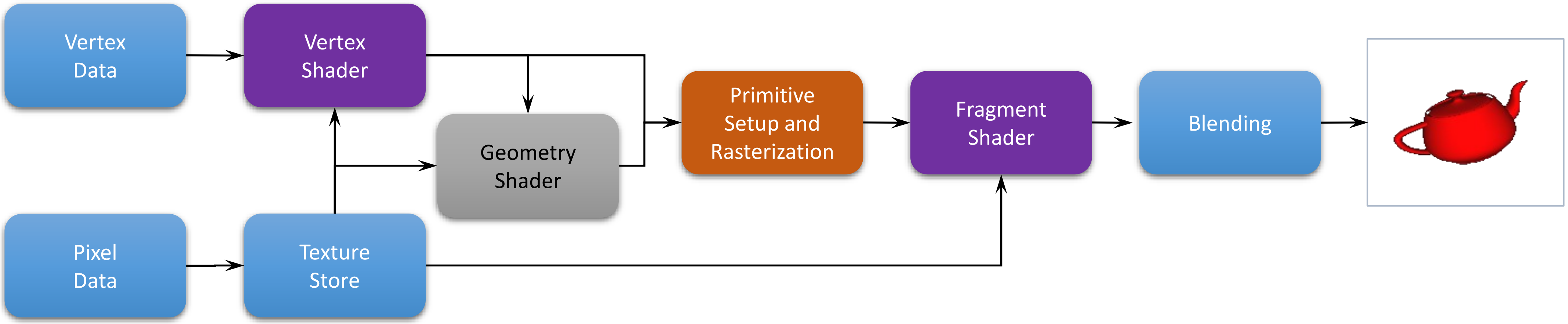
Programmable Pipeline (2004)

- OpenGL 2.0 programmable shaders (written in GLSL)
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage



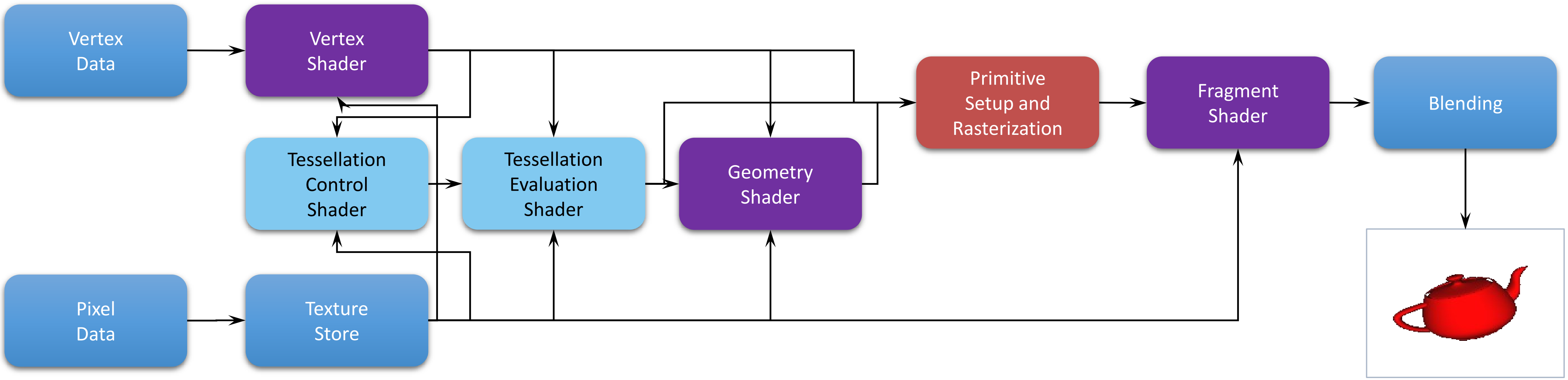
More Programmability

- OpenGL 3.2 (released 2009) added an additional shading stage – geometry shaders
  - modify geometric primitives within the graphics pipeline



The Latest Pipelines

- OpenGL 4.1 (released 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders
- Latest version is 4.6

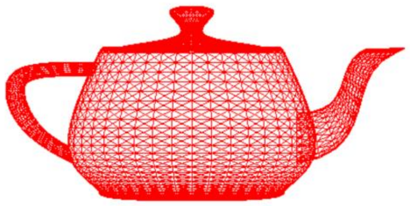


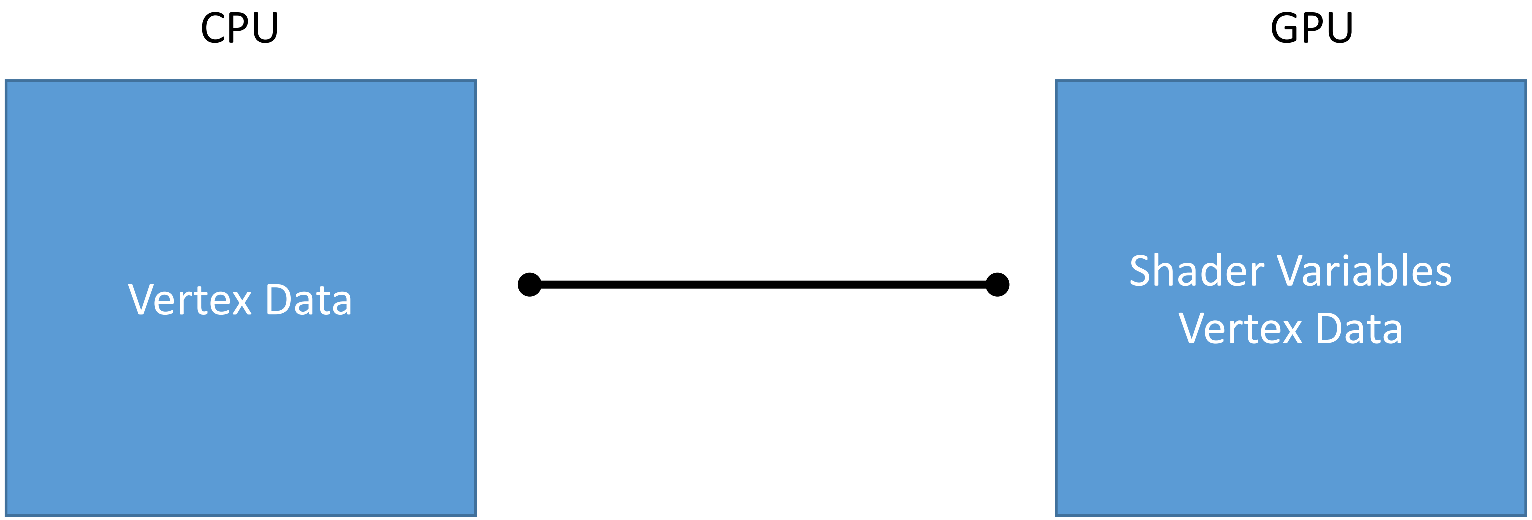
OpenGL ES and WebGL

- OpenGL ES 3.2
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- WebGL
  - JavaScript implementation of ES
  - Runs on most recent browsers

OpenGL Programming

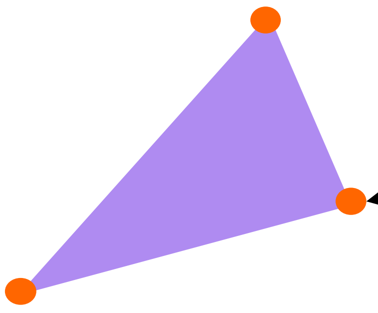
- Modern OpenGL programs essentially do the following steps:
  - Create shader programs (executed on GPU)
  - Create buffer objects and load data into them (executed on CPU/GPU)
  - “Connect” data locations (CPU) with shader variables (GPU)
  - Render





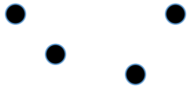
Representing Geometric Objects

- Geometric objects are represented using *vertices*
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space
- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in vertex buffer objects (VBOs)
- VBOs must be stored in vertex array objects (VAOs)

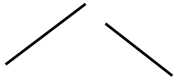


$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

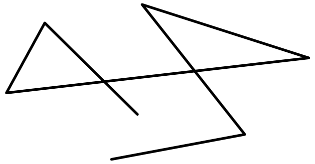
OpenGL Geometric Primitives



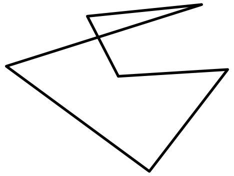
GL_POINTS



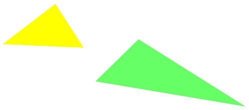
GL_LINES



GL_LINE_STRIP

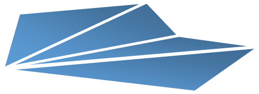


GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

Example Program

- Rendering a cube with colors at each vertex
  - initialize vertex data
  - organize data for rendering

Initializing the Cube's Data

- Build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

Initializing the Cube’s Data

- Before we can initialize our VBO, we need to create the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
vec4  vPositions[NumVertices];
vec4  vColors[NumVertices];
```

Cube Data

- Vertices of a unit cube centered at origin
  - sides aligned with axes

```
vec4 positions[8] = {
    vec4( -0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5,  0.5,  0.5, 1.0 ),
    vec4(  0.5,  0.5,  0.5, 1.0 ),
    vec4(  0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5, -0.5, -0.5, 1.0 ),
    vec4( -0.5,  0.5, -0.5, 1.0 ),
    vec4(  0.5,  0.5, -0.5, 1.0 ),
    vec4(  0.5, -0.5, -0.5, 1.0 )
};
```

Cube Data

- We'll also set up an array of RGBA colors

```
vec4 colors[8] = {
    vec4( 0.0, 0.0, 0.0, 1.0 ),  // black
    vec4( 1.0, 0.0, 0.0, 1.0 ),  // red
    vec4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    vec4( 0.0, 1.0, 0.0, 1.0 ),  // green
    vec4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    vec4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    vec4( 1.0, 1.0, 1.0, 1.0 ),  // white
    vec4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating a Cube Face from Vertices

- To simplify generating the geometry, define a function
- Create two triangles for each face and assigns colors to the vertices

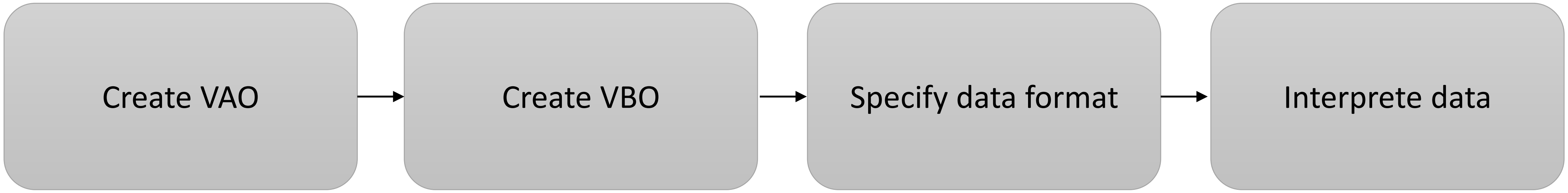
```
int Index = 0;  // global variable indexing into VBO arrays

void quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[b]; vPositions[Index] = positions[b]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[d]; vPositions[Index] = positions[d]; Index++;
}
```

Generating the Cube from Faces

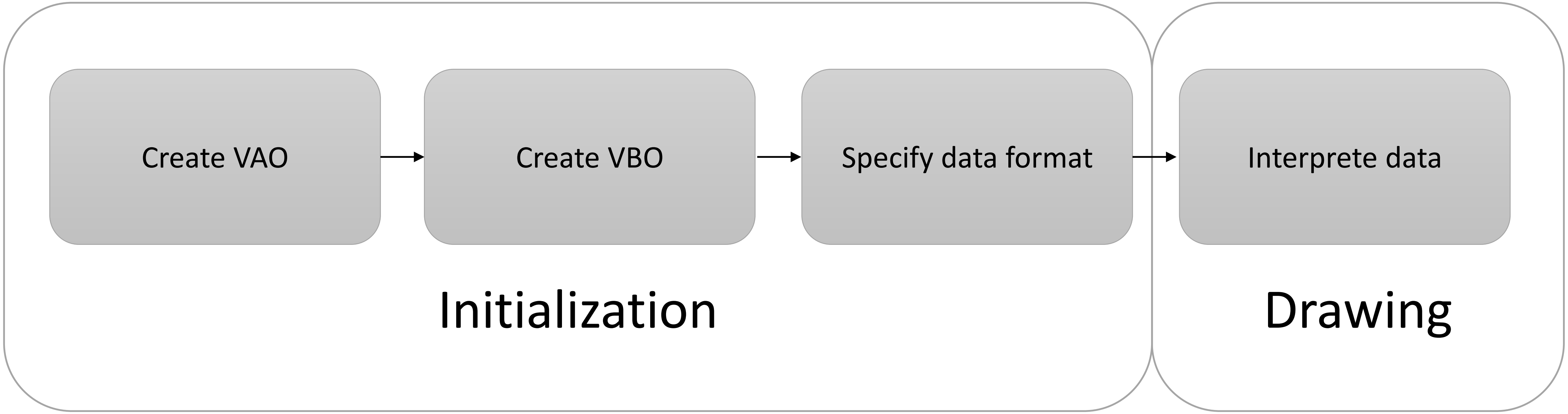
- Generate 12 triangles for the cube
  - 36 vertices with 36 colors

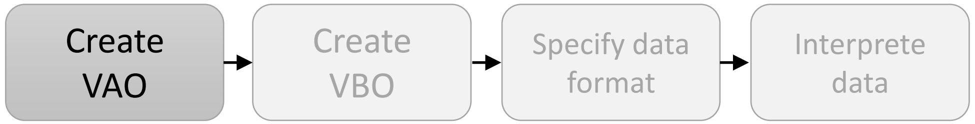
```
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Transfer vertex data to GPU



Transfer vertex data to GPU





VAOs in code

- Create a vertex array object

```
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );
```

Vertex Array Objects

- VAOs store the data of a geometric object
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering

Create VAO → Create VBO → Specify data format → Interprete data

Storing Vertex Attributes

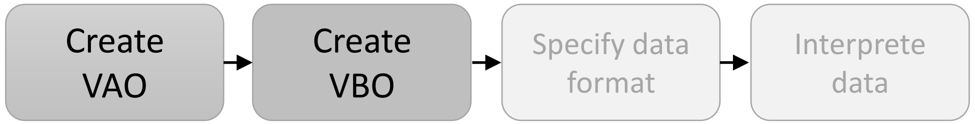
- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling

    ```
    glBindBuffer( GL_ARRAY_BUFFER, … )
    ```

  - load data into VBO using

    ```
    glBufferData( GL_ARRAY_BUFFER, … )
    ```

  - bind VAO for use in rendering `glBindVertexArray()`



VBOs in Code

- Create and initialize a buffer object

```
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER,
              sizeof(vPositions) + sizeof(vColors),
              NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
                 sizeof(vPositions), vPositions );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(vPositions),
                 sizeof(vColors), vColors );
```

Connecting CPU with GPU

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location

Vertex Array Code

- Associate shader variables with vertex arrays

```
GLuint vPosition =
    glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT,
    GL_FALSE, 0,BUFFER_OFFSET(0) );

GLuint vColor =
    glGetAttribLocation( program,"vColor" );
glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(sizeof(vPositions)) );
```

Vertex Array Code

- Associate shader variables with vertex arrays

Shader pointer

```
GLuint vPosition =
    glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT,
    GL_FALSE, 0,BUFFER_OFFSET(0) );

GLuint vColor =
    glGetAttribLocation( program,"vColor" );
glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(sizeof(vPositions)) );
```

Variable name in shader

Create VAO → Create VBO → Specify data format → Interprete data

In C++/OpenGL code

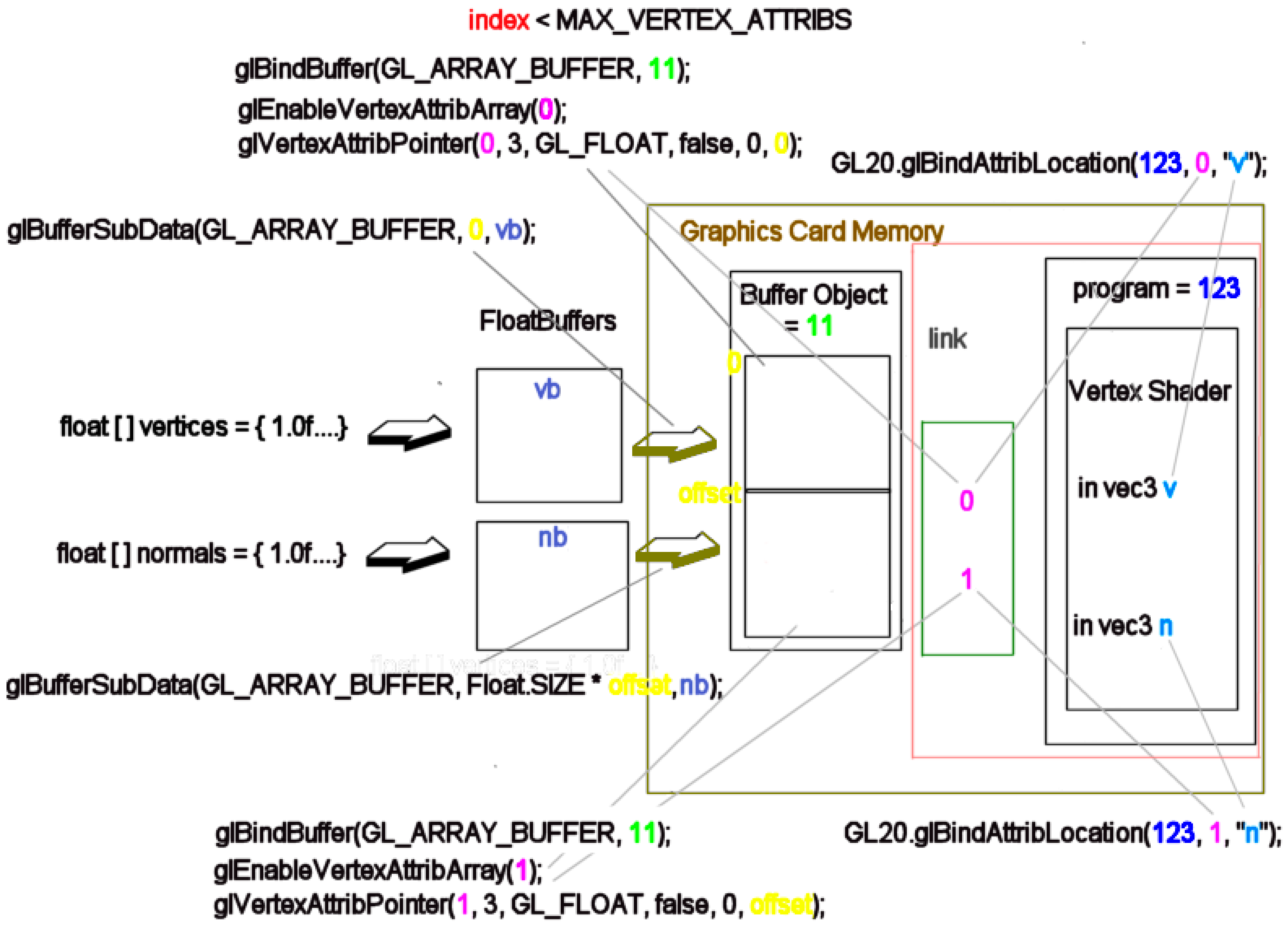
```
void Core::DrawVertexArray(const float * vertexArray, int numVertices, int elementSize )
{
    glVertexAttribPointer(0, elementSize, GL_FLOAT, false, 0, vertexArray);
    glEnableVertexAttribArray(0);

    glDrawArrays(GL_TRIANGLES, 0, numVertices);
}
```

Predefined index

In Shader code

```
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec2 vertexTexCoord;
layout(location = 2) in vec3 vertexNormal;
layout(location = 3) in vec3 vertexTangent;
```

Predefined index


index < MAX_VERTEX_ATTRIBS
glBindBuffer(GL_ARRAY_BUFFER, 11);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
GL20.glBindAttribLocation(123, 0, "v");
glBufferSubData(GL_ARRAY_BUFFER, 0, vb);
Graphics Card Memory
Buffer Object = 11
program = 123
FloatBuffers
link
0
Vertex Shader
vb
float [ ] vertices = { 1.0f....}
offset
in vec3 v
0
nb
float [ ] normals = { 1.0f....}
1
in vec3 n
glBufferSubData(GL_ARRAY_BUFFER, Float.SIZE * offset, nb);
glBindBuffer(GL_ARRAY_BUFFER, 11);
GL20.glBindAttribLocation(123, 1, "n");
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, offset);

Create VAO → Create VBO → Specify data format → Interprete data

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- Usually invoked in display callback
- Initiates vertex shader

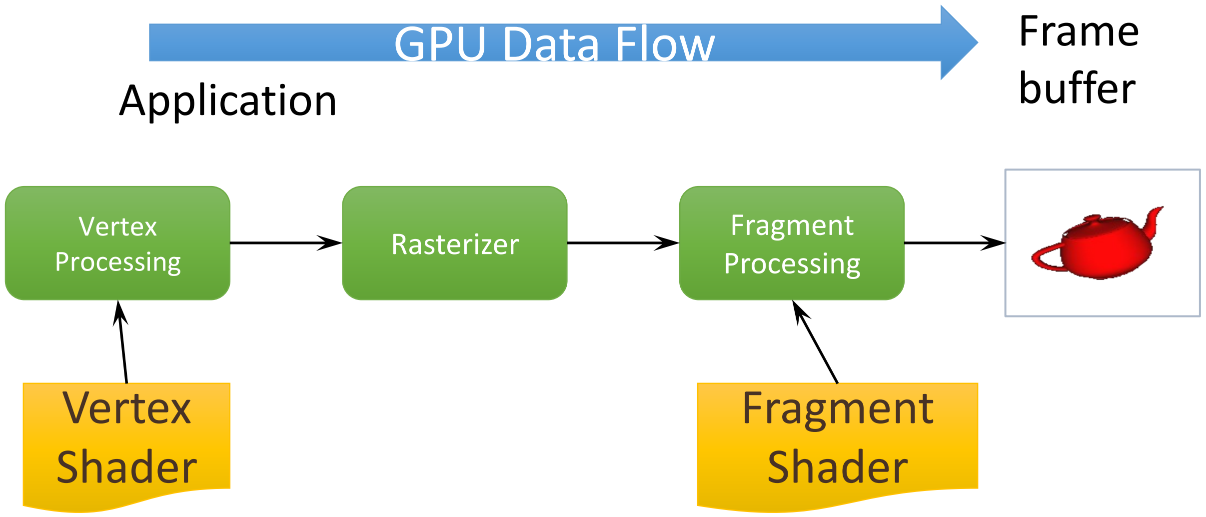
Create VAO → Create VBO → Specify data format → Interprete data

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- Usually invoked in display callback
- Initiates vertex shader



Example: Initialization

```
glGenVertexArrays(1, &asteroid_vao);
glBindVertexArray(asteroid_vao);
glBindBuffer(GL_ARRAY_BUFFER, asteroid_buffer_object);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);

glGenVertexArrays(1, &ship_vao);
glBindVertexArray(ship_vao);
glBindBuffer(GL_ARRAY_BUFFER, ship_buffer_object);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);

glBindVertexArray(0);
```

Example: Drawing

```
glBindVertexArray(asteroid_vao);
glDrawArrays(GL_LINE_LOOP, 0, num_asteroid_vertices);

glBindVertexArray(ship_vao);
glDrawArrays(GL_LINE_LOOP, 0, num_ship_vertices);

glBindVertexArray(0);
```

Shaders and GLSL

GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

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

Operators

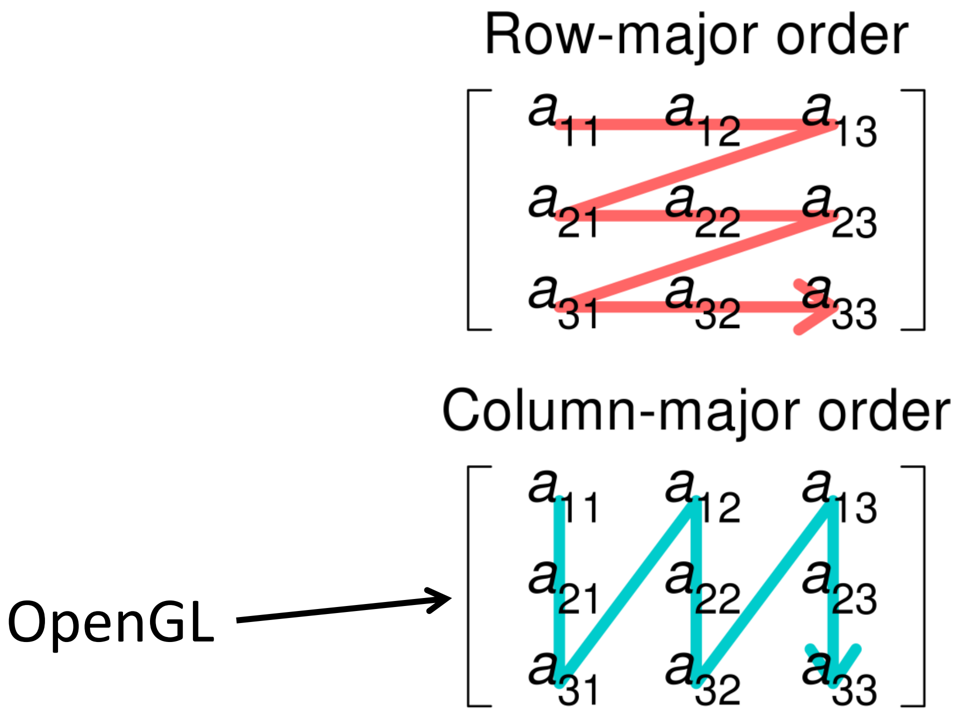
- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Operators

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```



Components and Swizzling

- Access vector components using either:
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

- For example:

```
vec3 v;
v[1], v.y, v.g, v.t
```
- all refer to the same element

- Component swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

GLSL: Vectors

- Constructors:

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

vec3 xyz = vec3(1.0); // [1.0, 1.0, 1.0]

vec3 xyz = vec3(vec2(1.0, 2.0), 3.0);
```

GLSL: Vectors

- Swizzle:

```
vec4 c = vec4(0.5, 1.0, 0.8, 1.0);

vec3 rgb = c.rgb;  // [0.5, 1.0, 0.8]

vec3 bgr = c.bgr;  // [0.8, 1.0, 0.5]

vec3 rrr = c.rrr;  // [0.5, 0.5, 0.5]

c.a = 0.5;         // [0.5, 1.0, 0.8, 0.5]
c.rb = 0.0;        // [0.0, 1.0, 0.0, 0.5]

float g = rgb[1];  // 1.0, indexing
```

GLSL: Matrices

- Constructors

```
mat3 i = mat3(1.0); // 3x3 identity matrix

mat2 m = mat2(1.0, 2.0,  // [1.0 3.0]
              3.0, 4.0); // [2.0 4.0]
```

- Matrix elements

```
float f = m[column][row];

float x = m[0].x; // element x of 1st column

vec2 yz = m[1].yz; // elements yz of 2nd column
```

Matrix as an array of vectors

Swizzle

GLSL: Vectors and Matrices

- Matrix-Vector operations:

```
vec3 xyz = // ...

vec3 v0 = 2.0 * xyz; // scaling
vec3 v1 = v0 + xyz;  // addition
vec3 v2 = v0 * xyz;  // scalar multiplication

mat3 m = // ...
mat3 v = // ...

mat3 mv = v * m;        // matrix * matrix
mat3 xyz2 = mv * xyz; // matrix * vector
mat3 xyz3 = xyz * mv; // vector * matrix
```

Qualifiers

- `in`, `out`
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out vec4 color;
```

- `uniform`
  - shader-constant, global variable from application

```
uniform float time;
uniform vec4 rotation;
```

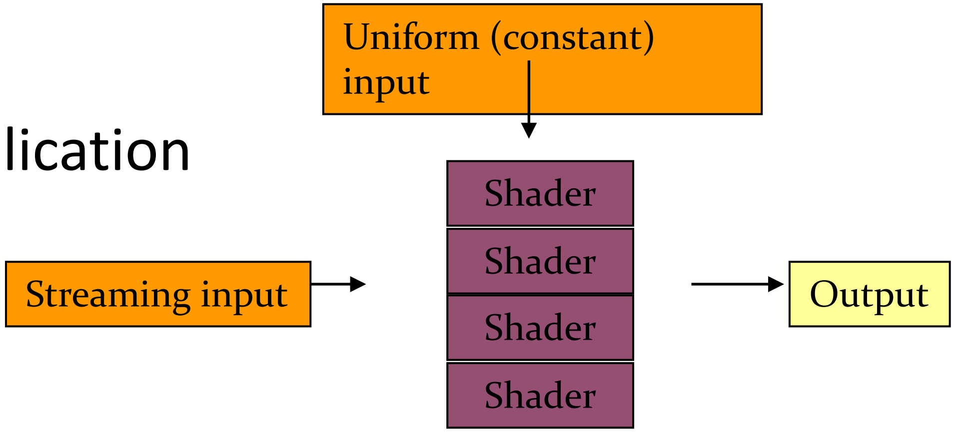
Qualifiers

- in, out
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out vec4 color;
```

- uniform
  - shader-constant, global variable from application

```
uniform float time;
uniform vec4 rotation;
```



Functions

- Built-in
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`
- User defined

Built-in Variables

- `gl_Position`
  - (required) output position from vertex shader
- `gl_FragCoord`
  - input fragment position
- `gl_FragDepth`
  - input depth value in fragment shader

Simple Vertex Shader for Cube Example

```
#version 430

in vec4 vPosition;
in vec4 vColor;

out vec4 color;

uniform mat4 ModelViewMatrix;

void main()
{
    color = vColor;
    gl_Position = ModelViewMatrix * vPosition;
}
```

Simple Vertex Shader for Cube Example

```
#version 430

in vec4 vPosition;
in vec4 vColor;

out vec4 color;

uniform mat4 ModelViewMatrix;

void main()
{
    color = vColor;
    gl_Position = ModelViewMatrix * vPosition;
}
```

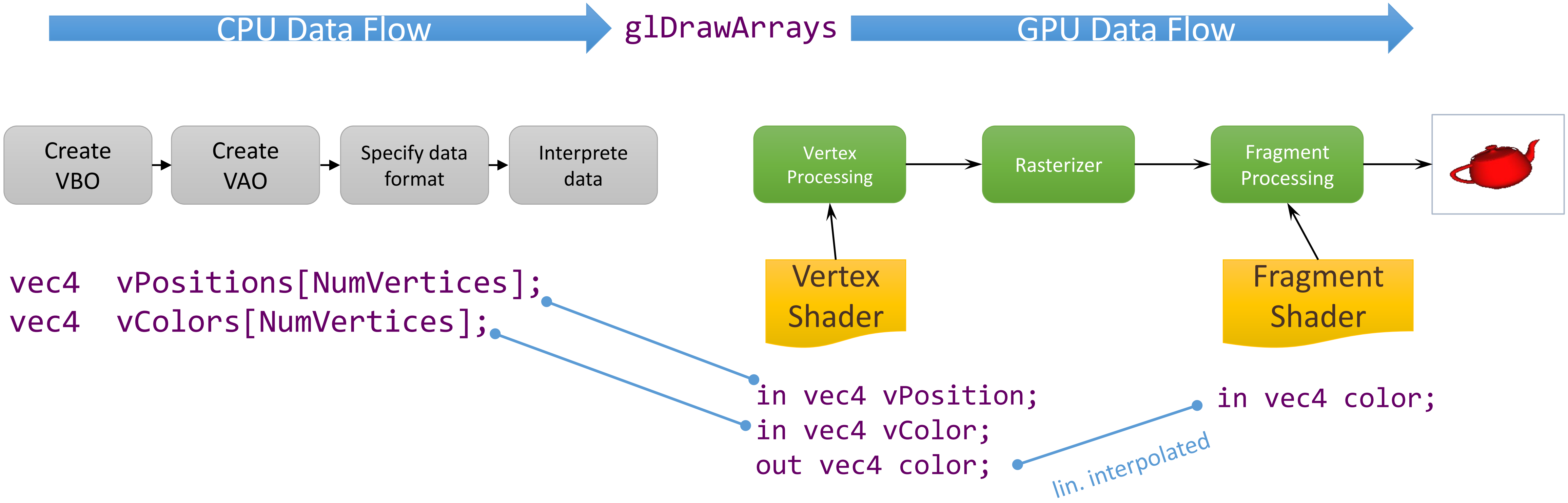
Variable declaration

Variable definition

The Simplest Fragment Shader

```
#version 430

in vec4 color;

out vec4 fColor; // fragment’s final color

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

Overview


CPU Data Flow
glDrawArrays
GPU Data Flow
Create VBO
Create VAO
Specify data format
Interprete data
Vertex Processing
Rasterizer
Fragment Processing
Vertex Shader
Fragment Shader
vec4 vPositions[NumVertices];
vec4 vColors[NumVertices];
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
lin. interpolated
in vec4 color;

Associating Shader Variables and Data

- Two ways of sending data to GPU
  - vertex shader **attributes** → app vertex attributes
  - shader **uniforms** → app provided uniform values