Introduction to OpenGL

Vo Hoai Viet
vhviet@fit.hcmus.edu.vn

Outline

- What Is OpenGL?
- Evolution of the OpenGL Pipeline
- OpenGL Application Development
- Shaders and GLSL

What Is OpenGL?

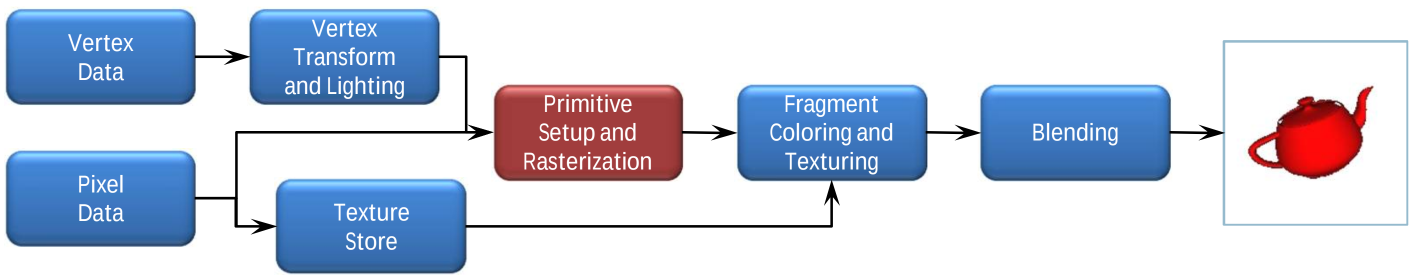
- OpenGL is a computer graphics rendering *application programming interface*, or API (for short)
  - With it, you can generate high-quality color images by rendering with geometric and image primitives
  - It forms the basis of many interactive applications that include 3D graphics
  - By using OpenGL, the graphics part of your application can be
    - operating system independent
    - window system independent

OpenGL

EVOLUTION OF THE OPENGL PIPELINE

In the Beginning ...

- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation



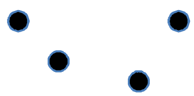
- The pipeline evolved
  - but remained based on fixed-function operation through OpenGL versions 1.1 through 2.0 (Sept. 2004)

OpenGL

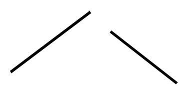
OPENGL PROGRAMMING 1.0

OpenGL's Geometric Primitives

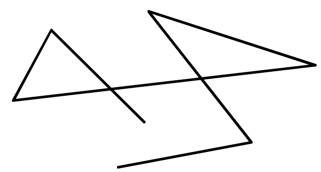
- OpenGL's Geometric Primitives



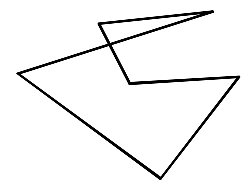
GL_POINTS



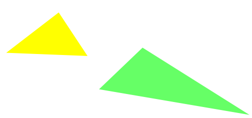
GL_LINES



GL_LINE_STRIP

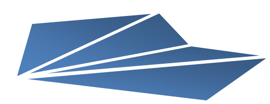


GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP

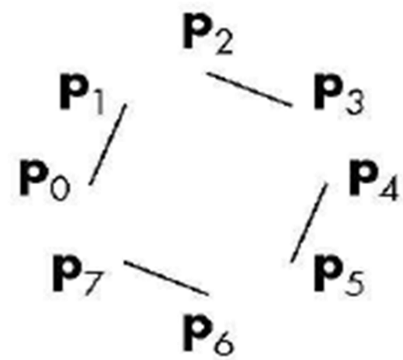


GL_TRIANGLE_FAN

Vertices and Primitives

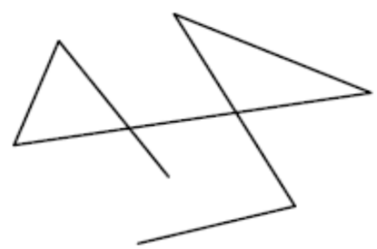
- Lines, **`GL_LINES`**
  - Pairs of vertices interpreted as individual line segments

GL_LINES

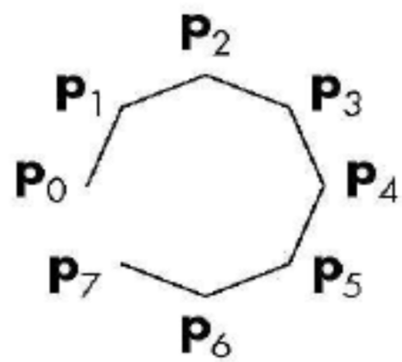


Vertices and Primitives

- Line Strip, **`GL_LINE_STRIP`**
  - series of connected line segments

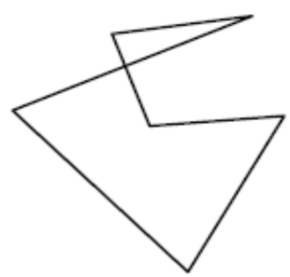


GL_LINE_STRIP

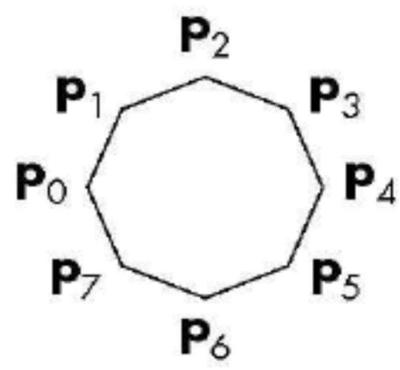


Vertices and Primitives

- Line Loop, **`GL_LINE_LOOP`**
  - Line strip with a segment added between last and first vertices



GL_LINE_LOOP

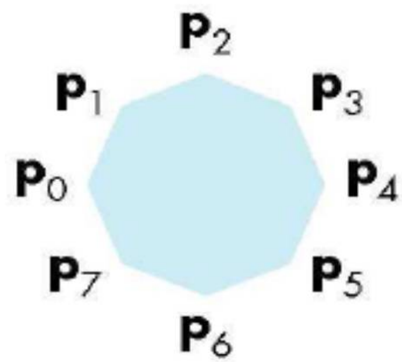


Vertices and Primitives

- Triangles, **`GL_TRIANGLES`**
  - triples of vertices interpreted as triangles

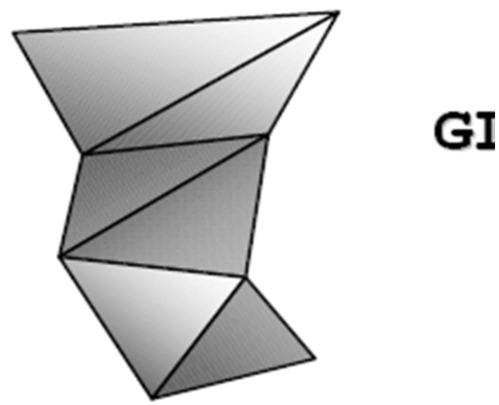
Vertices and Primitives

- Polygon, **`GL_POLYGON`**
  - boundary of a simple, convex polygon



Vertices and Primitives

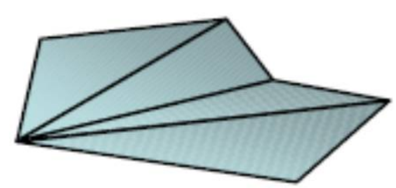
- Triangle Strip, **GL_TRIANGLE_STRIP**
  - linked strip of triangles



GL_TRIANGLE_STRIP

Vertices and Primitives

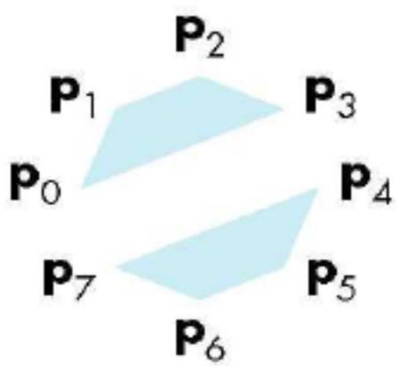
- Triangle Fan, **`GL_TRIANGLE_FAN`**
  - linked fan of triangles



GL_TRIANGLE_FAN

Vertices and Primitives

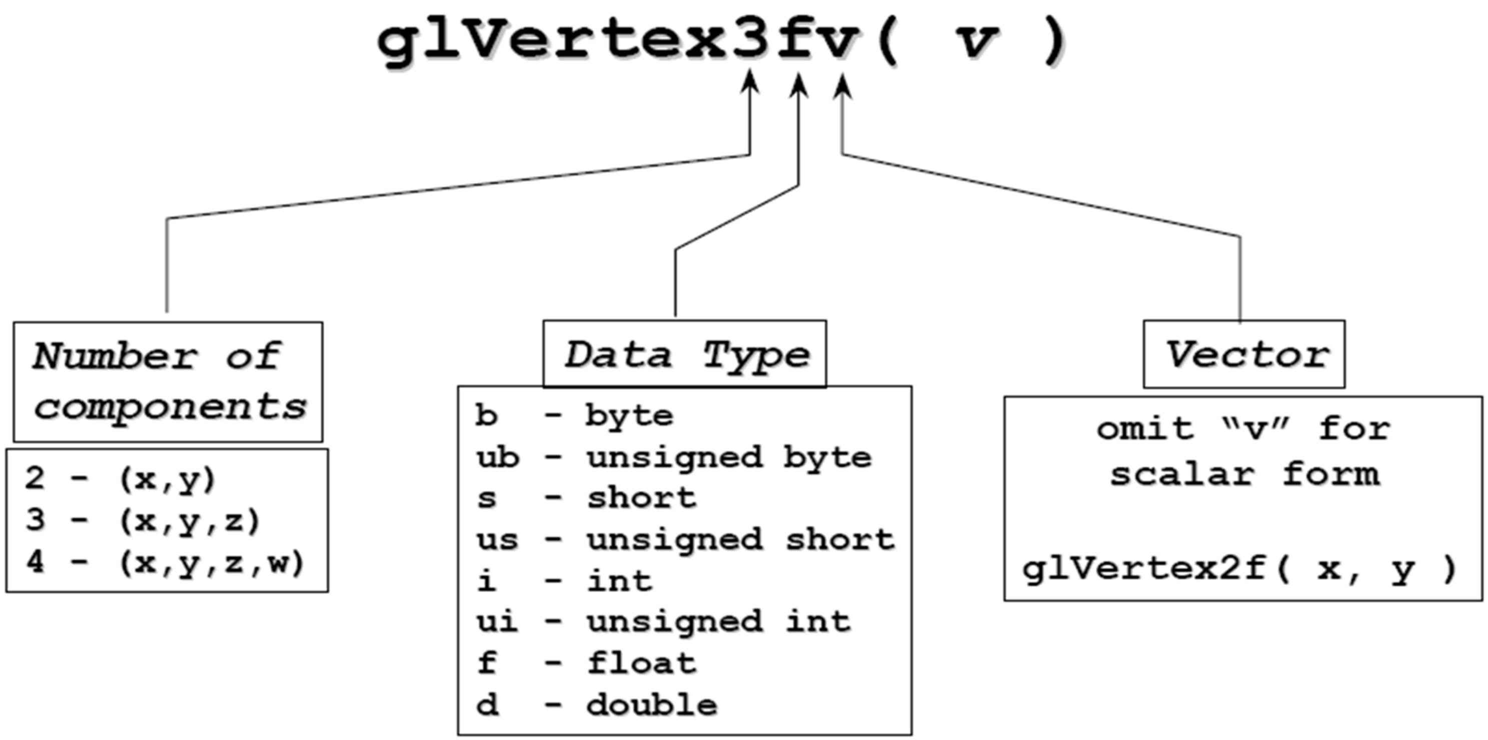
- Quads, **`GL_QUADS`**
  - quadruples of vertices interpreted as four-sided polygons



Vertices and Primitives

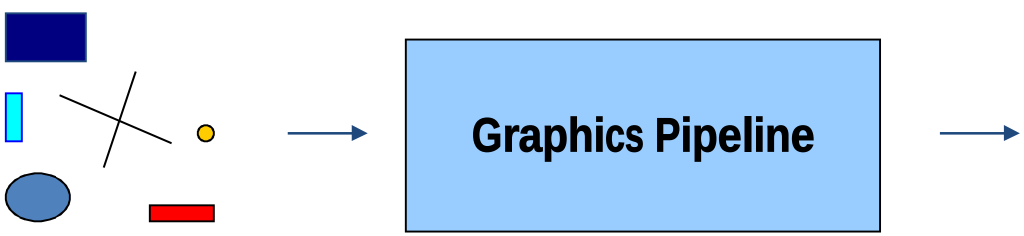
- Between glBegin/ glEnd, those opengl commands are allowed:
  - glVertex*() : set vertex coordinates
  - glColor*() : set current color
  - glIndex*() : set current color index
  - glNormal*() : set normal vector coordinates (Light.)
  - glTexCoord*() : set texture coordinates (Texture)

OpenGL Command Format



OpenGL Tutorial

- Rendering
  - Typically execution of OpenGL commands
  - Converting geometric/mathematical object descriptions into frame buffer values
- OpenGL can render:
  - Geometric primitives
    - Lines, points, polygons, etc…
  - Bitmaps and Images
    - Images and geometry linked through texture mapping



OpenGL and GLUT

- GLUT (OpenGL Utility Toolkit)
  - An auxiliary library
    - A portable windowing API
    - Easier to show the output of your OpenGL application
    - Not officially part of OpenGL
  - Handles:
    - Window creation,
    - OS system calls
      - Mouse buttons, movement, keyboard, etc…
    - Callbacks

How to install GLUT?

- Download GLUT
  - http://www.opengl.org/resources/libraries/glut.html
- Copy the files to following folders:
  - glut.h → VC/include/gl/
  - glut32.lib → VC/lib/
  - glut32.dll → windows/system32/
- Header Files:
  - #include <GL/glut.h>
  - #include <GL/gl.h>
  - Include glut automatically includes other header files

GLUT Tutorial

- Application Structure
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop

Sample Program

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

void main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( “Simple” );
    init();
    glutDisplayFunc( display );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

Sample Program

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

void main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc( display );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

Specify the display Mode – RGB or color Index, single or double Buffer

Sample Program

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

void main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc( display );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

**Create a window
Named "simple"
with resolution
500 x 500**

Sample Program

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

void main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc( display );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

Your OpenGL initialization code (Optional)

Sample Program

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

void main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
     glutInitDisplayMode( mode );
     glutInitWindowSize( 500,500 );
     glutCreateWindow( "Simple" );
     init();
     glutDisplayFunc( display );
    glutKeyboardFunc(key);
     glutMainLoop();
}
```

Register your call back functions

Sample Program

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

int main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode(mode);
    glutInitWindowSize(500,500);
    glutCreateWindow("Simple");
    init();
    glutDisplayFunc(display);
    glutKeyboardFunc(key);
    glutMainLoop();
}
```

The program goes into an infinite loop waiting for events

OpenGL Initialization

- Set up whatever state you're going to use
  - Don't need this much detail unless working in 3D

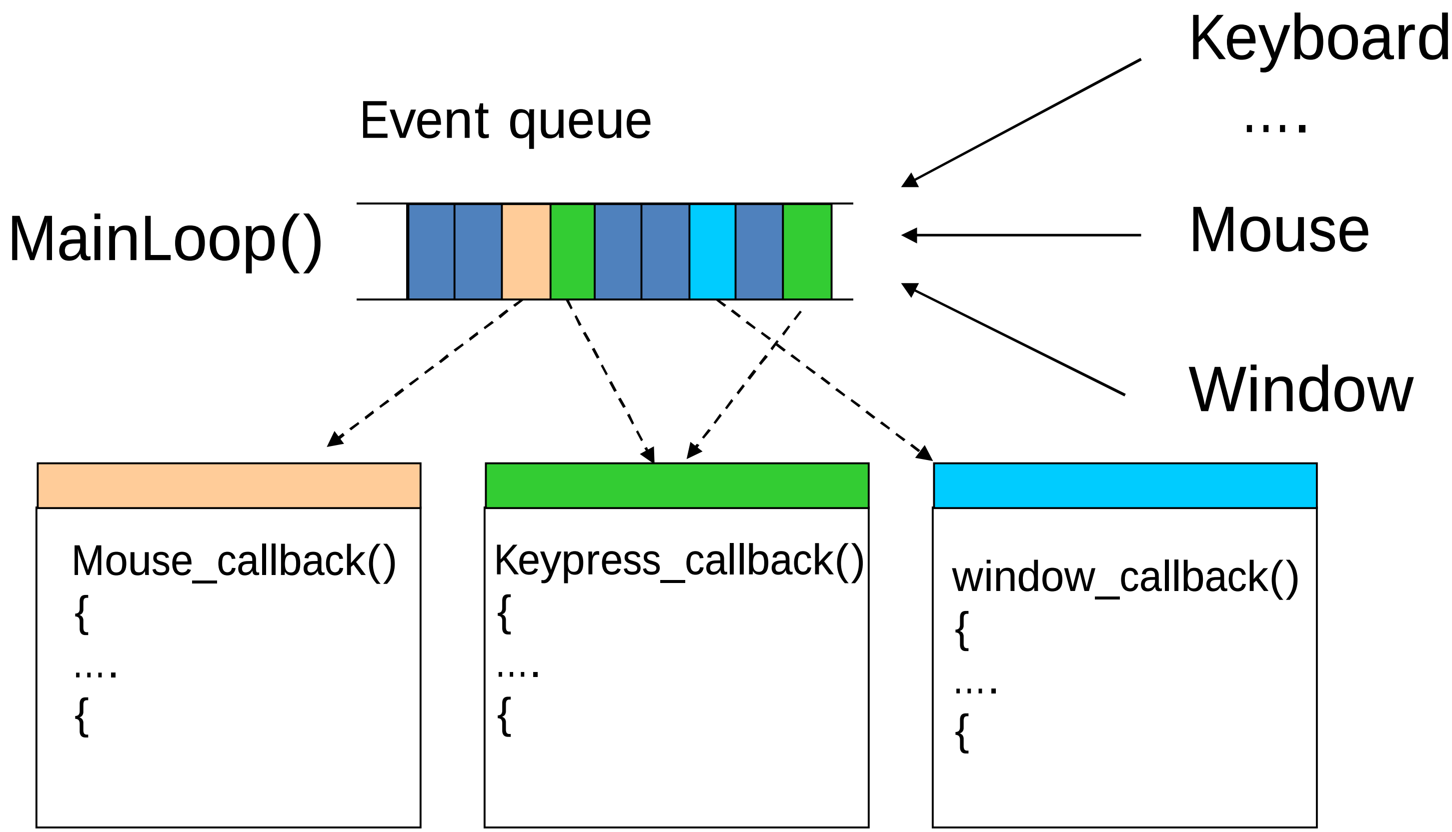
```
void init( void )
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-10, 10, -10, 10, -10, 20);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glEnable( GL_LIGHT0 );
    glEnable( GL_LIGHTING );
    glEnable( GL_DEPTH_TEST );
}
```

GLUT Callback functions

- **Event-driven:** Programs that use windows
  - Input/Output
  - Wait until an event happens and then execute some pre-defined functions according to the user's input
- **Events** – key press, mouse button press and release, window resize, etc.
- *Your OpenGL program will be in infinite loop*

Event Queue



Keyboard
....
Event queue
MainLoop()
Mouse
Window
Mouse_callback()
{
....
{
Keypress_callback()
{
....
{
window_callback()
{
....
{

Rendering Callback

- Callback function where all our drawing is done
- Every GLUT program must have a display callback
- glutDisplayFunc(***my_display_func***); /* this part is in main.c */

```
void my_display_func (void )
{
   glClear( GL_COLOR_BUFFER_BIT );
   glBegin( GL_TRIANGLE );
     glVertex3fv( v[0] );
     glVertex3fv( v[1] );
     glVertex3fv( v[2] );
   glEnd();
   glFlush();
}
```

Idle Callback

- Use for animation and continuous update
  - Can use *glutTimerFunc* or *timed callbacks* for animations
- glutIdleFunc(idle);

```
void idle( void )
{
        /* change
something */
    t += dt;
    glutPostRedisplay();
}
```

User Input Callbacks

- Process user input
- glutKeyboardFunc(my_key_events);

```
void my_key_events (char key, int x, int y )
{
    switch ( key ) {
      case ‘q’ : case ‘Q’ :
          exit ( EXIT_SUCCESS);
          break;
      case ‘r’ : case ‘R’ :
          rotate = GL_TRUE;
          break;
    }
}
```

Mouse Callback

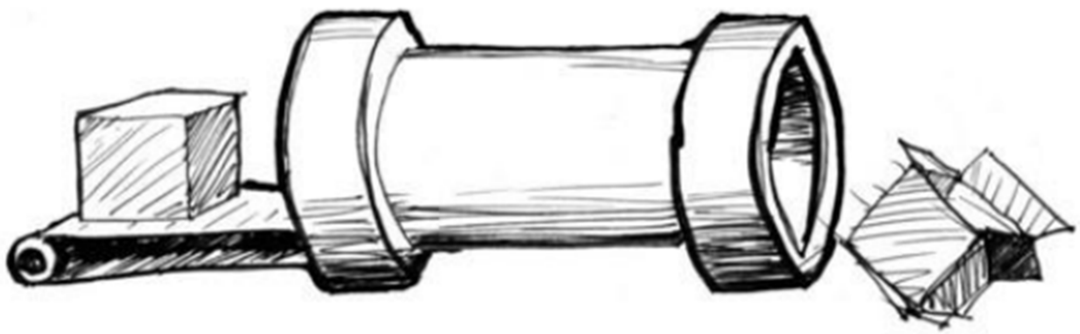
- Captures mouse press and release events
- glutMouseFunc(my_mouse);

```
void myMouse(int button, int state, int x, int y)
{
    if (button == GLUT_LEFT_BUTTON && state
      == GLUT_DOWN)
    {
    …
    }
}
```

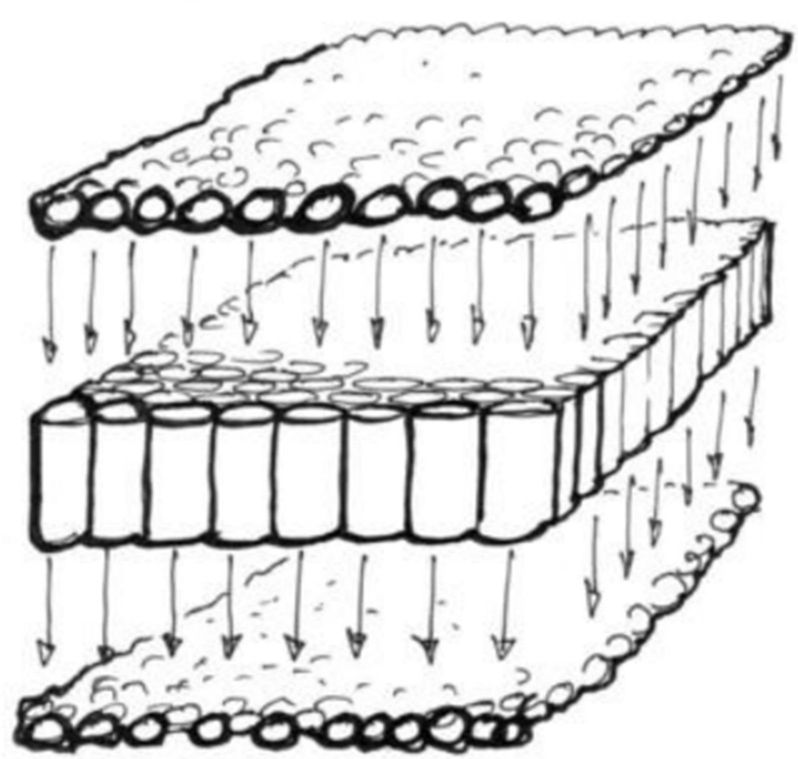
Events in OpenGL

Event	**Example**	**OpenGL Callback Function**
Keypress	KeyDown KeyUp	`glutKeyboardFunc`
Mouse	leftButtonDown leftButtonUp	`glutMouseFunc`
Motion	With mouse press Without	`glutMotionFunc` `glutPassiveMotionFunc`
Window	Moving Resizing	`glutReshapeFunc`
System	Idle Timer	`glutIdleFunc` `glutTimerFunc`
Software	What to draw	`glutDisplayFunc`

CPU Picture

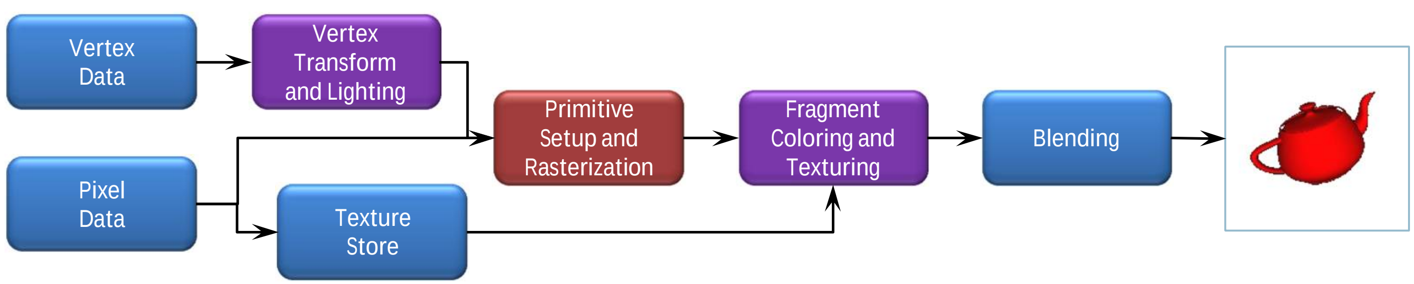


GPU Picture



Beginnings of The Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



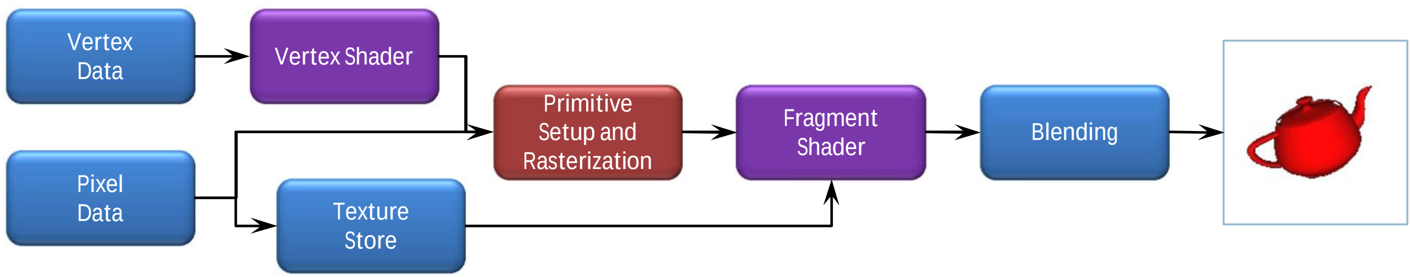
An Evolutionary Change

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

The Exclusively Programmable Pipeline

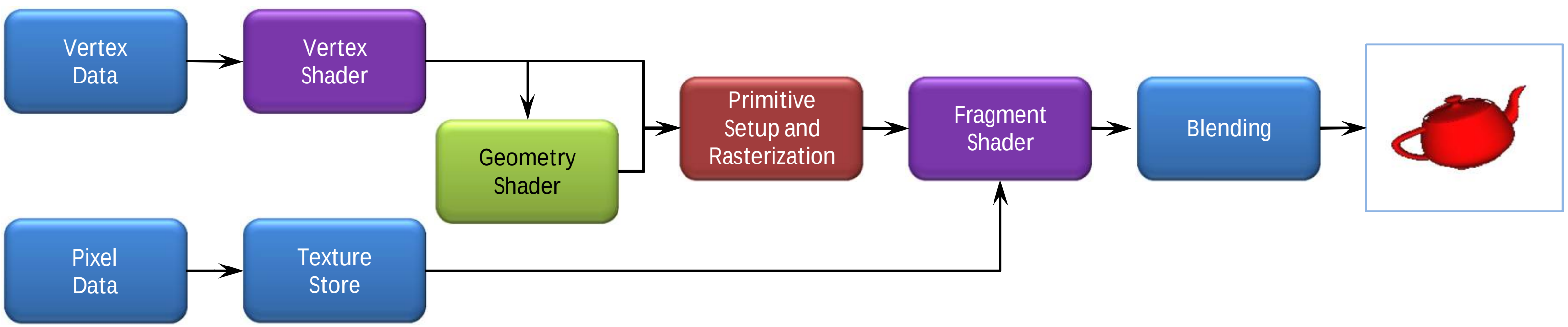
- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders



- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects

More Programmability

- OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – geometry shaders
  - modify geometric primitives within the graphics pipeline



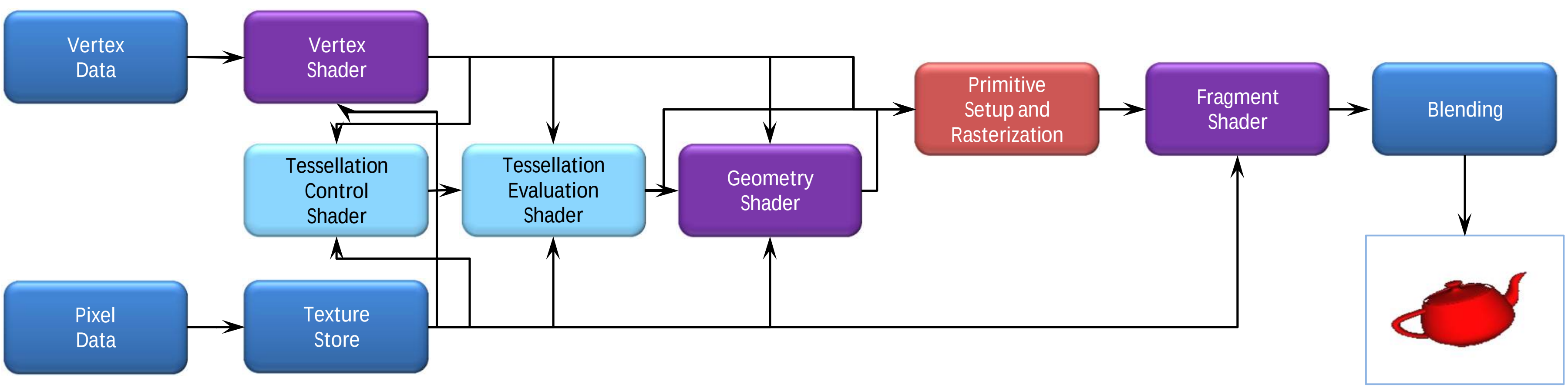
More Evolution – Context Profiles

- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
    - it's like `GL_ARB_compatibility`, only not insane ☺
  - currently two types of profiles: *core* and *compatible*

Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	Not supported

The Latest Pipelines

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders
- Latest version is 4.5



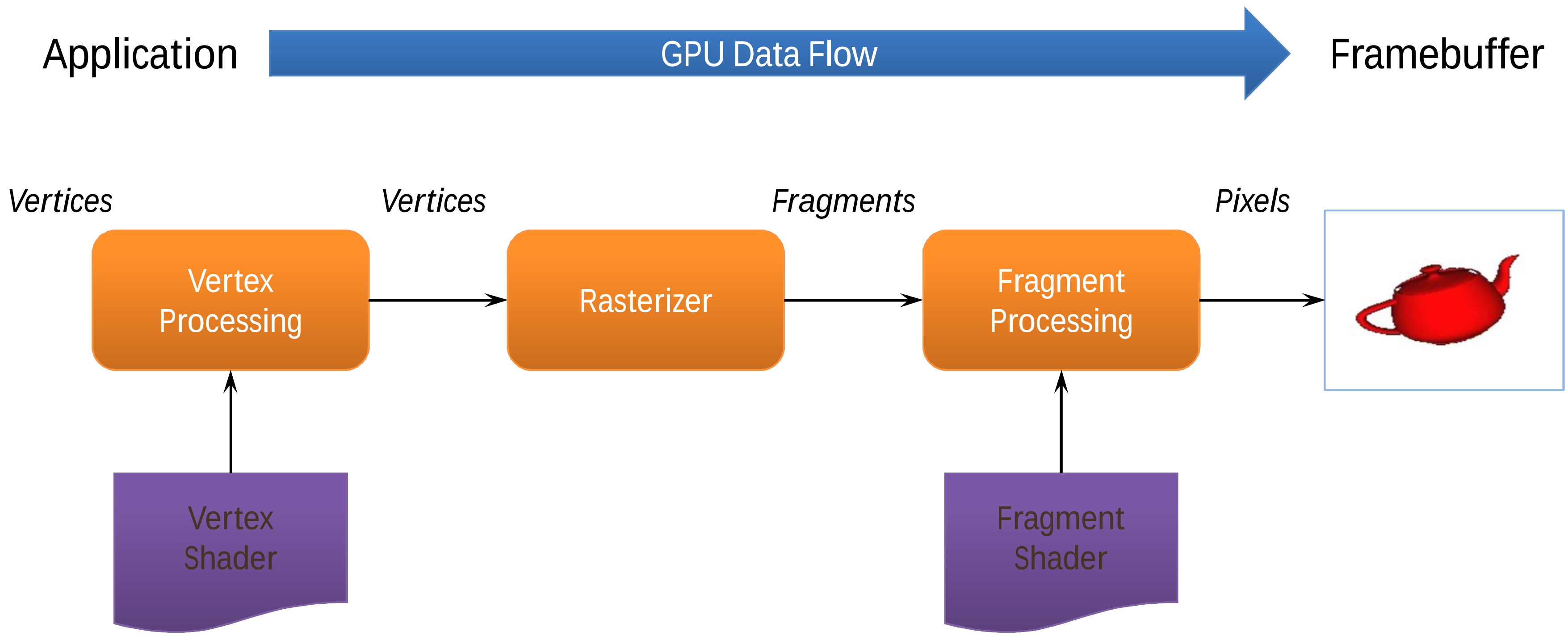
OpenGL ES and WebGL

- OpenGL ES 2.0
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- WebGL
  - JavaScript implementation of ES 2.0
  - Runs on most recent browsers

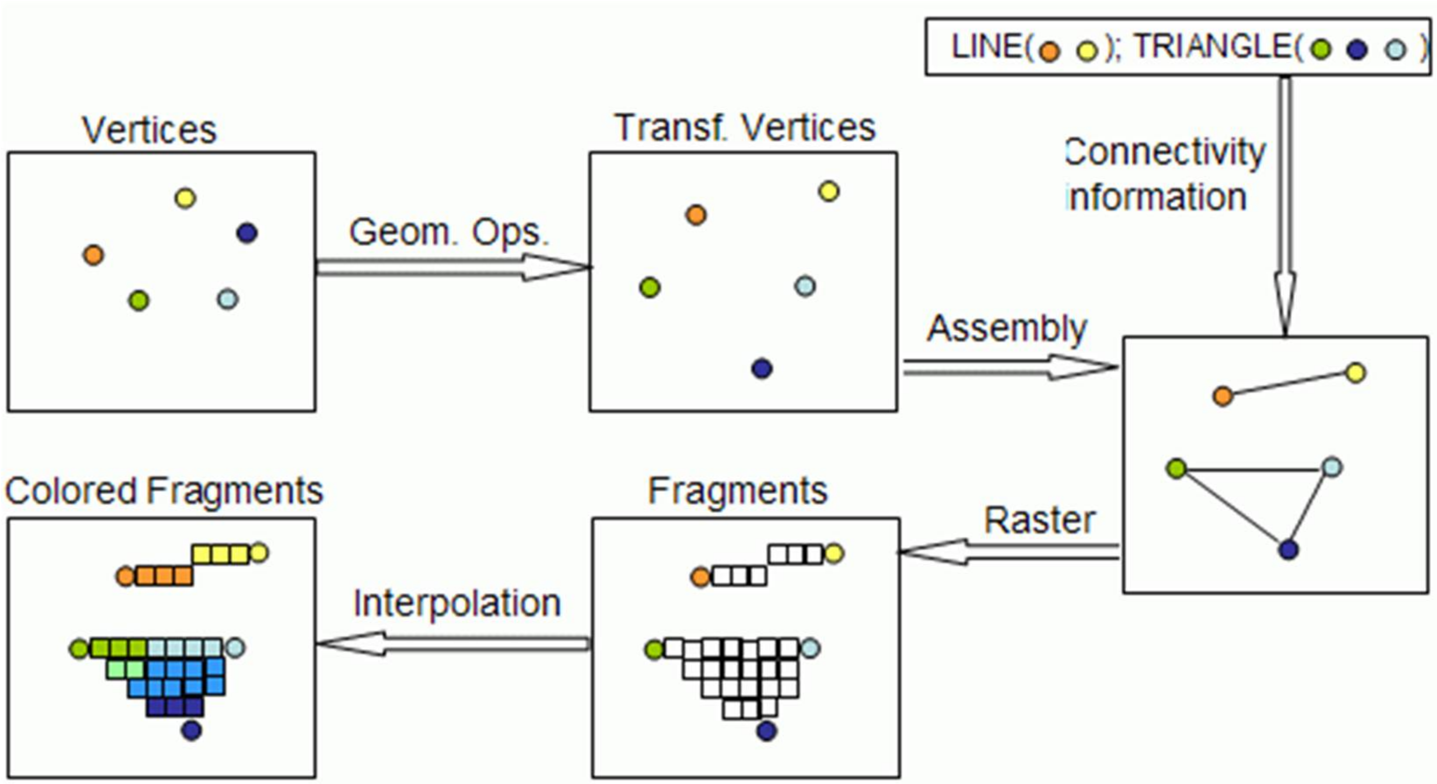
OpenGL

OPENGL APPLICATION DEVELOPMENT

A Simplified Pipeline Model



Graphic Pipeline



OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:
  - Create shader programs
  - Create buffer objects and load data into them
  - "Connect" data locations with shader variables
  - Render

Application Framework Requirements

- OpenGL applications need a place to render into
  - usually an on-screen window
- Need to communicate with native windowing system
- Each windowing system interface is different
- We use GLUT (more specifically, freeglut)
  - simple, open-source library that works everywhere
  - handles all windowing operations:
    - opening windows
    - input processing

http://www.transmissionzero.co.uk/software/freeglut-devel/

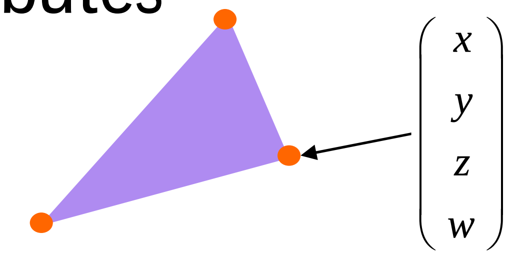
Simplifying Working with OpenGL

- Operating systems deal with library functions differently
  - compiler linkage and runtime libraries may expose different functions
- Additionally, OpenGL has many versions and profiles which expose different sets of functions
  - managing function access is cumbersome, and window-system dependent
- We use another open-source library, GLEW, to hide those details

http://glew.sourceforge.net/

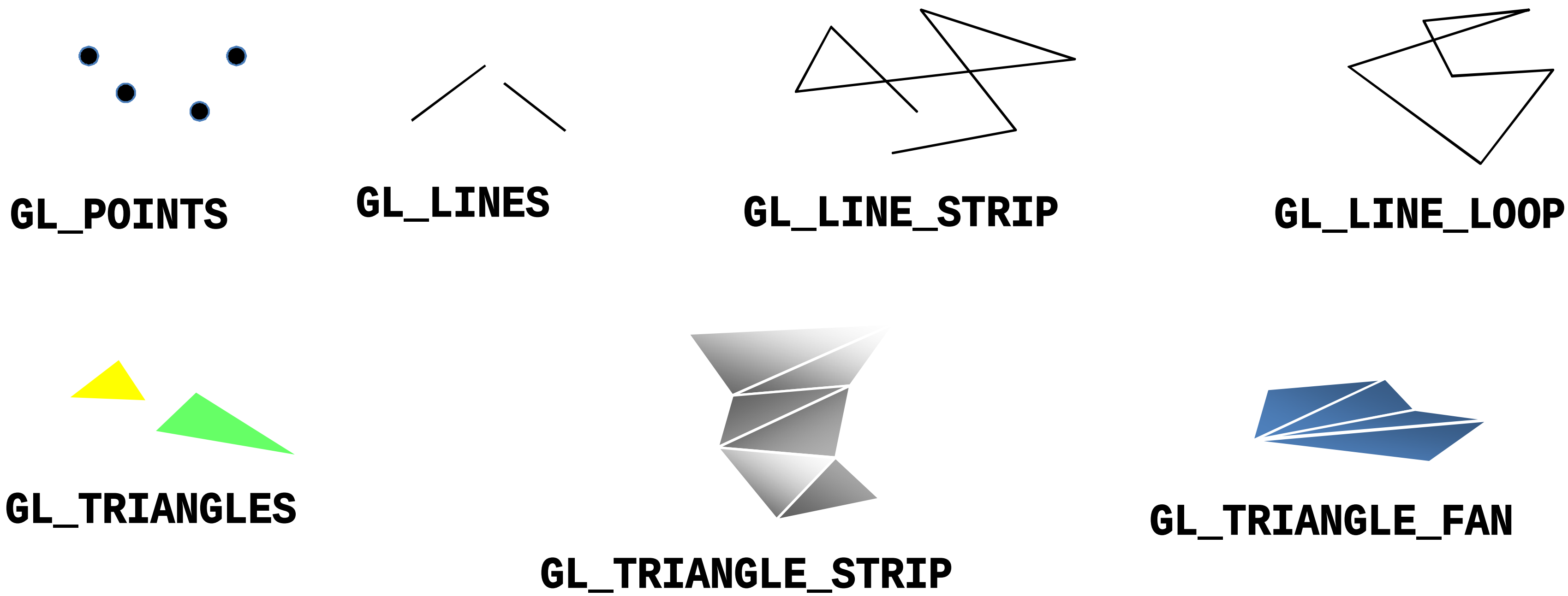
Representing Geometric Objects

- 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)



OpenGL's Geometric Primitives

- All primitives are specified by vertices



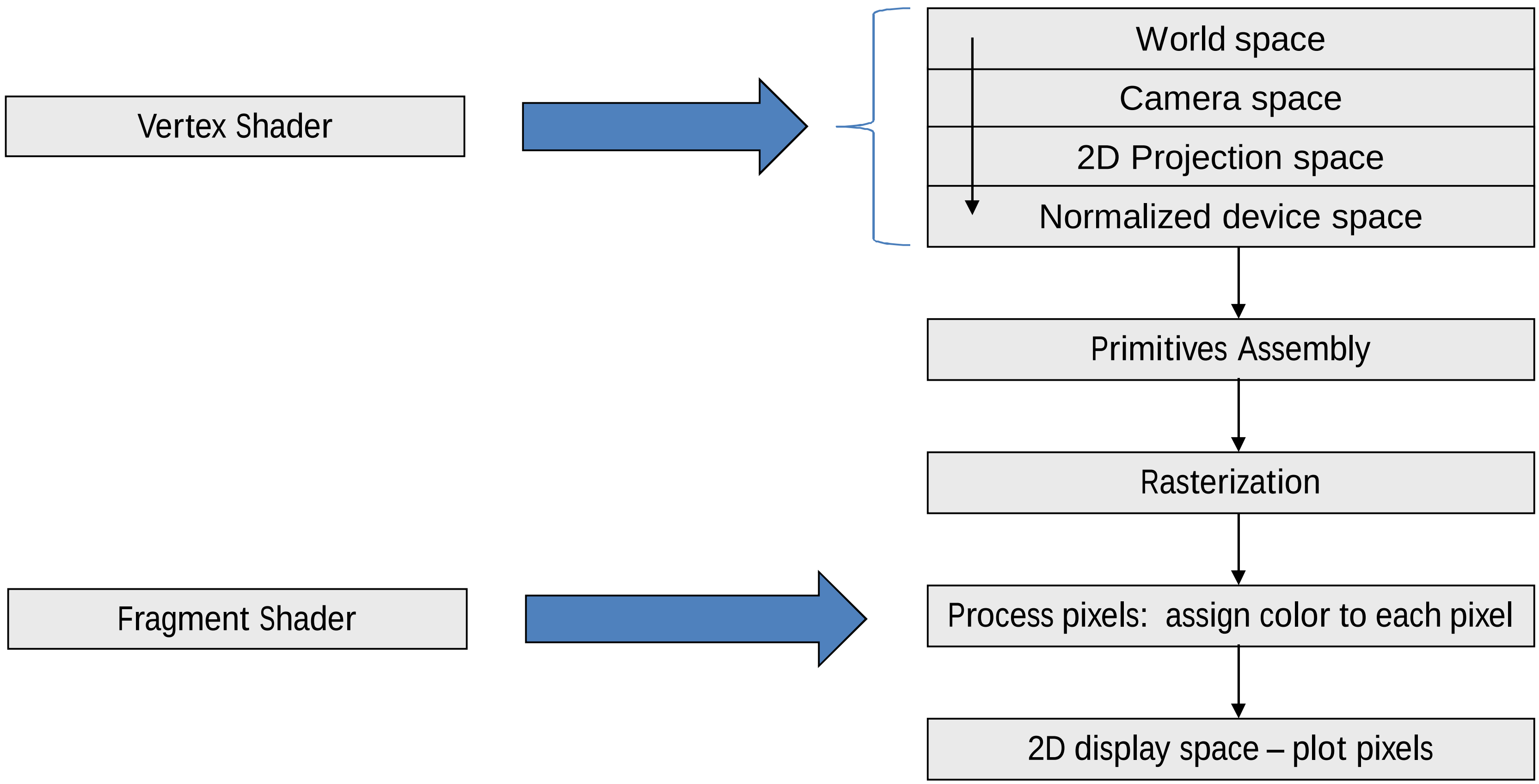
OpenGL

SHADERS AND GLSL

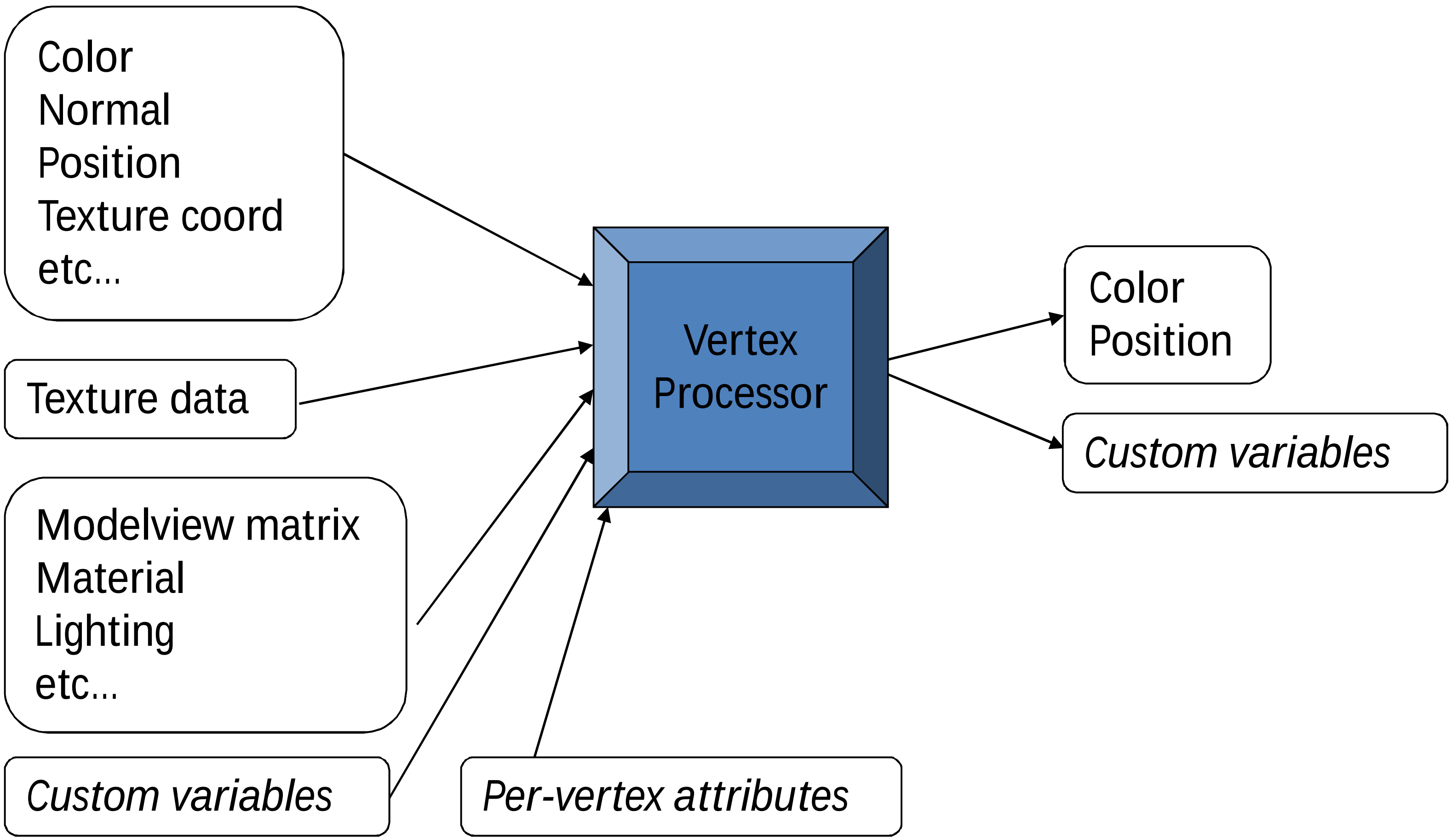
What is... the shader?

- The next generation:
  Introduce *shaders*, programmable logical units on the GPU which can replace the “fixed” functionality of OpenGL with user-generated code.
- By installing custom shaders, the user can now completely override the existing implementation of core per-vertex and perpixel behavior.

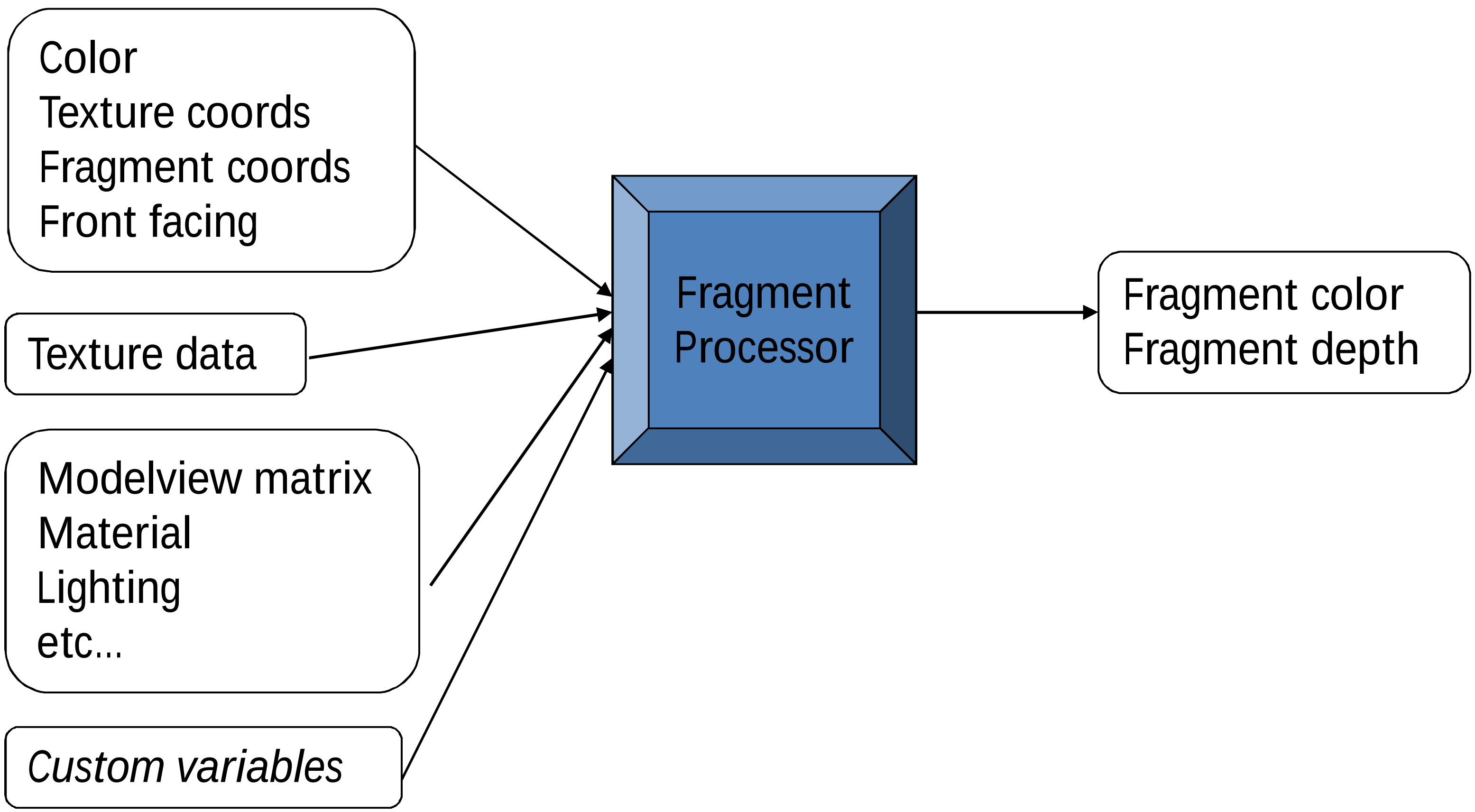
What is… the shader?



Vertex processor – inputs and outputs



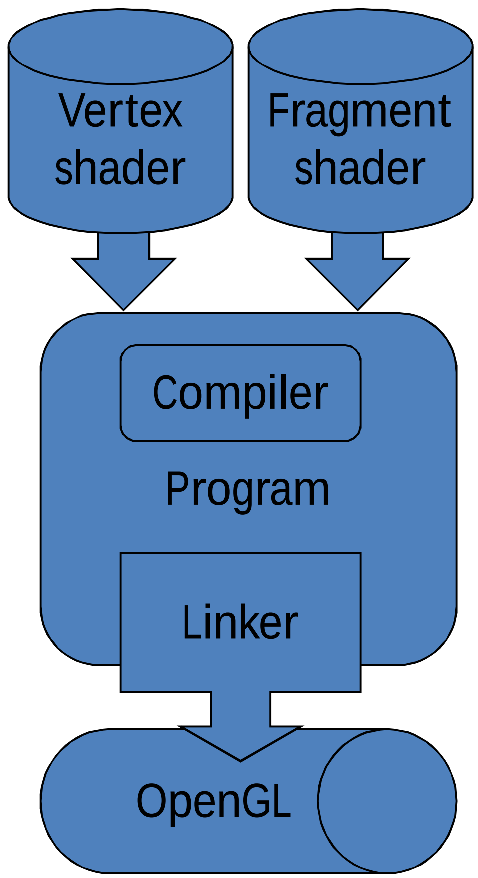
Fragment processor – inputs and outputs



How do the shaders communicate?

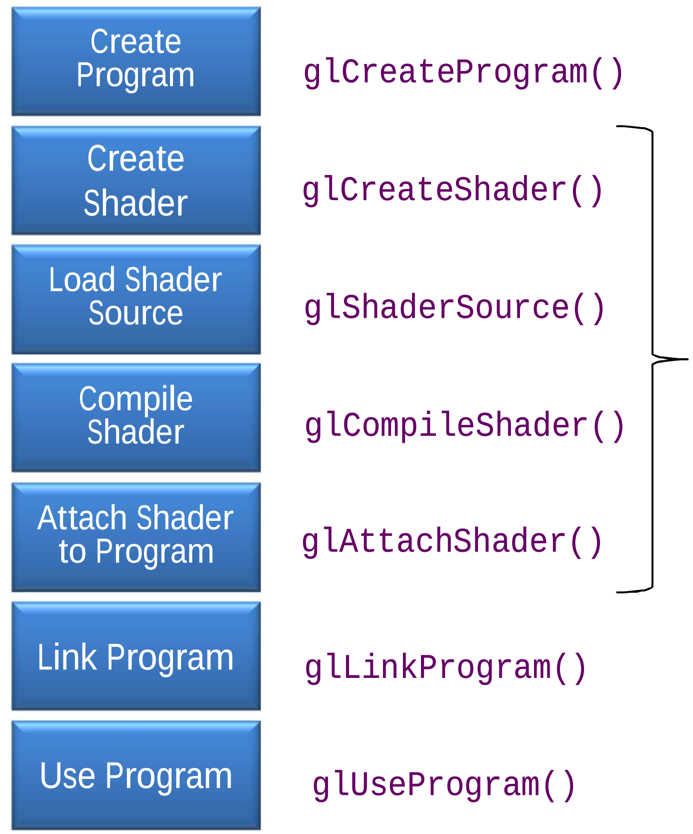
- There are three types of shader parameter in GLSL:
  - *Uniform parameters*
    - Set throughout execution
    - Ex: surface color
  - *Attribute parameters*
    - Set per vertex
    - Ex: local tangent
  - *Varying parameters*
    - Passed from vertex processor to fragment processor
    - Ex: transformed normal

Getting Your Shaders into OpenGL



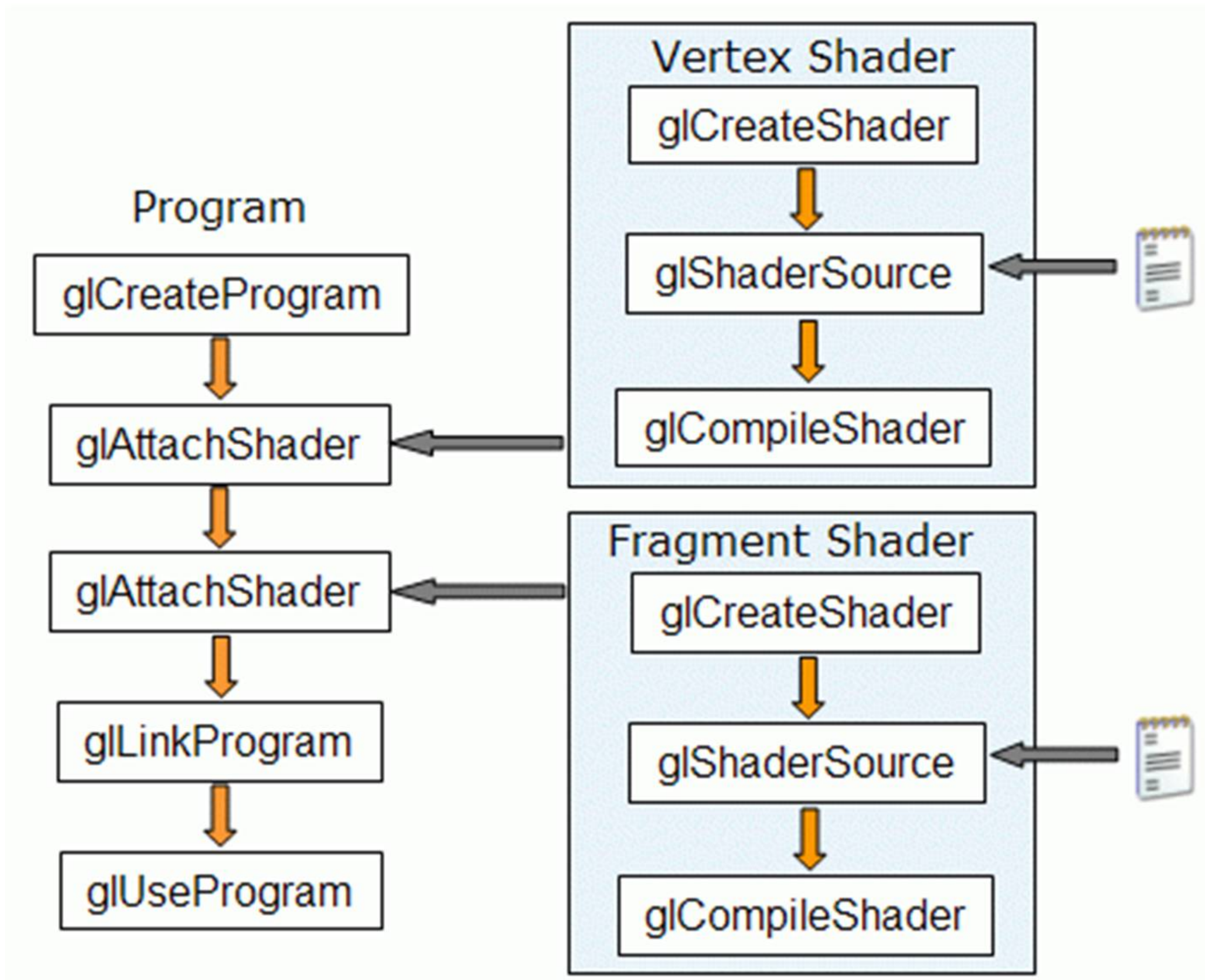
Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



These steps need to be repeated for each type of shader in the shader program

Shaders into OpenGL



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;
```

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;
```

Qualifiers

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

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

- `uniform`
  - shader-constant 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;

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

The Simplest Fragment Shader

```
#version 430

in vec4 color;

out vec4 fColor; // fragment’s final color

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

References

- An Introduction to OpenGL Programming, SIGGRAPH 2013
- *OpenGL Shading Language (GLSL),* Alex Benton, University of Cambridge