Interactive Computer Graphics: Lecture 5

Graphics APIs and Shading languages

Thanks to Markus Steinberger and
Dieter Schmalstieg, Dave Shreiner, Ed
Angel, Vicki Shreiner

Graphics APIs

Low-level 3D API

- OpenGL
- OpenGL ES
- DirectX, Direct3D
- Vulcan
- Mantle
- WebGL
- …

Graphics APIs

Low-level 3D API

- **OpenGL**
- OpenGL ES
- DirectX, Direct3D
- Vulcan
- Mantle
- WebGL
- …

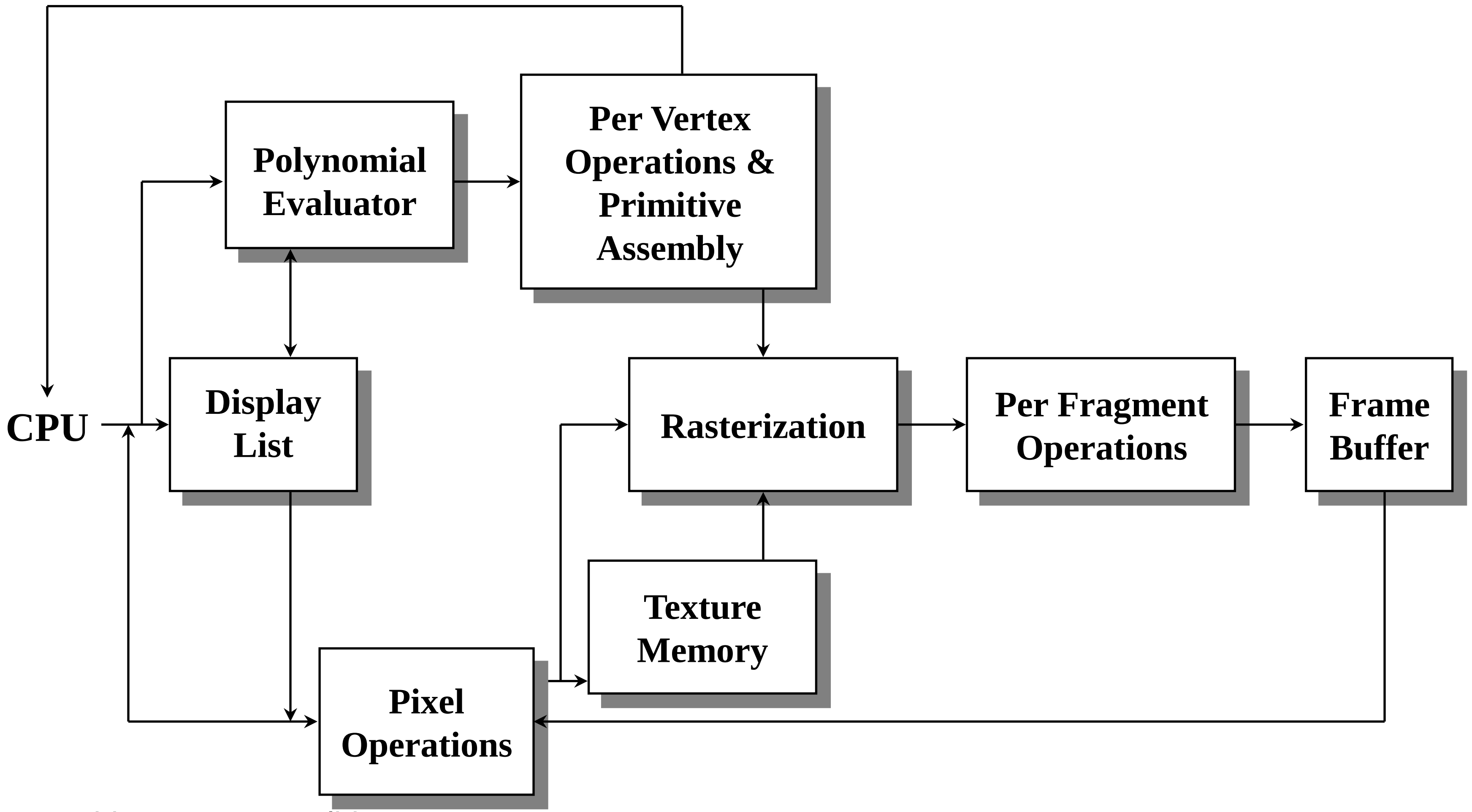
What is OpenGL?

- a low-level graphics API specification
  - not a library!
    - The interface is platform independent,
    - but the implementation is platform dependent.
  - Defines
    - an abstract rendering device.
    - a set of functions to operate the device.
  - "immediate mode" API
    - drawing commands
    - no concept of permanent objects

What is OpenGL?

- Platform provides OpenGL *implementation*.
  - Part of the graphics driver, or
  - runtime library built on top of the driver
- Initialization through platform specific API
  - WGL (Windows)
  - GLX (Unix/Linux)
  - EGL (mobile devices)
  - ...
- State machine for high efficiency!

OpenGL Architecture

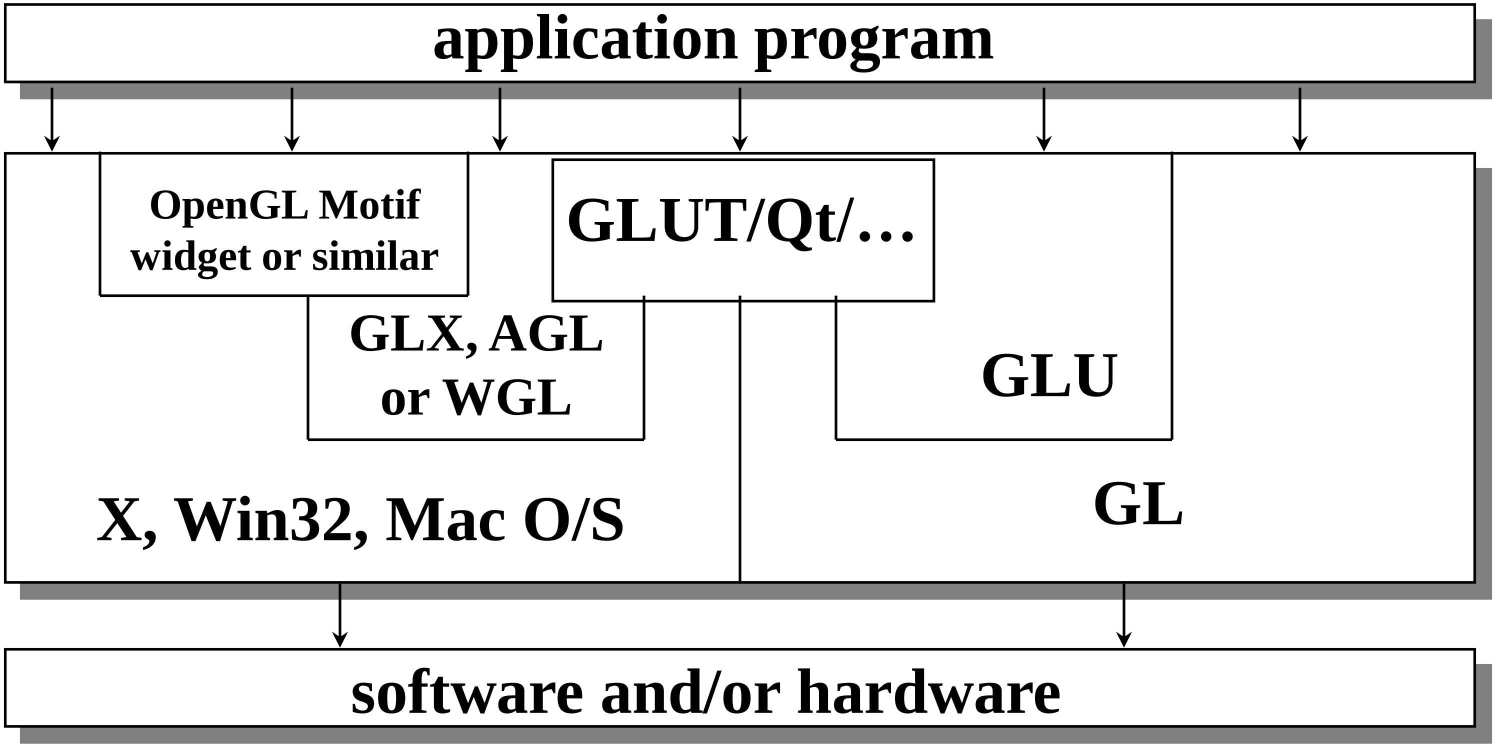


writing OpenGL programs

- Render window, i.e., context providing libraries (glut, Qt, browser SDKs etc.)

- setup and initialization functions
  – viewport
  – model transformation
  – file I/O (shader, textures, etc.)
- frame generation (update/rendering) functions
  – define what happens in every frame

OpenGL and Related APIs



Preliminaries

- Headers Files
    - #include <GL/gl.h>
    - #include <GL/glu.h>
    - #include <GL/glut.h>

    - Or in case of a Qt application
    - #include <QtOpenGL>
- https://www.opengl.org/resources/libraries/glut/spec3/spec3.html
- Enumerated Types
    - – OpenGL defines numerous types for compatibility
        - –GLfloat, GLint, GLenum, etc.

Preliminaries

- Easier with Qt but more overhead
- Headers Files
    - #include <QOpenGLWidget>
    - #include <QOpenGLFunctions>
    - …
- http://doc.qt.io/qt-5/qtopengl-index.html

OpenGL Basic Concepts

- Context
- Resources
- Object Model
  - Objects
  - Object Names
  - Bind Points (Targets)

Context

- Represents an instance of OpenGL
- A process can have multiple contexts
  - These can share resources
- A context can be *current* for a given thread
  - one to one mapping
    - only one current context per thread
    - context only current in one thread at the same time
  - OpenGL operations work on the current context

Resources

- Act as
  - sources of input
  - sinks for output
- Examples:
  - buffers
  - images
  - state objects
  - …

Resources

- Buffer objects
  - linear chunks of memory
- Texture images
  - 1D, 2D, or 3D arrays of *texels*
  - Can be used as input for *texture sampling*

Object Model

- OpenGL is object oriented
  - but in its own, strange way
- Object instances are identified by a *name*
  - basically just an unsigned integer handle
- Commands work on *targets*
  - Each target has an object currently *bound* to the target
    - That's the one commands will work with
- Object oriented, you said?
  - target ⇔ type
  - commands ⇔ methods

Object Model

- By binding a name to a target
  - the object it identifies becomes current for that target
    - “latched state”
    - change in OpenGL 4.5 (`EXT_direct_state_access`)
  - An object is created when a name is first bound.
- Notable exceptions: Shader Objects, Program Objects
  - Some commands work directly on object names.

Buffer Objects

- store an array of unformatted memory allocated by the OpenGL context (aka: the GPU)
- regular OpenGL objects
- can be used to store vertex data, pixel data retrieved from images or the framebuffer, and a variety of other things
- to set up its internal state, you must bind it to the context.

```
void glBindBuffer(enum target, uint bufferName)
```

- Immutable

```
void glBufferStorage(…);
```

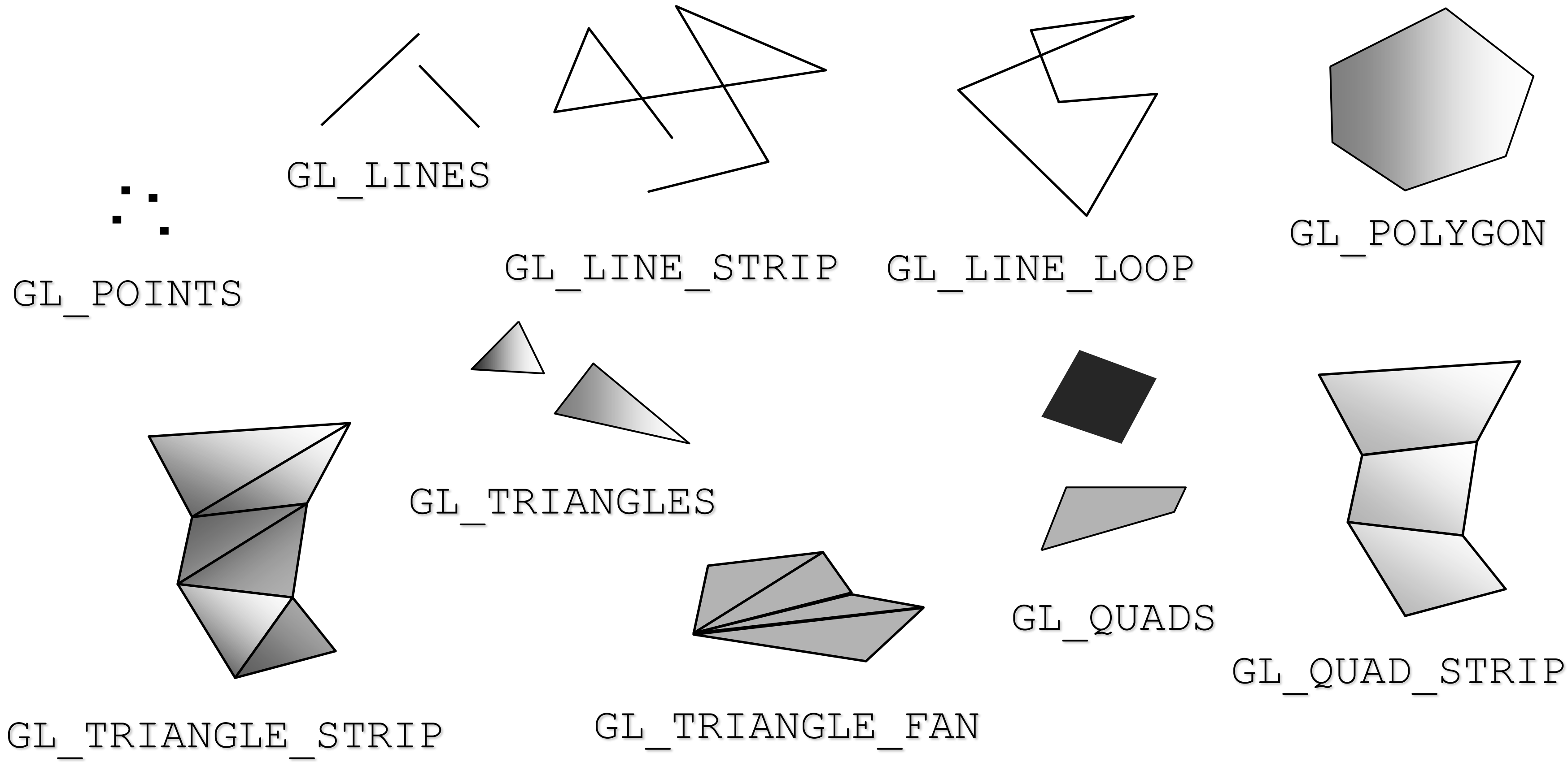
- or mutable depending on initialisation

```
void glBufferData(…)
```

Example: Buffer Object

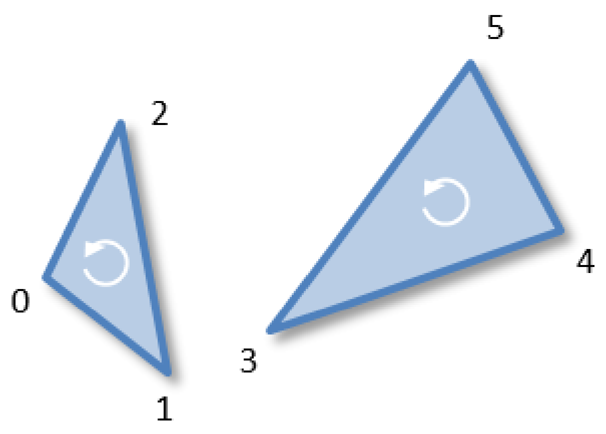
```
GLuint my_buffer;

// request an unused buffer object name
glGenBuffers(1, &my_buffer);

// bind name as GL_ARRAY_BUFFER
// bound for the first time ⇒ creates
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);

// put some data into my_buffer
glBufferStorage(GL_ARRAY_BUFFER, …);

// “unbind” buffer
glBindBuffer(GL_ARRAY_BUFFER, 0);

// probably do something else…
glBindBuffer(GL_ARRAY_BUFFER, my_buffer);
// use my_buffer…

glDrawArrays(GL_TRIANGLES, 0, 33);
// draw content example (type, startIdx, numer of elements)

// delete buffer object, free resources, release buffer object name
glDeleteBuffers(1, &my_buffer);
```

Primitive types

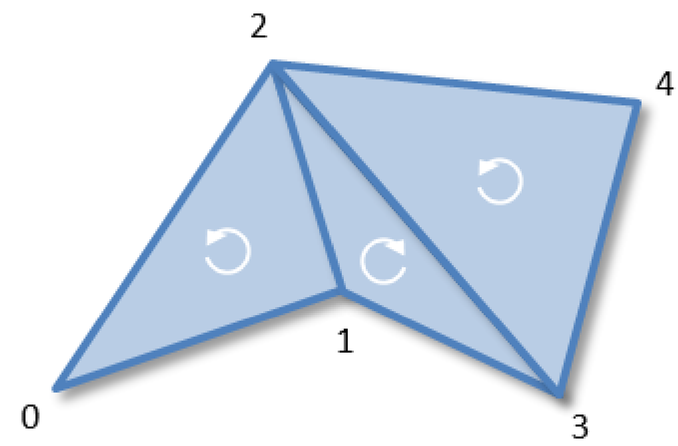


Primitive types

- triangle vertex orientations in OpenGL



GL_TRIANGLES



GL_TRIANGLE_STRIP

Draw Call

- After pipeline is configured:
  - issue *draw call* to actually draw something

primitive type

e. g.:

```
glBegin(GL_TRIANGLE_STRIP);
glColor3f(0.0, 1.0, 0.0);
glVertex3f(1.0, 0.0, 0.0);
…
glEnd();
```

Color “state”

vertex index

Buffer Objects -- drawing

- For continuous groups of vertices

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

- usually invoked in display callback
- initiates vertex shader

OpenGL Command Formats

glVertex3fv(*v*)

Number of components

```
2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

omit "v" for scalar form

```
glVertex2f( x, y )
```

writing (old) OpenGL programs

- pseudo example

```
#include <whateverYouNeed.h>

main() {
   InitializeAWindowPlease();

   glClearColor (0.0, 0.0, 0.0, 0.0);
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

   registerDisplayCallback(
     UpdateTheWindowAndCheckForEvents())
}

UpdateTheWindowAndCheckForEvents(){
glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
}
```

Matrix stack (old OpenGL)

- There used to be a stack of matrices for each of the matrix modes.
- The current transformation matrix in any mode is the matrix on the top of the stack for that mode.
- glPushMatrix pushes the current matrix stack down by one, duplicating the current matrix.
- glPopMatrix pops the current matrix stack, replacing the current matrix with the one below it on the stack.
- Initially, each of the stacks contains one matrix, an identity matrix.
- used to ‘save’ transformation state

Example Textures

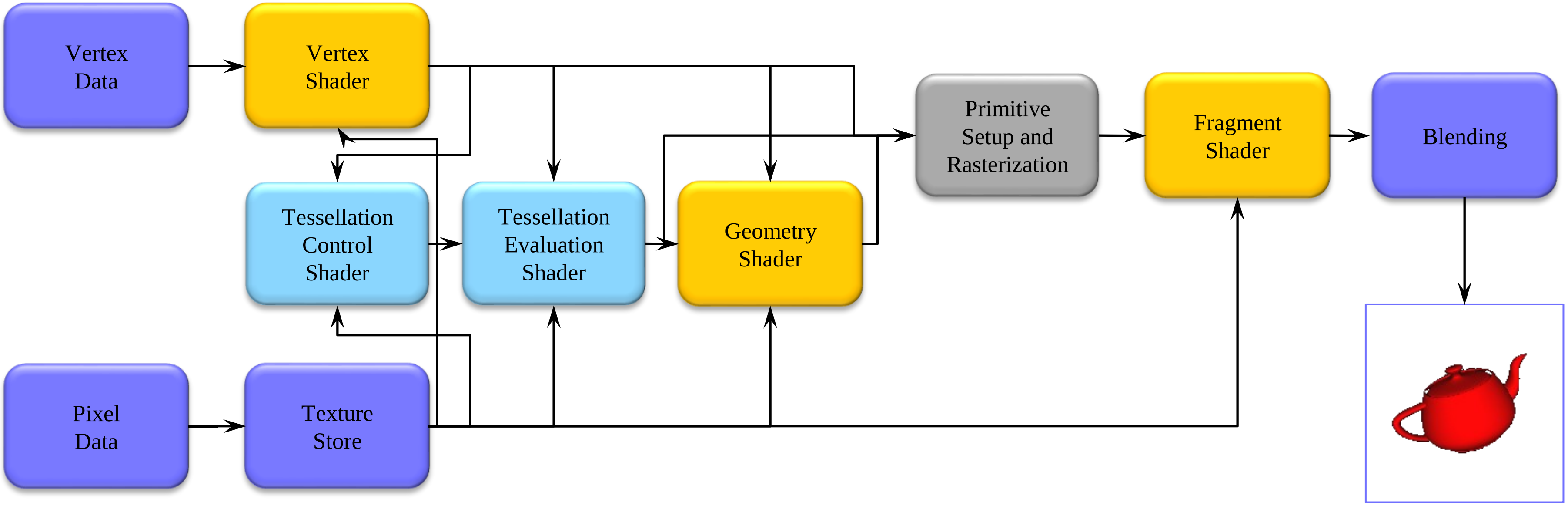
```
glEnable(GL_TEXTURE_2D);
glActiveTexture(GL_TEXTURE0);
textureImage = readPPM("pebbles_texture.ppm");
glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D, tex);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, textureImage->x,
textureImage->y, 0, GL_RGB, GL_UNSIGNED_BYTE, textureImage-
>data);
glBindTexture(GL_TEXTURE_2D, 0);
...
glBindTexture(GL_TEXTURE_2D, tex);
glutSolidTeapot(0.5);
glBindTexture(GL_TEXTURE_2D, 0);
```

OpenGL 4

- Enforces a new way to program with OpenGL
  - Allows more efficient use of GPU resources
- In contrast to “classic” graphics pipelines, modern OpenGL doesn’t support
  - Fixed-function graphics operations
    - Lighting, transformations, etc.
- All applications must use shaders and buffers for their graphics processing

OpenGL 4

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control and tessellation-evaluation shaders*
- Latest version is 4.3



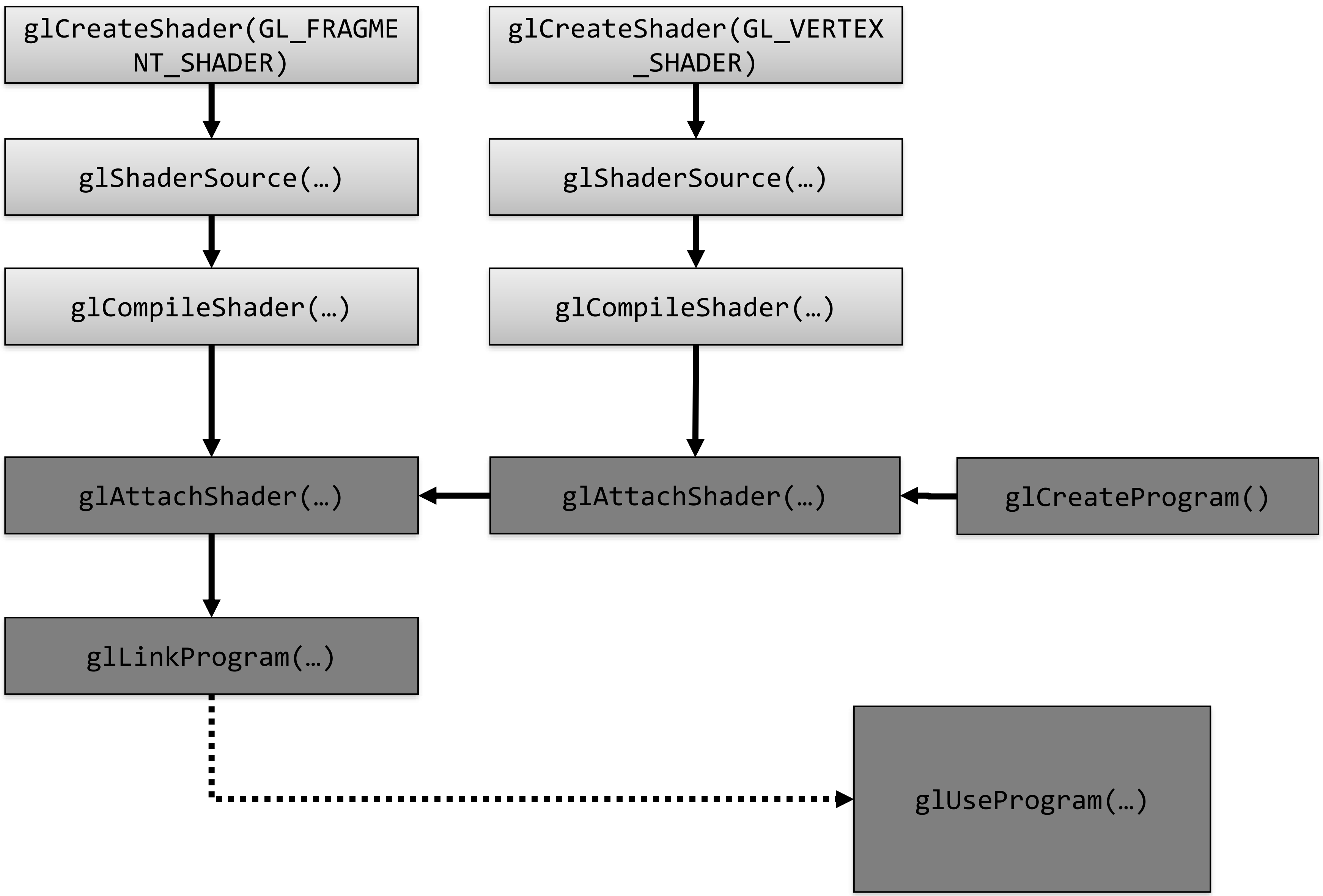
OpenGL 4

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

Shaders

- Shader Objects
  - parts of a pipeline (Vertex Shader, Fragment Shader, etc.)
  - compiled during runtime from GLSL code
    - OpenGL Shading Language
    - C-like syntax
- Program Object
  - a whole pipeline
  - Shader objects linked together during runtime
- OpenGL shader language: GLSL

Shaders


glCreateShader(GL_FRAGMENT_SHADER)
glShaderSource(…)
glCompileShader(…)
glCreateShader(GL_VERTEX_SHADER)
glShaderSource(…)
glCompileShader(…)
glAttachShader(…)
glAttachShader(…)
glCreateProgram()
glLinkProgram(…)
glUseProgram(…)

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
- Operators overloaded for matrix and vector operations

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

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

Components and Swizzling

For vectors can use [ ], xyzw, rgba or stpq

Example:

```
vec3 v;
```

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

Swizzling:

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

Qualifiers

- in, out
  - Copy vertex attributes and other variables to/from shaders
  - `in vec2 tex_coord;`
  - `out vec4 color;`
- Uniform: variable from application
  - `uniform float time;`
  - `uniform vec4 rotation;`

Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for

Functions

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

Built-in Variables

- `gl_Position`: output position from vertex shader
- `gl_FragColor`: output color from fragment shader
  - Only for ES, WebGL and older versions of GLSL
  - Present version use an out variable

Anatomy of a GLSL Shader

```
#version 400

uniform mat4 some_uniform;

layout(location = 0) in vec3 some_input;
layout(location = 1) in vec4 another_input;

out vec4 some_output;

void main()
{

}
```

Set by application (configuration values, e.g. ModelViewProjection Matrix)

Optional flexible register configuration between shaders

Output definition for next shader stage

Vertex Shader

- Processes each vertex
- Input: vertex attributes
- Output: vertex attributes
  - gl_Position

Rasterizer

- Fixed-function
- Rasterizes primitives
- Input: primitives
  - vertex attributes
- Output: fragments
  - interpolated vertex attributes

Fragment Shader

- Processes each fragment
- Input: interpolated vertex attributes
- Output: fragment color

Fragment Shader

- Interface to fixed-function parts of the pipeline (shader model > 4 – OpenGL4 requires to define these).
  - e. g. Vertex Shader:
    - `in int gl_VertexID;`
    - `out vec4 gl_Position;`
  - e. g. Fragment Shader:
    - `in vec4 gl_FragCoord;`
    - `out float gl_FragDepth;`

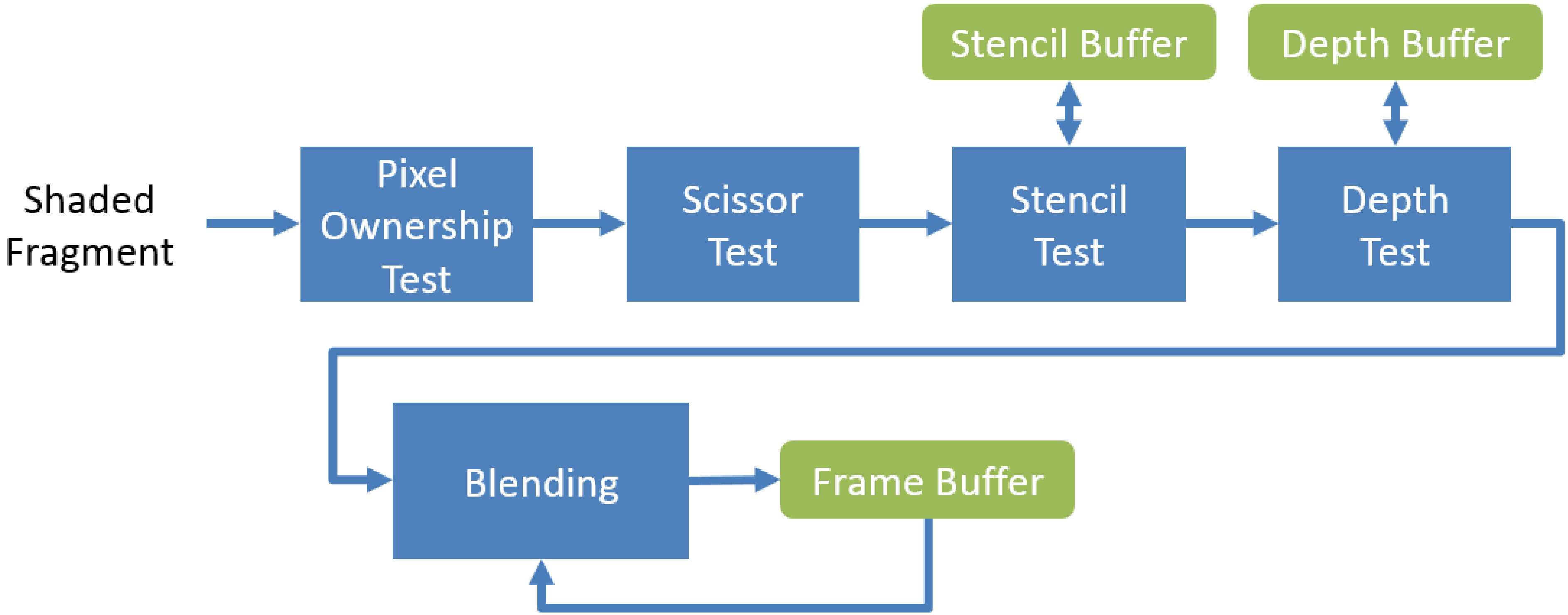
Example: Vertex Shader

```
#version 400

uniform mat4 mvMatrix;
uniform mat4 pMatrix;
uniform mat3 normalMatrix; //mv matrix without translation

uniform vec4 lightPosition_camSpace; //light Position in camera space

in vec4 vertex_worldSpace;
in vec3 normal_worldSpace;
in vec2 textureCoordinate_input;

out data
{
  vec4 position_camSpace;
  vec3 normal_camSpace;
  vec2 textureCoordinate;
  vec4 color;
}vertexIn;

//Vertex shader compute the vectors per vertex
void main(void)
{
  //Put the vertex in the correct coordinate system by applying the model view matrix
  vec4 vertex_camSpace = mvMatrix*vertex_worldSpace;
  vertexIn.position_camSpace = vertex_camSpace;
  //Apply the model-view transformation to the normal (only rotation, no translation)
  //Normals put in the camera space
  vertexIn.normal_camSpace = normalize(normalMatrix*normal_worldSpace);
  //Color chosen as red
  vertexIn.color = vec4(1.0, 0.0, 0.0, 1.0);
  //Texture coordinate
  vertexIn.textureCoordinate = textureCoordinate_input;
  gl_Position = pMatrix * vertex_camSpace;
}
```

Example: Fragment Shader

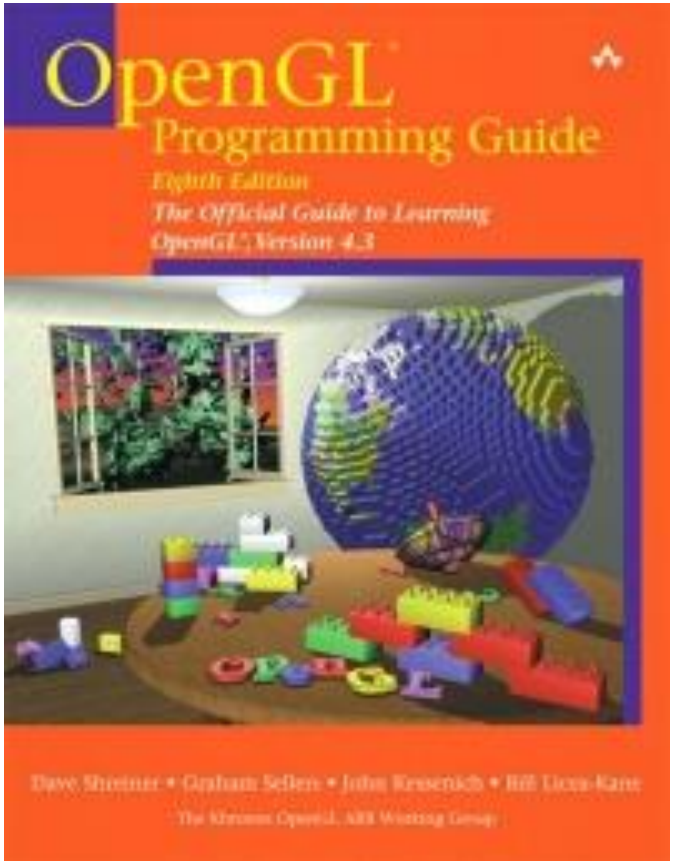
```
#version 400

uniform vec4 ambient;
uniform vec4 diffuse;
uniform vec4 specular;
uniform float shininess;

uniform vec4 lightPosition_camSpace; //light Position in camera space

in fragmentData
{
  vec4 position_camSpace;
  vec3 normal_camSpace;
  vec2 textureCoordinate;
  vec4 color;
} frag;

out vec4 fragColor;

//Fragment shader computes the final color
void main(void)
{
  fragColor =  frag.color;
}
```

Fragment Merging



Please read the OpenGL Programming Guide

- free full online version: http://www.glprogramming.com/red



OpenGL ES (Embedded Systems)

- OpenGL is just too big for embedded systems like mobile devices
- compact API, purely shader-based