CSE-170 Computer Graphics

Lecture 3

OpenGL

Dr. Renato Farias
rfarias2@ucmerced.edu

OpenGL

- Abstract API for drawing 2D and 3D graphics
- OpenGL is *only* concerned with rendering, another library must be used for interfacing with the windowing system, handling input, doing audio, etc.
  - In this course, that other library will be **freeglut**, but there are many alternatives (GLFW, SFML, Qt, etc.)

OpenGL

- OpenGL has a **state-based design**
- API calls tend to either
  1. query the current state
  2. modify the current state
  3. use the current state to render something
- This applies to OpenGL objects, too:
  - To modify an object, we need to bind it to the state system, then use function calls to use it or modify the state

OpenGL

- The collection of state information and OpenGL objects is called an **OpenGL context**
  - Almost always, this represents the one window our application is rendering to
  - An application *may* have more than one context or window, but it's uncommon

Older OpenGL

Version 1.0 was released on 6/30/1992.

Before version 2.0 (9/7/2004):

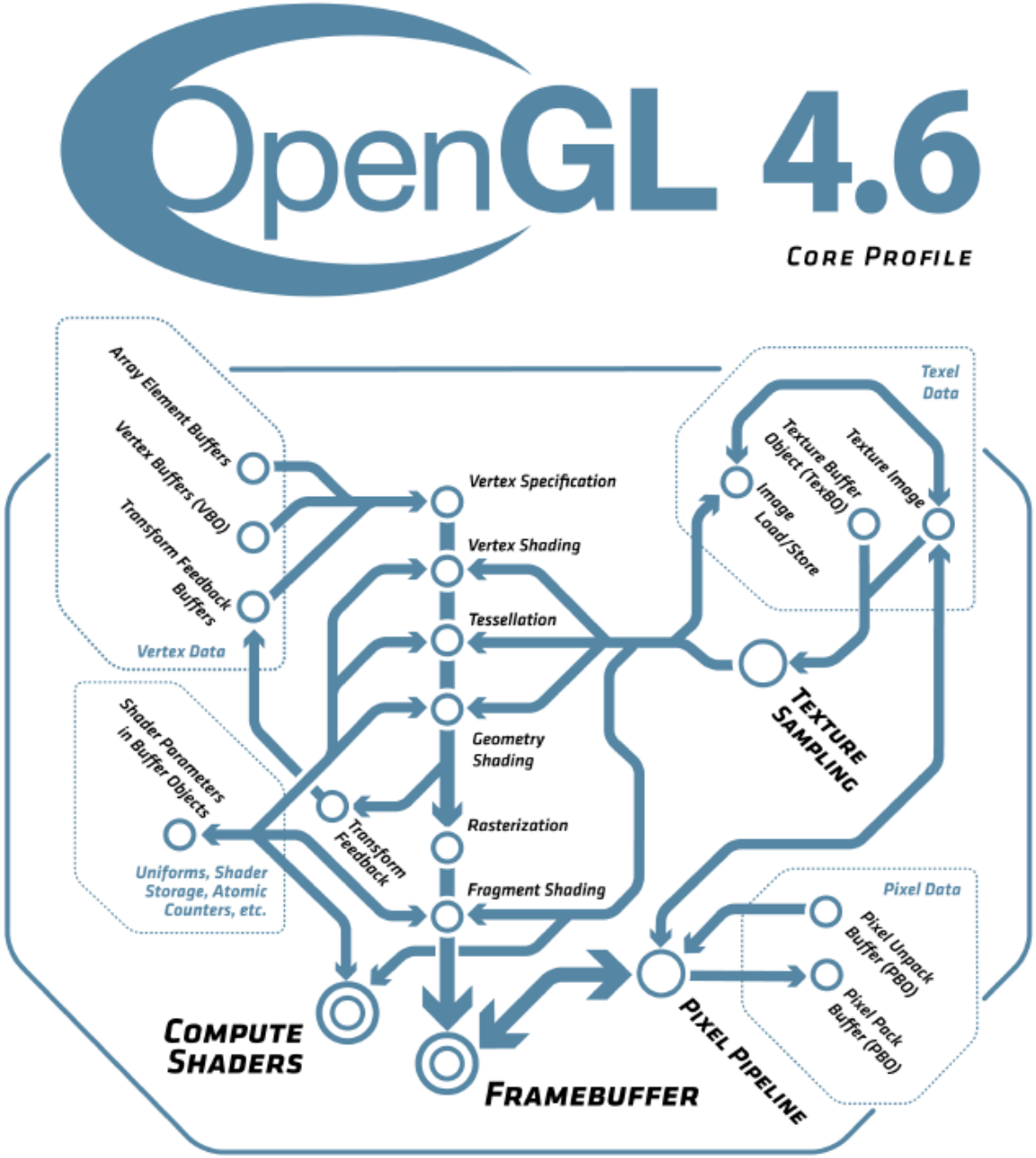
- Fixed-function pipeline
  - No stages were not programmable yet

Before version 3.0 (8/11/2008):

- Immediate mode was not deprecated yet, and so was still heavily used
  - glBegin/glEnd

Modern OpenGL (3.0 onwards)

- Modern OpenGL is based on user-defined buffer objects and programmable shaders
  - Shaders are C-like programs that are executed at the main stages of the rendering pipeline
  - Main ones: vertex and fragment shaders
- Newer versions are highly flexible
  - Programmer can implement/change several stages of the pipeline
  - Very flexible, efficient, and controllable, but more work for the programmer
  - Allows for loading extensions (for ex, using GLEW)



Cover page of glspec46.core.pdf (https://registry.khronos.org/OpenGL/specs/gl/)

Simplified OpenGL Pipeline View

You have to provide at least these “shaders”:
(C-like programs in the GLSL language)

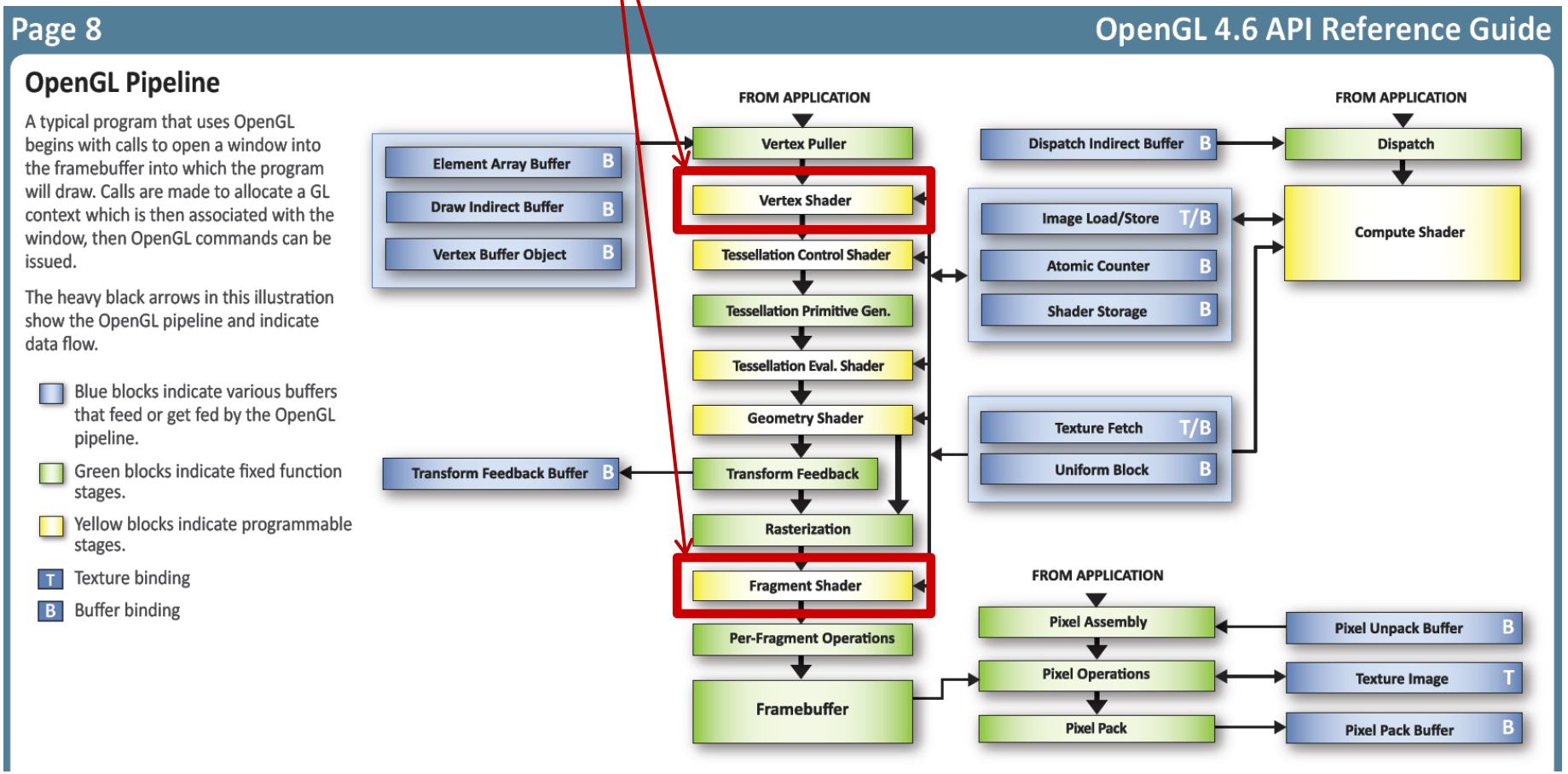


diagram from https://www.khronos.org/files/opengl46-quick-reference-card.pdf

Creating a window

- Before we render anything, we need to create a **context** (window)

```
int main( int argc, char** argv )
{
    glutInit( &argc, argv );

    glutInitWindowPosition( 100, 100 );
    glutInitWindowSize( 800, 600 );
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH );

    glutCreateWindow( "CSE-170 Computer Graphics" );

    (...)
```

Callback functions

- We also need to set a display function (at a minimum), and then enter the main loop

```
int main( int argc, char** argv )
{
    glutInit( &argc, argv );
    glutInitWindowPosition( 100, 100 );
    glutInitWindowSize( 800, 600 );
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH );
    glutCreateWindow( "CSE-170 Computer Graphics" );

    glutDisplayFunc( display );

    glutMainLoop();

    return EXIT_SUCCESS;
}
```

Callback functions

- You can set other callback functions for a variety of events, for ex:

```
glutDisplayFunc( display );
glutIdleFunc( display );
glutReshapeFunc( reshape );
glutKeyboardFunc( key_pressed );
glutKeyboardUpFunc( key_released );
glutSpecialFunc( key_special_pressed );
glutSpecialUpFunc( key_special_released );
glutMouseFunc( mouse_func );
glutMotionFunc( active_motion_func );
glutPassiveMotionFunc( passive_motion_func );
(...)
```

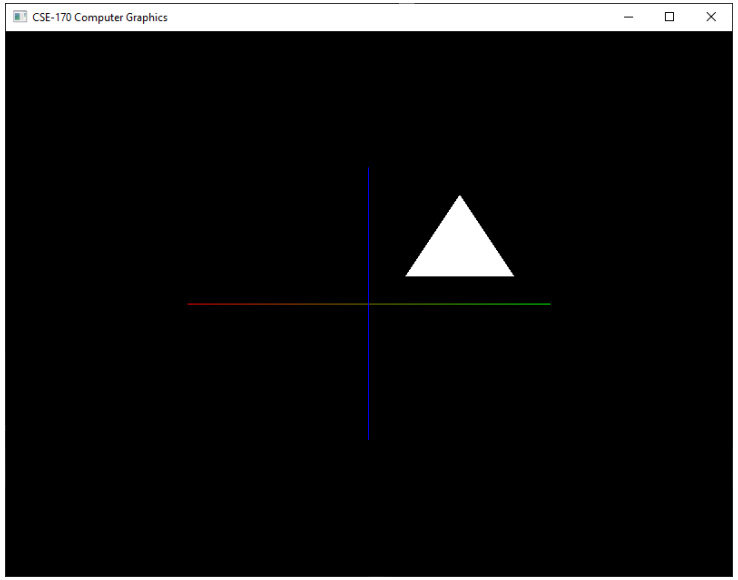
Display function

- The typical steps:
  - clear the buffer(s)
  - render the new frame
  - flush it to the screen (swap buffers)

```
void display()
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    //rendering here
    (...)

    glutSwapBuffers();
}
```

Drawing (immediate mode)

```
//rendering here
glBegin( GL_LINES );
        glColor3f( 1.0f, 0.0f, 0.0f );
        glVertex2f( -0.5f, 0.0f );
        glColor3f( 0.0f, 1.0f, 0.0f );
        glVertex2f( 0.5f, 0.0f );
glEnd();

glColor3f( 0.0f, 0.0f, 1.0f );
glBegin( GL_LINES );
        glVertex2f( 0.0f, -0.5f );
        glVertex2f( 0.0f, 0.5f );
glEnd();

glColor3f( 1.0f, 1.0f, 1.0f );
glBegin( GL_TRIANGLES );
        glVertex2f( 0.1f, 0.1f );
        glVertex2f( 0.4f, 0.1f );
        glVertex2f( 0.25f, 0.4f );
glEnd();
```



Drawing (modern way)

1) Write the shaders
2) Initialize the shaders
   - Send shader programs to GPU (& compile)
3) Organize geometry data in buffers and send to the GPU
   - Create needed buffer objects (VAO, VBO, etc.)
4) Draw
   - Enable the shader program and buffers
     - Everything already in the graphics card
   - Send a draw command

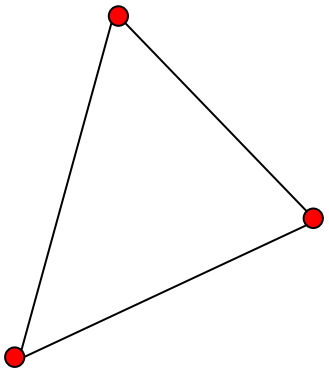
OpenGL drawing 1/4

1) Write shaders, for example:

vertex shader

Input: info per vertex,
for ex: position & color

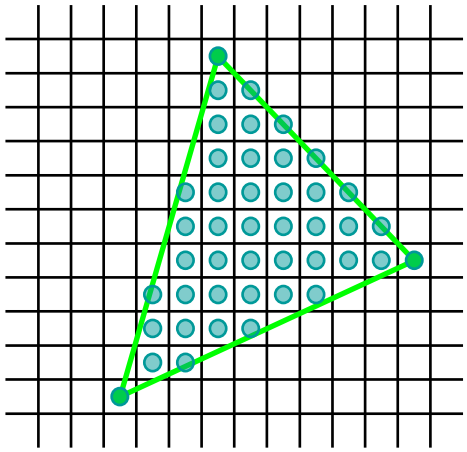
Output: transformed information per interior point of the triangle being rasterized, to be sent to frag shader



fragment shader

Input: info per pixel,
for ex: color

Output: transformed information (for ex: per-fragment lighting)



OpenGL drawing 1/4

1) Write shaders, for example:

vertex shader

```
#version 400
layout(location=0) in vec4 in_Position;
layout(location=1) in vec4 in_Color;
out vec4 vert_Color;

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

void main(void)
{
        gl_Position = projectionMatrix * viewMatrix * modelMatrix *
in_Position;
        vert_Color = in_Color;
}
```

fragment shader

```
#version 400
in vec4 vert_Color;
out vec4 frag_Color;

void main(void)
{
    frag_Color = vert_Color;
}
```

OpenGL drawing 2/4

2) Create and initialize shaders:

```
GLuint vertex_shader_id = glCreateShader( GL_VERTEX_SHADER );
const char* vsrc = ... // load shader from text file
glShaderSource( vertex_shader_id, 1, &src, NULL );
glCompileShader( vertex_shader_id );

GLuint fragment_shader_id = glCreateShader( GL_FRAGMENT_SHADER );
const char* fsrc = ... // load shader from text file
glShaderSource( fragment_shader_id, 1, &src, NULL );
glCompileShader( fragment_shader_id );

shader_program_id = glCreateProgram();
glAttachShader( shader_program_id, vertex_shader_id );
glAttachShader( shader_program_id, fragment_shader_id );
glLinkProgram( shader_program_id );
```

OpenGL drawing 3/4

3) Organize geometry data in buffers:

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

glEnableVertexAttribArray( 0 ); // first buffer: coordinates
glBindBuffer( GL_ARRAY_BUFFER, pos_buffer_id );
glBufferData( GL_ARRAY_BUFFER, sizeof_v, &v_data[0], GL_STATIC_DRAW );
glVertexAttribPointer( 0, 2, GL_FLOAT, GL_FALSE, 0, 0 );

glEnableVertexAttribArray ( 1 ); // second buffer: colors
glBindBuffer( GL_ARRAY_BUFFER, color_buffer_id );
glBufferData( GL_ARRAY_BUFFER, sizeof_c, &c_data[0], GL_STATIC_DRAW );
glVertexAttribPointer( 1, 4, GL_UNSIGNED_BYTE, GL_FALSE, 0, 0 );
```

OpenGL drawing 4/4

4) Send a draw call:

```
glUseProgram( shader_program_id ); // set shader program to use

//Sending uniforms to the GPU
//Note: this only has to be done again if the matrices change
glUniformMatrix4fv( proj_loc, 1, GL_FALSE, &proj_mat[0][0] );
glUniformMatrix4fv( view_loc, 1, GL_FALSE, &view_mat[0][0] );
glUniformMatrix4fv( model_loc, 1, GL_FALSE, &model_mat[0][0] );


glBindVertexArray( vao );
glDrawArrays( GL_TRIANGLES, 0, num_verts );
```

PA1

- In your first PA, you will use OpenGL's immediate mode to do some basic drawing, and freeglut to implement interactivity using callback functions.

- Instructions will be posted next Monday.

- We will use the modern shader-based approach from PA2 onwards.

More on OpenGL

- More sources of information on OpenGL:
  - Chapter 17 in our textbook (5th edition), "Using Graphics Hardware"
  - The OpenGL core specification and API reference card pdfs (in-depth)
  - OpenGL reference pages (https://registry.khronos.org/OpenGL-Refpages/gl4/)
  - The OpenGL wiki (https://www.khronos.org/opengl/wiki)