CS 428 – Lecture 2

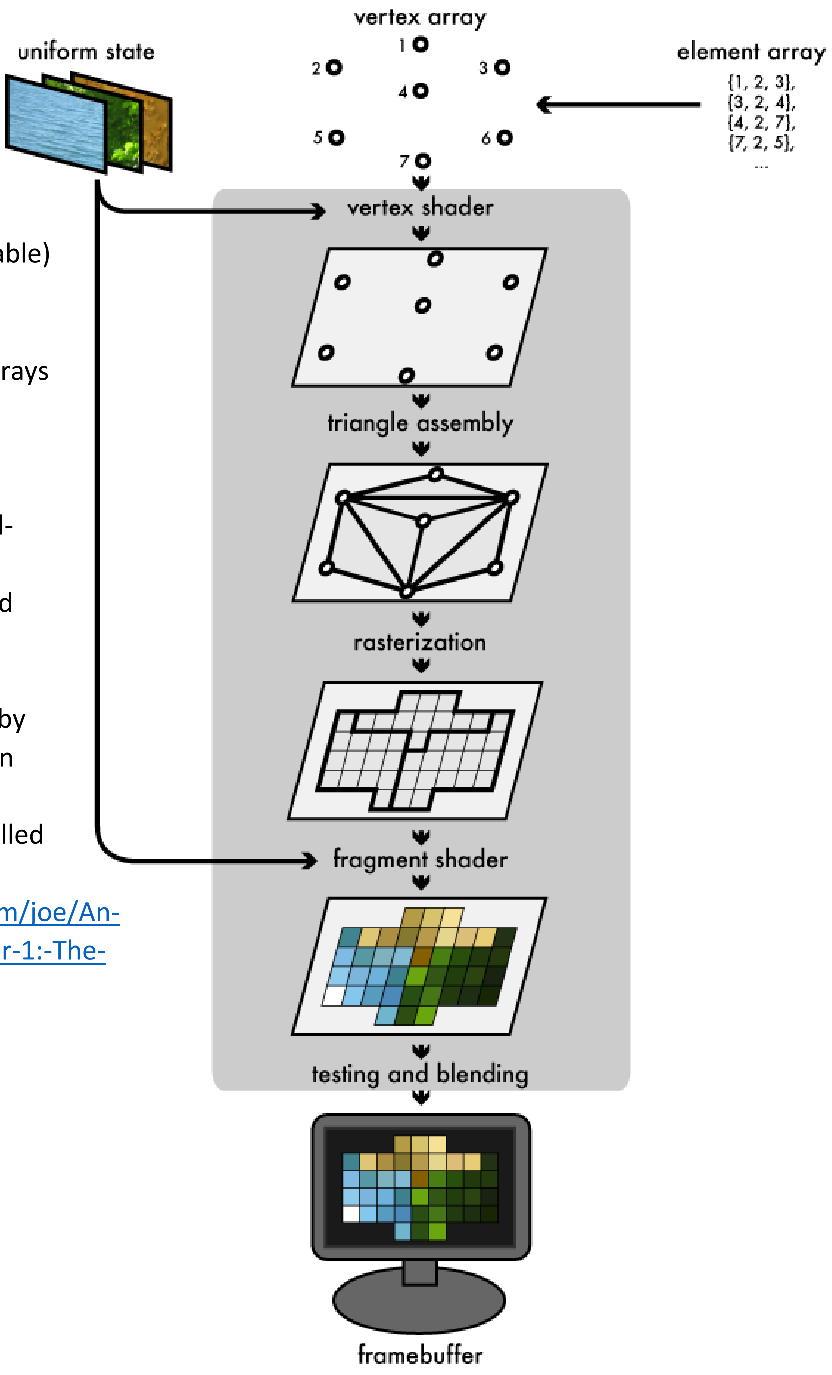
1/30/19

General Course Logistics

- Check out course website for new updates on rendering engines, code samples, and term project guidelines, required software libraries, etc
- To request edits and/or changes to website, initiate a pull request on github

OpenGL Programming

- Recommended development tools:
    - C++
    - Ubuntu 18.04 (or 16.04)
    - OpenGL 3.3
    - If all else fails, try using iLab machines or a VM (could be unstable)
- General OpenGL graphics pipeline
    - The host program fills OpenGL-managed memory buffers with arrays of vertices
    - These vertices are projected into screen space, assembled into triangles, and rasterized into pixel-sized fragments
    - Finally, the fragments are assigned color values and drawn to the framebuffer.
    - Modern GPUs get their flexibility by delegating the "project into screen space" and "assign color values" stages to uploadable programs called shaders.
    - Source: http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html

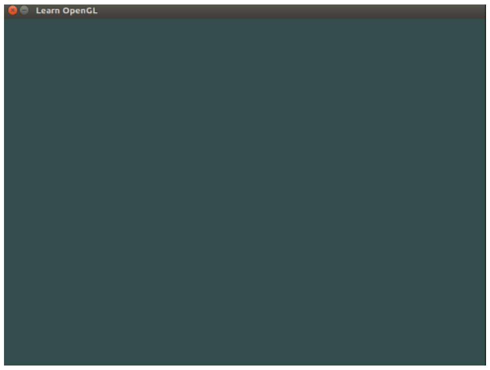


Code Example Walkthroughs

Follow along with the code sample posted at:
https://orionquest.github.io/CS428/lectures/lecture1.html

Window

The objective is to make an empty window of size 800x600 that we can close by pressing the ESC key that looks as follows



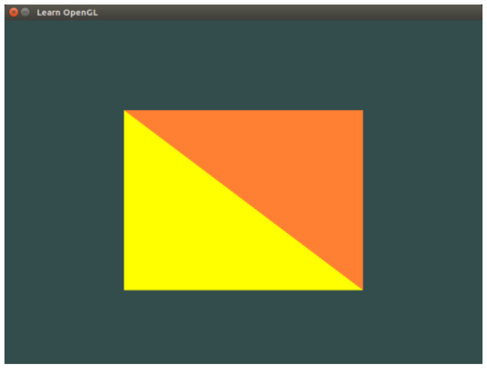
- Window management library used: glfw3
  - WindowHint:
    - Config version, profile, resizability, etc
    - Generally used in boilerplate code
  - CreateWindow(width, height, title, monitor, share)
    - Monitor = null → windowed, else set to monitor pointer to do full-screen
    - Share = null → do not share resources, else set to window pointer to share resources with another window.
  - SetKeyCallback(window, key_callback)
    - Key_callback is a void helper function that reads the key being activated as well as what action (press, release, hold, etc) and compares to two constants GLFW_KEY_ESCAPE and GLFW_PRESS
    - If true, calls glfwSetWindowShouldClose(window, GL_TRUE) where GL_TRUE is the OpenGL Boolean constant for true
  - glfwSetWindowShouldClose(window, true/false)
    - sets a Boolean flag property for window that indicates that the window should be closed when possible
- Remember to do null pointer checks and sanity checks! The more time you spend making various graceful exit and break points for your program, the less time you spend puzzling over what went wrong when it comes to debugging
- Additional window management library used: glew
  - SetFrameBuffer(window, &width, &height)
    - Here we use pointers to variables instead to allow for window to have dynamic size reference
    - This allows us to do things like programmatically set size of our frame buffer as well as programmatically get the size readings

- while(!glfwWindowShouldClose(window)) loop used to poll (continuously check until loop condition is false) for key press
  - Loop traps execution until window is told to close; good way to keep the operational part of your code running continuously
  - ClearColor sets color of "clear" pixels (i.e. background) to given color code
  - SwapBuffers(window) helps clear visual glitches in dynamic animations
    - Mechanics of how SwapBuffers works internally is not important at this time and will be reviewed at a later point

The next two codes sample we will analyze are posted at:
https://orionquest.github.io/CS428/lectures/lecture2.html

Shader

The first objective is to make a window of size 800x600 that we can close by pressing the ESC key that contains two triangles shaded two different colors that looks as follows:

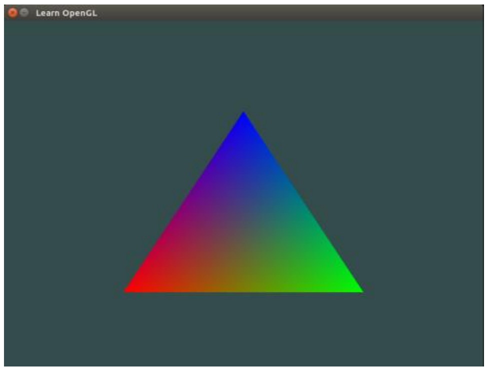


- General boilerplate code/window creation code is reused from previous example
- Vertices array
  - Type GLfloat – floating point number
  - Arranged in rows of 3, xyz coordinates of each vertex of the four points of the two triangles
  - Note that all z coordinates are set to 0; OpenGL by default renders a 3d environment, so we must specify that we want everything to be flat by giving our points a constant height of 0 instead
- Index arrays
  - Type GLuint – unsigned int
  - Specify which indices of the vertices array to use to define each triangle
- GenBuffers initializes the vertex buffer object (VBO) and element buffer objects (EBO)
- Create vertex and fragment shader objects
  - Call ShaderSource to explicitly set shader's source code by reference to a constant defined earlier instead of direct value for easier editing and/or abstraction to header files
  - Must be explicitly compiled by calling CompileShader due to shader's functioning in the GLShaderLanguage
  - Shaders are created per shader property to be created

- Shader programs
    - Created using CreateProgram()
    - Attach shaders to program by using AttachShader
    - Link the program by using LinkProgram
        - Note that once you have attached the shaders to your program and linked it, you can save time and space by freeing up the memory resources used to store the shaders
- Create vertex array objects (VBO) for each triangle to pass info to GPU
- General pipeline for object initialization:
    - Bind vertex array object using BindVertexArray → tells which vertex array to use
    - Copy vertices/indices into VBO and EBO respectively using BindBuffer and BufferData
    - Set vertex attribute pointers using VertexAttribPointer, EnableVertexAttribArray, and BindVertexArray
        - AttribPointer(index, size, type, normalized, stride, pointer)
            - Index – where to start within VBO
            - Size – number of vertices specified for this element
            - Type – type of each vertex value
            - Normalized – Boolean flag (GL_FALSE/GL_TRUE)
            - Stride – size in bits to go to next vertex in memory
    - Unbind VBO after done with use
- UseProgram, BindVertexArray, and DrawElements used to draw the actual pixels of the triangle on the screen continuously until the window closes

Triangle

The second objective is to make a window of size 800x600 that we can close by pressing the ESC key that contains one triangle that has red, green, and blue solid colors at the three corners, and a gradient that blends the colors in between that looks as follows:



- Note that we begin to see modularized code instead of boilerplate code that is shifted to the Shader.h header file
- class Shader
    - store GLSL code in separate file
    - read said file through input stream
    - try-catch block to modularize error handling
    - cast input stream into GLchar* (OpenGL string)

    - run general shader program creation pipeline from earlier code examples
    - also has a helper function void Use() which calls UseProgram(this->program)
        - this is a self-reference C++ keyword
- This custom Shader class allows us to simply call a constructor with two arguments which are files that determine the GLSL code (not shown here) and we have our shader programs successfully created, attached, linked, and ready to use simply by calling Shader.Use()
- Notice that this time VAO is arranged in rows of 6, with 3 xyz coordinates for the position, and 3 RGB values for the color the vertex starts out with
    - Adjust stride according to VAO layout
    - Set pointer to correct place in memory inside the VAO
- For each entry in the array
    - Set position attribute pointers
    - Set color attribute pointers