Durham University

Department of Computer Science

SM Computer Graphics Practical 1

Aim: WebGL is a Javascript API allowing you to develop browser-based and GPU accelerated graphics applications. This practical helps you understand the structure of WebGL and start programming with it. Because you are still new to WebGL, this practical serves as a preparation lesson, allowing you to get familiar with the way of constructing geometry data and passing the data to vertex shader and fragment shader for parallel processing and generating rendering results.

In this practical, since this is the first time for you to work with WebGL, you will expect to spend some time studying the structure and the statements of WebGL programs rather than developing a lot of WebGL codes.

Graphics Processing Unit (GPU) technology and programming:

- It comprises two main programmable parts, namely **vertex shader** and **fragment shader**. They accept and process geometry data in parallel.
- A WebGL program is divided into three parts, namely **main program**, **vertex shader program** and **fragment shader program**.
- In a simple graphics program, the main program comprises instructions to declare and maintain geometry data for 3D modeling and virtual environment construction. This part is processed by CPU.
- Vertex shader program comprises instructions to perform geometry data computation in parallel. There is a rasterization process in the GPU to accept output from vertex shader, converting them into **fragments**, which form the input of the fragment shader.
- The fragment shader program comprises instructions to process fragments in parallel.

All the above concepts will be properly taught in the coming lectures. Note that you are assumed to have a basic knowledge in HTML5, CSS3 and Javascript. Textbook sample codes are provided to let you quickly get familiar with the structure of a WebGL program, without spending time to duplicate codes that are common to all WebGL programs.

Task 1: Check your browser.

- WebGL may run on any platform, including smartphones. However, it is a good practice to ensure your browser is compatible with WebGL before you start to work. The compatibility test can be done by https://get.webgl.org/

Task 2: Check WebGL context.

- Use p1-GL_Context.html to check whether your browser can run HTML5 canvas (create a drawing area) and create a WebGL context based on the canvas (for displaying WebGL program outputs).
- Study the codes, making sure that you understand their meaning. You will expect every WebGL program will incorporate a variant of such context checking instructions.

Task 3: Study the structure of a WebGL program:

- Use p1-ColoredPoints.html and p1-ColoredPoints.js. The .html script creates a HTML5 canvas and the .js is the WebGL program, which comprises the main, vertex shader and fragment shader programs. All scripts under the ../lib directory consist of proper routines that are required for many WebGL programs.
- Study the main() function of p1-ColoredPoints.js:
    - getWebGLContext(canvas): test WebGL context

- initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE): initialize vertex and fragment shaders
- gl.getAttribLocation(gl.program, '**a_Position**'): a_Position is a variable that will use in the vertex shader in this example. This instruction is to allocate a proper memory space that allow the main program to pass data from the main program to the vertex shader through “a_Position”.
- gl.getUniformLocation(gl.program, '**u_FragColor**'): is similar to the above. Instead, it prepares the memory space for a fragment shader’s variable “u_FragColor”.
- This javascript captures the current mouse clicked position and define a position-dependent color. Such information will pass to vertex and fragment shader for processing and display.
- gl.vertexAttrib3f(a_Position, xy[0], xy[1], 0.0): pass a mouse clicked position to vertex shader through a_Position.
- gl.uniform4f(u_FragColor, rgba[0], rgba[1], rgba[2], rgba[3]): pass a color value to fragment shader through u_FragColor.
- gl.drawArrays (gl.POINTS, 0, 1): Active shaders to render the result by specifying the input is a list of points (gl.POINTS) where WebGL will read the vertex list from its first element (i.e. 0) and there is only one element (i.e. 1).
- The for loop in the main program will call the shaders “len” times to draw a point on the canvas each time.

Task 4: Understand parallel programming (continue to study p1-ColoredPoints.js):

- var VSHADER_SOURCE: define the vertex shader program, which is an independent program itself. In terms of programming, the instructions that you have written down in a vertex shader program will only be applied to process a single vertex. Conceptually, vertex shader will independently apply all instructions in the vertex shader program to each input vertex simultaneously. In practice, such a concurrency will be limited by the number of cores of a GPU. All output will be sent to the rasterization process in GPU (non-programmable) for generating fragments. In the example, the vertex shader program only accepts a vertex and sets its point size without applying any computation on it.
- var FSHADER_SOURCE: define the fragment shader program. It possess a similar characteristics as the vertex shader program. Instead, it will accept fragments from the rasterization process as its input. (Of course, the main program can still directly provide some inputs to the fragment shader program.) The instructions here will independently process each input fragment simultaneously. The fragment shader will write its outputs to the framebuffer. In the example, the fragment shader program only accepts color information from the main program and apply it to each fragment for rendering.

Take a break before proceed, since you should be filled with a lot of new terminologies at the moment.

Task 5: Draw a triangle (HelloTriangle.js):

- The idea of drawing a 2D triangle is that in the main program:
    1. Create an Float32Array to maintain all coordinates of the triangle (contains 6 elements, i.e. the three pairs of (x,y) coordinates)
    2. Use gl.bindBuffer() and gl.bufferData() to write all coordinates into the WebGL buffer object.
    3. Assign the buffer object to a_Position and enable it by gl.vertexAttribPointer() and gl.enableVertexAttribArray().
    4. Finally, call a single instruction gl.drawArrays(gl.TRIANGLES, 0, n) to utilize all three pairs of (x,y) coordinates in one go to draw a triangle with shaders, where n = 3.
    5. Note that the coordinate system of the canvas is that the original (0,0) is at the center of the canvas, where (-1.0, 1.0) at the top left-hand corner and (1.0, -1.0) at the bottom right-hand corner.
- Study HelloTriangle.js and answer the following questions:
    1. What has been done in the vertex shader?
    2. What has been done in the fragment shader?
    3. What is a_Position in the main program?
    4. How can you extend HelloTriangle.js to add another triangle for rendering (display)?
    5. Now, how do you expect to change if you want to render many triangles in the program?
- Change gl.TRIANGLES in gl.drawArrays() to gl.POINTS. In addition, change the point size in the vertex shader (ref: p1-ColoredPoints.js) to 10. See what you will get.

Task 6: Review questions:

- Can you identify the main, vertex shader, and fragment shader programs in p1-ColoredPoints.js or HelloTriangle.js?
- Can you identify which of the above three programs are run by the CPU or the GPU?
- Which WebGL instruction invokes vertex and fragment shader programs?
- What are the purposes of gl.bindBuffer() and gl.bufferData()?
- What are the purposes of gl.TRIANGLES and gl.POINTS in gl.drawArrays()?

Don’t worry if you cannot follow quite well. It takes time to get everything in place.