Week 01 - WebGL Introduction

Report

I. Student information

- Full Name: Ngo Nguyen Thanh Thanh
- Student ID: 21127690

II. Exercises:

1. The goal of the "createShader" function? Why do we have to pass code as string to create shaders?

- **Goal of the** createShader **function:**

The purpose of the createShader function is to:

1. **Create a shader object** of a specific type (either a **vertex shader** or a **fragment shader**) using the gl.createShader() function.
2. **Provide the source code** for the shader (written in GLSL - OpenGL Shading Language) using gl.shaderSource().
3. **Compile the shader** code into machine-readable code for the GPU using gl.compileShader().
4. **Check for compilation errors** by using gl.getShaderParameter() to check if the shader compiled successfully (via gl.COMPILE_STATUS).
5. If successful, the shader is returned for further use in the WebGL program. If it fails to compile, the function logs the error and deletes the shader object using gl.deleteShader().

- **Why do we have to pass code as a string to create shaders?**

Shaders in WebGL (and OpenGL) are written in GLSL (OpenGL Shading Language), a specialized programming language for defining visual effects like lighting, coloring, and texture mapping. Since shaders are compiled at runtime on the GPU, their source code must be passed **as a string** to the WebGL API.

- **String representation**: The shader source code is essentially plain text that defines the instructions for the GPU to execute. By passing it as a string, WebGL can handle and compile it correctly.
- **Flexibility**: GLSL code needs to be dynamically compiled by the WebGL context (the GPU), so it's not hardcoded in binary or precompiled

format; instead, it's passed as a string for flexibility and runtime customization.
In short, the createShader function needs the source code as a string because the WebGL API compiles the shader at runtime, and GLSL code is written as text, which is passed as a string to be processed by the GPU.

2. Explain the meaning and parameters (including value type and data range) of the following functions

a. gl.bufferData

The gl.bufferData function is used to provide data to a WebGL buffer (such as an array buffer or element array buffer). It initializes a buffer's data store with a specified array of values.

Parameters:

1. **target** (gl.ARRAY_BUFFER or gl.ELEMENT_ARRAY_BUFFER):
   - Specifies the type of buffer being used (either for vertex data or element data).
2. **data** (Float32Array, Uint16Array, Float64Array, etc.):
   - The actual data to store in the buffer. In your example, it's a Float32Array that holds the vertex coordinates.
   - It can be any typed array that holds numeric data.
3. **usage** (gl.STATIC_DRAW, gl.DYNAMIC_DRAW, gl.STREAM_DRAW):
   - Specifies how the data will be used:
     - gl.STATIC_DRAW: The data will rarely change (best for things like vertex data).
     - gl.DYNAMIC_DRAW: The data will change frequently.
     - gl.STREAM_DRAW: The data will change every time it’s drawn.

Example in the code:

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

- This initializes a buffer with the data in vertices (a Float32Array), and it specifies that the buffer data is static, meaning it won’t change often.

b. gl.vertexAttribPointer

The gl.vertexAttribPointer function specifies how WebGL should interpret the vertex buffer data (how to extract attributes from the buffer).

Parameters:

1. **index** (integer):
    - The index of the attribute in the shader. This is obtained using gl.getAttribLocation(program, "attributeName").
    - It tells WebGL which attribute variable in the shader program to associate with the buffer data (e.g., "coordinates").
2. **size** (integer):
    - The number of components per vertex attribute. For example, for a 3D vertex, this would be 3 (x, y, z).
    - It can range from 1 to 4, depending on the number of components in the attribute.
3. **type** (gl.FLOAT, gl.UNSIGNED_BYTE, etc.):
    - The data type of each component of the attribute.
    - In your case, it's gl.FLOAT, indicating each component is a floating-point number.
4. **normalized** (boolean):
    - Whether or not the data should be normalized when used (applies to integer types like gl.UNSIGNED_BYTE).
    - In your case, it's false, meaning it doesn’t normalize the values (i.e., it uses the raw floating-point values).
5. **stride** (integer):
    - The space, in bytes, between consecutive vertex attributes. If the data is tightly packed, this value is usually 0.
    - For the case of a single vertex attribute (e.g., just positions), it’s 0 (i.e., no extra space).

6. **offset** (integer):
    - The offset, in bytes, from the beginning of the buffer to the start of the vertex attribute.
    - Typically used when vertex data contains multiple attributes (e.g., positions, colors), and we need to specify where each attribute starts in the buffer.

Example in the code:

gl.vertexAttribPointer(coord, 3, gl.FLOAT, false, 0, 0);

- Here, it tells WebGL to use the attribute "coordinates", which has 3 components (x, y, z), each of type FLOAT. The data is not normalized (false), with no extra spacing between attributes (stride is 0), and the data starts from the beginning of the buffer (offset is 0).

c. gl.viewport

The gl.viewport function defines the area of the canvas that WebGL should use to draw the image.

Parameters:

1. **x** (integer):
    - The x-coordinate of the lower-left corner of the viewport. It’s measured in pixels from the left edge of the canvas.
2. **y** (integer):
    - The y-coordinate of the lower-left corner of the viewport. It’s measured in pixels from the bottom edge of the canvas.
3. **width** (integer):
    - The width of the viewport, in pixels.
4. **height** (integer):
    - The height of the viewport, in pixels.

Example in the code:

gl.viewport(0, 0, canvas.width, canvas.height);

- This sets the viewport to cover the entire canvas (from the bottom-left corner (0, 0) to the top-right corner (canvas.width, canvas.height)), so WebGL will draw the entire scene to fill the canvas.

Summary:

- **gl.bufferData**: Sets the data in a WebGL buffer (e.g., vertex positions).
- **gl.vertexAttribPointer**: Describes how to interpret the data in the buffer for use in shaders.
- **gl.viewport**: Specifies the area of the canvas where WebGL should render its output.

3. In What form are colors in WebGL transmitted/used ? What is the range of values of color parameters/color channels?

Color Format in WebGL:

In WebGL, colors are typically transmitted in the RGBA format, where:

- **R** represents the red channel,
- **G** represents the green channel,
- **B** represents the blue channel, and
- **A** represents the alpha (transparency) channel.

These channels are used to represent colors and their opacity in a format that can be processed by shaders in WebGL.

Range of Color Values:

- Each channel (R, G, B, A) holds a value within the range of 0.0 to 1.0.
    - **0.0** means the minimum intensity (black for R, G, and B, fully transparent for A).
    - **1.0** means the maximum intensity (full brightness for R, G, and B, fully opaque for A).

For example:

- **Fully red:** rgba(1.0, 0.0, 0.0, 1.0) means maximum red, no green, no blue, and fully opaque.
- **Transparent blue**: rgba(0.0, 0.0, 1.0, 0.0) means no red, no green, full blue, and fully transparent.

Additional Details:

- **Normalization:** For integer-based color values (e.g., when using gl.UNSIGNED_BYTE for color channels), the values are normalized to the range [0, 255] and are converted to floating-point values in the range [0.0, 1.0] when processed.
    - **Example:** A color represented as (255, 0, 0) in gl.UNSIGNED_BYTE for red, green, and blue channels will be normalized to (1.0, 0.0, 0.0) in WebGL.

In the example code:

the gl_FragColor is assigned a color using RGBA values:

gl_FragColor = vec4(0.0, 0.0, 0.0, 0.1);

- R = 0.0 (No red)
- G = 0.0 (No green)
- B = 0.0 (No blue)
- A = 0.1 (Semi-transparent)

Thus, this color would represent a black color with 10% opacity (90% transparency).

4. In the function clearGL, what will happen when the line gl.enable(gl.DEPTH_TEST) is uncommented?

When the line gl.enable(gl.DEPTH_TEST); is **uncommented**, WebGL will enable **depth testing**. This means WebGL will begin checking and comparing the depth (z-value) of points when drawing them on the canvas to determine whether a point should be drawn in front of or behind other points, depending on its position in 3D space.

Specifically, when this line is uncommented:

- **Depth testing** is enabled. Each point (or object) being drawn will have a corresponding depth value (z-value), which indicates its position along the z-axis (depth direction from the viewer).
- **When drawing objects (or points) on the canvas**, WebGL will compare the depth of the current object with objects that have been drawn before. If the current object has a smaller depth (i.e., closer to the viewer) than previously drawn objects, it will be rendered on top and replace them. Conversely, if the current object is farther away (larger depth), it won't be drawn, or it will be hidden.
- This is particularly useful when rendering 3D objects or points with varying depths. Without depth testing, all points or objects would be drawn in the order they are called, and no depth comparison would be made, potentially causing objects to incorrectly overlap or be hidden.

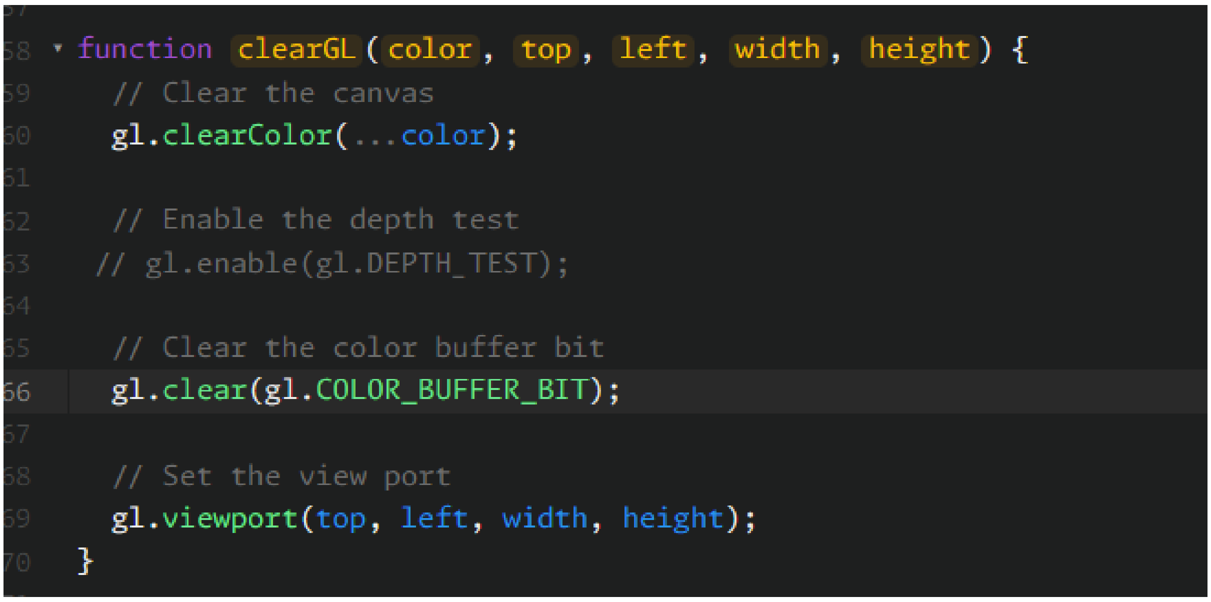
Example of effect:

Imagine you have a 3D scene with objects (e.g., a box or a sphere) where some points are in front of others and some are behind:

- With **depth testing enabled**, WebGL ensures that points closer to the viewer will be drawn in front of those that are farther away, producing a correct depth perception.
- Without depth testing, points are drawn in the order they are called, without considering depth, which could result in objects incorrectly being hidden or not showing up as expected.

Conclusion:

When you **uncomment** the line gl.enable(gl.DEPTH_TEST);, you enable depth testing in WebGL, which ensures that 3D objects are drawn correctly according to their depth, preventing incorrect hiding or overlapping of objects.

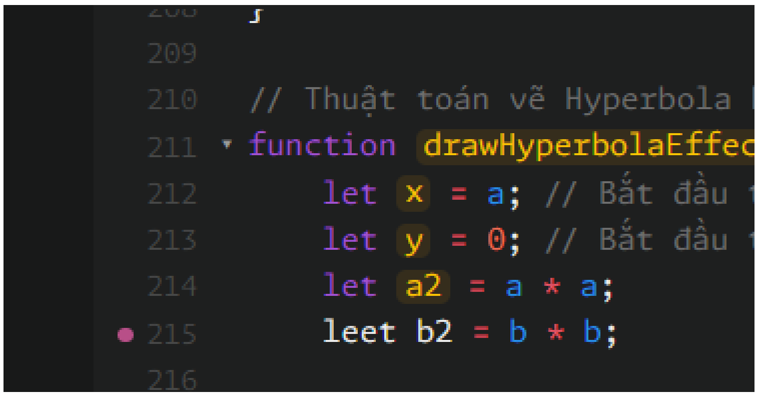


```
function clearGL(color, top, left, width, height) {
  // Clear the canvas
  gl.clearColor(...color);

  // Enable the depth test
 // gl.enable(gl.DEPTH_TEST);

  // Clear the color buffer bit
  gl.clear(gl.COLOR_BUFFER_BIT);

  // Set the view port
  gl.viewport(top, left, width, height);
}
```

5. When using Jsfiddle, if the program has an error (the code cannot run), how can we find out which line has the error?

Method 1: notes: this one only display the first error.

Syntax error: Red point indicates where the red point is.



```
209
210  // Thuật toán vẽ Hyperbola
211  function drawHyperbolaEffec
212        let x = a; // Bắt đầu
213        let y = 0; // Bắt đầu
214        let a2 = a * a;
● 215        leet b2 = b * b;
216
```

Method 2:

Example:

- "<a class='gotoLine' href='#399:1'>399:1</a> Uncaught ReferenceError: ddrawGrid is not defined"

Read the content of error: “ddrawGrid is not defined”. Then go find the line (this error may display in many lines) which has “ddrawGrid” to fix error by using the “control + F” and enter ddrawGrid.