Supplementary Handout 1: Drawing Elements

Learning Outcomes

- Draw an object from the element array

Contents

`drawArrays` vs `drawElements`

Recall that we've primarily utilized `drawArrays` for our graphics rendering. This method, however, doesn't account for indices. Typically, `drawArrays` comes into play for exceedingly straightforward geometries where specifying indices might be excessive, like when rendering a triangle or rectangle. Should your geometry creation require repeating numerous vertices, drawArrays might not be the optimal choice. The reason being, duplicating more vertex data leads to increased vertex shader calls.

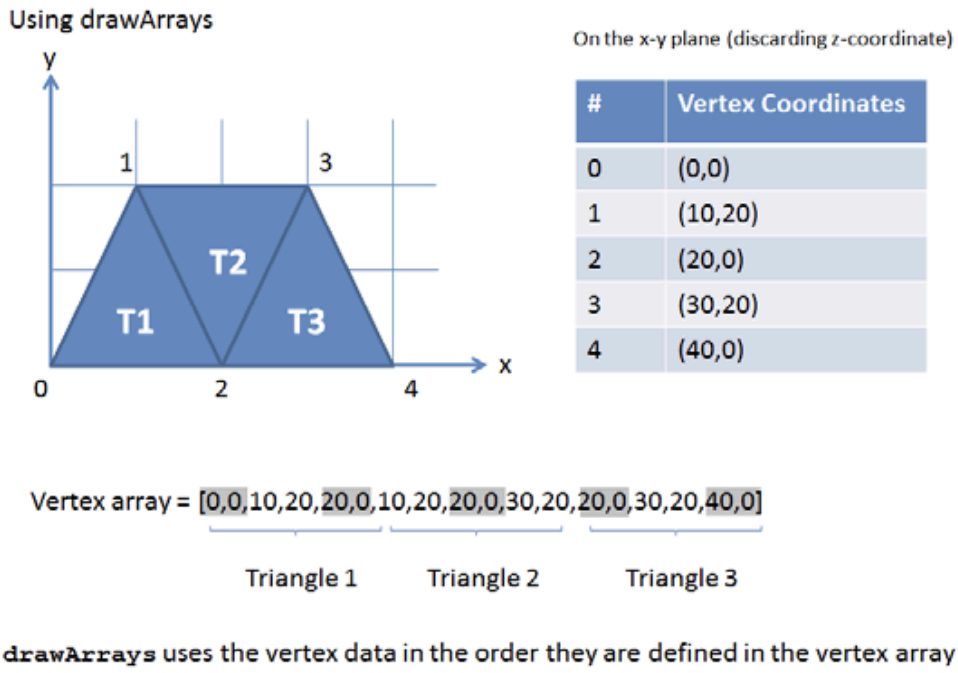

Fig. 1: Using drawArrays

In contrast to cases where no Index Buffer Object (IBO) is defined, `drawElements` enables the use of the IBO to instruct WebGL on how to render the geometry. It's important to note that `drawArrays` utilize Vertex Buffer Objects (VBOs), which causes the vertex shader to process repeated vertices multiple times. However, `drawElements` uses indices that allow vertices to be processed only once and utilized as many times as necessary as defined in the IBO. This functionality significantly reduces the memory and processing burden on the GPU.

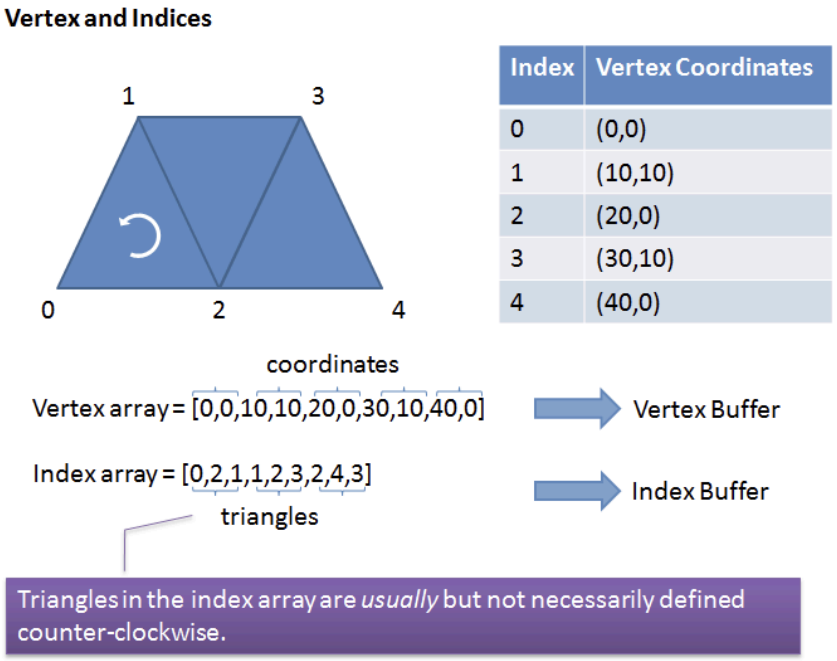

Fig. 2: Using drawElements

Drawing Elements

Rather than directly accessing the stored vertices of the geometry that need to be rendered, one can declare a separate array that contains the indices of the vertices to be drawn. This method allows for the reuse of the same point without having to declare it again.

```
const vertices = [
    0.5, 0.5, 1.0, 1.0,     // vertex 0
    -0.5, 0.5, 1.0, 1.0,    // vertex 1
    -0.5, -0.5, 1.0, 1.0,   // vertex 2
    0.5, -0.5, 1.0, 1.0,    // vertex 3
    0.5, -0.5, -1.0, 1.0,   // vertex 4
    0.5, 0.5, -1.0, 1.0,    // vertex 5
    -0.5, 0.5, -1.0, 1.0,   // vertex 6
    -0.5, -0.5, -1.0, 1.0   // vertex 7
];
```

```
const indices = [
// indices of vertices forming triangles
    0, 1, 2, 0, 2, 3,       // front
    0, 3, 4, 0, 4, 5,       // right
    0, 5, 6, 0, 6, 1,       // up
    1, 6, 7, 1, 7, 2,       // left
    7, 4, 3, 7, 3, 2,       // down
    4, 7, 6, 4, 6, 5        // back
];
```

The indices element array serves as a look-up table. It is an element array because each value represents an actual element (4-tuple point). An example of this is 0 means the 4-tuple (0.5, 0.5, 1.0, 1.0). This saves space since the 4-tuple for each vertex does not need to be redeclared if another surface has it as an endpoint.

Now that the array has been declared, the next step is to declare a buffer bound to the element array.

```
// create another buffer for indices
const indicesBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indicesBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint8Array(indices),
gl.STATIC_DRAW)
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

Inside the user-defined function `drawScene()`, the buffers for the vertices and indices can now rebound and be drawn using `gl.drawElements()`.

Syntax

```
void gl.drawElements(mode, count, type, offset);
```

Parameters

- **mode** - A GLenum specifying the type primitive to render. Possible values are:
  - `gl.POINTS`: Draws a single dot.
  - `gl.LINE_STRIP`: Draws a straight line to the next vertex.
  - `gl.LINE_LOOP`: Draws a straight line to the next vertex and connects the last vertex back to the first.
  - `gl.LINES`: Draws a line between a pair of vertices.
  - `gl.TRIANGLE_STRIP`
  - `gl.TRIANGLE_FAN`

  - `gl.TRIANGLES`: Draws a triangle for a group of three vertices.
- **count** - A GLsizei specifying the number of elements to be rendered.
- **type** - A GLenum specifying the type of the values in the element array buffer. Possible values are:
  - `gl.UNSIGNED_BYTE`
  - `gl.UNSIGNED_SHORT`
  - When using the OES_element_index_uint extension: `gl.UNSIGNED_INT`
- **offset** - A GLintptr specifying a byte offset in the element array buffer. Must be a valid multiple of the size of the given type.

```
gl.bindBuffer(gl.ARRAY_BUFFER, verticesBuffer);
gl.vertexAttribPointer(aPositionPointer, 4, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indicesBuffer);

// draw triangles based on indices array
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_BYTE, 0);
```

The same polygon in **element_array.html** is rendered with varying colors in **element_array_varying.html** with the simple code modification shown below in the vertex shader.

```
in vec4 a_color;//attribute vec4 a_color;
out vec4 v_color;//varying vec4 v_color;
...
void main() {
...
v_color = a_color;
}
```

Vertices and colors per vertex can be declared in the same array referenced by the same indices element array. This can be seen in **element_array_2.html**. As seen in the code below, `gl.ARRAY_BUFFER` is bound to the array containing both vertices and colors. The first function call is for binding to the vertex position and the next call to `gl.vertexAttribPointer()` is for binding to the color property.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexDataBuffer);
gl.vertexAttribPointer(aPositionPointer, 4, gl.FLOAT, false, 8*4, 0);
gl.vertexAttribPointer(colorLocation, 4, gl.FLOAT, false, 8*4, 4*4);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indicesBuffer);
```

```
// draw triangles based on indices array
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_BYTE, 0)
```

References

- Cantor, D., & Jones, B. (2012). WebGL beginner's guide. Packt Publishing Ltd.
- Matsuda, K., & Lea, R. (2013). WebGL programming guide: interactive 3D graphics programming with WebGL. Addison-Wesley.
- Clariño MAAD, 2020. CMSC 161 Laboratory Handout 06 - Drawing Elements and Applying Light