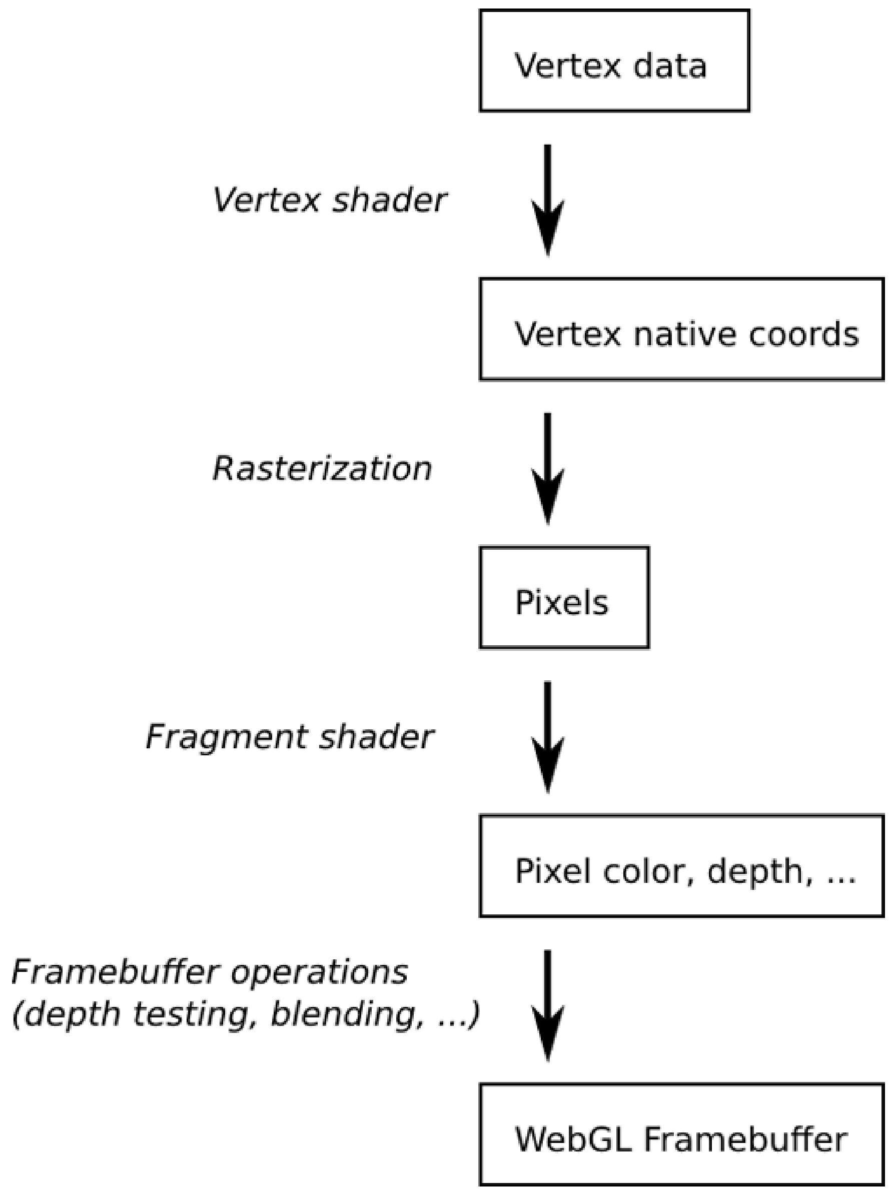
WebGL

- A thinner version of OpenGL based on OpenGL ES
  - OpenGL ES designed for embedded systems, thin version of OpenGL
- Compared to OpenGL, there are more responsibilities put on the programmer
  - Good (more control) and bad (have to write more code to get even something simple to show)

WebGL graphics pipeline



Structure of WebGL program

- Part of the program written in Javascript, part of it in GLSL (shader language)
  - In WebGL you have to write two shaders for even the most basic graphics applications
- Programmable pipeline programs for vertex processing and fragments are mandatory
  - Vertex shader: called once for each vertex in a primitive
  - Fragment shader: called for each pixel in the primitive (after rasterization occurs)

Basic shaders

Vertex Shader

```
attribute vec2 a_coords;
attribute vec3 a_color;
varying vec3 v_color;
void main() {
   gl_Position = vec4(a_coords, 0.0, 1.0);
   v_color = a_color;
}
```

Fragment Shader

```
precision mediump float;
varying vec3 v_color;
void main() {
   gl_FragColor = vec4(v_color, 1.0);
}
```

WebGL graphics context

```
canvas = document.getElementById("webglcanvas");
gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
```

- “webglcanvas” is the HTML Canvas tag id
- Once context is acquired, the shaders need to be compiled and linked

```
var vertexShader = gl.createShader( gl.VERTEX_SHADER );
gl.shaderSource( vertexShader, vertexShaderSource );
gl.compileShader( vertexShader );


var prog = gl.createProgram();
gl.attachShader( prog, vertexShader );
gl.attachShader( prog, fragmentShader );

gl.linkProgram( prog );
```

- Finally

```
gl.useProgram( prog );
```

Function to compile and link shaders

http://math.hws.edu/graphicsbook/c6/s1.html

```
/**
 * Creates a program for use in the WebGL context gl, and returns the
 * identifier for that program.  If an error occurs while compiling or
 * linking the program, an exception of type String is thrown.  The error
 * string contains the compilation or linking error.
 */
function createProgram(gl, vertexShaderSource, fragmentShaderSource) {
   var vsh = gl.createShader( gl.VERTEX_SHADER );
   gl.shaderSource( vsh, vertexShaderSource );
   gl.compileShader( vsh );
   if ( ! gl.getShaderParameter(vsh, gl.COMPILE_STATUS) ) {
      throw "Error in vertex shader:  " + gl.getShaderInfoLog(vsh);
   }
   var fsh = gl.createShader( gl.FRAGMENT_SHADER );
   gl.shaderSource( fsh, fragmentShaderSource );
   gl.compileShader( fsh );
   if ( ! gl.getShaderParameter(fsh, gl.COMPILE_STATUS) ) {
      throw "Error in fragment shader:  " + gl.getShaderInfoLog(fsh);
   }
   var prog = gl.createProgram();
   gl.attachShader( prog, vsh );
   gl.attachShader( prog, fsh );
   gl.linkProgram( prog );
   if ( ! gl.getProgramParameter( prog, gl.LINK_STATUS) ) {
      throw "Link error in program:  " + gl.getProgramInfoLog(prog);
   }
   return prog;
}
```

Data flow through WebGL

- The WebGL graphics pipeline renders an image. The data that defines the image comes from JavaScript. As it passes through the pipeline, it is processed by the current vertex shader and fragment shader as well as by the fixed-function stages of the pipeline. You need to understand how data is placed by JavaScript into the pipeline and how the data is processed as it passes through the pipeline.

Primitives and Attributes

- When WebGL is used to draw a primitive, there are two general categories of data that can be provided for the primitive. The two kinds of data are referred to as attribute variables (or just "attributes") and uniform variables (or just "uniforms")
- A primitive is defined by its type and by a list of vertices
- The difference between attributes and uniforms is that a **uniform variable has a single value that is the same for the entire primitive**, while **the value of an attribute variable can be different for different vertices**

Attributes and Uniforms

- One attribute that is always specified is the coordinates of the vertex.
- The vertex coordinates must be an attribute since each vertex in a primitive will have its own set of coordinates.
- Another possible attribute is color. WebGL allows you to specify a different color for each vertex of a primitive. On the other hand, maybe you want the entire primitive to have the same, "uniform" color; in that case, color can be a uniform variable.
- Other quantities that could be either attributes or uniforms, depending on your needs, include normal vectors and material properties.
- Texture coordinates, if they are used, are almost certain to be an attribute, since it doesn't really make sense for all the vertices in a primitive to have the same texture coordinates.
- If a geometric transform is to be applied to the primitive, it would naturally be represented as a uniform variable.

WebGL attributes

- It is important to understand that WebGL does not come with any predefined attributes, not even one for vertex coordinates.
- In the programmable pipeline, the attributes and uniforms that are used are entirely up to the programmer.
- As far as WebGL is concerned, attributes are just values that are passed into the vertex shader.
- Uniforms can be passed into the vertex shader, the fragment shader, or both.
- WebGL does not assign a meaning to the values. The meaning is entirely determined by what the shaders do with the values.
- The set of attributes and uniforms that are used in drawing a primitive is determined by the source code of the shaders that are in use when the primitive is drawn.

Details...

- When drawing a primitive, the JavaScript program will specify values for any attributes and uniforms in the shader program.
- For each attribute, it will specify an array of values, one for each vertex. For each uniform, it will specify a single value.
- The values will all be sent to the GPU before the primitive is drawn. When drawing the primitive, the GPU calls the vertex shader once for each vertex.
- The attribute values for the vertex that is to be processed are passed as input into the vertex shader.
- Values of uniform variables are also passed to the vertex shader.
- Both attributes and uniforms are represented as global variables in the shader, whose values are set before the shader is called.

Coordinate Systems

- As one of its outputs, the vertex shader must specify the coordinates of the vertex in the clip coordinate system
  - The default coordinate system in WebGL. The projection transform maps the 3D scene to clip coordinates. The rendered image will show the contents of the cube in the clip coordinate system that contains x, y, and z values in the range from -1 to 1; anything outside that range is "clipped" away.)
- It does that by assigning a value to a special variable named gl_Position. The position is often computed by applying a transformation to the attribute that represents the coordinates in the object coordinate system, but exactly how the position is computed is up to the programmer.

Final stages

- After the positions of all the vertices in the primitive have been computed, a fixed-function stage in the pipeline clips away the parts of the primitive whose coordinates are outside the range of valid clip coordinates (−1 to 1 along each coordinate axis).
- The primitive is then rasterized; that is, it is determined which pixels lie inside the primitive. The fragment shader is then called once for each pixel that lies in the primitive.
- The fragment shader has access to uniform variables (but not attributes). It can also use a special variable named gl_FragCoord that contains the clip coordinates of the pixel.
- Pixel coordinates are computed by interpolating the values of gl_Position that were specified by the vertex shader. The interpolation is done by another fixed-function stage that comes between the vertex shader and the fragment shader.

Primitives

- WebGL makes images using 7 basic primitives:
- The seven types of primitive are identified by the constants gl.POINTS, gl.LINES, gl.LINE_STRIP, gl.LINE_LOOP, gl.TRIANGLES, gl.TRIANGLE_STRIP, and gl.TRIANGLE_FAN, where gl is a WebGL graphics context.

Interpolator

- Other quantities besides coordinates can work in much that same way. That is, the vertex shader computes a value for the quantity at each vertex of a primitive.
- An interpolator takes the values at the vertices and computes a value for each pixel in the primitive.
- The value for a given pixel is then input into the fragment shader when the shader is called to process that pixel. For example, color in WebGL follows this pattern: The color of an interior pixel of a primitive is computed by interpolating the color at the vertices. In GLSL, this pattern is implemented using varying variables.

Varying

- A varying variable is declared both in the vertex shader and in the fragment shader.
- The vertex shader is responsible for assigning a value to the varying variable. The interpolator takes the values from the vertex shader and computes a value for each pixel.
- When the fragment shader is executed for a pixel, the value of the varying variable is the interpolated value for that pixel.
- The fragment shader can use the value in its own computations. (In newer versions of GLSL, the term "varying variable" has been replaced by "out variable" in the vertex shader and "in variable" in the fragment shader.)

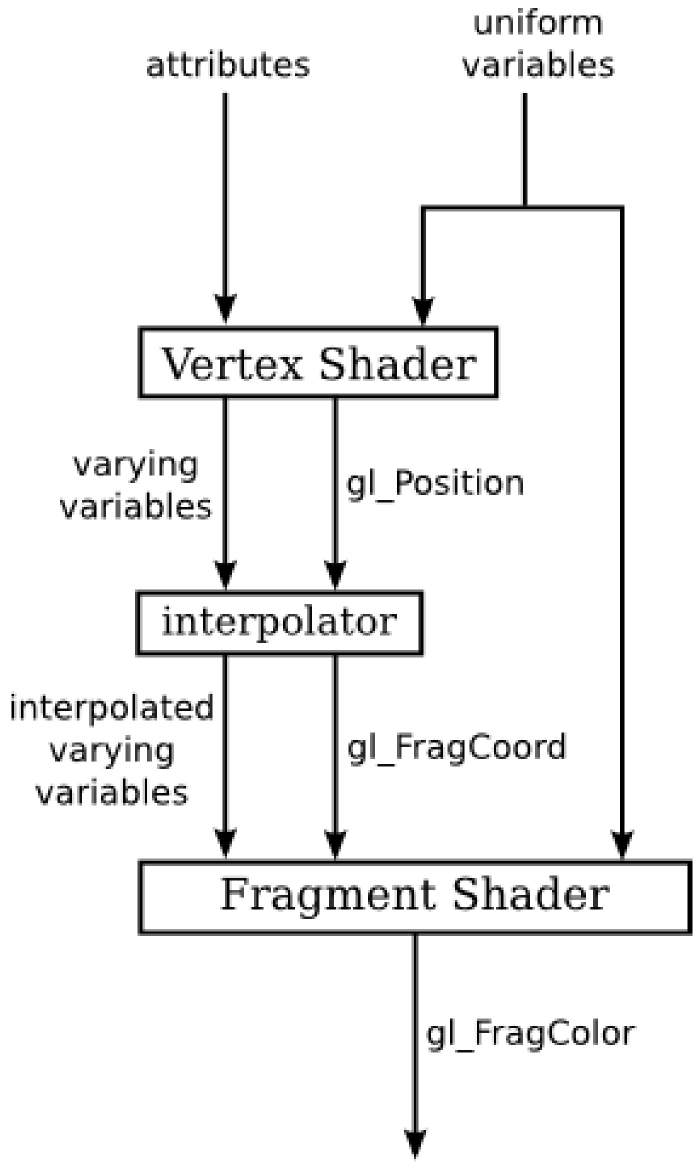
Fragments

- Varying variables exist to communicate data from the vertex shader to the fragment shader. They are defined in the shader source code. They are not used or referred to in the JavaScript side of the API. Note that it is entirely up to the programmer to decide what varying variables to define and what to do with them.
- After all that, the job of the fragment shader is simply to specify a color for the pixel. It does that by assigning a value to a special variable named gl_FragColor. That value will then be used in the remaining fixed-function stages of the pipeline.

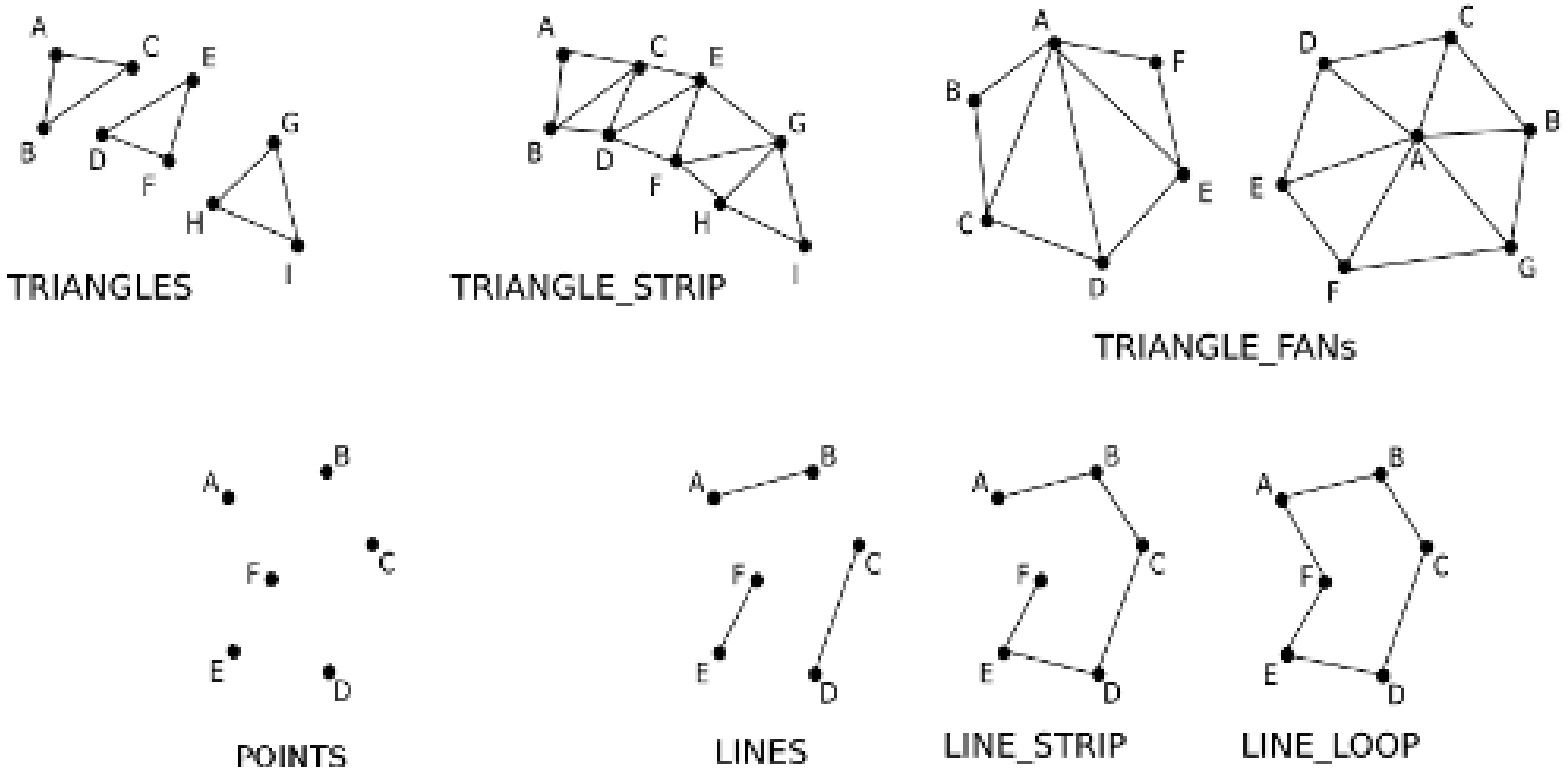
Summary

- The JavaScript side of the program sends values for attributes and uniform variables to the GPU and then issues a command to draw a primitive.
- The GPU executes the vertex shader once for each vertex.
- The vertex shader can use the values of attributes and uniforms. It assigns values to gl_Position and to any varying variables that exist in the shader. After clipping, rasterization, and interpolation, the GPU executes the fragment shader once for each pixel in the primitive.
- The fragment shader can use the values of varying variables, uniform variables, and gl_FragCoord. It computes a value for gl_FragColor

WebGL graphics pipeline



WebGL primitives



Drawing a primitve

- After the shader program has been created and values have been set up for the uniform variables and attributes, it takes just one more command to draw a primitive:

```
gl.drawArrays( primitiveType, startVertex, vertexCount );
```

- For example:

```
gl.drawArrays( gl.TRIANGLES, 0, 3 );
```

Simple WebGL example

- WebGL: can only work with VBO’s (vertex buffer objects)
  - This means that one has to allocate memory on the GPU and have a mechanism to populate it with data. Done by copying CPU arrays to VBOs

```
attributeCoords = gl.getAttribLocation(prog, "a_coords"); // a pointer to GPU memory
bufferCoords = gl.createBuffer(); // VBO instantiation
var coords = new Float32Array( [ -0.9,-0.8, 0.9,-0.8, 0,0.9 ] );
gl.bindBuffer(gl.ARRAY_BUFFER, bufferCoords); // WebGL is a state machine
gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STREAM_DRAW);
gl.vertexAttribPointer(attributeCoords, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(attributeCoords);
```

Simple WebGL triangle

- https://mypages.valdosta.edu/rpmihail/teaching/S18/CS4830/samples/basicwebgl.html
- In this example, we specify two vertex attributes: position and color
- Done by creating VBOs, populating them and associating them with a shader attribute
- REMEMBER: vertex shaders are called once per vertex and fragment shaders are called once per fragment
- Attributes are vectors, and can be of different types

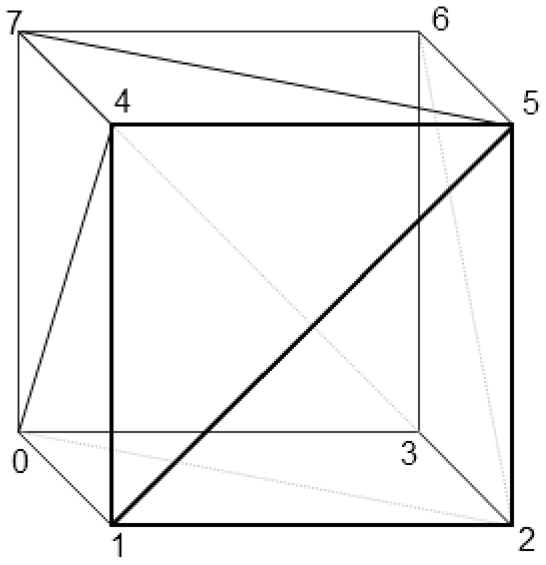
  (e.g.,: vec2 has x and y, vec3 has x, y, z)
  - Defined in vertexAttribPointer

vertexAttribPointer

- The first parameter to gl.vertexAttribPointer is the attribute location.
- The second is the number of values per vertex. For example, if you are providing values for a vec2, the second parameter will be 2 and you will provide two numbers per vertex; for a vec3, the second parameter would be 3; for a float, it would be 1.
- The third parameter specifies the type of each value. Here, gl.FLOAT indicates that each value is a 32-bit floating point number. Other values include gl.BYTE, gl.UNSIGNED_BYTE, gl.UNSIGNED_SHORT, and gl.SHORT for integer values.
  - Note that the type of data does not have to match the type of the attribute variable; in fact, attribute variables are always floating point. However, the parameter value does have to match the data type in the buffer. If the data came from a Float32Array, then the parameter must be gl.FLOAT. Book author will always use false, 0, and 0 for the remaining three parameters.

Drawing primitives

- In any case, one has to specify primitive vertex positions, done using a VBO
- Once a VBO is setup, consider mesh models made of many triangles. Below is a sample.
- Specifying 3 vertices for each triangle is inefficient. Why?



Solution

- Create a VBO with all UNIQUE vertices and setup an index buffer, that tells the GPU to draw triangles from vertices using **indices**
- gl.DrawElements can be used for drawing indexed face sets.
- With gl.drawElements, attribute data is not used in the order in which it occurs in the VBOs. Instead, there is a separate list of indices that determines the order in which the data is accessed.

Index buffer

- To use gl.drawElements, an extra VBO is required to hold the list of indices. When used for this purpose, the VBO must be bound to the target gl.ELEMENT_ARRAY_BUFFER rather than gl.ARRAY_BUFFER.
- The VBO will hold integer values, which can be of type gl.UNSIGNED_BYTE or gl.UNSIGNED_SHORT. The values can be loaded from a JavaScript typed array of type Uint8Array or Uint16Array.
- Creating the VBO and filling it with data is again a multi-step process

Using index buffers

- Creating index buffer

```
elementBuffer = gl.createBuffer();
gl.bindBuffer( gl.ELEMENT_ARRAY_BUFFER, elementBuffer );
var data = new Uint8Array( [ 2,0,3, 2,1,3, 1,4,3 ] );
gl.bufferData( gl.ELEMENT_ARRAY_BUFFER, data, gl.STREAM_DRAW );
```

- Assuming that the attribute data has also been loaded into VBOs, gl.drawElements can then be used to draw the primitive. A call to gl.drawElements takes the form

```
gl.drawElements( primitiveType, count, dataType, startByte );
```

Index buffer

- The VBO that contains the vertex indices must be bound to the ELEMENT_ARRAY_BUFFER target when this function is called. The first parameter to gl.drawElements is a primitive type such as gl.TRIANGLE_FAN.
- The count is the number of vertices in the primitive. The dataType specifies the type of data that was loaded into the VBO; it will be either gl.UNSIGNED_SHORT or gl.UNSIGNED_BYTE. The startByte is the starting point in the VBO of the data for the primitive; it is usually zero. (Note that the starting point is given in terms of bytes, not vertex numbers.) A typical example would be

```
gl.drawElements( gl.TRIANGLES, 9, gl.UNSIGNED_BYTE, 0 );
```