M30242-Graphics and Computer Vision

Lecture 03: Modelling and Drawing

Overview

- WebGL primitives
- Surface normal and vertex winding order
- Drawing methods
  - gl.drawArrays()
  - gl.drawElements()

WebGL Primitives

- You can use WebGL to create complex 3D models. However, these 3D models all have to be built up using the following three basic geometric primitives:
  - Triangles – for defining surfaces.
  - Lines – for drawing wireframes.
  - Point sprites – for create discrete particles (e.g., clouds, gas, etc).

Triangles

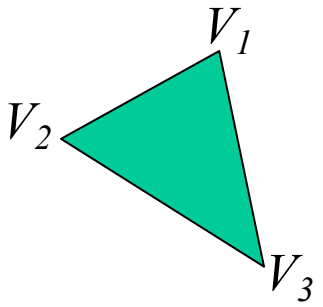
- Most 3D graphics hardware is highly optimized for drawing triangles.
- Triangles are seldom used as single, separate entities. Many triangles are connected for form polygon meshes to define surfaces.

Triangle Primitives

- Triangles, according to the way they are organised/connected, can form different geometric objects.
- WebGL provides three different triangle primitives:
  - **`gl.TRIANGLES`,**
  - **`gl.TRIANGLE_STRIP`, and**
  - **`gl.TRIANGLE_FAN`.**
- We will study the primitives in detail, but we first look at the properties of a single triangle.

Single Triangles

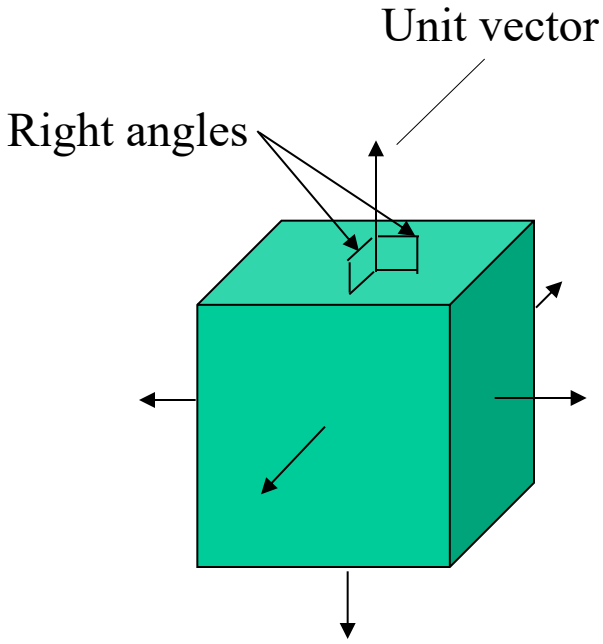
- A triangle is defined by three vertices (points).



- In other subjects, such as in geometry, we usually don't care about the order of the vertices. Given 3 vertices, the triangle is fully defined.
- But in CG, the order we list the vertices (called winding order) are important: it defines the surface normal of the triangle.

Surface Normals

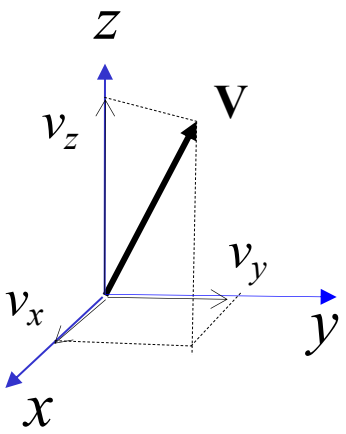
- In CG, a surface has two sides: inside and outside. A surface is only visible when we see it from outside. Seeing from inside, a surface is transparent.
- The outside (visible side) of a surface is indicated by its normal – a unit vector that is orthogonal to the surface where it adheres to and points away from the surface.



Vectors

- A vector is a geometric entity that has a direction and length.
- In math form, a vector is written as a column matrix:

$$\mathbf{V} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \quad \text{or} \quad \mathbf{V} = \begin{bmatrix} v_x, v_y, v_z \end{bmatrix}^T$$



where v_x, v_y and v_z are the components (projections) of the vector along each coordinate axes.

- Its length is calculated as

$$l = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

Unit Vector

- When using a vector to represent a direction, its size does not matter. In such cases, a unit vector is used.
- A unit vector is a vector that has unit length, i.e.,

$$l = \sqrt{v_x^2 + v_y^2 + v_z^2} = 1$$

We normally denote a unit vector with bold case letter $\mathbf{n}$ and its components by n_x, n_y and n_z, i.e.,

$$\mathbf{n} = \left[\mathrm{n}_x, \mathrm{n}_y, \mathrm{n}_z\right]^T \text{ and } \quad \sqrt{n_x^2 + n_y^2 + n_z^2} = 1$$

Vector Normalisation

- A non-unit vector can be normalised to make it a unit one
- A general vector V can be normalised by dividing its components by its length:

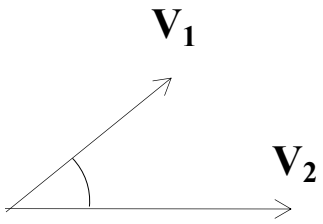
$$\mathbf{n} = \frac{\mathbf{V}}{l} = \frac{\mathbf{V}}{\sqrt{v_x^2 + v_y^2 + v_z^2}} = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} = \begin{bmatrix} v_x / l \\ v_y / l \\ v_z / l \end{bmatrix}$$

Dot Product of Vectors

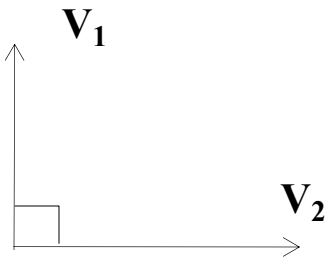
- The dot product of two vectors,v_1 and v_2 are defined as

$$\mathbf{V}_1 \bullet \mathbf{V}_2 = l_1 l_2 \cos\theta = \left(\sqrt{v_{1x}^2 + v_{1y}^2 + v_{1z}^2}\sqrt{v_{2x}^2 + v_{2y}^2 + v_{2z}^2}\right)\cos\theta$$

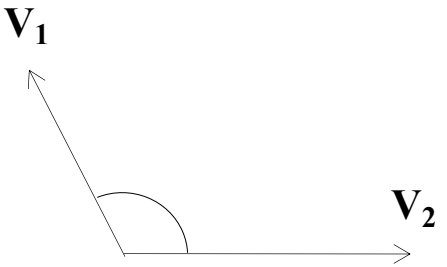
- We can tell the range of the angles between two vectors by their dot product:
  - if their dot product is 0, then the angle between them is 90 degrees,
  - if the dot product is greater than 0, then the angle between them is less than 90 degrees,
  - if the dot product is less than 0, then the angle between them is greater than 90 degrees.



θ<90, cosθ>0



θ=90, cosθ=0



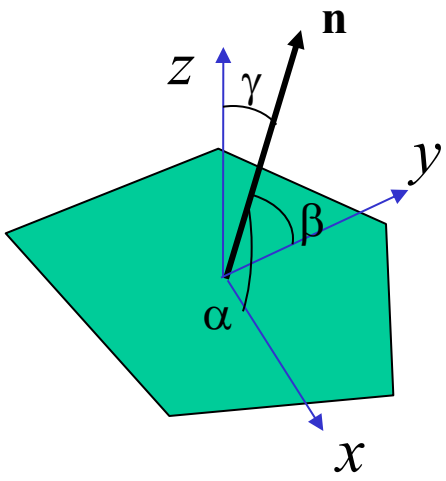
θ>90, cosθ<0

Calculation of Components of a Unit Vector

- A unit vector's components equal the dot products of the vector and the unit vectors representing the direction of the coordinate axes, i.e.,

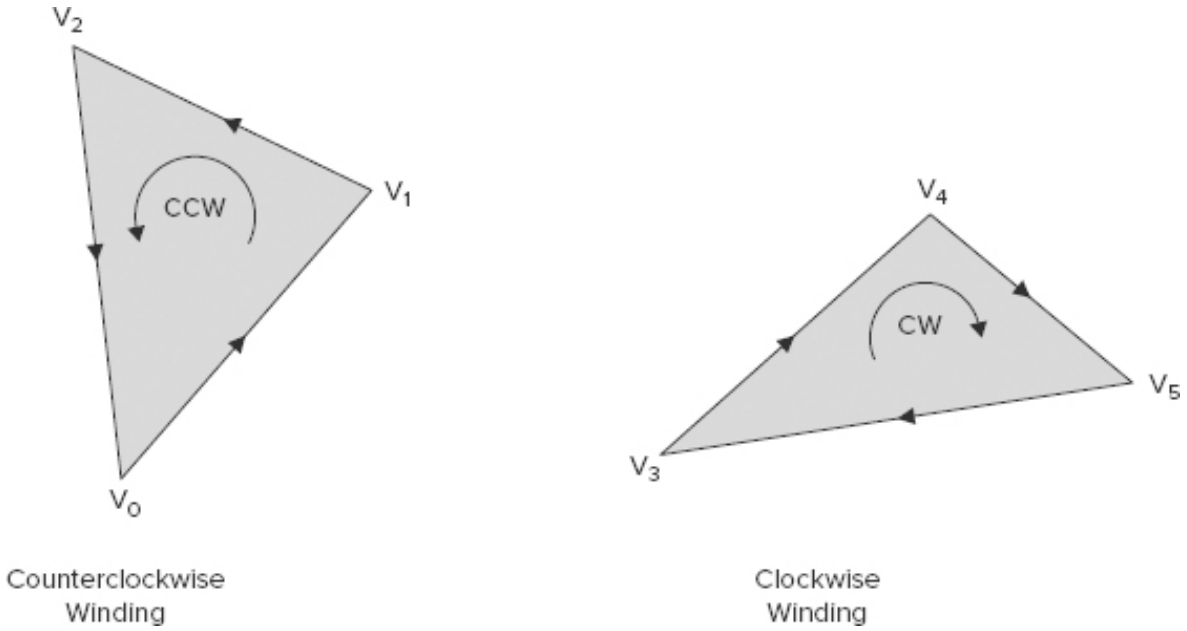
$$\mathbf{n} = \begin{bmatrix} n_x \\ n_y \\ n_y \end{bmatrix} = \begin{bmatrix} \cos\alpha \\ \cos\beta \\ \cos\gamma \end{bmatrix}$$

where α, β and γ are the angles between the vector and coordinate axes.



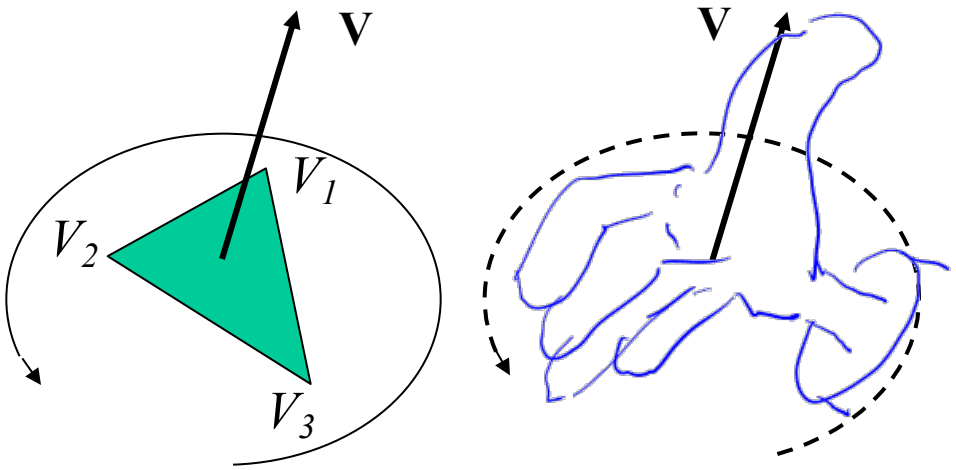
Vertex Wind Order & Surface Normal

- The winding could be clockwise (CW) and counter-clockwise (CCW)



CCW or CW

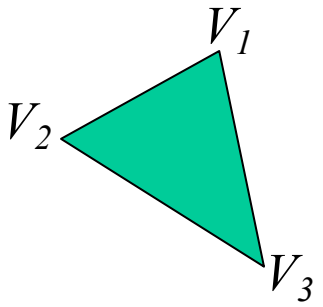
- CCW defines front-facing triangles, while CW defines triangles that are back-facing.



- This facing convention can be remember using the right-hand rule - pointing the thumb at the direction of the surface normal, then the direction of the other fingers indicates the order of winding

Specifying Normals

- Consider the triangle defined by vertices V1, V2, V3:



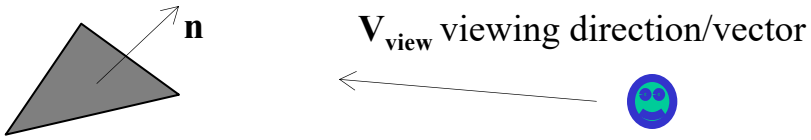
- By the convention, (V1, V2, V3), (V2, V3, V1) or (V3, V1, V2) define the same triangle. But (V1, V3, V2), (V3, V2, V1), or (V2, V1, V3) define a different triangle – the difference is in their facing direction (i.e., visibility).

Uses of Normals

- If the normal of a triangle points to the direction of viewer, the triangle is visible. We call the triangle is front-facing.
- If the normal points to the opposite side, the triangle is invisible and we call the triangle is back-facing.
- We normally do not rasterize triangles that are back-facing.
- This operation of removing such triangles is called back-surface culling.

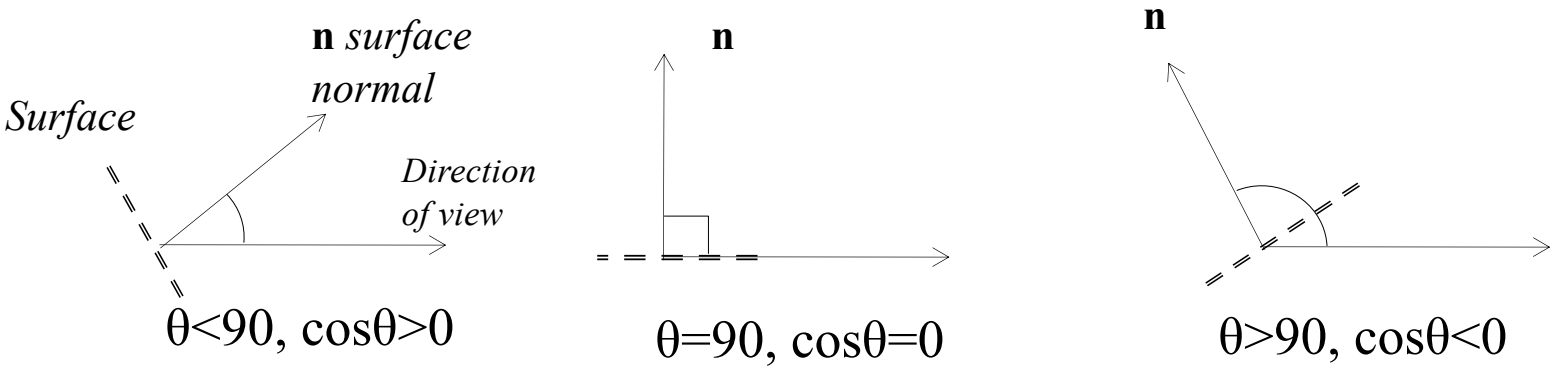
Cont'd

- Back facing triangles can be found by calculating the dot product of the surface normal, $\mathbf{n}$, and the vector representing the viewer's direction, $\mathbf{V}_{view}$.
- I.e., if $\mathbf{n}.\mathbf{V}_{view}>\mathbf{0}$, the triangle is back-facing.



Cont'd

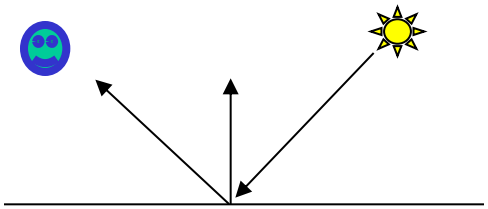
- The values (the sign + or -) are used to remove *back-facing* triangles.



Use of Normal for Shading

- Surface normal is also used in calculating the shade of a fragment/pixel.
- The shade of a fragment is determined by calculating the amount of light reflected from that fragment – the operation is called shading (more on this topic in later lectures).

How much light is reflected from this point? Or how bright/dark is the point when you see it from there?



It depends on many things: the direction of incidence (incident angle), surface material, where you see it,…

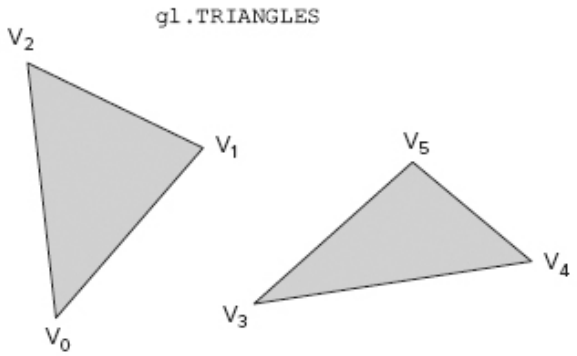
WebGL Primitives

- In WebGL, shapes are drawn using
  - Triangles, triangle strips or triangle fans
  - Lines, line strips and line loops
  - Points (point sprites)

- Advanced WebGL libraries use these primitives to define high level primitives such as sphere, cone, box, etc

gl.TRIANGLES

- In WebGL, independent/individual triangles are of the primitive type **`gl.TRIANGLES`**.



- If you use function `gl.drawArrays()` to draw triangles (there are other ways to draw triangles), you must specify three vertices for each triangle, and if you want to draw several triangles, they are drawn separately (i.e., vertices from one triangle are not automatically re-used in another triangle).

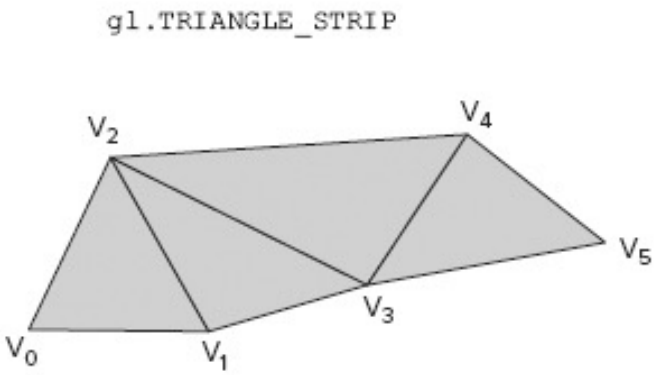
Cont'd

- Given the total number of vertices, *count*, the number of triangle you can be drawn is

Number of drawn triangles = *count* /3

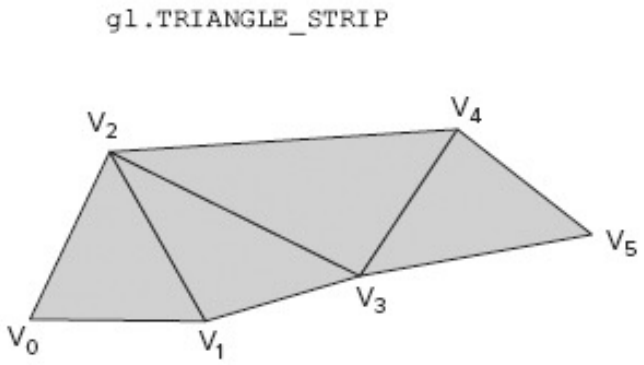
gl.TRIANGLE_STRIP

- For many geometric objects, it is more efficient to draw connected triangles.
- The primitive, **gl.TRIANGLE_STRIP**, has been designed for such use. A triangle strip reuses vertices between the triangles.



- This allows you to specify fewer vertices when building up the triangles for your geometric models.
- Fewer vertices mean less data to store, and less data to transfer from the memory to the GPU.

Drawing TRIANGLE_STRIP



- For the above triangle strip, the drawing procedure is as the following:
  - the first triangle in the strip is drawn from vertices (V0, V1, V2).
  - the second triangle is drawn from (V2, V1, V3).
  - the third triangle is from (V2, V3, V4), and
  - the fourth triangle is from (V4, V3, V5).
- We can see that, except for the first triangle, other triangles are created by adding a new vertex each time and reusing two vertices from the previous triangle.

Cont'd

- Put the vertex in a table we notice that for each pair of triangles, the last two vertices are reused in the second triangle and their order is reversed in the second triangle.

TRIANGLE NUMBER	CORNER 1	CORNER 2	CORNER 3
1	V_0	V_1	V_2
2	V_2	V_1	V_3
3	V_2	V_3	V_4
4	V_4	V_3	V_5

Note: the the order of the first two vertices are reversed in these two rows

Cont'd

- The **drawing pattern** shown in this example is followed for ***all triangle strips***. After the first triangle, *each new vertex, together with two vertices from the last triangle, creates a new triangle*.
- Note: the vertices for the first triangle were specified in **ccw** order, this order must be followed by all the other triangles in the strip.

Data Efficiency

- The number of triangles that are actually drawn when you use `gl.TRIANGLE_STRIP` and `gl.drawArrays()` will be

$$\text{Number of drawn triangles} = count - 2$$

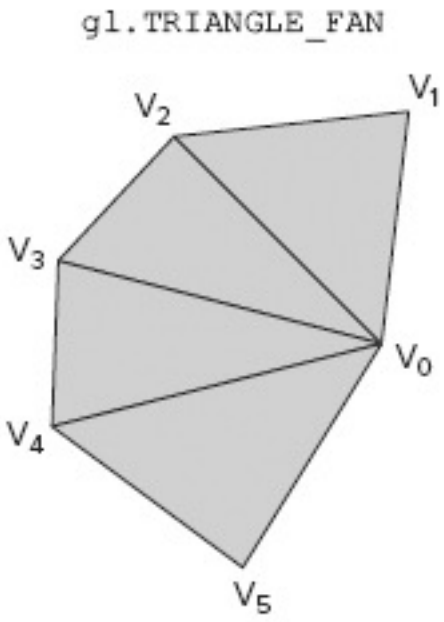
  where *count* is the total number of vertices.

- This gives a great improvement in data efficiency in comparison with *count/3* of the primitive `gl.TRIANGLES` when the value of *count* is big.

gl.TRIANGLE_FAN

- A triangle fan shares a common vertex (V_0), called origin.
- For a triangle fan, the first three vertices create the first triangle. The first vertex is also used as the origin for the fan, and after the first triangle every other vertex that is specified forms a new triangle with the previous vertex and the origin.
- E.g., (V0, V1, V2) creates the first triangle, (V0, V2, V3) creates the second, (V0, V3, V4) the third, and (V0, V4, V5) the fourth.
- The number of triangles that are drawn is also

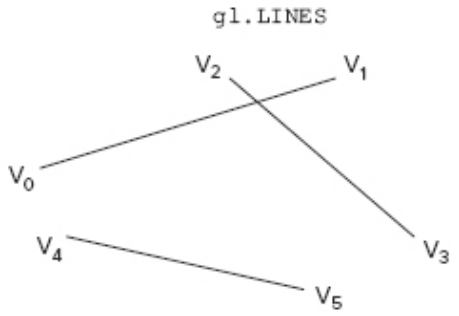
$$\text{Number of drawn triangles} = count - 2$$



WebGL Lines

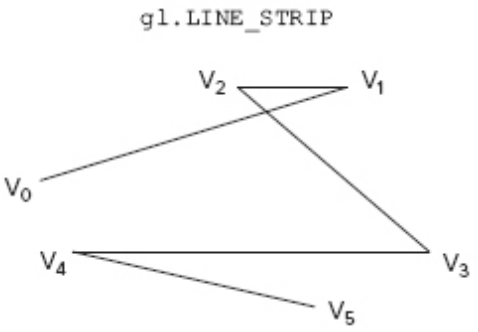
- **`gl.LINES`** are separate lines, e.g., lines between (V0, V1), (V2, V3), and (V4, V5)

  Number of drawn lines = vertex count/2



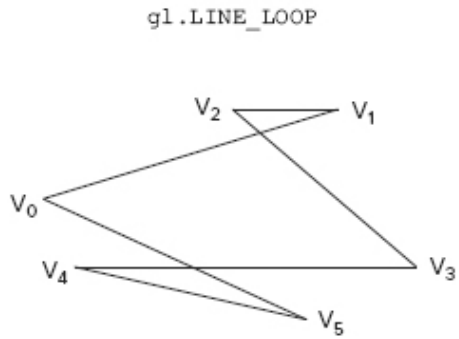
- **`gl.LINE_STRIP`** is a linked line, e.g., a line linked between (V0, V1), (V1, V2), (V2, V3), (V3, V4), and (V4, V5).

  Number of drawn lines = vertex count-1



- **`gl.LINE_LOOP`** dis line loop, e.g., loop formed with lines (V0, V1), (V1, V2), (V2, V3), (V3, V4), (V4, V5), and (V5, V0).

  Number of drawn lines = vertex count



Point Sprites

- The last primitive in WebGL is the point sprite, **`gl.POINTS`**.
- One point is rendered for each coordinate in the vertex array.
- When using a point sprite, one should also set the size of the point sprite in the vertex shader. This is done by writing the size in pixels to the built-in special variable **`gl_PointSize`**.
- E.g.,

```
<script id="shader-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexPos;
   void main() {
     gl_Position = vec4(aVertexPos, 1.0); //GLSL built-in
                                          //variable
     gl_PointSize = 5.0; ; //GLSL built-in variable
   }
</script>
```

Drawing Methods

- In WebGL, there are three methods for updating the drawing buffer:
  - `gl.drawArrays()`
  - `gl.drawElements()`
  - `gl.clear()`
- The first two methods are for drawing the primitives.
- Method `gl.clear()` is used to refresh the drawing buffer. It sets all the pixels to the colour that has been (previously) specified using the method `gl.clearColor()`.

gl.drawArrays()

- The prototype for gl.drawArrays() is as follows:

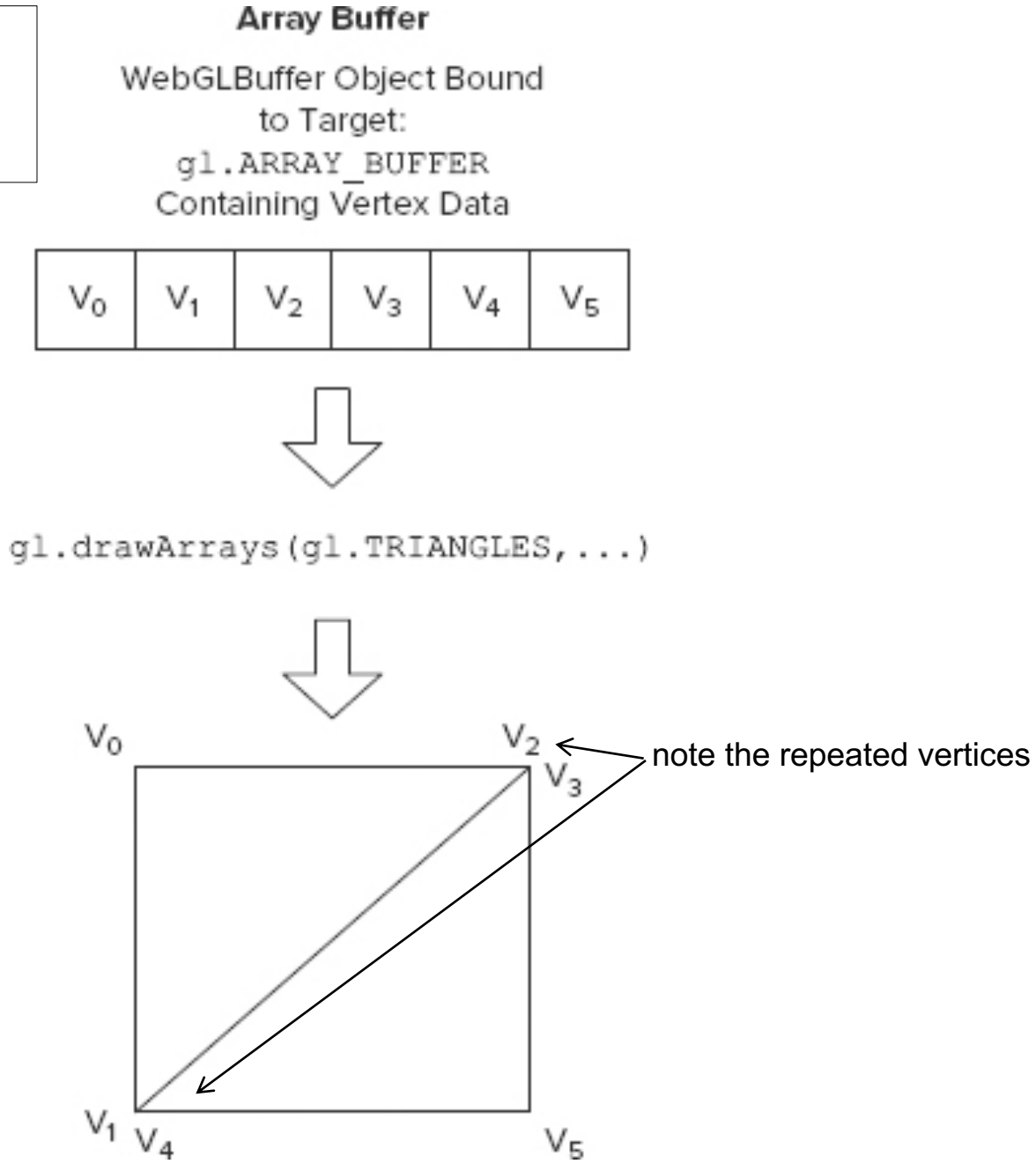
```
void drawArrays(GLenum mode, GLint first, GLsizei count)
```

  - `mode` specifies which primitives should be drawn. It could be: **`gl.POINTS`**, **`gl.LINES`**, **`gl.LINE_LOOP`**, **`gl.LINE_STRIP`**, **`gl.TRIANGLES`**, **`gl.TRIANGLE_STRIP`**, or **`gl.TRIANGLE_FAN`**
  - `first` specifies index (position) of the first data item in the array of vertex data that should be used as the first data entry.
  - `count` specifies how many vertices should be used.
- The function constructs a sequence of geometric primitives by successively transferring elements `first` through `first+count-1` of each enabled array to the specified primitive.

Cont'd

- `gl.drawArrays()` draws the primitives using the vertex data in the correct order in the *enabled* WebGLBuffer objects that are *bound* to the **`gl.ARRAY_BUFFER`** target.
- This means that before you can call `gl.drawArrrays()`, you must do the following:
  - *create* a WebGLBuffer object with `gl.createBuffer()`.
  - *bind* the WebGLBuffer object to the correct type, **`gl.ARRAY_BUFFER`**, using `gl.bindBuffer()`.
  - Load vertex data into the buffer using `gl.bufferData()`.
  - *enable* the generic vertex attribute array using `gl.enableVertexAttribArray()`.
  - Connect the correct data in the WebGLBuffer object with the attribute (e.g., the variable for vertex coordinates) in the vertex shader with by calling `gl.vertexAttribPointer()`.

WebGL Array Buffer



Cont'd

- The way `gl.drawArrays()` has been such designed that it makes mandatory to have the vertices for the primitives in correct order.
- The method is simple and fast if there are not many vertices to be shared.
- However, if you have an object that consists of a mesh of triangles that share a lot of vertices, it is better to use the method `gl.drawElements()`.

`gl.drawElements()`

- The method `gl.drawElements()` is also referred to as *indexed drawing*, which means the indices of the vertices are used in drawing.
- It is a way to reuse the vertices to save memory and data throughput.
- The method `gl.drawElements()` also uses the array buffers that contain the vertex data.
- But in addition, it uses an element array buffer (a WebGLBuffer object that is bound to the **gl.ELEMENT_ARRAY_BUFFER** target) to store the vertex indices of triangles.

Cont'd

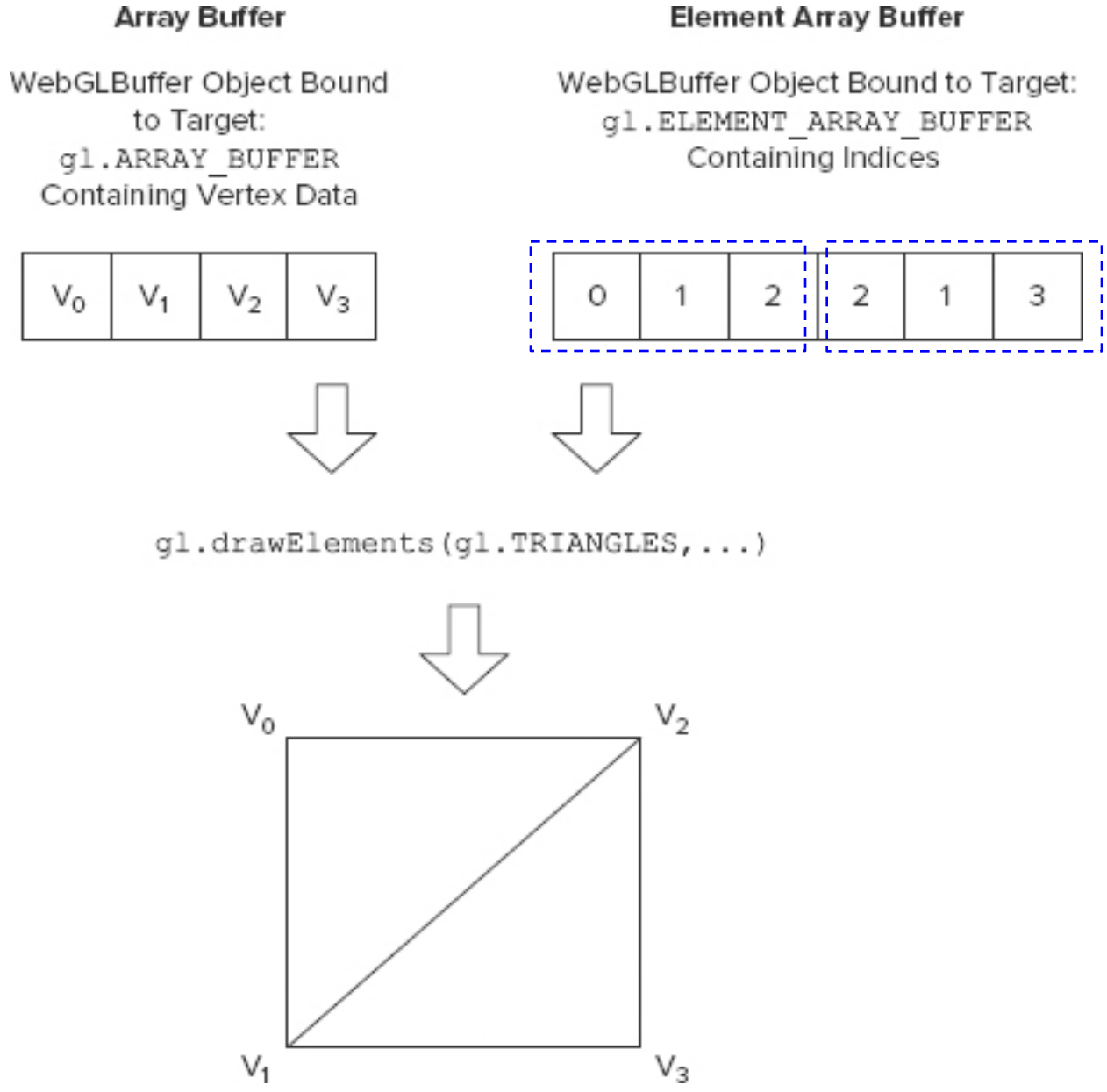
- When triangles share vertices, instead of storing the vertex data several copies, several items in the element array buffer point to the same data item in the vertex data buffer (of type **`gl.ARRAY_BUFFER`**)
- Therefore the vertex data in array buffers can be in **any** order since it is the indices in the element-array-buffer object that decide the order in which `gl.drawElements()` uses the vertices.

Function Prototype

- The prototype of drawElements() method:

```
void drawElements(GLenum mode, GLsizei count, GLenum
  type, GLintptr offset)
```

- The arguments are:
  - `mode` specifies the primitive you want to draw, the same as for the `drawArrays()` method.
  - `count` specifies how many indices you have in the buffer bound to the **`gl.ELEMENT_ARRAY_BUFFER`** target.
  - `type` specifies the data type for the element indices that are stored in the buffer bound to **`gl.ELEMENT_ARRAY_BUFFER`**. It could be
    - **`gl.UNSIGNED_BYTE`** or
    - **`gl.UNSIGNED_SHORT`**.
  - `offset` specifies the offset into the buffer bound to **`gl.ELEMENT_ARRAY_BUFFER`** where the indices start.


Array Buffer
WebGLBuffer Object Bound to Target:
gl.ARRAY_BUFFER
Containing Vertex Data
V0
V1
V2
V3
Element Array Buffer
WebGLBuffer Object Bound to Target:
gl.ELEMENT_ARRAY_BUFFER
Containing Indices
0
1
2
2
1
3
gl.drawElements(gl.TRIANGLES,...)
V0
V2
V1
V3

Usage

- When you can call `gl.drawElements()`, you need to set up the array buffers in the same way as for `gl.drawArrays()`.
- You also need to set up the element array buffer by doing the following:
  - Create a WebGLBuffer object using `gl.createBuffer()`.
  - Bind the WebGLBuffer object to the target **gl.ELEMENT_ARRAY_BUFFER** using `gl.bindBuffer()`.
  - Load indices that decide the order in which the vertex data are to be used into the buffer using `gl.bufferData()`.

Further Reading

- Anyuru, A., WebGL Programming – Develop 3D Graphics for the Web
  - Chapter 3, Section 1