Vertex Buffers are how you draw things in Vulkan. They are very much like Vertex Buffer Objects in OpenGL, but more detail is exposed to you (a lot more...).

But, the good news is that Vertex Buffers are really just ordinary Data Buffers, so some of the functions will look familiar to you.

First, a quick review of computer graphics geometry...
Vulkan Topologies – Requirements and Orientation

Polygons must be:
- Convex and
- Planar

Polygons are traditionally:
- CCW when viewed from outside the solid object

GL_TRIANGLES

It’s not absolutely necessary, but there are possible optimizations if you are consistent

OpenGL Topologies – Vertex Order Matters

VK_LINE_STRIP

VK_LINE_STRIP

Why is there a Requirement for Polygons to be Convex?
Graphics polygon-filling hardware can be highly optimized if you know that, no matter what direction you fill the polygon in, there will be two and only two intersections between the scanline and the polygon’s edges

What does “Convex Polygon” Mean?
We can go all mathematical here, but let’s go visual instead. In a convex polygon, a line between any two points inside the polygon never leaves the inside of the polygon.

Convex Not Convex

Why is there a Requirement for Polygons to be Planar?
Graphics hardware assumes that a polygon has a definite front and a definite back, and that you can only see one of them at a time

What if you need to display Polygons that are not Convex?
There is an open source library to break a non-convex polygon into convex polygons. It is called Polypartition, and is found here:
https://github.com/ivanfratric/polypartition

If you ever need to do this, contact me. I have working code …
Thanks to OpenGL, we are all used to drawing in a right-handed coordinate system. Internally, however, the Vulkan pipeline uses a left-handed system:

The best way to handle this is to continue to draw in a RH coordinate system and then fix it up in the projection matrix, like this:

\[
\text{ProjectionMatrix}[1][1] *= -1.;
\]

This is like saying "Y' = -Y".

---

A Colored Cube Example

---

Triangles in an Array of Structures

---

Filling the Vertex Buffer

---

A Reminder of What Init05DataBuffer Does

---
### Telling the Vulkan Pipeline about Input

When you come into the Pipeline later, but for now, know that a Vulkan pipeline is essentially a very large data structure that holds what OpenGL would call the state, including how to parse its input.

- **Vertex Input Stage**
  - **Input Assembly**
  - **Viewport**
  - **Color Blending**

### Telling the Pipeline about Its Input

We will come to the Pipeline later, but for now, know that a Vulkan pipeline is essentially a very large data structure that holds what OpenGL would call the state, including how to parse its input.

```
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets);
```

### Telling the Command Buffer what Vertices to Draw

We will come to Command Buffers later, but for now, know that you will specify the vertex buffer that you want drawn.

```
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance, &pDrawData);
```

### Vulkan Pipeline Specifications

- **Vertex Shader module**
- **Input Assembly State**
- **Depth/Stencil State**
- **Viewport State**
- **Dynamic State**
- **Depth Clamping**
- **Scissoring**
- **Tesselation Shaders, Geometry Shader**
- **Topology**

###管线布局

- **着色器模块**
- **输入装配状态**
- **深度/遮罩状态**
- **视口状态**
- **动态状态**
- **深度裁剪**
- **截断**
- **拓扑**

###管线属性

```cpp
struct vertex {
    float position[3];
    float normal[3];
    float color[3];
    float texCoord[2];
};
```

###管线创建

```cpp
vkCreatePipelineLayout(device, &pipelineLayoutCreateInfo, &pPipelineLayout);
```