**Vulkan Topologies**

- **VK_PRIMITIVE_TOPOLOGY_POINT_LIST**
- **VK_PRIMITIVE_TOPOLOGY_LINE_LIST**
- **VK_PRIMITIVE_TOPOLOGY_LINE_STRIP**
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST**
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP**
- **VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN**

**Non-indexed Buffer Drawing**

- **Struct vertex**
  - position
  - normal
  - color
  - texCoord

**Triangles Represented as an Array of Structures**

<table>
<thead>
<tr>
<th>Triangle 0-2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex 0:</td>
</tr>
<tr>
<td>(x, y, z)</td>
</tr>
<tr>
<td>(r, g, b)</td>
</tr>
<tr>
<td>(s, t)</td>
</tr>
<tr>
<td>Vertex 2:</td>
</tr>
<tr>
<td>(x, y, z)</td>
</tr>
<tr>
<td>(r, g, b)</td>
</tr>
<tr>
<td>(s, t)</td>
</tr>
<tr>
<td>Vertex 3:</td>
</tr>
<tr>
<td>(x, y, z)</td>
</tr>
<tr>
<td>(r, g, b)</td>
</tr>
<tr>
<td>(s, t)</td>
</tr>
</tbody>
</table>

**A Colored Cube Example**

- **Modeled in right-handed coordinates**
- **Actual Vertex Data**
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.

C/C++:

```c
struct vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
```

GLSL Shader:

```glsl
layout( location = 0 ) in vec3 aVertex;  // 0
layout( location = 1 ) in vec3 aNormal; // 1
layout( location = 2 ) in vec3 aColor;  // 2
layout( location = 3 ) in vec2 aTexCoord; // 3
```

Vulkan

```c
VkVertexInputAttributeDescription vviad[4]; // array per vertex input attribute
for (int i = 0; i < 4; ++i) {
    vviad[i].binding = i; // which binding description this is part of
    vviad[i].location = i; // location in the layout decoration
    vviad[i].offset = offsetof(struct vertex, vviad[i].attribute); // offset
    vviad[i].format = vviad[i].attribute; // e.g., VK_FORMAT_VEC3, VK_FORMAT_VEC4
}
```

Filling the Vertex Buffer

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.

```c
MyBuffer MyVertexDataBuffer;
Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
```

Does

```c
result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
```

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, OUT &vppc);
We will come to Command Buffers later, but for now, know that you will specify the vertex buffer that you want drawn.

```
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer);
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets );
const uint32_t vertexCount = sizeof( VertexData ) / sizeof( VertexData[0] );
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

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

```
struct vertex
JustVertexData[] = {
    // vertex #0: 
    { -1., -1., -1. },
    {  0.,  0., -1. },
    {  0.,  0.,  0. },
    {  1., 0. },
    // vertex #1: 
    {  1., -1., -1. },
    {  0.,  0., -1. },
    {  1.,  0.,  0. },
    {  0., 0. },
    . . .
```

```
int JustIndexData[] = 
{ 0, 2, 3,
  0, 3, 1,
  4, 5, 7,
  4, 7, 6,
  1, 3, 7,
  1, 7, 5,
  0, 4, 6,
  0, 6, 2,
  2, 6, 7,
  2, 7, 3,
  0, 1, 5,
  0, 5, 4,
  . . .
```

```
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );
typedef enum VkIndexType 
{ 
    VK_INDEX_TYPE_UINT16 = 0, // 0 – 65,535
    VK_INDEX_TYPE_UINT32 = 1, // 0 – 4,294,967,295
};
VkIndexType;
```

```
VkResult
Init05MyIndexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
    VkResult result = Init05DataBuffer(size, VK_BUFFER_USAGE_INDEX_BUFFER_BIT, pMyBuffer);
    // fills pMyBuffer
    return result;
}
Init05MyVertexDataBuffer( sizeof(JustVertexData), &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );
Init05MyIndexDataBuffer( sizeof(JustIndexData), &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );
```

```
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex,  vertexOffset, firstInstance);
```

```
// 0, 1 = firstBinding, bindingCount
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
const uint32_t vertexCount = sizeof(JustVertexData) / sizeof(JustVertexData[0]);
const uint32_t indexCount = sizeof(JustIndexData)  / sizeof(JustIndexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstIndex = 0;
const uint32_t firstInstance = 0;
// #ifdef VERTEX_BUFFER
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex,
firstInstance );
// #endif
// #ifdef INDEX_BUFFER
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex,
vertexOffset, firstInstance );
```

```
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
const uint32_t vertexCount = sizeof(JustVertexData) / sizeof(JustVertexData[0]);
const uint32_t indexCount = sizeof(JustIndexData)  / sizeof(JustIndexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstIndex = 0;
const uint32_t firstInstance = 0;
const uint32_t vertexOffset = 0;
// #ifdef VERTEX_BUFFER
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex,
firstInstance );
// #endif
// #ifdef INDEX_BUFFER
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex,
vertexOffset, firstInstance );
```

```
vkCmdDrawIndirect( CommandBuffers[nextImageIndex], buffer, offset, drawCount, stride);
```

```
typedef struct
VkDrawIndirectCommand 
{ 
    uint32_t    vertexCount;
    uint32_t    instanceCount;
    uint32_t    firstVertex;
    uint32_t    firstInstance;
} VkDrawIndirectCommand;
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Indexed Indirect Drawing (i.e., both Indexed and Indirect)

```c
vkCmdDrawIndexedIndirect(commandBuffer, buffer, offset, drawCount, stride);
```

```
typedef struct
{uint32_t    indexCount;
uint32_t    instanceCount;
uint32_t    firstIndex;
int32_t     vertexOffset;
uint32_t    firstInstance;
} VkDrawIndexedIndirectCommand;
```

Compare this with:

Sometimes a point that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn’t.

A color-interpolated cube like this actually has both. Point #7 above has the same color, regardless of what face it is in. However, Point #7 has 3 different normal vectors, depending on which face you are defining. Same with its texture coordinates.

Thus, when using index-ed buffer drawing, you need to create a new vertex struct if any of {position, normal, color, texCoords} changes from what was previously-stored at those coordinates.

### Sometimes the Same Point Needs Multiple Attributes

Where values do not match at the corners (texture coordinates)

Where values match at the corners (color)

The OBJ File Format – a triple-indexed way of Drawing

Note: The OBJ file format uses 1-based indexing for faces!