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

typedef enum VkPrimitiveTopology
{
    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,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
} VkPrimitiveTopology;

A Colored Cube Example

Triangles Represented as an Array of Structures

Non-indexed Buffer Drawing
Filling the Vertex Buffer

C/C++:

```c
MyBuffer MyVertexDataBuffer;

Init05MyVertexBuffer (IN MyVertexData buffer, OUT MyBuffer * pMyBuffer )
{
  VkDeviceSize size = sizeof (MyVertexData);
  result = Init05VertexBuffer (size, OUT MyBuffer * pMyBuffer );
  return result;
}
```

GLSL Shader:

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

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
Init05MyVertexBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
  result = Init05VertexBuffer (size, OUT MyBuffer * pMyBuffer );
  return result;
}
```

GLSL Shader:

```glsl
layout ( location = 3 ) in vec2 aTexCoord;
layout ( location = 2 ) in vec3 aColor;
layout ( location = 1 ) in vec3 aNormal;
layout ( location = 0 ) in vec3 aVertex;
```
We will come to Command Buffers later, but for now, know that you will specify the vertex buffer that you want drawn.

```cpp
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };  
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, 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.

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

```cpp
const uint32_t instanceCount = 1;
```
Indexed Indirect Drawing (i.e., both Indexed and Indirect)

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

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.

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

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

```
```

```

```