Vulkan Topologies

**VK_PRIMITIVE_TOPOLOGY_POINT_LIST**

- $V_0$
- $V_1$
- $V_2$
- $V_3$

**VK_PRIMITIVE_TOPOLOGY_LINE_LIST**

- $V_0$
- $V_1$
- $V_2$
- $V_3$

**VK_PRIMITIVE_TOPOLOGY_LINE_STRIP**

- $V_0$
- $V_1$
- $V_2$
- $V_3$
- $V_4$
- $V_5$

**VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST**

- $V_0$
- $V_1$
- $V_2$
- $V_3$
- $V_4$
- $V_5$

**VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP**

- $V_0$
- $V_1$
- $V_2$
- $V_3$
- $V_4$
- $V_5$
- $V_6$

**VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN**

- $V_0$
- $V_1$
- $V_2$
- $V_3$
- $V_4$
- $V_5$
- $V_6$

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
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

static GLuint CubeTriangleIndices[3] =
{ { 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 }
};
Triangles Represented as an Array of Structures

From the file `SampleVertexData.cpp`:

```cpp
struct vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
struct vertex VertexData[ ] = {
    // triangle 0-2-3:
    // vertex #0:
    { {-1., -1., -1.},
      { 0., 0., -1.},
      { 0., 0., 0.},
      { 1., 0. } },
    // vertex #2:
    { {-1.,  1., -1.},
      { 0., 0., -1.},
      { 0., 1.,  0.},
      { 1., 1. } },
    // vertex #3:
    { { 1.,  1., -1.},
      { 0., 0., -1.},
      { 1., 1.,  0.},
      { 0., 1. } },
};
```

Modeled in right-handed coordinates

Non-indexed Buffer Drawing

From the file `SampleVertexData.cpp`:

```cpp
struct vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
struct vertex VertexData[ ] = {
    // triangle 0-2-3:
    // vertex #0:
    { {-1., -1., -1.},
      { 0., 0., -1.},
      { 0., 0., 0.},
      { 1., 0. } },
    // vertex #2:
    { {-1.,  1., -1.},
      { 0., 0., -1.},
      { 0., 1.,  0.},
      { 1., 1. } },
    // vertex #3:
    { { 1.,  1., -1.},
      { 0., 0., -1.},
      { 1., 1.,  0.},
      { 0., 1. } },
};
```
Filling the Vertex Buffer

MyBuffer MyVertexBuffer;

Init05MyVertexBuffer( sizeof(VertexData), &MyVertexBuffer );
Fill05DataBuffer( MyVertexBuffer, (void *) VertexData );

VkResult
Init05MyVertexBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result;
    result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}

A Reminder of What Init05DataBuffer Does

VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = pMyBuffer->size = size;
    vbci.usage = usage;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer( LogicalDevice, IN &vbci, PALLOCATOR, OUT &pMyBuffer->buffer );
    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );  // fills vmr
    VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible();
    VkDeviceMemory vdm;
    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory( LogicalDevice, pMyBuffer->buffer, IN vdm, 0 );  // 0 is the offset
    return result;
}
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;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;
```

### VkVertexInputBindingDescription

```c
VkVertexInputBindingDescription vvibd[1];  // one of these per buffer data buffer
vvibd[0].binding = 0;          // which binding # this is
vvibd[0].stride = sizeof( struct vertex );            // bytes between successive structs
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

### VkVertexInputAttributeDescription

```c
VkVertexInputAttributeDescription vviad[4];      // array per vertex input attribute
vviad[0].location = 0;                  // location in the layout decoration
vviad[0].binding = 0;                   // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;       // x, y, z
vviad[0].offset = offsetof( struct vertex, position );                  // 0

vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC3;      // nx, ny, nz
vviad[1].offset = offsetof( struct vertex, normal );                  // 12

vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC3;      // r, g, b
vviad[2].offset = offsetof( struct vertex, color );                  // 24

vviad[3].location = 3;
vviad[3].binding = 0;
vviad[3].format = VK_FORMAT_VEC2;      // s, t
vviad[3].offset = offsetof( struct vertex, texCoord );                // 36
```

Repeat for Indexed???
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
VkPipelineVertexInputStateCreateInfo vpvisci;          // used to describe the input vertex attributes
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;
vpvisci.vertexBindingDescriptionCount = 1;
vpvisci.pVertexBindingDescriptions = vvibd;
vpvisci.vertexAttributeDescriptionCount = 4;
vpvisci.pVertexAttributeDescriptions = vviad;

VkPipelineInputAssemblyStateCreateInfo vpiasci;
vpasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpasci.pNext = nullptr;
vpasci.flags = 0;
vpasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vgpci, PALLOCATOR, OUT pGraphicsPipeline );
```
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.

Repeat for Indexed???
Drawing with an Indexed Buffer

```
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;
```

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

---

Drawing with an Indexed Buffer

```
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 );
```
**Drawing with an Indexed Buffer**

```c
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };  
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };  

vkCmdBindVertexBuffer(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);  
// 0, 1 = firstBinding, bindingCount  
vkCmdBindIndexBuffer(CommandBuffers[nextImageIndex], iBuffer, 0, VK_INDEX_TYPE_UINT32);  

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  

vkCmdDrawIndexed(CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance);  
```

**Indirect Drawing (not to be confused with Indexed)**

```c
typedef struct VkDrawIndirectCommand  
{  
  uint32_t vertexCount;  
  uint32_t instanceCount;  
  uint32_t firstVertex;  
  uint32_t firstInstanceId;  
} VkDrawIndirectCommand;

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

Compare this with:  

vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstanceId);  
```
Indexed Indirect Drawing (i.e., both Indexed and Indirect)

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

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

Compare this with:

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

Sometimes the Same Point Needs Multiple Attributes

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 match at the corners (color)

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

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

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