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

static GLfloat CubeVertices[3][8] = {
    {-1, -1, -1},
    {1, -1, -1},
    {1, 1, -1},
    {-1, 1, -1},
    {-1, -1, 1},
    {1, -1, 1},
    {1, 1, 1},
    {-1, 1, 1}
};

static GLfloat CubeColors[3][8] = {
    {0, 0, 0},
    {1, 0, 0},
    {0, 1, 0},
    {1, 1, 0},
    {0, 0, 1},
    {1, 0, 1},
    {0, 1, 1},
    {1, 1, 1}
};

static GLuint CubeTriangleIndices[3][6] = {
    {0, 2, 3},
    {0, 3, 1},
    {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. }}
};
```

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

```cpp
MyBuffer MyVertexDataBuffer;

InitMyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
FillDataBuffer( MyVertexDataBuffer, (void *) VertexData );

VkResult
InitMyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result;
    result = InitDataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}
```

A Reminder of What `InitDataBuffer` Does

```cpp
VkResult
InitDataBuffer( 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;
}
```
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++
struct vertex
{
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
```

**C/C++:**

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

**GLSL Shader:**

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

Telling the Pipeline about its Input

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

```c
VkVertexInputAttributeDescription vviad[4];      // array per vertex input attribute
// 4 = vertex, normal, color, texture coord
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
```
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.

```cpp
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;
vkvec.type = VK_VERTEX_BINDING_DESCRIPTION_TYPE_VARIOUS;
vkvec.pNext = nullptr;
vpvisci.vertexAttributeDescriptionCount = 4;
vkvec.pNext = nullptr;
vpvisci.pVertexAttributeDescriptions = vpvec;

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

VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
vgpci.stageCount = 2;                // number of shader stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;            // &vptsci
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdsci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;                       // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci,
                                      PALLOCATOR, OUT pGraphicsPipeline );
```

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.

```cpp
VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
vgpci.stageCount = 2;                // number of shader stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;            // &vptsci
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.pMultisampleState = &vpmsci;
vgpci.pDepthStencilState = &vpdsci;
vgpci.pColorBlendState = &vpcbsci;
vgpci.pDynamicState = &vpdsci;
vgpci.layout = IN GraphicsPipelineLayout;
vgpci.renderPass = IN RenderPass;
vgpci.subpass = 0;                       // subpass number
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci,
                                      PALLOCATOR, OUT pGraphicsPipeline );
```
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, 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);
```

```
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,
};
```

### Drawing with an Indexed Buffer

<table>
<thead>
<tr>
<th>Transmission Order</th>
<th>Actual Index Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Triangle 4-5-7</td>
</tr>
<tr>
<td>7</td>
<td>Triangle 0-3-1</td>
</tr>
<tr>
<td>6</td>
<td>Triangle 0-2-3</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Triangles**

- Triangle 0-2-3
- Triangle 0-3-1
- Triangle 4-5-7
vkCmdBindVertexBuffer( 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);

Remember that integer-indexed buffers are just BLOBs too.

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 }; 

vkCmdBindVertexBuffers( 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  

#ifdef INDEX_BUFFER  
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );  
#endif  
```

Note that there is no vertex-count! It is up to you to not exceed the number of vertices with your index numbers!

Indirect Drawing (not to be confused with Indexed)

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

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

Compare this with:

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

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

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

Compare this with:

```cpp
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

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