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

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

A Colored Cube Example

Modelled in right-handed coordinates

Non-indexed Buffer Drawing

Stream of Vertices

Triangles
A Preview of What `Init05DataBuffer` Does

```c
VkResult Init05DataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkBufferCreateInfo vbci;
    VkDeviceMemory vdm;

    vbci.size = size;
    vbci.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
    vbci.pNext = nullptr;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;

    result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
    result = vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );
    vbci.alignment = vmr.alignment;
    result = vkCreateBuffers( LogicalDevice, IN &vbci, 1, OUT &pMyBuffer->buffer );

    return result;
}
```

Filling the Vertex Buffer

```c
MyBuffer MyVertexBufferBuffer;

Init05MyVertexBufferBuffer(sizeof(VertexData), OUT &MyVertexBufferBuffer);

VkResult result = Init05MyVertexBufferBuffer(IN size, OUT &MyVertexBufferBuffer);

result = vkBindBufferMemory( LogicalDevice, IN pMyBuffer->buffer, IN vdm, 0 );
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
result = vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );
vbci.alignment = vmr.alignment;
result = vkCreateBuffers( LogicalDevice, IN &vbci, 1, OUT &pMyBuffer->buffer );

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 vertex input.

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

```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;
```

Telling the Pipeline about its Input

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

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

Ini05DataBuffer

```c
result = Init05MyVertexBufferBuffer(IN size, OUT &MyVertexBufferBuffer);
```

```
 result = Init05MyVertexBufferBuffer(IN size, OUT &MyVertexBufferBuffer);
```

```
 result = vkBindBufferMemory( LogicalDevice, IN pMyBuffer->buffer, IN vdm, 0 );
```

```
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
```

```
result = vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );
```

```
vbci.alignment = vmr.alignment;
```

```
result = vkCreateBuffers( LogicalDevice, IN &vbci, 1, OUT &pMyBuffer->buffer );
```

```
return result;
```

```
result = vkBindBufferMemory( LogicalDevice, IN pMyBuffer->buffer, IN vdm, 0 );
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );
result = vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );
```

```
vbci.alignment = vmr.alignment;
```

```
result = vkCreateBuffers( LogicalDevice, IN &vbci, 1, OUT &pMyBuffer->buffer );
```

```
return result;
```
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.

Telling the Command Buffer what Vertices to Draw

Always use the C/C++ construct sizeof, rather than hardcoding a count!

Drawing with an Index 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;

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), IN &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );

Init05MyIndexDataBuffer( sizeof(JustIndexData), IN &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer, (void *) JustIndexData );

Triangles

Stream of Vertices

Stream of Indices

Stream of Vertices

Stream of Indices

Stream of Vertices

Stream of Indices

Stream of Vertices

Stream of Indices

Triangles

Vertices

Triangles

Vertices

Triangles

Vertices

Indirect Drawing (not to be confused with Indexed)

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

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

Compare this with:

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

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

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

Sometimes the Same Point Needs Multiple Attributes

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

| v 1.710541 1.283360 -0.040860 |
| v 1.714593 1.273043 -0.041268 |
| v 1.706114 1.279109 -0.040795 |
| v 1.719083 1.277235 -0.041195 |
| v 1.722786 1.267216 -0.041939 |
| v 1.727196 1.271285 -0.041795 |
| v 1.730680 1.261384 -0.042630 |
| v 1.723121 1.280378 -0.037323 |
| v 1.714513 1.286599 -0.037101 |
| v 1.706156 1.293797 -0.037073 |
| v 1.702207 1.290297 -0.040704 |
| v 1.697843 1.285852 -0.040489 |
| v 1.709169 1.295845 -0.029862 |
| v 1.717523 1.288344 -0.029807 |
| v 1.712412 0.320344 -0.298677 |
| vn 0.1725 0.2557 -0.9512 |
| vn -0.1979 -0.1899 -0.9616 |
| vn -0.2050 -0.2127 -0.9554 |
| vn 0.1664 0.3020 -0.9387 |
| vn -0.2040 -0.1718 -0.9638 |
| vn 0.1645 0.3203 -0.9329 |
| vn -0.2055 -0.1698 -0.9638 |
| vn 0.4419 0.6436 -0.6249 |
| vn 0.4573 0.5682 -0.6841 |
| vn 0.5160 0.5538 -0.6535 |
| vn 0.1791 0.2082 -0.9616 |
| vn -0.2167 -0.2250 -0.9499 |
| vn 0.6624 0.6871 -0.2987 |
| vt 0.816406 0.955536 |
| vt 0.822754 0.959168 |
| vt 0.815918 0.959442 |
| vt 0.823242 0.955292 |
| fort 0.829102 0.958862 |
| fort 0.829590 0.955109 |
| fort 0.835449 0.958618 |
| fort 0.824219 0.951263 |
| fort 0.817383 0.951538 |
| fort 0.810059 0.951385 |
| fort 0.809570 0.955383 |
| fort 0.809082 0.959320 |
| fort 0.811035 0.946381 |
| . . . |
| f 73/73/75 65/65/67 66/66/68 |
| f 66/66/68 74/74/76 73/73/75 |
| f 74/74/76 66/66/68 67/67/69 |
| f 67/67/69 75/75/77 74/74/76 |
| f 75/75/77 67/67/69 69/69/71 |
| f 71/71/73 72/72/74 77/77/79 |
| . . . |

Sometimes the Same Point Needs Multiple Attributes

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!