Vulkan Topologies

The same as OpenGL topologies, with a few left out.

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

The data is contained in the file SampleVertexData.cpp.

Triangles Represented as an Array of Structures

This data is contained in the file SampleVertexData.cpp.

Non-indexed Buffer Drawing

Stream of Vertices

This data is contained in the file SampleVertexData.cpp.
We will come to the Pipeline Data Structure 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.

Telling the Pipeline about its Input

C/C++:

```
struct MyVertexDataBuffer
{
    VkDeviceSize size;
    VkBufferUsageFlags usage;
    MyBuffer* pMyBuffer;
};
```

GLSL Shader:

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

Always use the C/C++ sizes() construct rather than hardcoding the byte count!

Telling the Pipeline Data Structure about its Input

```
VkPipelineLayoutCreateInfo

VkPhysicalDeviceVertexInputStateCreateInfo
```

A Preview of What `init05DataBuffer` Does

```
result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer
```

Does
Telling the Command Buffer what Vertices to Draw

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

```c
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };
VkDeviceSize offsets[1] = { 0 };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );
const uint32_t firstInstance = 0;
const uint32_t firstVertex = 0;
const uint32_t instanceCount = 1;
const uint32_t vertexCount = sizeof( VertexData ) / sizeof( VertexData[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 byte count!

Drawing with an Index Buffer

```c
vkCmdBindIndexedBuffer(commandBuffer, commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );
```

```c
typedef enum VkIndexType
{ VK_INDEX_TYPE_UINT16 = 0, // 0 – 65,535
  VK_INDEX_TYPE_UINT32 = 1, // 0 – 4,294,967,295
};
VkIndexType;
```

Drawing with an Index Buffer

```c
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
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;
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
```

Drawing with an Index Buffer

```c
VkBuffer vBuffers[1] = { MyJustVertexDataBuffer.buffer };
VkBuffer iBuffer = { MyJustIndexDataBuffer.buffer };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets );
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;
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex, vertexOffset, firstInstance );
```

Indirect Drawing (not to be confused with Indexed)

```c
void VkCmdDrawIndirect( CommandBuffers[whichImageIndex], buffer, offset, drawCount, stride);
```

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

Compare this with:

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

\[
\begin{array}{l}
\text{typedef struct}
\text{VkDrawIndexedIndirectCommand}
\text{\{}
\text{uint32_t indexCount;}
\text{uint32_t instanceCount;}
\text{uint32_t firstIndex;}
\text{int32_t vertexOffset;}
\text{uint32_t firstInstance;}
\}\text{ VkDrawIndexedIndirectCommand;}
\end{array}
\]

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

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

Compare this with:

\[
\text{In Vulkan, “Indirect” means that you store the arguments in GPU memory and then give the vkCmd\_Xxx call a pointer to those arguments.}
\]

Sometimes a vertex 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. Vertex #7 above has the same color, regardless of what face it is in. However, Vertex #7 has 3 different normal vectors, depending on which face you are defining. Same with its texture coordinates.

Thus, when using indexed 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.

### Drawing an OBJ Object

```
MyBuffer MyObjBuffer; // global
    . . .
MyObjBuffer = VkOsuLoadObjFile( "filename.obj" ); // initializes and fills the buffer with triangles defined in GPU memory with an array of struct vertex
```

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

Sometimes the Same Vertex 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

```
v 7.150541 1.262991 -0.946550
v 7.149551 1.275403 -0.941368
v 7.151141 1.273108 -0.949783
v 7.198631 1.297735 -0.941915
v 7.122796 1.287215 -0.943038
v 7.127106 1.273285 -0.947105
v 7.139665 1.261384 -0.942530
v 7.173210 1.282778 -0.937223
v 7.145151 1.298596 -0.931011
v 7.180391 1.297615 -0.935158
v 7.152071 1.266207 -0.947014
v 7.157650 1.285822 -0.946889
v 7.159149 1.285845 -0.939862
v 7.175023 1.286334 -0.938877

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
vt 0.829102 0.958862
vt 0.829590 0.955109
vt 0.835449 0.958618
vt 0.824219 0.951263
vt 0.817383 0.951538
vt 0.810059 0.951385
vt 0.809570 0.955383
vt 0.809082 0.959320
vt 0.811035 0.946381

. . .
```