Vulkan Topologies

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_TRIANGLE_LIST_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
} VkPrimitiveTopology;

A Colored Cube Example

static GLuint CubeTriangleIndices[] =
{
    { 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:

```
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. }
};
```

Triangulated in right-handed coordinates.

Non-indexed Buffer Drawing

From the file SampleVertexData.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. }
};
```

Stream of Vertices

Vertex 7
Vertex 5
Vertex 4
Vertex 1
Vertex 3
Vertex 0
Vertex 2
Vertex 0

A Preview of What Init05DataBuffer Does

```
VkResult Init05DataBuffer( IN VkDeviceSize size, IN 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;
    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 );
    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, IN pMyBuffer->buffer, IN vdm, 0 );
    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;
```

Always use the C/C++ construct `offsetof`, rather than hardcoding the value!

### Vulkan

```
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;
```

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.

### VkPipelineVertexInputStateCreateInfo

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

```
VkPipelineCreationInfo 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.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
vgpci.pViewportState = &vpvsci;
vgpci.pRasterizationState = &vprsci;
vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vgpci.basePipelineIndex = 0;
```

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

```c
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, 2, 6, 7, 2, 7, 5,
    0, 1, 5, 0, 5, 4, 7
};
```

```
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 );
```
Drawing with an Index Buffer

```cpp
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 vertexOffset = 0;
vkCmdDrawIndexed(CommandBuffers[nextImageIndex], indexCount, instanceCount, firstIndex,
vertexOffset, firstInstance);
```

Indirect Drawing (not to be confused with Indexed)

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

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

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;
```

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 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.
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!