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:

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

struct vertex VertexData[3] =
{
    // 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. }
    }
};
```

Modelled 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[3] =
{
    // 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. }
    }
};
```

Transmission Order

Triangle 0-3-1

Triangle 2-3

Triangle 4-5-7

Actual Vertex Data

Draw

A Reminder of What Init05DataBuffer Does

```cpp
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 );
    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;
}
```
### 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/C++:

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

#### 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

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.

#### 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[1];
vpvisci->vertexAttributeDescriptionCount = 4;
vpvisci->pVertexAttributeDescriptions = vviad[4];
```

#### VkGraphicsPipelineCreateInfo

```c
vkGraphicsPipelineCreateInfo *vgpci;                // number of shader stages in this pipeline
vgpci->sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
v pci->pNext = nullptr;
vgpci->flags = 0;
vgpci->stageCount = 2;
vgpci->pStages = vpssci;
vgpci->pVertexInputState = &vpvisci;
vgpci->pInputAssemblyState = &vpiasci;
vgpci->pTessellationState = &vptsci;
vgpci->pViewportState = &vpvsci;
vgpci->pRasterizationState = &vprsci;
vgpci->pMultisampleState = ...
vgpci->basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
vpci->basePipelineIndex = 0;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vgpci, PALLOCATOR, OUT pGraphicsPipeline );
```
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;
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. },

    // ...
}
```

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

```c
const int JustIndexData[] =
{ 0, 2, 3,
  0, 3, 1,
  4, 5, 7,
  4, 7, 5,
  1, 3, 7,
  1, 7, 5,
  0, 4, 6,
  0, 6, 2,
  2, 6, 7,
  2, 7, 3,
  0, 1, 5,
  0, 5, 4,
};
```

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

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

Remember that integer-indexed buffers are just BLOBs too.

```c
VkResult Init05MyIndexDataBuffer(IN VkDeviceSize size, OUT MyBuffer * pMyBuffer)
{
    VkResult result = Init05DataBuffer(size, VK_BUFFER_USAGE_INDEX_BUFFER_BIT, pMyBuffer);
    // fills pMyBuffer
    return result;
}
```

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

```c
Init05MyIndexDataBuffer( sizeof(JustVertexData), &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );
```

```c
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, vertexDataBuffers, offsets );
vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );
vkCmdDrawIndexed( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```
Drawing with an Indexed Buffer

```c
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, vBuffers, offsets);
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;

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

#if defined VERTEX_BUFFER
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
#endif

#if defined INDEX_BUFFER
vkCmdDrawIndexed(CommandBuffers[nextImageIndex], vertexCount, 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!

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

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

Compare this with:

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

Sometimes the Same Point Needs Multiple Attributes

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!