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;

static GLuint CubeTriangleIndices[3][3] =
{
    { 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 },
    { 6, 1, 5 },
    { 0, 5, 4 }
};

This data is contained in the file SampleVertexData.cpp.
Triangles Represented as an Array of Structures

```c
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. }
};
```

This data is contained in the file SampleVertexData.cpp

Non-indexed Buffer Drawing

```c
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. }
};
```

From the file SampleVertexData.cpp:

Triangles

<table>
<thead>
<tr>
<th>Vertex 7</th>
<th>Vertex 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

A Preview of What Init05DataBuffer Does

```c
VkResult Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result;
    VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.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 result = vkAllocateMemory(LogicalDevice, IN &vmai, PALLOCATOR, OUT &pMyBuffer->vdm);
    pMyBuffer->vdm = vdm;
    result = vkBindBufferMemory(LogicalDevice, pMyBuffer->buffer, IN pMyBuffer->vdm, 0); // 0 is the offset
    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++ `offsetof()` construct rather than hardcoding the byte count!

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.

vkPipelineInputAssemblyStateCreateInfo vpiasci;
```
vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;```

vkPipelineVertexInputStateCreateInfo vpvisci;
```
vpvisci.vertexAttributeDescriptionCount = 4;
vpvisci.pVertexAttributeDescriptions = vviad;
vpvisci.pVertexBindingDescriptions = vvibd;
```

Always use the C/C++ `offsetof()` rather than hardcoding the byte offset!

Telling the Pipeline Data Structure about its Input

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.

VkPipelineVertexInputStateCreateInfo vpvisci;
```
vvisci = vs::struct vertex {
    glm::vec3 position;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoord;
};
```

VkVertexInputBindingDescription vvibd[4]; // array per buffer data buffer
```
vvibd[0].binding = 0;
vvibd[0].stride = sizeof(struct vertex);
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[0].offset = 0;
vvibd[1].binding = 0;
vvibd[1].stride = offsetof(struct vertex, position);
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[1].offset = offsetof(struct vertex, position);
vvibd[2].binding = 0;
vvibd[2].stride = offsetof(struct vertex, color);
vvibd[2].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[2].offset = offsetof(struct vertex, color);
vvibd[3].binding = 0;
vvibd[3].stride = offsetof(struct vertex, texCoord);
vvibd[3].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[3].offset = offsetof(struct vertex, texCoord);
```

Telling the Pipeline about its Input

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.

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vpvsci, OUT &GraphicsPipeline );
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 };  
VkDeviceSize offsets[1] = { 0 };  
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );
```

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

```cpp
vkCmdBindVertexArray( vertexDataBuffers, -1 );
```

```cpp
const uint32_t firstInstance = 0;
const uint32_t firstVertex = 0;
const uint32_t instanceCount = 1;
const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
```

```cpp
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

---

**Drawing with an Index Buffer**

```cpp
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, vertexDataBuffers, vertexOffsets );
```

```cpp
vkCmdBindIndexBuffer( commandBuffer, indexDataBuffer, indexOffset, indexType );
```

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

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

```cpp
Init05MyVertexDataBuffer(IN &MyJustVertexDataBuffer);  
Fill05DataBuffer( MyJustVertexDataBuffer, (void *) JustVertexData );
```

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

---

**Oklahoma State University**

Computer Graphics

mjb - December 26, 2022
**Drawing with an Index Buffer**

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

**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;  
```

In Vulkan, "indirect" means that you store the arguments in GPU memory and then give the vkCmdXxx call a pointer to those arguments.

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

Compare this with:

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

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

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

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

In Vulkan, "indirect" means that you store the arguments in GPU memory and then give the vkCmdXxx call a pointer to those arguments.

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

Compare this with:

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

**Sometimes the Same Vertex Needs Multiple Attributes**

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.
Sometimes the Same Point Needs Multiple Attributes

Where values match at the corners (color)

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

Where values match at the corners (color)

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

vn -0.1979 -0.1899 -0.9616
vn -0.2040 -0.1718 -0.9638
vn -0.2050 -0.2127 -0.9554
vn 0.1664 0.3020 -0.9387
vn -0.2040 -0.1718 -0.9638
vn 0.1645 0.3030 -0.9329
vn -0.2055 -0.1696 -0.9638
vn -0.2050 -0.2127 -0.9554
vn 0.1664 0.3020 -0.9387
vn 0.1645 0.3030 -0.9329
vn -0.2055 -0.1696 -0.9638
vn 0.1664 0.3020 -0.9387
vn 0.1645 0.3030 -0.9329

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;
};

V T N

v 1.816406 0.955536
v 1.827754 0.959168
v 1.810918 0.959442
v 1.823422 0.952920
v 1.829102 0.958862
v 1.829950 0.951099
v 1.835449 0.958518
v 1.828419 0.951263
v 1.817383 0.951538
v 1.810059 0.951385
v 1.809082 0.953200
v 1.811005 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 69/69/71 79/79/79 75/75/77
f 71/71/73 72/72/74 77/77/79
f 72/72/74 78/78/80 77/77/79
f 78/78/80 72/72/74 73/73/75
f 73/73/75 79/79/81 78/78/80
f 79/79/81 73/73/75 74/74/76
f 74/74/76 80/80/82 79/79/81
f 80/80/82 74/74/76 75/75/77

The OBJ File Format – a triple-indexed way of Drawing

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