



Drawing



Oregon State University

Mike Bailey

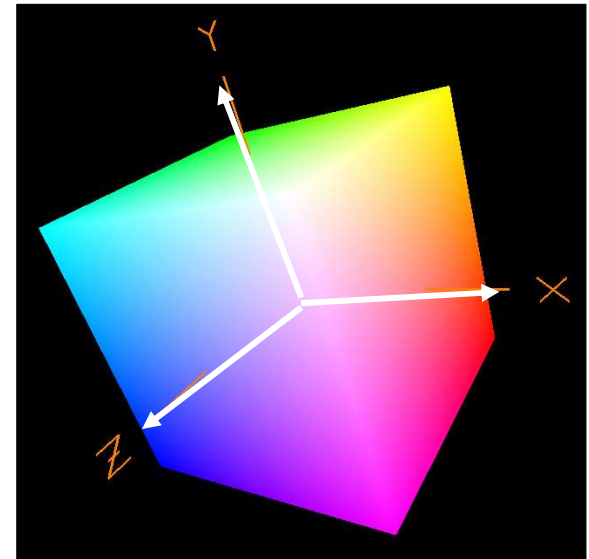
mjb@cs.oregonstate.edu



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)

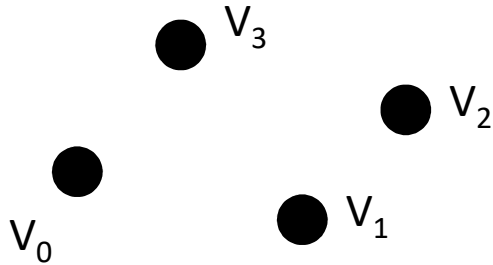


Oregon State University
Computer Graphics

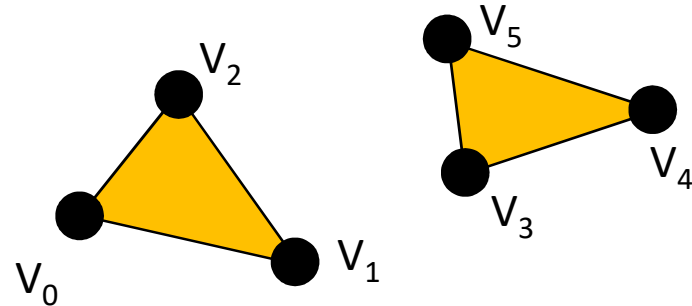


Vulkan Topologies

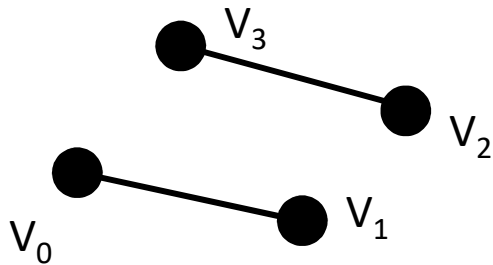
VK_PRIMITIVE_TOPOLOGY_POINT_LIST



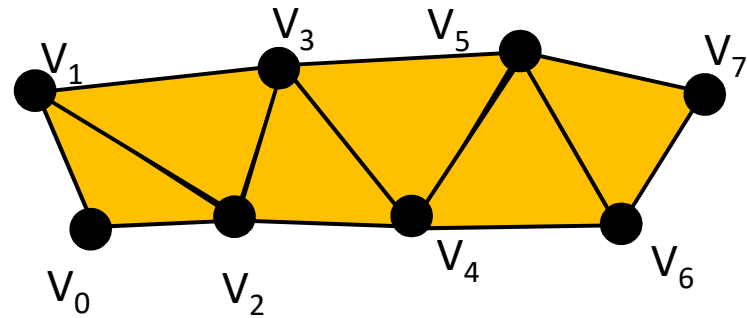
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST



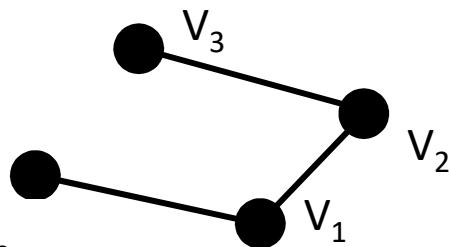
VK_PRIMITIVE_TOPOLOGY_LINE_LIST



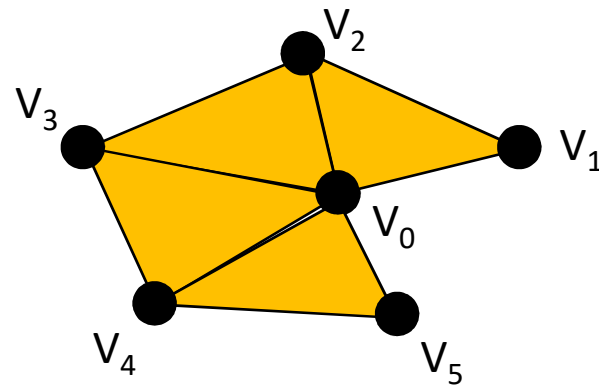
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP



VK_PRIMITIVE_TOPOLOGY_LINE_STRIP

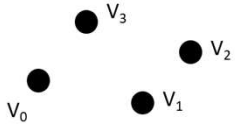


VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN

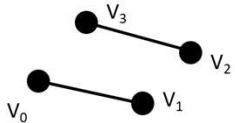


Vulkan Topologies

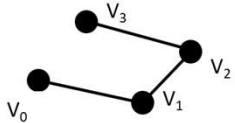
VK_PRIMITIVE_TOPOLOGY_POINT_LIST



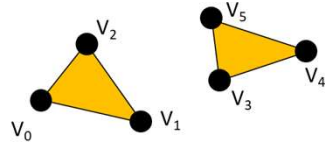
VK_PRIMITIVE_TOPOLOGY_LINE_LIST



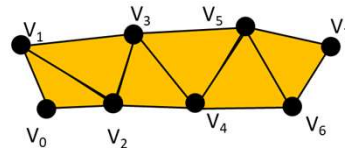
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP



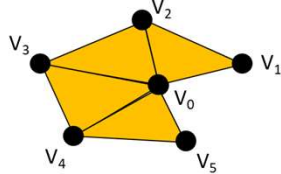
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST



VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP



VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN

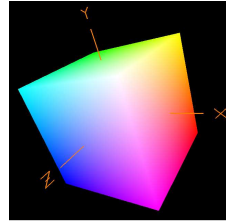
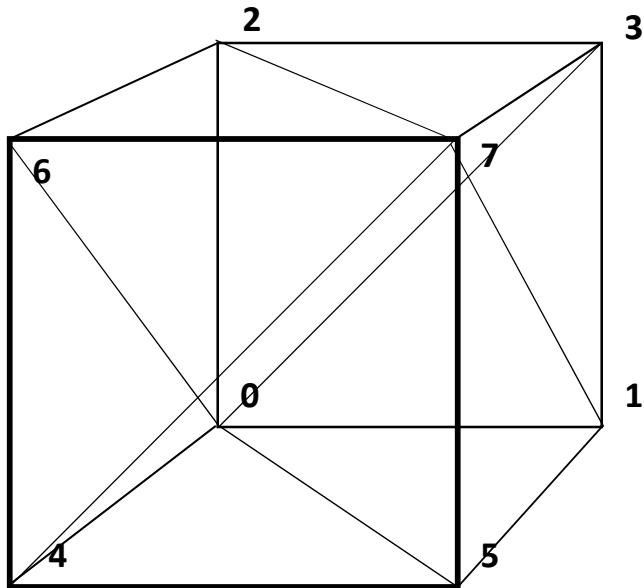


The same as OpenGL
topologies, with a few left out.

```
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



```
static GLfloat CubeColors [][3] =
{
    { 0., 0., 0. },
    { 1., 0., 0. },
    { 0., 1., 0. },
    { 1., 1., 0. },
    { 0., 0., 1. },
    { 1., 0., 1. },
    { 0., 1., 1. },
    { 1., 1., 1. },
};
```

```
static GLfloat CubeVertices [][3] =
{
    { -1., -1., -1. },
    { 1., -1., -1. },
    { -1., 1., -1. },
    { 1., 1., -1. },
    { -1., -1., 1. },
    { 1., -1., 1. },
    { -1., 1., 1. },
    { 1., 1., 1. },
};
```

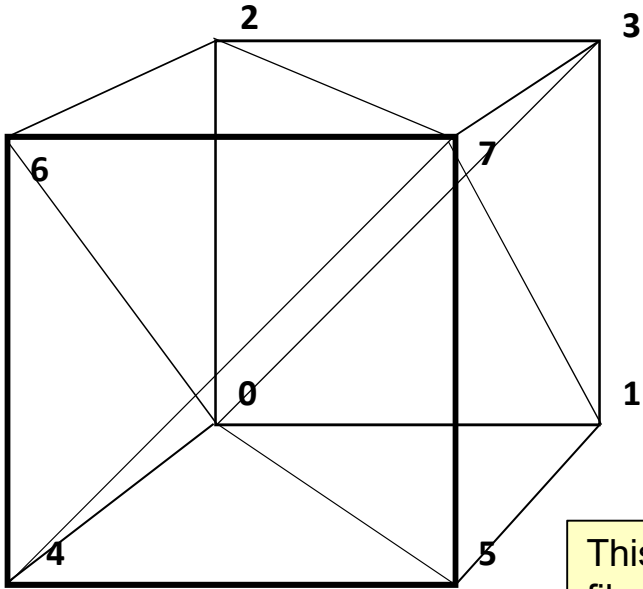
```
static GLuint CubeTriangleIndices [][3] =
{
    { 0, 2, 3 },
    { 0, 3, 1 },
    { 4, 5, 7 },
    { 4, 7, 6 },
    { 1, 3, 7 },
    { 1, 7, 5 },
    { 0, 4, 6 },
    { 0, 6, 2 },
    { 2, 6, 7 },
    { 2, 7, 3 },
    { 0, 1, 5 },
    { 0, 5, 4 }
};
```



Triangles Represented as an Array of Structures

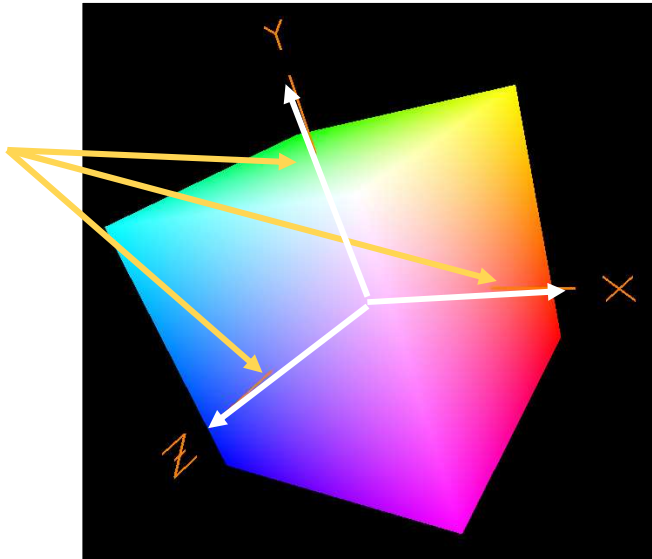
```
struct vertex
{
    glm::vec3    position;
    glm::vec3    normal;
    glm::vec3    color;
    glm::vec2    texCoord;
};

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. }
    },
};
```



This data is contained in the file *SampleVertexData.cpp*

Modeled in right-handed coordinates



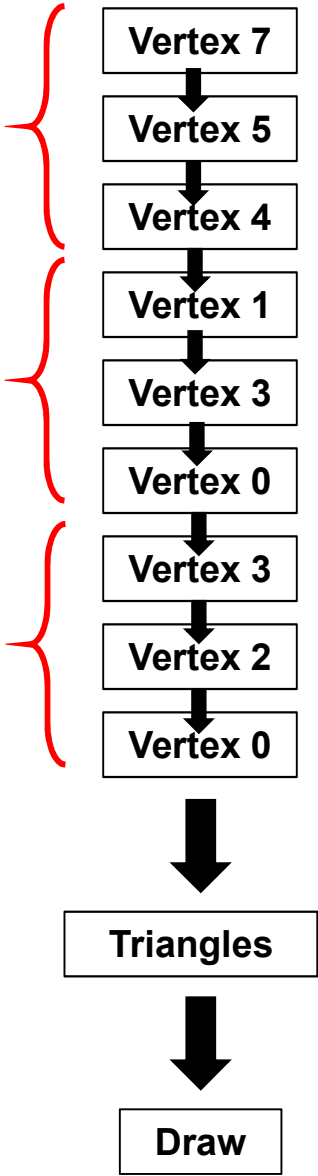
Non-indexed Buffer Drawing

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



Initializing and Filling the Vertex Buffer

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

MyBuffer MyVertexDataBuffer;

...

Init05MyVertexDataBuffer( sizeof(VertexData), OUT &MyVertexDataBuffer ); // create
Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData ); // fill

...

VkResult
Init05MyVertexDataBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result;
    result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}

VkResult
Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    ...
}
```

A Preview of What *Init05DataBuffer* Does

```
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
    VkResult result = VK_SUCCESS;
    VkBufferCreateInfo vbc;
    vbc.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbc.pNext = nullptr;
    vbc.flags = 0;
    vbc.size = pMyBuffer->size = size;
    vbc.usage = usage;
    vbc.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbc.queueFamilyIndexCount = 0;
    vbc.pQueueFamilyIndices = (const uint32_t *)nullptr;
    result = vkCreateBuffer ( LogicalDevice, IN &vbc, PALLOCATOR, OUT &pMyBuffer->buffer );

    VkMemoryRequirements vmr;
    vkGetBufferMemoryRequirements( LogicalDevice, IN pMyBuffer->buffer, OUT &vmr );    // fills 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, pMyBuffer->buffer, IN vdm, 0 );    // 0 is the offset
    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++:

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

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 aTexCoord;
```

```
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;    // read one value per vertex
```

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



Telling the Pipeline about its Input

```

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

```



```

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

```

```

VkVertexInputAttributeDescription  vviad[4];          // array per vertex input attribute
// 4 = vertex, normal, color, texture coord
vviad[0].location = 0;          // location in the layout decoration
vviad[0].binding = 0;          // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;    // x, y, z
vviad[0].offset = offsetof( struct vertex, position );          // 0

vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC3;    // nx, ny, nz
vviad[1].offset = offsetof( struct vertex, normal );          // 12

vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC3;    // r, g, b
vviad[2].offset = offsetof( struct vertex, color );          // 24

vviad[3].location = 3;
vviad[3].binding = 0;
vviad[3].format = VK_FORMAT_VEC2;    // s, t
vviad[3].offset = offsetof( struct vertex, texCoord );          // 36

```

Always use the C/C++ construct **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;    // 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;
```



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.

```
VkGraphicsPipelineCreateInfo    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 = vpsscsci;
    vgpci.pVertexInputState = &vpvisci;
    vgpci.pInputAssemblyState = &vpiasci;
    vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;    // &vptsci
    vgpci.pViewportState = &vpvsci;
    vgpci.pRasterizationState = &vprsci;
    vgpci.pMultisampleState = &vpmsci;
    vgpci.pDepthStencilState = &vpdsscsci;
    vgpci.pColorBlendState = &vpcbsci;
    vgpci.pDynamicState = &vpdscsci;
    vgpci.layout = IN GraphicsPipelineLayout;
    vgpci.renderPass = IN RenderPass;
    vgpci.subpass = 0;             // subpass number
    vgpci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
    vgpci.basePipelineIndex = 0;

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

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.

```
VkBuffer buffers[1] = { MyVertexBuffer.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] );
```

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

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



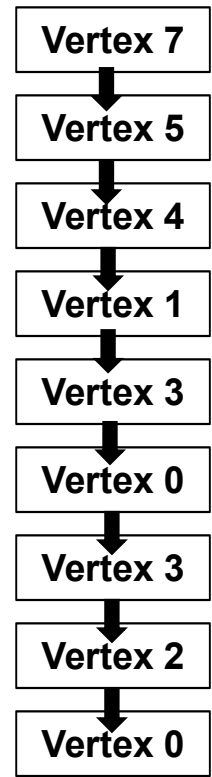
Drawing with an Index Buffer

```

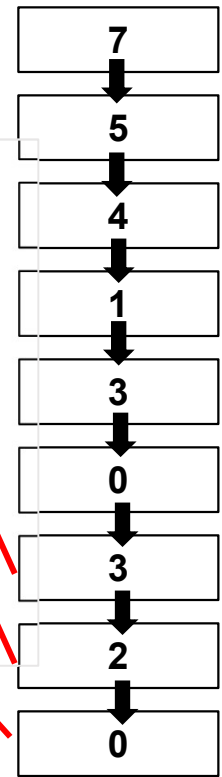
struct vertex JustVertexData[ ] =
{
    // vertex #0:
    {
        { -1., -1., -1. },
        { 0., 0., -1. },
        { 0., 0., 0. },
        { 1., 0. }
    },
    // vertex #1:
    {
        { 1., -1., -1. },
        { 0., 0., -1. },
        { 1., 0., 0. },
        { 0., 0. }
    },
    ...
};

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

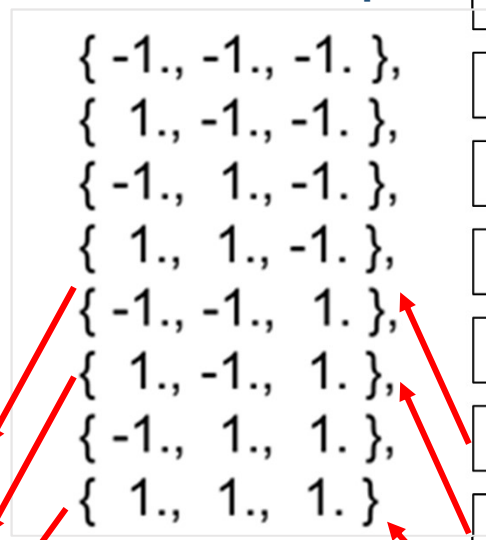
Stream of Vertices



Stream of Indices



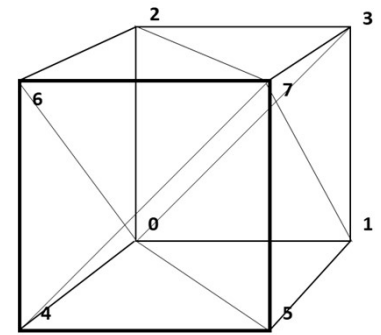
Vertex Lookup



This data is contained in the file **SampleVertexData.cpp**

Triangles

Draw



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;
```

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



```
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 );
```




```
VkBuffer vBuffers[1] = { MyJustVertexBuffer.buffer };
VkBuffer iBuffer      = { MyJustIndexDataBuffer.buffer };

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 firstInstance = 0;
const uint32_t vertexOffset = 0;

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



Indirect Drawing (not to be confused with *Indexed*)

```
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

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

Compare this with:

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



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

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

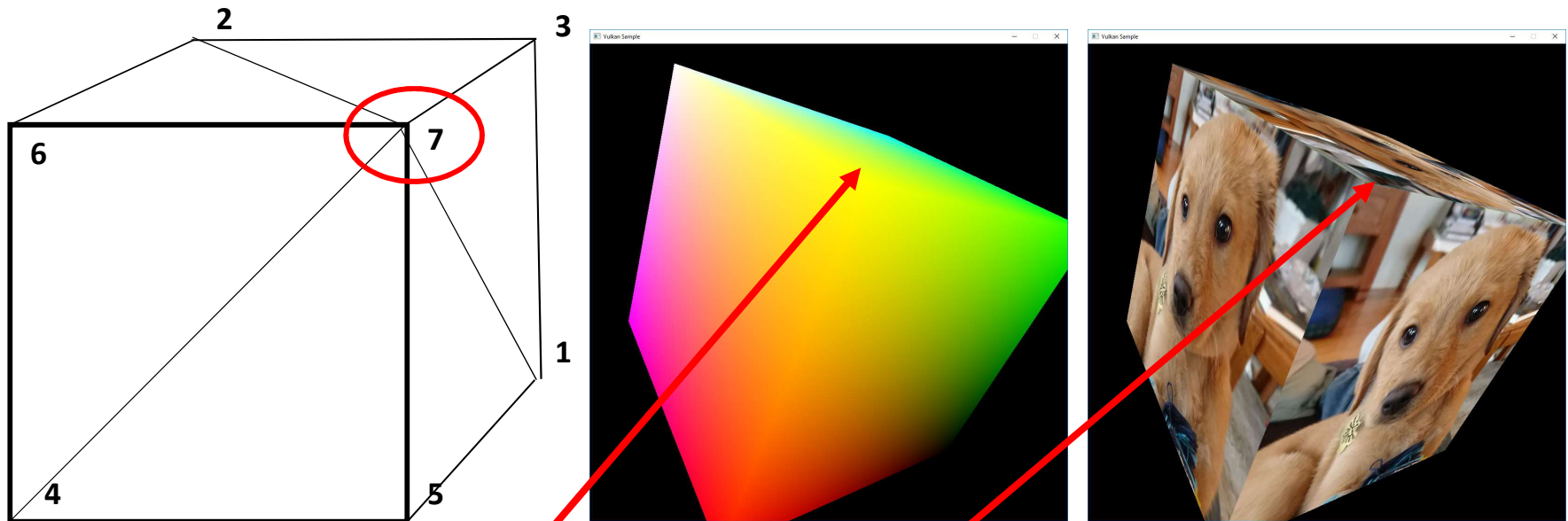
In Vulkan, "Indirect" means that you store the arguments in GPU memory and then give the vkCmdXxx call a pointer to those arguments

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

Compare this with:

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

Sometimes the Same Vertex Needs Multiple Attributes

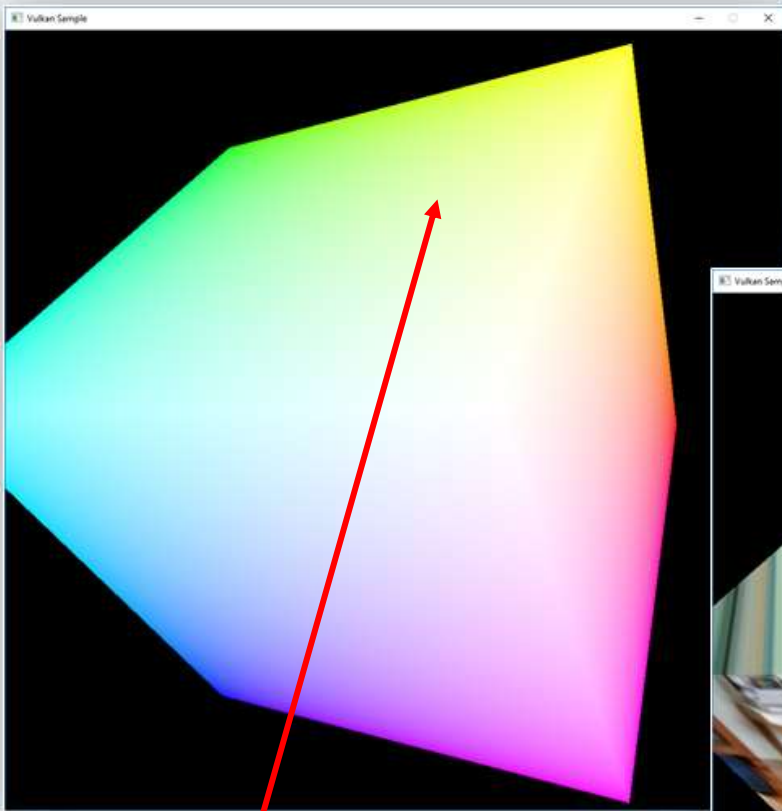


Sometimes a vertex that is common to multiple faces has the same attributes, no matter what face it is in. Sometimes it doesn't.

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.

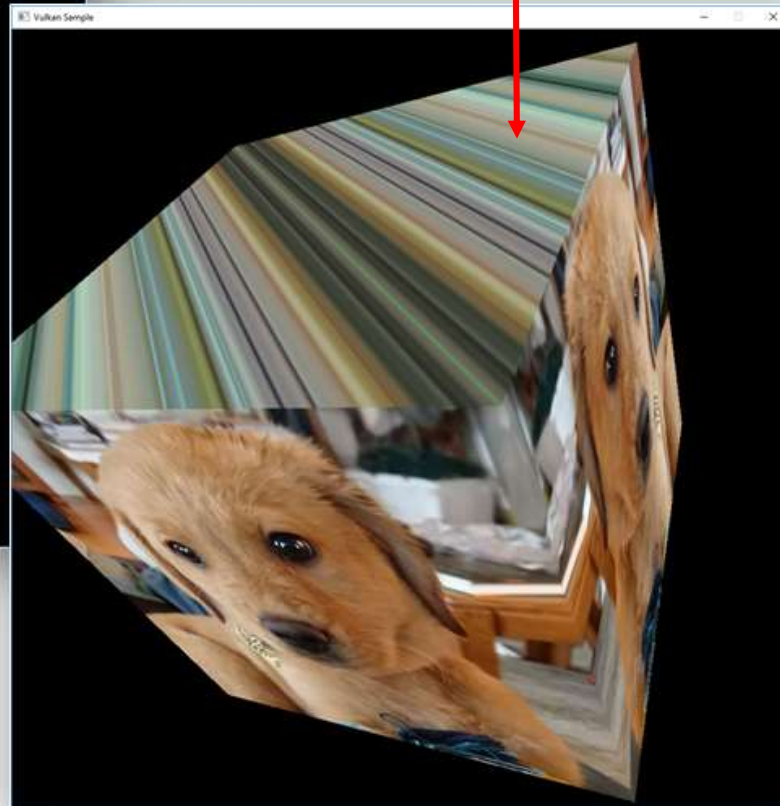
Thus, when using indexed buffer drawing, you need to create a new vertex struct if any of {position, normal, color, texCoords} changes from what was previously-stored at those coordinates.

Sometimes the Same Point Needs Multiple Attributes



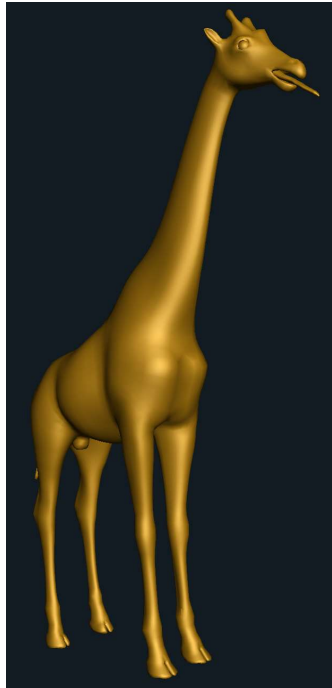
Where values match at the corners (color)

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



0

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



```
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
```

...

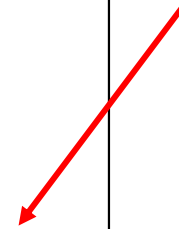
```
vn 0.1725 0.2557 -0.9512
vn -0.1979 -0.1899 -0.9616
vn -0.2050 -0.2127 -0.9554
vn 0.1664 0.3020 -0.9387
vn -0.2040 -0.1718 -0.9638
vn 0.1645 0.3203 -0.9329
vn -0.2055 -0.1698 -0.9638
vn 0.4419 0.6436 -0.6249
vn 0.4573 0.5682 -0.6841
vn 0.5160 0.5538 -0.6535
vn 0.1791 0.2082 -0.9616
vn -0.2167 -0.2250 -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
```

...

```
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 76/76/78 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
f 75/75/77 81/81/83 80/80/82
```

V / T / N



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

Drawing an OBJ Object

23

```
typedef struct MyBuffer
{
    VkDataBuffer    buffer;
    VkDeviceMemory  vdm;
    VkDeviceSize    size;    // in bytes
} MyBuffer;
```

```
MyBuffer MyObjBuffer;    // global
```

```
...
```

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

```
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;
```

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

```
VkBuffer    objBuffer[1] = { MyObjBuffer.buffer };
VkDeviceSize offsets[1] = { 0 };
vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, objBuffer, offsets );
```

```
const uint32_t firstInstance = 0;
const uint32_t firstVertex = 0;
const uint32_t instanceCount = 1;
const uint32_t vertexCount = MyObjBuffer.size / sizeof( struct vertex );
```

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

