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

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

Triangles Represented as an Array of Structures

Non-indexed Buffer Drawing

From the file SampleVertexData.cpp:

Modeled in right-handed coordinates
Filling the Vertex Buffer

C++:
```cpp
MyBuffer MyVertexDataBuffer;
Init05MyVertexBuffer( IN VkDeviceSize size, OUT MyBuffer * pMyBuffer )
{
    VkResult result = Init05DataBuffer( size, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, pMyBuffer );
    return result;
}
```

GLSL Shader:
```glsl
struct vertex
{
    glm::vec3 aVertex;
    glm::vec3 aNormal;
    glm::vec3 aColor;
    glm::vec2 texCoord;
};
vertex[0].binding = 0; // which binding description this is part of
vertex[0].location = 0; // location in the layout decoration
vertex[2].binding = 0; // which binding description this is part of
vertex[2].location = 1; // location in the layout decoration
vertex[3].binding = 0; // which binding description this is part of
vertex[3].location = 3; // location in the layout decoration
```

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.

```cpp
Telling the Pipeline about its Input

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;
```

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.

```cpp
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.

```cpp
```
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.

Telling the Command Buffer what Vertices to Draw

We will want to specify the vertex buffer that we want to draw.

struct vertex
{
  // 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,
};

Vertex 0
Vertex 1
Vertex 2
Vertex 3
Vertex 4
Vertex 5
Vertex 6
Vertex 7
Vertex 8

Triangles

Repeat for Indexed???

Copyright © 2016-2018, NVIDIA. All Rights Reserved.

Copyright © 2016-2018, NVIDIA. All Rights Reserved.

Drawing with an Indexed 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);

vkCmdDrawIndirect(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, 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), &MyJustVertexDataBuffer );
Fill05DataBuffer( MyJustVertexDataBuffer,               (void *) JustVertexData );
Init05MyIndexDataBuffer(  sizeof(JustIndexData), &MyJustIndexDataBuffer );
Fill05DataBuffer( MyJustIndexDataBuffer,                (void *) JustIndexData );

vBuffers[1] = { MyJustVertexDataBuffer.buffer };
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;

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

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

Copyright © 2016-2018, NVIDIA. All Rights Reserved.

Copyright © 2016-2018, NVIDIA. All Rights Reserved.

Indirect Drawing (not to be confused with Indexed)

typedef struct
{
  VkDrawIndirectCommand
  unk02 : vertexCount;
  unk03 : instanceCount;
  unk04 : firstVertex;
  unk05 : firstInstance;
} VkDrawIndirectCommand;

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)

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

```
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 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.

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

```
x 1 11064 1.203801 -0.946980
x 1 11459 1.270242 -0.840668
x 1 115814 1.296239 -0.947796
x 1 119653 1.277325 -0.941195
x 1 1220786 1.267156 -0.945309
x 1 1377196 1.271283 -0.947795
x 1 1370681 1.261184 -0.942830
x 1 1175113 1.269278 -0.357523
x 1 114513 1.269284 -0.027651
x 1 1197485 1.235693 -0.946489
x 1 150169 1.265481 -0.298862
x 1 117262 1.265481 -0.298875
```

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

```
x 0.1726  0.5097 -0.9512
x 0.1756 -0.9986 -0.2816
x 0.2036 -0.2197 -0.5854
x 0.0984  0.8202 -0.0367
x 0.2645 -0.7182 -0.9368
x 0.2035 -0.1666 -0.9386
x 0.4416  0.6466 -0.2049
x 0.4575  0.9956 -0.4841
x 0.5160  0.0350 -0.9506
x 0.7975  0.2809 -0.8816
x 0.2971 -0.2850 -0.9486
x 0.6804  0.8651 -0.2867
```

Where values match at the corners (color)

```
x 0.1726  0.5097 -0.9512
x 0.1756 -0.9986 -0.2816
x 0.2036 -0.2197 -0.5854
x 0.0984  0.8202 -0.0367
x 0.2645 -0.7182 -0.9368
x 0.2035 -0.1666 -0.9386
x 0.4416  0.6466 -0.2049
x 0.4575  0.9956 -0.4841
x 0.5160  0.0350 -0.9506
x 0.7975  0.2809 -0.8816
x 0.2971 -0.2850 -0.9486
x 0.6804  0.8651 -0.2867
```

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