Vertex Buffers

What is a Vertex Buffer?

Vertex Buffers are how you draw things in Vulkan. They are very much like Vertex Buffer Objects in OpenGL, but more detail is exposed to you (a lot more…).

But, the good news is that Vertex Buffers are really just ordinary Data Buffers, so some of the functions will look familiar to you.

First, a quick review of computer graphics geometry . . .
Geometry vs. Topology

Geometry: Where things are (e.g., coordinates)

Topology: How things are connected

Original Object

Geometry = changed
Topology = same (1-2-3-4-1)

Geometry = same
Topology = changed (1-2-4-3-1)

Vulkan Topologies

typedef enum VkPrimitiveTopology
{
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;}
Vulkan Topologies – Some OpenGL Topologies are Missing

OpenGL Topologies – Polygon Requirements

Polygons must be:

- **Convex** and
- **Planar**
Vulkan Topologies – Requirements and Orientation

Polygons must be:
• Convex and
• Planar

Polygons are traditionally:
• CCW when viewed from outside the solid object

OpenGL Topologies – Vertex Order Matters
What does “Convex Polygon” Mean?

We can go all mathematical here, but let’s go visual instead. In a convex polygon, a line between any two points inside the polygon never leaves the inside of the polygon.

Why is there a Requirement for Polygons to be Convex?

Graphics polygon-filling hardware can be highly optimized if you know that, no matter what direction you fill the polygon in, there will be two and only two intersections between the scanline and the polygon’s edges.
What if you need to display Polygons that are not Convex?

There is an open source library to break a non-convex polygon into convex polygons. It is called *Polypartition*, and is found here:

https://github.com/ivanfratric/polypartition

If you ever need to do this, contact me. I have working code …

Why is there a Requirement for Polygons to be Planar?

Graphics hardware assumes that a polygon has a definite front and a definite back, and that you can only see one of them at a time.
**Vertex Orientation Issues**

Thanks to OpenGL, we are all used to drawing in a right-handed coordinate system.

Internally, however, the Vulkan pipeline uses a left-handed system:

The best way to handle this is to continue to draw in a RH coordinate system and then fix it up in the projection matrix, like this:

\[ \text{ProjectionMatrix}[1][1] = -1; \]

This is like saying \( Y' = -Y \).

**A Colored Cube Example**

```cpp
static GLuint CubeTriangleIndices[6] = {
    { 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 }
};
```

```cpp
static GLfloat CubeColors[8] = {
    { 0.0, 0.0 },
    { 1.0, 0.0 },
    { 0.0, 1.0 },
    { 1.0, 1.0 },
    { 0.0, 0.1 },
    { 1.0, 0.1 },
    { 0.0, 1.1 },
    { 1.0, 1.1 },
};
```

```cpp
static GLfloat CubeVertices[8] = {
    {-1, -1, -1},
    { 1, -1, -1},
    { 1, 1, -1},
    {-1, 1, -1},
    {-1, -1, 1},
    { 1, -1, 1},
    {-1, 1, 1},
    { 1, 1, 1},
};
```
Triangles in 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[] =
{
    // 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. }
    }
};
```

Modeled in right-handed coordinates

Vertex Orientation Issues

This object was modeled such that triangles that face the viewer will look like their vertices are oriented CCW (this is detected by looking at vertex orientation at the start of the rasterization).

Because this 3D object is closed, Vulkan can save rendering time by not even bothering with triangles whose vertices look like they are oriented CW. This is called backface culling.

Vulkan's change in coordinate systems can mess up the backface culling.

So I recommend, at least at first, that you do no culling.

```cpp
VkPipelineRasterizationStateCreateInfo vprsci;

vprsci.cullMode = VK_CULL_MODE_NONE;

vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```
MyBuffer MyVertexDataBuffer;

Init05MyVertexDataBuffer( sizeof(VertexData), &MyVertexDataBuffer );
Fill05DataBuffer( MyVertexDataBuffer, (void *) VertexData );

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

Filling the Vertex Buffer

What Init05DataBuffer Does
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

```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;
```

C/C++

```c
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 input.

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.

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.
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
VkGraphicsPipelineCreateInfo vgpci;
v pci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
v pci.pNext = nullptr;
v pci.flags = 0;
v pci.stageCount = 2;  // number of shader stages in this pipeline
v pci.pStages = vpssci;
v pci.pVertexInputState = &vpvisci;
v pci.pInputAssemblyState = &vpiasci;
v pci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
v pci.pViewportState = &vpvsci;
v pci.pRasterizationState = &vprsci;
v pci.pMultisampleState = &vpmsci;
v pci.pDepthStencilState = &vpdssci;
v pci.pColorBlendState = &vpcbsci;
v pci.layout = IN GraphicsPipelineLayout;
v pci.renderPass = IN RenderPass;
v pci.subpass = 0;  // subpass number
v pci.basePipelineHandle = (VkPipeline) VK_NULL_HANDLE;
v pci.basePipelineIndex = 0;
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &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, buffers, 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 );
```

Better to do this than to hard-code a number