Vertex Buffers

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

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 . . .

What is a Vertex Buffer?

Vulkan Topologies

Polygons must be:
• Convex and
• Planar

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

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

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

Geometry vs. Topology

Where things are (e.g., coordinates)

How things are connected

Vulkan Topologies – Requirements and Orientation

It's not absolutely necessary, but there are possible optimizations if you are consistent

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
- 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

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
- 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

Vulkan Topologies – Requirements and Orientation

Polygons must be:
• Convex and
• Planar

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

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
- 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
OpenGL Topologies – Vertex Order Matters

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP  VK_PRIMITIVE_TOPOLOGY_LINE_STRIP

V_0  V_1  V_2  V_3

What does “Convex Polygon” Mean?

We could 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.

Convex  Not Convex

OK, now let’s go all mathematical. In a convex polygon, every interior angle is between 0° and 180°.

Convex  Not Convex

Between 0° and 180°  Greater than 180°

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.

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.

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.

OK  Not OK
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 GLM projection matrix, like this:

```
ProjectionMatrix[1][1] *= -1.;
```

This is like saying "Y' = -Y".

-----

A Colored Cube Example

### Triangles in an Array of Structures

```cpp
class vertex
{
public:
  glm::vec3 position;
  glm::vec3 normal;
  glm::vec3 color;
  glm::vec2 texCoord;
private:
  int id;
};
```

```cpp
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}
};
```

### Filling the Vertex Buffer

```cpp
MyBuffer MyVertexDataBuffer;
Init05MyVertexDataBuffer(sizeof(vertexData), &MyVertexDataBuffer);
Fill05DataBuffer(MyVertexDataBuffer, (void*)vertexData);
```

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

### A Reminder of What Init05DataBuffer Does

```cpp
//Init05DataBuffer(): VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pBuffer)
//VkResult = VK_SUCCESS;
//VkBufferCreateInfo vbci; vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
//vkCreateBuffer(LogicalDevice, &vbci, PALLOCATOR, &pBuffer->buffer);
//vkGetBufferMemoryRequirements(LogicalDevice, pBuffer->buffer, &vmr);
//VkMemoryAllocateInfo vmai; vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
//vkAllocateMemory(LogicalDevice, &vmai, PALLOCATOR, &pBuffer->vdm);
//vkBindBufferMemory(LogicalDevice, pBuffer->buffer, pBuffer->vdm, 0); // 0 is the offset
```
Telling the Pipeline Data Structure 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.

```
struct vertex
{
    glm::vec3 position;  // s
    glm::vec3 color;     // r, g, b
    glm::vec3 normal;    // nx, ny, nz
    glm::vec2 texCoord;  // s, t
}
```

### Vertex Input Stage

```
VkVertexInputStateCreateInfo vpvisci;           // used to describe the input vertex attributes
vpvisci.pVertexAttributeDescriptions = vvibd;   // array per vertex input attribute
vpvisci.pVertexBindingDescriptions = vvibd;     // array per vertex binding
vpvisci.vertexBindingDescriptionCount = 4;      // 4 = vertex, normal, color, texture coord
vpvisci.pNext = nullptr;
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
```

### Tesselation, Geometry Stage

```
vpvisci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;            // &vptsci
vpvisci.pVertexInputState = &vpvisci;
```

### Vertex Input attributes

```
vviad[0].offset = offsetof( struct vertex, position );                  // 0
vviad[0].binding = 0;                   // which binding description this is part of
vviad[0].location = 0;                  // location in the layout decoration
vviad[0].format = VK_FORMAT_VEC3;       // s
vviad[1].offset = offsetof( struct vertex, normal );                    // 12
vviad[1].binding = 0;                                           // which binding # this is
vviad[1].location = 1;                  // location in the layout decoration
vviad[1].format = VK_FORMAT_VEC3;       // nx, ny, nz
vviad[2].offset = offsetof( struct vertex, color );                  // 24
vviad[2].binding = 0;                                           // which binding # this is
vviad[2].location = 2;                  // location in the layout decoration
vviad[2].format = VK_FORMAT_VEC3;       // r, g, b
vviad[3].offset = offsetof( struct vertex, texCoord );                // 36
vviad[3].binding = 0;                                           // which binding # this is
vviad[3].location = 3;                  // location in the layout decoration
vviad[3].format = VK_FORMAT_VEC2;       // s, t
```

### Input Assembly

```
vpiasci.pNext = nullptr;
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
```

### Viewport

```
vgpci.pViewportState = &vpvsci;
vgpci.pNext = nullptr;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
```

### Topology

```
vgpci.pTopology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
```

### Fragment Shader Stage

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

### Color Blending Stage

```
result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN & vgpci, 0);
```

Don’t ever hardcode the size of an array! Always get the compiler to generate it for you.