Vertex Buffers

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

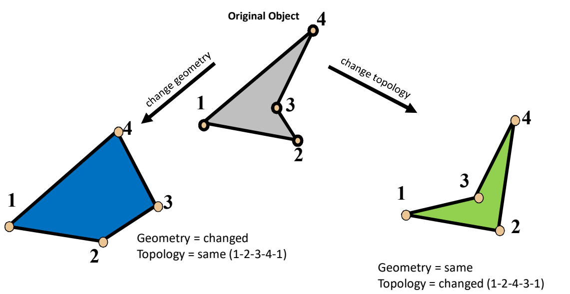
What is a Vertex Buffer?

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)

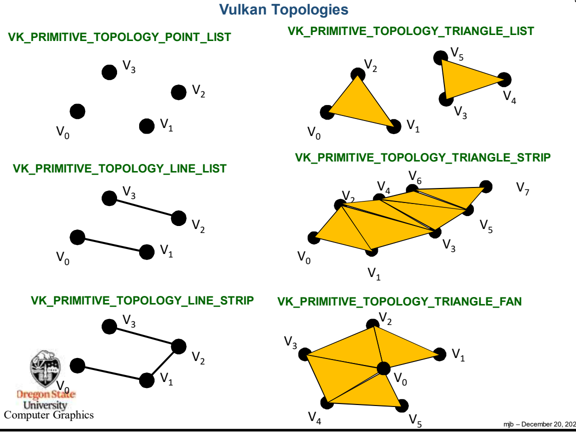
Topology:

How things are connected

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

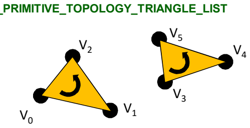


Vulkan Topologies – Requirements and Orientation

Polygons must be:
- **Convex** and
- **Planar**

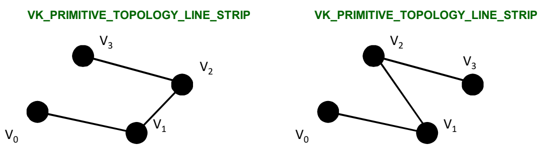
Polygons are traditionally:
- **CCW when viewed from outside the solid object**

VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST



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

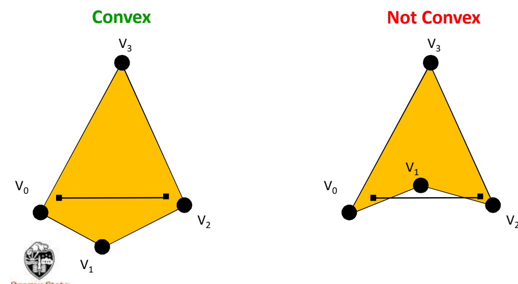
OpenGL Topologies – Vertex Order Matters



Oregon State University
Computer Graphics

What does "Convex Polygon" Mean?

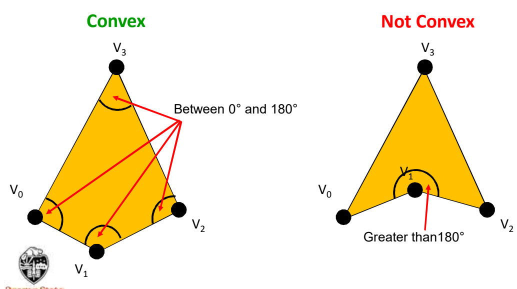
We could go all mathematical here, but let's go visual instead. In a convex polygon, a line between ***any*** two points inside the polygon never leaves the inside of the polygon.



Oregon State University
Computer Graphics

What does "Convex Polygon" Mean?

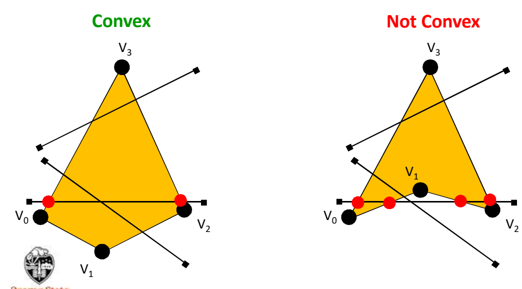
OK, now let's go all mathematical. In a convex polygon, every interior angle is between 0° and 180°.



Oregon State University
Computer Graphics

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

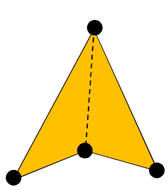


Oregon State University
Computer Graphics

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

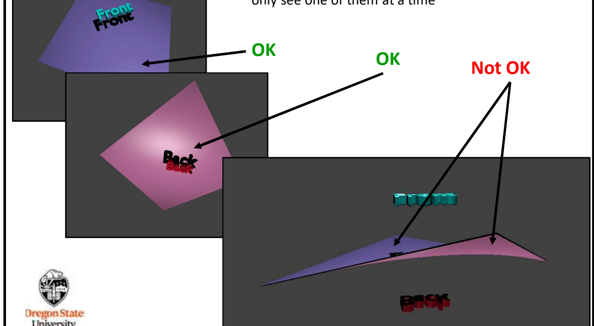


If you ever need to do this, contact me. I have working code …

Oregon State University
Computer Graphics

Why is there a Requirement for Polygons to be Planar?

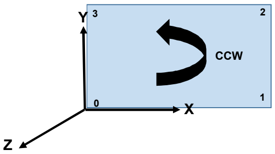
Graphics hardware assumes that a polygon has a definite front and a definite back, and that you can only see one of them at a time



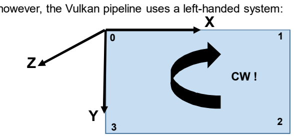
Oregon State University
Computer Graphics

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:

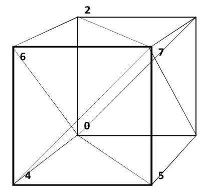


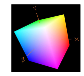
The best way to handle this is to continue to draw in a RH coordinate system and then fix it up in the GLM projection matrix, like this:

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

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

A Colored Cube Example





```
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. },
};
```

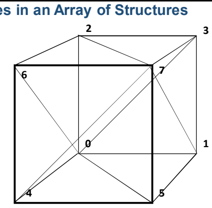
```
static GLfloat CubeVertices[ ][3] =
{
        { -1., -1., -1. },
        {  1., -1., -1. },
        { -1.,  1., -1. },
        {  1.,  1., -1. },
        { -1., -1.,  1. },
        {  1., -1.,  1. },
        { -1.,  1.,  1. },
        {  1.,  1.,  1. }
};
```

```
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 }
};
```

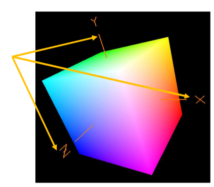

Triangles in an Array of Structures

From the file SampleVertexData.cpp:

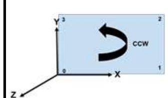
```
struct vertex
{
        glm::vec3     position;
        glm::vec3     normal;
        glm::vec3     color;
        glm::vec2     texCoord;
};

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. }
        },
```

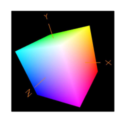


Modeled in right-handed coordinates



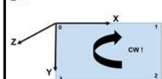
Vertex Orientation Issues





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***.

```
VkPipelineRasterizationStateCreateInfo          vprsci;
. . .
vprsci.cullMode = VK_CULL_MODE_NONE
vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

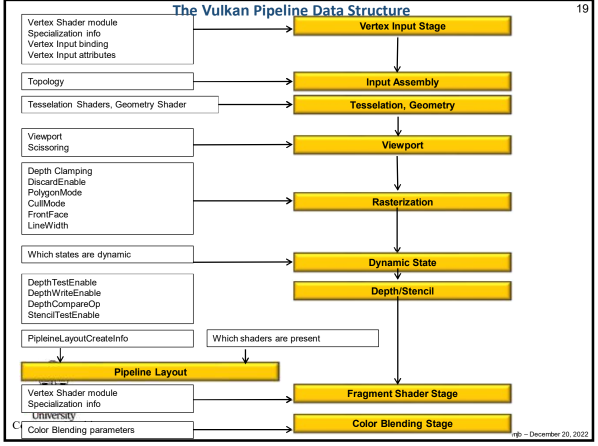

Filling the Vertex Buffer


```
MyBuffer        MyVertexDataBuffer;

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

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


```
VkResult
Init05DataBuffer( VkDeviceSize size, VkBufferUsageFlags usage, OUT MyBuffer * pMyBuffer )
{
        VkResult result = VK_SUCCESS;
        VkBufferCreateInfo  vbci;
                vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
                vbci.pNext = nullptr;
                vbci.flags = 0;
                vbci.size = pMyBuffer->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 );        // 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;
}
```

The Vulkan Pipeline Data Structure



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;
        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;
```

```
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;
```

Oregon State University Computer Graphics

Telling the Pipeline Data Structure 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
```

University Computer Graphics

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.

```
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;
```

Oregon State University Computer Graphics

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.

```
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 = vpssci;
        vgpci.pVertexInputState = &vpvisci;
        vgpci.pInputAssemblyState = &vpiasci;
        vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;        // &vptsci
        vgpci.pViewportState = &vpvsci;
        vgpci.pRasterizationState = &vprsci;
        vgpci.pMultisampleState = &vpmsci;
        vgpci.pDepthStencilState = &vpdssci;
        vgpci.pColorBlendState = &vpcbsci;
        vgpci.pDynamicState = &vpdsci;
        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 pGraphicsPipeline );
```

University Computer Graphics

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] = 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 );
```

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

~~const uint32_t vertexCount = 100;~~

Oregon State University Computer Graphics