The Vulkan Graphics Pipeline

Here’s what you need to know:

1. The Vulkan Graphics Pipeline is like what OpenGL would call “The State”, or “The Context”. It is a data structure.
2. The Vulkan Graphics Pipeline is not the processes that OpenGL would call “the graphics pipeline”.
3. For the most part, the Vulkan Graphics Pipeline Data Structure is immutable – that is, once this combination of state variables is combined into a Pipeline, that Pipeline never gets changed. To make new combinations of state variables, create a new Graphics Pipeline.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

The Graphics Pipeline Stages and what goes into them

The GPU and Driver specify the Pipeline Stages – the Vulkan Graphics Pipeline declares what goes in them.

The First Step: Create the Graphics Pipeline Layout

VkResult Init14GraphicsPipelineLayout() {
    VkResult result;
    VkPipelineLayoutCreateInfo vplci
        vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
        vplci.pNext = nullptr;
        vplci.flags = 0;
        vplci.setLayoutCount = 4;
        vplci.pSetLayouts = &DescriptorSetLayouts[0];
        vplci.pushConstantRangeCount = 0;
        vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;
    result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout);
    return result;
}

Let the Pipeline Layout know about the Descriptor Sets and Push Constants data structures have different sizes... of each you have. So, the exact structure of the Pipeline Layout depends on you telling Vulkan about the Descriptor Sets and Push Constants that you will be using.
A Pipeline Data Structure Contains the Following State Items:

- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissoring: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp, srcAlphaBlendFactor, dstAlphaBlendFactor, alphaBlendOp, colorWriteMask
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

**Bold/Italics** indicates that this state item can also be set with Dynamic State Variables

---

Creating a Typical Graphics Pipeline

```c
VkResult Init14GraphicsVertexFragmentPipeline(VkShaderModule vertexShader, VkShaderModule fragmentShader, OUT VkPipeline *pGraphicsPipeline)
{
    VkPipelineShaderStageCreateInfo vpssci[2];
    vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[0].pNext = nullptr;
    vpssci[0].flags = 0;
    vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
    vpssci[0].module = vertexShader;
    vpssci[0].pName = "main";
    vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;
    vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE; // an array containing one of these per buffer being used
    vvibd[0].binding = 0; // which binding # this is
    vvibd[0].stride = sizeof(struct vertex); // bytes between successive array members per vertex
    vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
    vpcbas.blendEnable = VK_FALSE;
    vpcbsci.logicOpEnable = VK_FALSE;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.blendOp = VK_BLEND_OP_ADD;
    vpcbsci.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
    vpcbsci.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
    vpcbsci.alphaBlendOp = VK_BLEND_OP_ADD;
    vpcbsci.colorWriteMask = VK_COLOR_WRITE_ALL;
    vpcbsci.blendEnable = VK_FALSE;
    vpcbsci.blendFactorSrcColor = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorSrcAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.blendFactorDstColor = VK_BLEND_FACTOR_ONE;
    vpcbsci.blendFactorDstAlpha = VK_BLEND_FACTOR_ZERO;
    vpcbsci.srcColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
    vpcbsci.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MIN...
**Link in the Per-Vertex Attributes**

VPVisci

vkPipelineVertexInputStateCreateInfo

vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

vpvisci.pNext = nullptr;

vpvisci.flags = 0;

VkVertexInputAttributeDescription

vpvisci.vertexBindingDescriptionCount = 1;

vpvisci.pVertexBindingDescriptions = vvibd;

vpvisci.vertexAttributeDescriptionCount = 4;

vpvisci.pVertexAttributeDescriptions = vviad;

Declare the binding descriptions and attribute descriptions

vkPipelineInputAssemblyStateCreateInfo

vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;

vpiasci.pNext = nullptr;

vpiasci.flags = 0;

vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;

These are defined at the top of the sample code so that you don't need to use confusing image-looking formats for positions, normals, and tex coords

What is “Primitive Restart Enable”?

vpiasci.primitiveRestartEnable = VK_FALSE;

“Restart Enable” is used with:

- Indexed drawing.
- Triangle Fan and “Strip” topologies

If vpiasci.primitiveRestartEnable is VK_TRUE, then a special “index” indicates that the primitive should start over. This is more efficient than explicitly ending the current primitive and explicitly starting a new primitive of the same type.

typedef enum VkIndexType{
    VK_INDEX_TYPE_UINT16 = 0, // 0 – 65,535
    VK_INDEX_TYPE_UINT32 = 1, // 0 – 4,294,967,295
} VkIndexType;

If your VkIndexType is VK_INDEX_TYPE_UINT16, then the special index is 0xffff. If your VkIndexType is VK_INDEX_TYPE_UINT32, it is 0xffffffff.
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

Triangle Strip #0:
Triangle Strip #1:
Triangle Strip #2:

What is the Difference Between Changing the Viewport and Changing the Scissoring?

Viewport:
Viewporting operates on vertices and takes place right before the rasterizer. Changing the vertical part of the viewport causes the entire scene to get scaled (srunched) into the viewport area.

Original Image

Scissoring:
Scissoring operates on fragments and takes place right after the rasterizer. Changing the vertical part of the scissor causes the entire scene to get clipped where it falls outside the scissor area.

Setting the Rasterizer State

VkPipelineRasterizationStateCreateInfo

VkPipelineViewportStateCreateInfo

Declare the viewport information

Declare the scissoring information

Group the viewport and scissor information together

Declare information about how the rasterization will take place
vprsci.depthClampEnable = VK_FALSE;

Depth Clamp Enable causes the fragments that would normally have been discarded because they are closer to the viewer than the near clipping plane to instead get projected to the near clipping plane and displayed. A good use for this is **Polygon Capping**: The front of the polygon is clipped, revealing to the viewer that this is really a shell, not a solid. The gray area shows what would happen with depthClampEnable (except it would have been red).

---

vprsci.depthBiasEnable = VK_FALSE;

Depth Bias Enable allows scaling and translation of the Z-depth values as they come through the rasterizer to avoid Z-fighting.

---

vice incorporates the multisampling states into a VkPipelineMultisampleStateCreateInfo struct.

```c
VkPipelineMultisampleStateCreateInfo vpmsci = {
    .sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .rasterizationSamples = VK_SAMPLE_COUNT_1_BIT,
    .sampleShadingEnable = VK_FALSE,
    .minSampleShading = 0,
    .pSampleMask = (VkSampleMask *)nullptr,
    .alphaToCoverageEnable = VK_FALSE,
    .alphaToOneEnable = VK_FALSE,
};
```

Declare information about how the multisampling will take place. **MultiSampling State**

We will discuss MultiSampling in a separate noteset.

---

vice incorporates the color blending states into a VkPipelineColorBlendAttachmentState struct.

```c
VkPipelineColorBlendAttachmentState vpcbas = {
    .blendEnable = VK_FALSE,
    .srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR,
    .dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR,
    .colorBlendOp = VK_BLEND_OP_ADD,
    .srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE,
    .dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO,
    .alphaBlendOp = VK_BLEND_OP_ADD,
    .colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT,
};
```

Color Blending State for each Color Attachment *

Create an array with one of these for each color buffer attachment. Each color buffer attachment can use different blending operations. **Color Blending State**

`Color_{new} = (1 - \alpha) \times \text{Color}_{existing} + \alpha \times \text{Color}_{incoming}`

\[0 \leq \alpha \leq 1\]

* A "Color Attachment" is a framebuffer to be rendered into. You can have as many of these as you want.
**Raster Operations for each Color Attachment**

```c
VkPipelineColorBlendStateCreateInfo vpcbsci;
vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
vpcbsci.pNext = nullptr;
vpcbsci.flags = 0;
vpcbsci.logicOpEnable = VK_FALSE;
vpcbsci.logicOp = VK_LOGIC_OP_COPY;
#ifdef CHOICES
VK_LOGIC_OP_AND
VK_LOGIC_OP_AND_INVERTED
VK_LOGIC_OP_COPY
VK_LOGIC_OP_COPY_INVERTED
VK_LOGIC_OP_INVERT
VK_LOGIC_OP_NAND
VK_LOGIC_OP_NAND
#endif
vpcbsci.attachmentCount = 1;
vpcbsci.pAttachments = &vpcbas;
vpcbsci.blendConstants[0] = 0;
vpcbsci.blendConstants[1] = 0;
vpcbsci.blendConstants[2] = 0;
vpcbsci.blendConstants[3] = 0;
```

This controls blending between the output of the fragment shader and the input to the color attachments.

---

**Which Pipeline Variables can be Set Dynamically**

```c
VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };
#ifdef CHOICES
VK_DYNAMIC_STATE_VIEWPORT       -- vkCmdSetViewport( )
VK_DYNAMIC_STATE_SCISSOR        -- vkCmdSetScissor( )
VK_DYNAMIC_STATE_LINE_WIDTH     -- vkCmdSetLineWidth( )
VK_DYNAMIC_STATE_DEPTH_BIAS     -- vkCmdSetDepthBias( )
VK_DYNAMIC_STATE_BLEND_CONSTANTS        -- vkCmdSetBendConstants( )
VK_DYNAMIC_STATE_DEPTH_BOUNDS   -- vkCmdSetDepthZBounds( )
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK  -- vkCmdSetStencilCompareMask( )
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK     -- vkCmdSetStencilWriteMask( )
VK_DYNAMIC_STATE_STENCIL_REFERENCE     -- vkCmdSetStencilReference( )
#endif
```

---

**The Stencil Buffer**

Here’s how the Stencil Buffer works:

1. While drawing into the Render Buffer, you can write values into the Stencil Buffer at the same time.
2. While drawing into the Render Buffer, you can do arithmetic on values in the Stencil Buffer at the same time.
3. When drawing into the Render Buffer, you can write-protect certain parts of the Render Buffer based on values that are in the Stencil Buffer.
Using the Stencil Buffer to Create a Magic Lens

1. Clear the SB = 0
2. Write protect the color buffer
3. Fill a square, setting SB = 1
4. Write-enable the color buffer
5. Draw the solids wherever SB == 0
6. Draw the wireframes wherever SB == 1

Using the Stencil Buffer to Perform Polygon Capping

1. Clear the SB = 0
2. Draw the polygons, setting SB = ~ SB
3. Draw a large gray polygon across the entire scene wherever SB != 0

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges

Z-fighting
Clear the SB = 0
for( each polygon )
{
  Draw the edges, setting SB = 1
  Draw the polygon wherever SB != 1
  Draw the edges, setting SB = 0
}

using the Stencil Buffer to better outline polygons

before

after

Using the Stencil Buffer to Perform Hidden Line Removal

Stencil Operations for Front and Back Faces

VkStencilOpState
vsosf;
// front
vsosf.depthFailOp = VK_STENCIL_OP_KEEP; // what to do if depth operation fails
vsosf.failOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation fails
vsosf.passOp = VK_STENCIL_OP_KEEP; // what to do if stencil operation succeeds

vsosf.compareOp = VK_COMPARE_OP_NEVER;

vsosf.compareMask = ~0;
vsosf.writeMask = ~0;
vsosf.reference = 0;

VkStencilOpState
vsosb;
// back
vsosb.depthFailOp = VK_STENCIL_OP_KEEP;
vsosb.failOp = VK_STENCIL_OP_KEEP;
vsosb.passOp = VK_STENCIL_OP_KEEP;
vsosb.compareOp = VK_COMPARE_OP_NEVER;
vsosb.compareMask = ~0;
vsosb.writeMask = ~0;
vsosb.reference = 0;
VkPipelineDepthStencilStateCreateInfo

vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.flags = 0;
vpdssci.depthTestEnable = VK_TRUE;
vpdssci.depthWriteEnable = VK_TRUE;
vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
vk_COMPARE_OP_NEVER -- never succeeds
vk_COMPARE_OP_LESS -- succeeds if new depth value is < the existing value
vk_COMPARE_OP_EQUAL -- succeeds if new depth value is == the existing value
vk_COMPARE_OP_LESS_OR_EQUAL -- succeeds if new depth value is <= the existing value
vk_COMPARE_OP_GREATER -- succeeds if new depth value is > the existing value
vk_COMPARE_OP_NOT_EQUAL -- succeeds if new depth value is != the existing value
vk_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if new depth value is >= the existing value
vk_COMPARE_OP_ALWAYS -- always succeeds
#endif
vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.;
vpdssci.maxDepthBounds = 1.;
vpdssci.stencilTestEnable = VK_FALSE;

Operations for Depth Values

Later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);

Putting it all Together! (finally…)

Later on, we will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer when Drawing

vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);

Sidebar: What is the Organization of the Pipeline Data Structure?

If you take a close look at the pipeline data structure creation information, you will see that almost all the pieces have a fixed size. For example, the viewport only needs 6 pieces of information—over:

- vV.viewport
  - vV.x = 0;
  - vV.y = 0;
  - vV.width = (float)Width;
  - vV.height = (float)Height;
  - vV.minDepth = 0.0f;
  - vV.maxDepth = 1.0f;

There are two exceptions to this—the Descriptor Sets and the Push Constants. Each of these two can be almost any size, depending on what you allocate for them. So, I think of the Pipeline Data Structure as consisting of some fixed-layout blocks and 2 variable-layout blocks, like this:

Fixed-layout Pipeline Blocks

Variable-layout Pipeline Blocks