What is 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 is meant to be 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 Pipelines.
4. The shaders get compiled the rest of the way when their Graphics Pipeline gets created.

Don’t worry if this is too small to read – a larger version is coming up.

There is also a Vulkan Compute Pipeline – we will get to that later.

The First Step: Create the Graphics Pipeline Layout

The Graphics Pipeline Layout is fairly static. Only the layout of the Descriptor Sets and information on the Push Constants need to be supplied.

```c
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;
}
```
Vulkan: A Pipeline Records the Following Items:

- Pipeline Layout: DescriptorSets, PushConstants
- Which Shaders are going to be used
- 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, 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 Variables

---

Creating a Typical Graphics Pipeline

```c
#def ASSUMPTIONS
vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[0].pNext = nullptr;
vpssci[0].flags = 0;

vkCreateGraphicsPipeline( )
```

These settings seem pretty typical to me. Let’s write a simplified Pipeline creator that accepts Vertex and Fragment shader modules and the topology, and always uses the settings in red above.

---

Creating a Graphics Pipeline from a lot of Pieces

[Diagram of a graphics pipeline]

---

Link in the Shaders

```c
#def ASSUMPTIONS
vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[0].pNext = nullptr;
vpssci[0].flags = 0;

vkCreateGraphicsPipeline( )
```

Use one `vpssci` array member per shader module you are using

---

[Diagram of a graphics pipeline]

---

Use one `vvibd` array member per vertex input array-of-structures you are using
Link in the Per-Vertex Attributes

Use one vviad array member per element in the struct for the array-of-structures element you are using as vertex input.

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”?

“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.

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 “Depth Clamp Enable”? 

```c
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:

![Polygon Capping](image)

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).

What is “Depth Bias Enable”? 

```c
vprsci.depthBiasEnable = VK_FALSE;
vprsci.depthBiasConstantFactor = 0.f;
vprsci.depthBiasClamp = 0.f;
vprsci.depthBiasSlopeFactor = 0.f;
```

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

MultiSampling State 

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

Declare information about how the multisampling will take place

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.

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

This controls blending between the output of each color attachment and its image memory.
Color Blending State for each Color Attachment

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

Which Pipeline Variables can be Set Dynamically

Uses for Stencil Operations

Magic Lenses

Polygon edges without Z-fighting
VkPipelineDepthStencilStateCreateInfo
- sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO
- pNext = nullptr
- flags = 0
- depthTestEnable = VK_TRUE
- depthWriteEnable = VK_TRUE
- depthCompareOp = VK_COMPARE_OP_LESS
- depthBoundsTestEnable = VK_FALSE
- front = vsosf
- back = vsosb
- minDepthBounds = 0.
- maxDepthBounds = 1.
- stencilTestEnable = VK_FALSE

Operations for Depth Values

VkGraphicsPipelineCreateInfo
- sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO
- pNext = nullptr
- flags = 0
- stageCount = 2
- pStages = vpssci
- pVertexInputState = &vpvisci
- pInputAssemblyState = &vpiasci
- pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr
- pViewportState = &vpvsci
- pRasterizationState = &vprsci
- pMultisampleState = &vpmsci
- pDepthStencilState = &vpdssci
- pColorBlendState = &vpcbsci
- pDynamicState = &vpdsci
- layout = IN GraphicsPipelineLayout
- renderPass = IN RenderPass
- subpass = 0
- basePipelineHandle = (VkPipeline) VK_NULL_HANDLE
- basePipelineIndex = 0

result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci, PALLOCATOR, OUT pGraphicsPipeline);

Later on, we will Bind the Graphics Pipeline to the Command Buffer when Drawing

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