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”.
2. There’s a lot that goes into it.
3. For the most part, the 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 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.

Graphics Pipeline Stages and what goes into Them

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.

VkResult
Init14GraphicsPipelineLayout() {
    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 Set and Push Constant layouts.
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
- Scissors: x, y, w, h
- Rasterization: cullMode, polygonMode, frontFace, lineWidth
- Depth: depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp
- 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

```cpp
VkResult Init14GraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader, VkPipeline vertexPipeline, VkPipeline fragmentPipeline, OUT VkPipeline *pGraphicsPipeline )

VkPipelineShaderStageCreateInfo vpssci[0];
vpssci[0].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
vpssci[0].module = (VkShaderModule *)&vertexShader;
vpssci[0].pName = "main";
vpssci[0].pNext = nullptr;

VkPipelineShaderStageCreateInfo vpssci[1];
vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
vpssci[1].module = (VkShaderModule *)&fragmentShader;
vpssci[1].pName = "main";
vpssci[1].pNext = nullptr;

VkPipelineColorBlendAttachmentState vpcbas[2];
vpcbas[0].blendEnable = VK_FALSE;
vpcbas[0].colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
vpcbas[0].alphaBlendEnable = VK_FALSE;
vpcbas[0].blendOp = VK_BLEND_OP_ADD;
vpcbas[0].srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
vpcbas[0].dstColorBlendFactor = VK_BLEND_FACTOR_DST_ALPHA;
vpcbas[0].colorBlendOp = VK_BLEND_OP_ADD;
vpcbas[1].blendEnable = VK_FALSE;
vpcbas[1].colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
vpcbas[1].alphaBlendEnable = VK_FALSE;
vpcbas[1].blendOp = VK_BLEND_OP_ADD;
vpcbas[1].srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
vpcbas[1].dstColorBlendFactor = VK_BLEND_FACTOR_DST_ALPHA;
vpcbas[1].colorBlendOp = VK_BLEND_OP_ADD;

VkPipelineColorBlendStateCreateInfo vpccbsci;
vpccbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
vpccbsci.blendEnable = VK_FALSE;
vpccbsi
```

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

Creating a Graphics Pipeline from a lot of Pieces

1. Create Pipeline Layout:
   - ShaderStage: 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

2. Create Pipeline State:
   - Viewport: x, y, w, h, minDepth, maxDepth
   - Scissors: x, y, w, h
   - Rasterization: cullMode, polygonMode, frontFace, lineWidth
   - Depth: depthTestEnable, depthWriteEnable, depthCompareOp
   - Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
   - Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, colorBlendOp

3. Create Pipeline Objects:
   - ShaderStage: pipelineShaders
   - Topology: vertexInputAttributes, vertexInputLayouts, vertexInputBindingDescriptions
   - Assembly: vertexInputBindingDescriptions

4. Chain Pipeline State:
   - DescriptorSets: setLayouts, setCounts, setInters
   - PushConstants
   - PushBuffers

5. Create Pipeline:
   - CreatePipelineLayout(): pipelineLayout
   - CreateGraphicsPipeline(): pipeline

Usage:

```cpp
pPipeline = createGraphicsPipeline( pipelineLayout, shaderStages, topology, vertexInputLayouts, vertexInputBindingDescriptions, descriptorSets, pushConstants, pushBuffers )
```
Link in the Per-Vertex Attributes

`VkPipelineVertexInputStateCreateInfo` is used to describe the input vertex attributes.

```c
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;
```

Use one `vpvisci` array member per element attribute descriptions.

![Diagram of vertex input state](image)

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.

<table>
<thead>
<tr>
<th>Primitive Topology</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_PRIMITIVE_TOPOLOGY_POINT_LIST</td>
<td>• Indexed drawing</td>
<td></td>
</tr>
<tr>
<td>VK_PRIMITIVE_TOPOLOGY_LINE_LIST</td>
<td>• Triangle Fan and “Strip” topologies</td>
<td></td>
</tr>
<tr>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is “Primitive Restart Enable”? 

“Restart Enable” is used with:
- Indexed drawing.
- Triangle Fan and “Strip” topologies.

If `vpvisci.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.

```c
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`.

When using the primitive restart code, the easy way to do it is like this:

```c
short int restartIndex = ~0;
```

 Declare the binding descriptions and attribute descriptions

 Declare the vertex topology

 Tessellation Shader info

 Geometry Shader info
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

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

What is “Depth Clamp Enable”?

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.

What is “Depth Bias Enable”?

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

Declare information about how the multisampling will take place

MultiSampling State

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.

vprsci.colorBlendEnable = VK_FALSE;
vprsci.colorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
vprsci.colorBlendOp = VK_BLEND_OP_ADD;
vprsci.colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;

vpcbas.colorBlendEnable = VK_FALSE;

This controls blending between the output of each color attachment and its image memory.

Color Blending State for each Color Attachment
**Color Blending State for each Color Attachment**

```
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;
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?**

```
vkDynamicState vn[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR, ...
```

These are the pipeline variables that can be set dynamically.

**Stencil Operations for Front and Back Faces**

```
VkStencilOpState vsosf;  // front
vsosf.depthFailOp = VK_STENCIL_OP_KEEP;
vsosf.failOp = VK_STENCIL_OP_KEEP;
vsosf.passOp = VK_STENCIL_OP_KEEP;
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;
```

**Uses for Stencil Operations**

- **Magic Lenses**
- Polygon edges without Z-fighting
Operations for Depth Values

VkPipelineDepthStencilStateCreateInfo vpdssci;
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;

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

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