The Vulkan Graphics Pipeline

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.

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() {
    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, colorBlendOp, srcAlphaBlendFactor, dstAlphaBlendFactor, alphaBlendOp, colorWriteMask
- DynamicState: which states can be set dynamically (bound to the command buffer, outside the Pipeline)

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

Creating a Typical Graphics Pipeline

```c
VkResult InitGraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader, VkPipeline *pGraphicsPipeline )
{
    VkPipelineCreateInfo pipelineInfo;
    memset( &pipelineInfo, 0, sizeof( pipelineInfo ) );
    pipelineInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_CREATE_INFO;
    pipelineInfo.layout = descriptorSet;
    pipelineInfo.basePipelineHandle = nullptr;
    pipelineInfo.basePipelineIndex = 0;
    pipelineInfo.pStages = &vpssci[0], &vpssci[1];
    pipelineInfo.stageCount = 2;
    pipelineInfo.pDynamicStates = nullptr;
    pipelineInfo.pDynamicStateCount = 0;

    vpssci[0].module = vertexShader;
    vpssci[0].pName = "main";
    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].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    vpssci[1].module = fragmentShader;
    vpssci[1].pName = "main";
    vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vpssci[1].pNext = nullptr;
    vpssci[1].flags = 0;
    vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

    pipelineInfo.pVertexInputState = &vprsci;
    pipelineInfo.pInputAssemblies = &vpmsci;
    pipelineInfo.pViewportState = &vpsci;
    pipelineInfo.pRasterizationState = &vprsci;
    pipelineInfo.pDepthStencilState = &vpscs;
    pipelineInfo.pColorBlendState = &vpcbs;
    pipelineInfo.pDynamicState = &vpdssci;

    VkPipelineVertexInputStateCreateInfo vertexInputStateCreateInfo;
    memset( &vertexInputStateCreateInfo, 0, sizeof( vertexInputStateCreateInfo ) );
    vertexInputStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

    VkPipelineInputAssemblyStateCreateInfo inputAssemblyStateCreateInfo;
    memset( &inputAssemblyStateCreateInfo, 0, sizeof( inputAssemblyStateCreateInfo ) );
    inputAssemblyStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;

    VkPipelineRasterizationStateCreateInfo rasterizationStateCreateInfo;
    memset( &rasterizationStateCreateInfo, 0, sizeof( rasterizationStateCreateInfo ) );
    rasterizationStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;

    VkPipelineDepthStencilStateCreateInfo depthStencilStateCreateInfo;
    memset( &depthStencilStateCreateInfo, 0, sizeof( depthStencilStateCreateInfo ) );
    depthStencilStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;

    VkPipelineColorBlendStateCreateInfo colorBlendStateCreateInfo;
    memset( &colorBlendStateCreateInfo, 0, sizeof( colorBlendStateCreateInfo ) );
    colorBlendStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;

    VkPipelineDynamicStateCreateInfo dynamicStateCreateInfo;
    memset( &dynamicStateCreateInfo, 0, sizeof( dynamicStateCreateInfo ) );
    dynamicStateCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;

    VkResult result = vkCreateGraphicsPipeline( device, VK_NULL_HANDLE, &pipelineCreateInfo, nullptr, pGraphicsPipeline );
    return result;
}
```

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.

Link in the Shaders

```c
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].pName = "main";
vpssci[0].module = vertexShader;
vpssci[0].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

vpssci[1].module = fragmentShader;
vpssci[1].pName = "main";
vpssci[1].sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vpssci[1].pNext = nullptr;
vpssci[1].flags = 0;
vpssci[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

VkPipelineInputViewportStateCreateInfo vpivsci;
vpivsci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_VIEWPORT_STATE_CREATE_INFO;
vpi scisi.offset = 0;
vpivsci.extent = viewSize;
vpi scisi.viewportCount = viewCount;
vpivsci.pix2d = true;
vpivsci.viewportIndex = 0;
vpivsci.pViewports = &vpiv;

VkPipelineColorBlendAttachmentState vpcbas;
vpcbas.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_ATTACHMENT_STATE_CREATE_INFO;
vpcbas.colorWriteMask = (1 << red) | (1 << green) | (1 << blue) | (1 << alpha);
vpcbas.blendEnable = VK_FALSE;
vpcbas.srcColorBlendFactor = VK_BLEND_FACTOR_ONE;
vpcbas.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO;
vpcbas.colorBlendOp = VK_BLEND_OP_ADD;
vpcbas.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
vpcbas.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
vpcbas.alphaBlendOp = VK_BLEND_OP_ADD;

VkPipelineColorBlendStateCreateInfo vpcbsci;
vpcbsci.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
vpcbsci.pAttachments = &vpcbas;

VkPipelineDynamicStateCreateInfo vpdssci;
vpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdssci.pNext = nullptr;
vpdssci.defaultSet = (vkShaderModule)0;
vpdssci.dynamicStateCount = 0;
vpdssci.pDynamicStates = nullptr;
```

Use one vpssci array member per shader module you are using

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

Options for vpiasci.topology

What is “Primitive Restart Enable”?
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

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

 VkViewport
vv
vv.x = 0; vv.y = 0;
vw.width = (float)Width;
vw.height = (float)Height;
nv.minDepth = 0.0f;
nv.maxDepth = 1.0f;

 VkRect2D vr
vr.offset.x = 0; vr.offset.y = 0;
vr.extent.width = Width;
vr.extent.height = Height;

VkPipelineViewportStateCreateInfo vpvsci
vpvsci.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
vpvsci.pNext = nullptr;
vpvsci.flags = 0;
vpvsci.viewportCount = 1;
vpvsci.pViewports = &vv;
vpvsci.scissorCount = 1;
vpvsci.pScissors = &vr;

Declare the viewport information

Declare the scissoring information

Group the viewport and scissor information together

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

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 (squished) into the viewport area.

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 vprsci
vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;

vkPipelineViewportStateCreateInfo
vkPipelineRasterizationStateCreateInfo
vkPipelineViewportAndScissorCreateInfo

Declare information about how the rasterization will take place
What is “Depth Clamp Enable”?

```cpp
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).

What is “Depth Bias Enable”?

```cpp
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**

```cpp
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.

**Color Blending State for each Color Attachment**

```cpp
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;
```

Create an array with one of these for each color buffer attachment. Each color buffer attachment can use different blending operations.
### VkPipelineColorBlendStateCreateInfo

- `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.

### VkPipelineDynamicStateCreateInfo

- `vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;`
- `vpdsci.pNext = nullptr;`
- `vpdsci.flags = 0;`
- `vpdsci.dynamicStateCount = 0;`
- `vpdsci.pDynamicStates = vds;`

### VkStencilOpState

- `stencilEnable = false;`
- `depthFailOp = VK_STENCIL_OP_KEEP;`
- `failOp = VK_STENCIL_OP_KEEP;`
- `passOp = VK_STENCIL_OP_KEEP;`
- `compareOp = VK_COMPARE_OP_NEVER;`
- `compareMask = ~0;`
- `writeMask = ~0;`
- `reference = 0;`

### Magic Lenses

- Uses for Stencil Operations
- Polygon edges without Z-fighting

- Magic Lenses
- Polygon edges without Z-fighting
vkCmdBindPipeline(CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipeline);