The Graphics Pipeline Data Structure (GPDS)
What is the Vulkan Graphics Pipeline Data Structure (GPDS)?

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. Since you know the OpenGL state, a lot of the Vulkan GPDS will seem familiar to you.

3. The current shader program is part of the state. (It was in OpenGL too, we just didn’t make a big deal of it.)

4. The Vulkan Graphics Pipeline is not the processes that OpenGL would call “the graphics pipeline”.

5. 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 GPDS.

6. The shaders get compiled the rest of the way when their Graphics Pipeline Data Structure gets created.

There are also a Vulkan Compute Pipeline Data Structure and a Raytrace Pipeline Data Structure – we will get to those later.
Vulkan 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

- Vertex Shader module
  - Specialization info
  - Vertex Input binding
  - Vertex Input attributes

- Topology

- Tessellation Shaders, Geometry Shader

- Viewport
  - Scissoring

- Depth Clamping
  - DiscardEnable
  - PolygonMode
  - CullMode
  - FrontFace
  - LineWidth

- Which states are dynamic

- DepthTestEnable
- DepthWriteEnable
- DepthCompareOp
- StencilTestEnable

- Fragment Shader module
  - Specialization info

- University

- Color Blending parameters
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
VkPipelineLayout GraphicsPipelineLayout; // global

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;
}
```

Why is this necessary? It is because the Descriptor Sets and Push Constants data structures have different sizes depending on how many 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 Graphics Pipeline Data Structure Contains the Following State Items:

- Pipeline Layout: Descriptor Sets, Push Constants
- Which Shaders to use (half-compiled SPIR-V modlukes)
- Per-vertex input attributes: location, binding, format, offset
- Per-vertex input bindings: binding, stride, inputRate
- Assembly: topology (e.g., `VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`)
- **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 be changed with Dynamic State Variables
Creating a Graphics Pipeline from a lot of Pieces

```cpp
vkCreateGraphicsPipeline( )
```

Array naming the states that can be set dynamically
Creating a Typical Graphics Pipeline

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.
The Shaders to Use

Use one \texttt{vpssci} array member per shader module you are using

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

`VkVertexInputAttributeDescription vviad[4];`  // an array containing one of these per vertex attribute in all bindings

// 4 = vertex, normal, color, texture coord
vviad[0].location = 0;  // location in the layout
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

#ifdef EXTRAS_DEFINED_AT_THE_TOP
// these are here for convenience and readability:
#define VK_FORMAT_VEC4 VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_XYZW VK_FORMAT_R32G32B32A32_SFLOAT
#define VK_FORMAT_VEC3 VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_STP VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_XYZ VK_FORMAT_R32G32B32_SFLOAT
#define VK_FORMAT_VEC2 VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_ST VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_XY VK_FORMAT_R32G32_SFLOAT
#define VK_FORMAT_FLOAT VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_S VK_FORMAT_R32_SFLOAT
#define VK_FORMAT_X VK_FORMAT_R32_SFLOAT
#endif

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

I #defined these 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

Use one `vviad` array member per element in the struct for the array-of-structures element you are using as vertex input
VkPipelineVertexInputStateCreateInfo *vpvisci = nullptr; // 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 = nullptr;
vpiasci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
vpiasci.pNext = nullptr;
vpiasci.flags = 0;
vpiasci.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
#ifdef CHOICES
VK_PRIMITIVE_TOPOLOGY_POINT_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_LIST
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
#endif
vpiasci.primitiveRestartEnable = VK_FALSE;

VkPipelineTessellationStateCreateInfo *vptsci = nullptr;
vptsci.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
vptsci.pNext = nullptr;
vptsci.flags = 0;
vptsci.patchControlPoints = 0; // number of patch control points

VkPipelineGeometryStateCreateInfo *vpgsci = nullptr;
vpgsci.sType = VK_STRUCTURE_TYPE_PIPELINE_GEOMETRY_STATE_CREATE_INFO;
vpgsci.pNext = nullptr;
vpgsci.flags = 0;
Options for vpiasci.topology

VK_PRIMITIVE_TOPOLOGY_POINT_LIST

 VK_PRIMITIVE_TOPOLOGY_LINE_LIST

 VK_PRIMITIVE_TOPOLOGY_LINE_STRIP

 VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST

 VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP

 VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
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.

```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, then the special index is **0xffffffff**.

That is, a one in all available bits.
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.x = 0;
    vv.y = 0;
    vv.width = (float)Width;
    vv.height = (float)Height;
    vv.minDepth = 0.0f;
    vv.maxDepth = 1.0f;

VkRect2D
    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?

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 (scrunch) 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.
VkPipelineRasterizationStateCreateInfo vprsci;
    vprsci.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
    vprsci.pNext = nullptr;
    vprsci.flags = 0;
    vprsci.depthClampEnable = VK_FALSE;
    vprsci.rasterizerDiscardEnable = VK_FALSE;
    vprsci.polygonMode = VK_POLYGON_MODE_FILL;
    #ifdef CHOICES
    VK_POLYGON_MODE_FILL
    VK_POLYGON_MODE_LINE
    VK_POLYGON_MODE_POINT
    #endif
    vprsci.cullMode = VK_CULL_MODE_NONE;    // recommend this because of the projMatrix[1][1] *= -1.;
    #ifdef CHOICES
    VK_CULL_MODE_NONE
    VK_CULL_MODE_FRONT_BIT
    VK_CULL_MODE_BACK_BIT
    VK_CULL_MODE_FRONT_AND_BACK_BIT
    #endif
    vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
    #ifdef CHOICES
    VK_FRONT_FACE_COUNTER_CLOCKWISE
    VK_FRONT_FACE_CLOCKWISE
    #endif
    vprsci.depthBiasEnable = VK_FALSE;
    vprsci.depthBiasConstantFactor = 0.f;
    vprsci.depthBiasClamp = 0.f;
    vprsci.depthBiasSlopeFactor = 0.f;
    vprsci.lineWidth = 1.f;

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

```
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;
vprsci.depthBiasConstantFactor = 0.f;
vprsci.depthBiasClamp = 0.f;
vprsci.depthBiasSlopeFactor = 0.f;

What is “Depth Bias Enable”? 

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

VkPipelineMultisampleStateCreateInfo
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

We will discuss MultiSampling in a separate noteset.
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.

```
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.

\[
    \text{Color}_{\text{new}} = (1. - \alpha) \times \text{Color}_{\text{existing}} + \alpha \times \text{Color}_{\text{incoming}}
\]

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

*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

```cpp
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;
#endif CHOICES
VK_LOGIC_OP_CLEAR
VK_LOGIC_OP_AND
VK_LOGIC_OP_AND_REVERSE
VK_LOGIC_OP_COPY
VK_LOGIC_OP_AND_INVERTED
VK_LOGIC_OP_NO_OP
VK_LOGIC_OP_XOR
VK_LOGIC_OP_OR
VK_LOGIC_OP_NOR
VK_LOGIC_OP_EQUIVALENT
VK_LOGIC_OP_INVERT
VK_LOGIC_OP_OR_REVERSE
VK_LOGIC_OP_COPY_INVERTED
VK_LOGIC_OP_OR_INVERTED
VK_LOGIC_OP_NAND
VK_LOGIC_OP_SET
#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

Just used as an example in the Sample Code

```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     -- vkCmdSetStencilReferences( )
#endif

VkPipelineDynamicStateCreateInfo vpdsici;
vpdsici.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
vpdsici.pNext = nullptr;
vpdsici.flags = 0;
vpdsici.dynamicStateCount = 0;                   // leave turned off for now
vpdsici.pDynamicStates = vds;
```
The Stencil Buffer

Here’s what the Stencil Buffer can do for you:

1. While drawing into the Back Buffer, you can write values into the Stencil Buffer at the same time.

2. While drawing into the Back Buffer, you can do arithmetic on values in the Stencil Buffer at the same time.

3. The Stencil Buffer can be used to write-protect certain parts of the Back Buffer.
You Can Think of the Stencil Buffer as a Separate Framebuffer, or, You Can Think of it as being Per-Pixel. Both are correct, but I like thinking of it "per-pixel" better.
Using the Stencil Buffer to Create a *Magic Lens*
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
I Once Used the Stencil Buffer to Create a *Magic Lens* for Volume Data

In this case, the scene inside the lens was created by drawing the same object, but drawing it with its near clipping plane being farther away from the eye position.
Using the Stencil Buffer to Perform Polygon Capping
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
Using the Stencil Buffer to Better Outline Polygons
Using the Stencil Buffer to Better Outline Polygons

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
}

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
    #ifdef CHOICES
    VK_STENCIL_OP_KEEP -- keep the stencil value as it is
    VK_STENCIL_OP_ZERO -- set stencil value to 0
    VK_STENCIL_OP_REPLACE -- replace stencil value with the reference value
    VK_STENCIL_OP_INCREMENT_AND_CLAMP -- increment stencil value
    VK_STENCIL_OP_DECREMENT_AND_CLAMP -- decrement stencil value
    VK_STENCIL_OP_INVERT -- bit-invert stencil value
    VK_STENCIL_OP_INCREMENT_AND_WRAP -- increment stencil value
    VK_STENCIL_OP_DECREMENT_AND_WRAP -- decrement stencil value
    #endif
    vsosf.compareOp = VK_COMPARE_OP_NEVER;
    #ifdef CHOICES
    VK_COMPARE_OP_NEVER -- never succeeds
    VK_COMPARE_OP_LESS -- succeeds if stencil value is < the reference value
    VK_COMPARE_OP_EQUAL -- succeeds if stencil value is == the reference value
    VK_COMPARE_OP_LESS_OR_EQUAL -- succeeds if stencil value is <= the reference value
    VK_COMPARE_OP_GREATER -- succeeds if stencil value is > the reference value
    VK_COMPARE_OP_NOT_EQUAL -- succeeds if stencil value is != the reference value
    VK_COMPARE_OP_GREATER_OR_EQUAL -- succeeds if stencil value is >= the reference value
    VK_COMPARE_OP_ALWAYS -- always succeeds
    #endif
    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 = VKCOMPARE_OPLesser;
#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

VK_COMPARE_OPLIMITER
VK_COMPARE_OPLESS
VK_COMPARE_OPEQUAL
VK_COMPARE_OPLIMITEROREQUAL
VK_COMPARE_OPGREATER
VK_COMPARE_OPNOTEQUAL
VK_COMPARE_OPGREATEROREQUAL
VK_COMPARE_OPALWAYS

-- never succeeds
-- succeeds if new depth value is < the existing value
-- succeeds if new depth value is == the existing value
-- succeeds if new depth value is <= the existing value
-- succeeds if new depth value is > the existing value
-- succeeds if new depth value is != the existing value
-- succeeds if new depth value is >= the existing value
-- always succeeds
Group all of the individual state information and create the pipeline

```
VkPipeline GraphicsPipeline; // global

VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;
#ifdef CHOICES
VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif
vgpci.stageCount = 2;                           // number of stages in this pipeline
vgpci.pStages = vpssci;
vgpci.pVertexInputState = &vpvisci;
vgpci.pInputAssemblyState = &vpiasci;
vgpci.pTessellationState = (VkPipelineTessellationStateCreateInfo *)nullptr;
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 &GraphicsPipeline );

return result;
```
When Drawing, We will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer

```cpp
VkPipeline GraphicsPipeline; // global

... 

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 – ever:

```c
VkViewport vv;
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 Graphics Pipeline Data Structure as consisting of some fixed-layout blocks and 2 variable-layout blocks, like this: