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.
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

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() {
    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;

    VkResult 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 modules)
- **Per-vertex input attributes**: location, binding, format, offset
- **Per-vertex input bindings**: binding, stride, inputRate
- **Assembly** (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

![Diagram of creating a Graphics Pipeline]

- **Descriptor Set Layouts**
- **Push Constants**
- **Shaders**
- **InputAssembly State**
- **Multisample State**
- **ColorBlend State**
- **Dynamic State**
- **Viewport State**
- **Rasterization State**
- **DepthStencil State**
- **ColorBlend State**
- **Pipeline layout**
- **RenderPass**
- **Pipeline**
- **Graphics Pipeline**
- **Array naming the states that can be set dynamically**

---

12/29/2022
Creating a Typical Graphics Pipeline

```c
VkResult Init14GraphicsVertexFragmentPipeline( VkShaderModule vertexShader, VkShaderModule fragmentShader,
                                              VkPrimitiveTopology topology, OUT VkPipeline *pGraphicsPipeline )
{
    #ifdef ASSUMPTIONS
        vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
        vprsci.depthClampEnable = VK_FALSE;
        vprsci.rasterizerDiscardEnable = VK_FALSE;
        vprsci.polygonMode = VK_POLYGON_MODE_FILL;
        vprsci.cullMode = VK_CULL_MODE_NONE;    // best to do this because of the projectionMatrix[1][1] *= -1.;
        vprsci.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
        vprsci.rasterizationSamples = VK_SAMPLE_COUNT_ONE_BIT;
        vpcbas.blendEnable = VK_FALSE;
        vpcbsci.logicOpEnable = VK_FALSE;
        vpdssci.depthTestEnable = VK_TRUE;
        vpdssci.depthWriteEnable = VK_TRUE;
        vpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
    #endif
    ...

    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

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

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].module = fragmentShader;
vpssci[1].pName = "main";
vpssci[1].pSpecializationInfo = (VkSpecializationInfo *)nullptr;

VkVertexInputBindingDescription vvibd[1]; // an array containing one of these per buffer being used
vvibd[0].binding = 0;
vvibd[0].stride = sizeof(struct vertex);
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

```

The Shaders to Use

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

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

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

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

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

#defined EXTRAS_DEFINED_AT_THE_TOP

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; // 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;

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_TRIANGLE_LIST_WITH_ADJACENCY
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
#endif

vpiasci->primitiveRestartEnable = VK_FALSE;

VkPipelineTessellationStateCreateInfo *vptsci;

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;

vpgsci->sType = VK_STRUCTURE_TYPE_PIPELINE_GEOMETRY_STATE_CREATE_INFO;
vpgsci->pNext = nullptr;
vpgsci->flags = 0;

Declare the binding descriptions and attribute descriptions

Declare the vertex topology

Tessellation Shader info

Geometry Shader info
What is “Primitive Restart Enable”?  

```cpp
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:
...

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.

---

**Setting the Rasterizer State**

```cpp
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; // recommend this because of the projMatrix[1][1] = -1.;
#define 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.;
#define 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;
#define 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
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;

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;
```

We will discuss MultiSampling in a separate notest.

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.

```cpp
VkPipelineColorBlendAttachmentState vpchas;
vpchas.blendEnable = VK_FALSE;
vpchas.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
vpchas.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
vpchas.colorBlendOp = VK_BLEND_OP_ADD;
vpchas.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
vpchas.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
vpchas.alphaBlendOp = VK_BLEND_OP_ADD;
vpchas.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.

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

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

* "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**

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
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*

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

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
}
### Stencil Operations for Front and Back Faces

```cpp
// front
VkStencilOpState vsosf;
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;

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

def 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

def

vsosb.compareOp = VK_COMPARE_OP_NEVER;
vsosb.writeMask = ~0;
vsosb.reference = 0;
```

---

### Stencil Operations for Front and Back Faces

- **Front:**
  - Depth Fail Op: VK_STENCIL_OP_KEEP
  - Fail Op: VK_STENCIL_OP_KEEP
  - Pass Op: VK_STENCIL_OP_KEEP
  - Compare Op: VK_COMPARE_OP_NEVER

- **Back:**
  - Depth Fail Op: VK_STENCIL_OP_KEEP
  - Fail Op: VK_STENCIL_OP_KEEP
  - Pass Op: VK_STENCIL_OP_KEEP
  - Compare Op: VK_COMPARE_OP_NEVER

---

### Using the Stencil Buffer to Perform Hidden Line Removal

Stencil operations are used to determine whether a pixel should be drawn or not based on the stencil value. The stencil buffer is used to keep track of the faces that have already been drawn. This allows for the removal of hidden lines during rendering.
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

vpdssci.depthBoundsTestEnable = VK_FALSE;
vpdssci.front = vsosf;
vpdssci.back = vsosb;
vpdssci.minDepthBounds = 0.;
vpdssci.maxDepthBounds = 1.;
vpdssci.stencilTestEnable = VK_FALSE;

#endif

vkCreateGraphicsPipelines

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

Putting it all Together! (finally…)

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.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 );
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:

```cpp
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: