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 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 modules)
- 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
- Blending: blendEnable, srcAlphaBlendFactor, dstAlphaBlendFactor,
- Scissoring
- Viewport

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.

**Bold/Italics** indicates that this state item can be changed with Dynamic State Variables.
**Link in the Per-Vertex Attributes**

- **Options for vpilasci.topology**
  - VK_PRIMITIVE_TOPOLOGY_POINT_LIST
  - VK_PRIMITIVE_TOPOLOGY_LINE_LIST
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
  - VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
  - VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN_WITH_ADJACENCY

**What is "Primitive Restart Enable"?**

- `vpilasci.primitiveRestartEnable = VK_FALSE;`

  - "Restart Enable" is used with:
    - Indexed drawing.
    - TRIANGLE_FACE and *_STRIP topologies

If `vpilasci.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.

- `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 `0xffffffff`.
- If your `VkIndexType` is `VK_INDEX_TYPE_UINT32`, then the special index is `0xffffffff`.

**What is "Primitive Restart Enable"?**

- `vpilasci.primitiveRestartEnable = VK_FALSE;`

  - "Restart Enable" is used with:
    - Indexed drawing.
    - TRIANGLE_FACE and *_STRIP topologies

If `vpilasci.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.

- `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 `0xffffffff`.
- If your `VkIndexType` is `VK_INDEX_TYPE_UINT32`, then the special index is `0xffffffff`.

**What is "Primitive Restart Enable"?**

- `vpilasci.primitiveRestartEnable = VK_FALSE;`

  - "Restart Enable" is used with:
    - Indexed drawing.
    - TRIANGLE_FACE and *_STRIP topologies

If `vpilasci.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.

- `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 `0xffffffff`.
- If your `VkIndexType` is `VK_INDEX_TYPE_UINT32`, then the special index is `0xffffffff`.

**What is "Primitive Restart Enable"?**

- `vpilasci.primitiveRestartEnable = VK_FALSE;`

  - "Restart Enable" is used with:
    - Indexed drawing.
    - TRIANGLE_FACE and *_STRIP topologies

If `vpilasci.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.

- `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 `0xffffffff`.
- 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:
...
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;
```

We will discuss MultiSampling in a separate noteset.

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.

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;
// Set properties here...
```

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

**Which Pipeline Variables can be Set Dynamically**

```cpp
VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR, ...
```

Just used as an example in the Sample Code.

**The Stencil Buffer**

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

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

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

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
}

Outlining Polygons the Naïve Way

Using the Stencil Buffer to Better Outline Polygons

Before

After
Using the Stencil Buffer to Perform Hidden Line Removal

Operations for Depth Values

Stencel Operations for Front and Back Faces

Putting it all Together! (finally...)

VkStencilOpState
- front
  - what to do if depth operation fails
  - what to do if stencil operation fails
  - what to do if stencil operation succeeds

Stencil Operations for Front and Back Faces

VkStencilOpState
- front
  - keep the stencil value as it is
  - set stencil value to 0
  - increment stencil value
  - decrement stencil value
  - bit-invert stencil value
  - increment stencil value
  - decrement stencil value

#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

VkStencilOpState
- back
  - keep the stencil value as it is
  - set stencil value to 0
  - increment stencil value
  - decrement stencil value
  - bit-invert stencil value
  - increment stencil value
  - decrement stencil value

#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

VkPipelineDepthStencilStateCreateInfo
- sType
- pNext
- flags

#ifdef CHOICES
  VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT
  VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT
  VK_PIPELINE_CREATE_DERIVATIVE_BIT
#endif

vkCreateGraphicsPipelines
- LogicalDevice
- VK_NULL_HANDLE
- 1
- OUT

Group all of the individual state information and create the pipeline
When Drawing, We will Bind a Specific Graphics Pipeline Data Structure to the Command Buffer

```c
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.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:

- **Fixed-layout Pipeline Blocks**
- **Variable-layout Pipeline Blocks**