What is the Vulkan Graphics Pipeline?

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.

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.
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**

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

- **Which states are dynamic**
  - DepthTestEnable
  - DepthWriteEnable
  - DepthCompareOp
  - StencilTestEnable

- **Tesselation, Geometry Shaders**

- **Rasterization**

- **Dynamic State**

- **Fragment Shader module**
  - Specialization info

- **Fragment Shader Stage**

- **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
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;
}
```

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
- **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 also be set with Dynamic Variables
Creating a Typical Graphics Pipeline

```
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;
  vpmsci.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.

Link in the Shaders

```
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;
#define BITS
VK_SHADER_STAGE_VERTEX_BIT
VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT
VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT
VK_SHADER_STAGE_GEOMETRY_BIT
VK_SHADER_STAGE_FRAGMENT_BIT
VK_SHADER_STAGE_COMPUTE_BIT
VK_SHADER_STAGE_ALL_GRAPHICS
VK_SHADER_STAGE_ALL
#endif
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;           // which binding # this is
  vvibd[0].stride = sizeof( struct vertex );  // bytes between successive
  vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

#define CHOICES
VK_VERTEX_INPUT_RATE_VERTEX
VK_VERTEX_INPUT_RATE_INSTANCE
```

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

VkVertexInputElementDescription *vviad[]; // 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

#define 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

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; // f, 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

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
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;

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;

// Tessellation Shader info
Tessellation Shader info

// Geometry Shader info
Geometry Shader info

// The following code is used to describe the input vertex attributes
#define 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
What is “Primitive Restart Enable”?

```c
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`, it is `0xffffffff`

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

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

or,

```c
int restartIndex = ~0;
```
One Really Good use of Restart Enable is in Drawing Terrain Surfaces with Triangle Strips

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

Declarations:

```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;

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 scissoring 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 (scrunched) 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.

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”?  

```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

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

```cpp
VkPipelineColorBlendAttachmentState vpcbas;
vpbas.blendEnable = VK_FALSE;
vpbas.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
vpbas.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR;
vpbas.colorBlendOp = VK_BLEND_OP_ADD;
vpbas.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
vpbas.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
vpbas.alphaBlendOp = VK_BLEND_OP_ADD;
vpbas.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.
Color Blending State 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;
    #ifdef 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_NOR
        VK_LOGIC_OP_EQUIVALENT
        VK_LOGIC_OP_INVERT
        VK_LOGIC_OP_OR
        VK_LOGIC_OP_NOR
        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?

```cpp
VkDynamicState vds[] = { VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR };
    #ifdef CHOICES
        VK_DYNAMIC_STATE_VIEWPORT
        VK_DYNAMIC_STATE_LINE_WIDTH
        VK_DYNAMIC_STATE_DEPTH_BIAS
        VK_DYNAMIC_STATE_BLEND_CONSTANTS
        VK_DYNAMIC_STATE_DEPTH_BOUNDS
        VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK
        VK_DYNAMIC_STATE_STENCIL_WRITE_MASK
        VK_DYNAMIC_STATE_STENCIL_REFERENCE
    #endif
    VkPipelineDynamicStateCreateInfo vpdsci;
    vpdsci.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
    vpdsci.pNext = nullptr;
    vpdsci.flags = 0;
    vpdsci.dynamicStateCount = 0;
    vpdsci.pDynamicStates = vds;
```

---

Oregon State University
Computer Graphics

mjb – September 17, 2018
Stencil Operations for Front and Back Faces

```cpp
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;
```

Uses for Stencil Operations

Magic Lenses

Polygon edges without Z-fighting

Oregon State University
Computer Graphics
Operations for Depth Values

```
// VkPipelineDepthStencilStateCreateInfo
vkpdssci:
    vkpdssci.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
    vkpdssci.pNext = nullptr;
    vkpdssci.flags = 0;
    vkpdssci.depthTestEnable = VK_TRUE;
    vkpdssci.depthWriteEnable = VK_TRUE;
    vkpdssci.depthCompareOp = VK_COMPARE_OP_LESS;
    // Operations for Depth Values
    vkpdssci.depthBoundsTestEnable = VK_FALSE;
    vkpdssci.front = vsosf;
    vkpdssci.back = vsosb;
    vkpdssci.minDepthBounds = 0.;
    vkpdssci.maxDepthBounds = 1.;
    vkpdssci.stencilTestEnable = VK_FALSE;

// 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 pGraphicsPipeline );
```

Putting it all Together! (finally...)

```
// Group all of the individual state information and create the pipeline
```
Later on, we will Bind the Graphics Pipeline to the Command Buffer when Drawing

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