The Graphics Pipeline Data Structure (GPDS)

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

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 a 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.

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

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)
- Scissors: x, y, w, h, minDepth, maxDepth
- Rasterization: cullMode, polygonMode, frontFace, lineWidth, depthTestEnable, depthWriteEnable, depthCompareOp
- Stencil: stencilTestEnable, stencilOpStateFront, stencilOpStateBack
- Blending: blendEnable, srcColorBlendFactor, dstColorBlendFactor, srcAlphaBlendFactor, dstAlphaBlendFactor, colorBlendOp, sourceOprandFactor, destinationOprandFactor, colorWriteMask, blendEquation

Dynamic/State: which states can be set dynamically (bound to the command/buffer, outside the Pipeline)

Bot/Relics indicates that this state item can be changed with Dynamic State Variables

Creating a Graphics Pipeline from a lot of Pieces

A Graphics Pipeline Data Structure Contains the Following State Items:
Creating a Typical Graphics Pipeline

The Shaders to Use

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:

. . .

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 (scrunched) into the viewport area.

**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.

What is “Depth Clamp Enable”?

```
vpvsci.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 depthClipEnable (except it would have been red).

What is “Depth Bias Enable”?

```
vprsci.depthBiasEnable = VK_FALSE;
vprsci.depthBiasConstantFactor = 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

Declare information about how the multisampling will take place.

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

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.

Raster Operations for each Color Attachment

The 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

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

Outlining Polygons the Naïve Way

1. Draw the polygons
2. Draw the edges
3. Z-fighting
Using the Stencil Buffer to Better Outline Polygons

Stencil Operations for Front and Back Faces

Operations for Depth Values

Putting it all Together! (finally...)

VkPipelineCreateInfo

// back
VkStencilOpState & vsosb = ...; // back
vsosb.pNext = nullptr; // back
vsosb.sType = VK_STRUCTURE_TYPE_PIPELINE_STENCIL_STATE_CREATE_INFO; // back

// front
VkStencilOpState & vsosf = ...; // front
vsosf.pNext = nullptr; // front
vsosf.sType = VK_STRUCTURE_TYPE_PIPELINE_STENCIL_STATE_CREATE_INFO; // front

for each polygon
{
    Clear the SB = 0
    Draw the edges, setting SB = 1
    Draw the polygon wherever SB != 1
    Draw the edges, setting SB = 0
}

Before
After

VkPipelineCreateInfo

// global
VkGraphicsPipeline; ...

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

x = 0;
y = 0;
width = float.Width;
height = float.Height;
minDepth = 0.0f;
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