Compute Shaders

Mike Bailey
mjb@cs.oregonstate.edu

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

Compute Shaders

VkGraphicsPipelineCreateInfo

- Shaders
- VertexInput State
- InputAssembly State
- Tesselation State
- Viewport State
- Rasterization State
- MultiSample State
- DepthStencil State
- ColorBlend State
- Dynamic State
- Pipeline layout
- RenderPass
- basePipelineHandle
- basePipelineIndex

VkGraphicsPipelineCreateInfo

- VkPipelineShaderStageCreateInfo
- VkPipelineVertexInputStateCreateInfo
- VkViewportStateCreateInfo
- VkPipelineRasterizationStateCreateInfo
- VkPipelineInputAssemblyStateCreateInfo
- VkPipelineDepthStencilStateCreateInfo
- VkPipelineColorBlendStateCreateInfo
- VkPipelineDynamicStateCreateInfo

Managing the Graphics Pipeline?

Here is how you create a much-simpler Compute Pipeline

This is a Particle System application, so we need Positions, Velocities, and (possibly) Colors

Start with Creating the Data Buffers

A Reminder about Data Buffers

Creating a Shader Storage Buffer
Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

```cpp
VkMemoryRequirements
result = vkGetBufferMemoryRequirements( LogicalDevice, Buffer, OUT &vmr );

VkMemoryAllocateInfo
vmai = {
    .sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .allocationSize = vmr.size,
    .memoryTypeIndex = FindMemoryThatIsHostVisible() // 0 and 0 are offset and flags
};

result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm );

result = vkBindBufferMemory( LogicalDevice, Buffer, IN vdm, 0 ); // 0 is the offset

result = vkMapMemory( LogicalDevice, IN vdm, 0, VK_WHOLE_SIZE, 0, &ptr );
<< do the memory copy >>
result = vkUnmapMemory( LogicalDevice, IN vdm );
```

Fill the Data Buffer

```cpp
VkResult
Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    void * pGpuMemory;
    vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT &pGpuMemory );
    memcpy( pGpuMemory, data, (size_t)myBuffer.size );
    vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
    return VK_SUCCESS;
}
```

And, since we have Data Buffers, we will need Descriptor Sets to Create the Pipeline Layout

```cpp
VkPipelineShaderStageCreateInfo
vpssci = {
    .sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .stage = VK_SHADER_STAGE_COMPUTE_BIT,
    .module = computeShader,
    .pName = "main",
    .pSpecializationInfo = (VkSpecializationInfo *)nullptr
};

VkComputePipelineCreateInfo
vcpci[1] = {
    .sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .stage = vpssci,
    .layout = ComputePipelineLayout,
    .basePipelineHandle = VK_NULL_HANDLE,
    .basePipelineIndex = 0
};
```

Create the Compute Pipeline

```cpp
#version 430
#extension GL_ARB_compute_shader : enable
layout( std140, set = 0, binding = 0 )  buffer  Pos
{
    vec4  Positions[]; // array of structures
};
```

Create the Compute Pipeline Layout

```cpp
VkPipelineShaderStageCreateInfo
vpssci = {
    .sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .stage = VK_SHADER_STAGE_COMPUTE_BIT,
    .module = computeShader,
    .pName = "main",
    .pSpecializationInfo = (VkSpecializationInfo *)nullptr
};

VkComputePipelineCreateInfo
vcpci[1] = {
    .sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO,
    .pNext = nullptr,
    .flags = 0,
    .stage = vpssci,
    .layout = ComputePipelineLayout,
    .basePipelineHandle = VK_NULL_HANDLE,
    .basePipelineIndex = 0
};
```
# The Particle System Compute Shader – The Physics

```cpp
#define POINT vec3
#define VELOCITY vec3
#define VECTOR vec3
#define SPHERE vec4

const VECTOR G = VECTOR(0., -9.8, 0.);
const float DT = 0.1;
const SPHERE Sphere = vec4(-100., -800., 0., 600.);  // x, y, z,  r
```

```cpp
uint gid = gl_GlobalInvocationID.x;  // the .y and .z are both 1 in this case
POINT p = Positions[ gid ].xyz;
VELOCITY v = Velocities[ gid ].xyz;
POINT pp = p + v*DT + .5*DT*DT*G;
VELOCITY vp = v + G*DT;
Positions[ gid ].xyz = pp;
Velocities[ gid ].xyz = vp;
```

## How About Introducing a Bounce?

```cpp
VELOCITY Bounce( VELOCITY vin, VECTOR n )  {
  VELOCITY vout = reflect( vin, n );
  return vout;
}

VELOCITY BounceSphere( POINT p, VELOCITY v, SPHERE s )  {
  VECTOR n = normalize( p - s.xyz );
  return Bounce( v, n );
}

bool IsInsideSphere( POINT p, SPHERE s )  {
  float r = length( p - s.xyz );
  return ( r < s.w );
}
```

## Dispatching the Compute Shader from the Command Buffer

```cpp
const int NUM_PARTICLES = 1000000;
const int NUM_WORK_ITEMS = 64;
const int NUM_WORK_GROUPS = NUM_PARTICLES / NUM_WORK_ITEMS;

vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatch( CommandBuffer, NUM_WORK_GROUPS, 1, 1 );
```

Or,

```cpp
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatchIndirect( CommandBuffer, Buffer, 0 );  // offset
```

## Remember the Compute Pipeline?

```
VkComputePipelineCreateInfo

Shaders
Pipeline layout
basePipelineHandle
basePipelineIndex

VkPipelineShaderStageCreateInfo

VkSpecializationInfo

which stage (COMPUTE)

VkShaderModule

vkCreateComputePipelines( )

vkCreatePipelineLayout( )

Descriptor Set

Layouts

Push Constants
```

The Bouncing Particle System Compute Shader – What Does It Look Like?
A Specialization Constant is a way of injecting an integer or Boolean constant into an .spv-compiled version of a shader right before the final compilation.

That final compilation happens when you call \( \text{vkCreateComputePipelines( )} \)

Without Specialization Constants, you would have to commit to a final value before the SPIR-V compile was done, which could have been a long time ago.

Shader Source

Specialization Constants

Pipeline Shader Stage

Final Compile

\[
\begin{align*}
\text{layout( constant_id = 0 ) const int } & \text{ numXworkItems } = 32; \\
\text{layout( local_size_x = numXworkItems, local_size_y = 1, local_size_z = 1 ) in;}
\end{align*}
\]

In the compute shader

\[
\begin{align*}
\text{result } & = \text{vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline)};
\end{align*}
\]