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
ColorBlend State
Dynamic State
Pipeline layout
RenderPass
basePipelineHandle
basePipelineIndex

VkPipelineShaderStageCreateInfo

VkPipelineVertexInputStateCreateInfo

VkVertexInputBindingDescription

Viewport
x, y, w, h, 
minDepth, 
maxDepth

offset
extent
Scissor

VkPipelineRasterizationStateCreateInfo

cullMode
polygonMode
frontFace
lineWidth

VkSpecializationInfo

which stage (VERTEX, etc.)

VkShaderModule

VkPipelineInputAssemblyStateCreateInfo

Topology

VkVertexInputAttributeDescription

binding
stride
inputRate
location

VkPipelineDepthStencilStateCreateInfo

depthTestEnable
depthWriteEnable
depthCompareOp
stencilTestEnable
stencilOpStateFront
stencilOpStateBack

VkPipelineColorBlendAttachmentState

depthWriteEnable
blendEnable
colorWriteMask
srcColorBlendFactor
dstColorBlendFactor
colorBlendOp
colorSrcBlendFactor
colorSrcBlendOp
colorOutBlendFactor
colorOutBlendOp

VkPipelineColorBlendStateCreateInfo


Remember the Graphics Pipeline?

Array naming the states that can be set dynamically

Graphics Pipeline
Here is how you create a much-simpler Compute Pipeline

Start with Creating the Data Buffers

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

```
layout( std140, set = 0, binding = 0 ) buffer Pos {
    vec4 Positions[ ]; // array of structures
};
layout( std140, set = 0, binding = 1 ) buffer Vel {
    vec4 Velocities[ ]; // array of structures
};
layout( std140, set = 0, binding = 2 ) buffer Col {
    vec4 Colors[ ]; // array of structures
};
```

You can use the empty brackets, but only on the last element of the buffer. The actual dimension will be determined for you when OpenGL examines the size of this buffer's data store.
A Reminder about Data Buffers

Creating a Shader Storage Buffer

```
VkBufferCreateInfo vbci;
    vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    vbci.pNext = nullptr;
    vbci.flags = 0;
    vbci.size = << buffer size in bytes >>;
    vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
    vbci.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    vbci.queueFamilyIndexCount = 0;
    vbci.pQueueFamilyIndices = (const int32_t) nullptr;
VkBuffer Buffer;
result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );
```
Vulkan: Allocating Memory for a Buffer, Binding a Buffer to Memory, and Writing to the Buffer

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

VkMemoryAllocateInfo vmai;
    vmai.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    vmai.pNext = nullptr;
    vmai.flags = 0;
    vmai.allocationSize = vmr.size;
    vmai.memoryTypeIndex = FindMemoryThatIsHostVisible( );

VkDeviceMemory vdm;
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

```c
VkResult Fill05DataBuffer( IN MyBuffer myBuffer, IN void * data )
{
    // the size of the data had better match the size that was used to init the buffer!
    void * pGpuMemory;
    vkMapMemory( LogicalDevice, IN myBuffer.vdm, 0, VK_WHOLE_SIZE, 0, &pGpuMemory );
    // 0 and 0 are offset and flags
    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

Create the Compute Pipeline Layout

```
VkSampler* Sampler;

VkDescriptorSetLayoutBinding ComputeSet[1];
ComputeSet[0].binding = 0;
ComputeSet[0].descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
ComputeSet[0].descriptorCount = 3;
ComputeSet[0].stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
ComputeSet[0].pImmutableSamplers = (VkSampler *)nullptr;

VkDescriptorSetLayoutCreateInfo vdslc;
vdslc.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
vdslc.pNext = nullptr;
vdslc.flags = 0;
vdslc.bindingCount = 1;
vdslc.pBindings = &ComputeSet[0];
result = vkCreateDescriptorSetLayout( LogicalDevice, &vdslc, PALLOCATOR, OUT &ComputeSetLayout );

VkPipelineLayoutCreateInfo vplci;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.flags = 0;
vplci.setLayoutCount = 1;
vplci.pSetLayouts = ComputeSetLayout;
vplci.pushConstantRangeCount = 0;
vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;
result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &ComputePipelineLayout );
```
Create the Compute Pipeline

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

```cpp
VkComputePipelineCreateInfo vcpci[1];
vcpci[0].sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
v pci[0].pNext = nullptr;
v pci[0].flags = 0;
v pci[0].stage = vpssci;
v pci[0].layout = ComputePipelineLayout;
v pci[0].basePipelineHandle = VK_NULL_HANDLE;
v pci[0].basePipelineIndex = 0;
result = vkCreateComputePipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline);
```

The Particle System Compute Shader -- Setup

```cpp
#version 430
#extension GL_ARB_compute_shader : enable
layout( std140, set = 0, binding = 0 ) buffer Pos
{
    vec4 Positions[ ]; // array of structures
};
layout( std140, set = 0, binding = 1 ) buffer Vel
{
    vec4 Velocities[ ]; // array of structures
};
layout( std140, set = 0, binding = 2 ) buffer Col
{
    vec4 Colors[ ]; // array of structures
};
layout( local_size_x = 64, local_size_y = 1, local_size_z = 1 ) in;
```
#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

... 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;

The Particle System Compute Shader – The Physics

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 );
}
The Particle System Compute Shader – How About Introducing a Bounce?

```
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;
if( IsInsideSphere( pp, Sphere ) )
{
    vp = BounceSphere( p, v, S );
    pp = p + vp*DT + .5*DT*DT*G;
}
Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;
```

Graphics Trick Alert: Making the bounce happen from the surface of the sphere is time-consuming. Instead, bounce from the previous position in space. If DT is small enough (and it is), nobody will ever know...

```
p' = p + v \cdot t + \frac{1}{2} G \cdot t^2
v' = v + G \cdot t
```

Dispatching the Compute Shader from the Command Buffer

```
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,
```
vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );
vkCmdDispatchIndirect( CommandBuffer, Buffer, 0 ); // offset
```
The Bouncing Particle System Compute Shader – What Does It Look Like?

Remember the Compute Pipeline?

1. `Descriptor Set Layout`
2. `Push Constants`
3. `VkPipelineLayoutCreateInfo`
4. `VkSpecializationInfo`
5. `VkShaderModule`
6. `VkPipelineShaderStageCreateInfo`
7. `VkPipelineCreateInfo` (which stage (COMPUTE))
8. `Shader` (Pipeline layout, basePipelineHandle, basePipelineIndex)
9. `VkCreatePipelineLayout()`
10. `vkCreatePipeline()`
11. `Compute Pipeline`
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 `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.

**In the compute shader**

```cpp
layout( constant_id = 0 ) const int numXworkItems = 32;
layout( local_size_x = numXworkItems, local_size_y = 1, local_size_z = 1 ) in;
```

**In the C/C++ program:**

```cpp
VkSpecializationMapEntry vsme[1]; // one array element for each Specialization Constant
vsme.constantID = 0;
vsme.offset = 0; // # bytes into the Specialization Constant
vsme.size = sizeof(int); // size of just this Specialization Constant
int numXworkItems = 64;
VkSpecializationInfo vsi;
vsii.mapEntryCount = 1;
vsii.pMapEntries = &vsme[0]; // size of all the Specialization Constants together
vsii.pData = &numXworkItems; // array of all the Specialization Constants
```
Linking the Specialization Constants into the Compute Pipeline

VkSpecializationMapEntry vsme[1];
   vsme.constantID = 0;
   vsme.offset = 0;
   vsme.size = sizeof(int);

int numXworkItems = 64;

VkSpecializationInfo vsi;
   vsi.mapEntryCount = 1;
   vsi.pMapEntries = &vsme[0];
   vsi.dataSize = sizeof(int);
   vsi.pData = &numXworkItems;

VkPipelineShaderStageCreateInfo vpssci;
   vpssci.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
   vpssci.pNext = nullptr;
   vpssci.flags = 0;
   vpssci.stage = VK_SHADER_STAGE_COMPUTE_BIT;
   vpssci.module = computeShader;
   vpssci.pName = "main";
   vpssci.pSpecializationInfo = &vsi;

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

result = vkCreateComputePipelines( LogicalDevice, VK_NULL_HANDLE, 1, &vcpci[0], PALLOCATOR, &ComputePipeline );