Compute Shaders

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu
This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

Compute Shaders

Here is how you create a Compute Pipeline

Start by Creating the Data Buffers

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

A Reminder about Data Buffers

Creating a Shader Storage Buffer

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

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 iont32_t) nullptr;
result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer );

\texttt{Creating a Shader Storage Buffer}

\texttt{VkBufferCreateInfo vbci;
    \texttt{vbci.sType} = \texttt{VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO};
    \texttt{vbci.pNext} = \texttt{nullptr};
    \texttt{vbci.flags} = \texttt{0};
    \texttt{vbci.size} = \texttt{buffer size in bytes};
    \texttt{vbci.usage} = \texttt{VK_USAGE_STORAGE_BUFFER_BIT};
    \texttt{vbci.sharingMode} = \texttt{VK_SHARING_MODE_EXCLUSIVE};
    \texttt{vbci.queueFamilyIndexCount} = \texttt{0};
    \texttt{vbci.pQueueFamilyIndices} = \texttt{(const iont32_t) nullptr};
    \texttt{result} = \texttt{vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &Buffer });
 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 );

 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, OUT &pGpuMemory);
     // 0 and 0 are offset and flags
     memcpy( pGpuMemory, data, (size_t)myBuffer.size );
     vkUnmapMemory( LogicalDevice, IN myBuffer.vdm );
     return VK_SUCCESS;
 }

 VkPipelineCreateInfo ComputePipeline;

csCl = (VkComputes * )ComputePipelineCreateInfo;

csCl.sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
csCl.pNext = nullptr;
csCl.flags = 0;
csCl.stage = vpssci;
csCl.layout = ComputePipelineLayout;

csCl.basePipelineHandle = VK_NULL_HANDLE;
csCl.basePipelineIndex = 0;

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

 #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;

 This is the number of work-item per work-group, set in the compute shader.
 The number of work-groups is set in the vkCmdDispatch( ) function call in the C/C++ program.
# The Particle System Compute Shader – The Physics

```
#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;
```

---

## How About Introducing a Bounce?

- **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 );`

---

### 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.

---

## Dispatching the Compute Shader from the Command Buffer

- `const int NUM_PARTICLES         = 1024*1024;`
- `const int NUM_WORK_ITEMS     =              64;`
- `const int NUM_X_WORK_GROUPS = NUM_PARTICLES / NUM_WORK_ITEMS;`

- `vkCmdBindPipeline( CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline );`
- `vkCmdDispatch( CommandBuffer, NUM_X_WORK_GROUPS, 1, 1 );`

---

### This is the number of work-groups, set in the C/C++ program.

- The number of work-items per work-group is set in a layout in the compute shader.