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 with Creating the Data Buffers

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 iont32_t) nullptr;

VkBuffer Buffer;
result = vkCreateBuffer ( LogicalDevice, IN &vbci, PALLOCATOR, OUT &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 OpenGL examines the size of this buffer's data store.

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

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.

Remember the Graphics Pipeline?

Remember the Graphics Pipeline?
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;
}
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

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;
```

The Particle System Compute Shader – 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(p, v, s);
}

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

The Particle System Compute Shader – The Physics

 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);
```

Remember the 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.

### Shader Source

```c
layout( constant_id = 0 ) const int numXworkItems = 32;
layout( local_size_x = numXworkItems, local_size_y = 1, local_size_z = 1 ) in;
VkSpecializationMapEntry vsme[1];  // one array element for each Specialization Constant
    vsme.constantID = 0;
    vsme.offset = 0;  // # bytes into the Specialization Constant array this one item is
    vsme.size = sizeof(int);  // size of just this Specialization Constant
int numXworkItems = 64;
VkSpecializationInfo vsi;
    vsi.mapEntryCount = 1;
    vsi.pMapEntries = &vsme[0];
    vsi.dataSize = sizeof(int);  // size of all the Specialization Constants together
    vsi.pData = &numXworkItems;  // array of all the Specialization Constants
```

### Linking the Specialization Constants into the Compute Pipeline

```c
vkSpecializationMapEntry vsme[1];
    vsme.constantID = 0;
    vsme.offset = 0;
    vsme.size = sizeof(int);
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);
```