The Rendering Draws the Particles by Reading the Position and Color Buffers

The Example We Are Going to Use Here is a Particle System

The Compute Shader Moves the Particles by Recomputing the Position and Velocity Buffers

The Data in your C/C++ Program will look like This

The Data in your Compute Shader will look like This

Note that .w and .vw are not actually needed. But, by making these structure sizes a multiple of 4 floats, it doesn’t matter if they are declared with the std140 or the std430 qualifier. I think this is a good thing.

The Data in your Compute Shader Data Structure... Highlighted boxes are ones that the Graphics Pipeline Data Structure also has. Note how less complicated this is!
result = vkCreatePipelineLayout( LogicalDevice, IN &vplci;
result = vkCreateDescriptorSetLayout( LogicalDevice, IN &vdslc, PALLOCATOR, OUT &vds);
.
.
VkDeviceMemory vdm;
result = vkAllocateMemory( LogicalDevice, IN &vmai, PALLOCATOR, OUT &vdm);
.
.
VkMemoryAllocateInfo vmai;
result = vkGetBufferMemoryRequirements( LogicalDevice,
VkMemoryRequirements vmr;
vplci.pPushConstantRanges = (VkPushConstantRange *)nullptr;
vplci.pushConstantRangeCount = 0;
vplci.pSetLayouts =
vplci.setLayoutCount = 1;
vplci.flags = 0;
vplci.pNext = nullptr;
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

Computing Memory for a Buffer, Binding a Buffer
A Reminder about Data Buffers
Creating a Shader Storage Buffer
Creating a Vulkan Data Buffer

```cpp
VkBuffer Buffer;
VkBufferCreateInfo vbci;
vbci.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vbci.pNext = nullptr;
vbci.flags = 0;
vbci.size = NUM_PARTICLES * sizeof(glm::vec4);
vbci.usage = VK_USAGE_STORAGE_BUFFER_BIT;
vbci.sharingMode = VK_SHARING_MODE_CONCURRENT;
vbci.queueFamilyIndexCount = 0;
vbci.pQueueFamilyIndices = (const iont32_t) nullptr;
result = vkCreateBuffer(LogicalDevice, IN &vbci, PALLOCATOR, OUT &posBuffer);
```

Allocating Memory and Binding the Buffer

```cpp
VkMemory Requirements
result = vkGetBufferMemoryRequirements(LogicalDevice, posBuffer, 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, posBuffer, IN vdm, 0); // 0 is the offset

MyBuffer myPosBuffer;
myPosBuffer.size = vbci.size;
myPosBuffer.buffer = PosBuffer;
myPosBuffer.vdm = vdm;
```

Fill the Buffers

```cpp
struct pos * positions;
vkMapMemory(LogicalDevice, IN myPosBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &positions);
for (int i = 0; i < NUM_PARTICLES; i++) {
    positions[i].x = Ranf(XMIN, XMAX);
    positions[i].y = Ranf(YMIN, YMAX);
    positions[i].z = Ranf(ZMIN, ZMAX);
    positions[i].w = 1.;
}
vkUnmapMemory(LogicalDevice, IN myPosBuffer.vdm);

struct vel * velocities;
vkMapMemory(LogicalDevice, IN myVelBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &velocities);
for (int i = 0; i < NUM_PARTICLES; i++) {
    velocities[i].x = Ranf(VXMIN, VXMAX);
    velocities[i].y = Ranf(VYMIN, VYMAX);
    velocities[i].z = Ranf(VZMIN, VZMAX);
    velocities[i].w = 0.;
}
vkUnmapMemory(LogicalDevice, IN myVelBuffer.vdm);

struct col * colors;
vkMapMemory(LogicalDevice, IN myColBuffer.vdm, 0, VK_WHOLE_SIZE, 0, OUT (void *) &colors);
for (int i = 0; i < NUM_PARTICLES; i++) {
    colors[i].r = Ranf(.3f, 1.);
    colors[i].g = Ranf(.3f, 1.);
    colors[i].b = Ranf(.3f, 1.);
    colors[i].a = 1.;
}
vkUnmapMemory(LogicalDevice, IN myColBuffer.vdm);
```

Allocate Memory and Bind the Buffer

```cpp
#include <stdlib.h>
#define TOP 2147483647. // 2^31 - 1

float Ranf(float low, float high) {
    long random( );         // returns integer 0 - TOP
    float r = (float)rand( );
    return low + r * (high - low) / (float)RAND_MAX;
}
```

The Particle System Compute Shader

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

The Data gets Divided into Large Quantities called Work-Groups, each of which is further Divided into Smaller Units called Work-Items

```
#WorkGroups = GlobalInvocationSize / WorkGroupSize
5 = 20 / 4
```

This is the number of work-items per work-group in the compute shader.
The number of work-groups is set in the
vkCmdDispatch(commandBuffer, workGroupCountX, workGroupCountY, workGroupCountZ);
invocation call in the application program.
The Data Needs to be Divided into Large Quantities called Work-Groups, each of which is further Divided into Smaller Units called Work-Items.

The Invocation Space can be 1D, 2D, or 3D. This one is 2D.

```
#define POINT vec3
#define VELOCITY vec3
#define VECTOR vec3
#define SPHERE vec4
  // xc, yc, zc, r
#define PLANE vec4
  // a, b, c, d
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; // where I am in the global dataset (6 in this example)
// (as a 1d problem, the .y and .z are both 1)
POIINT        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;
```

```
// plane equation:  Ax + By + Cz + D = 0
// (it turns out that (A,B,C) is the normal)
```

The Particle System Compute Shader – The Physics

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

```
BOOL
IsUnderPlane( POINT p, PLANE pl )
{
  float r = pl.x*p.x +  pl.y*p.y +  pl.z*p.z +  pl.w;
  return  (  r  <  0. );
}
```

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

```
VELOCITY
BouncePlane( POINT p, VELOCITY vin, PLANE pl)
{
  VECTOR  n = normalize(  VECTOR( pl.xyz )  );
  return Bounce( vin, n );
}
```

```
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?

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…
vkCmdDispatch

#define NUM_X_WORK_GROUPS (NUM_PARTICLES / NUM_WORK_ITEMS_PER_GROUP)
#define NUM_WORK_ITEMS_PER_GROUP 64
#define NUM_PARTICLES 1024 * 1024

result = vkCreateGraphicsPipelines(LogicalDevice, VK_NULL_HANDLE, 1, &vgpci, nullptr);

vkCmdBindPipeline(CommandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, ComputePipeline);

vkCmdPipelineBarrier(
  commandBuffer, VK_PIPELINE_STAGE_VERTEX_INPUT_BIT, VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE.ComputeShader_BIT,
  VK_ACCESS_SHADER_WRITE_BIT, VK_ACCESS_SHADER_READ_BIT, bufferMemoryBarrierCount, bufferMemoryBarriers);

vvibd[0].binding = 0; // which binding # this is
vvibd[0].location = 0; // location in the layout decoration
vvibd[0].stride = sizeof(glm::vec4); // bytes between successive structs
vvibd[0].offset = offsetof(struct pos, pos); // 0
vvibd[1].binding = 1;
vvibd[1].location = 1;
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
vvibd[1].stride = sizeof(glm::vec4);
vvibd[2].binding = 2;
vvibd[2].location = 2;
vvibd[2].stride = sizeof(glm::vec4);

vviad[0].offset = offsetof(struct pos, pos); // 0
vviad[0].format = VK_FORMAT_VEC4; // x, y, z, w
vviad[0].binding = 0; // which binding description this is part of
vviad[0].location = 0; // 0

vviad[1].offset = offsetof(struct vel, vel); // 0
vviad[1].format = VK_FORMAT_VEC4; // r, g, b, a
vviad[1].binding = 0; // 0
vviad[1].location = 1; // 1

vviad[2].offset = offsetof(struct col, col); // 0
vviad[2].format = VK_FORMAT_VEC4; // b, g, r, a
vviad[2].binding = 0; // 0
vviad[2].location = 2; // 2

vpvisci.pVertexAttributeDescriptions = vviad;
vpvisci.vertexAttributeDescriptionCount = 3;
vpvisci.pVertexBindingDescriptions = vvibd;
vpvisci.vertexBindingDescriptionCount = 3;
vpvisci.pNext = nullptr;
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

vpvisci.topology = VK_PRIMITIVE_TOPOLOGY_POINT_LIST;
vpvisci.flags = 0;
vpvisci.pNext = nullptr;
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;

vpvisor = CreatePipelineInputAssemblyStateCreateInfo(IN &vpvisor);

vpvisor.topology = VK_PRIMITIVE_TOPOLOGY_POINT_LIST;
vpvisor.flags = 0;
vpvisor.pNext = nullptr;
vpvisor.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;

vpvisorInputAssemblyStateCreateInfo_vpvisor;
vpvisorInputAssemblyStateCreateInfo_vpvisor.topology = VK_PRIMITIVE_TOPOLOGY_POINT_LIST;
vpvisorInputAssemblyStateCreateInfo_vpvisor.flags = 0;
vpvisorInputAssemblyStateCreateInfo_vpvisor.pNext = nullptr;
vpvisorInputAssemblyStateCreateInfo_vpvisor.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;

Displaying the Particles

We will come to the Pipeline later, but for now, know that a Vulkan Pipeline is essentially a very large data structure that holds (what OpenGl would call) the state, including how to parse its vertex input.

In this case, it was the vertex attributes that were passed into the pipeline. It was up to the programmer to describe what vertex attributes were present and in what format they were.

The first step would be to tell the pipeline which vertex attributes it was expecting to receive data for.

The vertex attributes were:

- aPosition (targeting the position attribute)
- aVelocity (targeting the velocity attribute)
- aColor (targeting the color attribute)

Each vertex attribute had its own binding and location. The binding determined the binding number of the vertex attribute, and the location determined the location of the vertex attribute within the layout decoration.

The format of each vertex attribute was also specified. This was important because Vulkan uses a flexible vertex format system, and the format of each vertex attribute could be specified as needed.

Once the vertex attributes were described, the command buffer could be used to issue commands to the pipeline. These commands would typically include binding the vertex attributes to the pipeline and setting other pipeline parameters, such as the vertex shader.

Once the pipeline was set up, the command buffer could be used to execute the compute shader. This would typically be done using a VkCmdDispatch command, which would specify the number of work-items to be computed and the number of work-groups to be used.

The compute shader would then be executed for each work-item, and the results would be used to update the state of the application program. The number of work-items per work-group was set in the compute shader, and this was the number of work-groups, set in the application program.

Additionally, the command buffer could be used to issue commands to the pipeline to set the pipeline's barriers. These commands would typically be used to indicate when the drawing stage was complete and the compute stage could begin.

Telling the Pipeline about its Input

Disp
VkBuffer buffers[] = {MyPosBuffer.buffer, MyVelBuffer.buffer, MyColBuffer.buffer };
size_t offsets[] = { 0, 0, 0 };
vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 3, buffers, offsets);
const uint32_t vertexCount = NUM_PARTICLES;
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw(CommandBuffers[nextImageIndex], NUM_PARTICLES, 1, 0, 0);

// vertexCount, instanceCount, firstVertex, firstInstance

VkBufferMemoryBarrier vbmb;
vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessFlags = 0;
vbmb.dstAccessFlags = VK_ACCESS_UNIFORM_READ_BIT;
vbmb.srcQueueFamilyIndex = 0;
vbmb.dstQueueFamilyIndex = 0;
vbmb.buffer = ??
vbmb.offset = 0;
vbmb.size = ??

const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier(commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT, VK_DEPENDENCY_BY_REGION_BIT, 0, nullptr, bufferMemoryBarrierCount, &vbmb, 0, nullptr);

// Setting a Pipeline Barrier so the Compute Waits for the Drawing