OpenGL Compute Shaders

Recent graphics hardware has become extremely powerful. A strong desire to harness this power for work that does not fit the traditional graphics pipeline has emerged. To address this, Compute Shaders are a new single-stage program. They are launched in a manner that is essentially stateless. This allows arbitrary workloads to be sent to the graphics hardware with minimal disturbance to the GL state machine.

In most respects, a Compute Shader is identical to all other OpenGL shaders, with similar status, uniforms, and other such properties. It has access to many of the same data as all other shader types, such as textures, image textures, atomic counters, and so on. However, the Compute Shader has no predefined inputs, nor any fixed-function outputs. It cannot be part of a rendering pipeline and its visible side effects are through its actions on shader storage buffers, image textures, and atomic counters.

Why Not Just Use OpenCL Instead?

OpenGL is great! It does a super job of using the GPU for general-purpose data-parallel computing. And, OpenCL is more feature-rich than OpenGL compute shaders. So, why use Compute Shaders ever if you've got OpenCL? Here's what I think:

- OpenCL requires installing a separate driver and separate binaries. While this is not a huge deal, it does take time and effort. When everyone catches up to OpenGL 4.3, Compute Shaders will just "be there" as part of core OpenGL.
- Compute Shaders use the GLSL language, something that all OpenGL programmers should already be familiar with (or will be soon).
- Compute shaders use the same context as does the OpenGL rendering pipeline. There is no need to acquire and release the context as OpenGL/OpenCL must do.
- I'm assuming that calls to OpenGL compute shaders are more lightweight than calls to OpenCL kernels are. (true?) This should result in better performance. (true? how much?)
- Using OpenCL is somewhat cumbersome. It requires a lot of setup (queries, platforms, devices, queues, kernels, etc.). Compute Shaders look to be more convenient. They just kind of flow in with the graphics.

The bottom line is that I continue to use OpenCL for the big, bad stuff. But, for lighter-weight data-parallel computing that interacts with graphics, I use the Compute Shaders.

Passing Data to the Compute Shader Happens with a Cool New Buffer Type – the Shader Storage Buffer Object

The tricky part is getting data in and out of the Compute Shader. This trickiness comes from the specification phrase: “In most respects, a Compute Shader is identical to all other OpenGL shaders, with similar status, uniforms, and other such properties. It has access to many of the same data as all other shader types, such as textures, image textures, atomic counters, and so on.”

Compute Shaders, looking like other shaders, haven't had direct access to general arrays of data (hacked access, yes, direct access, no). But, because Compute Shaders represent opportunities for massive data-parallel computations, that is exactly what you want them to have access to. Thus, OpenGL 4.3 introduced the Shader Storage Buffer Object. This is very cool, and has been needed for a long time!
Passing Data to the Compute Shader Happens with a Cool New Buffer Type – the Shader Storage Buffer Object

And, like other OpenGL buffer types, Shader Storage Buffer Objects can be bound to the context, making them easy to access from inside the Compute Shaders.

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 OpenGL Rendering Draws the Particles by Reading the Position Buffer

The Data Needs to be Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units called Work-Items

# Work Groups = \frac{\text{Global Invocation Size}}{\text{Work Group Size}}

[Diagram of shader storage buffer objects]

Glaring Buffer

GL_SHADER_STORAGE_BUFFER

Global Buffers

Local Buffers

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

20 total items to compute:

5 Work Groups

4 Work Items

A Mechanical Equivalent of a GPU

"Streaming Multiprocessor"

"CUDA Core"

"Data"
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.

The Global Invocation Size and Work Group Size:

\[
\text{Global Invocation Size} = \text{Work Group Size} \\
5 \times 4 = 20 \times 12 \\
4 \times 3 = 4 \times 3 \\
\text{Work Group Size} = 4 \times 3 \\
\text{Work Items} = 4 \times 3
\]

The Particle System Compute Shader -- Setup

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 the buffer's data store.

The Particle System Compute Shader -- Using the glslprogram C++ Class to Handle Everything

GLSLProgram *Particles = new GLSLProgram( );
bool valid = Particles->Create( "particles.cs" );
if( ! valid ) { . . . }
Particles->Use( );
Particles->DispatchCompute( NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1 );
Render->Use( ); // draw the particles
. . .
const vec3 G = vec3(0., -9.8, 0.);
const float DT = 0.1;

uint gid = gl_GlobalInvocationID.x; // the .y and .z are both 1 in this case

vec3 p = Positions[gid].xyz;
vec3 v = Velocities[gid].xyz;
vec3 pp = p + v*DT + .5*DT*DT*G;
vec3 vp = v + G*DT;

Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;

The Particle System Compute Shader – The Physics

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

vec3 p = Positions[gid].xyz;
vec3 v = Velocities[gid].xyz;
vec3 pp = p + v*DT + .5*DT*DT*G;
vec3 vp = v + G*DT;

if( IsInsideSphere(pp, Sphere) )
{
    vp = BounceSphere(p, v, Sphere);
    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?

const vec4 Sphere = vec4(-100., -800., 0., 600.);

vec3 vin, vec3 n
vec3 vout = reflect(vin, n);
return vout;

vec3 BounceSphere(vec3 p, vec3 v, vec4 s)
{
    vec3 n = normalize(p - s.xyz);
    return Bounce(v, n);
}

bool IsInsideSphere(vec3 p, vec4 s)
{
    float r = length(p - s.xyz);
    return (r < s.w);
}

The Bouncing Particle System Compute Shader – What Does It Look Like?