OpenGL Compute Shader – the Basic Idea

Paraphrased from the ARB_computeshader spec:

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?

OpenCL 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 even if you’ve got OpenCL? Here’s what I think:

• OpenCL requires installing a separate driver and separate libraries. 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.

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

The trick part is getting data into 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.”

OpenCL programs have access to general arrays of data, and also access to OpenGL arrays of data in the form of buffer objects. Compute Shaders, lacking 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 use.

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 indexed binding points, making them easy to access from inside the Compute Shaders.

OpenGL Context

Shader Storage Buffer Object

Texture0

Dest.

Texture1 Texture2 Texture3

Buffer0

Buffer2 Buffer1

Display

(Any resemblance this diagram has to a mother sow is accidental, but not entirely inaccurate…)

The OpenGL Rendering Draws the Particles by Reading the Position Buffer

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

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

The Data Needs to be Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items
Running the Compute Shader from the Application

```c
void glDispatchCompute(num_groups_x, num_groups_y, num_groups_z);
```

If the problem is 2D, then `num_groups_z = 1`

If the problem is 1D, then `num_groups_y = 1` and `num_groups_z = 1`

---

Writing a C++ Class to Handle Everything is Fairly Straightforward

Setup:

```c
GLSLProgram *Particles = new GLSLProgram();
if(!valid) { ... }
Particles->Use();
Particles->DispatchCompute(NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1);
Render->Use();
```

Using:

```c
Particles->Use();
Particles->DispatchCompute(NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1);
```

---

The Particle System Compute Shader — Setup

```c
layout( local_size_x = 128, local_size_y = 1, local_size_z = 1 ) in;
```

---

Special Pre-set Variables in the Compute Shader

```c
in uvec3 gl_NumWorkGroups; // Some number as in the glDispatchCompute call
const uvec3 gl_WorkGroupSize; // Some number as in the layout local_size_*
in uvec3 gl_WorkGroupID; // Which workgroup this thread is in
in uvec3 gl_LocalInvocationID; // Where this thread is in the current workgroup
in uvec3 gl_GlobalInvocationID; // Where this thread is in all the work items
in uint gl_LocalInvocationIndex; // 1D representation of the gl_LocalInvocationID (used for indexing into a shared array)
```

---

The Particle System Compute Shader — The Physics

```c
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 + 0.5 * DT * DT * G;
vec3 vp = v + G * DT;
Positions[gid].xyz = pp;
Velocities[gid].xyz = vp;
```
The Particle System Compute Shader – How About Introducing a Bounce?

```cpp
const vec4 SPHERE = vec4(-100., -800., 0., 600.); // x, y, z, r
// (could also have passed this in)
vec3
Bounce( 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 Particle System Compute Shader – How About Introducing a Bounce?

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

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, nobody will ever know...

```cpp
void glDispatchComputeGroupSize(
    num_groups_x,        num_groups_y,         num_groups_z,
    work_group_size_x, work_group_size_y, work_group_size_z);

void glDispatchCompute(
    num_groups_x,    num_groups_y,    num_groups_z);
```

In your C/C++ code, replace:

```cpp
#define NUMPARTICLES 1024*1024
#define WORK_GROUP_SIZE 128
#define NUMGROUPS ( NUMPARTICLES / WORK_GROUP_SIZE )
```

And, in your shader code, replace:

```cpp
layout( std140, binding=6 ) buffer Col
{
    vec4 Colors[ ];
};
```

Other Useful Stuff – Copying Global Data to a Local Array Shared by the Entire Work-Group

There are some applications, such as image convolution, where threads within a work-group need to operate on each other’s input or output data. In those cases, it is usually a good idea to create a local shared array that all of the threads in the work-group can access. You do it like this:

```cpp
layout( binding=6 ) buffer Col
{
    vec4 Colors[ ];
};
```

In the other way of dispatching a compute shader, you would have modified the parallel parameters by changing values in both the C/C++ code and in the shader code.
There are some applications it is useful to be able to return some numerical information about the running of the shader back to the application program. For example, here’s how to count the number of bounces:

```glsl
int bounces = 0;
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &bounces);
```

```c
int x, y, z = 0;
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &z);
```

```glsl
layout( std140, binding=7 ) buffer { atomic_uint bounceCount };
if( IsInsideSphere( pp, SPHERE ) )
{
    vp = BounceSphere( p, v, SPHERE );
    pp = p + vp*DT + .5*DT*DT*G;
    atomicCounterIncrement( bounceCount );
}
```

```glsl
glBindBuffer( GL_SHADER_STORAGE_BUFFER, countBuffer );
GLuint *ptr = (GLuint *) glMapBuffer( GL_SHADER_STORAGE_BUFFER, GL_READ_ONLY );
GLuint bounceCount = ptr[0];
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );
fprintf( stderr, "%d bounces
n", bounceCount );
```

Another example would be to count the number of fragments drawn so we know when all particles are outside the viewing volume, and can stop animating:

```glsl
int particleCount = 0;
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &particleCount);
```

```c
int x, y, z = 0;
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &z);
```

```glsl
layout( std140, binding=8 ) buffer { atomic_uint particleCount };
atomicCounterIncrement( particleCount );
```

```glsl
glBindBuffer( GL_SHADER_STORAGE_BUFFER, particleBuffer );
GLuint *ptr = (GLuint *) glMapBuffer( GL_SHADER_STORAGE_BUFFER, GL_READ_ONLY );
GLuint particleCount = ptr[0];
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );
If( particleCount == 0 ) DoAnimate = false; // stop animating
```

While we are at it, there is a cleaner way to set all values of a buffer to a preset value. In the previous example, we cleared the `countBuffer` by saying:

```glsl
glBindBufferBase( GL_ATOMIC_COUNTER_BUFFER, 7, countBuffer);
GLuint zero = 0;
glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &zero);
```

We could have also done it by using a new OpenGL 4.3 feature, `Clear Buffer Data`, which sets all values of the buffer object to the same preset value. This is analogous to the C function `memset()`:

```glsl
glBindBufferBase( GL_ATOMIC_COUNTER_BUFFER, 7, countBuffer);
GLuint zero = 0;
glClearBufferData( GL_ATOMIC_COUNTER_BUFFER, GL_R32UI, GL_RED, GL_UNSIGNED_INT, &zero );
```

Presumably this is faster than using `glBufferSubData`, especially for large-sized buffer objects (unlike this one).