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.

What Do We Need to Do Differently to Write a Compute Shader?

Not much!

1. A Compute Shader is created just like any other GLSL shader, except that its type is GL_COMPUTE_SHADER (duh…). You compile it and link it just like any other GLSL shader program.
2. A Compute Shader must be in a shader program all by itself. There cannot be vertex, fragment, etc. shaders in there with it. (I don’t understand why this is necessary.)
3. A Compute Shader has access to uniform variables and buffer objects but cannot access any pipeline variables such as attributes or variables from other stages. It stands alone.
4. A Compute Shader needs to declare the number of work-items in each of its work-groups in a special GLSL layout statement.

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. The 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!

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:

1. 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.
2. Compute Shaders use the GLSL language, something that all OpenGL programmers should already be familiar with (or will be soon).
3. Compute shaders use the same context as does the OpenGL rendering pipeline. There is no need to acquire and release the context as OpenCL+OpenGL must do.
4. 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?)
5. Using OpenGL 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.

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:

1. OpenGL Compute Shaders

OpenGL Compute Shader – the Basic Idea

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

Why Not Just Use OpenCL Instead?
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.

OpenGL Context
Buffer0
Buffer1
Texture0
Display
Dest.
Buffer2
Buffer3.

The OpenGL Rendering Draws the Particles by Reading the Position Buffer

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

Setting up the Shader Storage Buffer Objects in Your C/C++ Program

#define NUM_PARTICLES 1024*1024 // total number of particles to move
#define WORK_GROUP_SIZE 128 // # work-items per work-group

struct pos
{
  float x, y, z, w; // positions
};

struct vel
{
  float vx, vy, vz, vw; // velocities
};

struct color
{
  float r, g, b, a; // colors
};

// need to do the following for both position, velocity, and colors of the particles:
GLuint posSSbo;
GLuint velSSbo;
GLuint colSSbo;

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

20 total items to compute:

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

#WorkGroups = \( \frac{\text{GlobalInvocationSize}}{\text{WorkGroupSize}} \)

5 = \( \frac{20}{4} \)
The Data Needs to be Divided into Large Quantities called Work-Groups, each of which is further Divided into Smaller Units called Work-Items.

20x12 (=240) total items to compute:

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

The GlobalInvocationSize
WorkGroups =
WorkGroupSize
5x4 = 20x12 4x3

4 Work-Groups

5 Work-Groups

#WorkGroups =

Work-Items

Running the Compute Shader from the Application

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

Invoking the Compute Shader in Your C Program

void glDispatchCompute(   num_groups_x,    num_groups_y,    num_groups_z );

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 );
Particles->UnUse( ); Render->Use( ); // draw the particles
Render->UnUse( );

Setup:
Using:

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.

Special Pre-set Variables in the Compute Shader

in uvec3 gl_NumWorkGroups ;
const uvec3 gl_WorkGroupSize ;
in uvec3 gl_LocalInvocationID ;
in uvec3 gl_GlobalInvocationID ;
in uint gl_LocalInvocationIndex ;

in gl_WorkGroupID ;
in gl_NumWorkGroups - 1
in gl_LocalInvocationID ;
in gl_WorkGroupSize - 1
in gl_GlobalInvocationID ;
in gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID

The Particle System Compute Shader -- Setup

layout( local_size_x = 128, local_size_y = 1, local_size_z = 1 )

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 pp = p + v*DT + 0.5*DT*DT*G;
vec3 vp = v + G*DT;

if( IsInsideSphere( pp, Sphere ) )
{
  vp = BounceSphere( p, v, Sphere );
  pp = p + vp*DT + 0.5*DT*DT*G;
}

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

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...