## OpenGL Compute Shaders

### Computer Graphics

**OpenGL Compute Shader – the Basic Idea**

- A Shader Program, with only a Compute Shader in it
- Application Invokes the Compute Shader to Modify the OpenGL Buffer Data
- Application Invokes OpenGL Rendering which Reads the Buffer Data
- Another Shader Program, with pipeline rendering in it

### If We Know GLSL (and you do), 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 into 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!

### 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
```c
#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.

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

glGenBuffers( 1, &posSSbo);
glBindBuffer( GL_SHADER_STORAGE_BUFFER, posSSbo );
glBufferData( GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(struct pos), NULL, GL_STATIC_DRAW );
GLint bufMask = GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT ; // the invalidate makes a big difference when re-writing
struct pos *points = (struct pos *) glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct pos), bufMask );
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    points[i].x = Ranf( XMIN, XMAX );
    points[i].y = Ranf( YMIN, YMAX );
    points[i].z = Ranf( ZMIN, ZMAX );
    points[i].w = 1.;
}
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );

glGenBuffers( 1, &velSSbo);
glBindBuffer( GL_SHADER_STORAGE_BUFFER, velSSbo );
glBufferData( GL_SHADER_STORAGE_BUFFER, NUM_PARTICLES * sizeof(struct vel), NULL, GL_STATIC_DRAW );
struct vel *vels = (struct vel *) glMapBufferRange( GL_SHADER_STORAGE_BUFFER, 0, NUM_PARTICLES * sizeof(struct vel), bufMask );
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    vels[i].vx = Ranf( VXMIN, VXMAX );
    vels[i].vy = Ranf( VYMIN, VYMAX );
    vels[i].vz = Ranf( VZMIN, VZMAX );
    vels[i].vw = 0.;
}
glUnmapBuffer( GL_SHADER_STORAGE_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

20 total items to compute:

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

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

```c
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, posSSbo);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, velSSbo);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 6, colSSbo);
... 
gUseProgram(MyComputeShaderProgram);
glDispatchCompute(NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
... 
gUseProgram(MyRenderingShaderProgram);
```

Using the glslprogram C++ Class to Handle Everything

```c
GLSLProgram Particles, Render; // global variables
... 
Particles.Init();
if( ! valid ) { ... }
Particles.Use(); // compute the particles
Particles.DispatchCompute(NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1);
Particles.UnUse();
... 
Render.Use(); // draw the particles
... 
Render.UnUse();
```

Special Pre-set Variables in the Compute Shader

- `uvec3 gl_NumWorkGroups;`
- `uvec3 gl_WorkGroupID;`
- `uvec3 gl_LocalInvocationID;`
- `uvec3 gl_GlobalInvocationID;`
- `uint gl_LocalInvocationIndex;`

The Particle System Compute Shader -- Setup

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

The Particle System Compute Shader -- The Physics

```c
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?

```c
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 BounceSphere( vec3 p, vec3 v, vec4 s ){
    ... 
}
bool IsInsideSphere( vec3 p, vec4 s ){
    ... 
}
```
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?

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