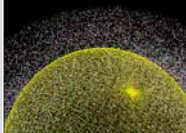




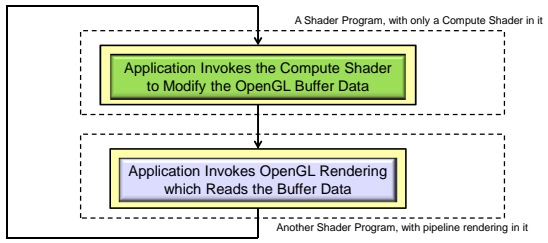
OpenGL Compute Shaders

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University

Oregon State University
Computer Graphics
mjb - August 11, 2012

OpenGL Compute Shader – the Basic Idea



Oregon State University
Computer Graphics
mjb - August 11, 2012

OpenGL Compute Shader – the Basic Idea

Paraphrased from the ARB_compute_shader 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.

Oregon State University
Computer Graphics
mjb - August 11, 2012

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 ever 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.

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

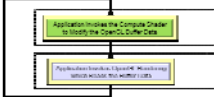
I suspect that a good example of a lighter-weight data-parallel graphics-related application is a **particle system**. This will be shown here in the rest of these notes. I hope I'm right.

Oregon State University
Computer Graphics
mjb - August 11, 2012

If I Know GLSL, What Do I 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. (why?)
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.



More information on items 3 and 4 are coming up . . .

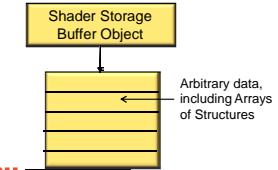
Oregon State University
Computer Graphics
mjb - August 11, 2012

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."

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

Thus, OpenGL 4.3 introduced the **Shader Storage Buffer Object**. This is very cool, and has been needed for a long time!



Shader Storage Buffer Objects are created with arbitrary data (same as other buffer objects), but what is new is that the shaders can read and write them in the same C-like way as they were created, including treating parts of the buffer as an array of structures – perfect for data-parallel computing!

Oregon State University
Computer Graphics
mjb - August 11, 2012

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.

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

DSU Oregon State University Computer Graphics mb - August 11, 2012

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

DSU Oregon State University Computer Graphics mb - August 11, 2012

Setting up the Shader Storage Buffer Objects

```

#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. (is it?)

DSU Oregon State University Computer Graphics mb - August 11, 2012

Setting up the Shader Storage Buffer Objects

```

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
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);
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 same would possibly need to be done for the color shader storage buffer object

DSU Oregon State University Computer Graphics mb - August 11, 2012

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.

#WorkGroups = $\frac{\text{GlobalInvocationSize}}{\text{WorkGroupSize}}$

$5 \times 4 = \frac{20}{4}$

DSU Oregon State University Computer Graphics mb - August 11, 2012

The Data Needs to be Divided into Large Quantities call 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.

#WorkGroups = $\frac{\text{GlobalInvocationSize}}{\text{WorkGroupSize}}$

$5 \times 4 = \frac{20 \times 12}{4 \times 3}$

DSU Oregon State University Computer Graphics mb - August 11, 2012

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

OSU Oregon State University Computer Graphics mb - August 11, 2012

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, posSSbo );
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, velSSbo );
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 6, colSSbo );
...
glUseProgram( MyComputeShaderProgram );
glDispatchCompute( NUM_PARTICLES / WORK_GROUP_SIZE, 1, 1 );
glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );
...
glUseProgram( MyRenderingShaderProgram );
glBindBuffer( GL_ARRAY_BUFFER, posSSbo );
glVertexPointer( 4, GL_FLOAT, 0, (void *)0 );
glEnableClientState( GL_VERTEX_ARRAY );
glDrawArrays( GL_POINTS, 0, NUM_PARTICLES );
glDisableClientState( GL_VERTEX_ARRAY );
glBindBuffer( GL_ARRAY_BUFFER, 0 );
```

OSU Oregon State University Computer Graphics mb - August 11, 2012

Special Pre-set Variables in the Compute Shader

In **uvec3** **gl_NumWorkGroups**; Some numbers as in the *glDispatchCompute* call
const uvec3 **gl_WorkGroupSize**; Some numbers 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 workitems
 In **uint** **gl_LocalInvocationIndex**; 1D representation of the *gl_LocalInvocationID* (used for indexing into a shared array)

```
0 ≤ gl_WorkGroupID ≤ gl_NumWorkGroups - 1
0 ≤ gl_LocalInvocationID ≤ gl_WorkGroupSize - 1
gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID
gl_LocalInvocationIndex = gl_LocalInvocationID * gl_WorkGroupSize.y * gl_WorkGroupSize.x +
gl_LocalInvocationID.y * gl_WorkGroupSize.x +
gl_LocalInvocationID.x
```

OSU Oregon State University Computer Graphics mb - August 11, 2012

The Particle System Compute Shader -- Setup

```
#version 430 compatibility
#extension GL_ARB_compute_shader : enable
#extension GL_ARB_shader_storage_buffer_object : enable;
struct pos
{
    vec4 pxyzw; // positions
};
struct vel
{
    vec4 vxyzw; // velocities
};
struct color
{
    vec4 crgba; // colors
};
layout( std140, binding=4 ) buffer Pos {
    struct pos Positions[ ]; // array of structures
};
layout( std140, binding=5 ) buffer Vel {
    struct vel Velocities[ ]; // array of structures
};
layout( std140, binding=6 ) buffer Col {
    struct color Colors[ ]; // array of structures
};
layout( local_size_x = 128, local_size_y = 1, local_size_z = 1 ) in;
```

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.

OSU Oregon State University Computer Graphics mb - August 11, 2012

The Particle System Compute Shader -- The Physics

```
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
```

$$p' = p + v \cdot t + \frac{1}{2} G \cdot t^2$$

$$v' = v + G \cdot t$$

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

OSU Oregon State University Computer Graphics mb - August 11, 2012

The Particle System Compute Shader -- How About Introducing a Bounce?

```
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 );
}
```

OSU Oregon State University Computer Graphics mb - August 11, 2012

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 ].pxyzw.xyz;
vec3 v = Velocities[ gid ].vxyzw.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 ].pxyzw.xyz = pp;
Velocities[ gid ].vxyzw.xyz = vp;

```

$$p' = p + v \cdot t + \frac{1}{2} G \cdot t^2$$

$$v' = v + G \cdot t$$

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...

OSU Oregon State University Computer Graphics mb - August 11, 2012

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

OSU Oregon State University Computer Graphics mb - August 11, 2012

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:

```

layout( std140, binding=6 ) buffer Col {
    struct color Colors[ ];
};

layout( shared ) vec4 rgba[ gl_WorkGroupSize.x ];

uint gid = gl_GlobalInvocationID.x;
uint lid = gl_LocalInvocationID.x;

rgba[ lid ] = Colors[ gid ].rgba;

memory_barrier_shared();

<< operate on the rgba array elements >>

Colors[ gid ].rgba = rgba[ lid ];

```

OSU Oregon State University Computer Graphics mb - August 11, 2012

Other Useful Stuff – Getting Information Back Out

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:

Application Program

```

gGenBuffers( 1, &countBuffer);
gBindBufferBase( GL_ATOMIC_COUNTER_BUFFER, 7, countBuffer);
gBufferData( GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), NULL, GL_DYNAMIC_DRAW);

GLuint zero = 0;
gBufferSubData( GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &zero);

```

Compute Shader

```

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 );
}

```

Application Program

```

gBindBuffer( GL_SHADER_STORAGE_BUFFER, countBuffer );
GLuint *ptr = (GLuint *) gMapBuffer( GL_SHADER_STORAGE_BUFFER, GL_READ_ONLY );
GLuint bounceCount = *ptr;
gUnmapBuffer( GL_SHADER_STORAGE_BUFFER );
printf( stderr, "%d bounces\n", bounceCount );

```

OSU Oregon State University Computer Graphics mb - August 11, 2012

Other Useful Stuff – Getting Information Back Out

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:

Application Program

```

gGenBuffers( 1, &particleBuffer);
gBindBufferBase( GL_ATOMIC_COUNTER_BUFFER, 8, particleBuffer);
gBufferData( GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), NULL, GL_DYNAMIC_DRAW);

GLuint zero = 0;
gBufferSubData( GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &zero);

```

Fragment Shader

```

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

atomicCounterIncrement( particleCount );

```

Application Program

```

gBindBuffer( GL_SHADER_STORAGE_BUFFER, particleBuffer );
GLuint *ptr = (GLuint *) gMapBuffer( GL_SHADER_STORAGE_BUFFER, GL_READ_ONLY );
GLuint particleCount = *ptr;
gUnmapBuffer( GL_SHADER_STORAGE_BUFFER );
if( particleCount == 0 )
    DoAnimate = false; // stop animating

```

OSU Oregon State University Computer Graphics mb - August 11, 2012

Other Useful Stuff – Getting Information Back Out

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:

Application Program

```

gBindBufferBase( GL_ATOMIC_COUNTER_BUFFER, 7, countBuffer);
GLuint zero = 0;
gBufferSubData( GL_ATOMIC_COUNTER_BUFFER, 0, sizeof(GLuint), &zero);

```

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

Application Program

```

gBindBufferBase( GL_ATOMIC_COUNTER_BUFFER, 7, countBuffer);
GLuint zero = 0;
gClearBufferData( GL_ATOMIC_COUNTER_BUFFER, GL_R32UI, GL_RED, GL_UNSIGNED_INT, &zero );

```

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

OSU Oregon State University Computer Graphics mb - August 11, 2012