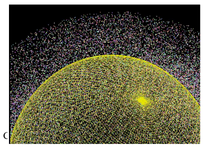
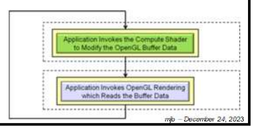
1

OpenGL Compute Shaders

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

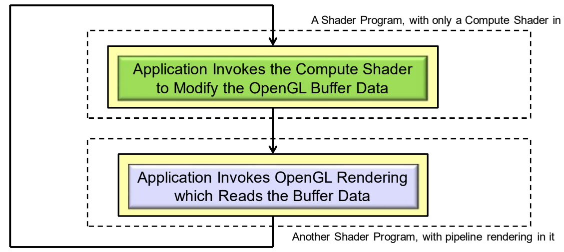




compute.shader.pptx

2

OpenGL Compute Shader – the Basic Idea



3

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.

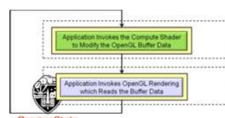
In most respects, a Compute Shader is identical to all other OpenGL shaders, with similar status, uniforms, and other such properties. It has access to many of the same data as all other shader types, such as textures, image textures, atomic counters, and so on. However, the Compute Shader has no predefined 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.

4

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.



More information on item 4 is coming up . . .

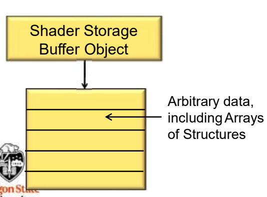
5

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.

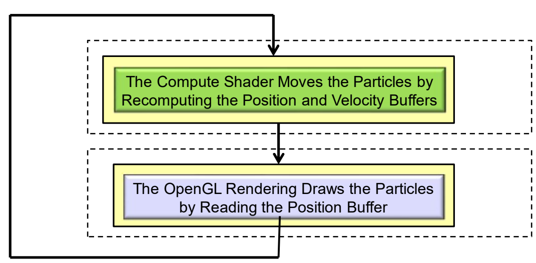
Thus, OpenGL 4.3 introduced the **Shader Storage Buffer Object**. This is very cool, and has been needed for a long time!



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!

6

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



Setting up the Shader Storage Buffer Objects in Your C/C++ Program 7

```
#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.

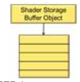
Computer Graphics

mjb – December 24, 2023

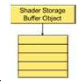
7

Setting up the Shader Storage Buffer Objects in Your C Program 8

```
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 );
```



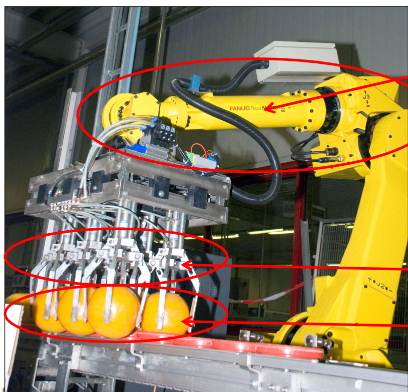
The same would possibly need to be done for the color shader storage buffer object

University
Computer Graphics

mjb – December 24, 2023

8

A Mechanical Equivalent of a GPU 9



"Streaming Multiprocessor"

"CUDA Cores"

"Data"

University
Computer Graphics

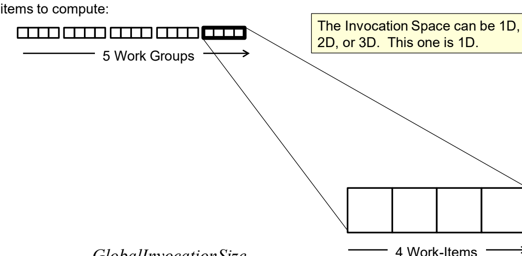
http://news.cision.com

mjb – December 24, 2023

9

The Data Needs to be Divided into Large Quantities call *Work-Groups*, each of which is further Divided into Smaller Units Called *Work-Items* 10

20 total items to compute:



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

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

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

Oregon State
University
Computer Graphics

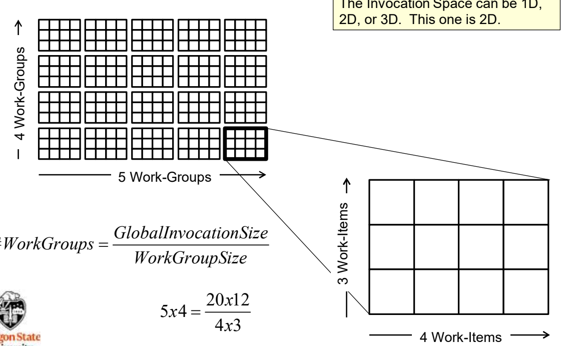
mjb – December 24, 2023

10

The Data Needs to be Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items 11

20x12 (=240) total items to compute:

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



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

$$5x4 = \frac{20x12}{4x3}$$

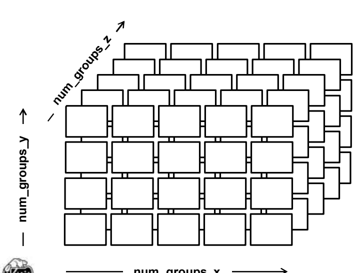
Oregon State
University
Computer Graphics

mjb – December 24, 2023

11

Running the Compute Shader from the Application 12

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

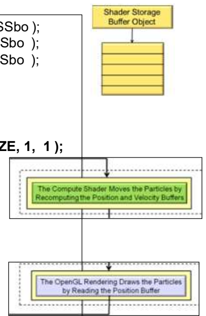
Oregon State
University
Computer Graphics

mjb – December 24, 2023

12

Invoking the Compute Shader in Your C Program


```
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 );
```



Using the glslprogram C++ Class to Handle Everything

The Setup:

```
GLSLProgram Particles, Render;        // global variables
. . .

Particles.Init( );
bool valid = Particles.Create( "particles.cs" );
if( ! valid ) { . . . }
```

The Use:

```
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

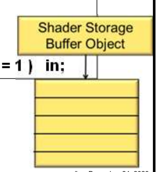
in	uvec3	**gl_NumWorkGroups ;**	Same numbers as in the *glDispatchCompute* call
const	uvec3	**gl_WorkGroupSize ;**	Same numbers as in the *layout* local_size_*
in	uvec3	**gl_WorkGroupID ;**	Which workgroup this thread is in
in	uvec3	**gl_LocalInvocationID ;**	Where this thread is in the current workgroup
in	uvec3	**gl_GlobalInvocationID ;**	Where this thread is in *all* the work items
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.z * gl_WorkGroupSize.y * gl_WorkGroupSize.x  +
                           gl_LocalInvocationID.y * gl_WorkGroupSize.x                       +
                           gl_LocalInvocationID.x
```


The Particle System Compute Shader -- Setup


```
#version 430 compatibility
#extension GL_ARB_compute_shader :                   enable
#extension GL_ARB_shader_storage_buffer_object :     enable;

layout( std140, binding=4 ) buffer  Pos
{
        vec4  Positions[  ];          // array of structures
};

layout( std140, binding=5 ) buffer  Vel
{
        vec4  Velocities[  ];         // array of structures
};

layout( std140, binding=6 ) buffer  Col
{
        vec4  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.

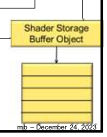


The Particle System Compute Shader – The Physics


```
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



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;
```

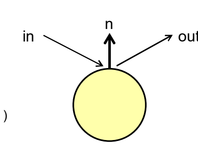
$$p' = p + v \cdot t + \frac{1}{2} G \cdot t^2$$

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



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 );
}
```



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

19

```
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;
```

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

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

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…

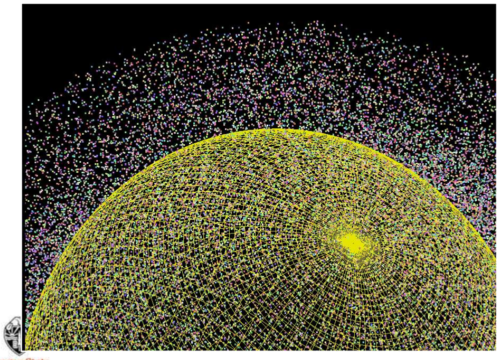
Oregon State University Computer Graphics

mjb – December 24, 2023

19

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

20



Oregon State University Computer Graphics

mjb – December 24, 2023

20