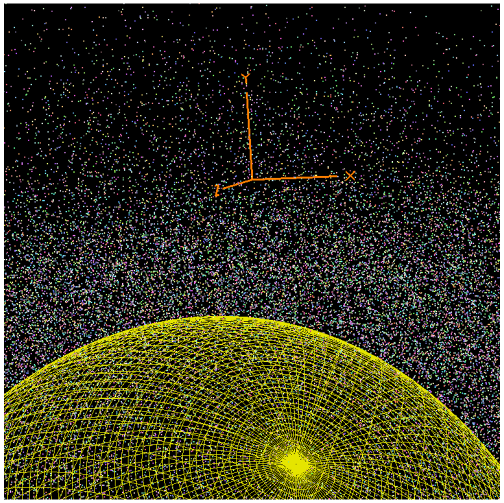
Combining GPU Data-Parallel Computing with OpenGL

Mike Bailey

mjb@cs.oregonstate.edu
Oregon State University

ACMSIGGRAPH



mjb – May 7, 2013

Mike Bailey

- Professor of Computer Science, Oregon State University

- Has worked at Sandia Labs, Purdue University, Megatek, San Diego Supercomputer Center (UC San Diego), and OSU

- Has taught over 5,000 students in his classes

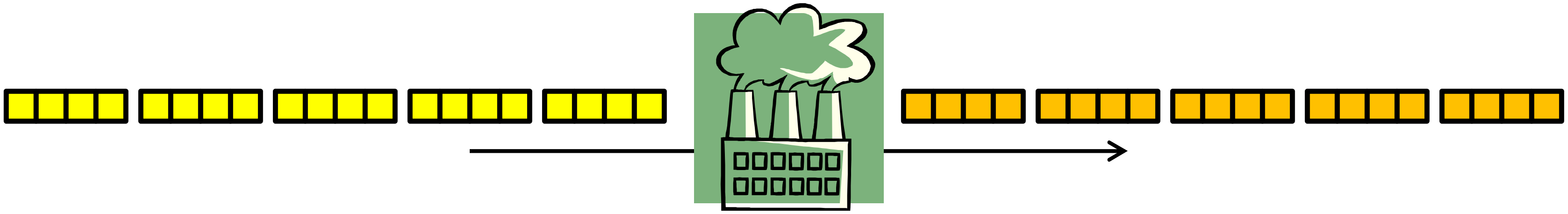
- mjb@cs.oregonstate.edu

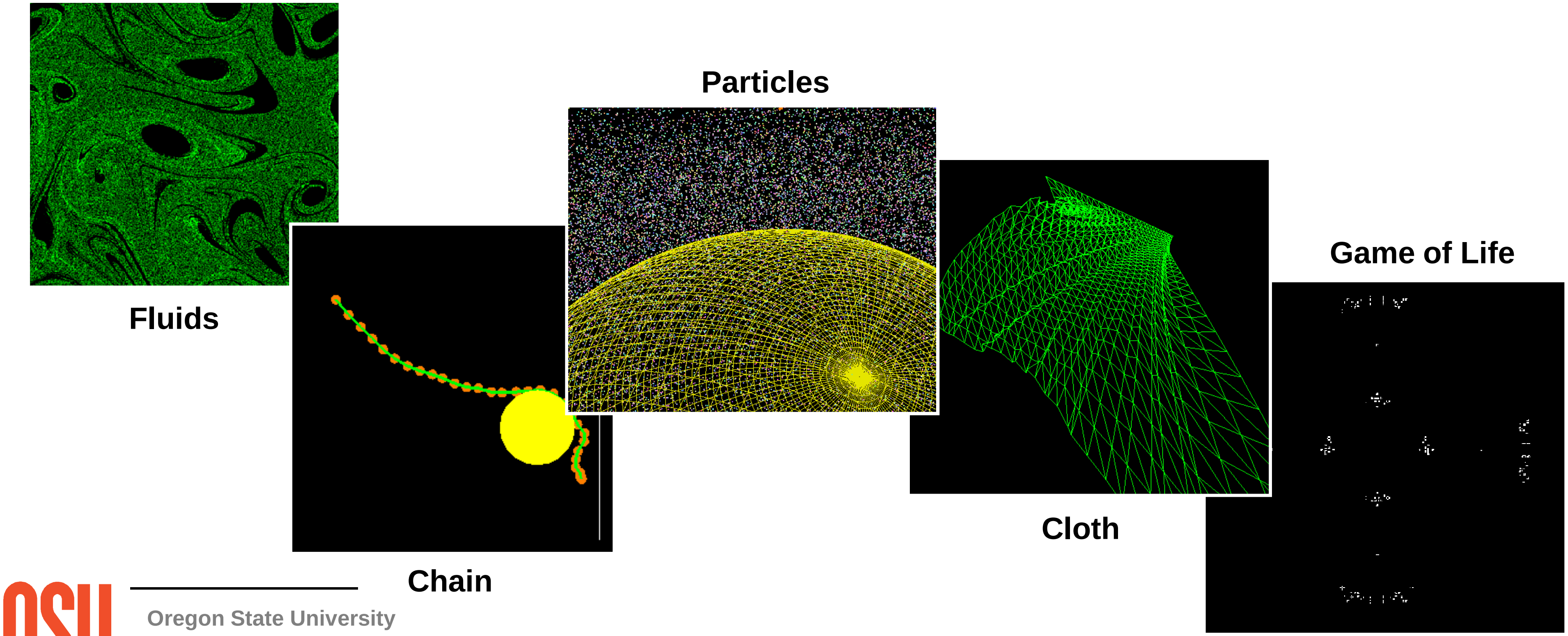
Topics

- **Introduction to Data Parallel Computing**
- **Introduction to OpenGL Vertex Buffers**
- **OpenGL Compute Shaders**
- **OpenCL**
- **References**

Introduction to Data Parallel Computing

The scenario – many pieces of data need to undergo the same operation:





Fluids

Chain

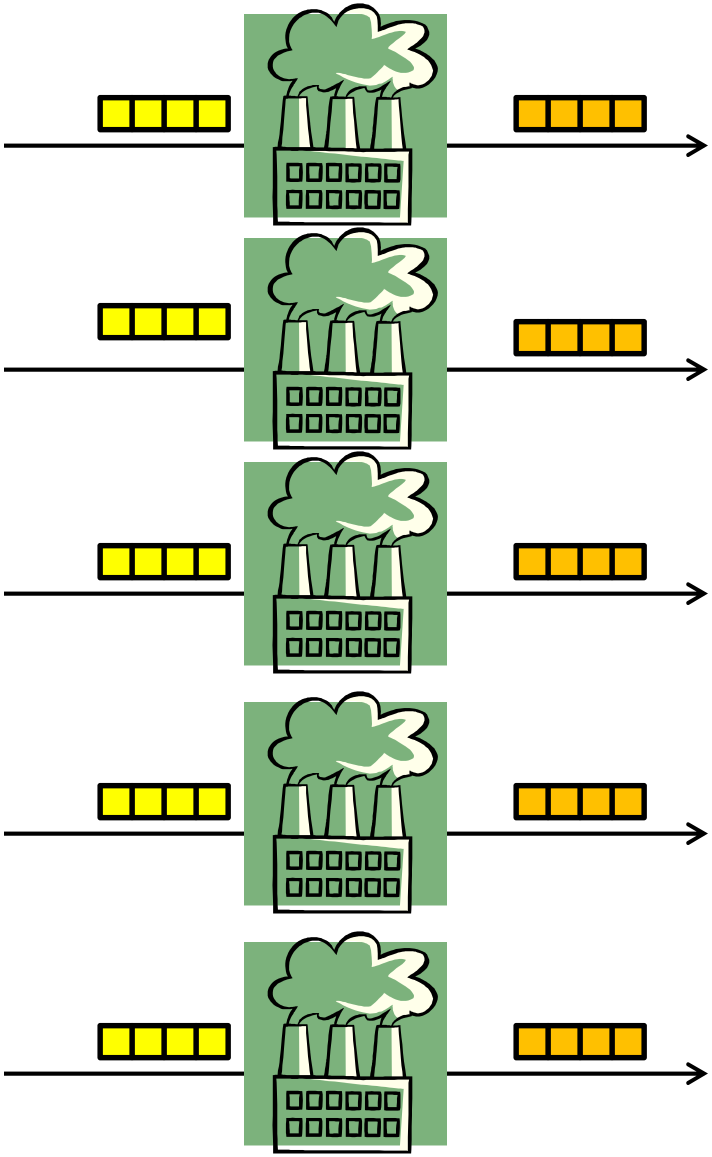
Particles

Cloth

Game of Life

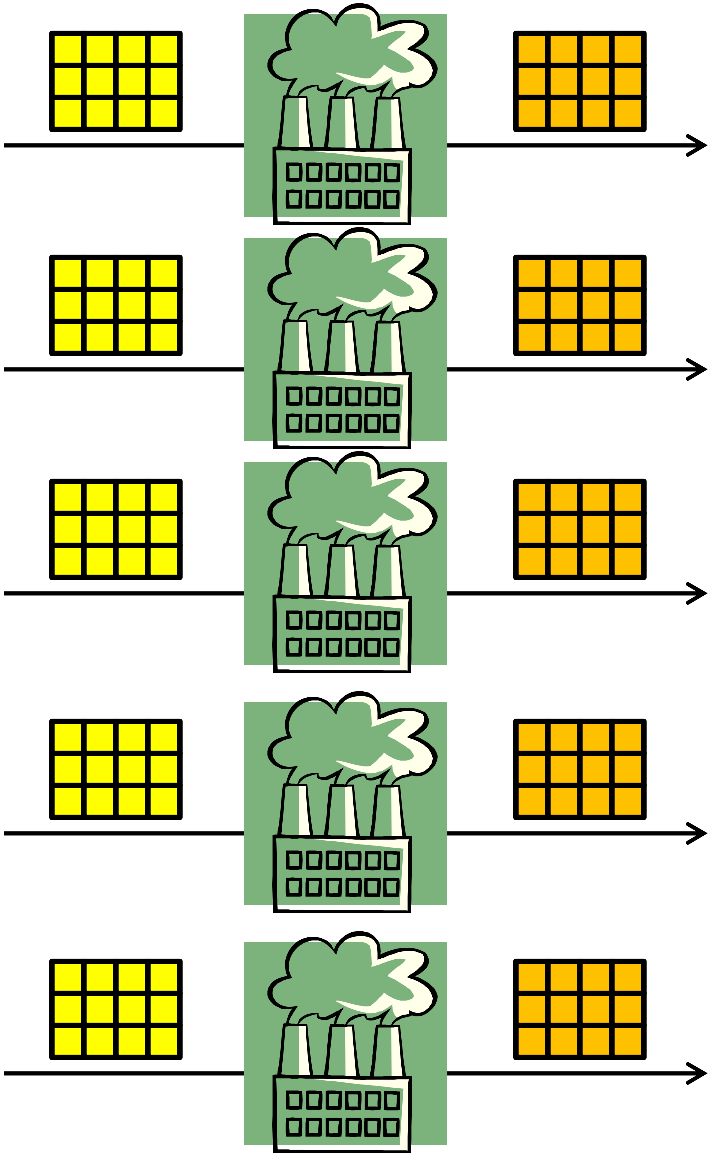
Introduction to Data Parallel Computing

The scenario – many pieces of data need to undergo the same operation:



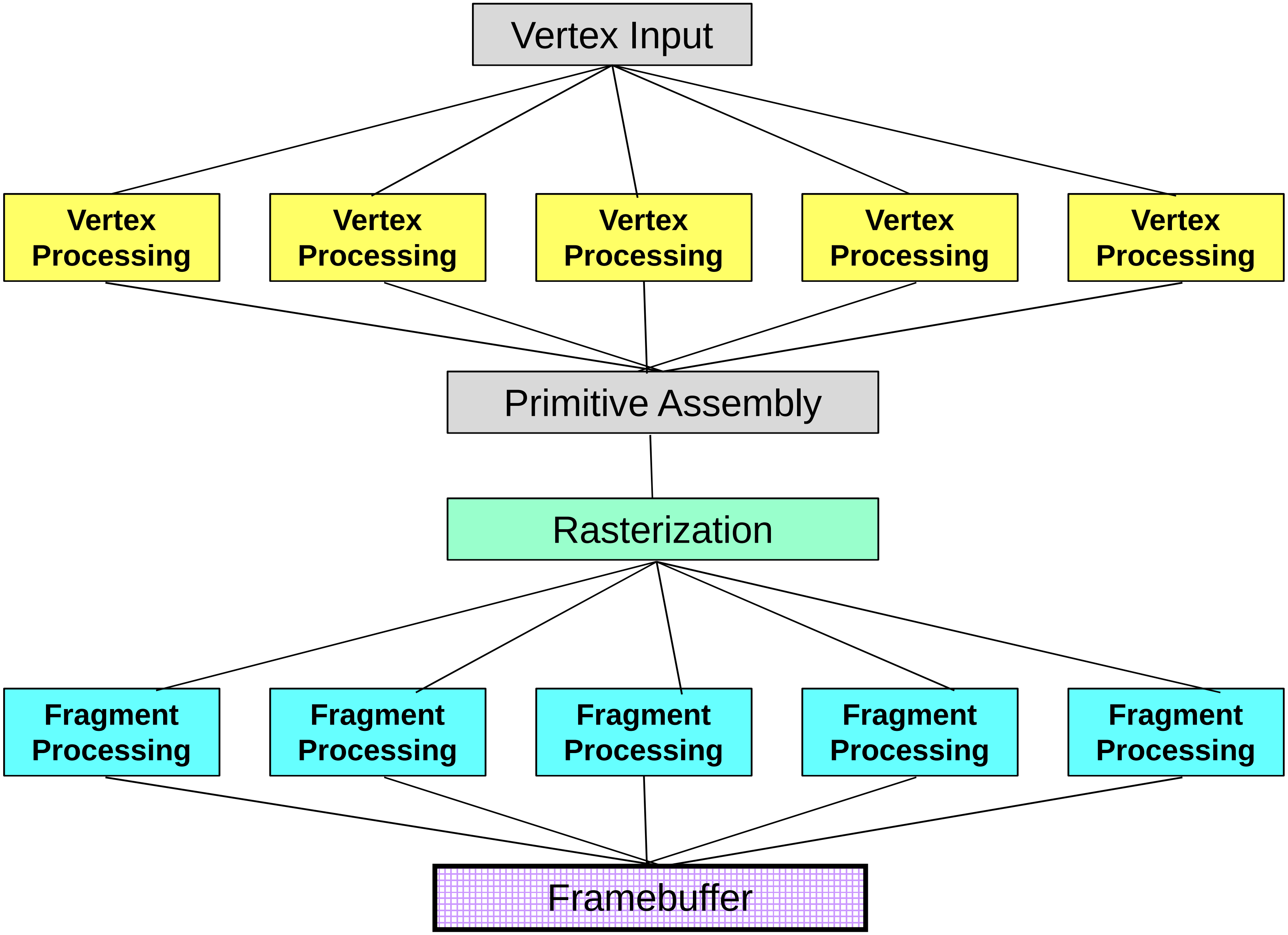
Introduction to Data Parallel Computing

The scenario – many pieces of data need to undergo the same operation:



For your convenience, the data can be thought of as 1D, 2D, or 3D.

Note: GPUs are designed to Handle Data Parallel Computing Well



Vertex Input

Vertex Processing **Vertex Processing** **Vertex Processing** **Vertex Processing** **Vertex Processing**

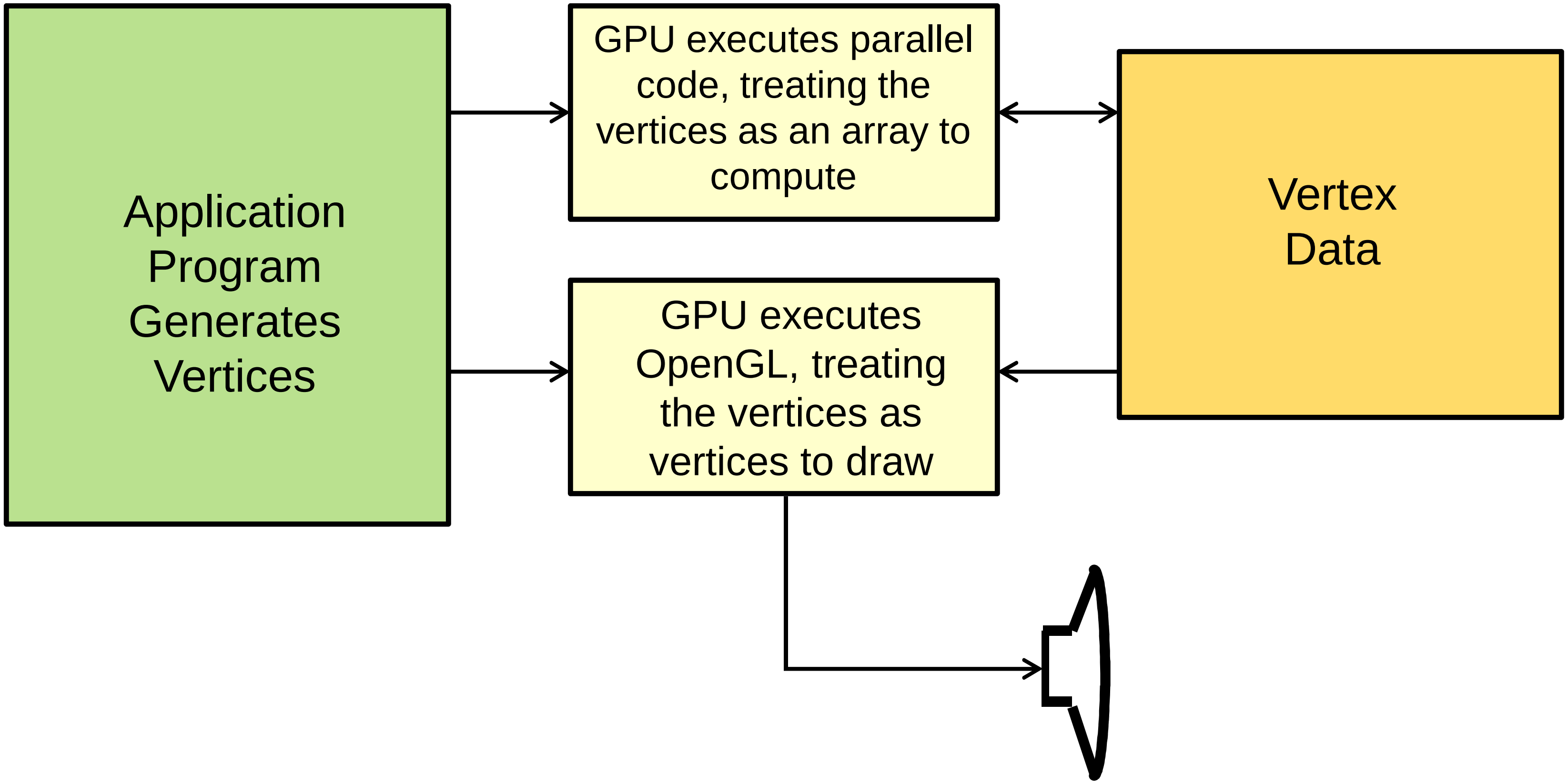
Primitive Assembly

Rasterization

Fragment Processing **Fragment Processing** **Fragment Processing** **Fragment Processing** **Fragment Processing**

Framebuffer

Data Parallel / OpenGL Vertex Interoperability: The Basic Idea



Application Program Generates Vertices

GPU executes parallel code, treating the vertices as an array to compute

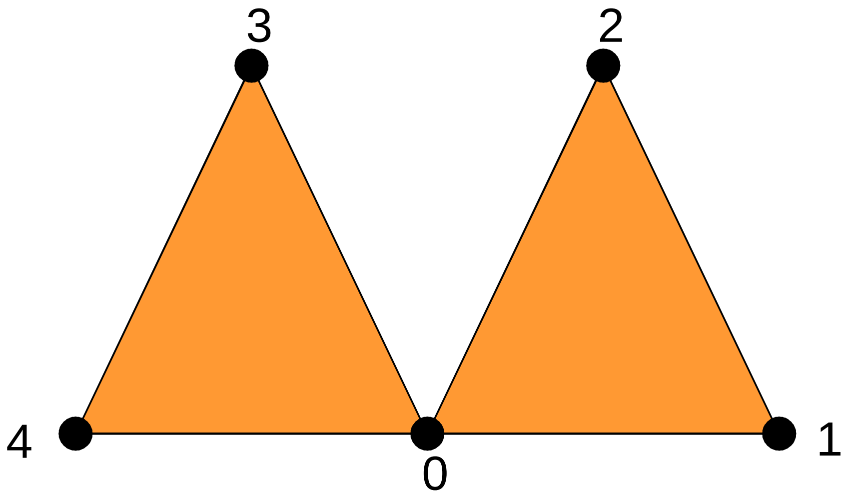
Vertex Data

GPU executes OpenGL, treating the vertices as vertices to draw

In the Beginning of OpenGL …

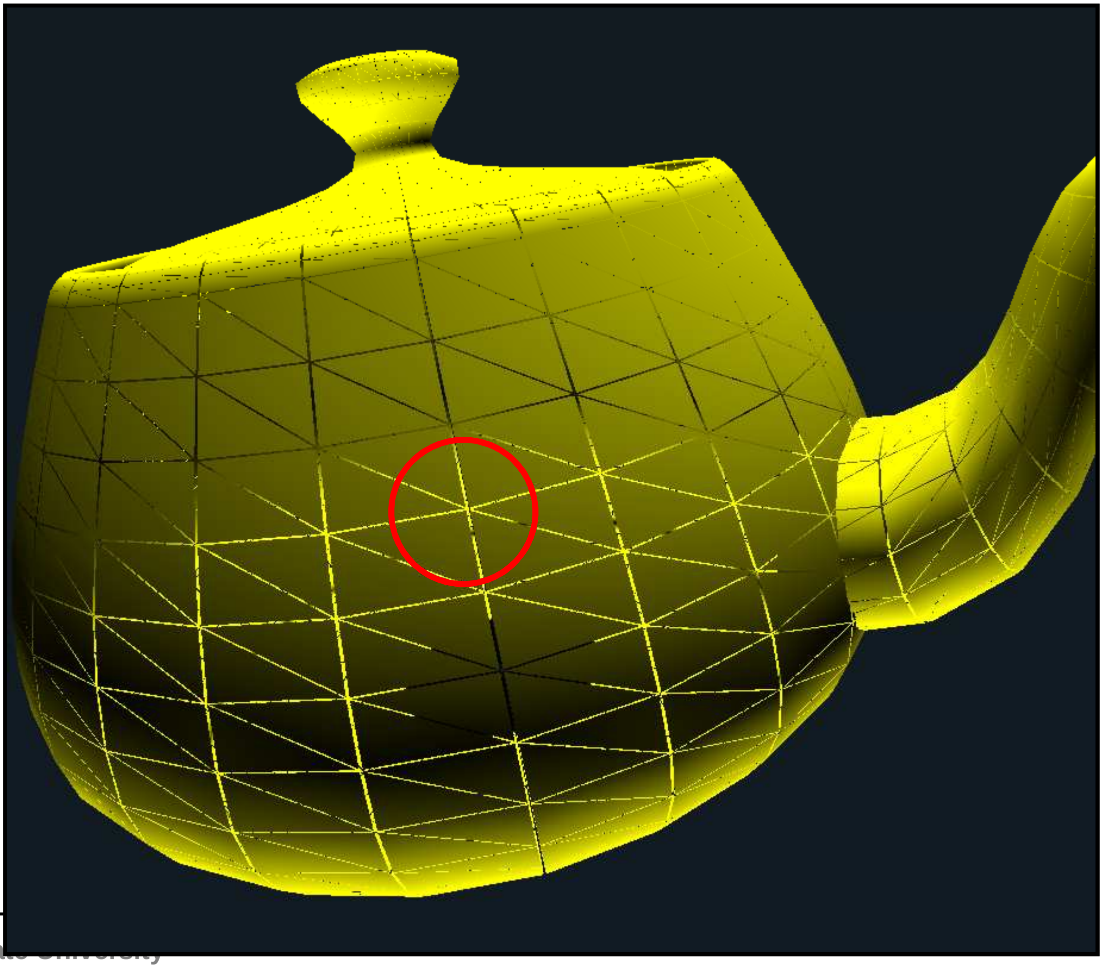
You listed the vertices individually:

```
glBegin( GL_TRIANGLES );
                glVertex3f( x0, y0, z0 );
                glVertex3f( x1, y1, z1 );
                glVertex3f( x2, y2, z2 );

                glVertex3f( x0, y0, z0 );
                glVertex3f( x3, y3, z3 );
                glVertex3f( x4, y4, z4 );
glEnd( );
```



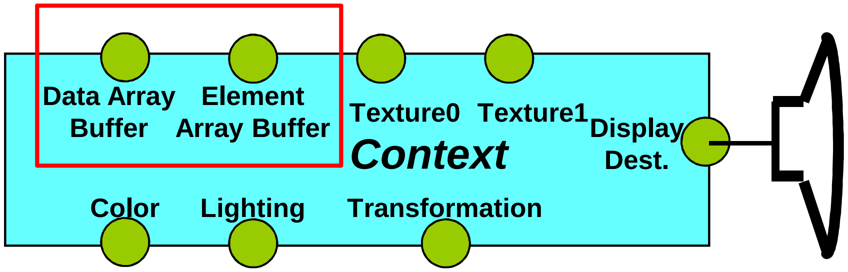
Then Someone Noticed That You Were Transmitting and Processing Each Vertex Several Times…

For example:



A Little Background -- the OpenGL *Rendering Context*

The OpenGL Rendering Context contains all the characteristic information necessary to produce an image from geometry. This includes transformations, colors, lighting, textures, where to send the display, etc.



Some of these characteristics have a default value (e.g., lines are white, the display goes to the screen) and some have nothing (e.g., no textures exist)

More Background – What is an OpenGL "Object"?

An OpenGL Object is pretty much the same as a C++, C#, or Java object: it encapsulates a group of data items and allows you to treat them as a unified whole. For example, a Vertex Buffer Object *could* be defined in C++ by:

```
class VertexBufferObject
{
                enum dataType;     // int, float, …
                void *memStart;
                int memSize;
};
```

Then, you could create any number of Vertex Buffer Object instances, each with its own characteristics encapsulated within it. When you want to make that combination current, you just need to bring in **("bind")** that entire object. When you bind an object, all of its information comes with it.

More Background – How do you Create an OpenGL "Object"?

In C++, objects are pointed to by their address.

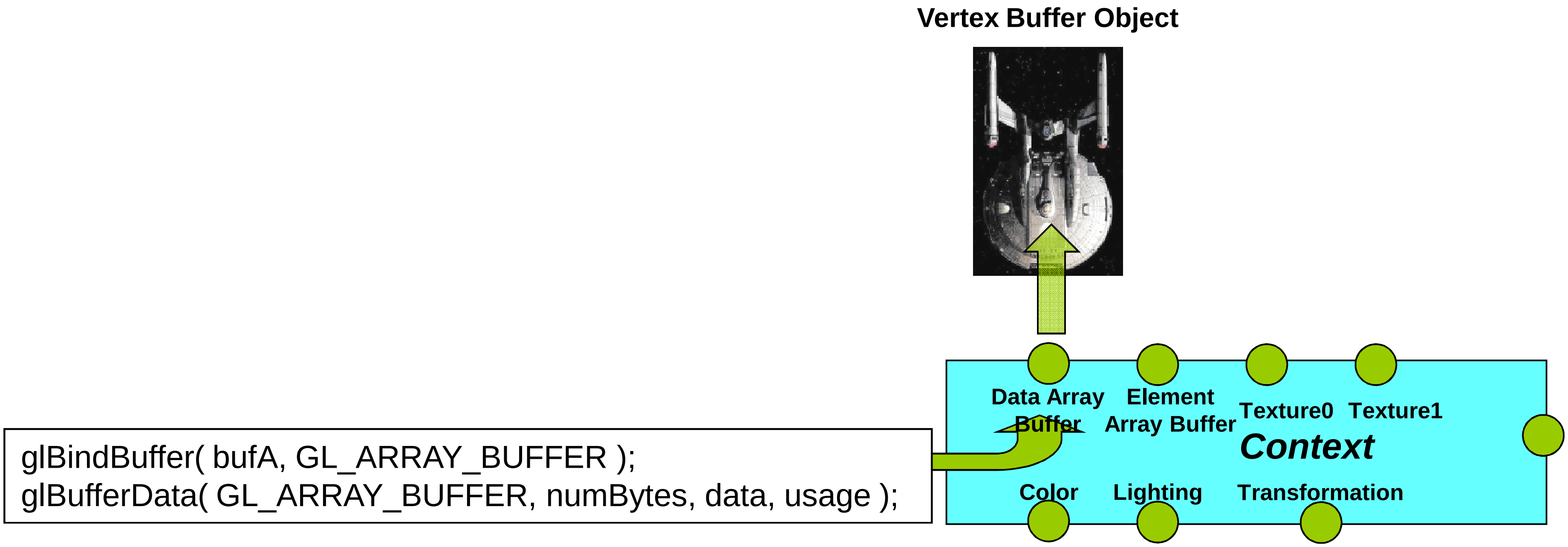
In OpenGL, objects are pointed to by an unsigned integer handle. You can assign a value for this handle yourself (not recommended), or have OpenGL generate one for you that is guaranteed to be unique. For example:

```
GLuint bufA;

glGenBuffers( 1, &bufA );
```

This doesn't actually allocate memory for the buffer object yet, it just acquires a unique handle. To allocate memory, you need to bind this handle to the Context.

More Background -- "Binding" to the Context

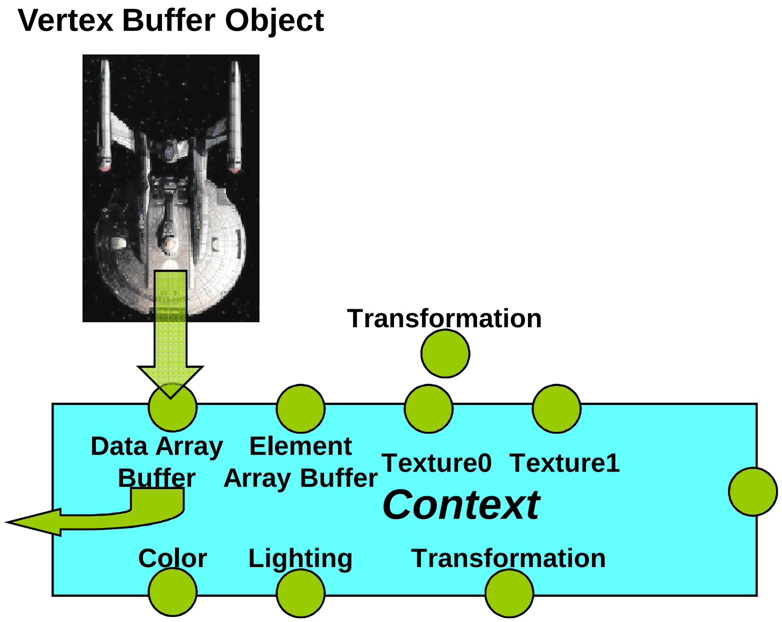
The OpenGL term "binding" refers to "attaching" or "docking" (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will "flow" through the Context into the object.



```
glBindBuffer( bufA, GL_ARRAY_BUFFER );
glBufferData( GL_ARRAY_BUFFER, numBytes, data, usage );
```

More Background -- "Binding" to the Context

When you want to *use* that Vertex Buffer Object, just bind it again. All of the characteristics will then be active, just as if you had specified them again.



```
glBindBuffer( bufA, GL_ARRAY_BUFFER );
```

Vertex Buffers: Putting Data in the Buffer Object

```
glBufferData( type,  numBytes,  data,  usage );
```

type is the type of buffer object this is:
GL_ARRAY_BUFFER to store floating point vertices, normals, colors, and texture coordinates

GL_ELEMENT_ARRAY_BUFFER to store integer vertex indices to connect for drawing

numBytes is the number of bytes to store in all. Not the number of numbers, but the number of *bytes*!

data is the memory address of (i.e., pointer to) the data to be transferred to the graphics card. This can be NULL, and the data can be transferred later.

Vertex Buffers: Putting Data in the Buffer Object

`glBufferData( type,  numbytes,  data,  usage );`

usage is a hint as to how the data will be used: GL_xxx_yyy

where xxx should be one of:

STATIC	this buffer will be written seldom ***from the CPU***
DYNAMIC	this buffer will be written often ***from the CPU***
STATIC	this buffer will be written often ***from the GPU***

and yyy should be:

DRAW	this buffer will be used for drawing

Vertex Buffers: Step #1 – Fill the Arrays

```
GLfloat  Vertices[  ][3] =
{
                { 1., 2., 3. },
                { 4., 5., 6. },
                . . .
};
```

Vertex Buffers: Step #2 – Create the Buffers and Fill Them

```
glGenBuffers( 1, &bufA );


glBindBuffer( bufA, GL_ARRAY_BUFFER );

glBufferData( GL_ARRAY_BUFFER, 3*sizeof(float)*numVertices, Vertices, GL_STATIC_DRAW );
```

Vertex Buffers: Step #3 – Activate the Array Types That You Will Use

glEnableClientState(type)

where *type* can be any of:

```
GL_VERTEX_ARRAY
GL_COLOR_ARRAY
GL_NORMAL_ARRAY
GL_SECONDARY_COLOR_ARRAY
GL_TEXTURE_COORD_ARRAY
```

- Call this as many times as you need to enable all the arrays that you will need.
- There are other types, too.
- To deactivate a type, call:

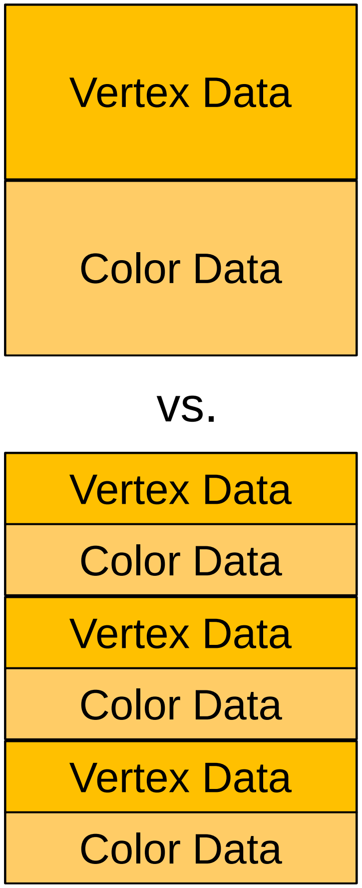
glDisableClientState(type)

Vertex Buffers: Step #4 – To Draw, First Bind the Buffers

```
glBindBuffer( bufA, GL_ARRAY_BUFFER );

glBindBuffer( bufB, GL_ELEMENT_ARRAY_BUFFER );
```

Vertex Buffers: Step #5 – Specify the Data

```
glVertexPointer( size, type, stride, rel_address);
glColorPointer( size, type, stride, rel_address);
glNormalPointer( type, stride, rel_address);
glSecondaryColorPointer( size, type, stride, rel_address);
glTexCoordPointer( size, type, stride, rel_address);
```



size is the spatial dimension, and can be: 2, 3, or 4

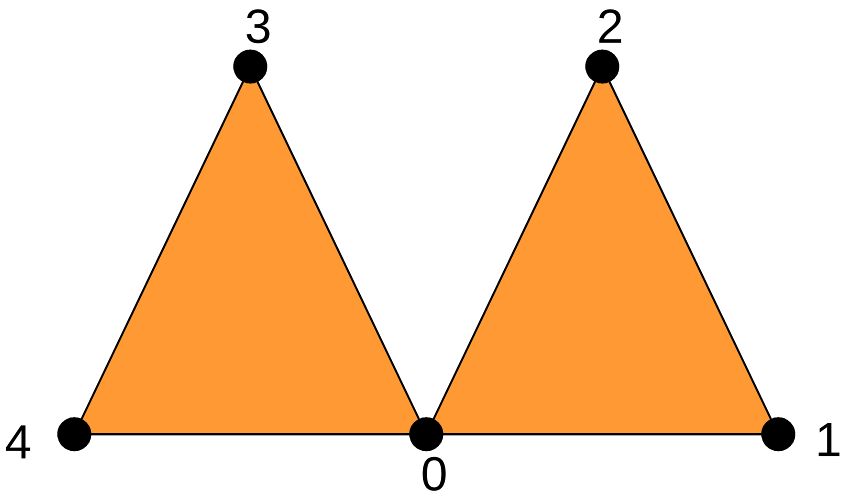
type can be:

- GL_SHORT
- GL_INT
- GL_FLOAT
- GL_DOUBLE

stride is the byte offset between consecutive entries in the array (0 means tightly packed)

rel_address, the 4th argument, is the relative byte address from the start of the buffer where the first element of this part of the data lives. Most of the time you use *(void *)0*

Vertex Buffers: Step #6 – Specify the Connections



If the vertices are not in order:

```
GLuint TriIndices[  ][3] =
{
                { 0, 1, 2 },
                { 0, 3, 4 }
};
glDrawElements( GL_TRIANGLES, 6, GL_UNSIGNED_INT, TriIndices );
```

If all the vertices are in order:

```
glDrawArrays( GL_TRIANGLES, 0, 6 );
```

Vertex Buffers: Writing Data Directly into a Vertex Buffer

Map the buffer from GPU memory into the memory space of the application:

```
glBindBuffer( bufA, GL_ARRAY_BUFFER );
glBufferData( GL_ARRAY_BUFFER, 3*sizeof(float)*numVertices, NULL, GL_STATIC_DRAW );

float * vertexArray = glMapBuffer( GL_ARRAY_BUFFER,  usage );
```

usage is an indication how the data will be used:

GL_READ_ONLY	the vertex data will be read from, but not written to
GL_WRITE_ONLY	the vertex data will be written to, but not read from
GL_READ_WRITE	the vertex data will be read from *and* written to

You can now use *vertexArray[ ]* like any other floating-point array.

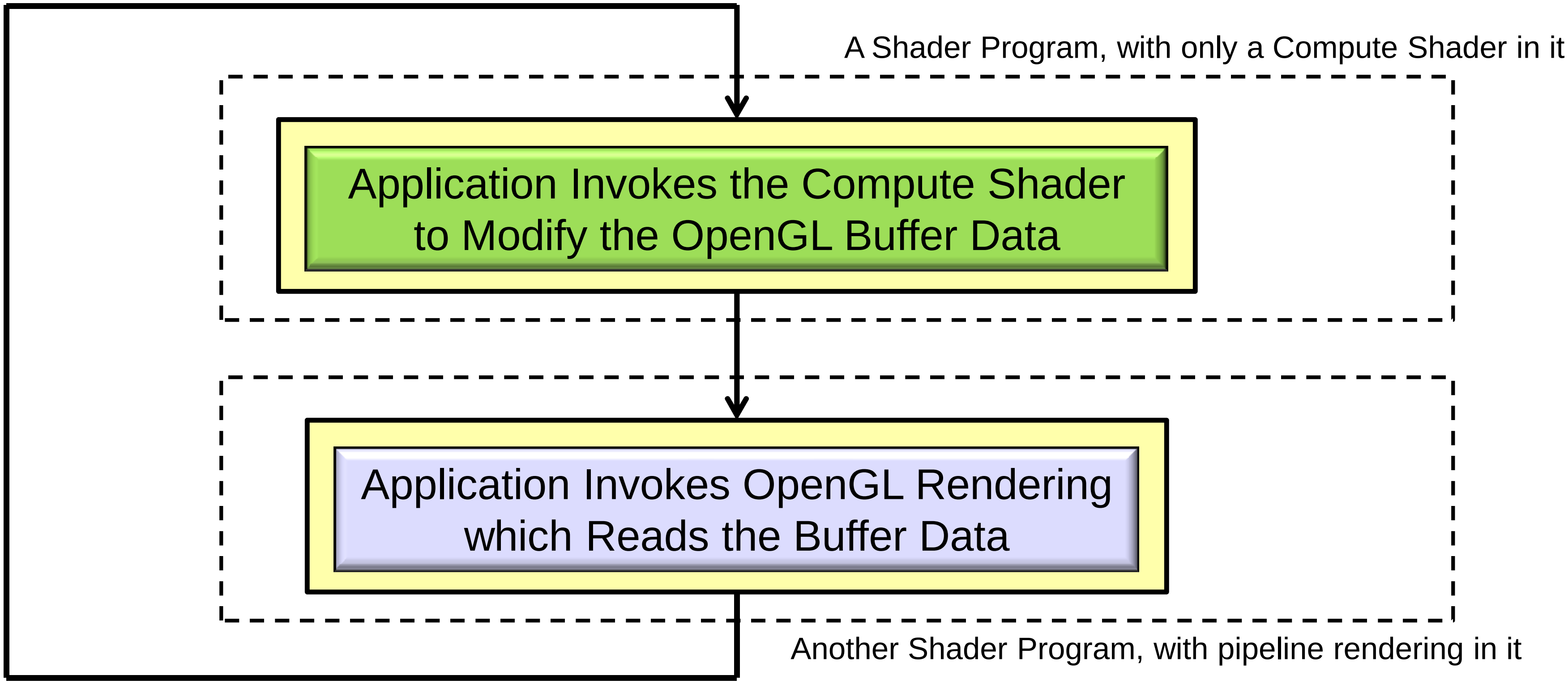
When you are done, be sure to call:

```
glUnMapBuffer( GL_ARRAY_BUFFER );
```

OpenGL Compute Shaders

mjb – May 7, 2013

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

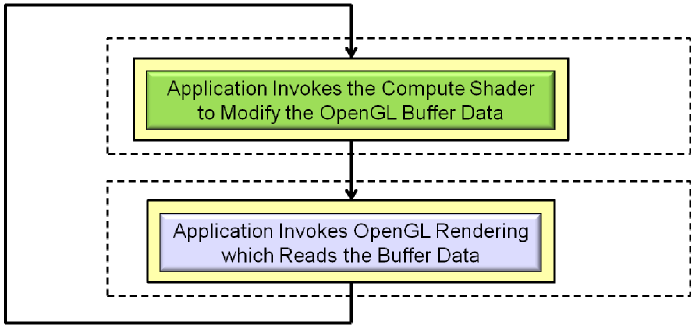
Application Invokes OpenGL Rendering which Reads the Buffer Data

Another Shader Program, with pipeline rendering in it

If I Know GLSL, What Do I Need to Do Differently to Write a Compute Shader?

Not much:

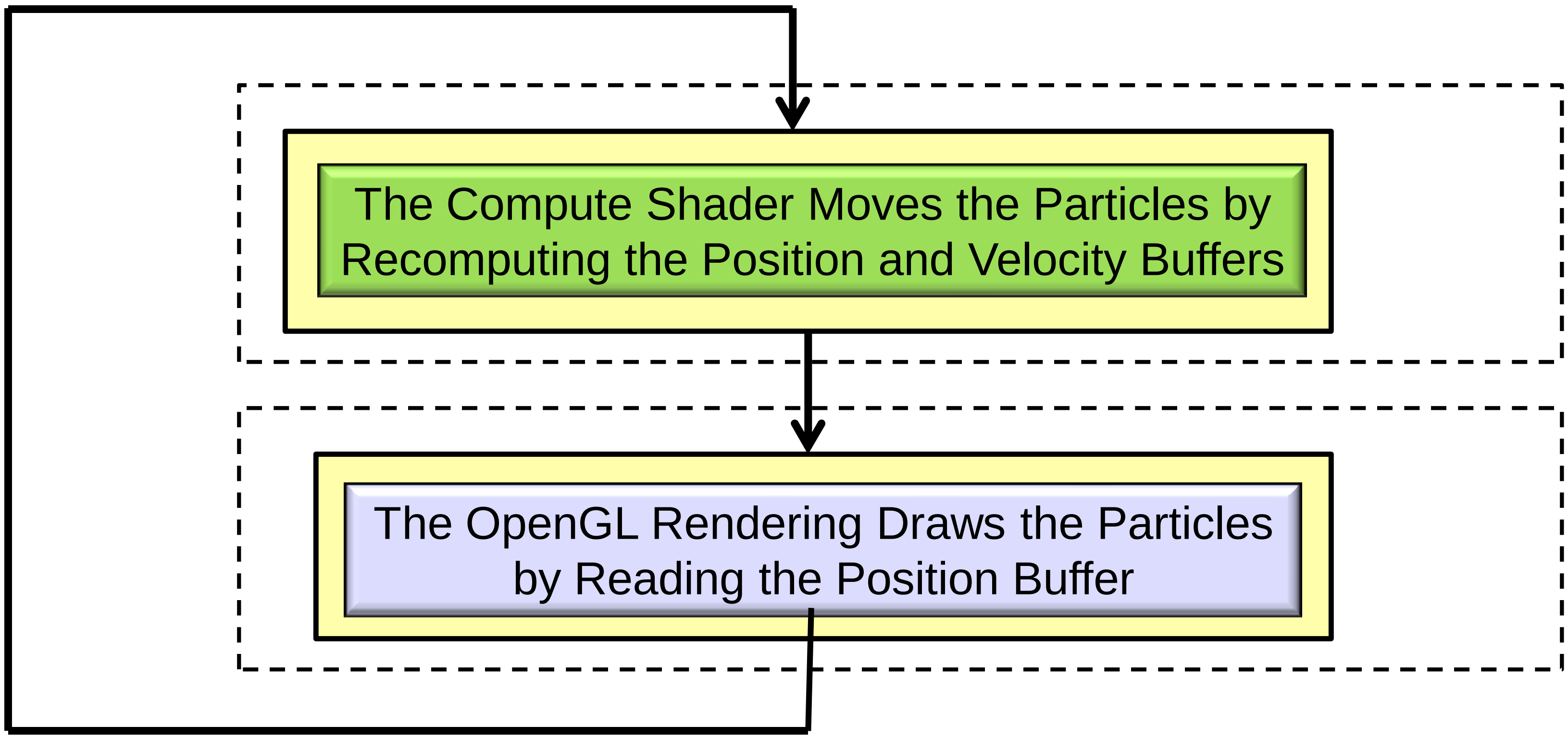
1. A Compute Shader is created just like any other GLSL shader, except that its type is GL_COMPUTE_SHADER (duh…). You compile it and link it just like any other GLSL shader program.
2. A Compute Shader must be in a shader program all by itself. There cannot be vertex, fragment, etc. shaders in there with it.
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.


Application Invokes the Compute Shader to Modify the OpenGL Buffer Data

Application Invokes OpenGL Rendering which Reads the Buffer Data

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

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

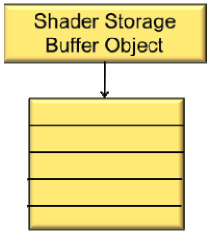
Setting up the Shader Storage Buffer Objects in Your C Program

```
#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;
```

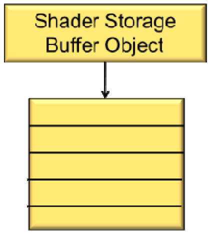
Note that .w and .vw are not actually needed. But, by making these structure sizes a multiple of 4 floats, it doesn't matter if they are declared with the std140 or the std430 qualifier. I think this is a good thing. (is it?)

Setting up the Shader Storage Buffer Objects in Your 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 );
```

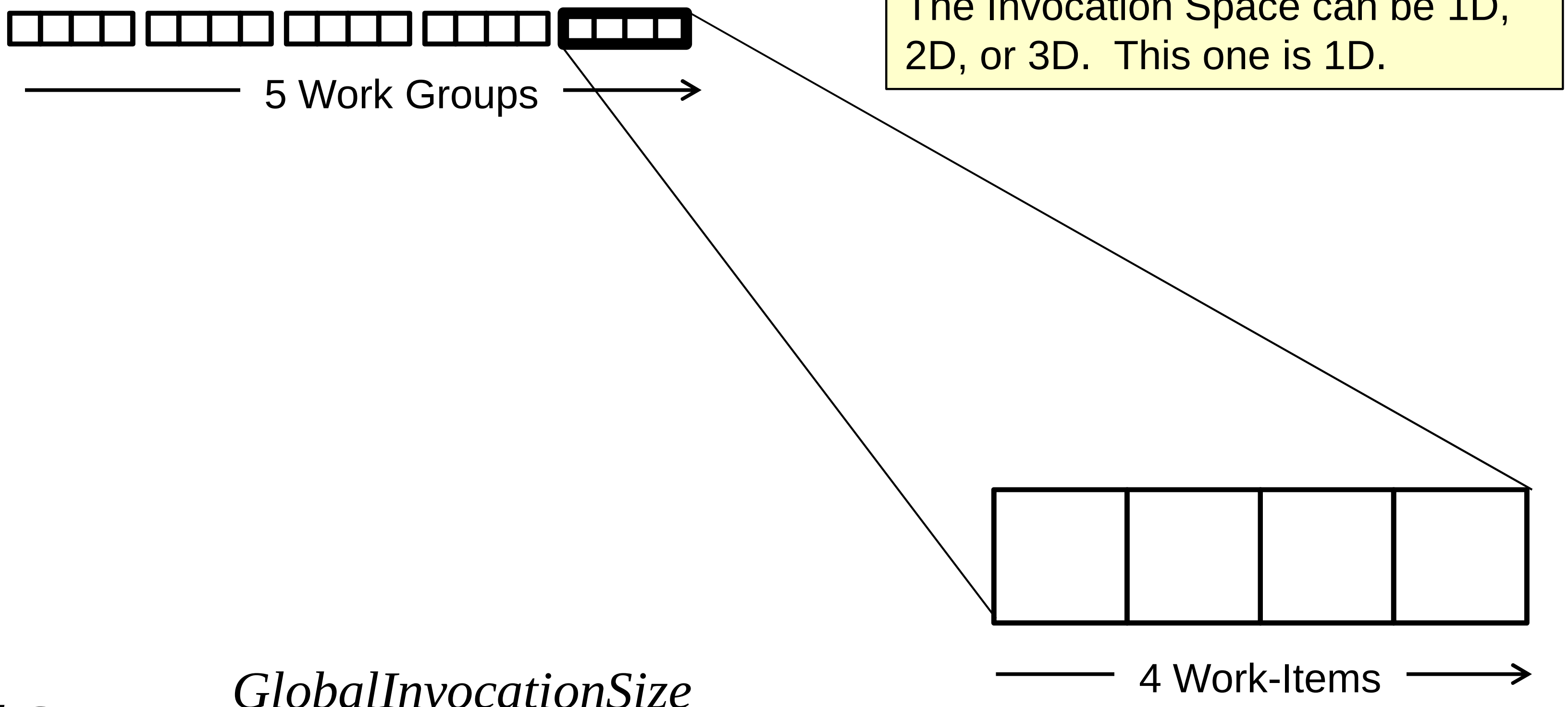


The same would possibly need to be done for the color shader storage buffer object

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:

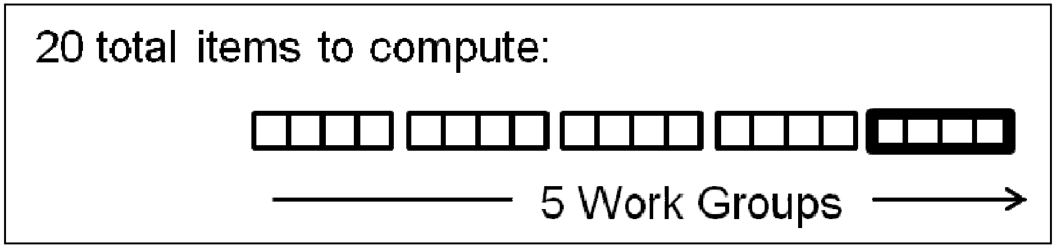
The Invocation Space can be 1D, 2D, or 3D. This one is 1D.

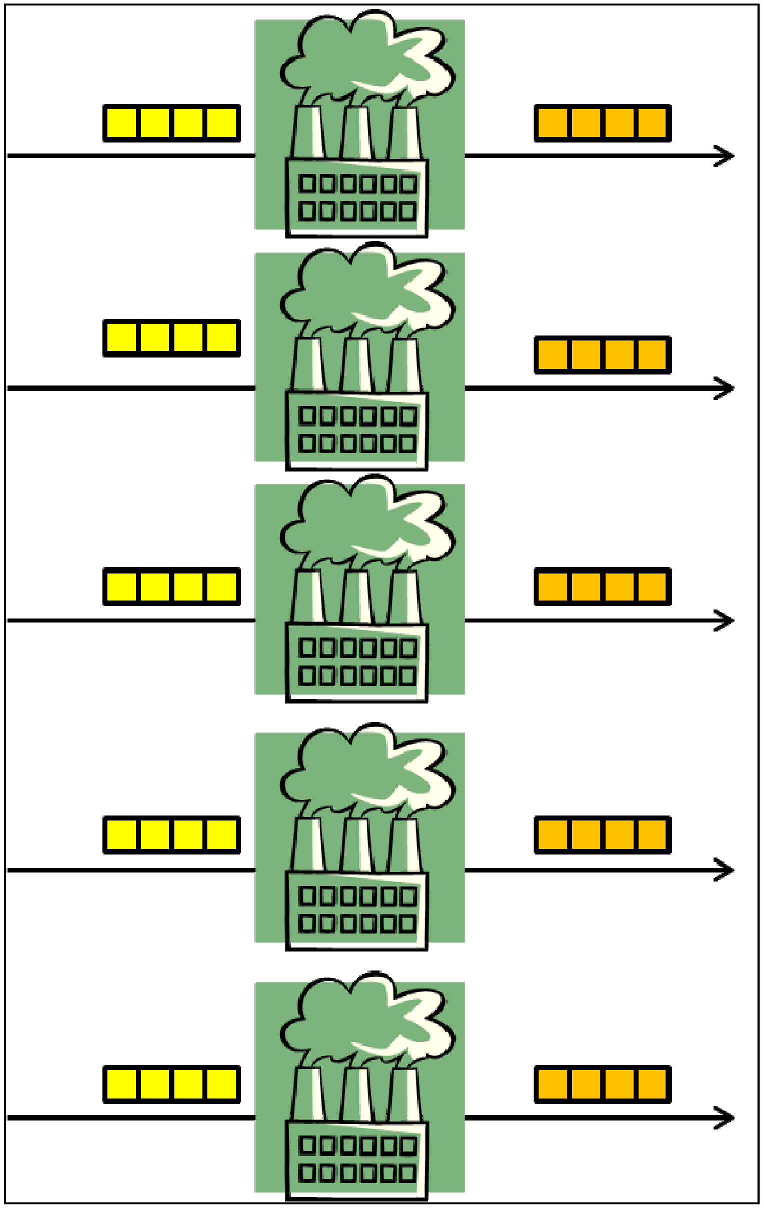


$$\#WorkGroups = \frac{GlobalInvocationSize}{WorkGroupSize}$$

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

The Similarity in Diagrams is not a Coincidence!

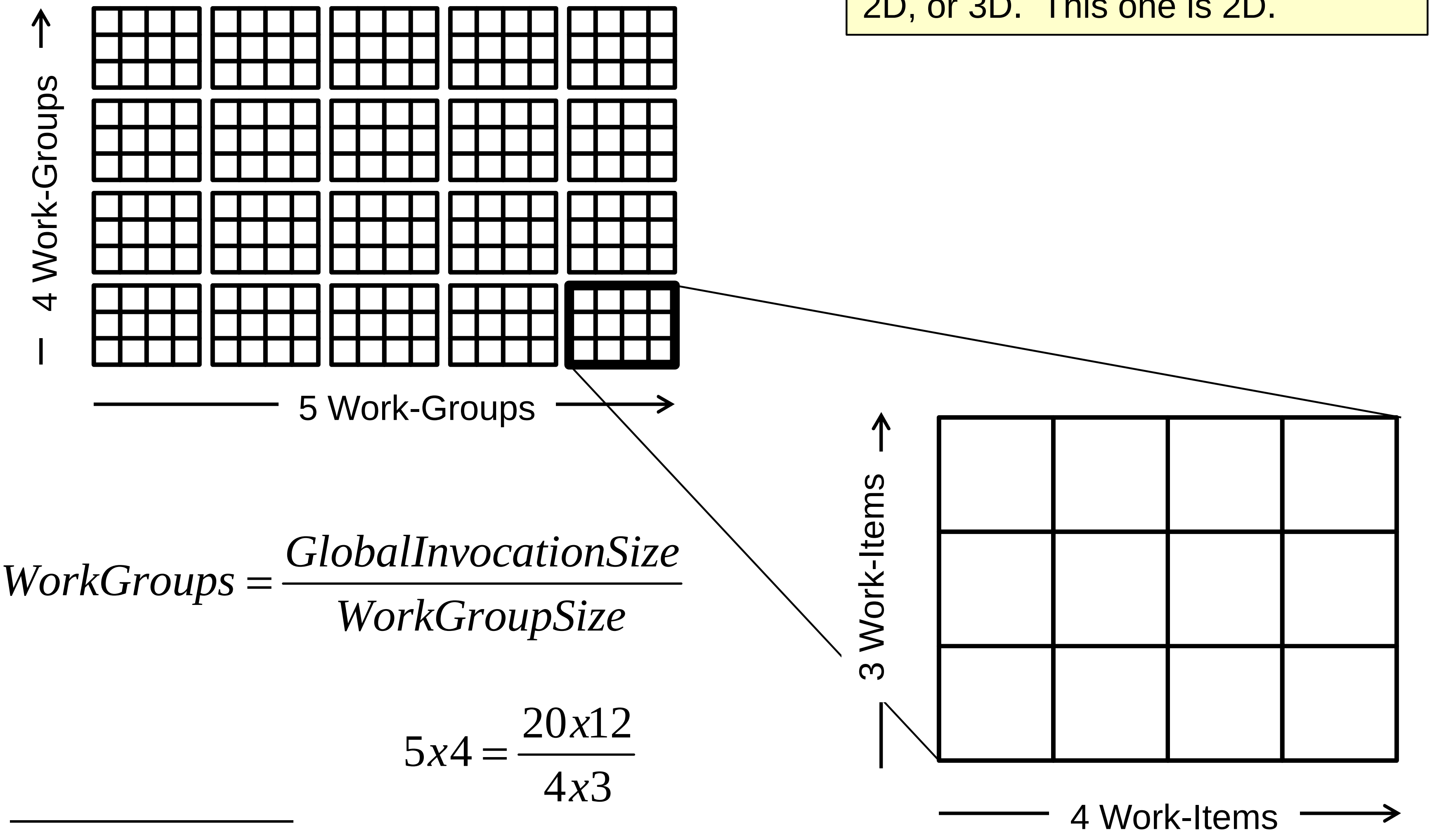




The Data Needs to be Divided into Large Quantities call Work-Groups, each of which is further Divided into Smaller Units Called Work-Items

20x12 (=240) total items to compute:

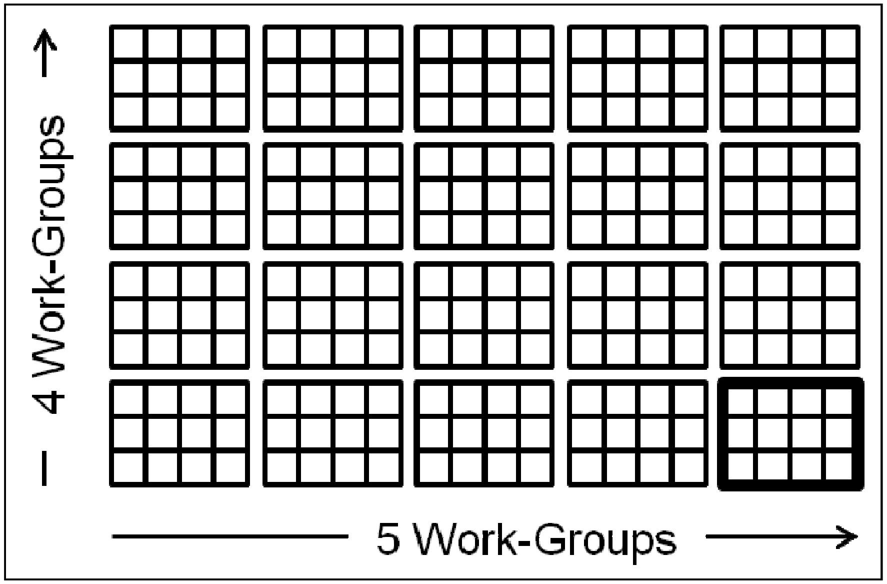
The Invocation Space can be 1D, 2D, or 3D. This one is 2D.

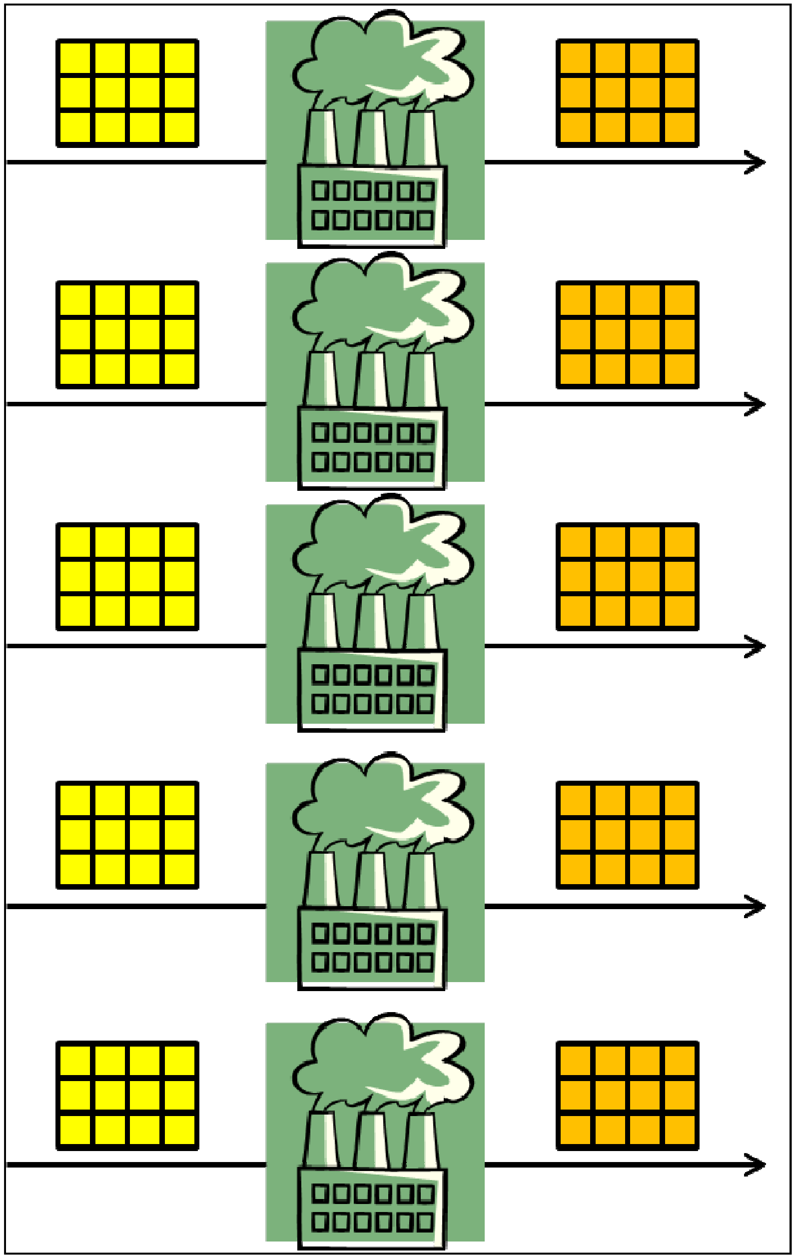


$$\#WorkGroups = \frac{GlobalInvocationSize}{WorkGroupSize}$$

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

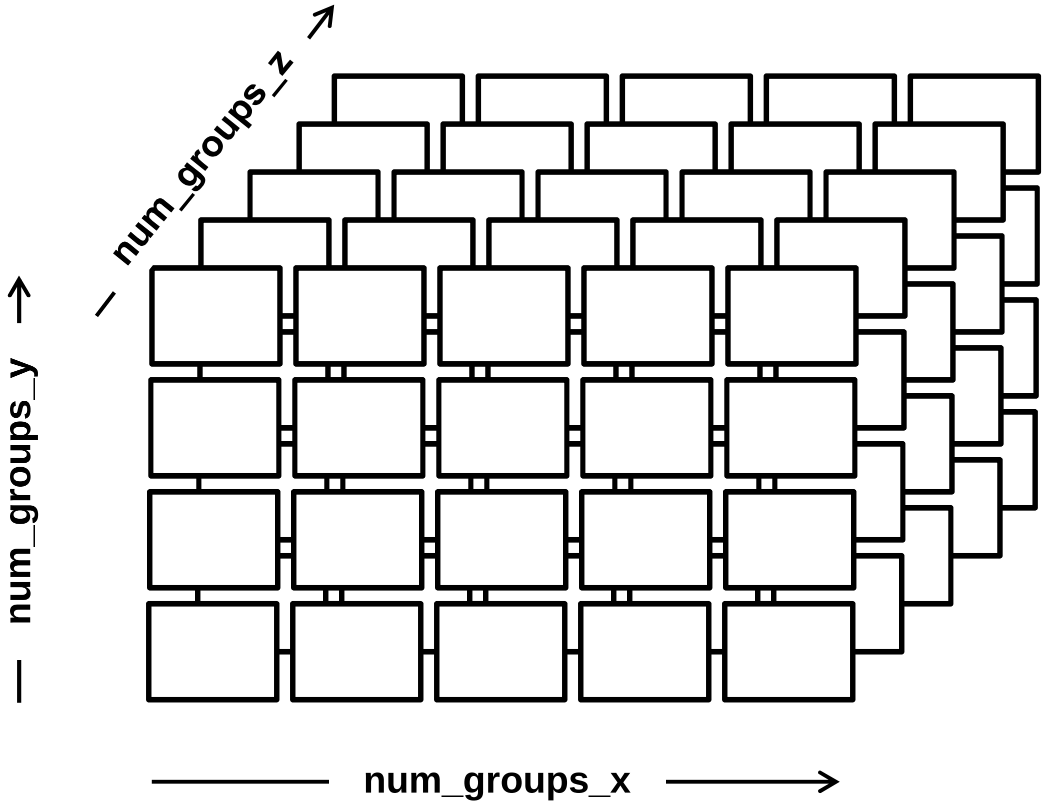
The Similarity in Diagrams is not a Coincidence!





Running the Compute Shader from the Application

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

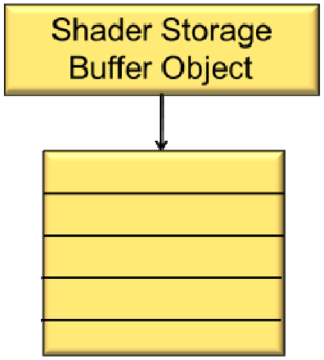


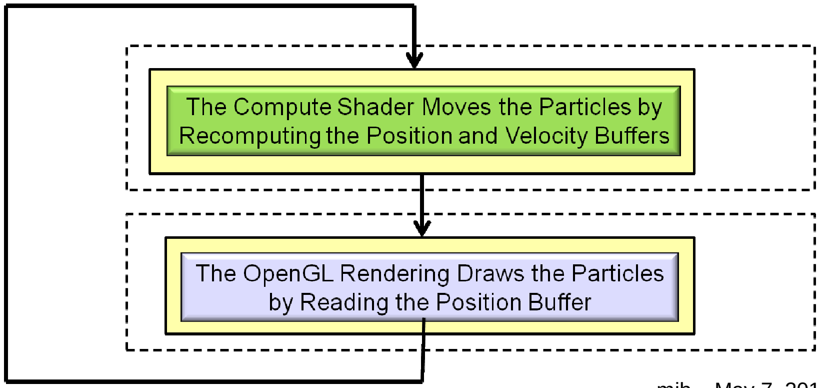
If the problem is 2D, then
num_groups_z = 1

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

Invoking the Compute Shader in Your C Program

```
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 );
```


Shader Storage Buffer Object


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

The OpenGL Rendering Draws the Particles by Reading the Position Buffer

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

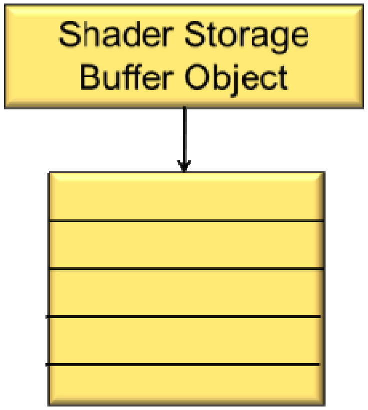
gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID

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

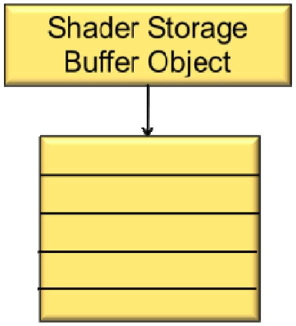
```
#version 430 compatibility
#extension GL_ARB_compute_shader :                  enable
#extension GL_ARB_shader_storage_buffer_object :   enable;

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;
```

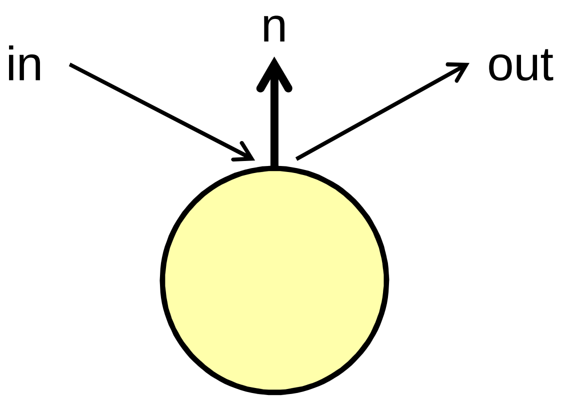

Shader Storage
Buffer Object

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

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

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?

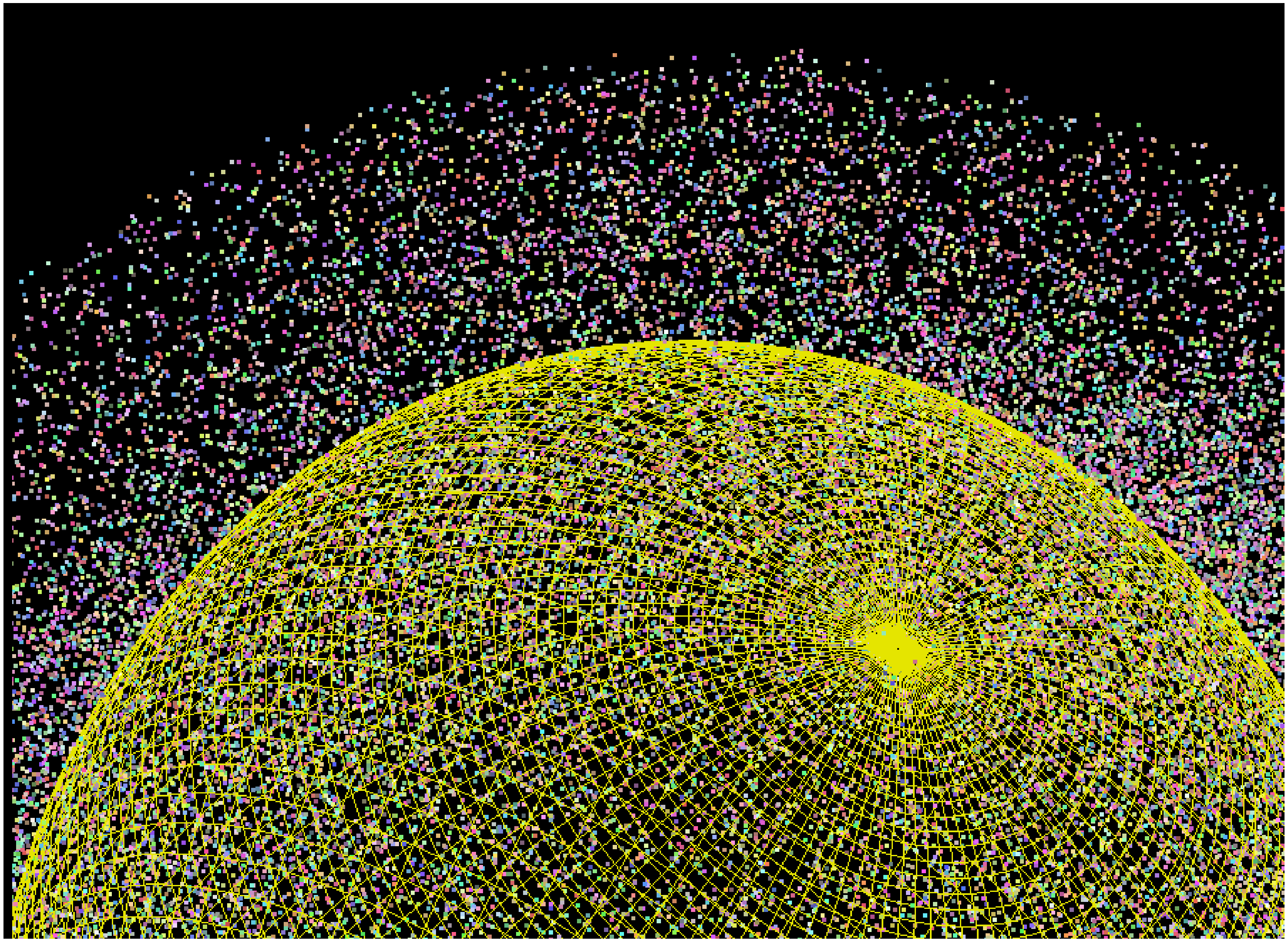
```
uint  gid = gl_GlobalInvocationID.x;               // the .y and .z are both 1 in this case

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;
```

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

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

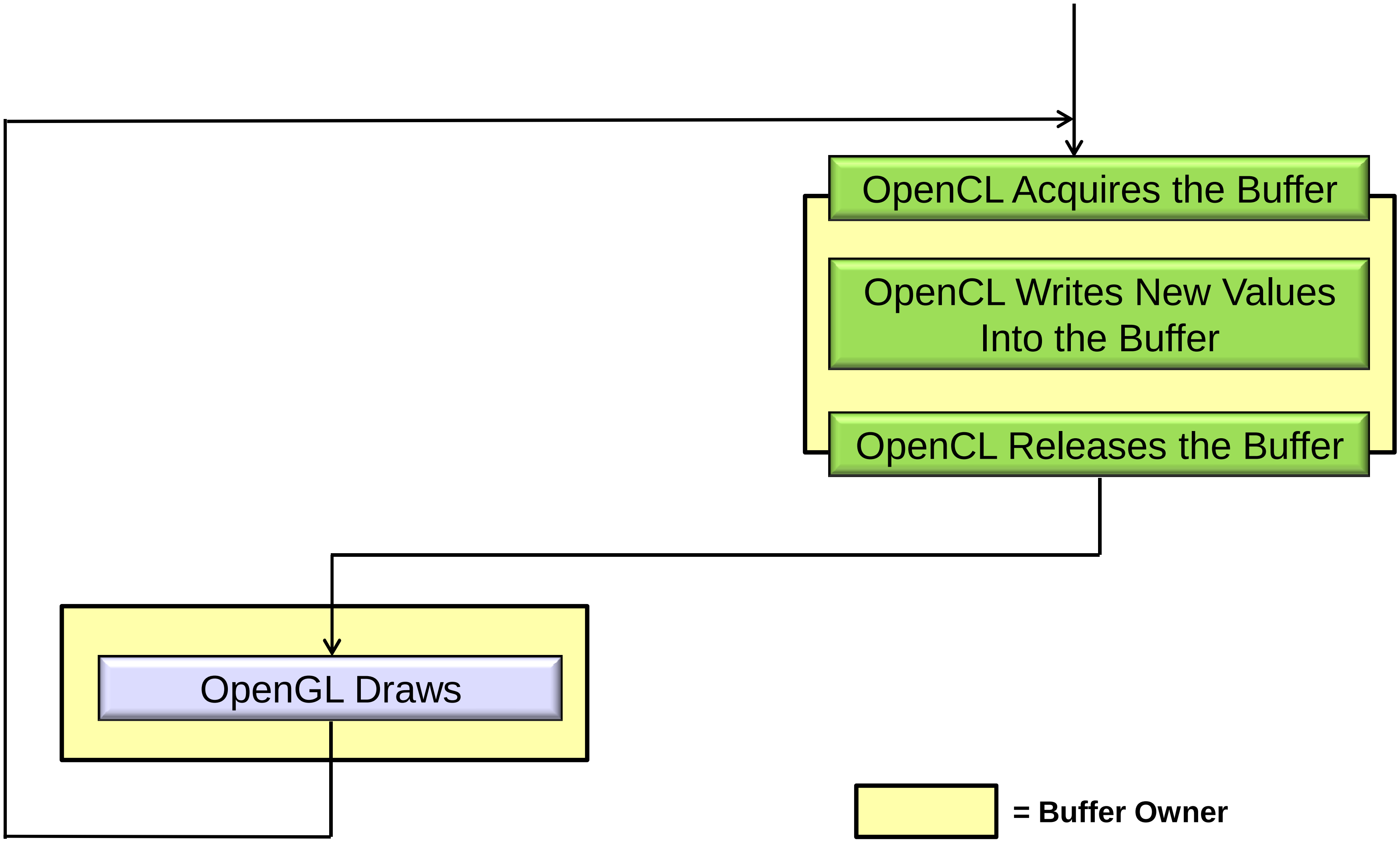
Graphics Trick Alert: Making the bounce happen from the surface of the sphere is time-consuming. Instead, bounce from the previous position in space. If DT is small enough, nobody will ever know…

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



OpenCL Computing Shaders

Either OpenGL or OpenCL Can Manipulate the Vertex Buffer at a Time, but not Both:
All of this Happens on the GPU



OpenCL Acquires the Buffer

OpenCL Writes New Values Into the Buffer

OpenCL Releases the Buffer

OpenGL Draws

🟨 **= Buffer Owner**

1. Program Header

```
#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <omp.h>


#ifdef WIN32
#include <windows.h>
#endif

#ifdef WIN32
#include "glew.h"
#endif

#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
#include "glui.h"

#include "cl.h"
#include "cl_gl.h"
```

Structures We Will Use to Fill the Vertex Buffers

```
// structs we will need later:

struct xyzw
{
                float x, y, z, w;
};


struct rgba
{
                float r, g, b, a;
};
```

OpenCL Global Variables

```
size_t  GlobalWorkSize[3]  =  { NUM_PARTICLES, 1, 1 };
size_t    LocalWorkSize[3]  =  { LOCAL_SIZE,          1, 1 };

gluint                               hPobj;      // host opengl object
gluint                               hCobj;      // host opengl object
struct xyzw *                        hVel;       // host c array
cl_mem                               dPobj;      // device memory buffer
cl_mem                               dCobj;      // device memory buffer
cl_mem                               dVel;       // device memory buffer

cl_command_queue                     CmdQueue;
cl_device_id                         Device;
cl_kernel                            Kernel;
cl_platform_id                       Platform;
cl_program                           Program;
```

A Deceptively-Simple Main Program

```
int
main( int argc, char *argv[ ] )
{
        glutInit( &argc, argv );
        InitGraphics(  );
        InitLists(  );
        InitCL(  );
        Reset(  );
        InitGlui(  );
        glutMainLoop(  );
        return 0;
}
```

Setting up OpenCL:
Querying the Existence of an OpenCL Extension

```
void
InitCL( )
{
        . . .

        status = clGetDeviceIDs( Platform, CL_DEVICE_TYPE_GPU, 1, &Device, NULL );
        PrintCLError( status, "clGetDeviceIDs: " );

        // since this is an opengl interoperability program,
        // check if the opengl sharing extension is supported
        // (no point going on if it isn't):
        // (we need the Device in order to ask, so we can't do it any sooner than right here)

        if(  IsCLExtensionSupported( "cl_khr_gl_sharing" )  )
        {
                fprintf( stderr, "cl_khr_gl_sharing is supported.\n" );
        }
        else
        {
                fprintf( stderr, "cl_khr_gl_sharing is not supported -- sorry.\n" );
                return;
        }
```

Querying the Existence of an OpenCL Extension

```
bool
IsCLExtensionSupported( const char *extension )
{
    // see if the extension is bogus:

    if( extension == NULL  ||  extension[0] == '\0' )
        return false;

    char * where = (char *) strchr( extension, ' ' );
    if( where != NULL )
        return false;

    // get the full list of extensions:

    size_t  extensionSize;
    clGetDeviceInfo( Device, CL_DEVICE_EXTENSIONS, 0, NULL, &extensionSize );
    char *extensions = new char [ extensionSize ];
    clGetDeviceInfo( Device, CL_DEVICE_EXTENSIONS, extensionSize, extensions, NULL );

    for(  char * start = extensions ;  ;  )
    {
        where = (char *) strstr( (const char *) start, extension );
        if( where == 0 )
        {
            delete [ ] extensions;
            return false;
        }

        char * terminator = where + strlen(extension);  // points to what should be the separator

        if( *terminator == ' '  ||  *terminator == '\0'  ||  *terminator == '\r'  ||  *terminator == '\n' )
        {
            delete [ ] extensions;
            return true;
        }
        start = terminator;
    }
}
```

Setting up OpenCL: The Interoperability Context

```
void
InitCL( )
{
                . . .

// get the platform id:

status = clGetPlatformIDs( 1, &Platform, NULL );
PrintCLError( status, "clGetPlatformIDs: " );

// get the device id:

status = clGetDeviceIDs( Platform, CL_DEVICE_TYPE_GPU, 1, &Device, NULL );
PrintCLError( status, "clGetDeviceIDs: " );

// 3. create a special opencl context based on the opengl context:

cl_context_properties props[  ] =
{
                CL_GL_CONTEXT_KHR,          (cl_context_properties) wglGetCurrentContext( ),
                CL_WGL_HDC_KHR,             (cl_context_properties) wglGetCurrentDC( ),
                CL_CONTEXT_PLATFORM,        (cl_context_properties) Platform,
                0
};

cl_context Context = clCreateContext( props, 1, &Device, NULL, NULL, &status );
PrintCLError( status, "clCreateContext: " );
```

Setting up OpenCL:
The Interoperability Context is Different for each OS

For Windows:

```
cl_context_properties props[ ] =
{
            CL_GL_CONTEXT_KHR,             (cl_context_properties) wglGetCurrentContext( ),
            CL_WGL_HDC_KHR,                (cl_context_properties) wglGetCurrentDC( ),
            CL_CONTEXT_PLATFORM,           (cl_context_properties) Platform,
            0
};
cl_context Context = clCreateContext( props, 1, &Device, NULL, NULL, &status );
```

For Linux:

```
cl_context_properties props[ ] =
{
            CL_GL_CONTEXT_KHR,             (cl_context_properties) glXGetCurrentContext( ),
            CL_GLX_DISPLAY_KHR,            (cl_context_properties) glXGetCurrentDisplay( ),
            CL_CONTEXT_PLATFORM,           (cl_context_properties) Platform,
            0
};
cl_context Context = clCreateContext( props, 1, &Device, NULL, NULL, &status );
```

For Apple:

```
cl_context_properties props[ ] =
{
            CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE,
                                           (cl_context_properties) kCGLShareGroup,
            0
};
cl_context Context = clCreateContext( props, 0, 0, NULL, NULL, &status );
```

Setting up OpenCL

```
void
InitCL( )
{
                . . .

// create the velocity array and the opengl vertex array buffer and color array buffer:

delete [  ] hVel;
hVel = new struct xyzw [ NUM_PARTICLES ];

glGenBuffers( 1, &hPobj );
glBindBuffer( GL_ARRAY_BUFFER, hPobj );
glBufferData( GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW );

glGenBuffers( 1, &hCobj );
glBindBuffer( GL_ARRAY_BUFFER, hCobj );
glBufferData( GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW );

glBindBuffer( GL_ARRAY_BUFFER, 0 );          // unbind the buffer

// fill those arrays and buffers:

ResetParticles( );
```

Setting the Initial Particle Parameters

```
void
ResetParticles( )
{
		glBindBuffer( GL_ARRAY_BUFFER, hPobj );
		struct xyzw *points = (struct xyzw *) glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
		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_ARRAY_BUFFER );


		glBindBuffer( GL_ARRAY_BUFFER, hCobj );
		struct rgba *colors = (struct rgba *) glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
		for( int i = 0; i < NUM_PARTICLES; i++ )
		{
				colors[ i ].r  = Ranf( 0., 1. );
				colors[ i ].g = Ranf( 0., 1. );
				colors[ i ].b = Ranf( 0., 1. );
				colors[ i ].a = 1.;
		}
		glUnmapBuffer( GL_ARRAY_BUFFER );

		. . .
```

Setting the Initial Particle Parameters

```
void
ResetParticles( )
{
        . . .

        delete [  ] hVel;
        hVel = new struct xyzw [ NUM_PARTICLES ];
        for( int i = 0; i < NUM_PARTICLES; i++ )
        {
                hVel[ i ].x = Ranf( VMIN, VMAX );
                hVel[ i ].y = Ranf( 0.      , VMAX );
                hVel[ i ].z = Ranf( VMIN, VMAX );
                hVel[ i ].w = 0.;
        }
}
```

Setting-up the Device-Side Buffers

```
void
InitCL( )
{
          . . .

// 5. create the opencl version of the velocity array:

dVel = clCreateBuffer( Context, CL_MEM_READ_WRITE, 4*sizeof(float)*NUM_PARTICLES, NULL, &status );
PrintCLError( status, "clCreateBuffer: " );

// 6. write the data from the host buffers to the device buffers:

status = clEnqueueWriteBuffer( CmdQueue, dVel, CL_FALSE, 0, 4*sizeof(float)*NUM_PARTICLES, hVel, 0, NULL, NULL );
PrintCLError( status, "clEneueueWriteBuffer: " );

// 5. create the opencl version of the opengl buffers:

dPobj = clCreateFromGLBuffer( Context, 0, hPobj, &status );
PrintCLError( status, "clCreateFromGLBuffer (1)" );

dCobj = clCreateFromGLBuffer( Context, 0, hCobj, &status );
PrintCLError( status, "clCreateFromGLBuffer (2)" );
```

Setup the Kernel Arguments…

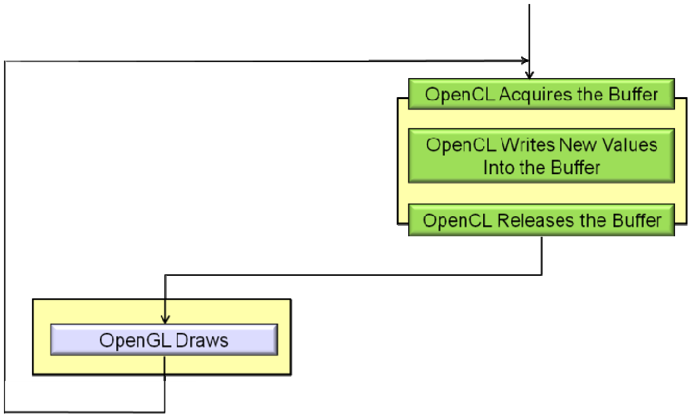
```
void
InitCL( )
{
            . . .

// 10. setup the arguments to the Kernel object:

status = clSetKernelArg( Kernel, 0, sizeof(cl_mem), &dPobj );
PrintCLError( status,"clSetKernelArg (1): " );

status = clSetKernelArg( Kernel, 1, sizeof(cl_mem), &dVel );
PrintCLError( status , "clSetKernelArg (2): " );

status = clSetKernelArg( Kernel, 2, sizeof(cl_mem), &dCobj );
PrintCLError( status,  "clSetKernelArg (3): " );
```

… to Match the Kernel’s Parameter List

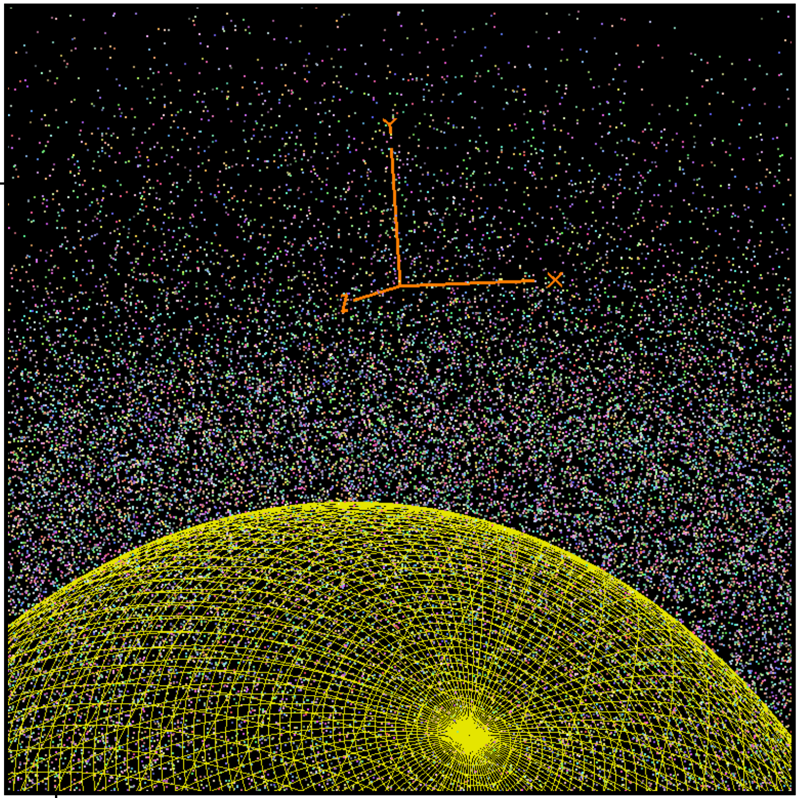
```
kernel
void
Particle( global point * dPobj,  global vector * dVel,  global color * dCobj )
{
            . . .
}
```

The “Idle Function” Tells OpenCL to Do Its Computing

```
void
Animate( )
{
        // acquire the vertex buffers from opengl:

        glutSetWindow( MainWindow );
        glFinish(  );

        cl_int status = clEnqueueAcquireGLObjects( CmdQueue, 1, &dPobj, 0, NULL, NULL );
        PrintCLError( status, “clEnqueueAcquireGLObjects (1) : “ );
        status = clEnqueueAcquireGLObjects( CmdQueue, 1, &dCobj, 0, NULL, NULL );
        PrintCLError( status, “clEnqueueAcquireGLObjects (2) : “ );

        double time0 = omp_get_wtime( );

        // 11. enqueue the Kernel object for execution:

        cl_event  wait;
        status = clEnqueueNDRangeKernel( CmdQueue, Kernel, 1, NULL, GlobalWorkSize, LocalWorkSize, 0, NULL, &wait );
        PrintCLError( status, "clEnqueueNDRangeKernel: " );

        status = clWaitForEvents( 1, &wait );
        PrintCLError( status, "clWaitForEvents: " );

        double time1 = omp_get_wtime( );
        ElapsedTime  =  time1 - time0;

        clFinish( CmdQueue );
        clEnqueueReleaseGLObjects( CmdQueue, 1, &dCobj, 0, NULL, NULL );
        PrintCLError( status, “clEnqueueReleaseGLObjects (1): “ );
        clEnqueueReleaseGLObjects( CmdQueue, 1, &dPobj, 0, NULL, NULL );
        PrintCLError( status, “clEnqueueReleaseGLObject (2): “ );

        glutSetWindow( MainWindow );
        glutPostRedisplay(  );
}
```



Redrawing the Scene:
The Particles



```
void
Display( )
{
        . . .

        glBindBuffer( GL_ARRAY_BUFFER, hPobj );
        glVertexPointer( 4, GL_FLOAT, 0, (void *)0 );
        glEnableClientState( GL_VERTEX_ARRAY );

        glBindBuffer( GL_ARRAY_BUFFER, hCobj );
        glColorPointer( 4, GL_FLOAT, 0, (void *)0  );
        glEnableClientState( GL_COLOR_ARRAY );

        glPointSize( 2. );
        glDrawArrays( GL_POINTS, 0, NUM_PARTICLES );
        glPointSize( 1. );

        glDisableClientState( GL_VERTEX_ARRAY );
        glDisableClientState( GL_COLOR_ARRAY );
        glBindBuffer( GL_ARRAY_BUFFER, 0 );

        glutSwapBuffers( );
        glFlush( );
}
```

13. Clean-up

```
void
Quit( )
{
        Glui->close( );
        glutSetWindow( MainWindow );
        glFinish( );
        glutDestroyWindow( MainWindow );


        // 13. clean everything up:

        clReleaseKernel(          Kernel   );
        clReleaseProgram(         Program  );
        clReleaseCommandQueue(    CmdQueue );
        clReleaseMemObject(       dPobj  );
        clReleaseMemObject(       dCobj  );

        exit( 0 );
}
```

particles.cl

```
kernel
void
Particle(  global point * dPobj,  global vector * dVel,  global color * dCobj  )
{
                int gid = get_global_id( 0 );                        // particle #

                point p    = dPobj[gid];
                vector v = dVel[gid];

                point pp    = p + v*DT + .5*DT*DT*G;                 // p'
                vector vp = v + G*DT;                                // v'
                pp.w = 1.;
                vp.w = 0.;

                if(   IsInsideSphere( pp, Sphere1 )   )
                {
                                vp = BounceSphere( p, v, Sphere1 );
                                pp = p + vp*DT + .5*DT*DT*G;
                }

                dPobj[gid] = pp;
                dVel[gid]    = vp;
}
```

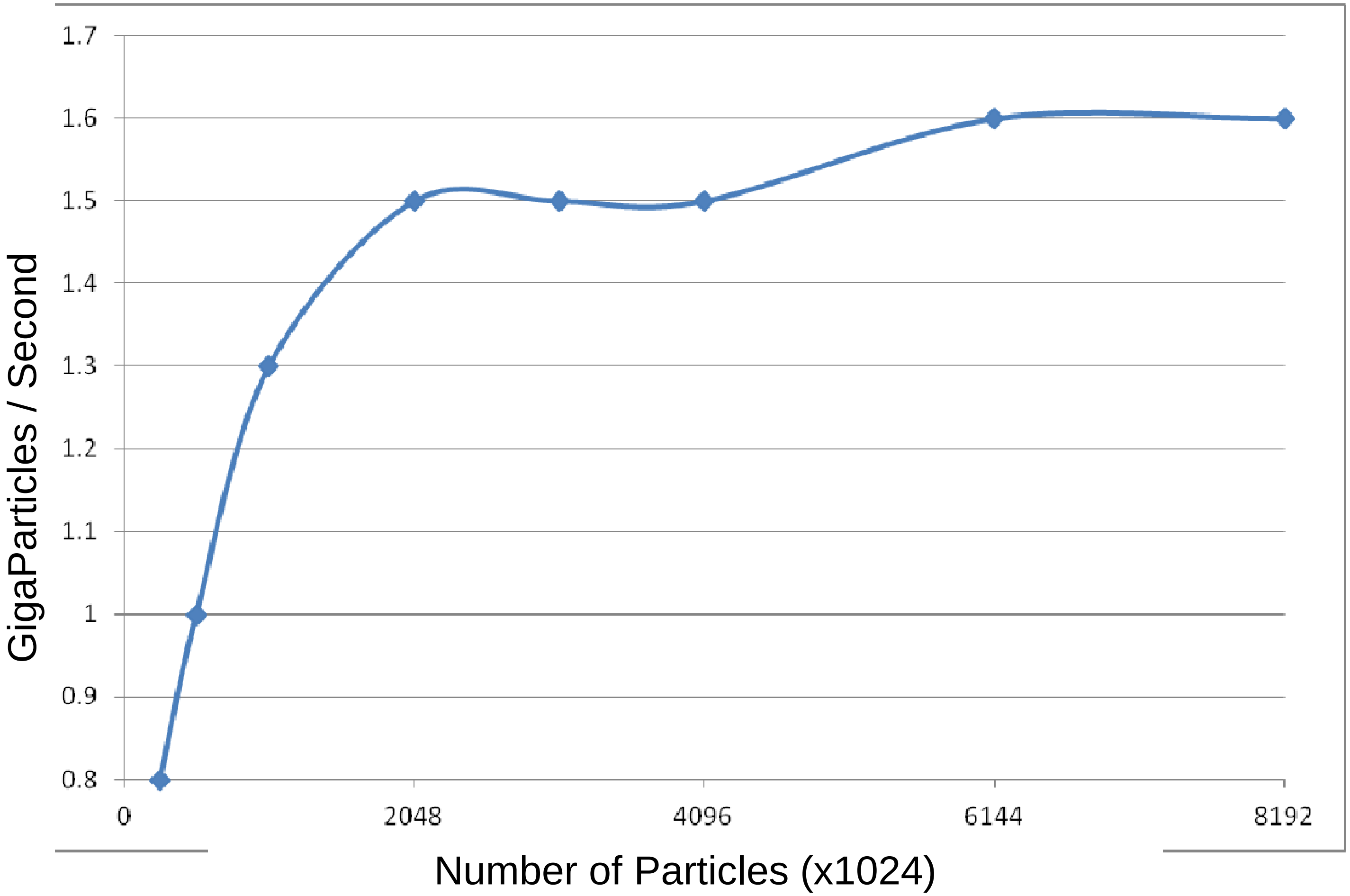
particles.cl

```
typedef  float4  point;
typedef  float4  vector;
typedef  float4  color;
typedef  float4  sphere;

constant float4 G              = (float4) ( 0., -9.8, 0., 0. );
constant float   DT            = 0.1;
constant sphere Sphere1 = (sphere)( -100., -800., 0.,  600. );

vector
Bounce( vector in, vector n )
{
                n.w = 0.;
                n = normalize( n );
                vector  out = in   -   2. * n * dot( in.xyz, n.xyz );
                out.w = 0.;
                return out;
}
```

particles.cl

```
vector
BounceSphere( point p, vector v,  sphere s )
{
                vector n;
                n.xyz = fast_normalize( p.xyz - s.xyz );
                n.w = 0.;
                return Bounce( v, n );
}


bool
IsInsideSphere( point p, sphere s )
{
                float r = fast_length( p.xyz - s.xyz );
                return  ( r < s.w );
}
```

Performance



How Do You Choose Between Compute Shaders and OpenCL?

OpenCL and Compute Shaders are *great*! They do a super job of using the GPU for general-purpose data-parallel computing. So, how do you choose between Compute Shaders and 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. Compute Shaders are "just there" as part of OpenGL 4.3.
- OpenCL is more feature-rich than OpenGL compute shaders.
- 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.
- Calls to OpenGL compute shaders appear to be more lightweight than calls to OpenCL kernels are. This should result in better performance.
- Using OpenCL is more involved. It requires more setup (queries, platforms, devices, queues, kernels, etc.). Compute Shaders are more convenient. They just flow in with the graphics.

How Do You Choose Between Compute Shaders and OpenCL?

The bottom line is that I use OpenCL for the big, bad stuff. But, for lighter-weight data-parallel computing, I've been using the Compute Shaders.

An example of a lighter-weight data-parallel graphics-related application is a **particle system**.

An example of a bigger, badder data-parallel graphics-related application is a **volume toolkit**.

References

- Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane, *OpenGL Programming Guide, 8th Edition*, 2013.

- Peter Pacheco, *An Introduction to Parallel Programming*, Morgan-Kaufmann, 2011.

- Aaftah Munshi, Benedict Gaster, Timothy Mattson, James Fung, and Dan Ginsburg, *OpenCL Programming Guide* Addison-Wesley, 2012.

- Benedict Gaster, Lee Howes, David Kaeli, Perhaad Mistry, and Dana Schaa, *Heterogeneous Computing with OpenCL*, Morgan-Kaufmann, 2012.