**OpenCL / OpenGL Vertex Buffer Interoperability: A Particle System Case Study**

Also, see the video at: http://cs.oregonstate.edu/~mjb/cs575/Projects/particles.mp4

**Mike Bailey**
mjb@cs.oregonstate.edu

This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](http://creativecommons.org/licenses/by-nc-nd/4.0/)

---

### OpenCL / OpenGL Vertex Interoperability: The Basic Idea

- Your C++ program writes initial values into the buffer on the GPU
- OpenCL acquires the buffer
- Each OpenCL kernel reads an \((x,y,z)\) value from the buffer
- Each OpenCL kernel updates its \((x,y,z)\) value
- Each OpenCL kernel writes its \((x,y,z)\) value back to the buffer
- OpenGL releases the buffer
- OpenGL draws using the \((x,y,z)\) values in the buffer on the GPU

---

### Some of the Inner Workings of OpenGL:
Feel Free to Detour Right to Slide #24 if You Don’t Want to Know This

In the Beginning of OpenGL ...

- You listed the vertices with separate function calls:
  - `glBegin(GL_TRIANGLES);`
  - `glVertex3f(x0, y0, z0);`
  - `glVertex3f(x1, y1, z1);`
  - `glVertex3f(x2, y2, z2);`
  - `glVertex3f(x3, y3, z3);`
  - `glVertex3f(x4, y4, z4);`
  - `glEnd();`

- Then someone noticed how inefficient that was, for three reasons:
  1. Sending large amounts of small pieces of information is less efficient than sending small amounts of large pieces of information
  2. The vertex coordinates were being listed in the CPU and were being transferred to the GPU every drawing pass
  3. Some vertices were listed twice

---

### Here’s What OpenGL Has Been Moving To: Vertex Buffer Objects

- `GLfloat CubeVertices[3][3] = {...}
- `GLfloat CubeColors[3][3] = {...}
- `GLuint CubeIndices[4][4] = {...}

---

### 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.

If we were implementing the OpenGL state as a C++ structure (which we’re not), we might do something like this:

```
struct context {
    float [4][4] Transformation;
    struct Texture * Texture;
    struct DataArrayBuffer * ArrayBuffer;
} Context;
```
More Background — How do you create a special OpenGL Array Buffer called a Vertex Buffer 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:

```c
GLuint buf;
glGenBuffers( 1, &buf );
```

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 — What is an OpenGL “Object”?

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

```c
struct DataArrayBuffer
{
    enum dataType;
    void * memStart;
    int memSize;
};
```

Then, you could create any number of Buffer Object instances, each with its own characteristics encapsulated within it. When you want to make that combination current, you just need to point the ArrayBuffer element of the Context to that entire struct (“bind”). When you bind an object, all of its information comes with it.

A Little Background — the OpenGL Rendering Context

It’s very fast to re-bind a different vertex buffer. It amounts to just changing a pointer.

```c
glBindBuffer( GL_ARRAY_BUFFER, buf );
```

Think of it as happening this way:

```c
float *data = Context.ArrayBuffer.memStart;
```

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.

Think of it as happening this way:

```c
real *data = Context.ArrayBuffer.memStart;
```

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.

Vertex Buffers: Putting Data in the Buffer Object

```c
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

*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 via memory-mapping.

Think of it as happening this way:

```c
Context.ArrayBuffer.memStart = CopyToGpuMemory( data, numBytes );
Context.ArrayBuffer.memSize = numBytes;
```
Preview: We are going to use a Particle System as a Case Study

Vertex Buffers: Putting Data in the Buffer Object

```c
void BufferData(type, numbytes, data, usage);
```

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

- **xx** can be:
  - STREAM this buffer will be written lots
  - STATIC this buffer will be written seldom and read often
  - DYNAMIC this buffer will be written often and read used often

- **yy** can be:
  - DRAW this buffer will be used for drawing
  - READ this buffer will be copied into
  - COPY not a real need for now, but someday...

- **GL_STATIC_DRAW** is the most common usage

Vertex Buffers: Step #1 – Fill the Arrays

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

```c
int numVertices = sizeof(Vertices) / (3*sizeof(GLfloat));
```

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

```c
void BufferData(type, numbytes, data, usage);
```

```c
void Buffer glGenBuffers(1, &buf);
```

```c
void Buffer glBindBuffer(GL_ARRAY_BUFFER, buf);
```

```c
void Buffer glBufferData(GL_ARRAY_BUFFER, 3*sizeof(GLfloat)*numVertices, Vertices, GL_STATIC_DRAW);
```

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

```c
void glEnableClientState(type);
```

- **type** can be any of:
  - GL_VERTEX_ARRAY
  - GL_COLOR_ARRAY
  - GL_NORMAL_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:
  ```c
  void disable glClientState(type);
  ```

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

```c
void Buffer glBindBuffer(GL_ARRAY_BUFFER, buf);
```

```c
Vertex Buffer Object
```

- Transformation
- Lighting
- Context
- Element Array Buffer
- Context
- Transformation
- Lighting
Vertex Buffers: Step #5 – Specify the Data

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

**Parameters:**
- **size**: 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.

Vertex Data

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Data</td>
<td>Color Data</td>
<td></td>
</tr>
</tbody>
</table>

The Data Types in a vertex buffer object can be stored either as “packed” or “interleaved”

Packed:
- `glVertexPointer(3, GL_FLOAT, 3*sizeof(GLfloat), 0);`
- `glColorPointer(3, GL_FLOAT, 3*sizeof(GLfloat), 3*numVertices*sizeof(GLfloat));`

Interleaved:
- `glVertexPointer(3, GL_FLOAT, 3*sizeof(GLfloat), 0);`
- `glColorPointer(3, GL_FLOAT, 6*sizeof(GLfloat), 3*sizeof(GLfloat));`

Vertex Buffers: Step #6 – Specify the Connections

```c
GLfloat Vertices[3][3] = {
    { x0, y0, z0 },
    { x1, y1, z1 },
    { x2, y2, z2 },
    { x3, y3, z3 },
    { x4, y4, z4 },
    { x5, y5, z5 }
};
```

```c
int numVertices = sizeof(Vertices) / (3*sizeof(GLfloat));
```

```c
glDrawArrays(GL_TRIANGLES, 0, numVertices);
```

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

```c
glBindBuffer(GL_ARRAY_BUFFER);
```

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

```c
float *vertexArray = glMapBuffer(GL_ARRAY_BUFFER, GL_READ_ONLY);
```

When you are done, be sure to call:

```c
glUnMapBuffer(GL_ARRAY_BUFFER);
```

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

We Now Pickup with the OpenCL Stuff

END DETOUR
Either OpenGL or OpenCL can use the Vertex Buffer at a time, but not both:

All of this happens on the GPU

OpenCL acquires the buffer

Each OpenCL kernel reads an (x,y,z) value from the buffer

Each OpenCL kernel updates its (x,y,z) value

Each OpenCL kernel writes its (x,y,z) value back to the buffer

OpenCL releases the buffer

OpenGL draws using the (x,y,z) values in the buffer on the GPU

Your C++ program writes initial values into the buffer on the GPU

#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/cl.h"
#include "CL/cl_gl.h"

/* structs we will need later: */
struct xyzw {
  float x, y, z, w;
};
struct rgba {
  float r, g, b, a;
};

size_t GlobalWorkSize[3] = { NUM_PARTICLES, 1, 1 };
size_t LocalWorkSize[3] = { LOCAL_SIZE, 1, 1 };
GLuint hPobj; // host opengl object for Points
GLuint hCobj; // host opengl object for Colors
struct xyzw * hVel; // host C++ array for Velocities
dPobj; // device memory buffer for Points
dCobj; // device memory buffer for Colors
dVel; // device memory buffer for Velocities
cl_command_queue CmdQueue;
cl_device_id Device;
cl_kernel Kernel;
cl_platform_id Platform;
cl_program Program;

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

#include <glew.h>
GLenum err = glewInit( );
if( err != GLEW_OK )
{
  fprintf( stderr, "glewInit Error
" );
}
Setting up OpenCL: The Interoperability Context

For Windows:

```c
CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE, (cl_context_properties) kCGLShareGroup,
0
```

For Linux:

```c
CL_CONTEXT_PROPERTY_USE_COL_SHAREGROUP_APPLE, (cl_context_properties) kCOLShareGroup,
0
```

For Apple:

```c
CL_CONTEXT_PROPERTY_USE_COL_SHAREGROUP_APPLE, (cl_context_properties) kCOLShareGroup,
0
```

Querying the Existence of an OpenCL Extension

```c
for(  char * start = extensions ;  ;  )
```

```c
for(  char * start = extensions ;  ;  )
```

Setting the Initial Particle Parameters, I

```c
for(  int i = 0; i < NUM_PARTICLES; i++ )
```

```c
for(  int i = 0; i < NUM_PARTICLES; i++ )
```

```c
for(  int i = 0; i < NUM_PARTICLES; i++ )
```

```c
for(  int i = 0; i < NUM_PARTICLES; i++ )
```
Setting the Initial Particle Parameters, II

```c
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    hVel[i].x = Ranf( &Seed, VMIN, VMAX );
    hVel[i].y = Ranf( &Seed, 0, VMAX );
    hVel[i].z = Ranf( &Seed, VMIN, VMAX );
    hVel[i].w = 0.;
}
```

Setting-up the Device-Side Buffers

```c
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, "clEnqueueWriteBuffer: " );
    // 5. create the opencl version of the opengl buffers:
    dPobj = clCreateFromGLBuffer( Context, CL_MEM_READ_WRITE, hPobj, &status );
    PrintCLError( status, "clCreateFromGLBuffer (1) " );
    dCobj = clCreateFromGLBuffer( Context, CL_MEM_READ_WRITE, hCobj, &status );
    PrintCLError( status, "clCreateFromGLBuffer (2) " );
}
```

This is how OpenCL and OpenGL Share the Same Memory Buffer

```c
void Animate( )
{
    // acquire the vertex buffers from opengl:
    glutSetWindow( MainWindow );
    glFinish( );
    cl_int status;
    status = clEnqueueAcquireGLObjects( CmdQueue, 1, &dPobj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueAcquireGLObjects (1) : " );
    status = clEnqueueAcquireGLObjects( CmdQueue, 1, &dCobj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueAcquireGLObjects (2) : " );
    Wait(); // note: only need to wait here because doing timing
    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: " );
    Wait();  // note: only need to wait here because doing timing
    double time1 = omp_get_wtime( );
    ElapsedTime = time1 - time0;
    clEnqueueReleaseGLObjects( CmdQueue, 1, &dCobj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueReleaseGLObjects (1): " );
    clEnqueueReleaseGLObjects( CmdQueue, 1, &dPobj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueReleaseGLObjects (2): " );
    Wait( );
    glutSetWindow( MainWindow );
    glutPostRedisplay( );
}
```

The OpenGL "Idle Function" Tells OpenCL to Do Its Computing

This is how OpenCL Manages Exclusive Access to the Memory Buffer
Redrawing the Scene:
The Particles

```
void Display()
{
    // Bind buffers
    glBindBuffer(GL_ARRAY_BUFFER, hPobj);
    glVertexPointer(4, GL_FLOAT, 0, (void*)0);
    glEnableClientState(GL_VERTEX_ARRAY);

    // Bind buffers again
    glBindBuffer(GL_ARRAY_BUFFER, hCobj);
    glColorPointer(4, GL_FLOAT, 0, (void*)0);
    glEnableClientState(GL_COLOR_ARRAY);

    // Set point size
    glPointSize(2.);
    glDrawArrays(GL_POINTS, 0, NUM_PARTICLES);

    // Disable client state
    glPointSize(1.);
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_COLOR_ARRAY);

    // Unbind buffers
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // Swap buffers and flush
    glutSwapBuffers();
    glFlush();
}
```

Redrawing the Scene:
The Performance

```
void Display()
{
    // Performance display
    if (ShowPerformance)
    {
        float start = glutGet(GLUT_ELAPSED_TIME);
        float end;
        float dt = end - start;
        float fps = 1.0 / dt;
        float giga_particles_per_second = (float)NUM_PARTICLES / dt;
        float giga_per_second = (float)NUM_PARTICLES / dt / 1000000000.0f;

        // Draw performance text
        glColor3f(1.0, 1.0, 1.0);
        DoRasterString(5.0, 5.0, 0.0, fmt, fps);
        DoRasterString(5.0, 5.0, 0.0, fmt, giga_per_second);
    }
}
```

13. Clean-up

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

    // Clean up memory
    clReleaseKernel(Kernel);
    clReleaseProgram(Program);
    clReleaseCommandQueue(CmdQueue);
    clReleaseMemObject(dPobj);
    clReleaseMemObject(dCobj);
    exit(0);
}
```

particles.cl (I)

```
typedef float4 point; // x, y, z – the w is unused
typedef float4 vector; // vx, vy, vz – the w is unused
typedef float4 color; // r, g, b – the w is unused
typedef float4 sphere; // xc, yc, zc, r

// despite what we think of the 4 components as representing,
// they are all referenced as .x, .y, .z, and .w
constant float4 G = (float4) (0., -9.8, 0., 0.); // gravity
constant float DT = 0.1; // time step
constant sphere Sphere1 = (sphere) (-100., -800., 0., 600.); // xc, yc, zc, r

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

particles.cl (II)

```
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 + G * (point)(.5*DT*DT); // 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 + G * (point)(.5*DT*DT);
    }

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

particles.cl (III)

```
vector Bounce(vector in, vector n)
{
    n.w = 0.;
    n = fast_normalize(n); // make it a unit vector
    vector out = in - n * (vector)(2.*dot(in.xyz, n.xyz));
    out.w = 0.;
    return out;
}
```

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

particles.cl (IV)

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

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

```
vector Bounce(vector in, vector n)
{
    n.w = 0.;
    n = fast_normalize(n); // make it a unit vector
    // This is the vector equation for "angle of reflection equals angle of incidence":
    vector out = in - n * (vector)(2.*dot(in.xyz, n.xyz));
    out.w = 0.;
    return out;
}
```

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

```
vector Bounce(vector in, vector n)
{
    n.w = 0.;
    n = fast_normalize(n); // make it a unit vector
    // This is the vector equation for "angle of reflection equals angle of incidence":
    vector out = in - n * (vector)(2.*dot(in.xyz, n.xyz));
    out.w = 0.;
    return out;
}
```

Remember from the OpenCL Assembly Language notes:
"The sqrt(x^2+y^2+z^2) assembly code is amazingly involved. I suspect it is an issue of maintaining highest precision. Use fast_sqrt(), fast_normalize(), and fast_length() when you can."
Jane Parallel's Performance

![Graph showing number of particles vs. GigaParticles per second]

Number of Particles (x1024)

GigaParticles per Second

2048 4096 8192 16384