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

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

---

### 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:

**In the Beginning of OpenGL …**

You listed the vertices with separate function calls:

```c
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 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

**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:

```c
struct context {
  float [4][4] Transformation;
  struct Texture * Texture0;
  struct DataArrayBuffer * ArrayBuffer;
  ...
};
```

**Detour**

**Color**

**Transformation**

**Display**

**Context**

**Texture0**

**ArrayBuffer**

---

---

---

---

---
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.

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 );
```

Vertex Buffers: Putting Data in the Buffer Object

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.

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

Think of it as happening this way:

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
- `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.
Computer Graphics

Preview: We are going to use a Particle System as a Case Study

Vertex Buffers: Putting Data in the Buffer Object

```c
void glVertexAttribPointer(GLuint index, GLsizeiptr stride, GLenum type,
                          GLboolean normalize, GLsizeiptr size,
                          const GLvoid *ptr, GLuint offset);
```

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

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

and YYY 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. },
  ...
};

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

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

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

// Bind the buffer
glBindBuffer( GL_ARRAY_BUFFER, buf );

// Fill the 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
// Enable the vertex array
glEnableClientState( GL_VERTEX_ARRAY );
```

- 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
glDisableClientState( type );
```

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

```c
// Draw
glDrawArrays( GL_TRIANGLES, 0, numVertices );
```
Vertex Buffers: Step #5 – Specify the Data

```c
glVertexPointer(size, type, stride, rel_address);
glColorPointer(size, type, stride, rel_address);
glNormalPointer(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`, or `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.

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

### Packed:
```c
glVertexPointer(3, GL_FLOAT, 3*sizeof(GLfloat), 0);
gColorPointer(3, GL_FLOAT, 3*sizeof(GLfloat), 3*numVertices*sizeof(GLfloat));
```

### Interleaved:
```c
glVertexPointer(3, GL_FLOAT, sizeof(GLfloat), 0);
gColorPointer(3, GL_FLOAT, 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 }
};

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

gDrawArrays(GL_TRIANGLES, 0, numVertices);
```

Vertex Buffers: Writing Data Directly into a Vertex Buffer

```c
glBindBuffer(buf, GL_ARRAY_BUFFER);
gBufferData(GL_ARRAY_BUFFER, 3*sizeof(float)*numVertices, NULL, GL_STATIC_DRAW);

float *vertexArray = glMapBuffer(GL_ARRAY_BUFFER, usage);

when you are done, be sure to call:
gUnMapBuffer(GL_ARRAY_BUFFER);
```

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>
#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"

1. Program Header

- include <stdio.h>
- include <math.h>
- include <stdlib.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"

2. Structures We Will Use to Fill the Vertex Buffers

```c
#include <stdio.h>
#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"

1. Program Header

- include <stdio.h>
- include <math.h>
- include <stdlib.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"

2. Structures We Will Use to Fill the Vertex Buffers

```c

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

3. OpenCL Global Variables

```c
#include <stdio.h>
#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"

1. Program Header

- include <stdio.h>
- include <math.h>
- include <stdlib.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"

2. Structures We Will Use to Fill the Vertex Buffers

```c

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

3. OpenCL Global Variables

```c

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
cl_mem dPobj; // device memory buffer for Points
cl_mem dCobj; // device memory buffer for Colors
cl_mem dVel; // device memory buffer for Velocities
cl_command_queue CmdQueue;
cl_device_id Device;
cl_kernel Kernel;
cl_platform_id Platform;
cl_program Program;

4. A Deceptively-Simple Main Program

```c
#include <stdio.h>
#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"

1. Program Header

- include <stdio.h>
- include <math.h>
- include <stdlib.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"

2. Structures We Will Use to Fill the Vertex Buffers

```c

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

3. OpenCL Global Variables

```c

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
cl_mem dPobj; // device memory buffer for Points
cl_mem dCobj; // device memory buffer for Colors
cl_mem dVel; // device memory buffer for Velocities
cl_command_queue CmdQueue;
cl_device_id Device;
cl_kernel Kernel;
cl_platform_id Platform;
cl_program Program;

```c

A Deceptively-Simple Main Program

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

GLEW – the GL Extension Wrangler

```c

#ifdef WIN32
GLenum err = glewInit();
if (err != GLEW_OK)
{
  fprintf(stderr, "glewInit Error\n");
}
#endif

This must wait to be called until after a graphics window is open!

Why? Because creating the window is what builds the graphics context.
### Setting up OpenCL: Querying the Existence of an OpenCL Extension

```c
void inICL() {
    ... // code here ...
}
```

### Setting up OpenCL: The Interoperability Context

```c
void inICL() {
    ... // code here ...
}
```

### Setting up OpenCL: The Initial Particle Parameters, I

```c
void ResetParticles() {
    ... // code here ...
}
```

---

### Querying the Existence of an OpenCL Extension

```c
void inICL() {
    ... // code here ...
}
```

### Setting up OpenCL: The Interoperability Context is Different for each OS (oh, good...)

For Windows:

```c
0x_context_properties props[ ] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties) Platform,
    0
};
```

For Linux:

```c
0x_context_properties props[ ] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties) Platform,
    CL_GL_CONTEXT_KHR, (cl_context_properties) glXGetCurrentContext(   ),
    0
};
```

For Apple:

```c
0x_context_properties props[ ] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties) kCGLShareGroup,
    CL_GL_CONTEXT_KHR, (cl_context_properties) glXGetCurrentContext(   ),
    0
};
```

### Setting the Initial Particle Parameters, II

```c
unsigned int Seed;
```

```c
void ResetParticles() {
    ... // code here ...
}
```
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, 0.00, VMAX );
    hVel[i].z = Ranf( &Seed, VMIN, VMAX );
    hVel[i].w = 0.0;
}
```

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

- Step #1: OpenGL creates the buffer on the GPU
- Step #2: OpenCL is told about it and creates a device pointer to the already-filled memory, just as if you had called clCreateBuffer() and clEnqueueWriteBuffer()

Setup the Kernel Arguments...

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

The OpenGL “Idle Function” Tells OpenCL to Do its Computing

```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, "clEnqueueReleaseGLObject (2): " );
    Wait();
    glutSetWindow( MainWindow );
    glutPostRedisplay( );
}
```

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

```c
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( );
}
```

Redrawing the Scene:
The Performance

```c
void Display( )
{
    . . .
    if( ShowPerformance )
    {
        char str[128];
        sprintf( str, "%6.1f GigaParticles/Sec", (float)NUM_PARTICLES/ElapsedTime/1000000000. );
        glDisable( GL_DEPTH_TEST );
        glMatrixMode( GL_PROJECTION );
        glLoadIdentity( );
        gluOrtho2D( 0., 100., 0., 100. );
        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity( );
        glColor3f( 1., 1., 1. );
        DoRasterString( 5., 5., 0., str );
    }
}
```

13. Clean-up

```c
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, I

Typedefs:

- `point`: x, y, z – the w is unused
- `vector`: vx, vy, vz – the w is unused
- `color`: r, g, b – the w is unused
- `sphere`: xc, yc, zc, r

```c
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
```

Constant float4s:

- `G`: gravity
- `DT`: time step
- `Sphere1`: xc, yc, zc, r

```c
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
```

Boolean function:

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

Particles class:

```c
kernel 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 class, II

Function to bounce particles from the sphere:

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

Particles class, III

Function to return the bounce vector:

```c
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;
}
```

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…

Particles class, IV

Function to return the bounce vector between two points:

```c
vector Bounce( point p, point q, sphere s )
{
    vector n;
    n.xyz = (p.xyz - s.xyz); // the vector from the sphere center to the point is the normal
    n = fast_normalize( n );
    return Bounce( point( q.xyz - s.xyz ), n );
}
```

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."