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

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

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 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

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 Color;
    float 4x4 Transformation;
    struct Texture * Texture;
    struct DataArrayBuffer * ArrayBuffer;
} Context;
```

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 Color;
    float 4x4 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:

```cpp
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:

```cpp
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.

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

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.

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

Think of it as happening this way:
`Context.ArrayBuffer.memStart = CopyToGpuMemory( data, numBytes );`
`Context.ArrayBuffer.memSize = numBytes;`

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

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

Vertex Buffers: Putting Data in the Buffer Object

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

- `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. },
   ...};
```

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

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

```c
glGenBuffers( 1, &buf );
glBindBuffer( GL_ARRAY_BUFFER, buf );
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
GLenum type = GL_VERTEX_ARRAY;
```

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

- `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.

Other types, too.

To deactivate a type, call:

```c
gDisableClientState( type );
```

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

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

- **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.

Vertex Buffers: Step #6 – Specify the Connections

```c
GLfloat Vertices[5][3] = {
  {x0, y0, z0},
  {x1, y1, z1},
  {x2, y2, z2},
  {x3, y3, z3},
  {x4, y4, z4},
};

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

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

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, 6*sizeof(GLfloat), 0);
  - glColorPointer(3, GL_FLOAT, 6*sizeof(GLfloat), 3*sizeof(GLfloat));

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, 6*sizeof(GLfloat), 0);
  - glColorPointer(3, GL_FLOAT, 6*sizeof(GLfloat), 3*sizeof(GLfloat));

Vertex Buffers: Writing Data Directly into a Vertex Buffer

- glBufferData(GL_ARRAY_BUFFER, 3*sizeof(float)*numVertices, NULL, GL_STATIC_DRAW);
- float *vertexArray = glMapBuffer(GL_ARRAY_BUFFER, usage);
- glUnMapBuffer(GL_ARRAY_BUFFER);

When you are done, be sure to call:
```
glUnMapBuffer(GL_ARRAY_BUFFER);
```

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

Vertex Buffers: Writing Data Directly into a Vertex Buffer

Map the buffer from GPU memory into the memory space of the application:
```
glBufferData(GL_ARRAY_BUFFER, 3*sizeof(float)*numVertices, NULL, GL_STATIC_DRAW);
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);
```

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

- All of this happens on the GPU

We Now Pickup with the OpenCL Stuff

**END DETOUR**

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

- 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
- OpenCL releases the buffer
- OpenGL draws using the (x,y,z) values in the buffer on the GPU
# Program Header

```c
#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"
```

## 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;
};
```

### 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
dPobj; // device memory buffer for Points
dCobj; // device memory buffer for Colors
dVel; // device memory buffer for Velocities
```

### 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
#include <GLEW.h>
GLenum err = glewInit( );
if( err != GLEW_OK )
{
    fprintf( stderr, "glewInit Error
" );
}
```

### Setting up OpenCL: Querying the Existence of an OpenCL Extension

```c
if( IsCLExtensionSupported( "cl_khr_gl_sharing" ) )
{
    fprintf( stderr, "cl_khr_gl_sharing is supported.
" );
    // 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 not supported – sorry!
" );
    } else
    {
        fprintf( stderr, "cl_khr_gl_sharing is supported – sorry!
" );
    }
```
Setting up OpenCL: The Interoperability Context

```c
void InitCL()
{
    ... // setup the platform and device
    status = clGetPlatformIDs(1, &Platform, NULL);
    PrintCLError(status, "clGetPlatformIDs: ");
    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_CONTEXT_PLATFORM, (cl_context_properties) Platform,
        CL_GL_CONTEXT_KHR, (cl_context_properties) wglGetCurrentContext(),
        CL_WGL_HDC_KHR, (cl_context_properties) wglGetCurrentDC(),
    };
    cl_context Context = clCreateContext(props, 1, &Device, NULL, NULL, &status);
    PrintCLError(status, "clCreateContext: ");
}
```

Setting the Initial Particle Parameters, I

```c
// fill those arrays and buffers:
void ResetParticles()
{
    glBindBuffer(GL_ARRAY_BUFFER, 0); // unbind the buffer
    glBufferData(GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, hPobj);
    glGenBuffers(1, &hPobj);
    glBufferData(GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, hCobj);
    glGenBuffers(1, &hCobj);
    glBufferData(GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW);
    glUnmapBuffer(GL_ARRAY_BUFFER);
    delete [] hVel;
    // create the velocity array and the opengl vertex array buffer and color array buffer:
    delete [] hVel;
    hVel = new struct xyzw[NUM_PARTICLES];
    struct xyzw *points = (struct xyzw *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
    glBindBuffer(GL_ARRAY_BUFFER, hPobj);
    for(int i = 0; i < NUM_PARTICLES; i++)
    {
        points[i].x = Ranf(&Seed, XMIN, XMAX);
        points[i].y = Ranf(&Seed, YMIN, YMAX);
        points[i].z = Ranf(&Seed, ZMIN, ZMAX);
        points[i].w = 1.0;
        glUnmapBuffer(GL_ARRAY_BUFFER);
    }
    return true;
}
```

Setting the Initial Particle Parameters, II

```c
// fill those arrays and buffers:
void ResetParticles()
{
    glBindBuffer(GL_ARRAY_BUFFER, 0); // unbind the buffer
    glBufferData(GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, hPobj);
    glGenBuffers(1, &hPobj);
    glBufferData(GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, hCobj);
    glGenBuffers(1, &hCobj);
    glBufferData(GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW);
    glUnmapBuffer(GL_ARRAY_BUFFER);
    delete [] hVel;
    // create the velocity array and the opengl vertex array buffer and color array buffer:
    delete [] hVel;
    hVel = new struct rgba[NUM_PARTICLES];
    struct rgba *colors = (struct rgba *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
    glBindBuffer(GL_ARRAY_BUFFER, hCobj);
    for(int i = 0; i < NUM_PARTICLES; i++)
    {
        colors[i].a = 1.0;
        colors[i].b = Ranf(&Seed, 0.0, 1.0);
        colors[i].r = Ranf(&Seed, 0.0, 1.0);
        colors[i].g = Ranf(&Seed, 0.0, 1.0);
        colors[i].w = 1.0;
        glUnmapBuffer(GL_ARRAY_BUFFER);
    }
    return true;
}
```
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) ");
}

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

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();
}

This is how OpenCL and OpenGL Share the Same Memory Buffer

This is how OpenCL Manages Exclusive Access to the Memory Buffer

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

Redrawing the Scene: The Particles

Setting-up the Device-Side Buffers

Setup the Kernel Arguments...

... to Match the Kernel's Parameter List

Note: you don't need an OpenGL-accessible buffer for the velocities. Velocities aren't needed for drawing.

Velocities are only needed to update point positions. The velocity buffer can just be done internally to OpenCL.
void Display()
{
  ...  
  if (ShowPerformance)
  {
    char str[128];
    sprintf(str, "%6.1f GigaParticles/Sec", (float)NUM_PARTICLES/ElapsedTime/1000000000. );
    glLoadIdentity();
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(0., 100., 0., 100. );
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor3f(1., 1., 1.);
    DoRasterString(5., 5., 0., str);
  }
}

Redraw the Scene:
The Performance

13. Clean-up

void Quit()
{
  Glui->close();
  glutSetWindow(MainWindow);
  glFinish();
  glutDestroyWindow(MainWindow);
  // 13. clean everything up:
  clReleaseKernel(Kernel);
  clReleaseProgram(Program);
  clReleaseMemObject(dPobj);
  clReleaseMemObject(dCobj);
  exit(0);
}

typedef float point;
typedef float vector;
typedef float color;
typedef float sphere;

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

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

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

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

Jane Parallel's Performance