CS 475/575 -- Spring Quarter 2022

Project #7A

Do 7A *or* 7B -- You cannot get credit for both

OpenCL / OpenGL Particle System

120 Points

Due: June 7 -- 23:59:59 -- No Bonus Days

This page was last updated: March 26, 2022

Introduction

Particle systems are used in games, movies, etc. to depict phenomena such as clouds, dust, fireworks, fire, explosions, water flow, sand, insects, wildebeests, etc. Once you know what they are, you can't stop seeing them.

To make a particle system work, you manipulate a collection of many 3D particles to exhibit some behavior. (Look here for more information.)

A bit of particle history: particle systems were first seen in the *Star Trek II: The Wrath of Khan* Genesis Demo. (Don't laugh -- that sequence was animated 40 years ago!)

In this project, you will use OpenCL and OpenGL together to make a cool particle system. (The degree of cool-ness is up to you.)

Getting the Visual Studio solution:

The file Particles2019Template.zip contains a complete Visual Studio 2019 solution for 1 M particles, one sphere bumper, and no color changes. You need to add to these files to complete the project. The files you need to change are sample.cpp and particles.cl . You need to make a second bumper and apply some sort of color change during the animation.

What the sample code does

The sample code already does the following: (this is included here if you want to follow the logic)

1. Here are the structs that will hold particle positions and particle colors:

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

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

```
};
```

Arrays of these structs will be created in GPU memory.
Why the .w and .a members of the structs? The .w is the homogeneous coordinate for positions. Most of the time it is 1.0, but it does have other uses. The .a is alpha, or transparency. It is not used here, but it could have been. So, those elements are here to allow for future expansion.
The velocities use the xyzw struct too, just for convenience.

2. Design your 3D environment. The particles need to start from somewhere. Where should that be? The particles need to start with initial velocities. What should those be?

3. Create an OpenGL buffer object to hold the particles' XYZW positions as an array-of-structures.
Create an OpenGL buffer object to hold the particles' RGBA colors as an array-of-structures.
Create a C++ array-of-structures to hold the particles' XYZ velocities.
(OpenGL buffer objects are used for position and color, but not velocity. This is because OpenGL will need to use positions and colors in the drawing, but not the velocities.)

4. Determine good starting values for the position, velocity, and color data structures.
For the position and color buffers, use glMapBuffer( ) to send the values directly to the GPU.
The velocity array is a C++ array-of-structures, so just fill it using C++ code.

```
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( .3f, 1. );
        colors[i].g = Ranf( .3f, 1. );
        colors[i].b = Ranf( .3f, 1. );
        colors[i].a = 1.;
}
glUnmapBuffer( GL_ARRAY_BUFFER );


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 );
}
```

5. Create two OpenCL buffers from the position and color OpenGL buffers using calls to **clCreateFromGLBuffer( )**. You don't need to transmit the data to these OpenCL buffers -- it is already there in the OpenGL buffers that they are now linked to.

6. For the velocity array, create an OpenCL buffer using **clCreateBuffer** and enqueue-transmit the C++ array-of-structures to it like you've done before using **clEnqueueWriteBuffer( )**.

7. Decide on a time step, **DT**. Your .cl program will need to know about it.

8. Create a .cl OpenCL kernel that will advance all of the particles by the timestep **DT**. The sample code shows giving the .cl program access to the particles' positions and velocities. You will need to use these, and will need to update them.

Requirements:

1. Your 3D environment needs to have at least ***two*** "bumpers" in it for the particles to bounce off of. Each bumper needs to be geometrically designed such that, given a particle's XYZ, you can quickly tell if that particle is inside or outside the bumper. To get the bounce right, each bumper must be able to know its outward-facing surface normal everywhere.

   Let's face it. Spheres are computationally "nice". In computer graphics, we *love* spheres. It is fast and straightforward to tell if something is inside or outside a sphere. Determining a normal vector for a point on the surface of a sphere is straightforward too.

   It is OK to assume that the two bumpers are separate from each other, that is, a particle cannot be colliding with both at the same time.

2. Your OpenCL .cl program must also handle the bounces off of your (>=2) bumpers. Be sure to draw these bumpers in your .cpp program so that you can see where they are. Again, spheres are "nice", and there are already functions available to draw them. This code goes in your program's ***Display( )*** function.

```
glColor3f( .9f, .9f, 0. );      // yellow
glPushMatrix( );
        glTranslatef( -100., -800., 0. );       // sphere center
        glutWireSphere( 600., 100, 100 );       // sphere radius, slices, stacks
glPopMatrix( );
```

   And then do this again for the second sphere.

3. The sample OpenCL code does not get the colors, change them, or put them back. But, *your* .cl kernel needs to dynamically change the color of the particles. You could base this on position, velocity, time, bounce knowledge, etc. ***But, the color of each particle needs to change in some predictable way during the simulation.***
   Note that OpenGL defines the red, green, and blue components of a color each as a floating-point value between 0. and 1.

4. Leave the local work-group size at some fixed value. You can pull this out of your experience with Project #6, or you can experiment with the timing.

5. Vary the total number of particles from something small-ish (~1024) to something big-ish (~1024*1024) in some increments that will look good on the graph.

6. Measure the performance using units that make sense. Following our class tradition of inventing our own cool-sounding units of measure, Joe Parallel used "Particles Per Second", or "MegaParticles Per Second", or "GigaParticles Per Second".

7. Make a table and a graph of Performance versus Total Number of Particles. Note that this will just be one graph with one curve.

8. Make a video of your program in action -- be sure your video is **Unlisted**.. You can use any video-capture tool you want. If you have never done this before, I recommend Kaltura, for which OSU has a site license for you to use. You can get the Kaltura noteset by clicking here. If you use Kaltura, be sure your video is set to **Unlisted.** If it isn't, then we won't be able to see and it and we can't grade your project.

9. Turn into *Teach*:
   1. Your source code (.cpp and .cl).
   2. Your commentary in a PDF file that is not inside a .zip file..

10. Your commentary PDF should include:
    1. **A web link to the video showing your program in action -- be sure your video is Unlisted.**
    2. What machine you ran this on
    3. What predictable dynamic thing did you do with the particle colors (random changes are not good enough)
    4. Include at least one screen capture image of your project in action

5. Show the table and graph
6. What patterns are you seeing in the performance curve?
7. Why do you think the patterns look this way?
8. What does that mean for the proper use of GPU parallel computing?

Advancing a Particle by DT

In the sample code, Joe Parallel wanted really badly to make the code look cleaner by treating (x,y,z) positions and velocities as single variables. To do this, he typedef'ed new variable types called point, vector, and color and took advantage of OpenCL's float4 variable type. (Unfortunately, there isn't a float3.)
He also stored a sphere definition as a float4: x, y, z, r.

```
typedef float4 point;           // x, y, z, 1.
typedef float4 vector;          // vx, vy, vz, 0.
typedef float4 color;           // r, g, b, a
typedef float4 sphere;          // x, y, z, r

constant sphere Sphere1 = (sphere)( -100., -800., 0.,  600. );
```

Joe Parallel also stored the (x,y,z) acceleration of gravity in a float4 and hard-coded a time step:

```
constant float4 G       = (float4) ( 0., -9.8, 0., 0. );
constant float  DT      = 0.1;
```

Now, given a particle's position **point p** and a particle's velocity **vector v**, here is how you advance it one time step:

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

        point  p = dPobj[gid];
        vector v = dVel[gid];
        color  c = dCobj[gid];

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

Bouncing is handled by changing the velocity vector according to the outward-facing surface normal of the bumper at the point right before an impact:

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

And then do this again for the second bumper object.

Assigning the new positions and velocities back into the global buffers happens like this:

```
        dPobj[gid] = pp;
        dVel[gid]  = vp;
        dCobj[gid] = ????;      // some change in color based on something happening in the simulation
```

Some utility functions you might find useful:

```
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 = fast_normalize( n );
	vector out = in - n*(vector)( 2.*dot(in.xyz, n.xyz) );  // angle of reflection equals
	                                                        // angle of incidence
	out.w = 0.;
	return out;
}


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

Make this Program Your Own

There is a constant character string near the top of the program called **WINDOWTITLE** where you set the string that goes in the graphics window title bar. Change *Joe Parallel* to your name.

Are you Getting an Error Message that says something about "UTF-8"?

This is the problem where Windows text editors put 2 marks at the end of a text line instead of the expected one mark. Refer to **Slide #43** of the **Project Notes** noteset.

Determining Platform and Device

The sample code includes code from the *printinfo* program. This will show what OpenCL capabilities are on your system. The code will also attempt to pick the best OpenCL environment. Feel free to change this if you think it has picked the wrong one.

Seeing Joe Parallel's Animation

Click here to see Joe Parallel's animation.
(It might work better to right click on the link and store the mp4 file locally and run it from there.)

Grading:

Feature	Points
Convincing particle motion	20
Bouncing from at least two bumpers	20
Predictable dynamic color changes (random changes are not good enough)	30
Performance table and graph	20
Commentary in the PDF file	30

Potential Total	**120**

Note: The motion, bouncing, and colors of the particles needs to be demonstrated with a link to a video. If it a Kaltura video, be sure it has been set to *Unlisted*.