Speed Comparison of Three Simulation Methods using GLSL

Jeffrey Bowles
Department of Computer Science
University of New Mexico
jbowles@riskybacon.com

September 28, 2012

1 Introduction

It is currently very popular to perform simulations on a Graphics Processing Unit (GPU) using CUDA or OpenCL. These tools are very nice to use as they do not require a knowledge of OpenGL, nor do they require that a GL context be created to use the GPU. However, application developers that have a working knowledge of OpenGL may want to perform a simulation and use the results in an OpenGL program. These developers are already familiar with using OpenGL function calls to move data on and off the GPU and are already familiar with writing shader programs on the GPU. These application developers do not need CUDA or OpenCL to perform these simulations, and they may not want to learn a new set of terminology, syntax and idioms to get the job done.

A quick search of the literature shows that there is more than one way to perform simulations using OpenGL and GLSL, but it isn't clear which method is the fastest. This paper discusses three different methods and their benchmarks. The methods used are Transform Feedback, Vertex Texture Fetch, and Pixel Buffer Object.

2 Methodology

All methods use two passes. The first pass computes the new positions of the particles and the second pass renders the particles. Each method was written to be as similar as possible to the other methods to ensure a fair comparison The tests were run on a 17" MacBook Pro, 2.3GH Intel Core i7, 8GB memory, AMD Radeon HD 6750M with 1G memory.

An OpenGL 3.2 core profile was used, with GLSL 1.5.

In all three cases, RK4 is used to determine new particle positions and velocities. The simulation is that of a gravity well. Half of the particles are started at (0, 0.1, 0) and the other half at (0, -0.1, 0). Initial velocities all have

the same magnitude, all particles have the same mass, and a gravity well exists at (0,0,0). While all particles have the same velocity magnitude, their directions are random.

3 Transform Feedback

This method was the slowest, but also the easiest to understand. The Transform feedback buffer is used in the first pass to update particle positions. This is the easiest method to reason about because the positions are always stored as a list of vertices. However, it was the hardest to develop because there aren't any good examples of how to use the transform feedback buffer in OpenGL 3.2.

4 Vertex Texture Fetch

This method was the fastest. Particle positions and velocities are stored in two RGBA floating point texture maps and are attached to an FBO. Two FBOs are created and "ping-ponging" between the two for each update and render pass is used to avoid writing to the same texture map that is being read.

Particle system updates are performed in the fragment shader and Multiple Render Targets (MRT) are used to write both the new position and the new velocity.

A Vertex Array Object (VAO) is that has a buffer object attached that contains a vertex for each particle that is represented in the texture map. Each vertex has a single, static attribute. This attribute is the texture coordinates used to index into the particle position texture map. It is important to generate texture coordinates that point to the center of the texel.

In the vertex shader, the texture coordinate attribute is used to read the particle's position out of the position texture map.

5 Pixel Buffer Object (PBO)

This was the middle-of-the road solution. As with the vertex texture fetch solution, two FBOs are created with position and velocity texture attached to the outputs.

Particle system update are performed in the fragment shader and MRTs are used to write both the new position and the new velocity.

As with the VTF method, a VAO is created with a buffer object that contains a vertex for each particle. The difference is that instead of texture coordinates as the attribute for each particle, the attribute will be the position of the particle. This eliminates the need to perform a texture lookup in the vertex shader. However, it does require a copy operation by the GL.

It is hoped that this copy operation will be a pointer change, and not an actual copy, but the results indicate that a copy occurs.

Using the pixel pack buffer, the positions in the texture map are copied into the buffer object. One might assume that the best performance would be acheived by marking the buffer object as GL_STREAM_COPY during the glBufferData() call, but through trial and error it was discovered that GL_STATIC_DRAW was by far the best method.

6 Results

Each version of the simulation was run for 1000 steps. Each particle was rendered as a point sprite with the size specified as glPointSize(1.0f). The number of particles is represented in millions and the frames per second were measured.

Particles	Transform Feedback	Vertex Texture Fetch	Pixel Buffer Object
0.25	200	500	333
0.50	100	333	250
0.75	50	250	167
1	40	200	125
2	22	100	71
3	13	67	50
4	10	52	41.7
5	8	41.7	29.4
6	6.8	33.3	24.4
7	5.7	29.4	20.8
8	5.1	26.3	18.5
9	4.5	22.7	16.4
10	4	20.8	14.7
11	3.7	19.23	7.1
12	3.4	16.7	6.25
13	3.2	16.1	failed
14	2.9	failed	failed
15	2.7	failed	failed

Failure was defined by runs that failed to produce any meaningful output, such as a black screen or garbled output. I believe that the issue is round-off error which caused a failure to fetch and update the proper texels.

7 Future Work

The performance tests should be ported to other systems. This should not be too difficult as the dependencies are all cross platform.