Simulation Techniques for Animation: GPU Accelerated Mass Spring System

Joe Withers*

Abstract

During this project I explored the feasibility of offloading computation onto the GPU, in soft body Mass-spring system simulations, focusing primarily on techniques that make use of OpenGL compute shaders. This report documents my implementation of such techniques, as well as my findings.

1 Implementation Diary

I used C++ and OpenGL to develop my implementation, utilising the NCCA Graphics Library NGL [Macey 2014] to interact with OpenGL, and Qt5 to create the user interface. I based my implementation around a demo of a Mass Spring System using RK4 (Runge-Kutta 4th Order) integration [Macey 2015], that uses NGL and Qt5 for it's user interface. I was able to use many aspects this implementation as 'boilerplate code', such as the camera movement and passing of basic geometry to OpenGL, which saved me a lot of time and allowed me to focus on developing the simulation itself. This implementation also provided an example of RK4 integration and calculations of spring forces according to Hooke's law, though these needed to be altered to make them suitable for use with GPU computation.

Before starting to implement the system on the GPU, I first had to establish what the data structures for the Mass-spring system would be, and to do this I looked at tutorial material covering game physics systems [Fiedler 2004a] hosted by Gaffer On Games. This provided me with an understanding of the necessary data I would need to store for each mass object in the system. I determined that each mass object would be a struct of two vectors:

- `Vec3` Position - This stores the position of the mass in world space coordinates.
- `Vec3` Velocity - This stores the combined speed and direction at which the mass is travelling.

To determine the data required to represent the springs I looked at tutorial material covering implementations of spring physics in game engines [Fiedler 2004b], hosted by Gaffer On Games. This article explains that the calculation of damped spring forces, according to Hooke's law, can be calculated by following:

$$F = -k(|x| - d)(x/|x|) - bv \tag{1}$$

where:

F = The force to apply to each of the mass points.
k = The spring constant.
$|x|$ = The distance between the two mass points.
d = The resting length of the spring.
b = The damping coefficient for the spring.
v = The relative velocity between the spring points.
I then determined that the data for the spring object could be represented with the following struct:

- `Unsigned Int` Start Index - This stores an integer that represents one of the connected masses, as an index into the array of masses.
- `Unsigned Int` End Index - This stores an integer that represents the other connected mass, as an index into the array of masses.
- `float` Resting Length - This stores the original resting length of the spring.
- `Vec3` Relative Velocity - This stores the combined speed and direction at which the masses are travelling to or from each other.

For my first attempt at GPU implementation I decided to store each the attributes for each mass struct and each spring struct in OpenGL textures, and use Image Load/Store commands for accessing and manipulating data. I implemented the following attributes as 1D textures:

- Mass positions - `Vec3, RGBA32F`, Size = Number of Masses.
- Spring velocity - `Vec3, RGBA32F`, Size = Number of Springs.
- Spring resting length - `float, R32F`, Size = Number of Springs.
- Spring start index - `unsigned int, R16UI`, Size = Number of Springs.
- Spring end index - `unsigned int, R16UI`, Size = Number of Springs.

To initialise these textures, I first dispatch a compute shader with workgroups equal to the dimensions of my JelloCube object. This sets the initial position for the masses in the mass positions texture, whilst also counting how many springs need to be created using an atomic counter. I then read the contents of atomic counter and create the textures necessary for spring attributes, whose length is equal to the number of springs counted by the atomic counter. The atomic counter is then reset, and the compute shader is then dispatched for a second time, however this time it calls a subroutine which stores the spring attributes into each corresponding texture.

Whilst this approach works, it relies on maintaining six textures at once, which is quite unwieldly. I also found that once the number of masses is increased past 1000, the number of springs needed to be stored exceeds the number of texels I can specify with glTexImage1D, at least on my system. I therefore decided to switch to using Shader Storage Buffer Objects, as their size is only limited by the amount of memory on the GPU, and they are much easier to maintain and manipulate compared to textures.

I have since changed the system to use Shader Storage Buffer Objects, which I have found to be much faster, allowing up to 1.5 million springs to be calculated at once with interactive framerates.

2 Research Report

References

FIEDLER, G., 2004. Physics in 3d. `https://gafferongames.com/post/physics_in_3d/`.

FIEDLER, G., 2004. Spring physics. `https://gafferongames.com/post/spring_physics/`.

*e-mail:joewithers96gmail.com

MACEY, J., 2014. Ngl the ncca graphics library. `https://github.com/NCCA/NGL`.

MACEY, J., 2015. Mass spring system using rk 4 integration. `https://github.com/NCCA/MassSpring`.