Homework 2 Report

Arrian Chi

Abstract— **In this midterm report, I present my current progress in the projects, the problems I encountered, and how I managed to mitigate each problem. I will also give an overview of what I plan to complete in the next coming weeks.**

I. Overview

In my proposal, I mentioned I wanted to create a cloth simulation for my hobby project, AlienGLRenderer. This is essentially a C++ implementation of the DEP/S algorithm we gone over class. However, much emphasis was made on making sure the cloth simulation could run in real-time (online rendering). In the following sections, I will discuss the progress I have made so far, the problems I encountered, and what I plan to do in the next coming weeks.

II. Progress

Currently, I have a working implementation of the cloth simulation in C++. I will discuss the key components of the simulation below:

A. *Cloth Mesh Creation and Parameterization*

The mesh of the cloth is represented as a equilateral triangluar grid of particles connected by edges / hinges. For now, I rendered them points connected by lines, highlighting the points that are fixed in the simulation. Other highlights present in the simulation are used for debug purposes only. The derivation of the points are all procedural and dependent on a user supplied number of edges of the cloth and the edge length. In addition, the user also specifies the young's modulus, the thickness, the gravitational pull, and the timestep of the simulation.

B. *Cloth Simulation*

The simulation is based on the DEP/S algorithm we went over in class. The simulation is implicit and is solved using the Newton-Raphson method. The forces acting on the cloth are the internal forces (stretching, bending, and shearing), the external forces (gravity), and the damping forces. The internal forces are calculated using the DEP algorithm, which is a generalization of the StVK model. The external forces are calculated using the gravitational pull and the damping forces are calculated using the velocity of the particles. The forces are calculated in the compute shader and the positions are updated in the update shader. The positions are updated using the implicit Euler method. The simulation is run until the accumulated error in the free forces is less than a certain tolerance.

C. *Current Performance*

The current implementation of the cloth simulation is technically realtime. (Soime figure) shows the initial frame times of the calculation. As you can see, not only is it true that the compute time increases as the number of vertices increases, but also the rate of the compute time is doubling for every square of n. This is clearly exponential growth. While it is true that the DEP algorithm is $O(n^2)$ where n is the number of vertices, we must also consider the growth of the number of hinges and edges as well.

III. Problems Encountered

I encountered numerous amounts of problems as I implemented the cloth simulation. I will discuss the most significant ones here.

A. *Renderer Design*

As I was developing the cloth simulation, I realized that my renderer implementation was coupled. The initial idea of my renderer design was that the client would first pass in a Scene object into a render call, then the renderer would call the OpenGL draw calls for each of the primitives in the scene. In OpenGL (which could be contextualized as an highly configurable state machine), one must supply state to the OpenGL context before a draw call is executed. The usual state for draws (which is bound to the context) includes the shader programs to be executed on the GPU, the vertex array objects specifying the format of the vertex data, the buffers containing the vertex data (the positions, the indices to those positions if using indexed rendering), and other buffers/data that the shaders need. The key point here is that the vertex array object had to be created for every known object type in my renderer because each new object had vertex data in different formats. Because I elected to have the renderer responsible for all drawing of the primitives, I stored all the vertex array objects forr each type there. So if I had to add a new object type (like my cloth), I was forced to modify the renderer class as well. This directly goes against the idea of the Open-Closed Principle in software design. A solution to this would've been to use a visitor design pattern to render the objects so that the renderer would just call the render function of the actual object, with no distinction of what each object is.

B. *Subtlties in C++ and OpenGL*

I was caught up in a few subtlties in the C++ language and OpenGL. For instance, I spent a huge amount of time relearning what C++ templates are. I built my cloth system derived from a particle system and I wanted to allow the user to specify different parameters per system. I thought that I could use templates to allow the user to specify the different parameters structs to the different systems at compile time. This was a pain to code though. Because templates were expanded upon instantiation of the object (it was a template that gets expanded into a class), finding which line in my template was wrong was a debugging nightmare. It also proved to be a less effective design than I thought it would be too…

Another subtlety I ran into was memory alignment issues in OpenGL. To start, the DEP algorithm assumes that the DOF vector is a $3 \times n$ by 1 column vector, where n is the number of vertices. Naturally, a naive implementation would've been to create an array of 3 by n floats to represent the DOF vector used for calculating the forces and use the same DOF vector for drawing (specifying the vertex format to assume a vertex stream of vec3). This would work if I didn't use my DOF vector in the other (compute/debug) shaders (specifying the DOF vector as a shader buffer storage object (SSBO)).

Before going further and to make things clear, I have a two shaders (programs run on a GPU): one shader used for the rendering of the cloth itself (cloth shader), another for highlighting the certain primitives such as edges, hinges, and fixed points (highlight shader). The cloth shader requires the dof positions (containing the actual vertex data) to be passed in as a buffer. However, the highlight shader requires some method of retrieving points to be highlighted, whether it is through pasing the positions in directly or passing in an array of indices to the positions. I chose the latter because I was already updating the buffer for the cloth shader with new positions. Adding another buffer of positions would not only take up more space and be redundant, but also take marginally more compute time. I do acknowledge this assumes that indexing into a SSBO (shader storage buffer object) is faster than transfering the data from the CPU to GPU. Now because I elected to do the latter, I had to deal with memory alignment issues with my buffer.

However, according to the OpenGL specification, when using UBOs or SSBOs, a vec3 is aligned to 16 bytes (4 floats) in GPU memory. This means that in the shader, when I try to index into an array of vec3s, the GPU will retrieve the result with 3 floats plus 4 bytes of padding. So when I copy the chunk of buffer data (the DOF vector) from CPU memory into GPU memory, I had to ensure that the total memory size of the dof buffer is a multiple of 16 bytes. Otherwise, the GPU would read 3 floats from the buffer, skip one float part of another point, and then read the next 3 floats, which is wrong. This means that after my force calculation, I had to create a way to send the data to the GPU such that my buffer acted as a vector of vec4s. This is still a naive answer however.

There is another solution to this issue without aligning the DOF vector to 16 bytes. Instead of having the GPU read the buffer as an array of vec4s/vec3s, I could pass the buffer to the GPU as an array of floats (avoiding the alignment issue entirely). Then the vertex stream for rendering for the shaders (specifically the highlight shader since the dof positions must be read as a SSBO instead of a vertex stream) would be the indices of the particles themselves. Then in the shader, I could use the particle indices to compute the indices of the components of each dof for that particle.

C. *Precision and accumulation of error*

The final problem I had to deal with was precision errors. The original DEP algorithm implemented in python used numpy, which defaulted to using double precision floats in all of its calculations. However, I was using single precision floats in my C++ implementation. When comparing results between the C++ and Python implementations, I noticed that the results of the C++ implementation were way off as the simulation progressed. Through debugging, I found that my implicit simulation loop was iterating more times than the Python implementation. The loop is depndent on whether or not the accumulated error in the free forces were less than the tolerance. If the tolerance or error was a value that was different than the ones used in the Python simulation, then the number of iterations could also be different, leading to different results every time step. This causes slight deviations between the two implementations, which accumulates as time goes on. Simply switching the error and tolerance values to double precision floats steered the iteration counts to be similar. Changing all values to doubles, got the simulation to be almost identical to the Python implementation (any further differences would derive from the methods Eigen uses to compute values i.e. solving matrices, calculating square roots, etc).

IV. Prospective Work

Using the Microsoft Visual Studio performance profiler, I found that the performance of the simulation was bottlenecked by 2 calculations: the solving of the matrix (18% of total CPU time) and computing the gradient and hessians of the bending elastic energies (60% of total CPU time). Here are a couple ways I will go about solving these issues:

A. *Optimizing the Jacobian Calculation*

B. *Optimizing the Bending Energy Gradient and Hessian Calculation*

In the original python implementation of the bending energy gradient and hessian, there are two major areas of

C. *Multithreading the Simulation*

D. *Using the Eigen Library properly*

V. Conclusion