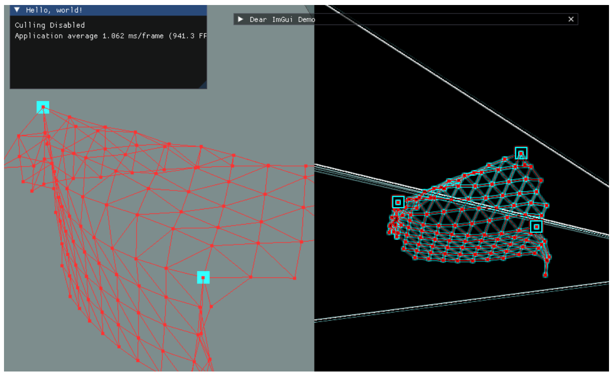
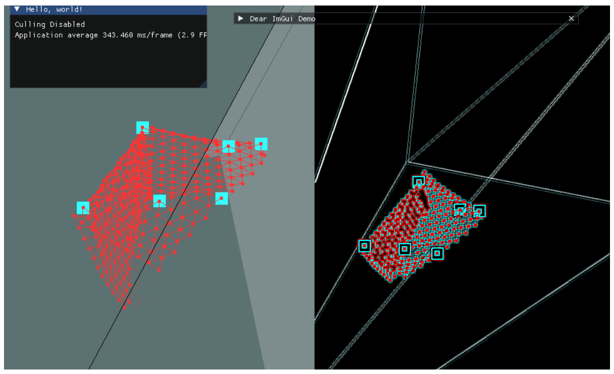
Final Report:
AlienGLRenderer Cloth Simulation

Arrian Chi



```
Culling Disabled
Application average 1.062 ms/frame (941.3 FP
```



```
Culling Disabled
Application average 343.460 ms/frame (2.9 FP
```

***Abstract*— Cloth simulations have been a staple problem in computer graphics for a long time. They are used in movies, video games, and even in scientific simulations. The DEP/S algorithm is a popular algorithm used to simulate cloth in real-time. In this project, I implemented the DEP/S algorithm in C++ using the Eigen library and OpenGL. This report talks about the intricacies and details that come into play when implementing a cloth simulation.**

I. Overview

Research in computer graphics from the past few decades to the present [1] [2] [3] has shown particular interest in the simulation of plates, cloth, and other thin materials. These simulations are used in movies, video games, and even in scientific simulation. Various algorithms have been developed to simulate these materials, such as geometric approximations from cable theory [4], more physically involved methods simulating the cloth as a connected mass-spring damper system [5], and even more physically accurate models con sidering the elastic energies within the discretized simulated cloth [6].

A realization of the last point is the well-known Discrete Elastic Plates / Shells numerical simulation algorithm, shown in Algorithm 1 [7] [1]. To explain, the algorithm discretizes the cloth into particle masses arranged in a mesh. In the simulation loop, where each iteration represents a step in time (thus a change in cloth position), it calculates the forces acting upon the particles to find what the next positions of the particles will be. The algorithm specifically uses the Newton-Raphson method to solve for the positions of the particles, so calculating and solving for the gradient (forces) and Hessians(Jacobian of forces) of elastic energies within the cloth is a necessary step. The elastic energies occur as stretching energies, along the edges between each particle, and bending energies at every edge adjacent to two faces of the mesh. An error is calculated at the end of the method to determine if the calculation has converged, and if not, the simulation continues to iterate until the error is less than a certain tolerance. Once the new position is obtained, the simulation continues to the next time step.

For this project, I wanted to integrate a feature into my hobby renderer, AlienGLRenderer. The renderer's main purpose is to serve as a sandbox for me to implement and showcase graphics features I am interested in the moment and to improve my skills in C++. Thus, I decided to implement cloth simulation feature using the DEP/S algorithm in it. Because my renderer is akin to a game engine, I made it my goal to make the cloth update and render in real-time with OpenGL and C++. A link to the project source can be found here: `https://github.com/dinoplane/gl_alien_renderer/tree/cloth-system`.

In my initial proposal, I posed a few questions I would like my project to answer. Through the course of the project, these questions have drastically changed, so to update, these questions now are:

1) How efficient can a cloth simulation be?
2) How do we simulate a cloth efficiently?
    a) Where could the DEP algorithm benefit from optimization?
        i) What parameters affect the efficiency of the algorithm?
    b) Is multi-threading useful or not?
    c) Could we afford a tradeoff between stability and speed?
    d) What is the source of instability?
3) What technical problems/constraints arise when simulating cloth?
    a) Are there technical complexities during development that were frustrating?

Algorithm 1 Discrete Elastic Plates

```
Require: $q(t_i), \dot{q}(t_i)$                                DOFs and velocities at $t_i$
        $e(m), h(n)$                                m edges and n hinges present in the plate
        $m, M$                                      masses of each particle (as a vector and matrix)
        $l_k, k_s$                                  undeformed length / stretching for each edge
        $\bar{\theta}, k_b$                         rest angles of each hinge, bending stiffness
        $\Delta t$                                  time step
        $\mathbf{f}^{\mathbf{ext}}$                 external forces
        free_index                                  Index of the free DOFs

Ensure: $q(t_{i+1}), \dot{q}(t_{i+1})$                         DOFs and velocities at $t = t_{i+1}$

 1: function COMPUTE_STRETCHING($q, e, l_k, k_s$)
 2:     $\mathbf{f}^{\mathbf{stretch}}, \mathbf{J}^{\mathbf{stretch}} \leftarrow 0, 0$      Initialize
 3:     for $i \leftarrow 1$ to $m$ do
 4:         x0, x1 $\leftarrow q(e(k,1)),\ q(e(k,2))$      Get DOFs of particles in edges
 5:         $\mathbf{f}^{\mathbf{grad}}, \mathbf{J}^{\mathbf{hess}} \leftarrow$ GRADES_HESSES(x0, x1, $l_k(i), k_s(i)$)      Refer to Appendix of [7]
 6:
 7:         $\mathbf{f}^{\mathbf{stretch}}(e(k)) \leftarrow \mathbf{f}^{\mathbf{stretch}}(e(k)) - \mathbf{f}^{\mathbf{grad}}$      Update total force/Jacobian
 8:         $\mathbf{J}^{\mathbf{stretch}}(e(k), e(k)) \leftarrow \mathbf{J}^{\mathbf{stretch}}(e(k), e(k)) - \mathbf{J}^{\mathbf{hess}}$
 9:     return $\mathbf{f}^{\mathbf{stretch}}, \mathbf{J}^{\mathbf{stretch}}$
10:
11: function COMPUTE_BENDING($q, h, k_b$)
12:     $\mathbf{f}^{\mathbf{bend}}, \mathbf{J}^{\mathbf{bend}} \leftarrow 0, 0$      Initialize
13:     for $i \leftarrow 1$ to $n$ do
14:         x0, x1, x2, x3 $\leftarrow q(e(k,1)),\ q(e(k,2)),\ q(e(k,3)),\ q(e(k,4))$      Get DOFs of particles in hinges
15:         $\mathbf{f}^{\mathbf{grad}}, \mathbf{J}^{\mathbf{hess}} \leftarrow$ GRADEB_HESSEB(x0, x1, x2, x3, $l_k(i), k_s(i)$)      Refer to Appendix of [7]
16:
17:         $\mathbf{f}^{\mathbf{bend}}(e(k)) \leftarrow \mathbf{f}^{\mathbf{bend}}(e(k)) - \mathbf{f}^{\mathbf{grad}}$      Update total force/Jacobian
18:         $\mathbf{J}^{\mathbf{bend}}(e(k), e(k)) \leftarrow \mathbf{J}^{\mathbf{bend}}(e(k), e(k)) - \mathbf{J}^{\mathbf{hess}}$
19:     return $\mathbf{f}^{\mathbf{bend}}, \mathbf{J}^{\mathbf{bend}}$
20:
21: function DISCRETE_ELASTIC_PLATES($q, \dot{q}$)
22:     Guess: $q^{(1)}(t_{i+1}) \leftarrow q(t_i)$
23:     $n \leftarrow 1$
24:     while error > tolerance do
25:
26:         $\mathbf{f}^{\mathbf{stretch}}, \mathbf{J}^{\mathbf{stretch}} \leftarrow$ COMPUTE_STRETCHING($q, e, l_k, k_s$)
27:         $\mathbf{f}^{\mathbf{bend}}, \mathbf{J}^{\mathbf{bend}} \leftarrow$ COMPUTE_BENDING($q, e, l_k, k_s$)
28:
29:         $\mathbf{f}^{\mathbf{tot}} \leftarrow \mathbf{f}^{\mathbf{bend}} + \mathbf{f}^{\mathbf{stretch}} + \mathbf{f}^{\mathbf{ext}}$      Aggregate forces
30:         $\mathbf{f} \leftarrow \frac{1}{\Delta t} m \odot \left[\frac{1}{\Delta t}\left[q^{(n)} - q^{(1)}\right] - \dot{q}\right] - \mathbf{f}^{\mathbf{tot}}$
31:
32:         $\mathbf{J}^{\mathbf{tot}} \leftarrow \mathbf{J}^{\mathbf{bend}} + \mathbf{J}^{\mathbf{stretch}}$      Aggregate Jacobians
33:         $\mathbf{J} \leftarrow \frac{1}{\Delta t} M - \mathbf{J}^{\mathbf{tot}}$
34:
35:         $\mathbf{f}_{\text{free}} \leftarrow \mathbf{f}$(free_index)
36:         $\mathbf{J}_{\text{free}} \leftarrow \mathbf{J}$(free_index, free_index)
37:
38:         $\Delta q_{\text{free}} \leftarrow \mathbf{J}_{\text{free}}^{-1} \mathbf{f}_{\text{free}}$
39:         $q^{(n+1)}$(free_index) $\leftarrow q^{(n)}$(free_index) $- \Delta q_{\text{free}}$      Update free DOFs
40:         error $\leftarrow$ sum($|\mathbf{f}_{\text{free}}|$)
41:         $n \leftarrow n + 1$
42:
43:     $q(t_{i+1}) \leftarrow q^{(n)}(t_{i+1})$      Update DOFs for next time step
44:     $\dot{q}(t_{i+1}) \leftarrow \dot{q}^{(n)}(t_{i+1})$
```

This is a lot of questions, so I will organize the report into 3 sections: Results, Process, and Issues.

II. Results

I tested my simulation on an AMD Ryzen 9 7900 (with integrated graphics). The configuration of the mesh was akin to that of a table cloth (see figure **??**) There are three iterations of my simulation: a dense matrix version, a sparse matrix version, and PARDISO solver version. A table of the data collected can be seen in figures **??**. The data is displayed on a graph in figures **??**. As one can see, each version's frame times are vary widely, each incurring improvement over the next.

Using the dense matrix version as a base, the sparse matrix incurred an $8\times$ speedup and the PARDISO solver version incurred a $10\times$ max speedup. These speedups are drastic and just goes to show how much dense matrices can be a bottleneck in the simulation. Comparing the sparse matrix version to the PARDISO solver version, the PARDISO solver version is at least $1.3\times$ faster than the sparse matrix version. This is expected since the PARDISO solver is more specialized towards solving the symmetric sparse matrices that the DEP algorithm produces.

The graphs also tell a telling story, the dense matrix version frame time grows consistently with the predicted $O(n^3)$ time complexity (mostly from the matrix solve), breaking 800ms per frame at 529 nodes. The sparse matrix and PARDISO solver versions grow at a much flatter linear-like rate, with the PARDISO solver breaking 81ms per frame at 529 nodes compared to the sparse matrix version's 87ms per frame.

Finally, it is worth mentioning that the ratio between the force calculation time and the solver time was consistently less than 1 after 81 nodes. This further proves that the solver is the bottleneck of the simulation in that version. On both the sparse matrix and PARDISO solver versions, the ratio rose and fell. This suggests that starting with a small number of nodes, the solver produced enough overhead that made it slower than the force calculation, but as the number of nodes increased, the force calculation became the bottleneck. This would not last long when the number of nodes increases even further, we are introduced to the solver bottleneck again.

Side Length	# Nodes	# DOFs	#Edge	# Hinge
1	4	12	5	1
2	9	27	16	8
3	16	48	33	21
4	25	75	56	40
5	36	108	85	65
6	49	147	120	96
7	64	192	161	133
8	81	243	208	176
9	100	300	261	225
10	121	363	320	280
11	144	432	385	341
12	169	507	456	408
13	196	588	533	481
14	225	675	616	560
15	256	768	705	645
16	289	867	800	736
17	324	972	901	833
18	361	1083	1008	936
19	400	1200	1121	1045
20	441	1323	1240	1160
21	484	1440	1365	1281
22	529	1575	1496	1408

Fig. 1: A table of plate properties per side length #

# Nodes	Dense	Sparse	PARDISO
4	2.14568	1.9417	1.97905
9	2.09046	1.86291	2.07927
16	2.19543	1.87658	1.90431
25	2.26691	1.9805	2.17057
36	2.32498	2.02624	2.2378
49	2.26632	2.17572	2.42311
64	2.55798	2.38529	3.17389
81	3.13174	3.07915	3.64856
100	4.36049	3.40046	5.1266
121	10.8998	5.11832	7.15082
144	29.2132	5.81131	11.5085
169	38.4439	14.5564	13.0989
196	58.0018	23.1701	22.7056
225	72.0361	27.0657	27.7778
256	117.31	33.3114	32.7209
289	133.053	44.3635	36.9456
324	210.592	50.7165	39.6869
361	266.549	66.6435	47.0635
400	323.052	81.0222	60.7701
441	574.367	84.1108	65.9397
484	638.038	92.7115	74.1809
529	873.053	105.039	81.0643

TABLE I: A table of average frame times (ms/frame) per plate per simulation

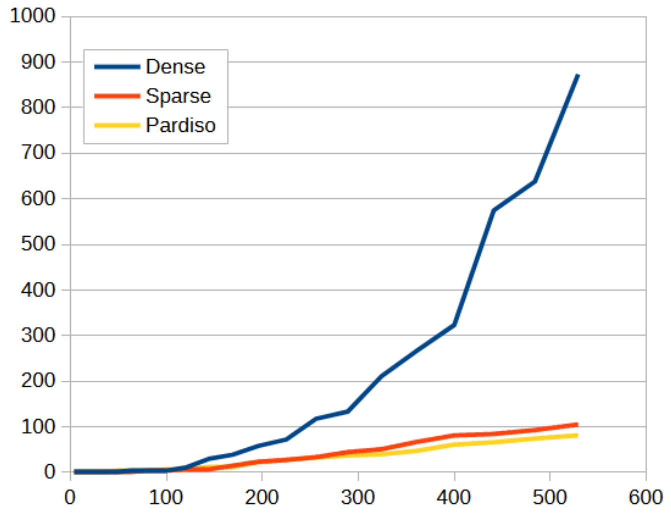


Fig. 2: Graph of average frame times per simulation

# Nodes	Dense	Sparse	PARDISO
4	1.189	1.178	0.031
9	4.007	0.328	0.124
16	3.856	2.932	0.323
25	3.166	1.867	0.468
36	2.496	1.728	0.649
49	1.726	1.521	0.741
64	1.294	1.145	0.862
81	0.982	1.223	0.912
100	0.689	0.955	0.880
121	0.565	0.895	1.007
144	0.448	0.827	1.131
169	0.322	0.891	1.101
196	0.298	0.734	1.118
225	0.237	0.969	1.159
256	0.189	0.904	1.179
289	0.175	0.627	1.194
324	0.143	0.629	1.115
361	0.123	0.544	1.305
400	0.110	0.575	1.210
441	0.102	0.703	1.134
484	0.085	0.549	1.140
529	0.075	0.513	1.069

TABLE II: A table of ratio of force calculation time to solver time

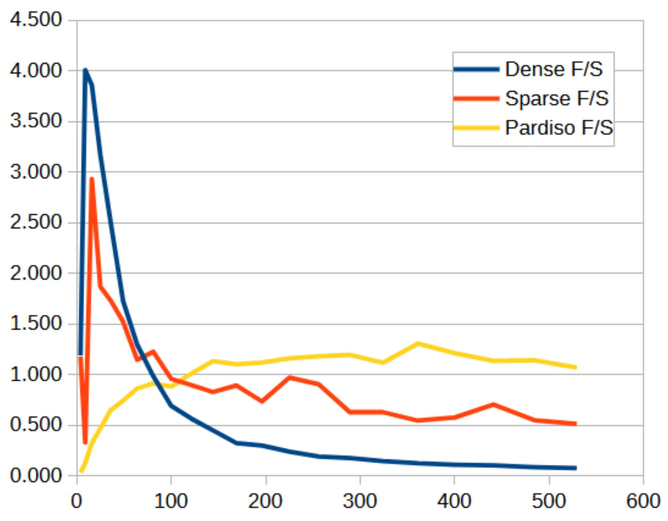


Fig. 3: Graph of ratio of force to solver time

# Nodes	D / S	D / P	S / P
64	1.072	0.806	0.752
81	1.017	0.858	0.844
100	1.282	0.851	0.663
121	2.130	1.524	0.716
144	5.027	2.538	0.505
169	2.641	2.935	1.111
196	2.503	2.555	1.020
225	2.662	2.593	0.974
256	3.522	3.585	1.018
289	2.999	3.601	1.201
324	4.152	5.306	1.278
361	4.000	5.664	1.416
400	3.987	5.316	1.333
441	6.829	8.710	1.276
484	6.882	8.601	1.250
529	8.312	10.770	1.296

TABLE III: Simulation Speedups (D=Dense, S=Sparse, P=PARDISO)

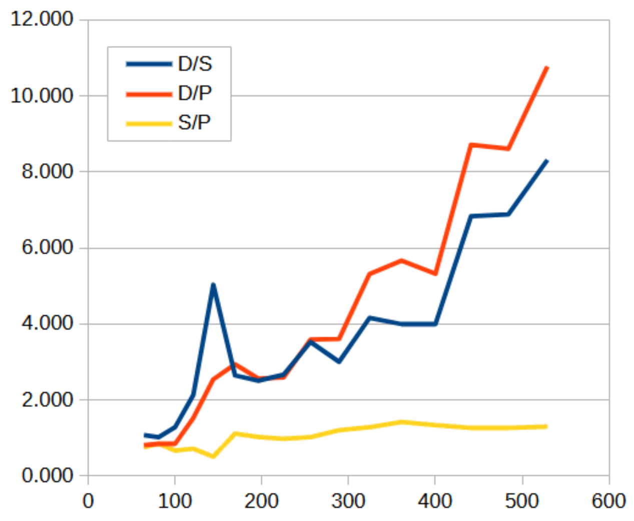


Fig. 4: Graph of simulation speedups

III. Process

In the course of development for the cloth simulation, many amusing and interesting problems arose that I believe warrant discussion. In this section, I will provide a more in-depth explanantion of the implementation and highlight some key implementation details of the cloth simulation.

A. Cloth Mesh Creation and Parameterization

The mesh of the cloth is represented as a equilateral trianglular grid of particles connected by edges / hinges. The point positions are procedurally derived 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.

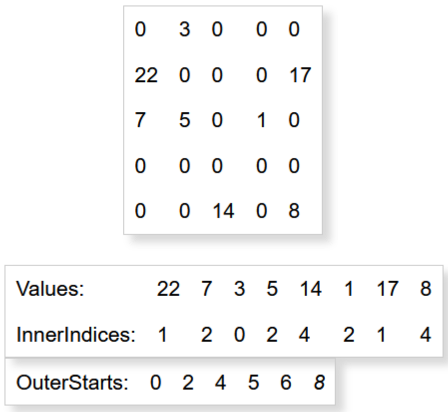
B. First iteration of the Discrete Elastic Plate / Shells Algorithm Implementation

My implementation of the DEP Algorithm started out as a transcription of a python implementation that we went over during class. I leveraged the Eigen Library [8] to handle the linear algebra computations in the algorithm. All calculations used the DenseMatrix object (which internally leverages a 2D array of numbers, a contiguous chunk of floats/doubles in memory). For the solver, I directly solved the matrix using the built in Eigen solver (LDLT solver). The outputs were verified by running the same input with the python implementation and comparing the results. This method of verification proved effective, since it revealed the first issue.

My renderer defaulted to using float (single precision floating point numbers) for all of its calculations. That is a good decision because there was no need for precision in calculations and floats are only 4 bytes in size (the same as a int on most environments). However, upon closer inspection, the python implementation used double precision floats (8 bytes in size). This caused the results of the C++ implementation to be off by a small amount. The error was small enough that it was negligible for small values of n, but as n increased, the error accumulated and the results diverged. This was fixed by changing all the floats to doubles in the C++ implementation. Rendering still used floats, so I had to create a buffer that cast all the double values to floats before sending it to the GPU.

As shown in the results, the performance of initial simulation was abysmal. Both steps of the simulation, the force/Jacobian calculation and the solver, started staggering when the cloth side length was about 13-14 nodes (588-675 DOFs). At this stage, the solver took about 65% of the total CPU time, while the force calculation took about 17%. I believed that these steps can be optimized with multi-threading, but that proved to be slightly wrong.

Eigen already allows users to enable multi-threading capabilities implemented for matrix arithmetic. However, for solving, it only supports multithreaded solves with sparse matrices (without additional extensions [8]). In pursuit of this optimization, I changed the dense matrices involved in the calculation to use sparse matrices. As seen in the previous section, there was a speedup when this was implemented, which I will explain shortly.



0	3	0	0	0
22	0	0	0	17
7	5	0	1	0
0	0	0	0	0
0	0	14	0	8

Values:	22	7	3	5	14	1	17	8
InnerIndices:	1	2	0	2	4	2	1	4
OuterStarts:	0	2	4	5	6	*8*		

Fig. 5: A visual representation of a sparse matrix in the CSR format

C. Second iteration of the DEP/S Algorithm: Sparse Matrices and Multithreading

Sparse matrices are matrices that have a large number of zero values compared to the number of nonzero values. If we used a dense matrix to represent a sparse matrix, there would be a whole lot of wasted space in memory. Additionally, when calculating arithmetic, sparse matrices would take a long time with no value because most of the arithmetic operations would end up being zero.

To solve this issue, sparse matrix implementations usually involve only 3 arrays: One for the nonzero values, one for the row indices of the nonzero values in the matrix, and one for the indices(into the value array) of the first nonzero value in each row per column (see Fig 5). This representation (also known as the Compressed Sparse Format) is much more compact, and because there are less elements to iterate through(skipping through all the zero values), the arithmetic operations and solving algorithms are faster. To create the sparse matrix, it is best to supply the constructor with an array of triplets (row, col, val) for each entry in the desired matrix. So I had to change the force calculations to append these elements to an existing array of triplets (a.k.a. sparse entries) for the Jacobian force matrix (the calculation of the hessians per edge/hinge are still done with fixed-size dense matrices). This method was fine since the bending jacobian and the stretching jacobian are added later in the algorithm and duplicate items in the sparse entry array are summed up. Using this scheme and changing the solver to use the Conjugate Gradient Iterative solver, I secured a 4x speedup (as seen in the above).

I previously mentioned that multithreading Eigen would be a slightly wrong move. In my tests, Enabling Eigen's multithreading did not provide a significant speedup (or any at all). Some theories as to why this happened include: the fact that the matrix arithmetic operations exhibit false sharing (where two threads are writing to the same cache line, causing the cache to be invalidated and rewritten, which is a costly operation) or the overhead of creating threads being too large (implying that Eigen doesn't use a threadpool in its multithreading capabilities) [9]. In the final implementation of this stage, I just disabled Eigen's multithreading capabilities and let the solver run on a single thread, since it proved to be just as good as the multithreaded version. Note that it could be possible for me to multithread the calculation across hinges/edges (assign each thread a set of hinges/edges to calculate the forces/Jacobians), but I did not pursue this optimization due to time constraints.

D. Final Iteration of the DEP/S Algorithm: PARDISO Solver

The final iteration of the DEP/S algorithm involved integrating the PARDISO solver from Panua Technologies [10] [11] [12]. PARDISO is a direct solver that uses the parallel sparse LU factorization method to solve the matrix. In order to fully utilize PARDISO's potential, I had to change the underlying sparse matrix representation to only use the upper triangular part of the matrix, since the solver has optimizations for when the matrix is symmetric. This is implemented as a simple check before adding to the array of triplets (reject if col ¡ row). Once the solver was integrated, the simulation experienced a

E. Rendering

The rendering of the cloth is simple. The cloth is rendered as a mesh of triangles, with each edge outlined with red lines. The fixed nodes of the simulation are highlighted with cyan (and the viewer can clearly see their effect on the rest of the points). In the underlying implementation, to go around the issue of memory alignment(more on this later), I keep track of two arrays of positions: one for the cloth simulation and one for rendering. After the positions are calculated, I copy the positions from the simulation array to the rendering array, and then send the rendering array buffer to the GPU to be rendered. The use of two arrays was a lapse in judgement on my part that I will explain later.

IV. Issues

I encountered numerous amounts of problems as I implemented the cloth simulation. Unlike problems I would like to answer with this project, these are issues that hindered my progress and/or still exist in the project. I will discuss the issues in the order that they were encountered.

A. Renderer Design

As I was developing the cloth simulation, I realized that my renderer implementation was brittle, because it 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 for 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. Some solutions to this include: using a visitor design pattern to render the objects, so that the renderer uses the common interface to render the objects, use a entity component system to separate the rendering logic from the object data, or use a factory pattern to create the vertex array objects for the objects and defining rendering logic for each object type.

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 derived my cloth system from a particle system interface because I wanted to allow the user to specify different parameters per system. Using templates opens the particle system to extension, so the user can customize the parameters and logic to supply to the system. The implementation of this (especially debugging) proved to be more arduious than I thought. Because templates were expanded upon instantiation of the object (it was a template that gets expanded into a class), the line that had the error of code was not indicated in the debug logs. This design also proved to be a less effective than perceived, supporting the idea that there are no zero cost abstractions [13].

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. My initial naive implementation consisted of 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 SSBOs and the std430 layout, 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.

To add on to this, OpenGL actually allows the user to pass an array of doubles in as a vertex stream using `glVertexAttribLFormat` [14] . This means I did not even need to cast the DOF vector from double to float before sending it to the GPU. The cast would still need to be done in the vertex shader since the vertex shader output for positions (`gl_Position`) is specified as a float type. I unfortunately realized this too late, therefore it is not implemented.

Finally, I initially proposed to use OpenGL compute shaders to do the simulation on the GPU. The DEP algorithm (save for the matrix solve) is embarassingly parallelizable, so it would be a good candidate for a compute shader. However, OpenGL does not support matrix operations beyond 4 by 4 dimensions, which means I would need to implement this myself. The conjugate gradient solve is also a parallelizable operation, but this requires significant time to debug (and GPU debugging is much different than CPU debugging since there is no way to step through programs). I concluded that this is a huge undertaking, so I decided to stick with the CPU implementation.

C. Eigen's implementation of sparse matrices

Although Eigen's sparse matrix modules yielded a massive speedup in the simulation, the steps to get there were astounding. The first issue I encountered was that Eigen sparse matrices do not support slicing. Slicing is the operation of taking a set of indices and using them to retrieve the respective rows and columns of a matrix to create a new submatrix. This is a common operation in Python Numpy, and it is supported by Eigen dense matrices too. But the sparse matrix API does not support this operation. Because I couldn't slice the final Jacobian matrix for the relevant free DOFs, I also couldn't blindly add triplets into the sparse entries to create that matrix. The sparse entries needed to be filled such that the final Jacobian matrix was the one directly used in the solver.

The solution comes to life when I made 2 observations. First, the values in the columns and rows of the Jacobian matrix (in the bending/stretching force calculations) corresponding to the fixed DOFs don't contribute to the Jacobian for free DOFs. Second, the only issue that needs attention when directly adding the values to the sparse entries for the Jacobian for free DOFs is the location(new row and column) of the values in the sparse matrix relative to the location of the original destination(old row and column). The new row and column is equal to the difference between the old row and column and how many fixed DOF rows and columns are before the old row and column. Although it is possible to calculate this new row and column at simulation time, it is much easier to just keep track of this old row/col to new row/col mapping in a separate array and index into it when adding the values to the sparse entries.

The second issue I encountered was that PARDISO does not use column major ordering for its sparse matrices. I originally set PARDISO to solve for a unsymmetric real matrix (which is wrong because the Jacobian is a symmetric matrix) and used column major ordering for that matrix (providing PARDISO with direct pointers to the underlying Eigen sparse matrix data). This didn't matter because the transpose of the symmetric matrix is the same as the matrix itself. However, when I changed the solver to solve for a symmetric real matrix (which only needed a upper triangular part of the matrix), I had to change the ordering of the calculation into row major ordering. This was a simple fix, but it was a huge headache to debug because the error messages from PARDISO were not helpful.

D. Degeneracies

It is known that the DEP/S algorithm is prone to some degeneracies, such as when an edge collapses or when the altitude collapses [3]. This may manifest int the calculation as a division by zero error during the solve, an infinite loop (the error never reaches the desired tolerance), or the cloth visually becomes a cloud of points (see figure **??**). These may arise when the solver is unable to reach a valid solution. This is more common in simulations with more DOFs as there is more freedom for numerical instability to occur. In my own tests, the most nodes I found that would reach the resting state without reaching numerical instability is 529 nodes (1575 DOFs, side length = 22). Of course, numerical instability could be mitigated by using a smaller timestep, but this would slow down the simulation (more time steps per second and more work to compute the next position at the next second). Another potential solution would be to check the solver solution before it updates the DOF vector. If the solution is degenerate, special care can be taken (i.e. halve the time step used for the rest of the iterations). Regardless, this problem requires more investigation to find a more robust solution.

V. Remarks

The definition of real-time in this project is ill-posed. The simulation could be considered real-time, but only consist of 4 nodes (not exactly a cloth). The simulation could also be considered real-time if the timestep is large enough that the simulation is not accurate or stable. In the course of development, I kept trying to push the limits of the simulation to see how fast it could improve upon the first implementation. If the simulation was slower than it is now, I would have needed to define the question more concretely.

Accuracy and speed are always at odds with each other it seems. It is possible to get the simulation to run $n = 22$ at 30 FPS if the error tolerance check was completely removed. Visually, there is no significant difference between that simulation and one with the error tolerance check, but the former is not accurate. The simulation is still confined by the nuance of numerical instability however (probably even more so). But now reducing the time step now has less of an effect on speed over all. This observation implies that the explicit integration method is worth looking into for future work.

Finally, the DEP/S algorithm serves as a good basis for other material deforming features such as collision, viscous drag, tearing, friction, etc. Each of these could be implemented as an addition into the external forces or as a complete subroutine integrated into the internal force/jacobian calculation. Regardless, these features also take significant time to implement, debug, and compute (adding more work to a simulation that is already loaded). It may be rewarding to look into these feature in the future.

VI. Conclusion

In this report, I have presented the results of my cloth simulation project, detailed the process of implementing the DEP/S algorithm in C++ using the Eigen library and OpenGL and addressed the numerous issues I encountered during the implementation. The results proved that it is possible to run the DEP/S algorithm in real-time on a cloth with adequately high fidelity. In the future, I may look into optimizing the simulation further, adding more features to the simulation, and investigating solutions to mitigate degeneracies. But for now, I believe a refactoring of the renderer design is in order.

References

[1] D. Baraff and A. Witkin, *Large Steps in Cloth Simulation*. New York, NY, USA: Association for Computing Machinery, 1 ed., 1998.

[2] E. Grinspun, A. N. Hirani, M. Desbrun, and P. Schröder, “Discrete shells,” in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA ’03, (Goslar, DEU), p. 62–67, Eurographics Association, 2003.

[3] R. Tamstorf and E. Grinspun, “Discrete bending forces and their jacobians,” *Graph. Models*, vol. 75, p. 362–370, Nov. 2013.

[4] J. Weil, “The synthesis of cloth objects,” *ACM Siggraph Computer Graphics*, vol. 20, no. 4, pp. 49–54, 1986.

[5] X. Provot *et al.*, “Deformation constraints in a mass-spring model to describe rigid cloth behaviour,” 1995.

[6] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, “Elastically deformable models,” *SIGGRAPH Comput. Graph.*, vol. 21, p. 205–214, Aug. 1987.

[7] S. MK Jawed, Lim, “Discrete simulation of slender structures,” 2023.

[8] G. Guennebaud, B. Jacob, *et al.*, *Eigen: A C++ template library for linear algebra*. Eigen Contributors, 2024. Accessed: 2024-11-20.

[9] L. K. Contributors, *False Sharing in the Linux Kernel*, 2024. Accessed: 2024-11-20.

[10] D. Pasadakis, M. Bollhöfer, and O. Schenk, “Sparse quadratic approximation for graph learning,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 9, pp. 11256–11269, 2023.

[11] A. Eftekhari, D. Pasadakis, M. Bollhöfer, S. Scheidegger, and O. Schenk, “Block-enhanced precision matrix estimation for large-scale datasets,” *Journal of Computational Science*, vol. 53, p. 101389, 2021.

[12] L. Gaedke-Merzhäuser, J. van Niekerk, O. Schenk, and H. Rue, “Parallelized integrated nested laplace approximations for fast bayesian inference,” 2022.

[13] C. Caruth, “There are no zero-cost abstractions.” CPPCon 2019, 2019.

[14] Khronos Group, *OpenGL Documentation*. Khronos Group. Accessed: 2024-11-20.