AN EXPERIMENTAL FAST APPROACH OF SELF-COLLISION HANDLING IN CLOTH SIMULATION USING GPU

by

Jichun Zheng

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Computer Graphics Technology

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. David M. Whittinghill

Department of CGT

Dr. Tim McGraw

Department of CGT

Dr. Christos Mousas

Department of CGT

Approved by:

Dr. Nathan W. Hartman

This thesis is dedicated to my parents and aunt. Sincere thanks for all the support you gave me.

ACKNOWLEDGMENTS

First of all, thanks to Dr. Tim McGraw and Dr. Bedrich Benes from whom I learned all the computer graphics knowledge. Also, thanks to my advisor Dr. David M. Whittinghill for his suggestions to this thesis. Thanks to Dr. Christos Mousas, Prof. Raymond P. Hassan, and Prof. Nasheet Zaman for hiring me as their TA, a great relief to my economic pressure so that this thesis is possible. Finally, thanks my best friends Xiao, Yucong, Yuanpei, Yiyun, and Zhiquan for all the help they give me when I was confused.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

GPU	*Graphics Processing Unit*
GM	*Global Memory*
CM	*Constant Memory*
RK4	*Runge Kutta 4th Order*
CCD	*Continuous Collision Detection*
VF	*Vertex-face*
EE	*Edge-edge*
BVH	*Bounding Volume Hierarchy*
AABB	*Axis-aligned Bounding Box*
Vec3	Vector with three elements

ABSTRACT

The problem of self-collision detection has always been evasive in real time cloth animation as it is very expensive to be implemented. While as rapid as today's evolution of graphics hardware, self-collision handling of cloth is hardly to be seen in almost all kinds of electronic graphics product.

This study describes a fast approach using GPU to process self-collision in cloth animation without significant compromise in physical accuracy. The proposed fast approach is built and works effectively on a modification of Mass Spring Model which is seen in a variety of cloth simulation study. Instead of using hierarchical data structure which needs to be updated each frame, this fast approach adopts a spatial hashing technique which virtually partitions the space where the cloth object locates into small cubes and stores the information of the particles being held in the cells with an integer array. With the data of the particles and the cells holding information of the particles, self-collision detection can be processed in a very limited cost in each thread launched in GPU regardless of the increase in the amount of particles. This method is capable of visualizing self-collision detection and response in real time with limited cost in accessing memory on the GPU.

The idea of the proposed fast approach is extremely straightforward, however, the amount of memory which is needed to be consumed by this method is its weakness. Also, this method sacrifices physical accuracy in exchange for the performance.

CHAPTER 1 PROBLEM AND PURPOSE

Introduction

3D electronic graphics products in present days are capable of rendering environments and objects with high physical fidelity. However, it is impossible to completely restore the real-world physical law with algorithms because the available computational power is limited. That is why it is meaningful for the community of computer graphics to use all kinds of methods and algorithms to approximate physical motion with an acceptable cost.

Throughout the problems of physical simulation that has been discussed for decades, a typical one of them is cloth simulation.

Imagine a piece of skirt is moving in accordance with the behavior of a human, or a very long ribbon held by a graceful gymnast, or large blanket being blew in windy weather. If the external condition remains constant, in the case of the blanket the strength of wind remains still, we will see points sampled from these moving objects can hardly be predicted and it will be a chaotic curve. That is why cloth simulation, like any other problem of physical simulation, is complex.

Fortunately, (Haumann & Parent, 1988) introduced a model which recognize the structure of a piece of cloth as a huge grid of particles of known mass connecting one another by springs. This understanding is called “Mass Spring Model” and is most widely practiced not only in academic but also industrial areas. One of the greatest advantages of this model is its compatibility to be carried on GPU.

Only the generative method is far from enough, as in physical simulation there is another unavoidable problem: collision. Two circumstances of collision are considered in cloth simulation: cloth-object collision and self-collision (or cloth-cloth collision).

Cloth-object collision refers to the occasion when part of the cloth object interacts with other rigid body. In the case of the Mass Spring Model where cloth objects are expressed by chained particles and rigid bodies are defined by triangles, the collision could be handled starting from the collision between particle and triangles. All that is left is, for each particle on the cloth, do the collision detection between the current particle and all the triangles in the scene. This process is no doubt time consuming since the number of triangles could be huge. We may integrate some

tree-like data structure to reduce the collision query and only check the collision between the current particle and its nearest triangle so that the performance is improved.

Another way to solve the cloth-object collision is represented by (Macklin et al., 2020) who expressed rigid body objects using SDF (signed distance field), an implicit method that returns the result of whether a vertex is inside of a geometry. While this method is fast and convenient, expressing a complicated high poly geometry using SDF is not an easy task.

Self-collision refers to the occasion when part of the cloth object interacts with itself, which is mainly focused by this study. This problem is intriguing especially when it comes to many particles. We may approach this problem using the same way as we deal with the cloth-object one. Yet, it is still not efficient enough. Is there a more straightforward but meanwhile very fast method to process self-collision detection in cloth simulation? The solution proposed by this study is trying to answer this question.

Problem Statement

The problem addressed by this study is that it is very expensive to handle self-collision detection of cloth animation in real time.

Suppose a cloth object generated by Mass Spring Model is paused in the middle of its simulation, and we would like to know if part of this cloth is colliding with itself. A very brutal way to answer the question is, for each triangle on the cloth, traverse all the other triangles and do a triangle-triangle collision detection, which leads to an $O(n^2)$ computational complexity. It is a nightmare if this process is to be done in real time. So, the problem of self-collision is also a problem of optimization.

Many claimed that they solved the issue. For example, (Bridson et al., 2002) introduced a method that output images of cloth simulation with satisfying self-collision detection. It also considered the friction resulted from wrinkles of a cloth. However, it assumed a linear movement of particles between timestep. The method compute collision between triangles and vertices with a cubic equation. In the end, despite the comprehensiveness, it is not a good option to be incorporated in real time. Another drawback is that because of the nature of his design, this method only works on CPU, which is not compatible with the purpose of this study.

Another very popular solution is to integrate a hierarchical data structure, such as octree, KD-tree, and BSP, to stratify the geometric structure of the cloth object and eliminate unnecessary

collision queries, which is a highly effective way to lower the time spent in collision detection. For example, (He & Cheng, 2010) described a quad tree-based data structure to organize the cloth geometry in its local tangent space. Also, studies carried by (Shapri & Bade, 2020) introduced a data structure organized with spheres clusters, which is basically the same idea as the conventional hierarchical ones. The difference is that in this study the particles in a particular tree node is represented by an encapsuling sphere, whereas for the conventional ones the particles are held by bounding boxes.

Despite the efficiency of hierarchical data structure to search for the nearest objects, the problem is that constructing the data structure itself in each frame is expensive, especially when there are many particles on a cloth object. Is it possible to search the nearest two particles on the cloth and detect the collision without using hierarchical data structure?

Purpose Statement

The purpose of this study is to provide an additional feasible option to the problem of self-collision handling in cloth simulation. Very likely the method proposed by this study is not perfect, but it could be a reference which other researchers can take inspiration from and potentially help the computer graphics community to achieve a best solution to this problem.

Research Question

1. How to design a fast approach to handle self-collision in cloth simulation without using hierarchical data structure?
2. How much faster this fast approach is than no optimization is applied?
3. Are there any weaknesses of this method?
4. How much memory is consumed by this method?
5. Is this method compatible to the graphics pipeline?

Significance

The significance of this study is endorsed by the fact that physical effect including cloth-simulations is widely seen in off-line rendered products, such as Computer Graphics animation movies in which single frame takes hours to render, but is hardly found in real-time rendered digital

product like video games. The reason of this phenomenon is because, as explained in the Problem Statement, cloth simulations with high fidelity are extremely expensive to process. So, it is reasonable and economic to integrate physical simulation in a situation where performance is not concerned. And this is one of the reasons why the difference of visual effect between real-time and off-line rendered graphics is huge.

One way to improve the overall quality of real time output is to contain more physical effect, such as cloth simulation. However, the computational resource in real time context is very limited. That is why it is significant to visualize cloth simulation as well as its the collision detection in real time.

Deliverable

An indispensable part of this paper is to implement the proposed fast approach into an executable that includes but not limited to the visualization of a cloth object using mass spring model, interface that controls the input parameters, integration of collision between cloth and static object, integration of cloth-cloth collision, some background geometry, and most importantly, the graph of performance test result.

The deliverable is not only developed to prove the feasibility of this fast approach, but also a source of data collection. The significance of the proposed study lies in the readiness to be implemented in the graphics pipeline.

Definition of Terms

Graphics Processing Unit (GPU) is, described by (Owens et al., 2008), “an electronic circuit which is designed for rapid parallel computing tasks and is therefore specialized at rendering images as output to display devices. Parallelism is the future of computing. Future microprocessor development efforts will continue to concentrate on adding cores rather than increasing single-thread performance”.

CUDA is defined by (Kirk & Hwu, 2020) as a “parallel computing platform and programming model that makes using a GPU for general purpose computing simple and elegant. The developer still programs in the familiar C, C++, Fortran, or an ever-expanding list of supported languages, and incorporates extensions of these languages in the form of a few basic keywords”.

OpenGL is, described by (Vries, 2014), “considered an API (an Application Programming Interface) that provides us with a large set of functions that we can use to manipulate graphics and images. However, OpenGL by itself is not an API, but merely a specification, developed and maintained by the pertaining technological corporation that holds the Intelligence Property of OpenGL”.

Mass Spring Model is a simplified physical model through which the cloth-like deformable objects are constructed. It considers the cloth as a grid of particles connecting one another with forces generated from spring constraints, which is the reason why this model is named. The parameters this model takes are mass of particles, spring constants, rest length between particles, gravity, time step, and a constant describing air resistance.

The Global Memory is physically located on the GPU, it is accessible from both host (CPU) and device (GPU). The capacity of global memory is huge, yet such convenience does not mean that the read and write speed to global memory is also high. It is visible to all kernel function and is accessible as long as the application runs.

Verlet Integration is a modified form of Newton’s equations of motion for numerical integration. It is used in various environment of simulation and computer graphics to calculate position of dynamic particles. In this study, the Verlet algorithm is used as the forward explicit Euler integrator to locate the position of particles between time steps.

Continuous Collision Detection (CCD) is a way to compute the exact moment when two moving objects collide with one another between two consecutive time steps. It is an advanced version of discrete collision detection which can only detect collision at the moment of a frame and does not detect collision in the interval between frames. To check the exact delta time of collision between frames, CCD is necessary.

A vertex-face or edge-edge (VF/EE) pair refers to the 2 particular cases in the context of the continuous collision detection. In practice, all cases of collision between two triangles are first classified into these 2 cases before proceeding to the next stage to find the precise point of which 2 triangles collide with one another.

The Bounding Volume Hierarchical (BVH) data structure is commonly seen in most physical simulation works to optimize the process of collision detection. It is typically used in triangle-based collisions for the purpose of high precision. In BVH, each triangle in the scene is

represented by a bounding volume, usually AABB, and is held in the node of a tree-like data structure that is generated out of a spatial partitioning method.

Assumptions

Specification of Experiment Environment

The performance of the algorithm varies depending on different specifications of hardware, therefore it is important to specify the configuration environment under which the experimental application is developed. In order to ensure the result of the experiment and the theoretical study are logically coherent, the final output and the performance measurement will be conducted on the system with the configurations described by Table 1.

Table 1. Specifications of the computer on which the method will be implemented.

CPU	Intel Core i7-7820HK
CPU Frequency	2.90GHz
RAM	DDR4 @ 2400MHz
GPU	NVIDIA GeForce GTX 1070
CUDA Driver Version	11.1/ 11.0
Global Memory Capacity	8192 MBytes

Physical Model

In this study, the proposed fast approach assumes that the cloth is generated through the Mass Spring model. The algorithm might not be compatible with other cloth generative ideas as the theory is developed upon an assumption that cloth is considered as connecting mass particles.

Delimitation

As the proposed study only focuses on the self-collision handling algorithm in cloth simulation, which includes the cloth-cloth collision detection method and the response resulting from the collision.

Cloth-object Collision

The collision handling method introduced in this study does not consider collisions between cloth geometry and other polygon objects.

Friction

The self-collision handling introduced in this paper does not consider frictions as part of collision response.

Limitations

Downside of CUDA

One of the possible limitations observed from this study is CUDA, which is not part of the graphics pipeline and could potentially hurt the performance. This limitation exists as a tradeoff to the APIs provided by CUDA. To reach the best performance, the method should be integrated in the graphics pipeline.

Limited Time Step

Since the geometric modeling of cloth object uses Verlet algorithm as the time step integrator, in order to keep the cloth object stable in the run time of the experiment, the time step has to be adjusted in a relatively trivial value, which could leave the cloth motion very slow on visual perception. There are ways to make the time step larger, however it is not within the concern of this study.

Evaluation of Visual Effects

Essentially, the fast approach addressed in this study aims to create a plausibly correct visual effect of cloth-cloth collision other than physically precise, the question as to how real the algorithm looks to humans cannot be answered in this study. This study only provides theoretical and practical discussion to the executability of a solution to a specific problem, additional works of human study are needed to evaluate the aesthetic and authenticity of this issue.

While this is a quantitative study trying to offer a solution to a specific problem, the visual effects of the final output, however, is a qualitative one and is out of the scope of this study.

CHAPTER 2 REVIEW OF LITERATURE

The self-collision approach proposed in this study is developed upon the Mass Spring Model, a generative method of deformable objects that is introduced by (Haumann & Parent, 1988). This model was later optimized further by (Provot, 1995). There are other models recognized in the area of cloth simulation, including the Geometry Model announced by (Kunii & Gtoda, 1990), and (Weil 1986) and the Continuum-based Model by (Feynman, 1986) and (Volino et al., 1995).

The Mass Spring Model is most widely used in various research. Ever since (Provot, 1995), one of the crucial problems that the cloth simulation community tried to solve is the time step. Eberhardt et al. (1995) described an explicit integration method to solve the stiff differential equation caused by the internal force, an intrinsic property of the Mass Spring Model, but the time step has to be extremely small to keep the cloth system away from explosion.

Mass Spring Model

A very important concept to be addressed before elaborating self-collision handling is the generative method of cloth objects. The fast approach that this study proposed is built on top of what (Haumann & Parent, 1988) introduced, a fundamental simulating method of cloth modeling.

The mass spring model is represented by a grid of particles connecting each other with springs. There are 3 types of constraints that determines the characteristics of the cloth object.

1. The structural constraints connect neighbor particles vertically and horizontally and is used when compute the magnitude of extension and compression, it constructs the physical frame of the cloth object.
2. The shear constraints connect particles diagonally and is used when compute the shearing force, it is the constraint that avoids the cloth object from collapse into a line.
3. The Bend constraints connect every other particle vertically and horizontally and is used when compute the bending force, it is the constraint that keep the cloth object from folding like a piece of paper.

The structure of different types of constraints are shown in figure 1.

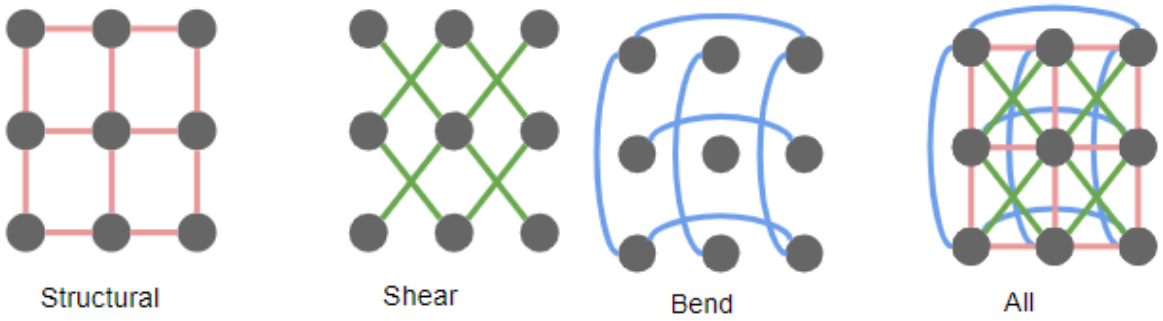

Figure 1. Overview of all types of constraints.

What the spring constraints are used for is to calculate inner force where Hooke's Law is applied. There are other forces to be considered, including gravity, wind, and damping. Damping is a very import force that prevent the cloth waving back and forth perpetually, it is a force applied constantly in the direction inverse to where the velocity is. As soon as the model is set up, the cloth can be simulated incrementally according to Newton's Second Law of Motion. Overall, the mass spring model could be expressed with the following equation.

$$F_{Net}(v) = M * g + F_{wind} - F_{damp} + \sum k(l_{current} - l_{rest}) = M * a$$

Where: "M" is mass of particle; "g" is a vector representing gravity and is pointing to the direction (0, -1, 0); "F_{wind}" is a vector for wind force; "F_{damp}" is a term that is always inverse to the direction of velocity, this term is the key that prevents the cloth system from moving back and forth perpetually; "k" is the spring constant, it holds the property of stiffness of the cloth system; "$l_{current}$" is the current length between the current particle and one of its neighbor particle; "l_{rest}" is the initial length between current particle and one of its neighbor particle when the cloth is in a balanced state. The term with the "∑" is the term of the inner force, which means the greater the distance of which the particle is away from it's initial position, the greater the force trying to drag it back to where it was. Also, the greater the "k" is, the less the cloth tends to yield to outer force.

Figure 2 shows comparison of typical mass spring model with different “k”.

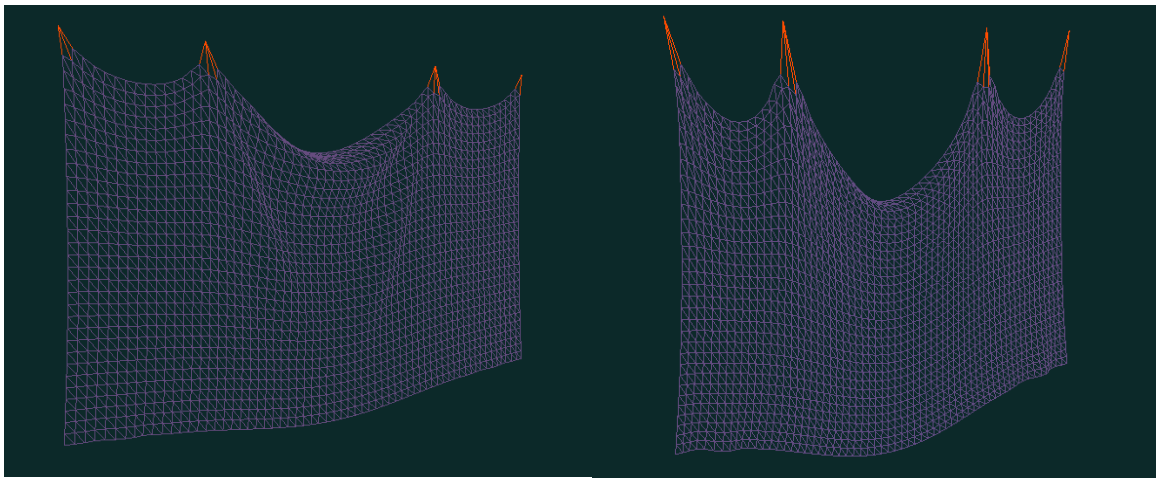

Figure 2. On the left is a cloth with high spring constant, on the right is the one with low spring constant.

Large Timestep

Baraff & Witkin (1998) brought an implicit integration method for the purpose of a larger time step, despite its solidity, it is not a solution to be performed real time. The method requires a convergence of a matrix inversion algorithm, and because such large time steps are taken and the correlation between error and the time step is linear, this approach is not a good option for the proposed fast approach in this study.

Another unavoidable problem in cloth animation is to deal with the self-collision. Provot (1997) introduced a method to quantify the deformation of a cloth system with a cone apex angle which generated from the normal of triangle patches. While this is a method featured with physical accuracy, the triangle-triangle collision algorithm provided in this paper takes significantly additional resources if to be implemented on GPU. This method requires dynamic thread controlling every frame since the number of collisions to be detected is constantly changing in run time.

Integration of Data Structure

Many studies were seen using spatial data structures to construct hierarchy nodes to represent the cloth geometry so that unnecessary collision queries are pruned. Bridson et al. (2002) used Axe Aligned Bounding Box (AABB) based tree for collision searching. But the downside is that each

frame the tree data structure must be rebuilt and is expensive. Others like (Lv et al., 2007) proposed to use a bounding sphere-based tree data structure to prune unnecessary collision query, however this method assumes the length of the constraints could be clamped within a certain extent, which is an operation that cannot be implemented in the context of multithreading. He & Cheng (2010) introduced a quadtree data structure separating the local uv space of the cloth, which could roughly filter out a possible collision. But the method's approach to the collision query works similar to what (Provot, 1997) proposed.

Past Related Works

The problem of self-collision detection and response has always been a major obstacle in the section of cloth simulation. Many have introduced valuable solutions.

GPU-Based Incremental Collision Handling

The GPU-based incremental collision handling method, proposed by (Tang et al., 2018), has shown robust and fast output. This method mainly involves two novel techniques.

One is to use spatial hashing to do incremental CCD. The idea of this method is based on a fact that only a small number of vertices will be influenced by response forces. The solution keeps track of the deformed vertices using spatial hashing, and then detects collision with high-level and low-level GPU culling algorithms. The second one is the GPU-based non-linear impact zone solver. It is used to compute a penetration status according to impact zone and is parallelized on GPU.

The output of the implementation from this technique proved reliable physical accuracy. Meanwhile, the non-linear impact zone solver allows the cloth object to run in a large time step, which leads to verisimilar cloth motion.

The work of (Tang et al., 2018) consists of 4 parts: integration, collision detection, and collision response computation. The simulation process is initialized in a state with no penetration on the cloth object. For each time step, the simulation process goes through the following 4 stages shown in figure below.

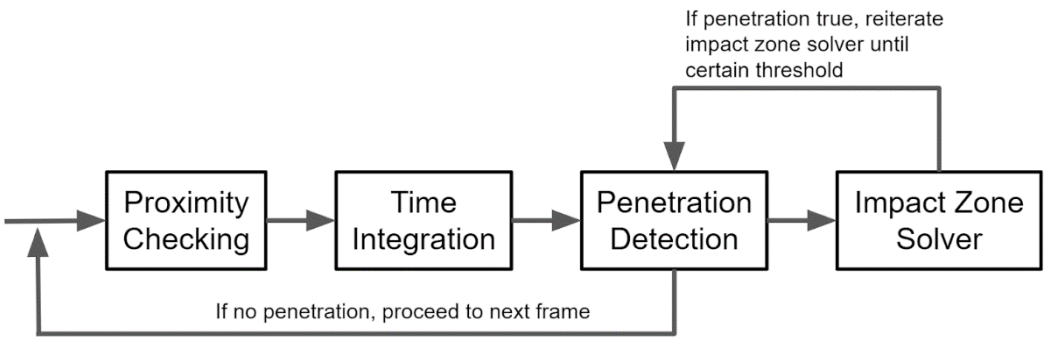

Figure 3. Process of incremental collision handling.

The proximity checking takes care of finding potential VF/EE constraints. Time Integration takes care of implicit time integrator with internal/external forces and proximity constraints. Penetration Detection is a part that keeps track of penetration using CCD. Impact Zone Solver: a reiterative process on GPU to resolve all the potential collision until a new penetration-free state is reached.

To do CCD between each overlapping VF/EE pair, (Tang et al., 2018) adopted the high-level culling technique with BVH structure to roughly get the potential overlapping VF/EE pairs. Next, another low-level culling method takes in the rough collision information and processes all the VF/EE pairs with reliable collision tests. The output of such a process is valuable for references. However, the cost is high as it is run reiteratively for multiple loops until the rough BVH returns false.

To improve the performance, Tang et al., introduced the incremental CCD algorithm that uses the information between two consecutive iterations to reduce the number of loops.

The collision response applied in this study is like the one from (Bridson et al., 2001). Once a VF/EE pair collision is detected, a repulsive force will be generated based on the proximity distance between the pair and added to the stiffness matrix. The proximity in this method is integrated into the time integration. One of the advantages of this technique is that it can also get static/kinetic friction force out of the relative tangential velocity between a VF/EE pair.

At the end of each iterative collision detection stage, the process has to know whether the cloth object is in penetration-free state. To do this, a non-linear impact zone solver is presented for this job.

Overall, the theoretical time complexity of this process can be deducted as follows: For each time step, the simulation process would need an extensive matrix to do the implicit integration which is O(1) since the GPU is optimized in doing linear algebra. The high-level culling would need O(log n), where n is the number of triangles of the cloth mesh. Tang et al., (2018) didn't disclose too much detail of how they handle the incremental CCD, but if all VF/EE pairs are distributed with their own thread, the time can be limited to O(1). The final impact zone solver would take O(1) since it is run parallelized for every vertex in the impact zone.

According to (Tang, et al., 2018), the final performance of the implementation is averagely run from 0.5 to 1.0 second per frame, which is approximately 2-3 FPS.

The theoretical time complexity of the algorithm is limited. However, the process runs the penetration detection and impact zone solver reiteratively before proceeding to the next frame even with the help of incremental CCD, which is very time consuming. Also, in the part of the incremental CCD and the impact zone solver, GPU resources will be diversified greatly in different circumstances.

Virtual Marble Technique

The idea of how (Wojciechowski & Galaj, 2016) solve the problem of self-collision in cloth simulation is that they treat each vertex on the cloth mesh as a virtual marble, see figure below. Instead of doing collision detection between vertices and cloth mesh, they reduce the problem to the collision detection between marbles. If the distance between two vertices is less than twice the length of the marble radius, a collision is detected, and the corresponding vertices are backed up to the point where there is no collision.

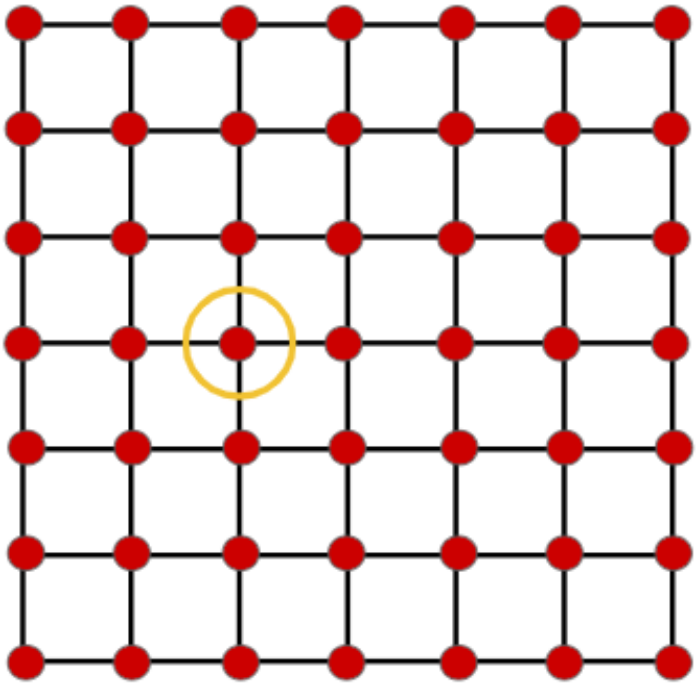

Figure 4. Each vertex is set up with a virtual marble. Collisions between marbles are simple and fast, the only downside is that the cloth object will never “collide” with itself.

The time complexity that this solution has is O(n) with parallel threading on GPU, since each marble on the cloth object still must traverse through all the others to detect a collision. This explains why the performance drops linearly with increases on the number of vertices. Their implementation can hold an acceptable frame rate when the number of vertices is less than 3600.

Bounding Volume Hierarchy

The Bounding Volume Hierarchy (BVH) adopted by (Liu et al., 1998) is featured with a binary tree data structure that holds the information of all the bounding volumes which contain the triangles of the cloth mesh. They used Axis Aligned Bounding Box (AABB), shown in figure below, as the bounding volume of triangles, and the AABBs are involved in the collision detection and response. Let the vertices of triangle Ti at time t denotated Vij(t), where j = 1, 2, 3. The AABB shall be described by two points Min(t) and Max(t), which can be easily obtained from the equations below.

$$Min(t).x = \min\left(v_{i1}(t).x, v_{i2}(t).x, v_{i3}(t).x\right)$$

$$Min(t).y = \min\left(v_{i1}(t).y, v_{i2}(t).y, v_{i3}(t).y\right)$$
$$Min(t).z = \min\left(v_{i1}(t).z, v_{i2}(t).z, v_{i3}(t).z\right)$$
$$Max(t).x = \max\left(v_{i1}(t).x, v_{i2}(t).x, v_{i3}(t).x\right)$$
$$Max(t).y = \max\left(v_{i1}(t).y, v_{i2}(t).y, v_{i3}(t).y\right)$$
$$Max(t).z = \max\left(v_{i1}(t).z, v_{i2}(t).z, v_{i3}(t).z\right)$$

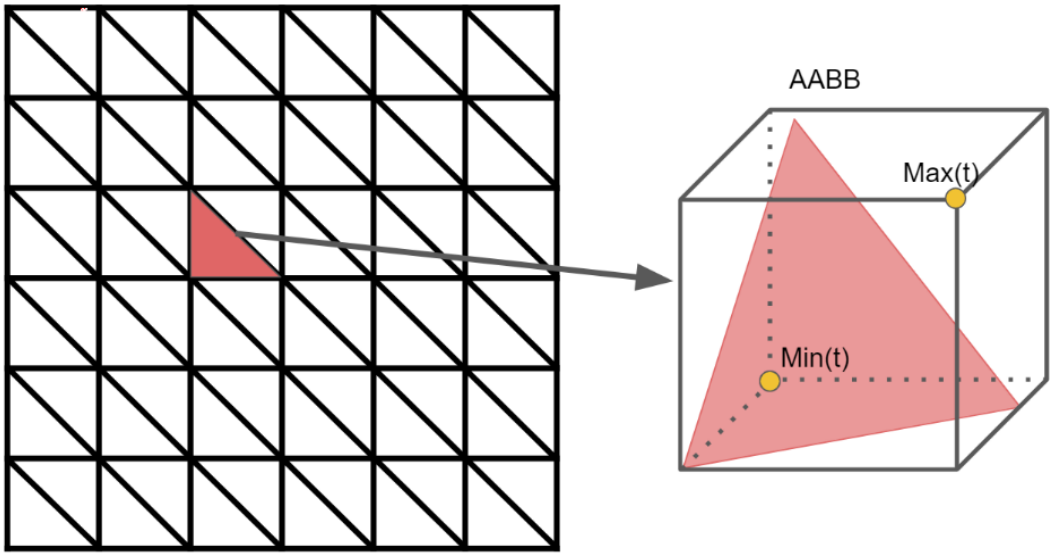

Figure 5. The cloth object is represented by the triangle array. Each triangle is wrapped by a bounding box.

The BVH method transfers the problem of collision detection between triangles to the problem of collision detection between bounding boxes. Two bounding boxes will be considered as overlapping if and only if their corresponding intervals overlap on the projections to all the 3 axes. Check collisions between bounding boxes are fast, however, it is still time consuming even run in multi-threading as each bounding box would have to go through all the others to know the collision status.

To improve the run time cost, (Liu et al., 1998) used a binary tree to contain all the bounding boxes, this method recursively partitions the cloth mesh into half until there is only 1 triangle in the leaf node. This process can be illustrated by figure below.

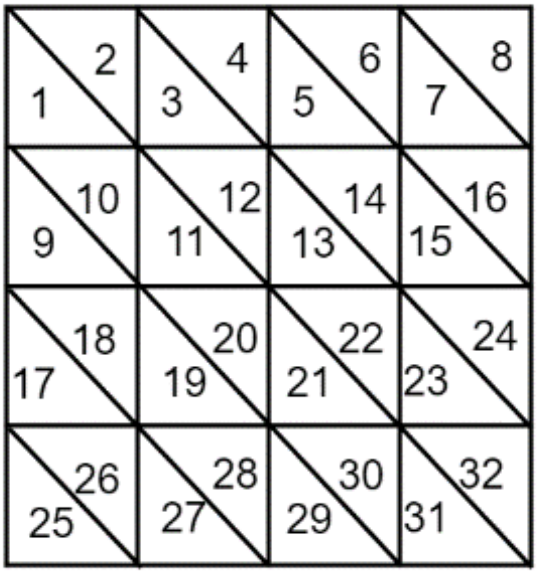

Figure 6. Each triangle on the cloth object is indexed.

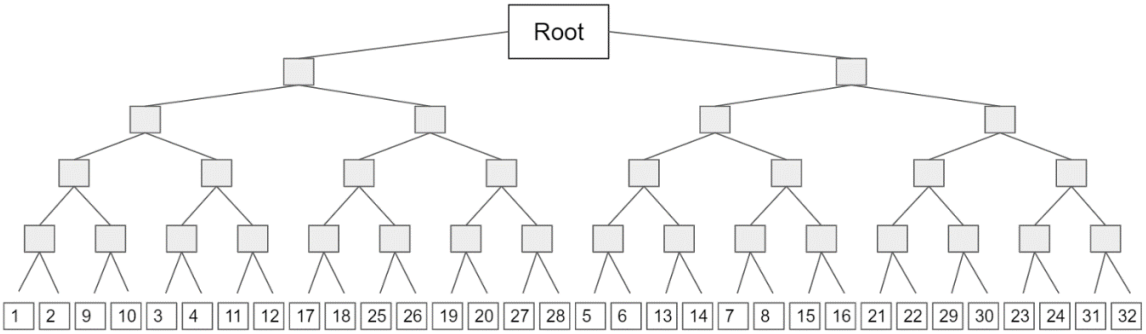

Figure 7. An example of a cloth mesh where each triangle has a unique index.

The binary tree above shows the result of the BVH method in the previous figure. The root node holds the bounding box of the whole cloth object, the level 2 nodes hold the bounding boxes for half of the cloth. Each lower-level child node stores the bounding box of half of the object represented by its parent.

With the hierarchical data structure, there's no need to traverse all the bounding boxes to know the collision status. To do a collision query for a bounding box A, simply search through the tree. The process is shown in the pseudo code.

```
Test_Collision(Bounding_Box A, Root_Node){
        If Root_Node is null, return true;
        End if;
        If A overlaps with Root_Node->Bounding_Box
                Test A overlaps RootNode->children->Bounding_Box;
        Else return false;
        End if;
}
```

Figure 8. Pseudo code shows the recursive function which traverse the BVH and perform collision detection.

This method has proved to have great improvement on the performance, in the final output the average total time in one simulation step could take down to 2.02 seconds per frame with 51*51 vertices on the cloth.

Since the information of the bounding box has to be updated every frame, the BVH tree would have to be constantly re-constructed, which takes O(log n) where n is the number of triangles. After the GPU has all the information of the bounding box, it would launch a thread for every triangle and need another O(log n) to search through the BVH to do collision detection. In terms of the space consumption, this process takes O(n) to store all the data of BVH.

Ray-traced Collision Detection

The way with which (Lehericey et al., 2015) approach to the problem of self-collision handling is to adopt the ray-tracing function which is provided by GPU as part of its hardware optimization.

The idea is to cast two rays out of each vertex of the cloth mesh with one of them going the direction of normal and the other going the opposite. Say we would like to test whether the potential collision pair (C1, C2) is collision true, we cast a ray in the direction of normal to see if it hits C2. If it hits C2 and the distance between C1 and the hit point is less than twice the cloth thickness, the test (C1 -> C2) returns true. However, only one test is not enough. The test of pair

(C1, C2) is decomposed into two tests: (C1 -> C2) and (C2 -> C1). To obtain the collision status between C1 and C2, both counterparts cast rays respectively from C1 to C2 and from C2 to C1. The reason is that performing only one of the tests could return false results, it is important to combine both test results to reach the final collision status. The process is illustrated in figure below.

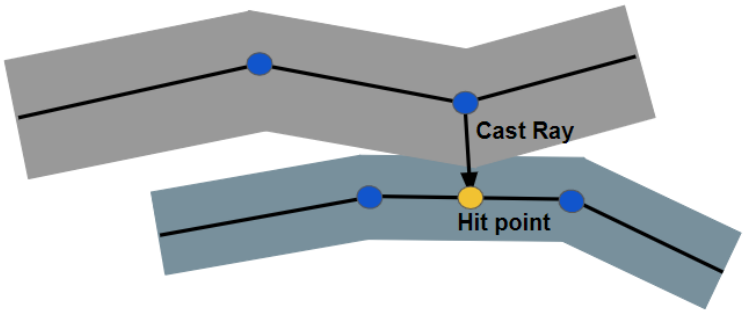

Figure 9. The ray-traced algorithm could also be extended to testing collisions on object-object and object-cloth.

Just like (Liu et al., 1998), Lehericey et al. (2015) adopted BVH as the accelerative structure to lower the cost.

The performance of the implementation of the solution has proved capable to be run in real time. With a scene of 10,000 vertices of static objects and up to 17,000 vertices of multiple cloth objects, the collision-detection takes an average of 4.3 millisecond. However, the actual FPS of the implementation was not disclosed.

This solution would typically require O(log n) for each thread allocated to perform ray-traced testing because of BVH, and an O(n) of space to store all the BVH data.

Bounding Sphere Hierarchy

Lv et al. (2007) approached the self-collision detection for cloth simulation with a new idea of BVH, they introduced bounding spheres instead of AABB. In addition, the way with which bounding spheres are used in the hierarchical data structure is different from the one with AABB. Shapri & Bade (2010) later used a very similar technique.

To make the cloth simulation system compatible with BSH, they integrate the cloth generation with a constrained particle-based model, which sets an upper limit to the length of the stretch and shear constraints between particles. With the length between particles in the mass spring model is clamped, the distance between each two neighboring particles must not be longer than L_{max}. Such restriction very much conforms to the physical rules since in reality a cloth material is only ductile within a certain range. However, the Mass Spring Model theoretically assumes that the cloth object can be infinitely extended if the force is big enough.

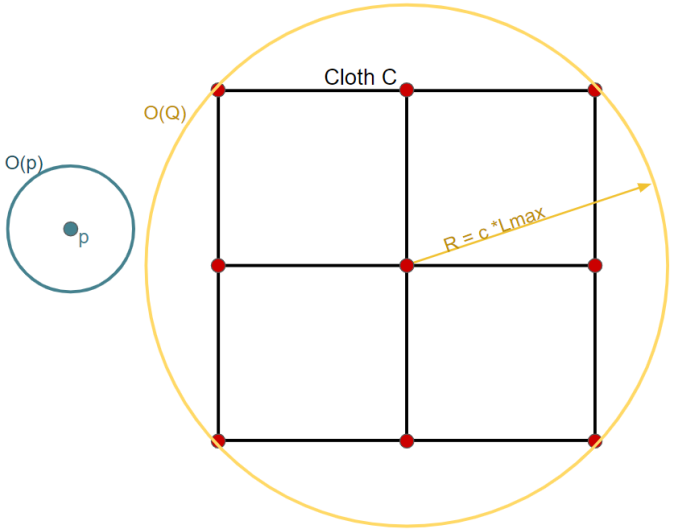

Figure 10. The bounding sphere O(p) is not colliding with bounding sphere O(Q), which means collision tests between particle P and all the triangles wrapped in bounding sphere O(Q) are pruned.

The constrained particle distance is the prerequisite of (Lv et al., 2007)'s BSH. The strict constraints on the stretch and shear connection means that it is possible to put a sphere with a constant radius to hold any two given particles on the cloth mesh. Naturally, we may contain all the particles on the cloth object with a single sphere O(Q), where the centroid is a chosen vertex and the radius is $c*L_{max}$, see figure 10. This feature can be exploited to prune unwanted collision tests. Meanwhile, for each particle P on the cloth object, it is attached with a smallest sphere O(P) that represents the moving track of P, where the centroid is the position of P(old) from last time step and the radius is the distance between P(old) and P(new).

The method of (Lv et al., 2007) is, as shown in figure 10, that they test collision between O(P) and O(Q) first. If false, then the polygon represented by O(Q) is pruned. If true, test precise collision between P and the triangles held in O(Q). The image below shows the whole process of BSH.

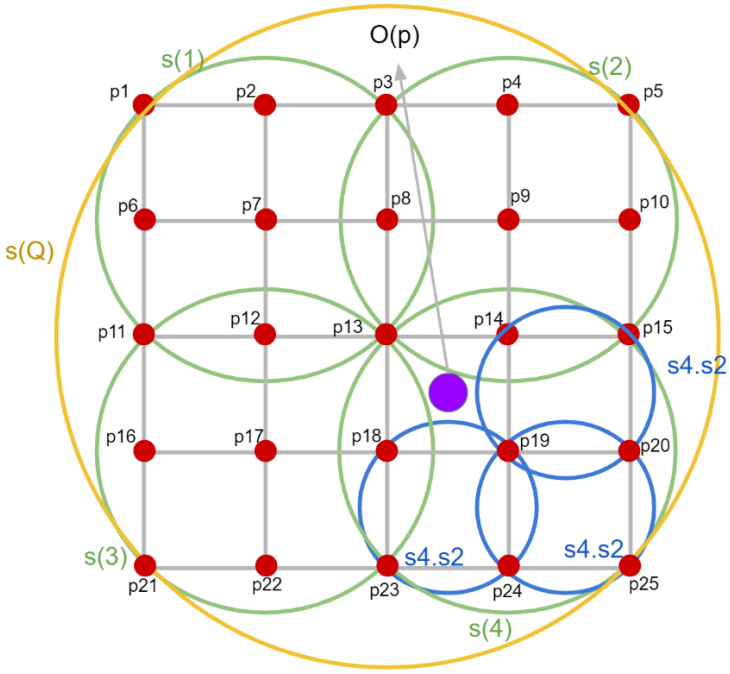

Figure 11. Image explains how the purple sphere O(P) is tested collision with the cloth C.

In the image above, first collision (O(P), s(Q)) is true. Then test collisions between O(P) and children of s(Q), and s(1), s(2) and s(3) are pruned. Next, test O(p) and children of s(4). Finally, perform particle-triangle test between P and the triangles in the s(4).s(1).

As shown in below, To maintain the BSH, each frame we would need O(log4 n) to construct the tree. And another O (log 4 n) is required to search through the tree to test every vertex. In terms of memory, BSH costs O(n) where n is the number of triangles.

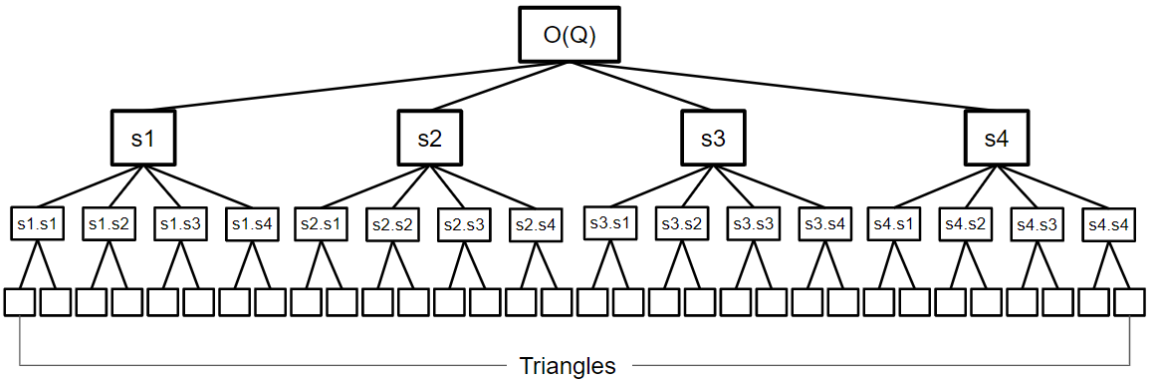

Figure 12. Visualization of BSH according to the given cloth mesh in figure 10.

Spatial Hashing Combined with Bounding Sphere

Another technique that is seen to handle the self-collision problem is the one introduced by (Pabst et al., 2010). In order to gain great performance, their primary task is to design an algorithm that is highly scalable to GPU. So that the implementation is able to achieve real time with multiple GPU connected.

Similar to the design of (Liu et al., 1998), Pabst et al., (2010) focused on collision detection between triangles. For each candidate triangle collision pair, they need to distinguish the precise type of orientation out of 6 VF and 9 EE pairs.

Before handling triangle pairs, they adopt the Spatial Hashing method to prune unwanted triangle pairs. The way Spatial Hashing works is to separate the space into cells, either uniform or hierarchical, so that target objects fit into single cells. To locate the cell holding the target, each one is assigned a unique hash value given its position. Since every cell can now be easily accessed, the collision detection is reduced to test overlapping triangles with the same cell hash value. To determine which cell does a specific triangle belong to, check the position of the triangle's centroid. The triangle is held by the cell that has the centroid even if the triangle extends to multiple cells.

Pabst et al., (2010) set up a rule that the cell holding the triangle's centroid is marked as the triangle's home cell, and the surrounding 28 cells are marked as the phantom cell if the bounding sphere of triangle overlaps with them. For the rest of the occasion the cells are marked as invalid. See figure below.

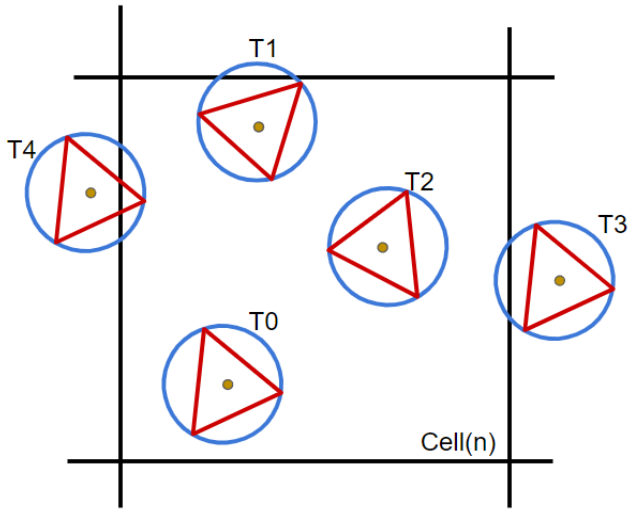

Figure 13 Cell n is marked as home to triangle T0, T1 and T2, and as phantom cell to T3 and T4.

The design of this method is to invoke kernel function for each potential collision pair. To identify each collision to its corresponding thread, they impose an order of sequence in which triangles pairs are tested. Take figure 13 as an example, the order to test collision between triangle pairs is seen in table 2.

Table 2. A lookup table to ensure all the triangle pairs are tested in a constant sequential order.

Potential Collision	Triangle ID	Triangle ID
0	T0	T1
1	T0	T2
2	T1	T2
3	T0	T3
4	T0	T4
5	T1	T3
6	T1	T4
7	T2	T3
8	T2	T4

The solution presented by (Pabst et al., 2010) proved to have high performance. Since they allocate 1 thread for every collision, the theoretical time complexity of this method is O(1). Also, searching through the lookup table to identify each collision pair takes O(log n), where n is the number of collisions. As for the spatial hashing and the hierarchy, it needs O(n) to hold all the data, where n is the number of cells.

While this method is fast and highly scalable, it could cause 1 problem, which is that the simulation process would lead to large deviations of thread number. From time to time there will be worst cases where resources reserved from GPU could differ from 0 to hundred thousands. In real time graphics, a predictable and steady cost is usually prioritized.

With all the previous works scrutinized, this study describes a straightforward fast approach where no hierarchical data structure is to be applied. It is designed to be implemented on GPU and the computational complexity resulting from this method on each thread and each frame is very limited. But nothing comes without a price, the disadvantage of this method will be discussed in the limitation.

CHAPTER 3 METHODOLOGY

A Modification to Mass Spring Model

To maximize the validity of the self-collision method to be described later, this study proposed to modify the typical quad layout in the conventional mass spring model into an equilateral triangulated structure. The idea of the spring constraints remains intact. Figure below shows the layout of modified structure.

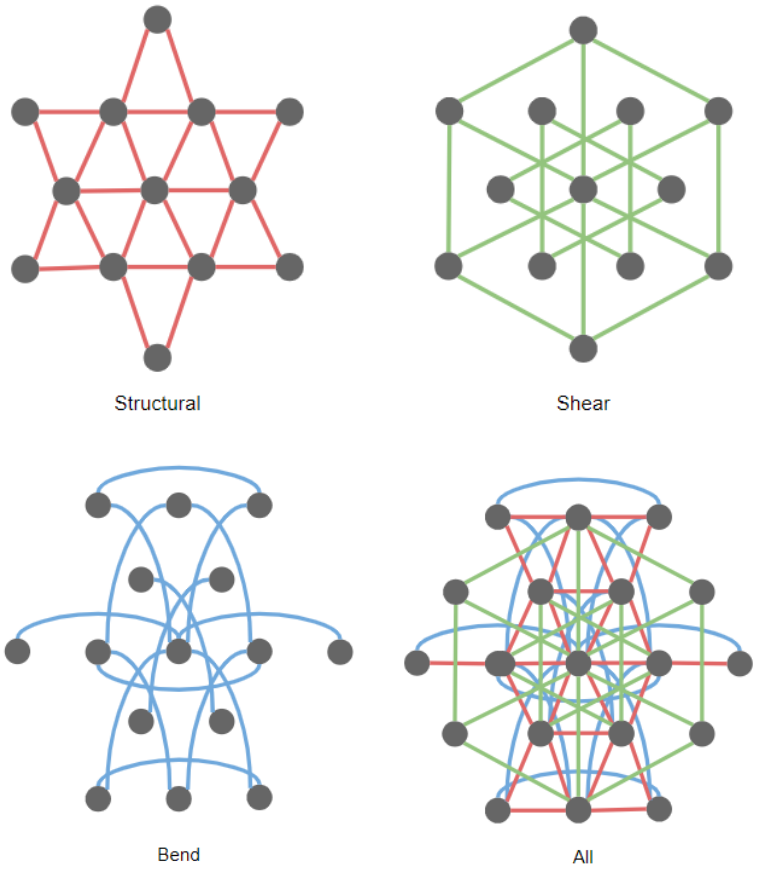

Figure 14. Overview of modified sampling layout.

Aligning the particle using this layout immediately solves an issue that is often seen in cloth simulation, which is the inconsistence between different direction of folding in cloth. The layout of typical Mass Spring Model usually leads to conspicuous artifacts if the cloth folds in a diagonal

direction when the resolution of the cloth is relatively low. This is a problem not out of the Mass Particle Model itself but of the way with which the vertices are connected as triangles.

Figure below shows the appearance of artifacts. The area circled on the right shows the artifact caused by the cloth folding in the direction of yellow arrow shown on the left. But if cloth folds in the direction of blue arrow, artifact is hardly to be seen. This inconsistence comes from the asymmetric quad layout, shown on the left in the figure below. This is because the cloth object is folding in the diagonal direction of the yellow arrow in the left image. If the cloth folds otherwise with the blue arrow, artefacts will not be seen.

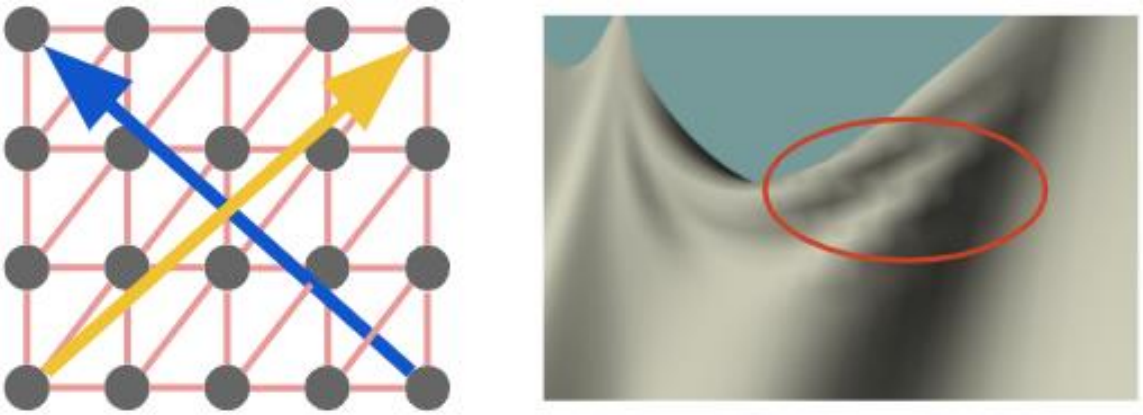

Figure 15. The circled place shows the artefact caused by the inconsistent normal.

This issue can be solved by infinitely increase the resolution of the grid, but this is very uneconomic. However, with the modification of an equilateral triangle layout, shown in figure below, connections would not be packed into an unsmoothed geometry no matter to which direction the grid folds. Normal will be more consistent regardless of the folding direction. On the right is the actual output of this layout.

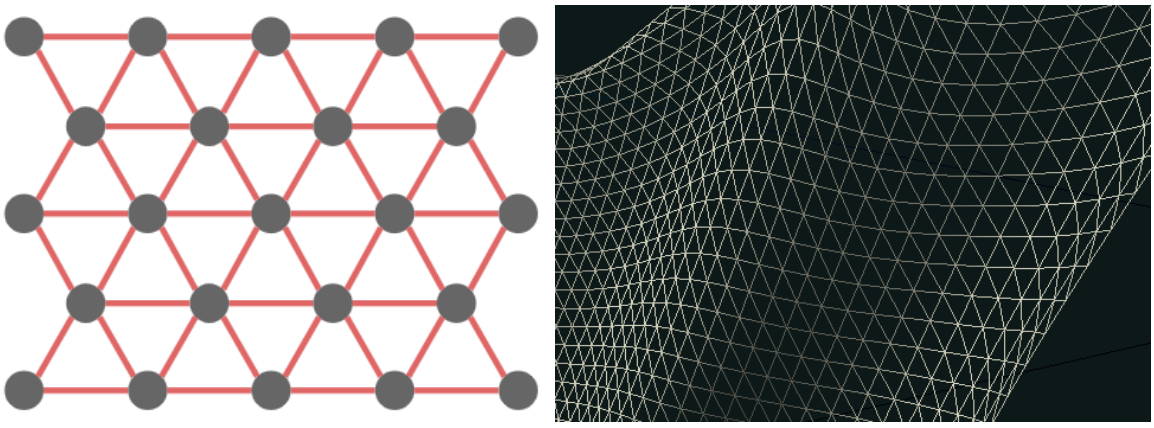

Figure 16. On the left is an example of equilateral triangle layout.

However, the downside of this layout is that, for each particle, the number of constraints that are needed to be sampled is 18, whereas in the conventional layout it is only 12. As shown in Figure below. Still, compare to the benefit the system obtained from it, this change is considered a good tradeoff.

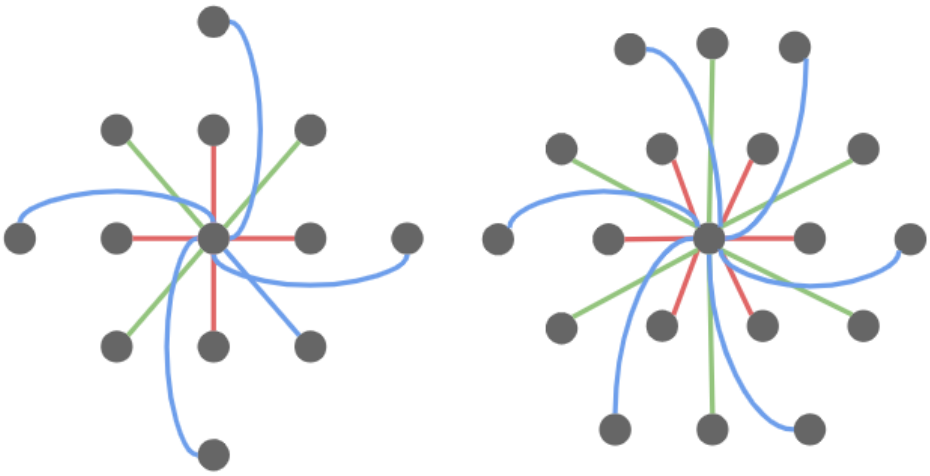

Figure 17. Constraints to be sampled for each particle. On the left is the quad layout, where only 12 neighbor particles are sampled. On the right is the equilateral triangle layout, where 18 particles are sampled.

The Fast Approach of Cloth-cloth Collision

Virtual Sphere

To compute the self-collision with a low cost, this study proposed a similar technique of (Wojciechowski & Galaj, 2016) such that the mass particles are treated as spheres. The radius of spheres equals to half of the Rest Length of constraints, an input parameter indicating the length between particles when the cloth is in the balanced state. By doing this, the cloth object is covered by the spheres as illustrated in the figure below. We can see that the question as to how to handle the cloth-cloth collision is transferred to how to handle the sphere-sphere collision.

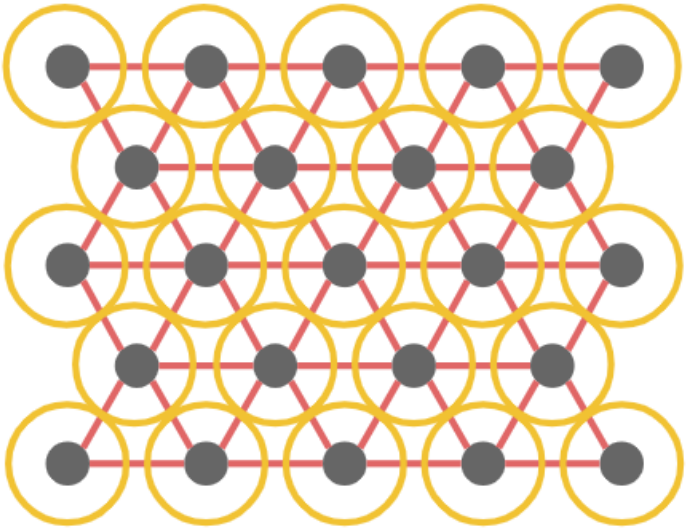

Figure 18. This triangulated layout makes spheres packed more tightly and can cover more area of the grid with relatively less radius.

Spatial Hashing

To check the collision between two spheres is cheap. But it is unwise to traverse all the others since that will result an O(n) computational complexity in each frame for each thread, which is a level that cannot be accepted even for GPU.

As for the optimization of sphere-sphere collision detection, this study proposes to partition the space of where the cloth object exists into cells. This idea is inspired by the marble technique adopted by (Pabst et al., 2010). The difference is that the size of the cell in this study is set to an extend that each time and each cell can only hold 1 center of a sphere, shown in figure below. The

illustration shows a situation in the context of 2D. In this case, the size of the cells must ensure that each cell would hold only 1 sphere in every moment. Also, at the same time we would like the size to be as big as possible to save the global memory. As a result, the optimized length of the cells would be $\sqrt{2}$ times sphere radius in the 2D case. In 3D the length of the unit cell should be:

$$L = \frac{2\sqrt{3}}{3} * Radius$$

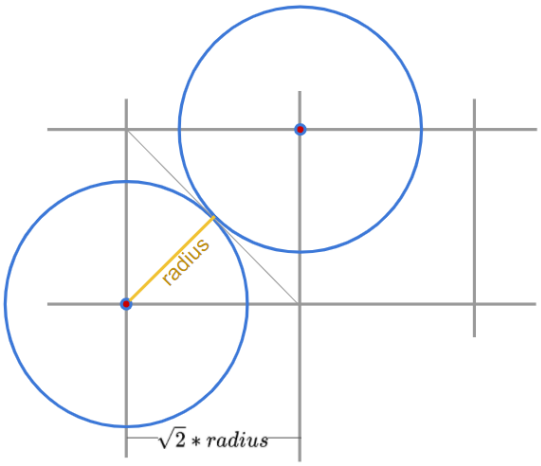

Figure 19. A 2D example describing how to determine the size the of the cell's dimension.

The position of the cells is hashed into a unique ID so that this number could be used as an index to access a data array that stores the information of sphere ID which some cells hold. Since the partitioned cells are the axis-aligned cubes, they can be defined by two variables. The first is the length just mentioned above, the second is the vertex position with the least value of x, y and z out of the 8 vertices of the cube.

Given the position P and the dimension of the space partitioned by this method, we may use a simple hash function, shown below, to assign each cell with a unique ID.

$$cell\_ID = Cell.y * Dim.x * Dim.z + Cell.z * Dim.x + Cell.x$$

Where cell is represented by a vector with 3 elements (vec3) indicating its position in the space; Dim is also a vec3 with the information of how many cells are there respectively in the x, y and z direction.

Only spatial hashing is not enough, to do self-collision detection fast we also need to have all the spheres hashed. Fortunately, these spheres can be represented by the threads launched in GPU, we may assign an ID to each sphere by accessing the parameters when kernel is invoked.

For each particle, rather than travers all the other others, this method traverse through the neighbor cells of the one where the current particle locates. Figure below illustrated the 2D version of the partitioned space holding 2 vertices of the cloth mesh. s1 is represented by the current thread and is in Cell 17. To find potential collisions, it traverses the 24 (5*5 - 1) neighbor cells (tinged blue) to see if any of them holds another sphere. The reason why it must be at least the neighbor 24 cells rather than 8 (3*3 - 1) is because of the occasion shown in figure 19. Also, traversing the 7*7 neighbor cell is unnecessary because the spheres in the outer range are too far to collide with the sphere in the middle.

If the neighbor cell has the value “0”, this implies there is no sphere currently in this cell. If the value in that cell is not “0”, it could be used as the index to access the position of the sphere from the corresponding thread. However, in CUDA, information shared between threads are only within blocks. To let any two threads accessing data from one another, we save the information into the global memory on GPU as an array.

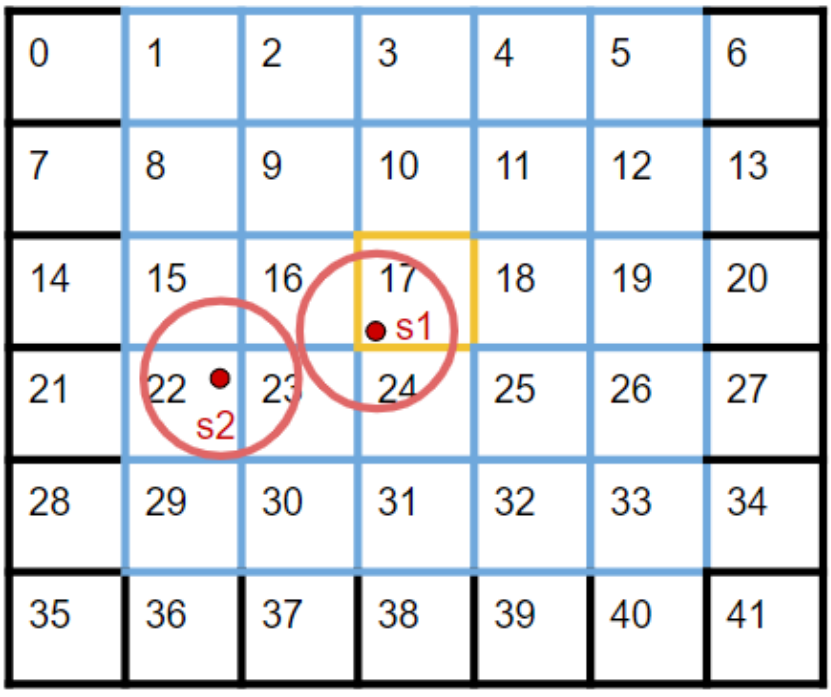

Figure 20. Traverse the neighbor 5*5 cells to find collision.

This way, the computational complexity for each thread is limited to a fixed level regardless of the quantity of particles. This is the very reason why theoretically it holds the potential to be a fast approach. In the implementation of this study which is 3D-based, each thread would go over the 124 (5*5*5 - 1) neighbor cells and check if it holds any sphere.

However, the price to exchange for the speed is that this method is memory consuming because it relies on a very long array that holds the particle hash ID shown in figure below.

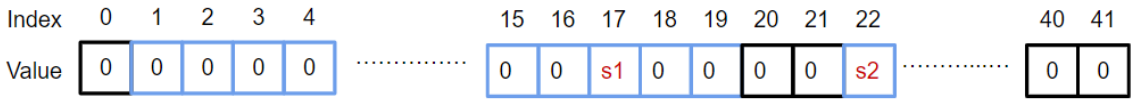

Figure 21. A picture describing the array holding the data of previous figure.

The size of this array depends on the number of cells. Fortunately, there's space of optimization in that most value being held in the array is "0", indicating there is no particle in that cell. The optimization of the memory management in sparce data array will be discussed in the future works.

Workflow

The figure below explains the workflow of the self-collision algorithm proposed by this study. For each thread, given the current position of the vertex, the algorithm predicts the position in the next frame using Mass Spring Model. But this new position will not be updated into the buffer object before collision test.

There are two more things to do. First, each thread saves the information of the predicted position into the global memory preparing to be accessed by other threads; second, each thread saves the sphere hash ID to the spatial hashing array in the global memory such that other threads can fetch the current thread ID in a particular cell.

With all the information needed, each thread, representing the corresponding vertex, check the neighbor 124 cells one by one to see if there is a sphere. If there is a sphere, it accesses global memory for its position and make collision detection.

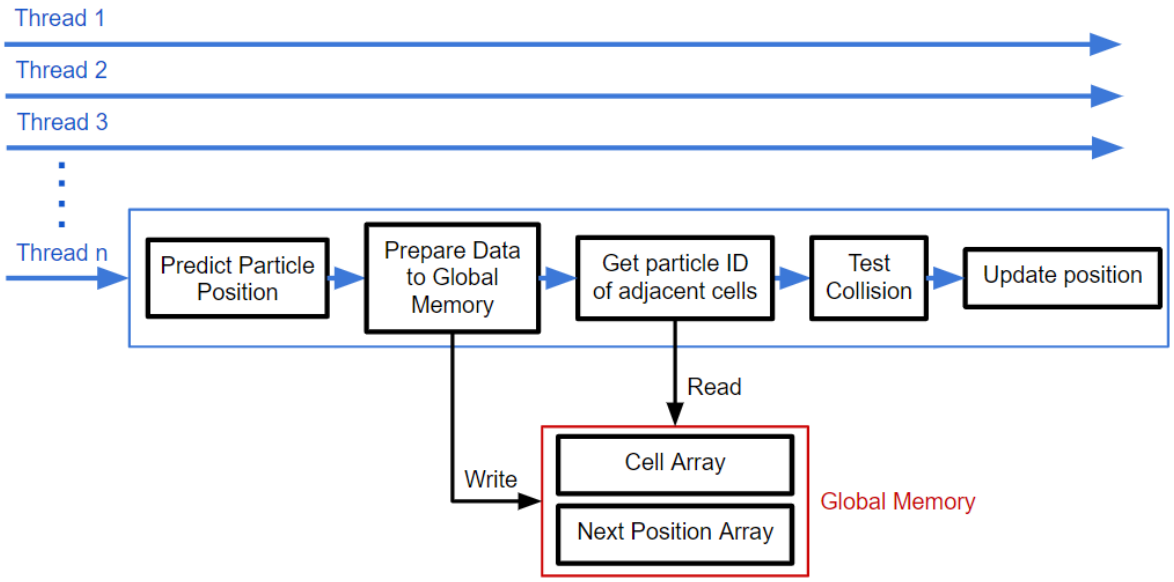

Figure 22. The blue arrow indicates the workflow represented by each thread that controls the vertex; the cell array is stored in the Global Memory which is accessible to all the threads.

Implementation and Configuration

Since the self-collision fast approach is implemented on GPU using CUDA and have the OpenGL rendered into output, there are several strategies that this study would adopt. Noted that the final performance of the output could be tremendously influenced by the way with which the fast approach is put into practice.

Ping-pong Buffer

One of the problems that needs to be constantly considered when programming on the GPU side is the read-write conflict. While putting barrier between read and write operations can effectively avoid this issue, this solution causes all threads waiting thus possibly undermines the performance. Fortunately, there is a way to forestall the problem of read-write conflict without invocating barrier operation, which is to use Ping-pong buffer.

All this technique takes is 2 pointers to VBOs. For each frame, the program alternatively passes these 2 pointers to the kernel function as read buffer and write buffer.

```
__global__
Launch_Kernel(float* readBuffer, float* writeBuffer);

void passVBOPointer() {
|       //initialize 2 pointers locally
|       float* VBOPointer1, VBOPointer2;
|       //alternatively feed kernel function with 2 pointers each frame
|       Kernel(!pp ? VBOPointer1: VBOPointer2, !pp ? VBOPointer2: VBOPointer1);
|       //switch boolean state so that next frame the pointers get switched as well
|       pp = !pp;
}
```

Figure 23. Pseudocode of Ping-pong buffer.

The kernel function does not know the pointers are exchanging, all it knows is that it always reads from the "read buffer" and writes to the "write buffer". Hence, we avoid the read-write collision by read and write to different location.

Continuous Collision Detection

Another pitfall that this study might encounter is the algorithm of collision response. Intuitively this can be done by pushing two spheres away from each other to an extend that collision is no more. However, putting aside the mathematical incorrectness, the most serious issue that could be resulted from this idea is the constant oscillating of particles. It is mostly seen when the cloth object is pushed to a complicated situation, shown in figure below.

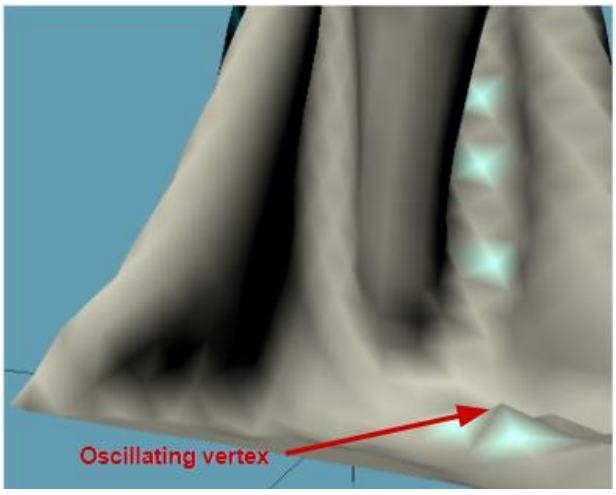

Figure 24. The unnatural spike pointed by red arrow indicates vertex is flipping back and forth.

This instability could be alleviated by tuning down time step, yet the problem persists. The source of this problem is the time when the sphere is pushed away, it collides to the third sphere which is an event that is beyond the reach of the program. To solve this, we need to use a continuous collision detection technique to know where the physically correct contact position from the beginning so that no secondary collision could take place.

The problem and solution are shown in figure below. "S1" is a moving sphere at a particular moment. The dotted circle is the predicted position of where "S1" will be in the next moment and collision is detected between "S1" and "S2". On the left is discrete collision solution that leads to problem of second time collision between "S1" and "S3". On the right is the continuous collision detection that solves secondary collision.

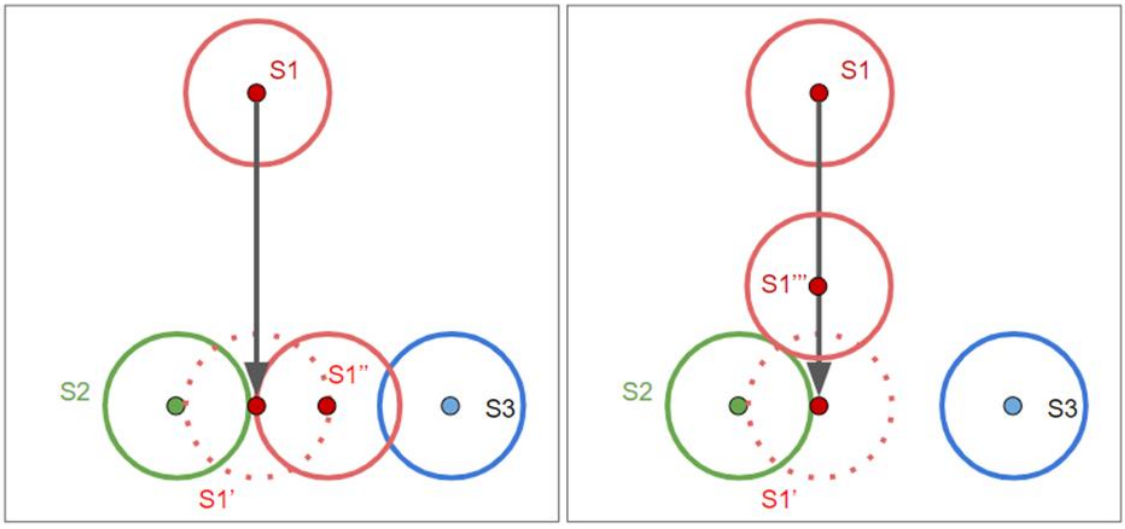

Figure 25. Illustration of Continuous Collision Detection

To calculate the exact point when the two spheres collide each other in the instance above, we calculate the delta time between two frames when the collision takes place. We know that when two spheres collide, the distance between them equals to the sum of their radii. Thus, we set up the flowing equation and try to solve Δt.

$$\text{length}(S_1 + V_1 * \Delta t, S_2 + V_2 * \Delta t) = r_1 + r_2$$

R1 and r2 are given radius. The current position of S_1 and S_2 are known. We also know the predict position S_1' and S_2', then V1 and V2 can be calculated by

$$V_1 = S_1' - S_1$$
$$V_2 = S_2' - S_2$$

Extend the equation we have the following form.

$$\text{length}(V_1, V_2)^2 * \Delta t^2 + 2 * \text{dot}(S_2 - S_1, V_2 - V_1) * \Delta t + \text{length}(S_1, S_2)^2 - (r_1 + r_2)^2 = 0$$

We realize it's a quadratic equation against Δt. Let:

$$a = \text{length}(V_1, V_2)^2$$
$$b = 2 * \text{dot}(S_2 - S_1, V_2 - V_1)$$
$$c = \text{length}(S_1, S_2)^2 - (r_1 + r_2)^2$$

The final Δt shall be obtain from:

$$\Delta t = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Following is the pseudo code of the implementation of CCD.

```
//A function returns the delta time between this moment "t0" and the next at "t1"
//the delta time is the time when two spheres physically contact each other
//s1, s2 are starting position of two spheres; v1, v2 are velocity; r1, r2 are radii
//this function solved the quadratic equation: length(s1 + v1*dt, s2 + v2*dt) = r1 + r2;
//this function does not check discriminant because it is called whenever collision happens
float timeOfCollision(vec3 s1, vec3 s2, vec3 v1, vec3 v2, float r1, float r2){
|       float a = length(v1, v2) * length(v1, v2);
|       float b = 2 * dot((s2 - s1), (v2 - v1));
|       float c = length(s1, s2) * length(s1, s2) - (r1 + r2) * (r1 + r2);
|       return (-b - sqrt(b*b - 4*a*c)) / 2*a;
}

//pos_s1 and pos_s2 represent the reference of actual position of collision
void posOfCollision(vec3 s1, vec3 s2, vec3 v1, vec3 v2, float r1, float r2, vec3& pos_s1,
vec3& pos_s2){
|       pos_s1 = s1 + v1 * timeOfCollision(s1, s2, v1, v2, r1, r2);
|       pos_s2 = s2 + v2 * timeOfCollision(s1, s2, v1, v2, r1, r2);
}
```

Figure 26. Pseudocode of Continuous Collision Detection.

Overall Application

The process of the executable to be implemented by this study is shown in figure below. The blue arrows indicate a closed loop that will be carried every frame.

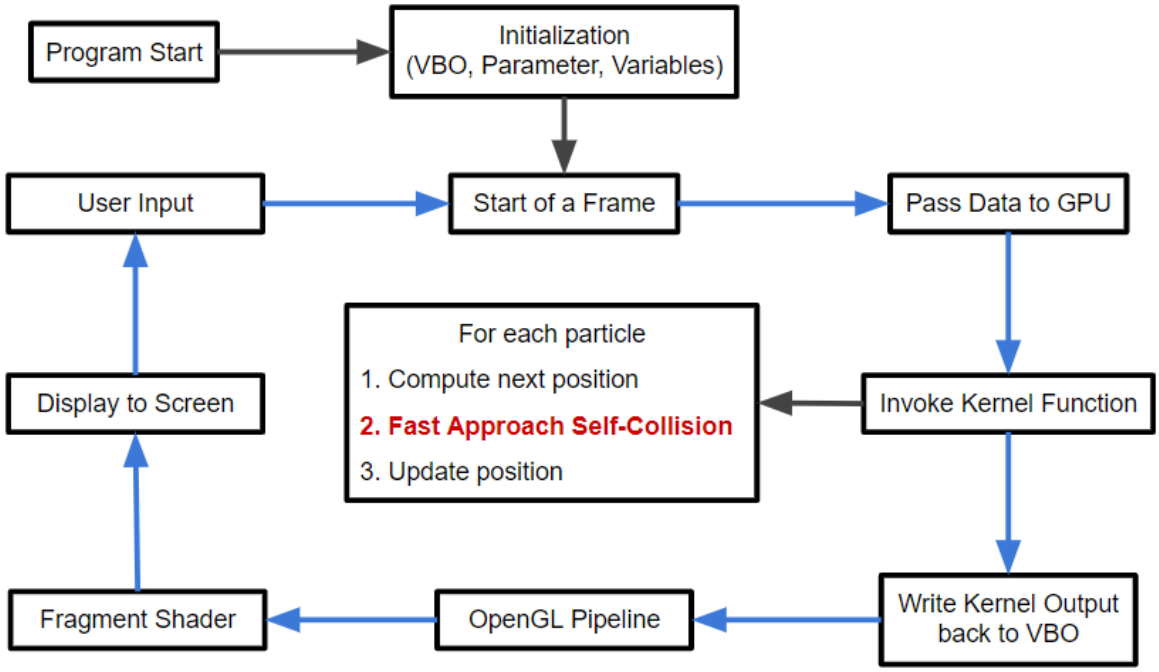

Figure 27. Flow chart of the application

The fast approach self-collision detection algorithm in the picture above is implemented with the pseudo code shown below. This is a method to be called by GPU.

```
//The function takes 2 arguments
//1st is the current particle Position
//2nd is the cell array that holds particle ID, it is a very long array allocated in GM
void clothCloth_Collision(particlePos, cellArray[nCells]) {
    //get the cellID of the cell that holds the current particle
    //hashCellID() is a function that returns an unique integer with the given cell position
    unsigned int cellID = hashCellID(particlePos);

    //use the cellID as the index to write particle ID into the cellArray
    cellArray[cellID] = currentParticle_ID;
    //traverse the 125 neighbor cells
    for (each neighbor cell) {
        //get neighbor cell index
        neighborCellID = hashCellID(neighborCellPos);

        //access cellArray using neighbor cell index
        if (cellArray[neighborCellID] == 0) { //0 means no particle
            //if no particle in that cell, jump to next loop
            continue;}
        else {//if there is a particle
            //get particleID in that cell
            Particle_neighborCell = cellArray[neighborCellID];

            //if that particle is neighbor particle, jump to next loop
            if (Particle_neighborCell is neighbor particle) {

                continue; }
            else {
                //the particle in the neighbor cell is not neighbor particle
                //detect collision
                if (sphereCollision(currentParticle_ID, Particle_neighborCell)){
                    CollisionResponse(currentParticle_ID);}
            }
        }
    }
}
```

Figure 28. Pseudo code of the fast approach to handle self-collision

CHAPTER 4 PERFORMANCE AND CONCLUSIONS

Output

The output of the experiment is shown in the pictures.

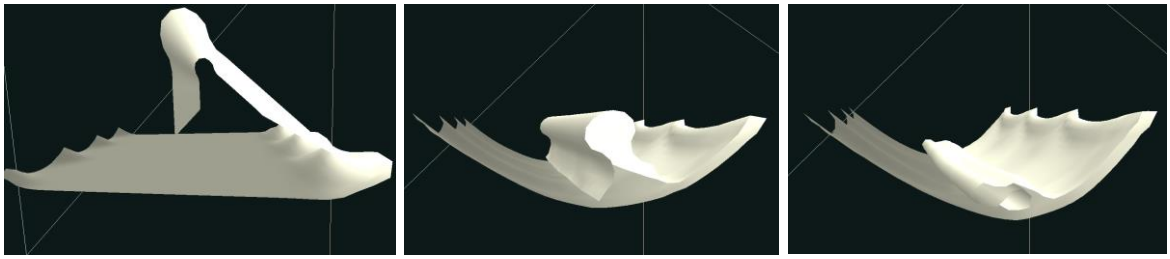

Figure 29. Output of the fast approach

To test the robustness of the fast approach, the cloth object is initialized with one edge perpendicular to itself in that this could be an unfavorable circumstance for self-collision algorithms. The result proved that this fast approach successfully prevented the cloth object from cutting through itself while remain steady at the same time.

Performance

The performance of the implementation of this study is listed in the table down below.

Table 3. Performance of the output of the self-collision method

Number of Vertices	Frame Per Second	Time Spent in GPU Computation (millisecond)	Frame Per Second	Time Spent in GPU Computation (millisecond)
	With fast approach	With fast approach	With no self-collision	With no self-collision
30 * 60 = 1,800	58.84 ± 3.54	4.12 ± 0.73	56.88 ± 6.59	1.44 ± 0.24
68 * 136 = 9,248	59.91 ± 0.73	4.62 ± 0.36	59.91 ± 0.45	1.60 ± 0.29
91 * 182 = 16,562	59.95 ± 0.31	6.70 ± 0.27	59.27 ± 3.64	2.42 ± 0.35
110 * 220 = 24,200	59.75 ± 1.19	7.42 ± 0.53	59.55 ± 3.09	2.58 ± 0.35
126 * 252 = 31,752	59.94 ± 0.37	8.56 ± 0.74	59.99 ± 0.05	2.95 ± 0.39
140 * 280 = 39,200	59.91 ± 0.39	12.29 ± 0.84	59.99 ± 0.08	3.95 ± 0.40
153 * 306 = 46,818	47.69 ± 4.74	16.70 ± 2.42	59.30 ± 3.81	4.80 ± 0.41
165 * 330 = 54,450	46.04 ± 2.32	18.68 ± 1.22	59.71 ± 2.32	5.17 ± 0.35
176 * 352 = 61,952	44.78 ± 2.66	19.29 ± 1.49	59.99 ± 0.05	5.73 ± 0.36
186 * 372 = 69,192	39.85 ± 0.96	21.91 ± 0.46	59.62 ± 2.65	5.57 ± 0.35
196 * 392 = 76,832	35.16 ± 1.76	25.19 ± 1.13	59.88 ± 1.07	6.27 ± 0.35
206 * 412 = 84,872	33.62 ± 1.87	26.46 ± 1.41	59.99 ± 0.02	6.48 ± 0.34
215 * 430 = 92,450	29.69 ± 2.23	30.85 ± 2.81	59.99 ± 0.02	6.88 ± 0.33
223 * 446 = 99,458	28.74 ± 2.58	32.09 ± 3.34	59.99 ± 0.03	7.27 ± 0.46
231 * 462 = 106,722	25.83 ± 2.35	35.96 ± 3.90	59.99 ± 0.02	8.04 ± 0.33
240 * 480 = 115,200	24.10 ± 1.38	37.24 ± 1.98	59.66 ± 2.95	8.01 ± 0.38

It can be seen from the table that when the input number of vertices is 30000 or lower the FPS basically holds stable at 59, this is because OpenGL lock the frame rate at 60 FPS no matter how fast the method is.

To perceive the collected data more clearly, the following charts indicates the change of FPS and time consumed in GPU according to the linear increase in the number of vertices.

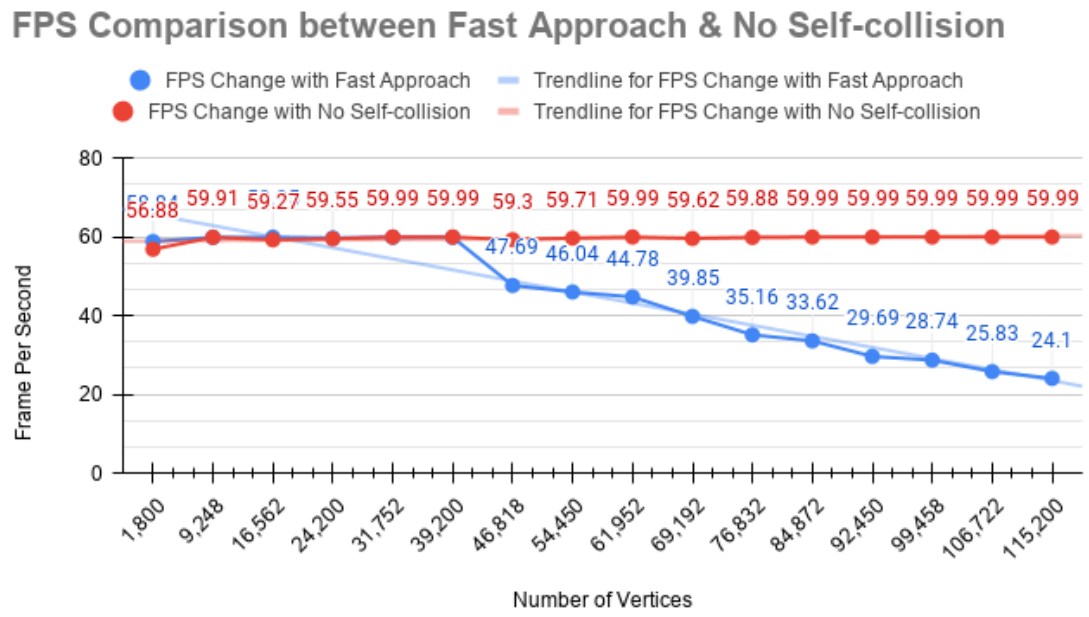

Figure 30. Comparison of FPS against increasing number of vertices between running with the fast approach and without

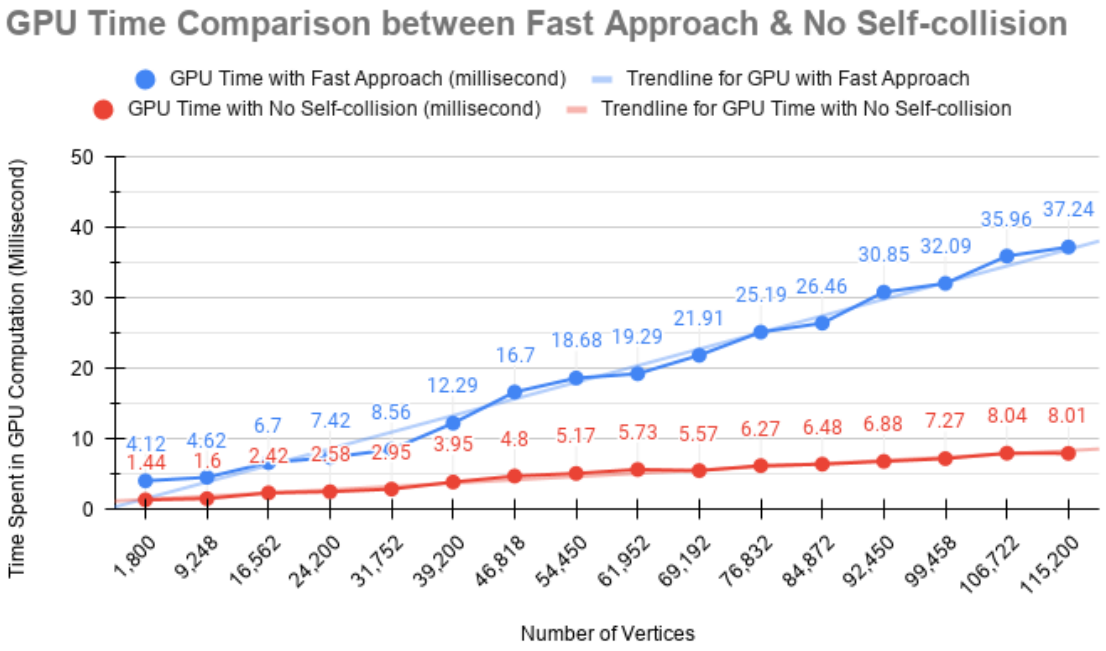

Figure 31. Comparison of Time (millisecond) spent in kernel function between running with fast approach and without against increasing number of vertices.

Noted that the time spent in kernel invocation is linearly correlated to the cloth resolution. This observation significantly undermines the design of the algorithm described in this study, since the time complexity for each thread is O(1), which means technically the time spent in GPU computation should be constant regardless the varying in the number of vertices.

However, simply because threads are parallelized in GPU does not mean they are run simultaneously. The number of threads that GPU can process at the same time is limited, that is why they are processed in batches when there are too many of them. Even though for each thread is O(1), it is O(n) for the GPU where "n" is the number of threads. This explains the linear change of time in terms of the number of vertices, where each vertex is represented by one thread in the application.

To have a better understanding of the performance of the fast approach self-collision method, a brute force search method was recorded. Below is the statistics of the performance running the same application with brute force algorithm.

Table 4. Performance of running self-collision detection with brute force search with the same increase step of vertex amount.

Number of Vertices	FPS with brute force	GPU Time with brute force (millisecond)
30 * 60 = 1,800	38.04 ± 3.87	21.99 ± 0.58
68 * 136 = 9,248	8.94 ± 0.20	107.97 ± 0.96
91 * 182 = 16,562	2.67 ± 0.23	375.05 ± 3.73
110 * 220 = 24,200	1.74 ± 0.06	566.16 ± 7.10
126 * 252 = 31,752	1.35 ± 0.21	741.16 ± 9.51
140 * 280 = 39,200	0.71 ± 0.04	1358.65 ± 17.05
153 * 306 = 46,818	0.53 ± 0.09	2123.08 ± 23.69
165 * 330 = 54,450	0.44 ± 0.11	2542.27 ± 38.28
176 * 352 = 61,952	0.43 ± 0.17	3681.43 ± 98.09
186 * 372 = 69,192	0.59 ± 0.16	4021.34 ± 42.21
196 * 392 = 76,832	0.27 ± 0.15	5421.01 ± 97.01
206 * 412 = 84,872	0.40 ± 0.19	6080.51 ± 80.63
215 * 430 = 92,450	0.33 ± 0.18	6593.60 ± 60.84
223 * 446 = 99,458	0.32 ± 0.19	8003.76 ± 59.82
231 * 462 = 106,722	0.35 ± 0.19	9893.80 ± 47.89
240 * 480 = 115,200	0.99 ± 0.39	10709.60 ± 94.78

The statistics above is visualized by the chart below.

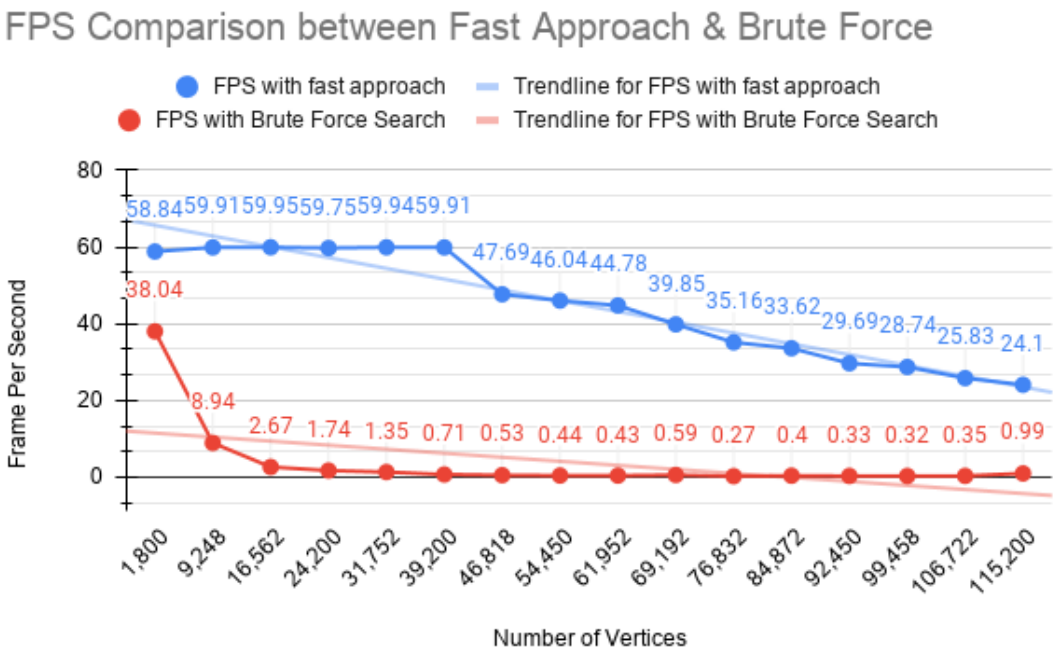

Figure 32. FPS comparison between running with the fast approach self-collision algorithm and brute force search.

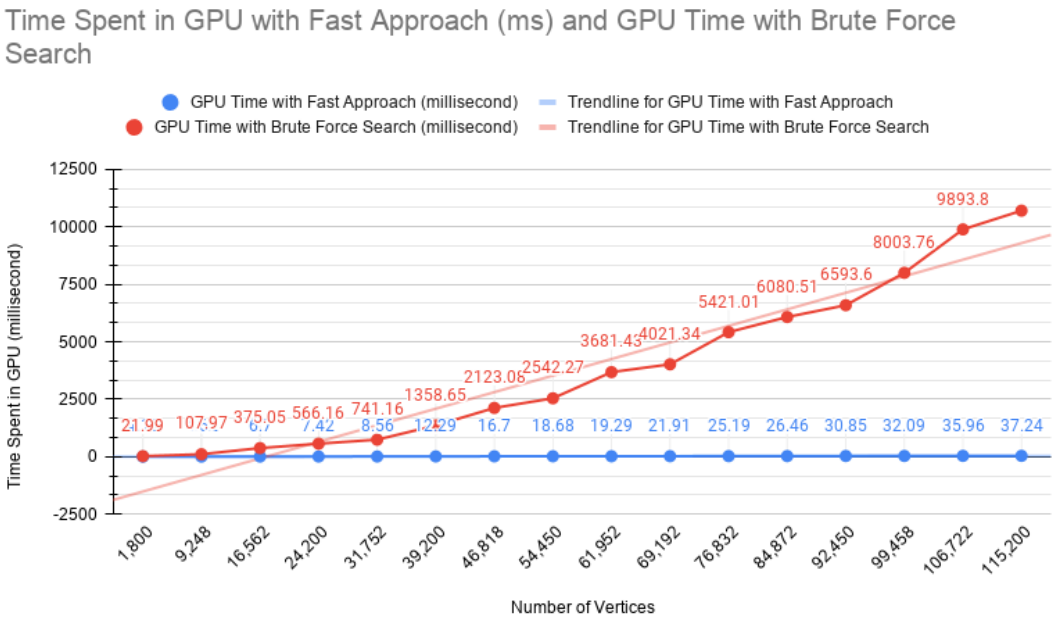

Figure 33. Time spent in executing kernel function in millisecond between the fast approach method and brute force search.

The following table gives the result of memory consumption on GPU.

Table 5. Memory consumption of the implementation

Number of Vertices	Number of Partitioned Cells	Memory Consumption (MB)
30 * 60 = 1,800	$3.369 * 10^5$	1.306
68 * 136 = 9,248	$3.932 * 10^6$	15.12
91 * 182 = 16,562	$9.432 * 10^6$	36.17
110 * 220 = 24,200	$1.676 * 10^7$	64.21
126 * 252 = 31,752	$2.521 * 10^7$	96.53
140 * 280 = 39,200	$3.448 * 10^7$	131.98
153 * 306 = 46,818	$4.508 * 10^7$	172.50
165 * 330 = 54,450	$5.643 * 10^7$	215.89
176 * 352 = 61,952	$6.870 * 10^7$	262.78
186 * 372 = 69,192	8.118 * 107	310.47
196 * 392 = 76,832	$9.484 * 10^7$	362.67
206 * 412 = 84,872	$1.103 * 10^8$	421.73
215 * 430 = 92,450	$1.252 * 10^8$	478.66
223 * 446 = 99,458	$1.396 * 10^8$	533.67
231 * 462 = 106,722	$1.552 * 10^8$	593.26
240 * 480 = 115,200	$1.743 * 10^8$	666.22

The relationship between the number of vertices and memory consumption are visualized in the chart below.

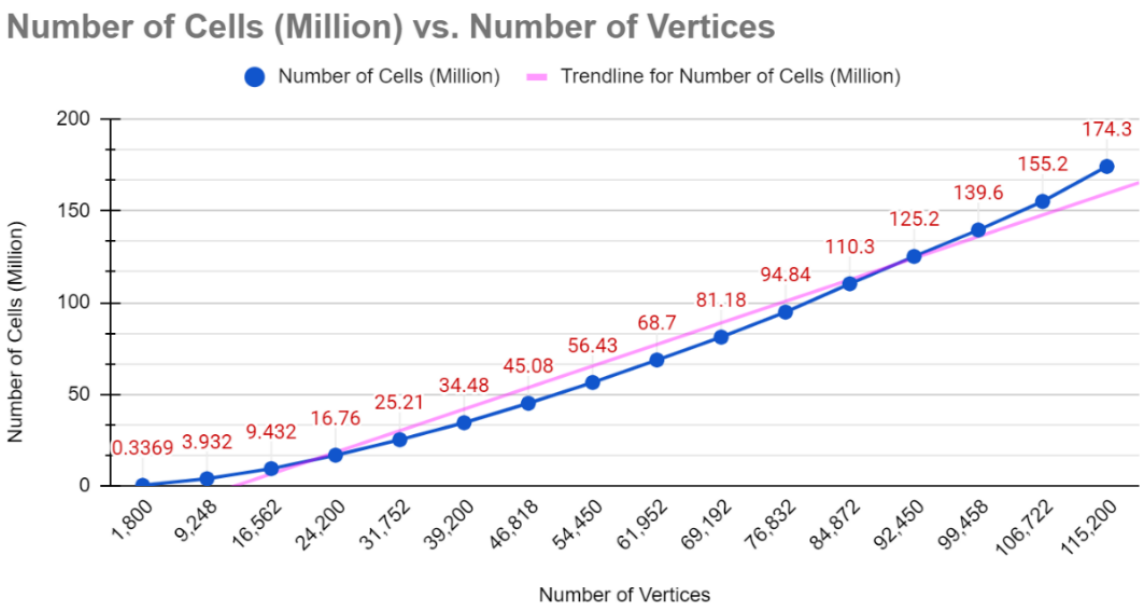

Figure 34. Number of hashing cells in million versus number of vertices.

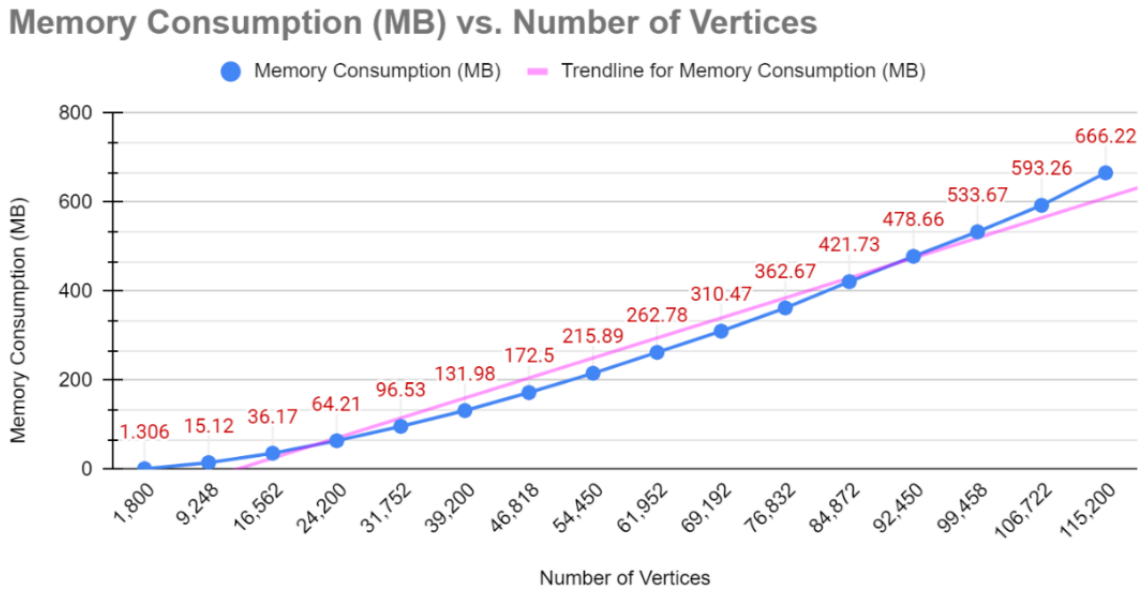

Figure 35. Memory consumption in Megabytes versus number of vertices.

Noted that the memory recorded only concerns the fast approach self-collision algorithm, it does not include those consumed by cloth generative method. That's why there are no statistics for memory measurement for experiments run with no self-collision algorithm or brute force.

Conclusion

The greatest advantage of the proposed self-collision detection algorithm is speed. The frame rate basically reaches to 60 FPS when cloth resolution is set to 20,000 vertices or below. This means the cloth object has approximately 40,000 triangles which is more than enough for most real time rendered graphics products such as commercial video games. In most triple-A games on PC or console, the number of polygons for a fine modeled character top at 35,000.

The method described in this study is featured with a O(1) time complexity for each unit resource allocated from GPU, because a single thread only needs to traverse limited data to compute collision detection regardless of the cloth resolution. In addition, in some worst cases when vertices gather and chains of collisions occur, the cost is still maintained at a relatively low level because the CCD process is not expensive as well.

While the visual effect of the self-collision handling method in this study is not entirely physically correct, it prevents the cloth from clipping into itself which is often seen ruining the realism.

However, the biggest flaw of this method is the global memory consumption on GPU which is O(n), where n is the number of vertices. Most of the cost comes from the spatial hashing technique since we need to distinguish which vertex is held in every cell. In addition, this problem is amplified because the dimension of cells must be set to a degree such that no more than 1 sphere should appear in a single cell. The smaller the size of the cell, the more the number of it. The size of the spatial hashing array allocated on the global memory equals the number of cells partitioned in the space where the cloth object is marked to have self-collisions. The method described in this study exchanges the memory for the speed.

Future Works

Optimizing Memory Consumption

As explained earlier, the biggest memory consumption comes from the integer array on the GPU. Because the index of this array corresponds to the hash ID of each cell and the value represents the sphere ID, what happens in runtime is that more than 90 percent of the value in the spatial hashing array is “0”. This leaves us the space of optimization.

The idea is that since there are too many elements in the spatial hashing array holding the value “0”, there is no point in saving it from the first place. One way to save the memory in this case is to use a Sparse Octree. The process is illustrated with a 2D example below. Suppose there is an 2D area that is spatial hashed.

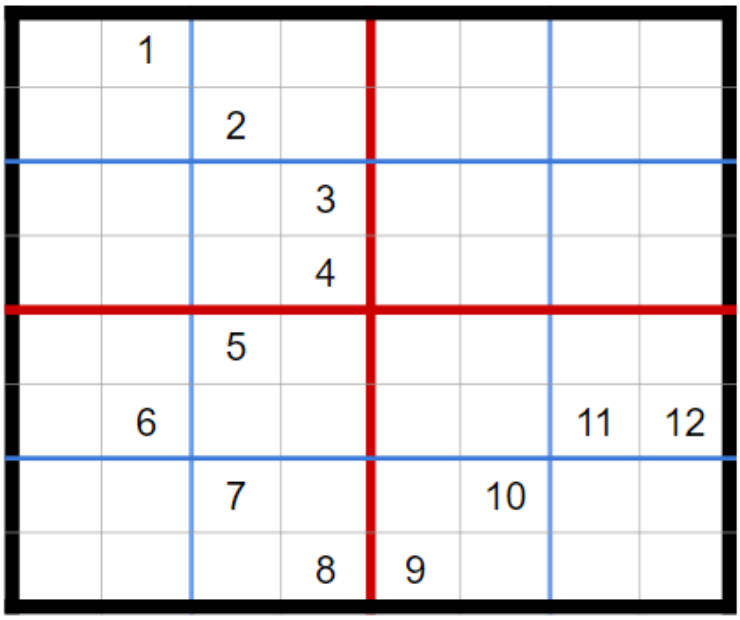

Figure 36. A 2D area which has been spatial hashed. Some grids hold a value, whereas others do not.

It’s clear to see that most grids do not have a value. The spatial hashing in this study assigns a “0” to those without a value and store all of them. A more efficient way is to use a quadtree and the data can be organized in a way shown in the following picture.

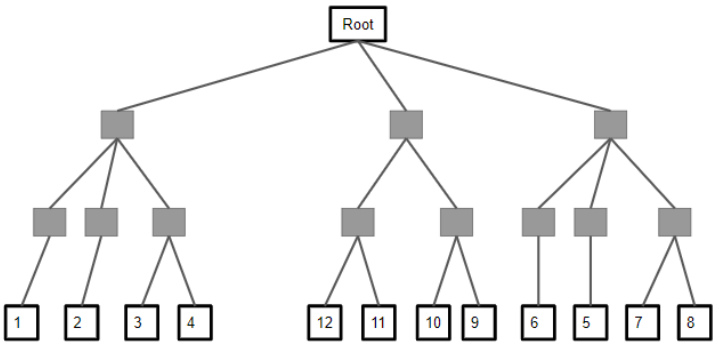

Figure 37. The layout of the quadtree data structure given the circumstance in the previous picture.

The tradeoff is that this hierarchy has to be updated every frame, and in this case it takes $O(\log_4 n)$ to construct the tree. “n” is the number of vertices. For each thread, it takes another $O(\log_4 n)$ to search the tree and obtain information from nearby hashed cells to get the ID from other thread.

Dynamic Spatial Hashing

Another hidden problem with the spatial hashing technique is that it qualifies the space in which the cloth object is able to perform self-collision. Essentially, spatial hashing marks the coordinate of a particular area of space. If the cloth object were moved out of the boundary spatial hashing is no more and each thread has no access to the nearby spheres.

To solve this, we may use an AABB to represent the Spatial Hashing area and attach its position to one or more vertices on the cloth object. This way, the cloth object is capable of self-collision detection regardless of its whereabout.

APPENDIX SOURCE CODE

Fast approach self-collision detection

```
__device__
void clthClthCollision(glm::vec3 Pos, glm::vec3& nextPos, unsigned int x, unsigned int y)
{

    //get the cellID
    unsigned int cllID = hashCellID(Pos, fxVar.spcSt, fxVar.spcDim, fxVar.cellUnit);
    hashID[x * fxVar.height + y] = cllID;
    //write cell and particle information to spatial hash array
    cellArray[cllID] = x * fxVar.height + y;

    //sync thread before proceed to collision detection
    __syncthreads();

    //solution fast but consumes memory
    glm::vec3 cellPos = getCellCoord(Pos, fxVar.spcSt, fxVar.spcDim, fxVar.cellUnit);
    glm::vec3 tempCellPos, mdir = glm::vec3(0.0f);
    unsigned int tempCellID, tempx, tempy = 0;
    bool collFlag = false;
    //traverse the 125 neibor cell
    for (int tx = -2; tx < 3; tx++) {
        for (int ty = -2; ty < 3; ty++) {
            for (int tz = -2; tz < 3; tz++) {
                tempCellPos.x = cellPos.x + tx * fxVar.cellUnit;
                tempCellPos.y = cellPos.y + ty * fxVar.cellUnit;
                tempCellPos.z = cellPos.z + tz * fxVar.cellUnit;

                //get neighbor cellID
                tempCellID = hashCellID(tempCellPos, fxVar.spcSt, fxVar.spcDim,
fxVar.cellUnit);
                //if nothing in that cell continue
                if (cellArray[tempCellID] == 0) {
                    continue;
                }
                else {
                    //get the thread index of that particle
                    getIndFromParticleID(cellArray[tempCellID], tempx, tempy);
                    //if neighbor thread
                    if ((tempx - x) * (tempx - x) + (tempy - y) * (tempy - y) < 6) {
                        continue;
                    }
                    else {
                        //detect collision
                        mdir = nextPos - nextPosArray[tempx * fxVar.height +
tempy];//access nextPos from other thread
                        if (glm::length(mdir) < 2.0f * fxVar.sphR) {

                            //for debug
                            collFlag = true;

                            //move particle
                            nextPos += glm::normalize(mdir) * 0.5f * (2.0f * fxVar.sphR -
glm::length(mdir));
                        }
                    }
                }
            }
        }
    }
```

Kernel Function of Cloth Computation

```
__global__
void computeParticlePos_Kernel(unsigned int width, unsigned int height) {
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x > width -1 || y > height -1) return;

    //current pos and last frame pos
    glm::vec3 Pos = readFromVBO(ppReadBuff, x, y, fxVar.OffstPos);
    glm::vec3 lastPos = readFromVBO(ppWriteBuff, x, y, fxVar.OffstPos);
    //normal
    glm::vec3 normal = ComputeNomral(ppReadBuff, x, y, Pos);
    writeToVBO(normal, ppWriteBuff, x, y, fxVar.OffstNm);
    //ForceNet
    glm::vec3 ForceNet = computeForceNet(Pos, ppReadBuff, ppWriteBuff, x, y);
    //Acceleration
    glm::vec3 Acc = ForceNet/cVar.M;
    //velocity
    glm::vec3 Vel = readFromVBO(ppReadBuff, x, y, fxVar.OffstVel);
    Vel += Acc * cVar.stp;
    writeToVBO(!cVar.frz ? Vel: glm::vec3(0.0f), ppWriteBuff, x, y, fxVar.OffstVel);

    //init next position
    glm::vec3 nextPos;
    //pin/unpin vertices
    if ((x == width/2 && (y % 16== 0)) || (x ==width/2 && y == height-1)) {
        glm::vec3 dir = glm::vec3(0.0f, 0.0f, 0.0f) - Pos;
        //glm::vec3 dir = glm::vec3(1.0f, 0.0f, 0.0f);
        nextPos = Pos + 0.001f * (!cVar.frz ? dir : glm::vec3(0.0f)) * cVar.folding;
    }
    else {
        nextPos = !cVar.frz? VerletAlg(Pos, lastPos, Acc, cVar.stp): Pos;
    }

    //before collision store temporary nextPos
    nextPosArray[x * fxVar.height + y] = nextPos;

    __syncthreads();

    clthClthCollision(Pos, nextPos, x, y);

    //collision Color
    debugCollisionCol(x, y);

    //fix damping term bug
    fixDampTermBug(Pos, nextPos, x, y);
    //update next position to VBO
    writeToVBO(nextPos, ppWriteBuff, x, y, fxVar.OffstPos);

}
```

Mass Spring Model

```
__device__
glm::vec3 computeForceNet(glm::vec3 currPos, float* readBuff, float* writeBuff,
                          unsigned int x, unsigned int y  ) {


    glm::vec3 innF = computeInnerForce(ppReadBuff, ppWriteBuff, x, y, currPos);

    glm::vec3 vel = readFromVBO(ppReadBuff, x, y, fxVar.OffstVel);

    glm::vec3 Fwind = cVar.WStr *
        glm::vec3(1.0f + cVar.WDir.x * glm::sin(cVar.offsCo.x * currPos.z + cVar.cyclCo.x
* cVar.time),
            cVar.WDir.y * glm::sin(cVar.offsCo.y * currPos.y + cVar.cyclCo.y *
cVar.time),
            cVar.WDir.z * glm::cos(cVar.offsCo.z * currPos.y + cVar.cyclCo.z *
cVar.time));

    //*********************************
    //F = m*g + Fwind - air * vel* vel + innF - damp = m*Acc;
    glm::vec3 netF =
        + 1.0f * cVar.M * glm::vec3(0.0f, cVar.g, 0.0f)
        + 1.0f * Fwind
        - 1.0f * cVar.a * vel * (glm::length(vel))
        + 1.0f * innF
        - 1.0f * cVar.Dp * vel * (glm::length(vel));

    return netF;
}
```

Continuous Collision Detection

```
__device__
float deltaTimeCCD(glm::vec3 s1, glm::vec3 s2, glm::vec3 v1, glm::vec3 v2, float r) {
    float a = glm::length(v1- v2) * glm::length(v1 - v2);
    float b = 2.0f * glm::dot((s2 - s1), (v2 - v1));
    float c = glm::length(s1 - s2) * glm::length(s1 - s2) - 4 * r * r;
    return (-b - glm::sqrt(b * b - 4 * a * c)) / (2.0f * a);
}
```

REFERENCES

Adam Wojciechowski, Tomasz Gałaj. (2016, 8). GPU Assisted Self-Collisions of Cloths. *Journal of Applied Computer Science, 24*(2), 39-54. doi:https://eczasopisma.p.lodz.pl/JACS/article/view/266

B. Eberhardt, A. Weber, W. Strasser. (1996, September). A fast, flexible, particle-system model for cloth draping. (T. Möller, Ed.) *IEEE Computer Graphics and Applications, 16*(5), 52 - 59. doi:10.1109/38.536275

Baraff, D., Witkin, A. (1998). Large steps in cloth simulation. In S. Cunningham, W. Bransford, S. Aluru, M. Wang, C. C. Yang, M. F. Cohen, & T. Murali (Ed.), *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (pp. 43-54). Orlando Florida USA: Association for Computing MachineryNew YorkNYUnited States. doi:https://doi.org/10.1145/280814.280821

Bing He, Liu Cheng. (2010, July). A Quad Tree Based Self-collision Detection Method for Cloth Simulation. (L. Tan, & N. Lee, Eds.) *Journey of Computers, 5*(7), 1070-1077. doi:doi: 10.4304/jcp.5.7.1070-1077

David R. Haumann, Richard E. Parent. (1988, November). The behavioral test-bed: Obtaining complex behavior from simple rules. (H. Huang, D. Panozzo, H. Seo, T. L. Kunii, N. Adabala, D. G. Aliaga, . . . L.-P. Chau, Eds.) *The Visual Computer*, 332–347. doi:10.1007/bf01908878

Feynman, C. R. (1987). Modeling the appearance of cloth. (D. Zeltzer, Ed.) *Thesis (M.S.)--Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1987.* doi:http://hdl.handle.net/1721.1/14924

François Lehericey, Valérie Gouranton, Bruno Arnaldi. (2015). GPU ray-traced collision detection for cloth simulation. *VRST '15: 21th ACM Symposium on Virtual Reality Software and Technology* (pp. 47-50). Beijing : Association for Computing Machinery. doi:https://doi.org/10.1145/2821592.2821615

Jen-Duo Liu, Ming-Tat Ko, Ruei-Chuan Chang. (1998). A simple self-collision avoidance for cloth animation. *Computers & Graphics, 22*(1), 117-128. doi:https://www.sciencedirect.com/science/article/abs/pii/S0097849397000873?via%3Dihub

Lv, M. Y., Li, F. M., Tang, Y., & Bi, W. H. (2007). A fast self-collision detection method for cloth animation based on constrained particle-based model. *Second Workshop on Digital Media and Its Application in Museum & Heritages (DMAMH 2007).* (pp. 140-145). Chongqing, China: IEEE. doi:https://doi.org/10.1109/DMAMH.2007.5

Miles Macklin, Kenny Erleben, M. Muller, Nuttapong Chentanez, Stefan jeschke, Zach Corse. (2020, April). Local Optimization for Robust Signed Distance Field Collision. (V. B. Zordan, Ed.) *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, *3*(1). doi:https://dl.acm.org/doi/10.1145/3384538

Min Tang, Tongtong Wang, Zhongyuan Liu, Ruofeng Tong, Dinesh Manocha. (2018, Dec). I-cloth: incremental collision handling for GPU-based interactive cloth simulation. (M. Alexa, Ed.) *ACM Transactions on Graphics*. doi:https://doi.org/10.1145/3272127.3275005

Nathan Mitchell, Mridul Aanjaneya, Rajsekhar Setaluri, Eftychios Sifakis. (2015, October). Non-manifold level sets: a multivalued implicit surface representation with applications to self-collision processing. *ACM Transactions on Graphics*. doi:https://doi.org/10.1145/2816795.2818100

Nur Saadah Mohd Shapri, Abdullah Bade. (2020). An efficient self-collision handling between cloth surfaces based on spherical cluster technique. In Y. Hyun Seung, K. Enami, N. Magnenat-Thalmann, C. Jim, S. Inoue, P. Zhigeng, & J.-I. Park (Ed.), *VRCAI '10: Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and its Applications in Industry* (pp. 215–220). Seoul, South Korea: Association for Computing Machinery. doi:https://doi.org/10.1145/1900179.1900225

Pascal Volino, Martin Courchesne, Nadia Magnenat Thalmann. (1995). Versatile and efficient techniques for simulating cloth and other deformable objects. In S. G. Mair, & R. Cook (Ed.), *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (pp. 137–144). New York: Association for Computing Machinery. doi:https://doi.org/10.1145/218380.218432

Provot, X. (1995). Deformation constraints in a mass-spring model to describe rigid cloth behaviour. In W. A. Davis, & P. Prusinkiewicz (Ed.), *Proceedings of Graphics Interface '95* (pp. 147-154). Quebec, Canada: Canadian Human-Computer Communications Society. doi:https://doi.org/10.20380/GI1995.17

Provot, X. (1997). Collision and self-collision handling in cloth model dedicated to design garments. *Computer Animation and Simulation*, 177-189. doi:https://doi.org/10.1007/978-3-7091-6874-5_13

R. Bridson, S. Marino, R. Fedkiw. (2003). Simulation of clothing with folds and wrinkles. In R. Parent, K. Singh, D. Breen, & M. C. Lin (Ed.), *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (pp. 28–36). Goslar, Germany: Eurographics Association. doi:https://doi.org/10.1145/1198555.1198573

Robert Bridson, Ronald Fedkiw, John Anderson. (2002, July). Robust treatment of collisions, contact and friction for cloth animation. (M. Alexa, Ed.) *ACM Transactions on Graphics*, *21*(3). doi:https://doi.org/10.1145/566654.566623

Simon Pabst, Artur Koch, Wolfgang Straßer. (2010, July). Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Computer Graphics Forum, 29*(5), 1605-1612. doi: https://doi.org/10.1111/j.1467-8659.2010.01769.x

Somasundaram, A. (2016). Selective and dynamic cloth fold smoothing with collision resolution. In A. Pearce, E. Enderton, C. Horvath, & J. Gibbs (Ed.), *DigiPro '16: Proceedings of the 2016 Symposium on Digital Production* (pp. 11-12). New York: Association for Computing Machinery. doi:https://doi.org/10.1145/2947688.2947691

Tosiyasu L. Kunii, Hironobu Gotoda. (1990, November). Singularity theoretical modeling and animation of garment wrinkle formation processes. (N. Magnenat-Thalmann, H. Huang, D. Panozzo, & H. Seo, Eds.) *The Visual Computer*, 326-336. doi:https://doi.org/10.1007/BF01901019

Vassilev, T. I. (2010). Comparison of several parallel API for cloth modelling on modern GPUs. In B. Rachev, & A. Smrikarov (Ed.), *CompSysTech '10: Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies* (pp. 131-136). Sofia, Bulgaria: Association for Computing Machinery. doi:https://doi.org/10.1145/1839379.1839403

Volino, P., & Magnenat-Thalmann, N. (2001). Comparing efficiency of integration methods for cloth simulation. *Proceedings. Computer Graphics International 2001* (pp. 265-272). Hong Kong, China: IEEE. doi:10.1109/CGI.2001.934683

Weil, J. (1986). The synthesis of cloth objects. In D. C. Evans, & R. J. Athay (Ed.), *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (pp. 49-54). New York: Association for Computing Machinery. doi:https://doi.org/10.1145/15922.15891

Zeller, C. (2005). Cloth simulation on the GPU. In J. Buhler (Ed.), *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches* (pp. 39-es). Los Angeles California: Association for Computing Machinery. doi:https://doi.org/10.1145/1187112.1187158