Comparison of Collision Handling Methods for Cloth using GP-GPU

by

Toby Ross, B.Sc. (Hons)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

September 2014

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Toby Ross

September 1, 2014

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Toby Ross

September 1, 2014

Acknowledgments

I would like to thank Michael Manzke for the advice and support he has offered throughout this process.

Toby Ross

University of Dublin, Trinity College
September 2014

Comparison of Collision Handling Methods for Cloth using GP-GPU

Master of Science in Computer Science (Interactive Entertainment Technology)

Toby Ross

University of Dublin, Trinity College, 2014

Supervisor: Michael Manzke

The physically based animation of cloth has been researched for over two decades, since which it has become a common and convincing appearance in film. In real time applications such as video games, cloth simulations are starting to become common, but the simulations lag behind the standard set in CGI and literature. In particular, collisions are often neglected, especially self-collisions (where cloth hits cloth, or itself). The processes normally used to guarantee robust collision handling are poorly suited to the limitations of a real time application. This paper explores the viability of achieving robust cloth simulation in real time on the GPU.

A [Bridson et al., 2002] style pipeline is fully implemented on the GPU, and each part of the pipeline is benchmarked. An experiment is performed to discern the importance of thickness parameter (a distance within which collisions are registered) to performance. Further to this, a GPU design and partial implementation is discussed.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Physically based animation is a major area of computer graphics aimed at developing methods by which real world phenomena can be efficiently recreated in virtual worlds. It is primarily of use in the domain of entertainment. Physically based animation is used frequently for films and games.

There is a broad difference between physical simulation, and physically based animation. In the former, the interest is in precisely and accurately recreating reality, which is typically of interest for scientific and engineering pursuits. In the latter, accuracy is rarely a focus, and desired only up to the point of plausibility and appeal. Physically based animation is hence typically achieved using a mixture of simplifications and approximations. As an example, one of the major pillars of physically based animation (rigid body simulation) is based off the false assumption of true rigidity for many solid objects.

There is a considerable difference between the techniques that are viable for use in film - where simulation is performed offline - and techniques that are viable for use in games (or other real-time applications). Though it would be an over simplification to state that efficiency is a non-issue for films, it would not be unreasonable to spend seconds, or minutes, on a single frame. Comparatively, simulation in a real time application

has to be on the order of milliseconds. Games are expected to run at a minimum of 30FPS (frames per second), equating to 33ms to perform every operation, with 60FPS (16ms per frame) often preferred. Within these budgets, much is typically reserved for rendering, artificial intelligence and so forth, so realistically any physically based animation will have to be performed in a few milliseconds.

Traditionally, research into physically based animation is separated into a variety of different areas. Two of the most established areas are the previously mentioned rigid-bodies, and particle systems. Particle systems are relatively simple, and can be used to provide an impressive variety of effects at low cost. Closer to the cutting edge is simulation of deformable objects (often termed softbodies) and fluids. Fluid simulation can be used to create plausible animations of liquids, gases, as well as flame. The soft-bodies are even broader, including hair, rope, and cloth as well as solids which squash, stretch, and fracture.

Use of these different types of simulation in film are extremely common, and extremely effective both when used to simulate the fantastic, and the realistic. Games lag far behind: Physically based animation in games is used in only a small proportion of the situations where, in an ideal world with infinitely powerful computers, it could be. The small amount of time afforded to physically based animation has already been mentioned, but if anything there is another factor that makes physically based animation in games especially tricky: interactivity.

One could classify physical simulations in games into four groups: those which have no relation to the player, those which are affected by the player, those which affect the player, and those which are both affected by the player and affect the player. For the first group, offline techniques are often applicable. Simply record the simulation that takes place offline, and replay it as necessary at runtime. For the second group, care needs to be taken over the robustness of the simulation, as no longer does the designer have control over what can happen. There has to be a certainty that the player will not be able to put the simulation into a broken state. For the third group, care needs to be taken to ensure that the unpredictable physics do not have an adverse effect on the gameplay. As an example, imagine a football game which correctly simulates the knuckling of a ball - a contentious phenomenon which causes a non-spinning ball to unpredictably swerve. This would improve the realism of the simulation, but is

unpredictable realism preferable in a game? Worse, what if the physical simulation left the player in a game-over state due to some chaotic element of the simulation? Obviously, the fourth group simply combines the difficulties of the second and third. All of this means that the simulations in games have to be more robust, if anything, than those in films.

Given the above difficulties, it is unsurprising that physically based animation has not reached ubiquity in games. Yet, there are examples of all previously mentioned types of physically based animation in games. Particle systems are used so frequently that it is not worth an individual mention. Rigidbody physics have been used in many games successfully, often as a core mechanic. Angry Birds, undeniably one of the most popular titles of all time, tasks players with destroying enemy fortifications with a catapult. The fortifications are made up of simple 2D shapes, and as the catapulted birds hit the structures, they collapse, simulated using the 2D rigidbody engine Box2D. There are few examples of softbody physics in games. One of the best examples is Star Wars: The Force Unleashed (and its sequel), which utilise an engine called Digital Molecular Matter to simulate the how solids bend and break under contact. Examples including rock crumbling, glass shattering, wood splintering, plants bending, and jelly deforming and wobbling as objects are fired into it.

Cloth simulation, a subset of softbody simulation, is becoming a regular appearance in modern video games. Cloth simulation can add a great deal of impact to the visual appeal of a character. Batman: Arkham City sees the protagonist's famous cape simulated, moving with every step, jump, punch and kick, and it makes it that much easier to immerse oneself into the role. Other instances of cloth in games are similar: used for a mage's robes, or a hunter's cloak. Though cloth simulation is no longer rare in games, the simulations are far less advanced than those which would be seen in film or in the literature. As of 2014, no major title has dealt with self-collisions for cloth (where cloth collides with itself, or another bit of cloth). The simple reasoning for this is that cloth simulation is complex and expensive. Simulating the movement of the cloth in an unconstrained environment can be done relatively cheaply, but detecting collisions and then resolving them in a robust manner cannot.

Collision handling is typically the most expensive element of a physical simulation regardless of type, and this is particularly true for cloth. It is critical for the collision

detection and handling of cloth to be absolutely robust: a single missed collision can lead to a permanent and unappealing tangle, where portions of cloth self-intersect and become trapped. Ensuring that collisions aren't missed is not easy with cloth because of its thinness. The thickness of a piece of cloth is likely to be on the order of millimetres (or less), and this thwarts many standard collision handling methods. The more robust methods necessary for cloth collision detection come at a severe cost, and it is apparently for this reason that game developers decide to ignore self-collision handling altogether.

For robust and full cloth simulations to become viable in real time, more efficient simulations must be achieved. One way in which physical simulations (and physically based animation) can be accelerated is by exploiting parallelism. For some time now, computer architecture design has seen processing power split between multiple cores, rather than concentrated into a single core. This has been the preferred direction because of various issues that occur when trying to create single cores which exceed 3-4 GHz. Modern consumer processors commonly have two or four cores, with both six and eight core processors starting to become available. With graphics processing units (GPUs), the parallelism is much finer. Modern graphics cards commonly have hundreds, or thousands of cores. These are very well suited to common graphics techniques, but they are increasingly used for non-graphical applications, a concept termed GPGPU (general purpose computation on graphics processing units). The theoretical throughout of a GPU is far, far higher than that of a CPU. For well suited tasks, speed ups of 100x and above are not uncommon.

The tasks typical to a physics simulation have proven many times to be well suited to graphics processors. This is unsurprising: GPUs are suited to tasks that split easily into similar small parts. GPUs are known as SIMD processors, which means that they execute the same instruction on many pieces of data. This is precisely the kind of work necessary in a physics simulation. As example, a collision detection routine might require performing a proximity test between thousands of pairs of triangles. Early GPGPU endeavours awkwardly exploited graphics APIs such as OpenGL to perform computations, utilising vertex and fragment shaders, textures and the framebuffer for general computation. More recently, frameworks such as OpenCL and CUDA have been developed to give a more natural interface for GPGPU.

1.2 Contributions

The primary contributions of this dissertation is an exploration of cloth self-collision handling methods with a particular focus on their viability as part of real time applications such as video games. To this end, a GPGPU pipeline has been implemented for the simulation of cloth.

A popular, truly robust method for coping with self-collisions in cloth has been implemented on the GPU based on [Bridson et al., 2002] and has been analysed with a view to exploring its real time performance. Experiments have also been performed to evaluate the importance of a particular parameter (thickness) to the cost of handling collisions.

Untangling, where self-intersections are reversed, has been discussed in the literature for serial implementations, but not thus far for the GPU. This paper describes a partial design and implementation of GPU untangling based on the ideas of [Baraff et al., 2003] and [Wicke et al., 2006].

Chapter 2

State of the Art

There are a handful of things that need to be achieved to produce a high quality cloth simulation. Internal forces, those which hold the cloth together and give it is strength, must be modelled. External forces, such as gravity, or wind, must be applicable. The effects of these on the cloth's state must then be simulated using some form of integration over time. Collisions between the cloth and other surfaces (typically rigid) must be simulated, importantly ensuring that the cloth does not 'clip' through said surfaces. Finally, and perhaps most challengingly, 'self-collisions', between multiple pieces of cloth or between different parts of a single piece of cloth must be detected and dealt with.

There are a number of important factors in a cloth simulation. To produce a simulation of a cloth like object, one has to consider the internal modelling, and this is discussed in Section 2.1. To progress the simulation, integration should be considered, discussed in Section 2.2. Collisions, both between cloth and non-cloth objects, and between the cloth and itself (or two pieces of cloth) must be detected and resolved, and these issues are covered in Section 2.3. Even with the above three elements in place, simulation is expensive, so improving performance is a major concern. Finally, in Section 2.4, the parallelisation of cloth simulation will be discussed.

2.1 Cloth Models

In reality, cloth is constructed from the interweaving of threads, themselves constructed from spun fiber. The interlocking network resulting from this process has considerable strength due to friction. Different types of fiber present different physical behaviours, and the weight of the thread (or yarn), as well as the tightness or the weave, and the style of the weave, also effect physical properties.

Simulating the behaviour of cloth (and more broadly, extremely thin, flexible materials) has been an area of research for many decades. [Nealen and Müller, 2006] presents a broad overview of research into the modelling of deformable objects, including cloth.

[Terzopoulos et al., 1987] presents a continuum based model. The simulation is based on a simplified elasticity model, and differential equations are used to model deformable objects including cloth. To solve the partial differential equations rendered by this method, finite element analysis or finite differencing methods are used. The paper works on the basis that a deformable object (or in the case of cloth, a deformable surface) can be defined by its rest state (or non-deformed shape) and parameters which define how a force deforms it.

Finite element/difference methods are common tools for dealing with deformable object simulation to this day, but in the case of cloth simulation it has largely been set aside. The prevalent methods for simulating cloth since the 1990s have been particle based. [Provot, 1995] is considered seminal in its proposition of a mass-spring system, but in truth [Breen et al., 1994b] argued the viability of particle based systems over Finite Element techniques. Particle based systems, it was argued, are better positioned to represent the complicated mechanism of fibers that make up a piece of cloth.

As touched upon before, a microscopic examination of cloth highlights the network of threads. Parallel groups of thread are known as the warp and weft respectively, with the warp group perpendicular to the weft. Particle based systems posit that the crossover point between a warp and weft thread can be modelled as point masses (or particles) in a discretised model. This is justified as the tension at crossover points is so great that threads are effectively clamped together.

[Breen et al., 1994b] represents the thread-level structural constraints using energy

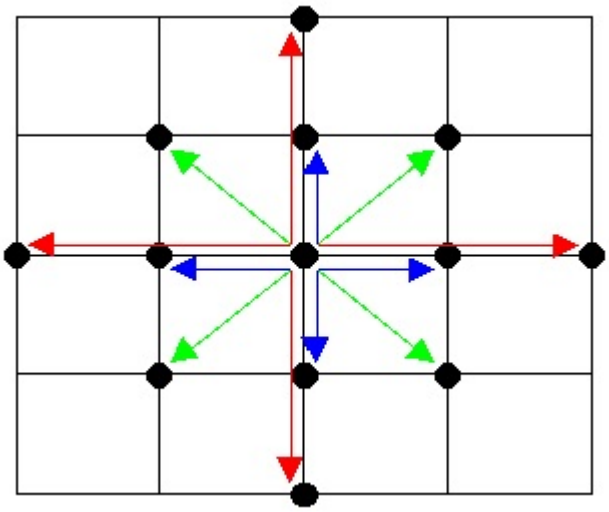


Figure 2.1: The three types of spring

functions which encapsulate the various interactions found in a piece of cloth. Particle positions are then solved for using standard energy minimisation techniques. [Provot, 1995] has a simpler solution for modelling cloth using a particle system: each particle becomes a point mass with neighbouring particles attached by springs.

The springs are attached as follows. In a rectangular mesh of particles, a coordinate pair [i, j] refers to the ith column and jth row. Three types of spring are created to give the mesh cloth like characteristics:

1. **Structural Springs** link the particles [i, j] to both [i + 1, j] and [i, j + 1] (each particle to the four directly up, down, left, and right)
2. **Shear Springs** link [i, j] to [i + 1, j + 1] and [i + 1, j] to [i, j + 1] (each particle to the four particles at north east, south east, south west, and north west)
3. **Flexion Springs** link [i, j] to [i + 2, j] and [i, j + 2] (each particle to the four particles **two** above, down, left, and right)

The four types of spring are illustrated in Figure 2.1.

The three springs give the cloth strength in specific ways. The structural springs resist deformation along the primary axes, and the shear springs resist deformation in the diagonal directions. Finally, the flexion springs resist the bending of the cloth, and

are often known simply as bending or bend springs. Each spring applies an 'internal' force to the attached particles that pulls (or pushes) the particles towards their original distance. The further from the original or resting distance the particles are, the stronger the applied force, in accordance with Hooke's Law:

$$\mathbf{F} = -k\mathbf{x} \tag{2.1}$$

where $\mathbf{F}$ refers to the force, k to the spring coefficient, and $\mathbf{x}$ to the vector between the attach points of the spring (in the case of a mass-spring system, this is the vector between the two particles).

To an impressive extent, cloth modelled using the mass-spring model is able to recreate the behaviour of cloth in the real world. One exception to this rule is 'super-elasticity' a term coined by Provot in [Provot, 1995] to refer to springs that hugely over extend, as tends to happen near attachment points (where the cloth is fixed to a point in world space). This highlights a flaw with the mass-spring model: real cloth is barely extensible (10% extension is a typical maximum in woven materials).

[Provot, 1995] offers a procedure - by his own admission 'ad hoc' - to cope with these over-elongated springs called the dynamic inverse. After each update of the cloth's position, the springs are iterated over, calculating the gap between their rest length and their current length. Any that have gone too far are then shortened, by moving the attached particles closer to each other. This process is repeated until no spring is over extended. There are various other solutions for the over-extended springs - a simple one is to strengthen over-elongated springs.

The mass-spring model is largely ubiquitous, but there has been considerable success with a more modern model. [Müller et al., 2007] presents Position Based Dynamics, where positional constraints are used in the place of springs to achieve deformable object simulation. This has applications for many types of deformable objects ([Macklin et al., 2014] uses the concept to build a unified physics system including rigidbodies, fluids and gases) but in the original paper it is used to implement a cloth simulation.

Positional constraints, as the name suggest, are functions of particle positions. For

the cloth model, two types of constraint are used. The first, a 'stretching' constraint limits the distance between adjacent particles, and the second, a 'bending' constraint limits the angle between triangles which share an edge. The key behind Position Based Dynamics is how these constraints are enforced. The constraints are formed into a non-linear system that is solved numerically using a variant of the Projected Gauss Seidel method, where a systeme is solved by iteratively solving for one variable.

One of the major advantages of Position Based Dynamics is that positional constraints can take a much wider range of forms, and hence achieve a much wider range of behaviours, than springs. It is also arguably more intuitive to create positional constraints, and to tune them, than it is to tune the springs in a mass-spring cloth model. This variety is illustrated in [Müller et al., 2007] with a balloon constraint, which models air pressure inside a closed cloth mesh.

2.2 Integration

With a cloth model in mind, it remains to decide how to use the model to advance from one time to the next. There a number of major considerations: efficiency - how fast can it be performed, accuracy - how close are the results to an analytical solution, and stability - the simulation's ability to converge to a stable state. A number of different methods have been proposed for use with cloth simulation. These are typically divided into two classes: explicit and implicit. The former calculates a future state based purely on the past state, whereas the latter solves an equation (or system) involving both the current and future state. Generally, explicit schemes are used for their efficiency, as while implicit schemes are more accurate and stable, they are considerably more expensive.

[Provot, 1995] implements the performance light explicit (or forward) euler integrator. At each state, the integrator can be used to determine a change in position and velocity which, when added to the old position and velocity, advance the state:

$$\Delta \mathbf{x} = \Delta t \mathbf{v}_t \tag{2.2}$$

$$\Delta \mathbf{v} = \Delta t \frac{\mathbf{F}_t}{m} \tag{2.3}$$

where $\mathbf{F}$ refers to force, $\mathbf{v}$ to velocity and $\mathbf{x}$ to position. Quite clearly, the procedure is extremely cheap, but it is also extremely inaccurate and more problematically, widely unstable. Stability is a complicated issue in numerical mathematics, and relates to various a number of different factors in a physics simulation. One thing that puts a lot of strain on an integrator's stability is springs, and in particular, stiff springs. Discussions on the stability of various integration methods can be found in [Eberly, 2010].

Springs are extremely sensitive to instability issues, and the stiffer the springs are, the more prone the system will be to exploding. Only when the timestep (the length of time between the current and future timestep) is lowered does the stability start to improve. While the explicit euler method is cheap to perform a single update, it is only stable if multiple updates are used per frame.

This procedure is extremely cheap, but without a small timestep (Δt), it is extremely inaccurate, and not particularly stable. One popular explicit alternative is the Runge-Kutta (or Runge-Kutta 4) method, which is far more accurate and less prone to instability.

[Baraff and Witkin, 1998] instead uses an implicit scheme, arguing that 'The bottleneck in most cloth simulation systems is that time steps must be small to avoid numerical instability'. Implicit schemes can stably take large time steps (though, dependent on how collisions are dealt with, small timesteps may still be necessary). They use the Backward Euler method, an implicit scheme:

$$\Delta \mathbf{x} = \Delta t(\mathbf{v}(t) + \Delta \mathbf{v}) \tag{2.4}$$

$$\Delta \mathbf{v} = \frac{1}{m}\mathbf{F}(t) \tag{2.5}$$

Their formulation generates a sparse system matrix where the change in velocity (acceleration) is solved for using the conjugate gradient method, and from there the change in position (velocity) follows.

Another commonly used integration technique is Verlet integration. It is very often

used for particle systems, and hence it, and variants of it, are used regularly in cloth simulation, assuming the cloth is simulated using one of the particle based models. The Verlet technique is unusual in that velocities are not directly used:

$$\mathbf{x}_{n+1} = 2\mathbf{x}_n - \mathbf{x}_{n-1} + \mathbf{a}_n \Delta t^2 \tag{2.6}$$

One such example of the use of a Verlet style integrator is in Position Based Dynamics [Müller et al., 2007], where at the end of each timestep, the velocity is set as:

$$\mathbf{v} = (\mathbf{p} - \mathbf{x})/\Delta t \tag{2.7}$$

$$\mathbf{x} = \mathbf{p} \tag{2.8}$$

Here, $\mathbf{p}$ stands for the position that has been calculated for the particle over the timestep, and $\mathbf{x}$ to the position of the particle at the beginning of the timestep. Hence, though the velocity is explicitly stored, it is 'in exact correspondence with a *Verlet* integration step'.

2.3 Collisions

As with most types of physical simulation, coping with collisions and contact is of crucial importance. It is most typically the most expensive area of the simulation, and quite often the most complicated too. Cloth itself poses a particular issue due to how thin it is. Generally speaking collision handling can be split into how collisions are detected, and how they are resolved.

2.3.1 Collision Detection

Collision detection methodologies can be classified as either discrete or continuous. Discrete methods test for collision states at a particular time, whereas continuous methods try to determine whether equations are going to occur during a time period.

For the reasons explained above, continuous collision detection, which is more costly and typically more complex than discrete collision detection, is often used.

In most cases, collision detection solutions are split into two methods: broadphase, and narrow phase. In the narrow phase, elements are specifically tested to discover whether or not a collision has occurred. However, to avoid testing n^2 elements, broadphase testing is used to quickly cull obvious cases of non-collision using various heuristics. This section will cover first broadphase improvements, before discussing narrowphase calculations.

Bounding Volume Hierarchies are a common acceleration structure for use in collision detection. The concept is discussed comprehensively in a survey on collision detection for deformable objects[Teschner and Kimmerle, 2005].

Algorithm 1 Bounding Volume Hierarchy Test

```
1: procedure TRAVERSE(A,B)
2:     if A and B do not overlap then return
3:     if A and B are leaves then return intersection
4:     else
5:         for all children of A: A[i], and B: B[j] do TRAVERSE ( A[i] , B[j] )
```

Algorithm 1 can either be used to test intersection between two objects represented by two hierarchies with top level bounding volumes M and N respectively by calling TRAVERSE(M, N). Alternatively, self-intersections can be tested by calling TRAVERSE(M, M).

The overlap test is key. As the algorithm's name suggests, the overlap test is based on cheap bounding volume tests (various types of bounding volume can be used, including axis-aligned boxes and spheres).

A number of decisions have to be made with the use of a Bounding Volume Hierarchy, such as the type of bounding volume to use, how to build it in the first instance, and how to update it each frame. Most formulations of BVHs use some kind of 'k-DOP' as bounding volume. k-DOP stands for discrete oriented polytope, and is a generalization of the axis-aligned bounding box (which is a 6-DOP). k refers to the number of directions that are used to bound the object. With an AABB, six are chosen: $(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), (0,0,-1)$. Higher degree DOPs

tend to involve using diagonal axes which result in beveled bounding volumes. A 26-DOP, for example, is a standard AABB, with each edge and vertex beveled.

There are many techniques for building bounding volume hierarchies. A standard one is to find a bounding volume that contains an entire object, before recursively splitting it down the longest side, until a single element remains in the bounding volume. For a deformable object, the BVH needs to be updated every frame. A naive way of doing this is to simply rebuild the tree every frame, but this is significantly more costly than 'refitting' it, where one simply updates the bounding volumes. A rebuilt tree will likely be of better quality, so optimal methodologies typically use refitting with periodic rebuilding.

[Bergen, 1997] explored the issue of Bounding Volume Hierarchies for collision detection of deformable objects, including a section on refitting versus rebuilding. They state that 'experiments have shown that for models composed of over 6000 triangles, refitting an AABB tree is about ten times as fast as rebuilding it'. They note the drawbacks or refitting too - the bounding volumes in a refitted tree are liable to overlap more than the boxes in a rebuilt tree. This leads to the search having to descend farther down the tree, and thus worse performance. However, in cases where the mesh is not fractured, they 'found no significant performance deterioration for intersection testing, even for more severe deformations'.

The Bounding Volume Hierarchy is not the only high-level culling concept suggested for use with cloth. There are various instances where spatial subdivision has been proposed as an alternative, including [Teschner et al., 2003]. The scheme works by implicitly splitting the world into small AABBs, or, into a 3-D grid of cells. In a first pass, for each vertex, the cell is determined, and the coordinates are hashed, before being stored in a hash table. In the second pass, elements (eg. triangles or tetrahedra) are looped over. For each element, a standard hashtable lookup is performed with any cell the element intersects. These vertex-element pairs are then highlighted as those being potential colliders.

The method is efficient, but does not highlight edge-edge collisions. This is justified on the basis that 'the relevance of an edge test is unclear in case of densely sampled objects'. Many cloth simulations do in fact deal with the edge-edge case, but, strong

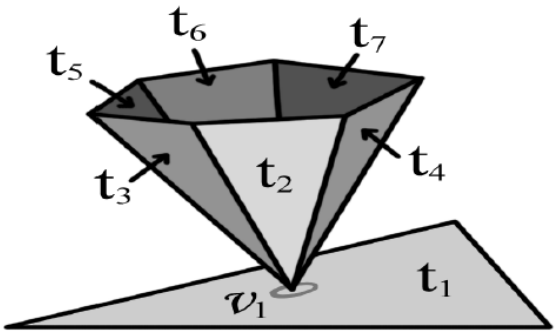


Figure 2.2: A single vertex, shared between six triangles

results have been achieved without (such as with [Müller et al., 2007]).

Collision detection is the major bottleneck in cloth simulation, and a lot of work has gone into trying to improve high and low level culling algorithms to reduce the number of elemental tests that have to be performed. One such example is Representative Triangles, the primary contribution of [Curtis et al., 2008]. It seeks to solve the problem that many features (vertices and edges) are shared between elements (triangles), and hence many duplicated elementary tests occur in a naive setup. With Representative Triangles, each feature is 'represented' by a just one triangle.

As example, illustrated in Figure 2.2, a single vertex, v_1, collides with a triangle, t_1. The vertex is shared between six triangles, t_2 through t_7. The vertex may end up being tested against t_1 for each of the six triangles, despite only one test being necessary. However, with representative triangles, just one triangle, say t_2 would be v_1's representative. Culling elementary tests in this way was not novel, but the strategy of culling them by attaching features to their representative triangles is far more efficient than previous strategies of using run-time databases to detect whether a particular test has been used before. The database method is far more costly than representative triangles, leading to improved performance in experiments - the method is enormously better than not culling redundant elementary tests at all.

Other culling algorithms have been suggested for similar purposes. [Tang et al., 2009a] proposes the orphan set, which deals with the high level of redundancy in the element tests between adjacent triangles. Where 15 tests are necessary by default between two

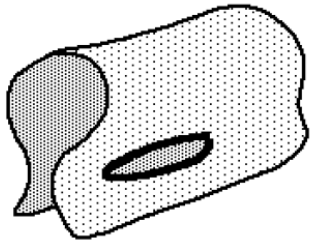


Figure 2.3: Self-Intersection due to a Loop

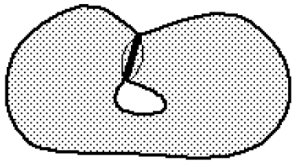


Figure 2.4: Self-Intersection due to Contour Overlap

triangles, only four are actually necessary. More over, many of these tests will have already been performed in tests between non-adjacent triangles. Another example is the non-penetration filter from [Tang et al., 2010] which uses a conservative coplanarity test to cull elementary tests prior to solving the cubic equations used for continuous collision detection.

Very specifically, some optimisations have been proposed for the use in cloth collision handling. [Volino and Thalmann, 1994] makes a number of useful contributions including some insights on the self-intersection of cloth. The paper describes the two properties of a surface which can lead to self-intersection.

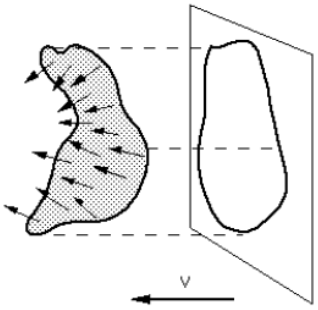


Figure 2.5: Property that ensures no self-intersection

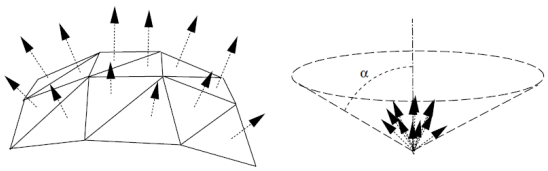


Figure 2.6: A Normal Cone contains all of the normals of a surface

1. The surface is curved enough that it loops around and intersects as in Figure 2.3
2. The contour of the surface is overlapping as in Figure 2.4

These are reworded as a pair of mathematical properties which, if both true, guarantee a surface does not self-intersect. These are illustrated in Figure 2.5.

1. There exists a vector, $\mathbf{V}$, for which the dot product with any normal from the surface is positive.
2. The projection of the surface's contour onto a plane orthogonal with $\mathbf{V}$ does not self-intersect.

The first property can be efficiently dealt with using the concept of the normal cone proposed in [Provot, 1997]. For a particular surface, its normal cone is a cone which contains all of the normals of the surface as illustrated in Figure 2.6. Such a cone can be described entirely by its axis (the vector from the cone's origin point through the center of the cone), and its apex angle (the angle between the two edges of the cone). If the apex angle of the cone is smaller than Π, then it suffices to say that there is a vector for which the dot product is greater than 0 for every normal.

The normal cone test can be made part of a Bounding Volume Hierarchy broadphase. The cones for each node can be built recursively by working bottom up a hierarchy as part of a standard refit.

As for the contour test, there is considerable thinking that it can in fact be ignored, as is done in both [Volino and Thalmann, 1994] and [Provot, 1997]. The latter justifies this on the basis that self-collision due to a severely non-convex contour 'never happened in our simulations, even when modelling clothes using real clothes patterns'. Even so, [Tang et al., 2009a] implements a continuous version of the contour test.

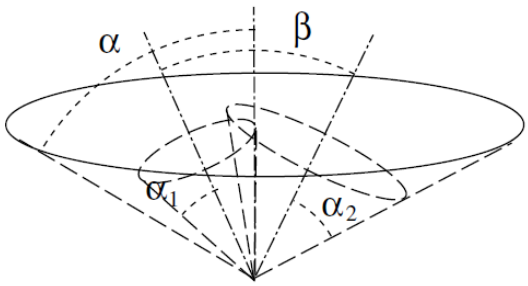


Figure 2.7: A new cone is created from two sub-cones

Moving onto the narrowphase testing, the thinness of cloth has meant that the general focus is on continuous methods rather than discrete methods. [Provot, 1997] is an early authority on continuous collision detection for cloth. It describes two tests which allow the continuous detection of the collision between a vertex and a triangle, and between two edges, which together cover all collisions that can occur. Respectively, one has to test whether a vertex becomes coplanar with the three vertices of the triangle, and, whether the four points which define the two edges become coplanar.

Vertex co-planar with a triangle:

$$(\overrightarrow{\mathbf{AB}}(t) \times \overrightarrow{\mathbf{AC}}(t)) \cdot \overrightarrow{\mathbf{AP}}(t) = 0 \tag{2.9}$$

where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are the three points of a triangle, and where $\mathbf{P}$ is the vertex. Hence, the function $\overrightarrow{\mathbf{AB}}(t)$ is a vector between $\mathbf{A}$ and $\mathbf{B}$. If the movement of vertices over a timestep is linear (an oft-used assumption in physical simulation), this resolves to a cubic equation where one can solve for t. For each real root found, further checks are necessary to determine that there is a vertex-triangle collision. First, that the time is between the current time and the end of the timestep, and second that, at that time, the vertex is not just coplanar, but internal, to the triangle. If multiple roots satisfy this, just the first represents a collision.

Two edges co-planar:

$$(\overrightarrow{\mathbf{AB}}(t) \times \overrightarrow{\mathbf{CD}}(t)) \cdot \overrightarrow{\mathbf{AC}}(t) = 0 \tag{2.10}$$

where **A** and **B** are the beginning and end of one edge, and **C** and **D** the beginning and end of the other edge. Again, finding that the points become coplanar is not enough to determine a collision. For each real root that is between the current time and the end of the timestep, it is necessary to check whether the colliding points between the infinite lines are between the end points of the edges.

Using these two above tests, one can accurately determine whether there is a collision between moving triangles.

Despite the fact that theoretically, any collision can be coped with using the continuous collision detection system above, according to [Bridson et al., 2002], numerical inaccuracy still means that self-intersection can occur. The paper describes a combination of methods which, together, achieve a guarantee of no dynamic self-intersection.

The scheme uses both discrete and continuous detection. For the discrete detection, feature pairs (edge-edge and vertex-face) are tested for proximity, to see whether they are closer than a certain 'thickness' value.

2.3.2 Collision Response

Once collisions have been detected, the effects of the collision have to be reflected in the movement of the cloth. Much like there are various methods for collision detection, there are also many for collision response.

[Provot, 1997] describes how collisions can be dealt with in a mass-spring system, using a basic impulse based response to implement both restitution (impact) and friction for elements that are in contact:

$$\mathbf{v} = \begin{cases} \mathbf{v}_T - k_f|\mathbf{v}_N|\frac{\mathbf{v}_T}{|\mathbf{v}_T||} - k_d\mathbf{v}_N & \text{if } \mathbf{v}_T >= k_f|\mathbf{v}_N| \\ -k_d\mathbf{v}_N & \text{if } \mathbf{v}_T < k_f|\mathbf{v}_N| \end{cases} \tag{2.11}$$

where $\mathbf{v}$ is velocity, $\mathbf{v}_T$ the tangential component of velocity, $\mathbf{v}_N$ the normal component of velocity, k_f the coefficient of friction and k_d the coefficient of restitution. These equations are for a point/particle colliding with a stationary and immovable object. Similar equations apply for moving particles in collision with each other.

However, the paper goes on to explain that using impulses as above will not in fact avoid every case of self penetration. Dealing with multiple simultaneous collisions (or, multiple collisions, over a timestep, with dependencies on each other) is a significant problem in physics simulation. The obvious solution is to perform a round of collision detection to determine where the first impact will occur, moving forward just that far in time, before dealing with that impact before repeating the entire process. This, unfortunately, is impractical for real time simulation as it becomes prohibitively slow when multiple collisions occur in quick succession. [Mirtich, 2000] is a well known strategy for dealing with this problem. It works by simulating the movement of objects that are not in colliding states as normal, only backing up the simulation of an object if it comes in to contact. In this way, it manages to resolve collisions in order, but does not needlessly resimulate.

Unfortunately, as explained in [Bridson et al., 2002], timewarp 'works well except when there are a small number of contact groups which unfortunately is the case for cloth as the entire piece of cloth has every node in contact with every other through the mass-spring network'. This in essence justifies a position that dealing with collisions in contact order is impractical, particularly when performance is an issue.

The solution in [Provot, 1997] is to determine so called 'zones of impact' or 'impact zones' - areas of cloth which will involve multiple collisions over the timestep, and to treat them as rigid. The zones are built up iteratively, with each node/particle in its own zone, with zones successively merged if nodes from separate zones interact with each other. It is intuitive that any node that is part of an impact zone cannot possibly intersect with any other node that is part of the zone because their relative positions will be static over the timestep. It is not a method with a great deal of justification in reality, but it is a method that has seen use since, including in [Bridson et al., 2002], where it is used as part of the scheme which guarantees no dynamic self-intersection.

Indeed, [Bridson et al., 2002] uses a number of steps to produce such a robust simulation. As described previously, they use both discrete and continuous collision detection. The discrete detection is used to determine proximity between elements of cloth, flagging where elements are 'too close', which is taken to mean closer than the thickness of the cloth would permit, and continuous detection is used to check whether actual intersection occurs between elements over a timestep.

To deal with collisions they use a number of stages.

1. A round of discrete detection is used at t, to determine whether any parts of the cloth are within the thickness parameter. Elements which are, and that are approaching eachother, are first dealt with using an impulse which simulates an inelastic collision (the relative velocity is removed). Following this, a spring like replusion is used which pushes the elements apart towards the thickness distance, and a further impulse, along the collision tangent, is used to apply friction.

2. A round of continuous detection is used, to detect collisions between t and $t+\Delta t$ to check for intersections that will occur, and a round of discrete detection is used at $t+\Delta t$ to check whether elements are too close at the end of the timestep. This will pick up both cases where elements are approaching and going to collide, and cases where elements are moving apart. In the case of the former, the collision is coped with as in step (1), where in the latter just the spring repulsion is applied.

3. Step (2) is repeated some number of times

4. After some number of repetitions, if any collisions are still being detected in the continuous detection, then Rigid Impact Zones are used as in [Provot, 1997].

The above scheme guarantees that, if the state of the cloth at t is collision free, the state of the cloth at $t + \Delta t$ will also be collision free. That may sound perfect, but, as discussed in [Baraff et al., 2003], it is not. 'Given the kind of guarantees that Bridson's method [[Bridson et al., 2002]] provides, the ability to untangle cloth geometry may seem unnecessary: if you start Bridson's algorithm with no intersectionts, it will maintain the invariant. A problem, however, is that if outside constraints force cloth to intersect, the method can never recover. In production animation, this happens all the time, when cloth becomes pinched between intersecting character geometries.'

They go on to explain that, while in real life geometry does not intersect, it is common practice for parts of a character's model/skeleton to intersect itself in certain poses. As example, a squatting character will bend his knee such that the backs of lower and upper leg come into contact. In real life, the flesh deforms and rests against itself. In an animation pose, it's likely that the joints simply intersect each other. This poses two issues: how should the cloth behave when it is forced to intersect with objects, and

potentially cloth. Second, when the intersection ends, how can we ensure that the cloth is not tangled. A case of cloth being trapped between joints is shown in Figure 2.8.

[Baraff et al., 2003] describes two methods which work together to solve the above issues. Flypapering is used to ensure good behaviour of cloth in highly constrained situations, and Global Intersection Analysis is used to untangle tangled cloth, if it should arise. The flypapering method works by detecting 'pinched' cloth particles and essentially adhering them to the object/objects that trap them. This method is being used in Figure 2.8.

The Global Intersection Analysis method works by first determining a 'collision curve' on an intersecting piece of cloth. An example is shown between cloth and a sphere in Figure 2.9. In the left side, the collision is shown, and the red curve marks a contour of where the actual intersection takes place. The right image shows the contour mapped to the cloth and sphere individually, with white spots indicating the vertices interior to the contour. Having detected such a curve, and the interior particles, disentanglement can be achieved by applying attraction forces between the white particles shown. Generating the curve/contour, and interior particles, is key.

In any intersection between two triangles, there is a precise line which intersects between them. The three possibilities are shown in Figure 2.10. By combining the line segments from a collision between two meshes (or two parts of the same mesh), a collision curve can be found. Given such a curve, it is necessary to find the vertices that are interior. A flood fill algorithm determines which side of the curve has less vertices in it, and this is named interior. As such, regions of particles which ultimately need to pass back through each other are determined, allowing untangling to occur.

2.4 GPGPU Parallelisation

Despite the length of time for which cloth simulation has been studied, it is still an area where performance is often prohibitive. One way in which performance can undoubtedly be improved in physical simulation is by mapping it to parallel hardware, which is often more suited to the types of computation necessary. Parallel solutions have been sought on multi-core CPUs, and more recently many-core GPUs, particularly with the

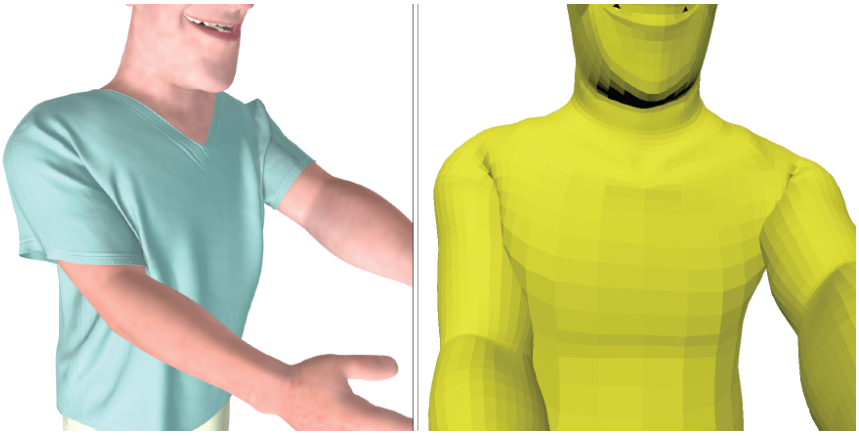


Figure 2.8: The intersection of a posed character, and proper behaviour of cloth in that situation

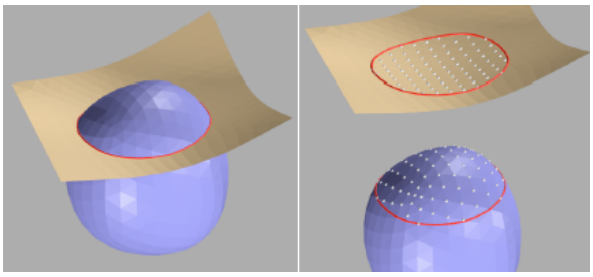


Figure 2.9: The collision curve between a piece of cloth and a sphere

advent of the GPGPU paradigm.

Some research into GPU based simulation specifically sought to use elements of the graphics processing pipeline (such as the rasteriser) for aspects such as collision detection, such as [Baciu and Wong, 2004]. However, there are limits to the potential of image-based methods, so the focus here on is on object-space simulation.

One of the first examples of GPUs being used for cloth simulation was from NVIDIA's Simon Green, who presented a basic cloth simulation which incorporated basic behaviour based on the traditional mass-spring model. Particle data is stored in a floating-point texture. Using a series of fragment shader passes, particle positions are updated based on Verlet integration, before having constraints applied to them (enforcing distance constraints between particles, and collision with sphere and plane).

A slightly more advanced version was presented in 2005 and released as a white paper

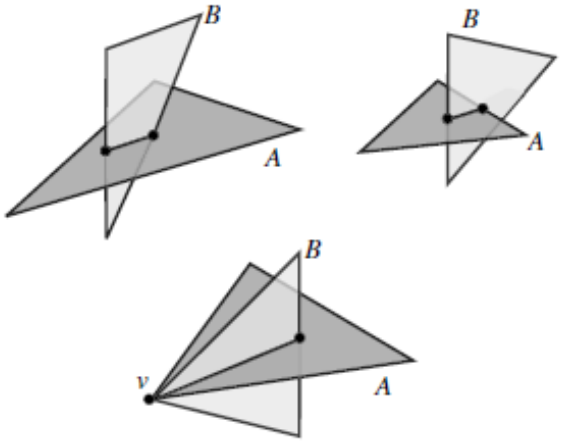


Figure 2.10: The three different lines created in triangle-triangle intersection

in 2007[Zeller, 2005]. The major difference between Zeller's formulation and Green's is that Zeller more smartly handles the spring constraints between particles. For interdependent constraints (eg. two constraints which move the same particle), enforcement must be sequential for the simulation to converge. Hence, the springs are split into groups so that no interdependent springs are processed in the same pass. Zeller simulates the structural and shear springs from [Provot, 1997], and use 8 groupings/passes to simulate them all. Collisions are dealt with by detecting whether a cloth particle is inside one of the collision objects, before moving it out.

Though both of the above systems are efficient simulations, they are far inferior to much earlier sequential methods in terms of the simulation quality. Crucially, neither of them deal with self-collision detection/resolution, a staple of high quality cloth simulation.

More recently, research into parallel cloth simulation has caught up significantly with what can be expected from a sequential program. [Tang et al., 2011a] describes a GPU based pipeline for continuous collision detection with deformable objects, utilising the GPU as a stream processor via the CUDA platform. In the stream processing paradigm, data is represented as streams which are then worked on by kernels. They boast impressive performance with 'inter-object and intra-object computations on models composed of hundreds of thousands of triangles in tens of milliseconds'. The scheme is designed to be flexible enough to take advantage of culling methods such as those described in the collision detection section.

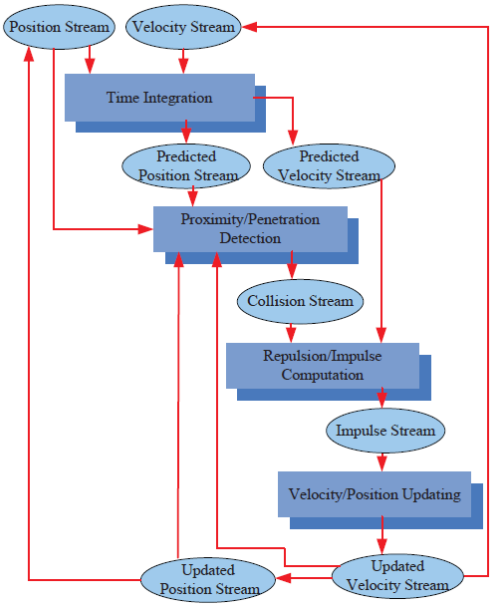


Figure 2.11: The GPU Streaming Pipeline

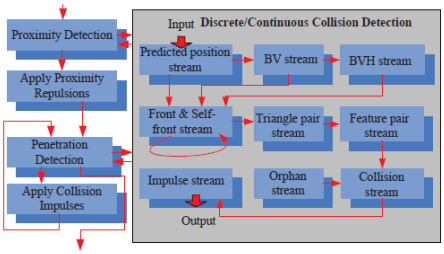


Figure 2.12: A closer look at the collision part of the pipeline

The work has been incorporated into [Tang et al., 2013], which describes a fully GPU-based method for simulating cloth. For the simulation of extremely high resolution cloth, their results show a speedboost of over 2 orders of magnitude over single threaded CPU based simulations, and over 1 order of magnitude over multi-threaded CPU simulations.

Their pipeline has two major stages, time integration (for which they have implemented both explicit RK4, and implicit integration), and collision handling, heavily based on [Bridson et al., 2002]. The pipeline is shown in Figures 2.11 and 2.12, with the second picture giving an in detail look at the collision handling section.

Chapter 3

Experiments & High Level Design

As mentioned in the introduction, the goals of the project were to explore the viability of performing cloth simulation, including the highly complex and costly self-collision handling, in real time. Due to impressive results achieved across research into physically based animation, it was decided the research should be performed utilising the relatively new area of GPGPU.

In particular, three experiments were devised:

1. To benchmark a fully developed GPU version of robust CPU-based standard described in [Bridson et al., 2002].
2. To explore the way that the thickness of the cloth (a margin within which collisions are registered using discrete collision detection) affects the cost of collision detection & handling, based on a hypothesis that higher thickness values will drastically reduce the cost of collision handling.
3. To compare the performance of the standard Bridson model against a combination of discrete collision detection and untangling (e.g. [Baraff et al., 2003]). In particular, a quantitative comparison of costs and a qualitative comparison of appeal/believability.

Four different collision handling methods are necessary to perform the above tests - these have already been summarised in Sections 2.3.1 and 2.3.2.

1. Proximity based discrete collision detection
2. Continuous collision detection
3. Rigid Impact Zones
4. Untangling

It was decided that the design should be modular. The system should work with all four methods, any three, any two, one, or none, without a hitch. As such, the pipeline was designed as illustrated in Figure 3.1. The integration module takes a cloth state, and, based on the internal modelling of the cloth and external forces (such as gravity, and interaction forces directed by the mouse), calculates the new velocity for each point in the mesh. The collision handling methods are then depended on to alter these the candidate velocities to ensure collisions are resolved.

After all collision handling has occurred, the altered candidate velocities are used to update the cloth to its new position, at which point the simulation moves to the next timestep. The order of the collision handling methods is as so because: untangling should occur before any other phase, because elements involved in untangling typically need to be excluded from further handling. The further three phases are ordered as in [Bridson et al., 2002].

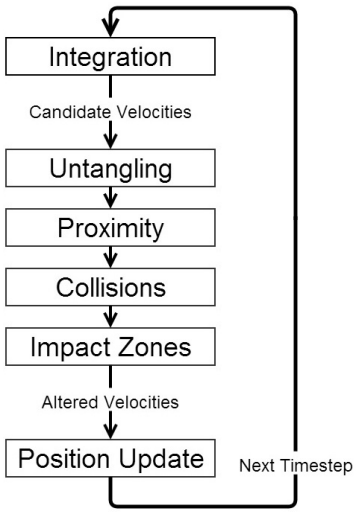


Figure 3.1: The full pipeline

Two real possibilities were available for the cloth model - a mass-spring model derived from [Provot, 1997], or a position based dynamics model from [Müller et al., 2007]. In the end, the mass-spring model was used as it is still the most popular method, and the method that both [Bridson et al., 2002] and [Baraff et al., 2003] are based on.

Various different integrators were explored to see which would best fit the aims of the project. On the low end, the highly efficient explicit euler integrator was considered, but was found to be too unstable for high resolution meshes of cloth with stiff springs. The high end implicit integrators were also considered, such as [Baraff and Witkin, 1998], but these seemed prohibitively expensive, despite the accuracy. Eventually, a happy medium was found in the approximated implicit integrator from [Kang et al., 2000].

The discrete collision detection, continuous collision detection, and rigid impact zones are designed to follow the example set in [Bridson et al., 2002]. For the discrete and continuous detection, mapping to the GPU is relatively simple, and [Tang et al., 2013] discusses GPU based implementations of both.

[Tang et al., 2013] does not implement the rigid impact zones, noting that their scheme cannot be entirely relied on: "For some complex scenarios, repulsion-based and impulse-based collision handling can not resolve all the penetrations. Some more sophiscated methods, such as [...] impact zones can be used. This is a good avenue for future work". Hence, a GPU based version of the process is developed based on the insight that the merging of impact zones can be considered as a standard graph problem.

For the untangling method, again a GPU approach had to be developed anew. The CPU-based approach outlined in [Wicke et al., 2006] was followed. There are a number of steps involved. First, on a discrete basis the triangle-triangle intersections are found in the mesh. These intersections are then grouped up into paths. These paths are used to identify those particles which have crossed to the wrong side of the mesh - and those particles are then pushed back to the right side. The exact process by which this occurs depends on the type of intersection path.

Each of the collision handling methods require testing pairs of triangles, or pairs of triangle features against each other. To ensure that this can be achieved in better than $O(n^2)$ time, a broadphase was needed. [Teschner and Kimmerle, 2005] argues persuasively for the use of Bounding Volume Hierarchies, and this is supplemented

using various optimisations including normal cones [Provot, 1997] and representative triangles [Curtis et al., 2008].

3.1 The CUDA Framework

It is intended that the entire pipeline should be implemented on the GPU with minimal operation on the CPU. The CUDA framework is used for controlling the GPU. A brief discussion of the major CUDA concepts follows.

Using CUDA, functions can be written in C/C++ that run on the GPU. Global functions are GPU functions which can be launched by the CPU, whereas Device functions are GPU functions that are called from the GPU. Three major levels of parallelism are offered: threads, blocks, and streams.

When a global function is called (also known as a kernel launch), it is called using a number of threads and blocks. Each block is made up of a number of threads. The code inside the global function is run by every thread. The code can include the index of the thread and block, and so despite the fact that every thread will run the same code, they can do it on different data by using the thread and block identifiers as indices into arrays. Obviously, using conditional programming, different threads can also end up running different instructions.

Parallelism can be achieved by using just blocks of just one thread each, but there are significant advantages to using multiple threads per block. Up to 32 threads can be run simultaneously (as fast as a single thread) in a warp. Also, synchronisation can be performed between threads in the same block (ensuring that all threads do X before moving onto Y). Unfortunately, there is a limit to the number of threads per block (512 on older GPUs, and 1024 on more modern ones), and this means that massive parallelism can only be achieved using both threads and blocks.

The final layer of parallelism is provided by streams, which allow multiple kernels to be run simultaneously. Kernels launched into the same stream will run sequentially, but two kernels launched into different streams will run in parallel. This is useful for performing different tasks, with no dependency, at the same time.

Getting good performance on the GPU is all about ensuring the GPU is being occupied fully at all times. It needs to be constantly processing lots of data with little downtime.

Two other important factors for efficient GPU programming are data and execution coherency. For good data coherency, nearby threads should be working on nearby data. If, for example, a kernel is launched to work on an array, the first thread should deal with array[0], the second array[1], and so forth.

For good execution coherency, nearby threads should be working on the same instructions. If conditional instructions are to be used, ideally nearby threads will take the same path. As an example, if there are 512 threads executing a kernel with an if-else statement that half will evaluate as true, and half false, it is far better to use if(threadId.x $<$ 256), than if(threadId.x % 2 == 0).

A common problem that arises during any parallel programming is the race condition. A race condition occurs when multiple threads simultaneously try and modify particular values, which can lead to incorrect results. CUDA offers a simple solution for this, called atomic instructions. Atomic instructions implement read-modify-write operations in a way which is guaranteed not to be interfered with by other threads. Multiple threads which use atomic operations on a single value will end up being run serially, so they should only be used when simultaneous operation on a single value is rare.

It is hoped that this brief discussion will be enough to ground the future descriptions of the project's implementation. Many further resources are available on NVIDIA's website.

Chapter 4

Implementation & Low Level Design

4.1 Pipeline

As discussed in the previous chapter, the cloth simulation was designed as a linear modular pipeline. At the beginning of each timestep, the state of the cloth is advanced through time using the integrator. To be more exact, the integrator calculates the next velocity for each particle in the cloth mesh. This velocity is then used at the end of the pipeline to calculate a new position.

In between the integration step and the position update step, the velocity of each particle may be altered by one of the various collision handling methods. The pipeline is designed so that collision handling methods can be incorporated easily, slotted in and out as needed. Four collision handling methods have been implemented, and each takes advantage of broadphase acceleration via a Bounding Volume Hierarchy.

The remainder of this chapter will describe the implementation details of the cloth model, integration, and collision handling methods.

4.2 Libraries Used

A number of different tools where used for the implementation. Alongside CUDA, the Thrust library, which is supplied as part of the CUDA SDK, which provides GPU-based implementations of commonly used parallel functions such as reductions, prefix sums, sorting and so on. The CULA Dense library was also used, which provides GPU-based linear algebra functions (such as matrix multiplication and linear system solving).

4.3 Cloth Model

The standard 'Provot' cloth model is used, covered in Section 2.1, where a rectangular mesh of particles is connected by a network of springs. Data is stored that pertains to the particles, springs, and triangles that make up the cloth. Ultimately, this forms three data structures.

The Particle System

- Mass (Float)
- Position (3D Vector)
- Velocity (3D Vector)
- Fixed (Boolean)
- Force (3D Vector)
- Attached Forces (3D Vector)
- # of Springs (Integer)
- # of Triangles (Integer)
- Triangle Indices (Up to 6 integer indices into triangle system)
- Normal (3D Vector)
- Impulses (3D Vector)
- # of Impulses (Integer)
- Changed (boolean)

Most of the above items should be self explanatory. The fixed boolean indicates whether the particle is immovable, which can be used to suspend the cloth in midair, for example. The attached forces stores the sum of forces applied to the particles attached to the

current particle, and the number of springs is also stored. These two values are stored for the purpose of integration. For collision detection and rendering, details about the triangles each particle is a part of are stored. For collision handling, an accumulator stores the total impulse to be applied, as well as the number of impulses to be applied. Finally, the changed boolean value stores whether the particle has been affected by the previous round of collision handling, which is used for a simple acceleration in the broadphase.

The Triangle System

- Normal (3D Vector)
- Particle Indices (Integer indices into particle system)
- Representative Particles (3 Booleans)
- Representative Edges (3 Booleans)

For each triangle is stored the normal, and three indices which refer to the three particles which make up the triangles. Further to this, three boolean values are stored, one for each particle, which refers to whether the triangle is its representative (in short, each particle is represented by exactly one triangle, this is used to accelerate collision handling. To implement representative triangles from [Curtis et al., 2008], covered in Section 2.3.1.

The Spring System

- Index to first particle (Integer)
- Index to second particle (Integer)
- Rest length of spring (Float)
- Spring coefficient (Float)
- Damping coefficient (Float)

Finally, the above details are stored for each spring. The indices refer to the first and second particle that are attached by the spring.

Each of the three systems above are represented using a Structure of Arrays. Hence, the particle system is not stored as an array of particles, but a set of arrays for particle masses, positions, velocities, and so on. This offers a considerable performance boost.

4.3.1 Cloth Rendering

Rendering cloth was not one of the primary objectives of the project, but as it is not entirely trivial, here follows the method used. To render the cloth using a normal lighting technique, each particle needs an associated normal. The normals are acquired via a two step process. First, the normals of the triangles are calculated according to Equation 4.1. Second, the normals for each particle is calculated as the average of the normals of the triangles the particle is constituent of, according to Equation 4.2.

$$\hat{\mathbf{n}}_t = \frac{(\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)}{|(\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)|} \tag{4.1}$$

$$\hat{\mathbf{n}}_p = \frac{\sum_{i=0}^{m} \mathbf{n}_i}{|\sum_{i=0}^{m} \mathbf{n}_i|} \tag{4.2}$$

Here, p_i refers to the position of the ith vertex of a triangle, and n_i refers to the normal of the ith triangle a particle is a part of.

However, this process only deals with rendering a single side of the cloth, which can cause some odd effects with lighting. Despite the fact that the cloth model doesn't consider the cloth as having sides, it does have to be rendered as if it is two sided and somewhat thick. Hence, the cloth is rendered twice, once using the normals as calculated above and then again using the negated version of those normals. Additionally, to achieve thickness, the vertex shader offsets the particle positions a small distance along its normal (once positive, once negative). Unfortunately, around the edge of the cloth, a gap can be visible using the above formulation. To fix this, the particles on the edge of the cloth mesh are offset by a much smaller amount.

This extra process gives appealing looking cloth with a tangible weight to it.

4.4 Integration

The integration scheme used to advance the cloth model through time is an approximation of the implicit euler method, from [Kang et al., 2000]. In the previous section,

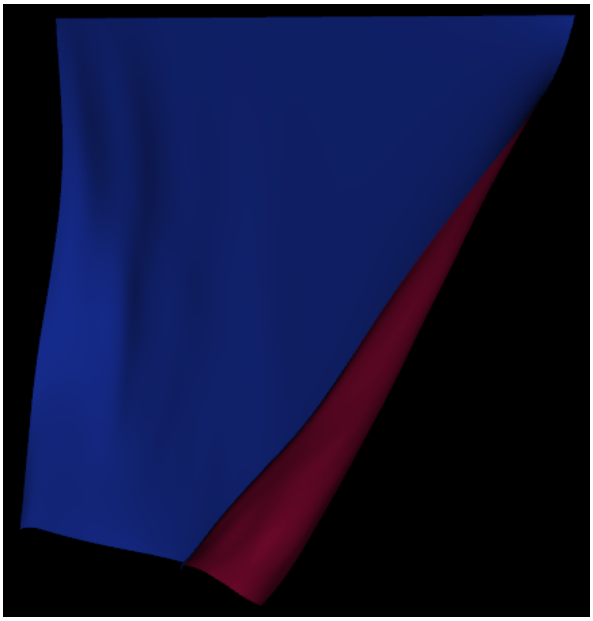


Figure 4.1: Two-sided thick cloth using two render passes

the cloth model, and all of the data stored about the cloth, was discussed in detail. In this section, a thorough description of how the integration is implemented follows.

[Kang et al., 2000] outlines a series of formulae which can be used to calculate the next state for each particle:

$$\mathbf{F}_{si}^{t} = \sum\nolimits_{\forall j|(i,j)\in S} k_{ij}(|\mathbf{x}_j - \mathbf{x}_i| - l_{ij})\frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|} \tag{4.3}$$

$$\mathbf{F}_{vi}^{t} = \sum\nolimits_{\forall j|(i,j)\in S} k_{ij}h(\mathbf{v}_j^t - \mathbf{v}_i^t) \tag{4.4}$$

$$\mathbf{F}_{ei}^{t} = \mathbf{g}m_i \tag{4.5}$$

$$\mathbf{F}_{i}^{t} = \mathbf{F}_{si}^{t} + \mathbf{F}_{vi}^{t} + \mathbf{F}_{ei}^{t} \tag{4.6}$$

The above four equations calculate the force that is being applied to a particle i, $\mathbf{F}_i^t$.

S is the set of all springs. A spring (i, j) is a spring which attaches particle i to j. The position of particle i is referred to as $\mathbf{x}_i$, and its velocity as $\mathbf{v}_i$. k_{ij} is the spring coefficient of spring i, j. Finally, h is the length of the timestep (typically 1/60 s, and $\mathbf{g}$ is the acceleration due to gravity $(0, -9.81, 0)^T$).

The calculations referred to by Equations 4.5 and 4.6 are simple, but some thought has to go into accumulating the sums in Equations 4.3 and 4.4 in parallel. There are two main possibilities for this. The first is, for each particle, to iterate through the attached particles and accumulate the values. However, this particle centric system ends up doing double the necessary work as the force applied by a spring to a particle is equal and opposite to the force it applies on the other particle.

Better is a spring centric system, where the calculations are performed for each spring and the values then distributed to the particles. In parallel though, this provides a race condition, as many springs will be adding to the same particle's force accumulator. The solution for this was inspired by [Zeller, 2005]. The idea is to separate the springs into groups such that no two springs in the same group affect a single particle.

Splitting the springs into these groupings is fairly simple. For example, the horizontal structural springs connect particles to those directly to the left, and right. They can be split into two groups which satisfy the above principle as so: The first group includes springs (0,1), (2,3), (4,5), and so on. The second group includes springs (1,2), (3,4), (5,6), and so on. The separation is similar for the vertical structural springs, shear springs, and bending springs. The groupings for structural and shear springs are shown in Figure 4.2.

Then, simply, each group of springs is dealt with in turn, with a synchronisation barrier between each group. This ensures that the computation for the first group is fully executed before the computation for the second, and the race condition is avoided. This completes the process of calculating $\mathbf{F}_t^i$.

The approximate integrator proceeds as follows:

$$\Delta \mathbf{v}_i^{t+h} = \frac{\mathbf{F}_i^t h + h^2 k \sum_{\forall j | (i,j) \in S} \mathbf{F}_j^t h / (m_j + h^2 k n_j)}{m_i + h^2 k n_i} \tag{4.7}$$

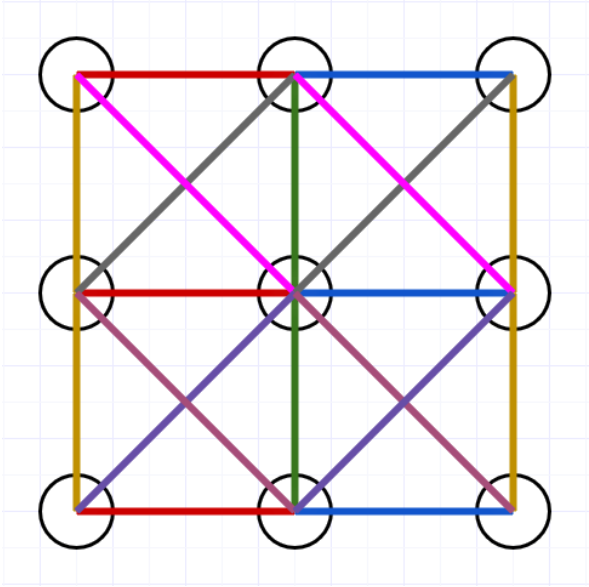


Figure 4.2: Spring Groups

$$\mathbf{v}_i^{t+h} = \mathbf{v}_i^t + \Delta\mathbf{v}_i^{t+h} \tag{4.8}$$

$$\mathbf{x}_i^{t+h} = \mathbf{x}_i^t + \mathbf{v}_i^{t+h} \tag{4.9}$$

There is little to say about the final two equations here - these are simple equations to calculate the next position and velocity. The first equation however is the core of why the approximate integrator is so stable. The change in velocity for particle i is effected not only by the force applied to particle i, but also by the forces applied to all of the particles particle i is attached to.

To help, an intermediate variable is accumulated - part of Equation 4.7.

$$\mathbf{F}_{ai}^t = \sum\nolimits_{\forall j|(i,j)\in S} \mathbf{F}_j^t h/(m_j + h^2 k n_j) \tag{4.10}$$

After $\mathbf{F}^t$ has been calculated for every particle, $\mathbf{F}_a^t$ is calculated. For every spring (i, j), $\mathbf{F}_i^t$ is added to $\mathbf{F}_{aj}^t$, and $\mathbf{F}_j^t$ is added to $\mathbf{F}_{ai}^t$. Once again, a race condition will occur if

the springs are dealt with in parallel, and hence the 12 spring groupings are used.

Once $\mathbf{F}_a^t$ has been calculated, Equations 4.7 through 4.9 are used to update the state of the cloth. As explained in the introduction to the chapter, the position update actually occurs at the end of the pipeline, after any collision handling has occurred. As such, the velocity calculated using Equation 4.7 is essentially the *candidate velocity*, which is then checked to see whether it is valid (i.e. does not cause collisions), and changed if it is not.

4.5 Broadphase Detection

All of the different narrow phase collision detection/handling methods boil down to performing tests between pairs of triangles, or pairs of features (vertex-face or edge-edge). Naively testing all potential pairings is prohibitively expensive, so a Bounding Volume Hierarchy is used to accelerate detection. The Bounding Volume Hierarchy is used to efficiently cull blatant cases of non-collision, leaving a small set of cases to test more precisely.

The Bounding Volume Hierarchy implemented is designed to work with each of the different collision detection methods. The hierarchy is built during initialisation, and refit as needed, typically with each round of collision detection. The Bounding Volume Hierarchy is then traversed to detect overlapping leaf nodes.

4.5.1 Hierarchy Construction

The cloth exists in a 3D world, but it is at times useful to consider it two dimensional, and it is this two dimensional version of the cloth which is used to build the tree hierarchy. The coordinates of each particle then run from $(0, 0)$ to $(N-1, N-1)$, where N refers to the width/height of the mesh (and thus N^2 is the number of particles in the mesh).

The leaves of the BVH are constructed from pairs of adjacent triangles. Figure 4.3 shows how leaves and triangles are numbered. It shows the first and second triangles

(in pink and blue respectively) which together would make up the first leaf, and the second leaf (shown in yellow). The coordinates of the leaves then run from $(0, 0)$ to $(N - 2, N - 2)$.

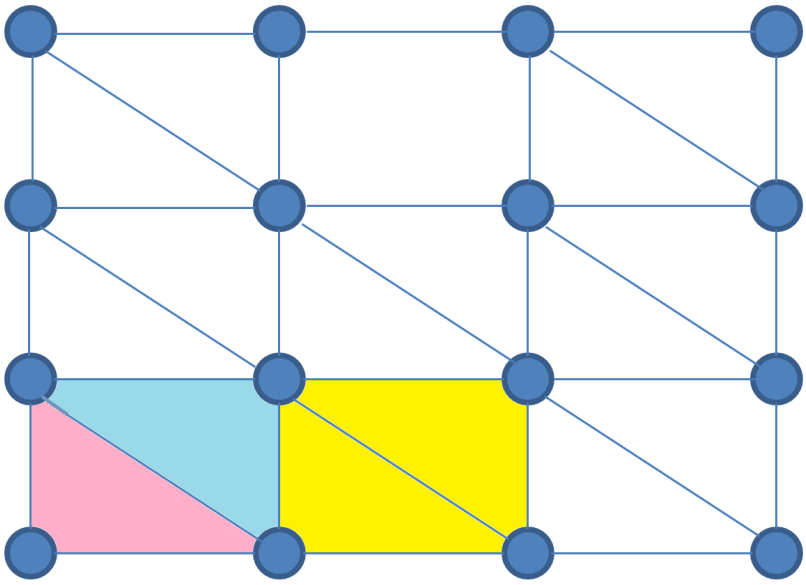


Figure 4.3: Triangles and leaves

Starting at the root, the leaves are then classified recursively by splitting the node in half, first by the x-axis, then by the y-axis. This is performed on the CPU, and constructs a hierarchy with a useful property: every node is made up of a contiguous surface. An example hierarchy is shown in Figure 4.4. The first seven images show different levels of the hierarchy: the first three and four last. Red AABBs indicate nodes, and blue indicate leaves. The final two images shows how the hierarchy works in 3D. The first shows the state of the cloth, and the second shows the hierarchy on top of it at the third lowest depth.

As the hierarchy is built at initialisation, it is constructed on the CPU before being moved across to the GPU. On the CPU, the data is stored using a standard tree representation, where each node points to its children. This makes no guarantees about the position of nodes in memory which is disastrous for data coherency. The GPU tree is stored in a contiguous block of memory, within which each depth of the tree is stored contiguously.

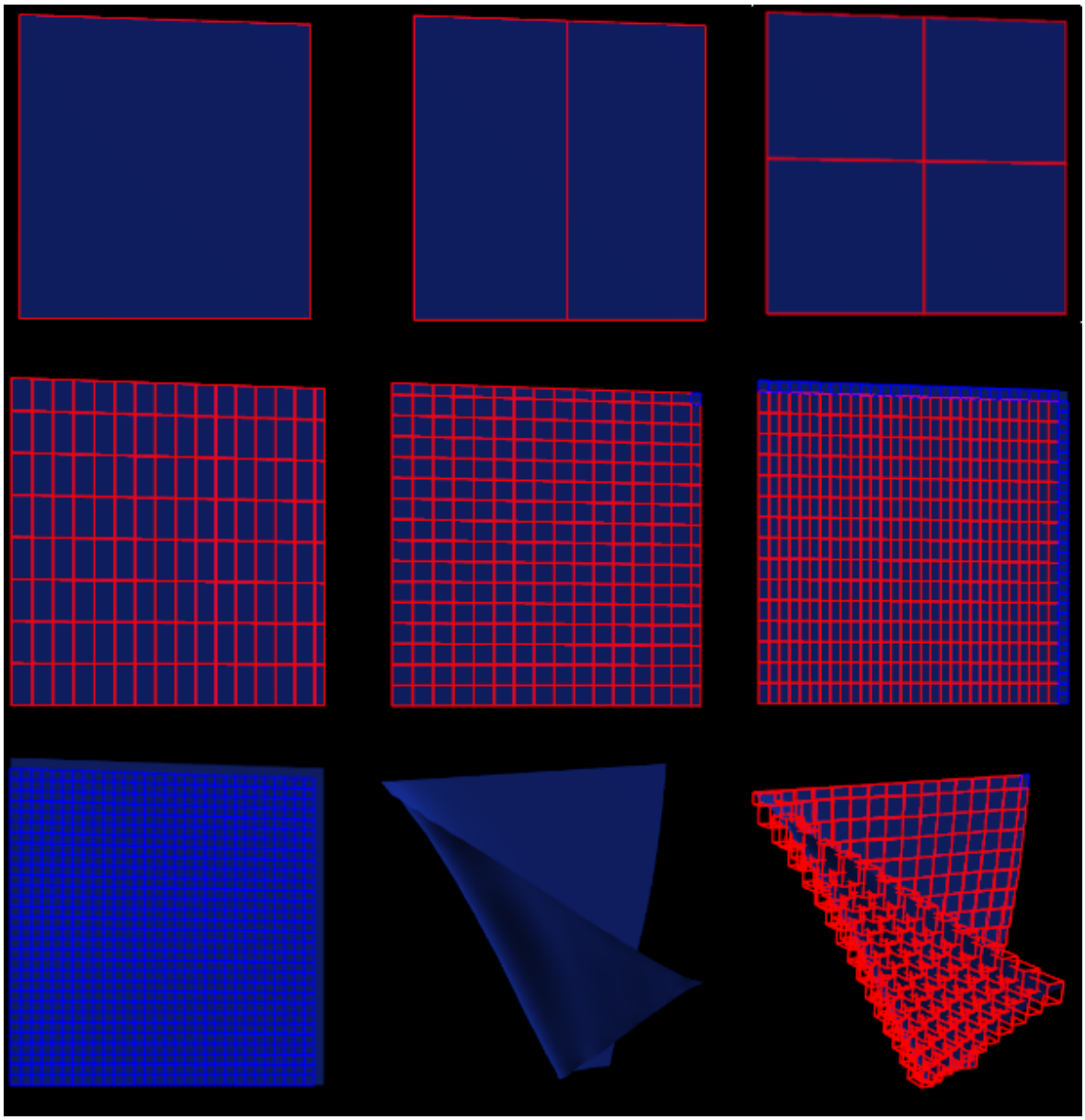


Figure 4.4: The BVH created using the 2D construction

4.5.2 Refitting the Hierarchy

The hierarchy is refitted on the GPU using a bottom-up method. Starting at the maximum depth, each node is refit before moving up to the next depth. The refitting method differs somewhat dependent on the collision handling method, but in every case involve generating new AABBs for each node. For the discrete collision detection, the AABBs are inflated by the thickness amount. For the continuous collision detection (used for the impact zones as well), the AABBs contain the swept triangle, or, the triangle both at the start of the frame and at the end. For untangling, where triangle-triangle intersections are sought, the AABBs are standard.

In sequential code, refitting can be achieved using a head-recursive function. It is trickier to map it to the GPU.

It is mandatory for obvious reasons that the children of a node must be refit prior to the node itself. To parallelise the BVH refit using CUDAs two level parallelism (threads & blocks), the tree is split into a top section and a bottom section as shown in Figure 4.5. In the small example, the tree has been split into a top section (orange) and a bottom section. The bottom section is then split into four subtrees (blue, pink, green, and purple).

The top section will be dealt with by a single block of threads. Starting at the bottom of the section, each node is refit by a thread, before synchronisation, which ensures that no thread starts dealing with depth x before depth $x+1$ is fully computed. After the threads are synchronised, the computation moves up a level, and repeats. To aid this, an array stores indices into the BVH for the first node at each depth. This is summarised in pseudocode in Algorithm 2.

Algorithm 2 Refitting the top of the tree from depth T upwards

```
1: procedure RefitTop
2:     for Depth T to Depth 0 do
3:         Offset = First node at this depth
4:         Refit node[offset + threadid]
5:         Synchronise Threads
```

Refitting the bottom section of the tree is more complex, as the width of the tree (maximum number of nodes at any depth) may exceed CUDAs maximum number of

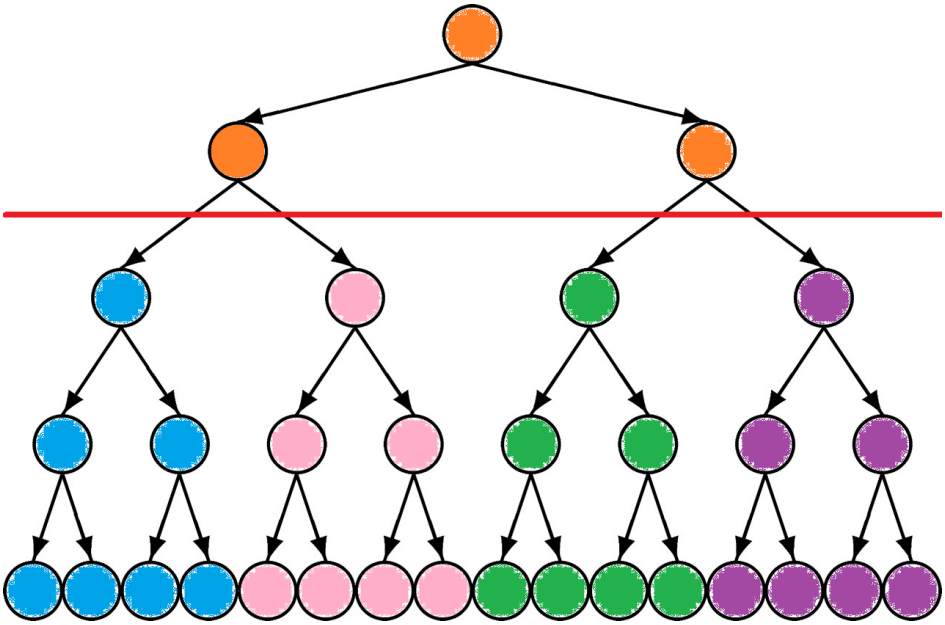


Figure 4.5: The two levels of the tree

threads per block. Thus, the parallelisation must be split over multiple blocks. CUDA does not support synchronisation between blocks - and what this entails is that a node must be dealt with by the same block as its children. Hence, the bottom section of the tree is split into several subtrees (enough that no subtree's width exceeds the maximum number of threads per block).

For each subtree the procedure is very much similar to that described for the top of the tree. For refitting the top of the tree, an array is used which stores the index of the first node at each depth of the tree. For refitting the bottom, an array is used which stores the index of the first node at each depth for each subtree.

Algorithm 3 Refitting the bottom of the tree

```
1: procedure RefitTop
2:     Subtree = blockid
3:     for Depth D to Depth T + 1 do
4:       Offset = First node at this depth of subtree
5:       Refit node[offset + threadid]
6:       Synchronise Threads
```

The algorithm ensures that, within a given subtree, computation on depth $x + 1$ will

complete prior to computation on depth x. No guarantees are made about execution between subtrees, but as the subtrees are independent of each other, this is not an issue.

These two kernels are run one after the other (bottom first, top second), and accomplish a full parallel refit of the tree. It effectively accomplishes the two key factors for good CUDA performance: good data coherency and good execution coherency. Similar threads will indeed be dealing with similar data, and be executing the same code.

4.5.3 Traversing the Hierarchy

Given a refitted BVH, it remains to traverse the tree to determine all pairs of overlapping leaves. For this, a NVIDIA developer blogpost [Karras, 2012] was referred to.

On the CPU, the standard way to traverse a BVH to search for self-intersections is described in Section 2.3.1 and Algorithm 1 in what is termed 'simultaneous traversal'. However, as explained in [Karras, 2012], simultaneous traversal doesn't map especially well to the GPU's massive parallelism because it has high data divergence.

A simpler idea, 'independent traversal', which would be far slower on the CPU, tends to be faster on the GPU. In independent traversal, the entire BVH is checked against each leaf. In parallel, each leaf is dealt with by a single thread and no thread cooperation is necessary.

The testing is performed on a depth first basis. This could be implemented using recursive function calls, but this can cause stack overflow (the stack for function calls is smaller than that on a CPU) and high execution divergence. Instead, the stack is managed manually leading to an efficient BVH traversal scheme that should minimise both data and execution divergence.

At each stage of the algorithm, a test is performed between two nodes to determine whether they overlap. This overlap test is spplemented with an extra test based on the normal cone optimisation described in Section 2.3.1. The test cheaply determines whether self-collision is possible in a contiguous surface. Due to the construction of

the hierarchy, every node is guaranteed to be a contiguous surface. As such, the test is made up of two parts: is the leaf node part of the node its being tested against, and, if so, is the normal cone angle lower than π.

4.6 Discrete Proximity Detection & Handling

The first of the four collision handling methods is based on proximity. Any vertex-face or edge-edge pair that are closer than a thickness value are pushed apart using impulses.

The broadphase returns a list of pairs referring to two overlapping leaf nodes. A thread is used to deal with each pair. As explained in Section 4.5.1, each leaf is made up of two triangles, and hence each pair defines four triangle-triangle pairs which have to be tested. Each triangle-triangle pair then defines six vertex-face pairs and nine edge-edge pairs that have to be tested.

To further cull these tests, the notion of representative triangles covered in Section 2.3.1 is used. The boolean variables stored per triangle (covered in Section 4.3 indicate whether or not a particular triangle represents a feature. This ensures that a particular vertex will only be tested against a particular face once.

Both the edge-edge and vertex-face proximity tests are described in [Bridson et al., 2002]. The vertex-face test boils down to two phases: first, the vertex is projected onto the plane which contains the face. The projection is then tested to see whether it lies inside the triangle. If the projection is inside the triangle and if the distance between the projection of the vertex and vertex is less than the thickness, a collision is registered. For the edge-edge test, the closest points between the two infinite lines the edges are part of are found. The points are then clamped to the edges, and tested to see whether they are closer than the thickness value.

Two impulses are then applied for every such collision. The first is an inelastic impulse - it stops the features from approaching each other by removing their relative velocity along the collision normal (which is the vector between the projection and vertex in a vertex-face collision and the vector between the closest points in an edge-

edge collision). The second impulse attempts to push the features apart towards the thickness value. Details on these impulses and how they are calculated can be found in [Bridson et al., 2002].

The impulses are atomically added to the particle's impulse vector, and each time this happens the particles number of impulses is incremented also. After the impulses have been calculated, each particle 'applies' its impulses, by adding the impulse divided by the number of impulses, to its velocity.

The division is necessary because the impulses are calculated in parallel. Imagine a case where a vertex is too close to two adjacent faces. Impulses are applied to stop the approach exactly. However, the two impulses applied together will bounce the vertex away elastically. The division prevents this from occurring, albeit slowing the speed of convergence.

4.7 Continuous Collision Detection & Handling

The second collision handling method aims to detect future collisions, and prevent them from occurring. The tests determine whether, given the current velocity of the particles, any two feature pairs (vertex-face or edge-edge) will intersect over the timestep.

As with the discrete handling, the broadphase returns lists of leaf pairs which define four triangle-triangle tests which in turn define 6 vertex-face tests and 9 edge-edge tests, which are once again culled using representative triangles. The continuous tests are considerably more involved than the discrete proximity tests from the previous section. Indeed, each requires solving for the roots of a cubic equation, to test for the point at which four points (either the end points of the two edges or the vertex and the three points of the triangle) become coplanar.

The roots of the cubic equation then have to be checked to see whether there is indeed intersection. This is identical to the proximity testing that is used for the discrete proximity detection, except that a small epsilon value is used, not a thickness value. In the discrete test, the vector between the colliding points is used as the collision normal. For the continuous test, the colliding points are in the same place, and hence this vector

cannot be used. Instead, the normal of the triangle (for a vertex-face collision) or cross product of edges (for an edge-edge collision) is used.

Also, because the distance between the two colliding points is likely very small, the collision normal is instead calculated as the cross product of the two edges, and the normal of the triangle.

Not all of the roots need to be checked. Roots outside of 0 and h are scrapped, and the remaining roots are tested in ascending order. Roots need not be checked if an earlier root is a collision. As mentioned in [Bridson et al., 2002], checking for proximity at the end of the timestep (even if it is not a root) can aid robustness.

One of the major issues that exists with this testing method is determining the roots of the cubic equation. Cubic equations can be solved using the algebraic cubic formulae, but this is prone to various robustness issues. The *Newton-Raphson* method based solver from the Self-CCD library was ported to CUDA compatible code for this purpose.

Given a detected collision, an impulse is applied along the collision normal that removes the relative velocity, identical to the first impulse discussed in the previous section. This should ensure that the collision no longer occurs over the course of the timestep, but, the effect can be cancelled out by a different impulse applied to the same particles. Also, there is no guarantee that an impulse applied to stop one collision occurring over the timestep won't cause a different collision to occur over the timestep.

A typical implementation of the continuous collision detection routine will use some number of rounds of continuous collision detection to try and deal with all of the collisions. Each round of continuous collision detection demands another round of broadphase detection. An obvious but significant optimisation is possible here. A feature's movement is altered if it is calculated that it will collide with any other feature. It is obvious that two unchanged features that do not collide when testing for round i will not collide in round $i + 1$. Hence, the leaves of the BVH used for the broadphase detection are supplemented with a boolean value that indicates whether or not any of the particles in the leaf (four particles in each case) have changed. These values are then propagated up the tree using a logical or. Then, when traversing the tree, any pair of nodes which have *FALSE* values for their **Changed** variable do not involve any collisions. This significantly reduces the number of unnecessary tests in

subsequent rounds of continuous collision detection.

4.8 Impact Zones

The two collision handling tools discussed so far represent a cheap way to deal with many collisions, and an expensive way to deal with all collisions. The third collision handling method provides a cheap way to deal with all collisions at the cost of realism and, potentially, believability).

As discussed in more detail in Section 2.3.2, an impact zone is a portion of the cloth (a set of particles) which are estimated as being rigid. The linear and angular momentum of the particles are, as with a rigidbody, uniform.

Impact zones are formed of groups of particles that are involved in dependent collisions. In [Bridson et al., 2002], the approach to calculate which particles are in which zones is as follows: Each particle starts in its own list - its own impact zone. Whenever a collision occurs, the lists containing the four involved nodes are merged. The impact zones are grown until the cloth is collision free.

To achieve a similar effect on the GPU, a round of continuous collision detection finds the collisions in the mesh that will occur over the timestep. The problem can then be thought of as a connected component search in graph theory. A connected component is a subgraph in which every vertex is connected to every other vertex, but no vertex is connected to a vertex outside of the subgraph. Each particle is represented by a vertex, and an edge exists between two vertices if the vertices are involved in the same collision.

To proceed, each particle needs to know which impact zone it is in. Two particles will have the same identifier if and only if they are in the same zone. To achieve this, the GPU based algorithm from [Soman et al., 2010] was implemented, which finds the connected components in a graph.

Having found the impact zones, the center of mass, average momentum, angular momentum and inertia tensor is calculated. These are then used to calculate the velocity

for each particle in an Impact Zone. The equations to calculate these are found in [Bridson et al., 2002].

4.9 Untangling

Instead of preventing collisions, as with the previous three methods, an untangling method attempts to reverse intersections that have already occurred.

The GPU implementation here follows the CPU-based design from [Wicke et al., 2006]. The first part of the process is to detect the triangle-triangle intersections in the mesh. Once again, the BVH is traversed to find overlapping leaf nodes. The leaf pairs each define four triangle-triangle tests. The triangle-triangle tests were implemented based upon [Möller, 1997]. The intersection between two triangles is either a point, or a line, and the intersection method returns these for each intersection.

The intersections need to be grouped together into what [Baraff et al., 2003] terms an intersection path. Similarly to how Impact Zones are merged (refer to the previous section), triangle-triangle intersections are merged into the larger intersection paths. In this case, the vertex in the connected component search is a triangle-triangle intersection, and the edges are connections between those intersections. Those connections are found by detecting any 'collisions' between the intersections (those intersections which start/end in the same place). A naive $O(n^2)$ method is used to find these collisions (which was offset by the cheapness of the test) but a broadphase acceleration method could have been used here.

In Section 4.5.1, the idea that the cloth could be considered, at once, in 2D and 3D was discussed. The same concept is necessary for classifying collision paths and untangling cloth. An intersection path in 3D is, in 2D, a pair of intersection paths. Each triangle-triangle intersection (a line) is converted from 3D to 2D as follows:

A 3D triangle-triangle intersection is defined by a start point, an end point, and the two involved triangles. The barycentric coordinates of the start/end points are found for each triangle. The barycentric coordinates are then used to determine, where in the 2D versions of the triangles, the intersection starts and ends.

At this point, for each 3D intersection path, we have two 2D intersection paths. The 2D intersection paths have to be classified. A full classification of all the possible types of intersection path is in [Wicke et al., 2006]. Four different path types are dealt with: closed, inside-inside (II), boundary-inside (BI) and boundary-boundary (BB). A closed path is one that is a full loop. A path is inside-inside if neither of its ends finish on a boundary, boundary-inside if one does, and boundary-boundary if neither does.

For each tangle, we have a 3D intersection path or, its equivalent pair of 2D intersection paths. Handling the tangle is dependent on the types of the two 2D paths. Various pairings are possible (and others are impossible - a closed path will always pair with another closed path). Three examples are closed-closed, BB-II, and BI-BI.

For all but the BI-BI pairing, classifying which particles are internal to a path is necessary. In the case of [Baraff et al., 2003] and [Wicke et al., 2006], this is achieved using a floodfill algorithm. The floodfill algorithm however doesn't seem to map especially well to a massively parallel architecture. Instead, a scanline style algorithm was developed, which, unsurprisingly, as a major part of the standard GPU pipeline (it is used for filling the pixels of a polygon) maps very well to the GPU.

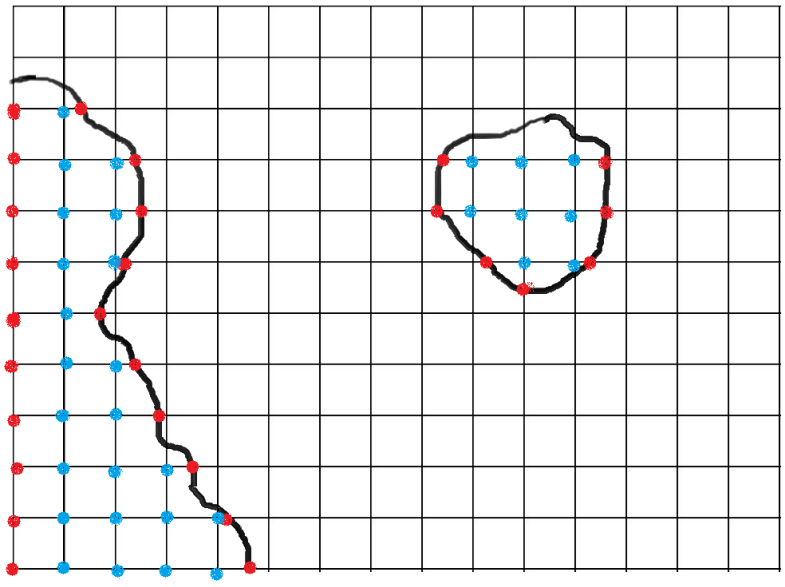


Figure 4.6: The 'scanline' algorithm

Figure 4.6 illustrates the idea behind the scanline algorithm with a BB path and a

closed path. The grid represents the cloth mesh, with the crossover points in the grid representing particles. The red points indicate where the intersection paths cross the horizontal lines, and the blue every particle between two red points.

The first part of the scheme designates the red points. One thread per intersection is used to check whether the intersection crosses any horizontal lines in the 2D mesh. If they do, the crossing point (along the x-axis) is stored. For BB paths, the cloth boundary is considered part of the path for this process.

The second part of the scheme uses one thread per horizontal line. Every particle between two red points (inclusive) is classified as being internal to the path. The scanline kernel can be used for each intersection path in parallel, using CUDA's streams.

Having classified the particles, the next objective is to pull the particles on the wrong side back through the cloth. Following [Wicke et al., 2006], each wrong-side particle is attracted towards another particle. This mapping of particles to particles is achieved by function fitting.

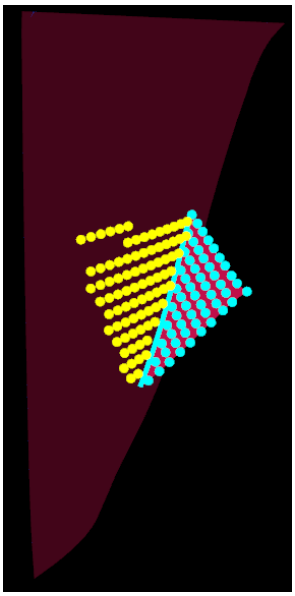


Figure 4.7: The blue (wrong-side) particles of a BB-II intersection map to the yellow particles

Implementation of untangling for all path types was not achieved. Only untangling the BB-II pairing was proceeded with past the particle classification stage. A function

is created which maps the BB 2D intersection path onto the II 2D intersection path (as these are both known quantities) using function fitting (the process is summarised in [Wicke et al., 2006]. The wrong-side particles are then mapped, using the fitted function, to the points on the cloth mesh they should be attracted towards. The CULA Dense Library, which provides CUDA based linear algebra functions, was used to fit the function by solving a linear system. An example of the mapping between the particles is shown in Figure 4.7.

Attractive impulses are then used to pull the points together. This effectively copes with the common BB-II tangle.

To summarise the approach:

1. Triangle-triangle intersections are found in the cloth mesh
2. The intersections are grouped into paths using a connected component search
3. A pair of 2D Intersection Paths is created from each 3D Intersection Path
4. The 2D intersection paths are classified as BB, BI, II, or Closed
5. Particles are classified as internal/external for partitioning (BB/Closed) paths
6. Functions are fitted which map a 2D path to its partner
7. The function is then used to map internal particles to their targets
8. Impulses are applied which attract internal particles to their targets

4.9.1 Finishing the Implementation

The primary contribution of this work (as opposed to [Baraff et al., 2003] - which coped with closed-closed paths - and [Wicke et al., 2006] which coped with all paths) is the parallelisation of the untangling.

The implementation only deals with BB-II paths currently, but it would not be a great deal of work to generalise it to the other path types. The single exception is the BI-BI path, which doesn't work on the basis of inside/outside classification for particles.

As well as dealing with the rest of the path types, the implementation would have to be furthered if it was to slot into place alongside the other collision handling methods (a sheer necessity for the scheme to have any practical use). In particular, particles that

are involved in untangling need to be unperturbed by the collision handling methods: the collision handling must not prevent the cloth from passing back through itself.

Each internal particle would thus need to be removed from the collision handling. This could be achieved with a boolean variable indicating whether or not it was involved in the untangling process.

Chapter 5

Results

5.1 Experiment 1

The first experiment was to benchmark a fully developed GPU version of the robust standard from [Bridson et al., 2002]. Measurements were taken for the individual parts of the pipeline - integration, proximity based handling, continuous collision handling, and impact zone based handling - over a series of scenarios.

In each scenario, a piece of cloth with 2000 triangles was used, modelled with a thickness of 7mm. For each of the benchmarks listed above, the average, maximum, and standard deviation were measured. The times stated are, in every case, in milliseconds. A limit of five rounds of continuous collision detection is used.

In the first scenario, the cloth starts in a rest state flat on the ground. A corner is then picked up, and pulled across so that the cloth is folded along the diagonal, where it again rests, with half the cloth on top of the other half.

The second scenario picks off where the first ends. The corner which was folded on top is picked up again, and flicked back towards its original position, putting the cloth back into a flat state.

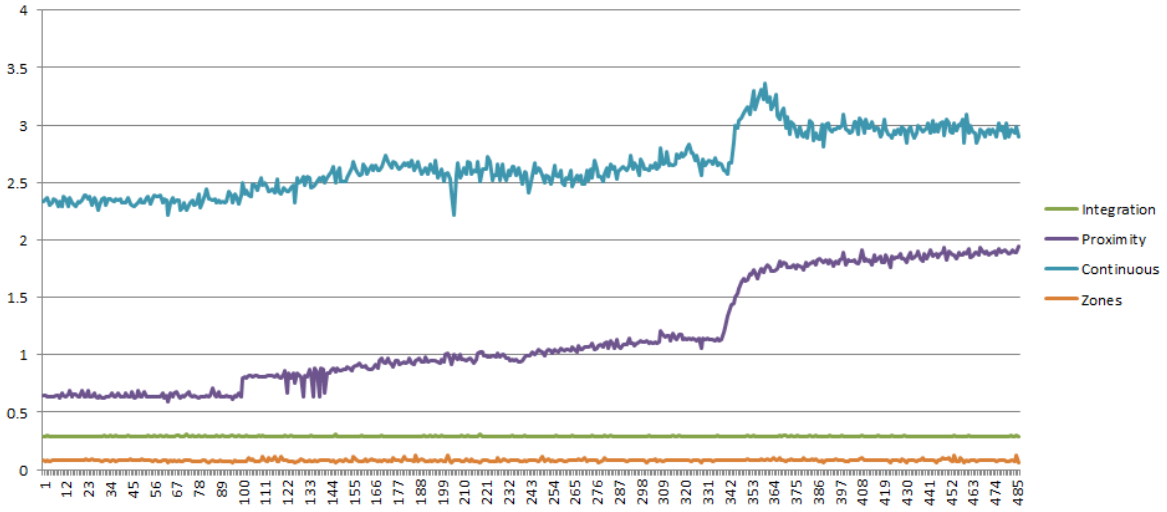
The third scenario again starts with the cloth lying flat. The middle point of the cloth is picked up, lifting the cloth off of the ground. This is a complex scenario as the edges

of the cloth all fly towards the center as the cloth clumps up.

The fourth scenario sees the cloth, that has been picked up in scenario three, dropped. This scenario is particularly stressful as the cloth crumples into a heap, with many dependent collisions.

In the final scenario, the cloth starts suspended in mid-air. The middle of the cloth is fixed, and when the simulation begins the two halves of the cloth speed towards each other. In particular, this tests the pipeline's ability to cope with a fast collision that involves much the cloth simultaneously.

Scenario 1



	Average	Maximum	Standard Deviation
Integration	0.29	0.31	0.00
Proximity	1.16	1.95	0.46
Continuous	2.66	3.36	0.25
Zones	0.08	0.13	0.01

The first scenario illustrates a few main things. The approximated integrator being used has a low, constant cost, as expected given its explicit nature. Against the cost of the collision handling, it is almost negligible. This relatively unconstrained test does not see the Impact Zones utilised whatsoever. At the beginning of this simulation, no collisions are occurring but a handful of collisions need to be tested by both the

proximity and continuous routines. At around the 340th frame, the cloth that is being folded over starts to approach the cloth that still lies on the ground. The proximity handling becomes permanently more expensive: those pieces of cloth that rest on top of each other are in constant contact. As the collision occurs, the continuous routine sees a sudden burst as it resolves the most tricky collisions, which are ones that happen at speed and which would be missed by the proximity based handling.

Scenario 2

	Average	Maximum	Standard Deviation
Integration	0.29	0.31	0.00
Proximity	1.20	2.04	0.51
Continuous	2.61	3.32	0.34
Zones	0.08	0.31	0.01

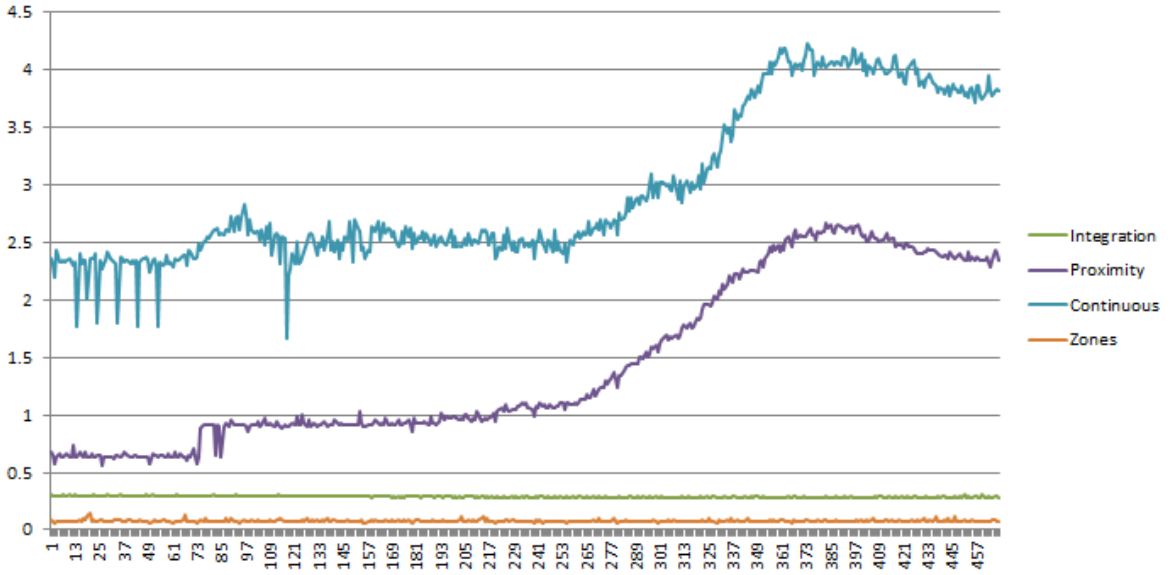
The second scenario starts as the first ends, and the cloth is unfolded back to its original position. Unsurprisingly, the right hand side of the second graph closely mimics the left hand side of the first, as the cloth resumes its original and unconstrained position. At the start of the test, the cost of detecting and handling collision quickly reduces as the portions of cloth that are resting on top of each other are moved apart.

The major takeaways from the first two scenarios are that there is a persistent cost to testing collisions even when none are occurring that is not insubstantial. In almost every circumstance, less continuous tests will be necessary than proximity tests but

the considerably higher cost of a single continuous test means that it is always the more expensive part of the pipeline. Furthermore, while the cost of proximity based handling tends to change smoothly, the cost of continuous handling varies more wildly.

For brevity, the integration is being excluded from the future tables as it is essentially constant regardless of the scenario.

Scenario 3



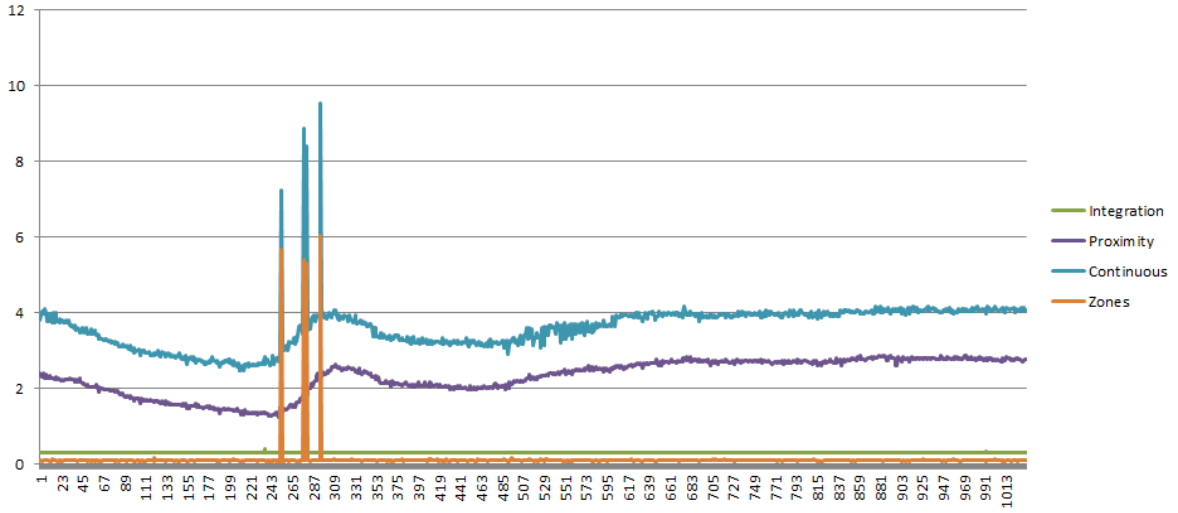
	Average	Maximum	Standard Deviation
Proximity	1.43	2.67	0.713
Continuous	2.95	4.23	0.67
Zones	0.08	0.148	0.01

The third scenario is far more intensive than the first or second. Here, the cloth is pulled up by a point in its center. Naturally, the edges of the cloth move inward at this point. The cloth clumps together naturally, meaning that lots of complex collisions have to be resolved.

Correlating with the previous finding, the continuous collision detection is still the most costly part of the pipeline - and in this scenario the impact zones are still not called upon. In this scenario the cost of the entire pipeline is far higher than in scenarios 1 and 2, although by the end of the scenario (where the cloth is suspended in midair

fully), the cloth reaches a near equilibrium and the costs start to fall.

Scenario 4

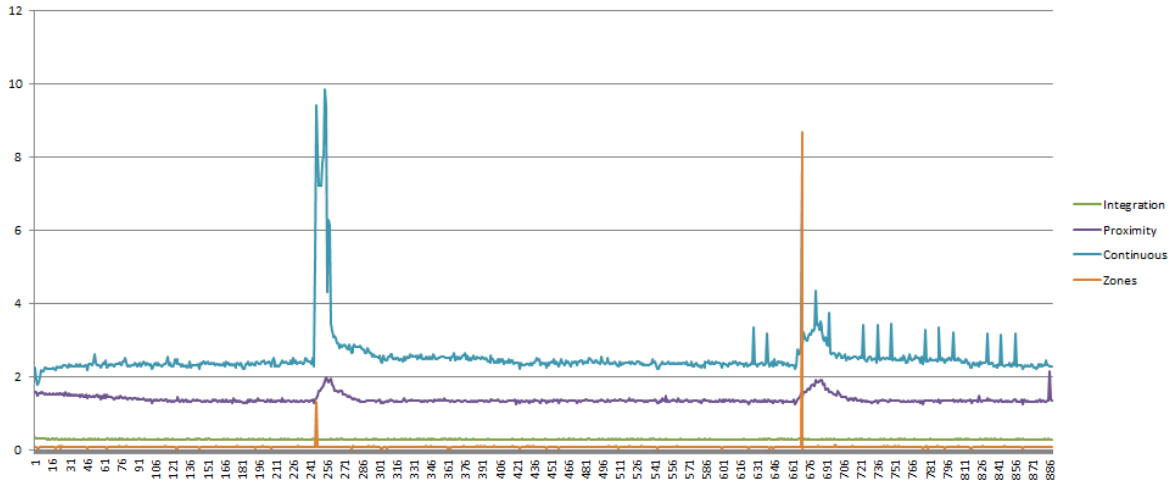


	Average	Maximum	Standard Deviation
Proximity	2.30	2.87	0.46
Continuous	3.60	0.11	0.56
Zones	0.11	6.03	0.34

The fourth scenario starts where the third ends, as the cloth is dropped. This is a highly constrained scenario as the cloth approaches the ground fairly hard and compresses, ensuring that many collisions occur. Here, the impact zones are indeed used. Spikes are seen in the continuous collision detection representing times where the five rounds of collision detection are needed and exhausted. The impact zones then take over, and resolve the collisions. Pleasingly, the impact zones are only used in three frames, which ensures that the rigidity will be unnoticeable to a viewer.

The maximum number of rounds of continuous collision detection can be used to put a rough upper bound on the cost of computation for even a robust pipeline. As can be seen from the three spikes though, there is a considerably variance on how long the five rounds took, implying that more collisions were tricky to resolve in the third spike than the first or second. A firmer bound might be found by limiting, as well as the number of rounds of continuous collision handling, the number of continuous collision tests.

Scenario 5



	Average	Maximum	Standard Deviation
Proximity	1.38	2.13	0.11
Continuous	2.52	9.85	0.70
Zones	0.09	8.67	0.29

The final experiment is also quite intensive, but in a slightly different way to the previous test. The collisions in this test are fast and all occur at more or less the same time. This forms two very clear spikes that occur first when the cloth hits originally, and second as it reapproaches after bouncing away.

5.2 Experiment 2

In the second experiment, the effect of the thickness value on the speed of collision is examined. The test itself is the same scenario as Experiment 1's Scenario 5, but instead of the 7mm used for thickness, a variety of measures from 1mm to 20mm are used. Theoretically, this will affect the speed of collision detection in two main ways.

For the proximity based detection, the AABBs in the BVH will scale with the thickness value (as each AABB is inflated by the thickness). This means more narrowphase tests will be necessary. Also, collisions will occur slightly earlier with a higher thickness when cloth approaches itself.

However, this should be more than countered by the fact that, with a higher thickness, collisions will be resolved more quickly, and more importantly, the proximity based handling will deal with a higher proportion of collisions compared to the continuous handling.

The following tables give average, maximum, and standard deviation values for six seconds of the simulation, which includes the collision of the cloth, and a period after.

1mm	Average	Maximum	Standard Deviation
Proximity	1.42	2.29	0.16
Continuous	3.10	17.10	2.50
Zones	1.23	29.49	4.06

3mm	Average	Maximum	Standard Deviation
Proximity	1.41	2.04	0.13
Continuous	2.78	15.81	22.69
Zones	0.40	22.69	1.96

5mm	Average	Maximum	Standard Deviation
Proximity	1.36	1.97	0.11
Continuous	2.67	13.83	1.44
Zones	0.18	10.56	0.83

7mm	Average	Maximum	Standard Deviation
Proximity	1.37	2.14	0.12
Continuous	2.62	10.84	1.11
Zones	0.16	9.36	0.8

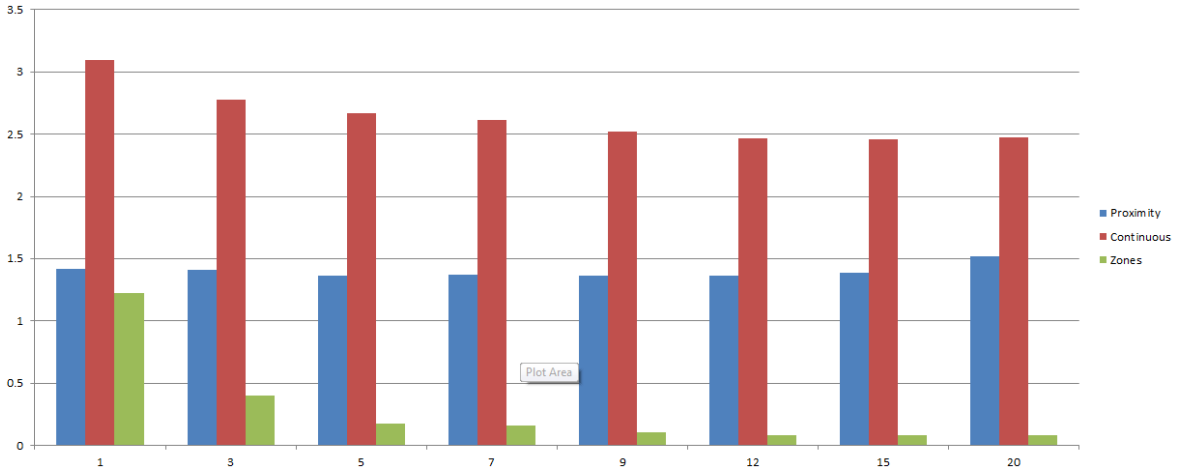
9mm	Average	Maximum	Standard Deviation
Proximity	1.36	2.05	0.12
Continuous	2.52	9.22	0.69
Zones	0.11	8.91	0.46

12mm	Average	Maximum	Standard Deviation
Proximity	1.36	2.16	0.15
Continuous	2.46	5.65	0.39
Zones	0.08	0.13	0.01

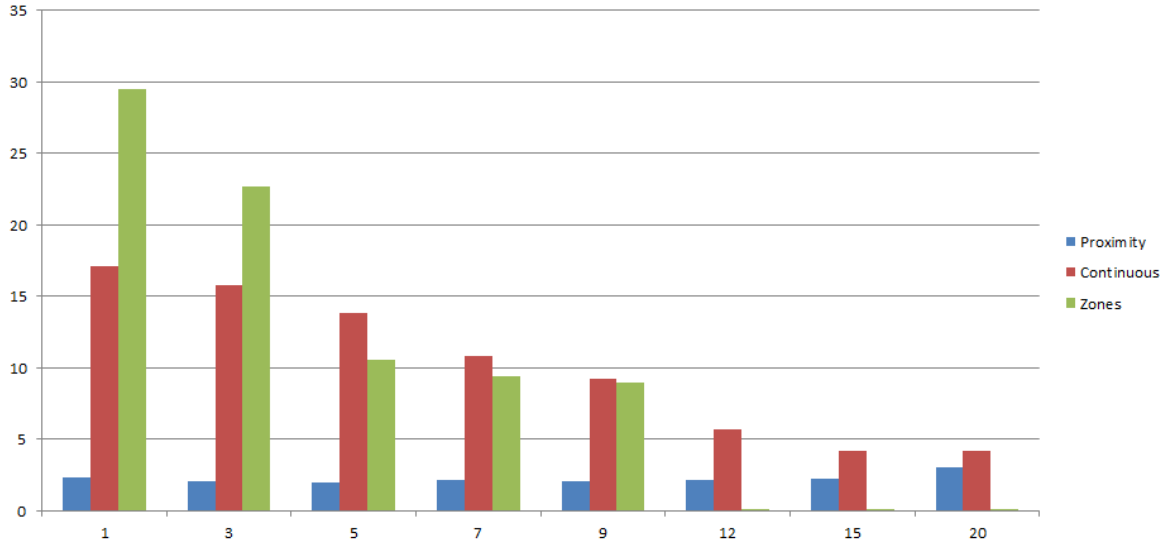
15mm	Average	Maximum	Standard Deviation
Proximity	1.39	2.25	0.15
Continuous	2.46	4.18	0.30
Zones	0.08	0.13	0.01

20mm	Average	Maximum	Standard Deviation
Proximity	1.52	3.01	0.21
Continuous	2.47	4.18	0.14
Zones	0.08	0.13	0.01

The data in these tables is compiled into two graphs, which show, respectively, the averages for each of the three pipeline stages, and the maximums for each of the three stages.



The graphs illustrate enormous performance differences between cloth simulations using thickness values. The graph of averages displays only marginal differences for the proximity and continuous handling methods. This is likely due to the scenario in question: the cloth is only in contact for a brief proportion of the six seconds. The graph of maximums displays a far more interesting result, and displays just how much



difference is made by increasing the thickness value. For this particular simulation, real time performance is not achieved with 1mm or 3mm cloth, but real time performance is very comfortably achieved by the 12, 15, and 20mm cloth which do not ever need to call on the Impact Zones to resolve the collision.

5.3 Experiment 3

As discussed in Section 4.9, the implementation of untangling was not completed, and hence the third experiment could not be completed. It was intended to provide a qualitative and quantitative examination of the comparison between a Bridson style collision handling pipeline and a pipeline which combined the cheap proximity based handling and untangling.

While this level of analysis cannot be provided, the incomplete implementation leads to some remarks regarding the ease of mapping untangling to the GPU. The untangling process described in [Baraff et al., 2003] and [Wicke et al., 2006] is non-history based. They work by analysing the state of the cloth and determining where intersections have occurred (as opposed to history based solutions, covered in Section 2.3.2, which check to see when the cloth changes side by examining past states.

Unfortunately, the non-history based approach, called Global Intersection Analysis, tends towards producing lots of small but different problems. It is often the case during untangling that many small intersections, of different types, occur. These may involve just one or two particles each, and each of these would need kernels launched to cope with them. This is precisely the kind of computation which is unsuited to the GPU. It may in fact be better to move data from the GPU to the CPU for untangling, before moving the results back to the GPU.

GPU based untangling is still a worthwhile area for future research, but it is hard to imagine that it will map as well to the GPU as other parts of the pipeline. It is very much recommended that any further research also considers the history based solutions such as [Volino and Thalmann, 1995].

Chapter 6

Conclusion

6.1 Limitations & Future Work

As discussed at length in Section 4.9, the implementation of a GPGPU based solution for untangling cloth is incomplete, and this would obviously be a worthy avenue for future work, despite reservations that untangling using Global Intersection Analysis in the style of [Baraff et al., 2003] and [Wicke et al., 2006] may not be an ideal candidate for massively parallel architectures.

The focus of this work being on self-collisions in particular, only basis cloth-plane collisions were implemented. General collision handling between cloth and rigidbody was not implemented, but doing so would not be especially complicated. In almost every way imaginable, the cloth-cloth collisions coped with in the current implementation are more complex, and costly to perform.

The implementation was in places based off of an assumption that the cloth mesh was square, and there is a guarantee that triangles come in pairs which make up a square. For practical implementation, regular triangular meshes are preferred. In particular, the construction of the BVH would have to be changed, as would the method by which particles are classified as inside/outside internal paths for untangling.

6.2 Closing Thoughts

The objectives of this project centralised on exploring the viability of achieving more complex cloth simulations more efficiently, with an eye towards real time, rather than offline performance. Though cloth simulation is starting to become a fairly common appearance in modern video games, the simulations are typically neglectful of collisions, particularly the more complex cloth-cloth (or self-collisions).

To produce simulations which are truly robust, fairly sophisticated collision handling is necessary. Cloth simulation was first explored in the literature in the late 1980s - it wasn't until 2002 that a scheme was proposed which could guarantee self-collisions would be dealt with while guaranteeing self-intersection will not occur. More recently, research has started exploring the possibility of achieving cloth simulation in parallel, initially using multi-threading on the CPU, but increasingly using the massively parallel GPU architecture. Recently, [Tang et al., 2013] performed a large portion of the [Bridson et al., 2002] pipeline via the GPU. The primary focus of their work was exploiting consumer GPUs to perform "high resolution" cloth simulation. They focus on simulations of cloth meshes involving 20k, 200k and 2m triangles respectively. They benchmarked dropping their cloth onto a Buddha statue, with frametimes of 3.1s, 38.8s, and 137.8s respectively.

In this work, focus was firmly on real time performance. Additionally, there was an interest in exploring how a fully robust collision handling could be achieved on the GPU. Both Impact Zones (the element of the [Bridson et al., 2002] scheme not implemented in [Tang et al., 2013]), and untangling were explored. Additionally, experiments were run to explore how the thickness value (a distance within which collisions are detected and handled) affects the cost of collision handling.

Quantitatively, the results covered in Chapter 5 illustrate the way the robust GPU pipeline works when put through its paces on a range of scenarios designed to produce countless self-collisions. They also illustrate the enormous difference that higher thickness values can achieve. An unrealistically large thickness can appear obvious with gaps appearing between cloth that should be in contact, but it also allows the discrete collision handling to deal with the vast majority of contact situations.

In much the same way that discrete collision detection alone can be used to produce extremely impressive rigid body simulations, discrete collision detection alone, tied with a large thickness value, can produce impressive cloth simulations at reasonable costs. In the short to medium term, high thicknesses may well be key to how cloth simulations can deal with self-collisions within a game's frame budget.

Despite these findings, it is entirely understandable that, with the current state of hardware, cloth simulations tend to neglect many cloth-object collisions and all cloth-cloth collisions. There is undoubtedly room for improvement in the implemented pipeline, such as a number of optimisations that were included in [Tang et al., 2013], but not so much that it is reasonable to imagine fitting a robust cloth pipeline into the portion of millisecond likely reserved for physically based animation in a modern game's budget for the forseeable future. Optimistically, one could conclude that the novel GPU implementation of impact zones moves us a step closer to seeing robust cloth simulations in real time.

Non-literature References

NVIDIA Flameworks *https://developer.nvidia.com/flameworks*

Angry Birds *https://www.angrybirds.com/*

Batman: Arkham City *http://rocksteadyltd.com/#arkham-city*

Box2D Engine *http://box2d.org/*

Digital Molecular Matter *http://www.pixelux.com/*

Star Wars: The Force Unleashed *http://www.starwars.com/games-apps/star-wars-the-force-unleashed*

Self CCD *http://gamma.cs.unc.edu/SELFCD/*

Bibliography

[Baciu and Wong, 2004] Baciu, G. and Wong, W. S.-K. (2004). Image-based collision detection for deformable cloth models. *IEEE transactions on visualization and computer graphics*, 10(6):649–63.

[Baraff and Witkin, 1998] Baraff, D. and Witkin, A. (1998). Large steps in cloth simulation. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH '98*, pages 43–54.

[Baraff et al., 2003] Baraff, D., Witkin, A., and Kass, M. (2003). Untangling cloth. *ACM Transactions on Graphics (TOG)*, pages 862–870.

[Bergen, 1997] Bergen, G. (1997). Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, pages 1–14.

[Breen et al., 1994a] Breen, D. E., House, D. H., and Wozny, M. J. (1994a). A Particle-Based Model for Simulating the Draping Behavior of Woven Cloth. *Textile Research Journal*, 64(11):663–685.

[Breen et al., 1994b] Breen, D. E., House, D. H., and Wozny, M. J. (1994b). Predicting the drape of woven cloth using interacting particles. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94*, pages 365–372.

[Bridson et al., 2002] Bridson, R., Fedkiw, R., and Anderson, J. (2002). Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics (ToG)*.

[Choi and Ko, 2005] Choi, K. and Ko, H. (2005). Stable but responsive cloth. *ACM SIGGRAPH 2005 Courses*.

[Curtis et al., 2008] Curtis, S., Tamstorf, R., and Manocha, D. (2008). Fast collision detection for deformable models using representative-triangles. *... of the 2008 symposium on Interactive ...*, 1(212):61–70.

[Eberly, 2010] Eberly, D. (2010). *Game physics*. CRC Press, 2 edition.

[Kang et al., 2000] Kang, Y.-M., Choi, J.-H., Cho, H.-G., and Park, C.-j. (2000). Fast and stable animation of cloth with an approximated implicit method. *Proceedings Computer Graphics International 2000*, pages 247–255.

[Karras, 2012] Karras, T. (2012). Thinking Parallel, Part II: Tree Traversal on the GPU.

[Lauterbach et al., 2010] Lauterbach, C., Mo, Q., and Manocha, D. (2010). gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Computer Graphics Forum*, 29(2):419–428.

[Macklin et al., 2014] Macklin, M., Müller, M., Chentanez, N., and Kim, T.-Y. (2014). Unified particle physics for real-time applications. *ACM Transactions on Graphics*, 33(4):1–12.

[Mirtich, 2000] Mirtich, B. (2000). Timewarp rigid body simulation. *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*, pages 193–200.

[Möller, 1997] Möller, T. (1997). A Fast Triangle-Triangle Intersection Test. *Journal of Graphics Tools*, 2(2):25–30.

[Müller et al., 2007] Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. (2007). Position based dynamics. *Journal of Visual ...*.

[Nealen and Müller, 2006] Nealen, A. and Müller, M. (2006). Physically based deformable models in computer graphics. *Computer Graphics ...*.

[Provot, 1995] Provot, X. (1995). Deformation constraints in a mass-spring model to describe rigid cloth behaviour. *Graphics interface*.

[Provot, 1997] Provot, X. (1997). *Collision and self-collision handling in cloth model dedicated to design garments.*

[Redon et al., 2000] Redon, S., Kheddar, A., and Coquillart, S. (2000). An algebraic solution to the problem of collision detection for rigid polyhedral objects. *Proc. of IEEE Conference on Robotics ...*, (April):3733–3738.

[Redon et al., 2002] Redon, S., Kheddar, A., and Coquillart, S. (2002). Fast Continuous Collision Detection between Rigid Bodies. *Computer Graphics Forum*, 21(3):279–287.

[Selle et al., 2009] Selle, A., Su, J., Irving, G., and Fedkiw, R. (2009). Robust high-resolution cloth using parallelism, history-based collisions, and accurate friction. *IEEE transactions on visualization and computer graphics*, 15(2):339–50.

[Soman et al., 2010] Soman, J., Kishore, K., and Narayanan, P. J. (2010). A fast GPU algorithm for graph connectivity. *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8.

[Tang, 2010] Tang, M. (2010). Self-CCD: Continuous Collision Detection for Deforming Objects.

[Tang et al., 2009a] Tang, M., Curtis, S., Yoon, S.-E., and Manocha, D. (2009a). ICCD: interactive continuous collision detection between deformable models using connectivity-based culling. *IEEE transactions on visualization and computer graphics*, 15(4):544–57.

[Tang et al., 2011a] Tang, M., Manocha, D., Lin, J., and Tong, R. (2011a). Collision-streams: fast gpu-based collision detection for deformable models. *Symposium on interactive 3D ...*, 1(212):63–70.

[Tang et al., 2009b] Tang, M., Manocha, D., and Tong, R. (2009b). Multi-core collision detection between deformable models. *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling on - SPM '09*, page 355.

[Tang et al., 2010] Tang, M., Manocha, D., and Tong, R. (2010). Fast continuous collision detection using deforming non-penetration filters. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10*, page 7.

[Tang et al., 2011b] Tang, M., Manocha, D., Yoon, S.-E., Du, P., Heo, J.-P., and Tong, R.-F. (2011b). VolCCD. *ACM Transactions on Graphics*, 30(5):1–15.

[Tang et al., 2013] Tang, M., Tong, R., Narain, R., Meng, C., and Manocha, D. (2013). A GPU-based Streaming Algorithm for High-Resolution Cloth Simulation. *Computer Graphics Forum*, 32(7):21–30.

[Terzopoulos et al., 1987] Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K. (1987). Elastically deformable models. *ACM Siggraph Computer ...*, 21(4):205–214.

[Teschner et al., 2003] Teschner, M., Heidelberger, B., and Müller, M. (2003). Optimized spatial hashing for collision detection of deformable objects.

[Teschner and Kimmerle, 2005] Teschner, M. and Kimmerle, S. (2005). Collision detection for deformable objects. *Computer Graphics ...*, xx(x).

[Volino and Thalmann, 1994] Volino, P. and Thalmann, N. (1994). Efficient self collision detection on smoothly discretized surface animations using geometrical shape regularity. *Computer Graphics Forum*.

[Volino and Thalmann, 1995] Volino, P. and Thalmann, N. (1995). *Collision and self-collision detection: Efficient and robust solutions for highly deformable surfaces*.

[Wicke et al., 2006] Wicke, M., Lanker, H., and Gross, M. (2006). Untangling cloth with boundaries. *... of Vision, Modeling, and Visualization (VMV ...*.

[Zeller, 2005] Zeller, C. (2005). Cloth simulation on the GPU. *ACM SIGGRAPH 2005 Sketches*, page 2003.