Implicit and Explicit Integration Methods in Cloth Simulation

Submitted in partial fulfilment
of the requirements of the degree
Bachelor of Science (Honours)
of Rhodes University

Gavin Hayler, Shaun Bangay and Adele Lobb

7th November 2004

Abstract

The purpose behind this project is to examine different methods of integration that can be used in cloth simulation and to determine what uses the methods are best suited for. Past work in the field is investigated and how the different methods work is explained. Implementation issues are discussed and some results are presented. It is determined that the explicit methods, though unstable at large time steps, is best suited to real time applications, such as computer games. The implementation of the implicit method fails to perform as expected and claimed by its authors and it is decided that further investigation into the reason is required. If the implicit method were to work as claimed, it would be perfectly suited to an offline rendering application such as animation rendering.

Acknowledgements

Many thanks to my supervisors Shaun Bangay and Adele Lobb for their constructive criticism and guiding me through this project.

An extra special thanks goes to Shaun Bangay, without whom this project would have been a lot more difficult, if possible at all. Your explanations and research into the maths is very much appreciated.

I am also very grateful to my mom and dad, Jean and Mike Hayler, for sacrificing their resources to enable me to complete this year and project and for their never-ending support.

Thanks to Angeline Paxton for supporting me, always believing in me and just generally putting up with me.

I would also like to acknowledge the financial and technical support of the project of Telkom SA, Business Connexion, Comverse SA, and Verso Technologies through the Telkom Centre of Excellence at Rhodes University.

Lastly, thanks to my lab-mates for their input, especially Robin Smith and Jon Leigh. Thanks guys.

Contents

Chapter 1

Introduction

Cloth modelling refers to the process involved in simulating the behaviour and look of cloth, and has been an interest of animators, programmers and engineers for some time. In more recent years, however, it has become more researched with the explosion of the gaming and CGI (Computer Generated Imagery) industries. Each have different needs from the models or solutions that they employ.

Game developers need a model which is fast enough to be implemented in a game and used in real-time. They need believable looking cloth which allows a user to play the game without any latency or slowing down of the system. Without this property, the user is not drawn into the reality of the game or is aware of the computations taking place behind the scenes, making it a much less enjoyable experience and making the game less likely to sell.

In contrast, CGI animators require the model to perform as close to real cloth as possible but the time taken to process the cloth is not of primary importance, since the actual rendering is done offline on a render farm (a large number of computers dedicated to the task of rendering a scene).

There are basically two areas to a cloth simulation, cloth dynamics and collision detection, but we will only be investigating cloth dynamics in the project - more specifically we will be looking at different methods of integration, the importance of which will be explained later.

Many different types of integration have been used in the past, but they usually fall into the category of either explicit integration or implicit integration. We will be examining the two different integrations and the way the integrations used have migrated from explicit to implicit methods as the area of cloth modelling has matured

Chapter 2

Related Work

2.1 The Physical Model

Previously, a number of different physical models have been used to model cloth, such as:

- a continuum, as described by Amirbayat et al. [Amirbayat and Hearle 1989]
- Eischen et al. [Eischen et al. 1996] use a non-linear shell model;
- and Baraff and Witkin [Baraff and Witkin 1998] use a system of interconnected flat triangles that are treated as a continuum.

These models have been superseded by a system of particles which is now more commonly used to represent the cloth. This model is defined well by Choi et al. [Choi and Ko 2002], who draw their model from an idea that started with Breen et al. [Breen et al. 1994] who were first to apply a particle system to the simulation of cloth. Choi et al. [Choi and Ko 2002] describe the model as a mesh of interconnected particles which approximate the cloth. These particles are connected by a system of springs which are assigned attributes depending on the type of interaction being simulated (stretching or compression). In general, variations of this physical model are now used throughout the cloth modelling field.

2.2 Integration

There are two main types of integration used in cloth modelling - explicit integration and implicit integration. Explicit integration dominated the field for some time, because it is the most intuitive

and the fastest solution, until Baraff and Witkin [Baraff and Witkin 1998] introduced the idea of implicit integration into cloth modelling. We shall now examine these two integration methods.

2.2.1 Explicit Integration

The most commonly mentioned examples of explicit integration are the Euler and Runge-Kutta methods. These methods are fast and easy to implement, but require a small time step to maintain stability, as each new result is based on the previous result, and errors in one time step can create dramatic problems in later time steps. Because of this, this type of integration is usually used in real-time applications such as games where the time step is small and the regularity of the system cannot be determined due to the multitasking environment of today's operating systems. [Rudomin and Castillo 2002].

Explicit integration attempts to solve the system based on the information available at time t_0, and is unaware of what the state of the system will be at the end of the time step. It takes no notice of wildly changing derivatives, caused by compounded errors, as it continues to solve the system. This is the reason that the time step has to be small to maintain stability, so that there will not be any huge changes that could corrupt the results [Baraff and Witkin 1998]. Each step of an explicit method is faster than that of an implicit method, however the more particles that exist in the system, the smaller the time step needed to keep the stability of the system [Oshita and Makinouchi 2001]. Therefore, providing the particle system used to approximate the cloth has relatively few particles in it, explicit integration works well. In general, game developers try to keep the number of vertices for each object to a minimum, making it an excellent application for explicit integration.

Many real-time applications do, in fact, use explicit integration for their cloth engines. One example of this in industry is the game Hitman: Codename 47. The paper written by the author of its physics engine [Jakobsen 2001] discusses the methods used to create the cloth dynamics in their game. A variant of explicit integration called Verlet integration was used and is, as the paper shows, relatively easy to implement. Other papers whose aim is real-time simulation from an academic point of view, also use explicit methods, and obtain good results[Rudomin and Castillo 2002],[Oshita and Makinouchi 2001]. For example: Rudomin et al. [Rudomin and Castillo 2002] achieved no less than 30 frames per second and often up to 60 frames per second while producing realistic results, however, they make no mention of the stability of the system.

2.2.2 Implicit Integration

Baraff and Witkin [Baraff and Witkin 1998] introduced the idea of implicit integration into the cloth modelling world. This new method allows large time steps, and is able to handle large numbers of particles without much variance in the running times (ie: 5%). It is largely used for accurate simulation of high resolution models, where the application is not intended to be real-time, namely animations.

Baraff and Witkin [Baraff and Witkin 1998] describe their integration as "backward" Euler, because it starts from the output state having the values for the position of the particle: $(x_0 + \triangle x)$ and $(v_0 + \triangle v)$ where x_0 and v_0 are the position and velocity at the start of the time step and $\triangle x$ and $\triangle v$ are the change in position and velocity during the step. It then uses a forward Euler step to run the system backward in time. This forces it to find an output state whose derivative points back to where the system started from at the beginning of the time step, which brings an extra level of consistency to the integration.

Choi and Ko [Choi and Ko 2002] implement a variant of Baraff and Witkin's implicit integration, and demonstrate that they are able to use a time step of up to 100 seconds and still keep the system stable, even though the animation is choppy. However, when they use a more realistic time step, they are able to achieve most convincing results using a clothed animated character. Etzmuß et al. [Etzmuß et al. 2001] also use an implicit integration for their system, and are able to achieve realistic results.

Chapter 3

Methodology

3.1 Physical Model

The physical model that is used for this project is based on the model that is outlined by Choi et al. [Choi and Ko 2002], and is the most widely used. The cloth is converted into a 3D model with a certain number of vertices, n. These vertices are then represented by n particles in a particle system, and springs or relationships are set up between certain particles. There are two different types of springs used in the cloth model, based on the type of interaction they are attempting to simulate: springs for compression interaction and springs to simulate a stretching interaction. Particles that are directly adjacent to each other are linked with a stretching interaction (the red lines in figure 3.1) and particles that are two particles apart and linked with a compression interaction (the blue lines in the same figure). These different interactions are used to calculate the forces that are being exerted on a particular particle by its surrounding particles. Choi et al. decided that the stretching and compression interactions are different enough to warrant different formulas, whereas often the formulas used in the interaction calculations are very similar if not the same. Each force, due to the interactions with a particular particle, is calculated and added to produce the resultant force acting upon the particle in question.

3.1.1 Stretching

As mentioned before, particles that are directly adjacent to one another are used to calculate the forces being exerted due to stretching interaction. This interaction is represented by a linear spring model, where the force acting on particle i due to the deformations between particle i and particle j is calculated using

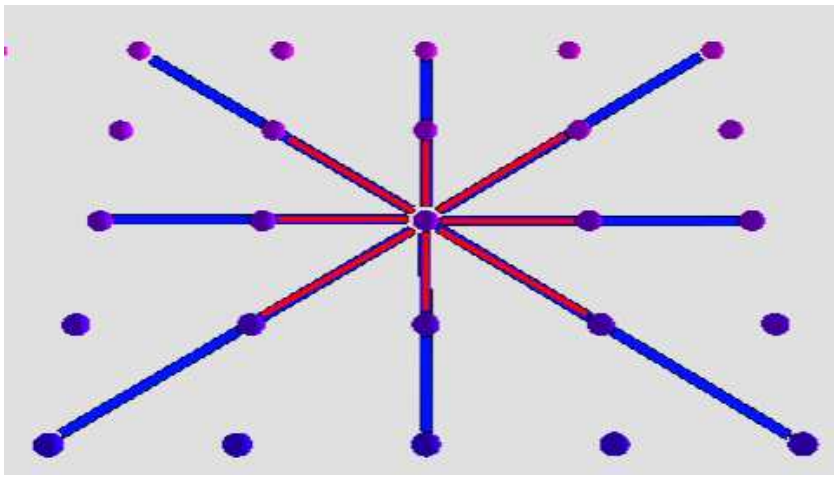

Figure 3.1: Physical Model of Cloth

$$f_i = \begin{cases} k_s(|\mathbf{x}_{ij}| - L)\frac{\mathbf{x}_{ij}}{|\mathbf{x}_{ij}|} & : \quad |\mathbf{x}_{ij}| \geq L \\ 0 & : \quad |\mathbf{x}_{ij}| < L \end{cases} \tag{3.1}$$

where $\mathbf{x}_{ij} = \mathbf{x}_j - \mathbf{x}_i$, L is the natural rest length of the spring (when no forces are being exerted on the spring), and k_s is a predetermined spring constant. [Choi and Ko 2002]

As can be seen from equation 3.1, there is no force being exerted if the spring's current length is less than its rest length. This is because this interaction only deals with stretching, which is where the spring's current length is greater than its rest length (IE: it is being stretched).

3.1.2 Compression

The compression interaction forces are computed with the following formula:

$$f_i = f_b(|\mathbf{x}_{ij}|)\frac{\mathbf{x}_{ij}}{|\mathbf{x}_{ij}|} \tag{3.2}$$

where

$$f_b(|x_{ij}|) = k_b\kappa^2(\cos\frac{\kappa L}{2} - \text{sinc}(\frac{\kappa L}{2}))^{-1} \tag{3.3}$$

and the curvature k of an arc can be expressed in terms of the distance between two points.

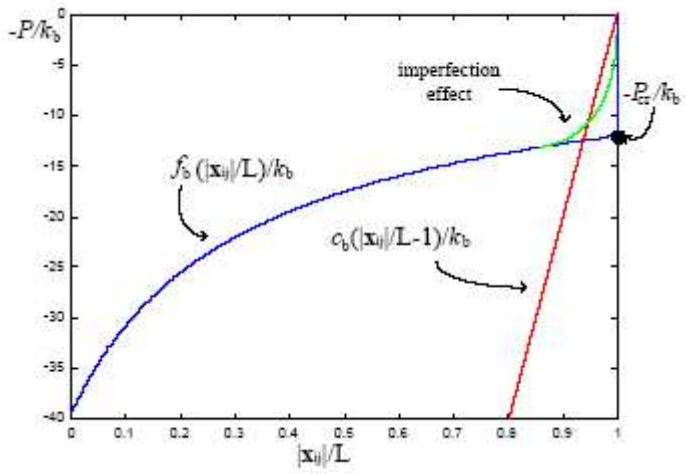

Figure 3.2: Curves of P vs. $|\mathbf{x}_{ij}|$

κ can be calculated using the following formula:

$$\kappa = \frac{2}{L}\text{sinc}^{-1}(\frac{|\mathbf{x}_{ij}|}{L}) \tag{3.4}$$

where $\text{sinc}(x) = \frac{\sin(x)}{x}$.

Unfortunately, since no useful definition for the function $\text{sinc}^{-1}(x)$ could be found, the best way to use it in implementation is to create a lookup table with all the values that could be needed, and write a function that looks up the correct value when the corresponding value is passed to it.

In real systems, geometric imperfections in the structure cause the fabric to begin buckling when the compressive force is first applied. In figure 3.2 (taken from Choi et al.[Choi and Ko 2002]), the blue curve shows the dependence of the function f_b on the distance between particles $|\mathbf{x}_{ij}|$. As is shown by the figure, when the compressive force P is initially applied (the top right corner), if we were to just use the function f_b, the structure would remain straight until the compressive force reached the bucking force or critical force P_{cr}. In order to model real systems more closely, we would have to model an actual curve (the green curve in the figure) that shows an amount of deflection even when the compressive force is small. This actual curve would then need to approach the behaviour of f_b as the compressive force increases.

To model this characteristic, another function (f_b^*) is used in the implementation, which simulates this behaviour at small compressive forces, and as the force increases behaves as f_b would.

$$f_b^* = \begin{cases} c_b(|\mathbf{x}_{ij}|) - L & : \quad f_b < c_b(|\mathbf{x}_{ij}| - L) \\ f_b & : \quad \text{otherwise} \end{cases} \tag{3.5}$$

where c_b is an arbitrary compression constant, similar to our spring constant k_s[Choi and Ko 2002].

3.1.3 Damping

One final attribute needs to be added to the model of the cloth: damping. Many models in the past have introduced a damping term into the model in order to keep it stable, but the side-effect of this is that the cloth begins to move unrealistically and is resistant to natural movement. The damping term introduced into the model by Choi et al. is merely to model the natural property of cloth to dissipate its energy. Without this term, the simulated cloth can move in large unrealistic oscillations. The damping force exerted on particle i due to the interaction from particle j is shown below.

$$f_i = -k_d(\mathbf{v}_i - \mathbf{v}_j) \tag{3.6}$$

where k_d is a damping constant of our choice and $\mathbf{v}_i$ and $\mathbf{v}_j$ are the velocites of the two particles. This damping force is added to every interaction force that is calculated.

3.1.4 Pseudo Code

A pseudo code example for the force calculations would be as follows:

```
Add external forces to each particle (eg: gravity, wind, etc.);
For each particle, calculate the stretching force and add
   a damping force before adding it to the particle;
For each particle, calculate the compression force as done
   for the stretching force;
After completing this procedure for the particles, each
   particle contains the resultant force that is being
   applied to it taking into account all types of forces.
```

3.2 Integration

Integration (in terms of cloth modelling) is the process of calculating the next position that each particle must move to and its velocity based on the forces acting upon that particle, its current position and its current velocity (more information about the particle can be required for certain methods). A time step is a period of time, that is chosen before the integration is run, by which the system is advanced on each iteration. Therefore, if the time step is 1 and the current time is n, after the next time step, the system will be at time $n + 1$. As the system is advanced through time, the state the simulation is in at some point corresponds to the state that a real life example of the simulation would be in at that same point in time.

The integration step calculates the positions and velocities of the particles based on the forces that are acting on the particles, thus the integration step occurs after all the interactions have taken place and have calculated the forces that are being exerted on the individual particles.

3.2.1 Explicit Integration

As was mentioned before, explicit integration was the most widely used method of integration for some time. It is still quite widely used for certain applications for one reason - it is fast. It is not a particularly stable method at large time steps or at large numbers of particles, but in real-time simulations, speed is the main driving force and the models used in such applications are usually low in detail.

The method of explicit integration that was used in this project is a method called Verlet integration and is taken from Jakobsen [Jakobsen 2001]. Verlet is different from the classic Euler integration which is defined as

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_n(\triangle t) \tag{3.7}$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n(\triangle t) \tag{3.8}$$

where $\mathbf{x}_n$, $\mathbf{v}_n$, $\mathbf{x}_{n+1}$ and $\mathbf{v}_{n+1}$ are the current position and velocity and the new position and velocity of the particle respectively, and $\triangle t$ is the time step. $\mathbf{a}_n$ is the acceleration of the particle and is computed using Newton's law:

$$\mathbf{f}_n = m\mathbf{a_n} \tag{3.9}$$

where $\mathbf{f}_n$ is the accumulated force acting on the particle (as computed beforehand).

Verlet, on the other hand, does not explicitly use the velocity of the particle in the calculation. Instead, it uses the current position $\mathbf{x}_n$ and the old position $\mathbf{x}_{n-1}$ to calculate the new position.

$$\mathbf{x}_{n+1} = 2\mathbf{x}_n - \mathbf{x}_{n-1} + \mathbf{a}_n(\triangle t)^2 \tag{3.10}$$

$$\mathbf{x}_{n-1} = \mathbf{x}_n \tag{3.11}$$

$$\mathbf{x}_n = \mathbf{x}_{n+1} \tag{3.12}$$

Verlet integration is more stable than other explicit integration methods, because the velocity is implicitly given instead of explicitly stored, which makes it harder for the velocity and position to get out of sync. It works because the term $2\mathbf{x}_n - \mathbf{x}_{n-1}$ from equation 3.10 is equal to $\mathbf{x}_n + (\mathbf{x}_n - \mathbf{x}_{n-1})$ and $\mathbf{x}_n - \mathbf{x}_{n-1}$ is an approximation of the current velocity (or the distance travelled during the previous time step). Verlet integration is often used when simulating molecular dynamics, however, is not always accurate, as energy may dissipate or leave the system, but it is fast and relatively stable. [Jakobsen 2001]

It is possible to introduce a small amount of drag or damping into the system by changing the integration calculation from equation 3.10 to something which resembles $\mathbf{x}_{n+1} = 1.99\mathbf{x}_n - 0.99\mathbf{x}_{n-1} + \mathbf{a}_n(\triangle t)^2$, but as explained in section 3.1.3, this sort of damping may make the cloth unrealistically resistant to movement, and in our system, the damping term was added during the force calculations using the method described by Choi et al.

Pseudo Code:

```
For each particle
Calculate next position using past and current position
Old position takes current position value
Current position takes new position value
Next particle
```

Implicit Integration

3.2.2 Implicit Integration

Implicit integration was first introduced into cloth simulation by Baraff and Witkin [Baraff and Witkin 1998] in 1998 and was a revolutionary step in cloth modelling. In contrast to the explicit methods which use current or past information about the particle to compute the next position and velocity, implicit integration estimates the next position and uses that in a calculation to see if the prediction was correct. Depending upon the order of the implicit function, this procedure may be performed more than once. The implicit integration that was used in this paper was a method described by Choi et al. [Choi and Ko 2002], which they term a semi-implicit integration method with a second-order backward difference formula (BDF).

In their formula, they use the partial differentials $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ which are calculated using the formulas given throughout the paper. The first partial differential $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ is calculated with values taken from the two types of interaction in the physical model: stretching and compression. These equations are given below in 3.13 for the stretching interaction and 3.14 for the compression interaction.

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} = \begin{cases} k_s \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}} + k_s(1 - \frac{L}{|\mathbf{x}_{ij}|})(\mathbf{I} - \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}}) & : \quad |\mathbf{x}_{ij}| \geq L \\ 0 & : \quad |\mathbf{x}_{ij}| < L \end{cases} \tag{3.13}$$

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} = \frac{df_b^*}{d|\mathbf{x}_{ij}|}\frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}} + \frac{df_b^*}{|\mathbf{x}_{ij}|}(\mathbf{I} - \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}}) \tag{3.14}$$

The terminology and values used in the above equations are the same ones that are used in the physical model in section 3.1.1 and 3.1.2. The results from equation 3.13 and equation 3.14 are then added together to form the final partial differential $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ which is used in the integration calculation. The second partial differential $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ is calculated using equation 3.15 and uses the damping constant k_d which was mentioned in section 3.1.3.

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{v}_j} = k_d \mathbf{I} \tag{3.15}$$

The change in position and hence the next position can be calculated by solving the following

equation for $(\mathbf{x}_{n+1} - \mathbf{x}_n)$:

$$(\mathbf{I} - \triangle t\frac{2}{3}\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \triangle t^2\frac{4}{9}\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{x}})(\mathbf{x}_{n+1} - \mathbf{x}_n)$$

$$= \frac{1}{3}(\mathbf{x}_n - \mathbf{x}_{n-1}) + \frac{\triangle t}{9}(8\mathbf{v}_n - 2\mathbf{v}_{n-1})$$

$$+\frac{4 \triangle t^2}{9}\mathbf{M}^{-1}(\mathbf{f}_n - \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\mathbf{v}_n) - \frac{2 \triangle t}{9}\mathbf{M}^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{v}}(\mathbf{x}_n - \mathbf{x}_{n-1}) \tag{3.16}$$

If we calculate the left term of the left-hand side(LHS) and the right-hand side(RHS) we will have two matrices filled with the correct data. To solve for $\triangle \mathbf{x}$, we need to multiply the RHS matrix by the inverse of the LHS matrix. If the current position $\mathbf{x}_n$ is added to the resultant $\triangle \mathbf{x}$, it will produce the next position $\mathbf{x}_{n+1}$.

Using the next position, we are then able to calculate the following velocity $\mathbf{v}_{n+1}$ by solving

$$\mathbf{v}_{n+1} = \frac{1}{\triangle t}(\frac{3}{2}\mathbf{x}_{n+1} - 2\mathbf{x}_n + \frac{1}{2}\mathbf{x}_{n-1}) \tag{3.17}$$

The forces that will be exerted on each particle at the next time step are calculated using

$$\mathbf{f}_{n+1} = \mathbf{f}_n + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}_{n+1} - \mathbf{x}_n) + \frac{\partial \mathbf{f}}{\partial \mathbf{v}}(\mathbf{v}_{n+1} - \mathbf{v}_n) \tag{3.18}$$

Pseudo Code:

```
Fill matrices with data from particles;
Calculate partial differentials;
Evaluate the next position, next velocity, and next force equations;
Assign all calculated values back into the particles.
```

Chapter 4

Implementation

The following chapter explains some of the more interesting concepts behind the implementation of this project.

Visual C++ was used for the implementation as the development language and environment simply because of familiarity with the language. Furthermore, through experience it has proved to be faster than a number of other languages, such as Java. The algorithms presented in the previous chapter can however be implemented in most languages. In the final implementation, the numerical systems have been applied to a graphical representation of cloth. The OpenGL 3D graphics API has been used to render the cloth simulation as a 3D deformable model of a piece of cloth.

4.1 Physical Model

The physical model that was used in our implementation, as was stated in chapter 3.1, is a particle system, in which the particles are linked together with springs, and particles that are near one another react to each other's movement. In order to create these sorts of relationships in the implementation, the links between the particles had to be stored in some way.

In our implementation, we chose to create a number of classes and create links or constraints between those classes. Figure 4.1 and figure 4.2 contain the listings of the *Constraint* class and the *Particle* class respectively, which will be explained shortly.

The *Constraint* class can be thought of as a spring which is connecting two particles together. In the class, there is a pointer to a particle and a value *RestLength*. The *RestLength* contains the distance (the same value referenced in the algorithms presented previously in chapter 3) between

```
class Constraint
{
public:
    Constraint(){};
    ~Constraint(){};
    Particle *p;       //Pointer to particle in main array
    float RestLength;  //Rest length between two particles
};
```

Figure 4.1: Constraint Class

one particle and the particle that is being pointed to by *p* when there are no forces being exerted between the two particles.

The *Particle* class, a unique instantiation of which is used to represent each particle in the system, holds the information about a particle that is needed for the calculations, such as its position, velocity, the force currently being exerted on the particle, and so on. The *CVector3* class simply holds 3 values (the x, y, z values) and provides some useful vector functionality.

The interesting parts of the *Particle* class are the *Svs* and *Cvs Constraint* arrays and the unique identifier (UID). The UID is simply that - it is used to uniquely identify each particle, which is useful when performing the implicit integration calculations. The *Svs* and *Cvs Constraint* arrays hold the link or spring information between this particle and the particles that surround it. Thus, the *Svs* array contains constraints which point to the particle object representing the particles that are adjacent to this particle (IE: the particles which take place in the stretching interaction force calculations).

These relationships will be explained further later on.

The *ParticleSystem* class, listed in figure 4.3, is where the main work of the cloth simulation gets done. For now, the particle array *Parts* is of interest. This array contains all of the *Particle* objects used in the system - one for each particle. It is to these objects that the pointer in the *Constraint* class actually points.

Figure 4.4 explains the relationships between the particles and their constraints in a diagrammatic form. In the figure, the particle system consists of a number of particles, each of which is a *Particle* object. Each of the *Particle* objects contains information about the particle and the arrays of *Constraint* objects as well. Taking, for example, an element from the *Svs* array (constraints

```
class Particle
{
public:
    Particle(){};
    ~Particle(void){};
    void SetVector(int thevector, float x, float y, float z);
    void SetVector(int thevector, CVector3 &refVec);
private:
    CVector3 Pos;          //Vector holding the current position
    CVector3 OldPos;       //Vector holding the previous position
    CVector3 Force;        //Vector holding the current force
    CVector3 OldForce;     //Vector holding the previous force
    CVector3 Velocity;     //Vector holding current velocity
    CVector3 OldVelocity;  //Vector holding previous velocity

    Constraint *Svs;       //Stretching interaction constraints
    Constraint *Cvs;       //Compression interaction constraints

    int SvsSize, CvsSize;  //Size of above arrays
    int PartID;            //Unique ID
};
```

Figure 4.2: Particle Class

```
class ParticleSystem
{
public:
    ParticleSystem(){};
    ParticleSystem(int numOfParts, Particle *parts);

    ~ParticleSystem();
    void TimeStep();
    void AccumulateForces();
    void holdParts(int *holdparts, int size);

    int NUM_PARTICLES;    // No of particles in system
    CVector3 m_vGravity;  // Gravity
    float m_fTimeStep;    // Time step
    Particle *Parts;      // Array of particles
private:
    void Verlet();        // Explicit integration method
    void Implicit();      // Implicit integration method

    void Type1Int();      // Stretching interaction force calculation
    void Type2Int();      // Compression interaction force calculation
    ...
    ...
};
```

Figure 4.3: ParticleSystem Class

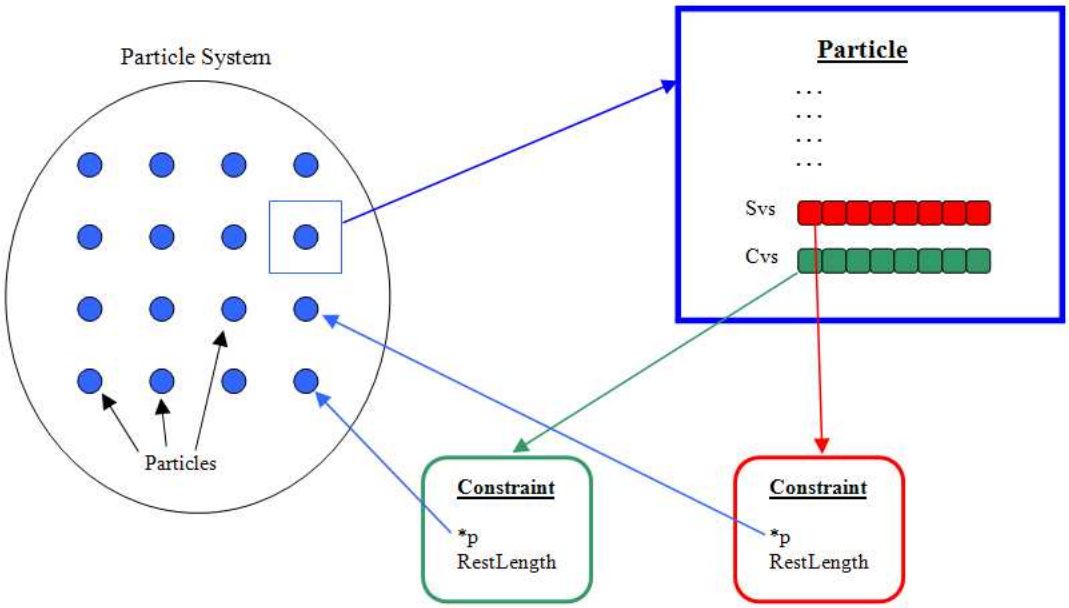

Figure 4.4: Particle System Relationships

regarding stretching interaction) and expanding it, we can see that it is just an instantiation of the *Constraint* class with the same properties. The *p* pointer points back into the particle system at the single particle that the original particle is linked to.

4.2 Force Calculations

The force calculations were implemented as the algorithms in section 3.1 explain. The only notable element of implementing the force calculations was trying to find a way around the non-existent (or so it seems) definition of the $\text{sinc}^{-1}(y)$ function, and to speed up the $\text{sinc}(x)$ calculation, which are used in the calculation of the compression interaction force.

In section 3.1.2, we defined $\text{sinc}(x)$ as $\frac{\sin(x)}{x}$. If we were to recalculate this trigonometric equation for every compression interaction force calculation, it would become quite expensive with regards to speed. Using the definition of $\text{sinc}(x)$, we create the *sincy* function listed in figure 4.5. The way the function works is that the first time it is called, it populates an array

```
// for x in range 0 to SINC_MAX
double sincy (double x)
{
   static bool initialized = false;
   static double table [MAX_SINC_RES];
   if (!initialized)
   {
       for (int i = 0; i < MAX_SINC_RES; i++)
       {
           // number in range 0 to SINC_MAX.
           double v = SINC_MAX * (double) i / (double) MAX_SINC_RES;
           if (v != 0.0)
             table[i] = sin (v) / (v);
           else
             table[i] = 1.0;
       }
       initialized = true;
   }

   int index = (int) ((double) MAX_SINC_RES * (x / SINC_MAX));

   if ((index < 0) || (index >= MAX_SINC_RES))
   {
       return 0.0;
   }
   return table[index];
}
```

Figure 4.5: *sincy* Function Definition

with results of passing the function meaningful values of x. After that though, the values are kept and merely accessed and returned on future calls to the function, which speeds up the force calculation because it no longer continually re-evaluates the same calculations.

The *asinc* function in figure 4.6 works in a similar way, using the lookup table and populating it on the first call to the function, however we are not provided with an easy to use definition of what $\text{sinc}^{-1}(y)$ is, as we were with $\text{sinc}(x)$. After a substantial search, we finally found the formula used in the main body of the *asinc* function on a math forum.

Some optimizations that were used in the implementation of the two force calculation methods (stretching and compression) were that global variables were used instead of local variables, and pointers and pointer arithmetic replaced normal array addressing. If the reader would like to see the code of these functions, please refer to the source code on the accompanying CD in the file “ParticleSystem.cpp” and the methods *Type1Int() and Type2Int()*.

```
// for y in range -0.3 to 1
double asinc (double y)
{
   static bool initialized = false;
   static double table [MAX_SINC_RES];
   if (!initialized)
   {
       for (int i = 0; i < MAX_SINC_RES; i++)
       {
           // number in range -0.3 to 1
           double x = (1.3 * (double) i / (double) MAX_SINC_RES) - 0.3;
           double f = 2*x + 3*x*x*x/10 + 321*x*x*x*x*x/2800 +
                      3197*x*x*x*x*x*x*x/56000 +
                      445617*x*x*x*x*x*x*x*x*x/13798400 +
                      1766784699*x*x*x*x*x*x*x*x*x*x*x/89689600000 +
                      317184685563*x*x*x*x*x*x*x*x*x*x*x*x*x/25113088000000 +
                      14328608561991*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x/
                      1707689984000000 + 6670995251837391*x*x*x*x*x*x*
                      x*x*x*x*x*x*x*x*x*x*x/1165411287040000000;
           table[i] = f;
       }
       initialized = true;
   }

   int index = (int) (MAX_SINC_RES * ((y + 0.3) / 1.3));

   if ((index < 0) || (index >= MAX_SINC_RES))
   {
       return 0.0;
   }
   return table[index];
}
```

Figure 4.6: *asinc* Function Definition

```
void ParticleSystem::Verlet()
{
    // Move particles according to their acceleration and velocity
    for (int i = 0; i < NUM_PARTICLES; i++)
    {
       CVector3 &x = Parts[i].Pos;
       CVector3 temp = x;
       Vector3 &oldx = Parts[i].OldPos;
       CVector3 &a = Parts[i].Force;

       x = x + x - oldx + a * m_fTimeStep * m_fTimeStep;
       oldx = temp;
    } // for i
}// Verlet
```

Figure 4.7: Verlet Integration

4.3 Explicit Integration

The explicit integration implementation is relatively simple, following the guidelines set out in section 3.2.1 and by Jakobsen [Jakobsen 2001]. The Verlet integration function is listed in figure 4.7 and works as follows:

- We cycle through each particle,
- calculate the particle's new position,
- assign the current position to the old position
- and assign the new position to the current position.

This effectively moves the particles depending upon their current position, old position and the force being exerted on the particle. There were no real problems in the implementation of this integration method that the reader needs to be aware of.

4.4 Implicit Integration

As seen in section 3.2.2, the implicit integration algorithm makes use of matrices to calculate the next position of the particles. Matrix calculations tend to be relatively slow since if there are n particles in the system, some of the matrices used in the calculation end up being $n \times n$ sized matrices or more likely $3n \times 3n$ because each particle has 3 values that make up each of its vector properties (such as its position).

Due to this fact, we attempted to circumvent the matrix calculations by adjusting the integration calculations to look more like the calculations performed in the explicit calculations from section 3.2.1, IE: iterating through each particle and performing the necessary calculations as opposed to assigning the values from each particle into an array and doing it all at once.

In order to do this, we had to approximate the partial differentials $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ which are matrix based. These partial differentials were approximated using equation 4.1 and equation 4.2 respectively.

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \approx \frac{\mathbf{f}_n - \mathbf{f}_{n-1}}{\mathbf{x}_n - \mathbf{x}_{n-1}} \tag{4.1}$$

$$\frac{\partial \mathbf{f}}{\partial \mathbf{v}} \approx \frac{\mathbf{f}_n - \mathbf{f}_{n-1}}{\mathbf{v}_n - \mathbf{v}_{n-1}} \tag{4.2}$$

Using these approximated partial differentials, we used the same equations from Choi et al. [Choi and Ko 2002], equation 3.16 and equation 3.17, for the position and velocity calculations which were altered slightly. Instead of using the matrices that are used in the original calculations, we performed the position calculation three times - once for each component of the position vector, and the velocity calculation three times - once for each component of the velocity vector. There were also checks put in place to prevent divide-by-zero errors and other out-of-bounds situations.

Although this produced a system which was only marginally slower than the explicit solution, the system was extremely unstable, even with a very small time step and a few particles. This was obviously not acceptable considering the claims by Choi et al. that their system was extremely stable with large time steps and high numbers of particles.

The solution was converted to include matrices, so had to be changed a great deal. One of the more interesting challenges included creating and filling the matrices for the partial differentials. An $n \times n$ matrix was needed for the system, where n is the number of particles. However, each particle had three values representing each component, which would not fit into an $n \times n$ matrix. The solution for this was to use a 3×3 matrix to represent each particle - $(x, y, z) \times (x, y, z)$. These 3×3 matrices were then inserted into a larger matrix which was $m \times m$ where m is the number of matrices (which was equal to the number of particles). This effectively provided a matrix of size $3n \times 3n$.

In the implementation, only the partial differential from the stretching interaction was used to make the solution easier to understand. It was discovered from tests with the explicit method, that removing the compression interaction's contribution to the partial differential only created small differences in the movement of the cloth - IE: when the cloth is forced in on itself, there is no resistance to cloth collapsing in on itself. The absence of this resistance is not particularly dire and has very little effect on the stability of the system.

In the calculation of the partial differential, if the particles in question are not connected, a zeroed 3×3 matrix is inserted into the larger partial differential matrix as can be seen from the partial listing of the implicit integration method in figure 4.8 which shows the calculation of the partial differentials. This is in keeping with the springs that were set up between particles in the physical model. We are only interested in particles that have some sort of relationship. It is here that the unique identifier that was specified in the *Particle* class in figure 4.2 is useful. We can test between the UID and the particle retrieved from the current particle's connection list to see if the particle is valid. The *IDExists()* method in the listing does this and returns the index of the connected particle in the current particle's connection list if the two particles are connected or -1 if they are not.

This index can then be used to access the correct linked particle or bypass the calculation if it is -1. If the calculation is bypassed, the matrix that is inserted into the larger partial differential matrix is zeroed, complying with the regulations set out earlier regarding unconnected particles.

The equation to calculate the partial differential for stretching interaction is given again below:

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} = \begin{cases} k_s \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}} + k_s(1 - \frac{L}{|\mathbf{x}_{ij}|})(\mathbf{I} - \frac{\mathbf{x}_{ij}\mathbf{x}_{ij}^T}{\mathbf{x}_{ij}^T\mathbf{x}_{ij}}) & : \quad |\mathbf{x}_{ij}| \geq L \\ 0 & : \quad |\mathbf{x}_{ij}| < L \end{cases}$$

As can be seen from the above equation, matrices and their transposes are used. There are usually plenty of function full matrix classes around, however, usually the matrices used in calculations are either 3×3 or 4×4. They are not usually very flexible, which is exactly what is needed for this simulation. One could write their own matrix class, but many functions are needed such as inversions, which are not easy to write. Because of this and the time constraints placed on this project, we chose to use a pre-made matrix class with all the functionality that was needed. After searching through a number of different classes, one was found that provided us with all the functionality required. The speed and efficiency of the class has been brought into question, however.

```
int index = -1;
//Calculate and Populate Partial Differential Matrices
for (int i = 0; i < NUM_PARTICLES; i++)
{
   for (int j = 0; j < NUM_PARTICLES; j++)
   {
     // If the PartID exists in the current particles interaction list
     index = IDExists(Parts[i].Svs, Parts[i].SvsSize, j);

     if (index != -1)
     {
       //Calculate the partial diff for Type 1 interaction
       float length = 0, rest = 0;
       tempvec = Parts[j].Pos - Parts[i].Pos;
       length = tempvec.Magnitude();
       rest = Parts[i].Svs[index].RestLength;

       // using equation (3)
       if (length >= rest)
       {
         Xij(1,1) = tempvec.x;
         Xij(2,1) = tempvec.y;
         Xij(3,1) = tempvec.z;
         val = ((Xij.t())*Xij);
         scalarval = val.AsScalar();
         Matrix xxt = (Xij*(Xij.t())) / scalarval;
         Matrix ixxt = Identity33 - xxt;

         Matrix testy = (((double)Ks*(1.0 - (rest/length)))*(ixxt));
         typeonediff = (((double)Ks) * ( xxt )) + testy;
       }
       else
       {
         typeonediff = 0.0;
       }

     } // if (index != 1)
     else
     {
       //Put zeroed 3x3 matrix into larger matrix
       typeonediff = 0.0;
     }

     //assign values into larger partial diff matrix
     for (int loopi = 1; loopi <= 3; loopi++)
     {
       for (int loopj = 1; loopj <= 3; loopj++)
       {
         m_partDiffx(3*i+loopi,3*j+loopj) = typeonediff(loopi,loopj);
       }
     }
   }//for j
}//for i

//Partial differential matrix for the velocity term
m_partDiffv = Identity3n3n * Kd;
```

Figure 4.8: Implicit Partial Differentials

The transpose functionality can be seen in the listing in figure 4.8 being used on the matrix *Xij* as *Xij.t()*. The rest of the calculations in the listing are simply an implementation of the above formula. After the 3×3 matrix has been calculated, it is assigned (component by component) into the $3n \times 3n$ partial differential matrix.

The $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ partial differential was defined earlier as

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{v}_j} = k_d \mathbf{I}$$

The matrix that was used to represent this partial differential is a 3×3 matrix. Its initialization can be seen at the end of the listing, where a 3×3 identity matrix is multiplied by our predefined damping constant *Kd*.

If the reader would like to see how the rest of the implicit integration was implemented, please refer to the source code on the CD in the file "ParticleSystem.cpp" in the method *Implicit()*.

4.5 Final Implementation

The following are just some examples of what was eventually produced and the stages that were traversed to get there. Figure 4.9 shows a very basic example of the particles representing the cloth, which is then extended in figure 4.10 to have a higher number of particles in the system. In figure 4.11, figure 4.12, and figure 4.13, the cloth has been textured and is being anchored in different positions. In figure 4.11, the cloth is being held by the top two corners, in figure 4.12 the cloth is held at four points slightly in from the corners, and in figure 4.13, it is being held by the center point. These positions were chosen arbitrarily to test the cloth's responsiveness to different constraints. In figure 4.14, the cloth has been moved from side to side in a whip-like action to demonstrate its motion further. The executable for this system has been provided on the accompanying CD.

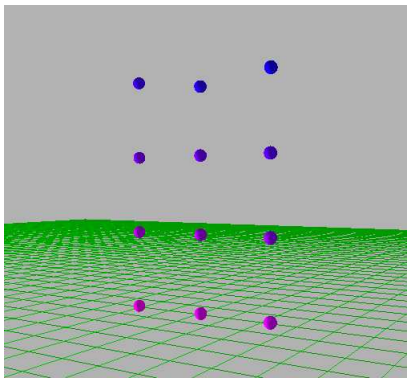

Figure 4.9: Example Cloth Simulation Image 1

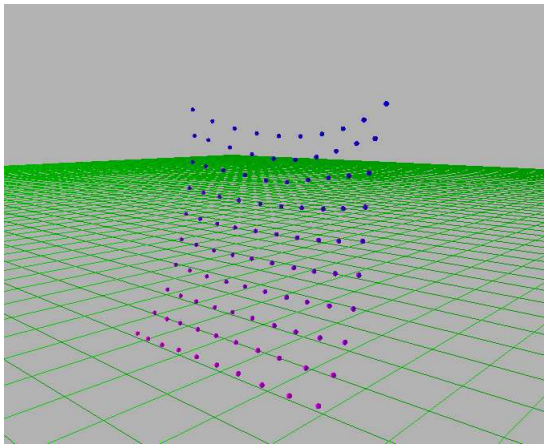

Figure 4.10: Example Cloth Simulation Image 2

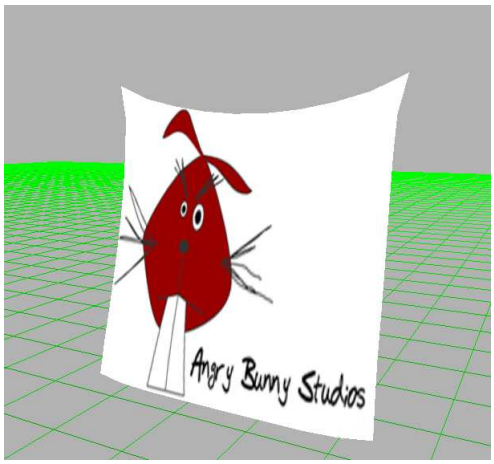

Figure 4.11: Example Cloth Simulation Image 3

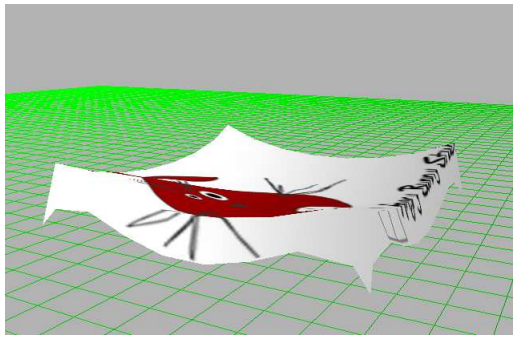

Figure 4.12: Example Cloth Simulation Image 4

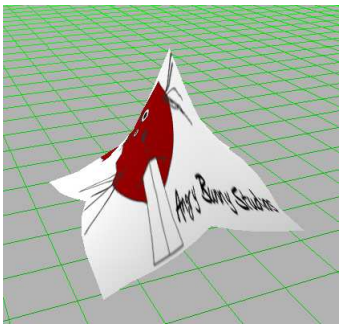

Figure 4.13: Example Cloth Simulation Image 5

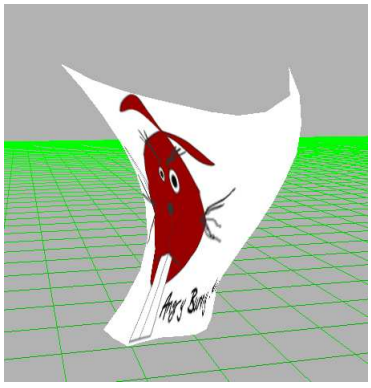

Figure 4.14: Example Cloth Simulation Image 6

Chapter 5

Results

The results that follow are those of the tests done for the explicit or Verlet integration and the implicit integration.

For testing the integration methods, a machine with the following hardware is used: a 1.70GHz Intel Pentium 4 CPU, 512MB of RAM, and a nVIDIA Quadro 4 XGL graphics card with 128MB DDR SDRAM. This testing machine is running Windows XP and the test program uses OpenGL as the graphics API.

The grid size refers to the number of particles in the system (IE: a 5 particle by 5 particle square grid); the FPS (frames per second) is the number of frames that were able to be rendered per second; and the stable time step is the highest value that could be used as the time step while still keeping the system stable.

The results of the two methods cannot be precisely compared, because one method and its data structures may be more optimized than another. In addition FPS is not a completely accurate measure of the speed of the method, one is still able to get a general idea of the efficiency of an algorithm by examining the results.

5.1 Explicit Integration

Table 5.1 and Figure 5.1 contain the results of the tests performed on the explicit or Verlet integration method.

As one can see from figure 5.1, as the number of particles in the system increases, the frames

Grid Size	FPS (Frames/Second)	Stable Time Step
5x5	85	8.5
7x7	85	7.5
10x10	85	7.0
15x15	85	7.0
30x30	83	6.5
50x50	38	6.0
100x100	10	5.5
500x500	0.5	4.0

Table 5.1: Explicit Integration Results Table

per second (FPS) stays the same until the point at which the video card is no longer the bottle neck of the application; at which point the CPU starts to take strain and the FPS decrease rapidly as the work continues to increase. The stable time step value decreases almost linearly as the number of particles increase.

For an example of the instability of the system, using the implementation provided on the accompanying CD, if the particles are stretched far enough apart the system explodes.

5.2 Implicit Integration

Our implementation of the implicit integration method was much slower than the explicit method which was expected, but what was not expected was the lack of stability in the system.

Choi et al. claim in their paper [Choi and Ko 2002] that their implicit method is highly stable, but in our implementation, this was not the case at all. The produced system was stable at small time steps, but quickly became unstable as the time step was increased. This defeats the point of using the implicit method over the explicit method, as there is no stability gain for the performance loss. This technique and variations of it have been found in a number of papers and all have claimed that it is stable, which leads us to assume that the results we obtained using our method were incorrect. We found that the system was barely stable at a grid size of 10x10 and a time step of 0.5, and it was only able to deliver 1.5FPS, which is much lower than the explicit method and a very small stablity range in comparison.

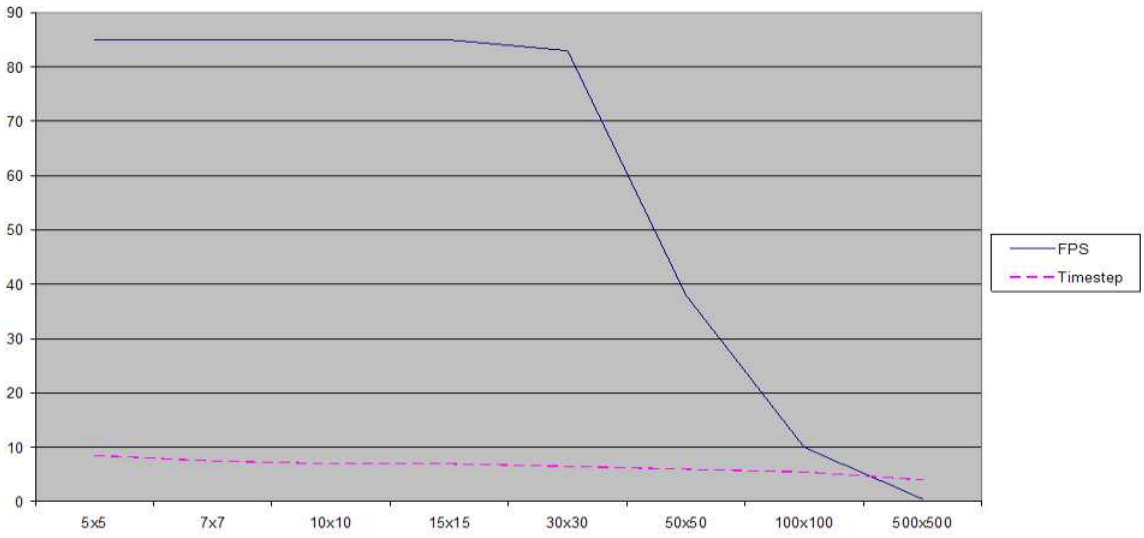

Figure 5.1: Explicit Integration Results

Chapter 6

Conclusion

As was mentioned in section 2.2.1, the explicit Verlet method has been used in the game industry already and has proved to be successful. Our findings show that this method is indeed suited to a real-time application as long as the number of particles in the system are kept low and the time step is kept small, which is what game developers try to achieve regardless in order to optimize rendering speeds.

Our implementation of the implicit integration technique presented by Choi et al. [Choi and Ko 2002] did not perform as expected and as Choi et al. claimed it would. It is unknown why this method is so unstable, and investigation is required to discover the flaw. If we were to assume that the method does deliver what Choi et al. claim, it would be perfectly suited to an offline application such as animation rendering, because it is able to handle large time steps and large numbers of particles while still remaining stable and accurate, which is exactly what is required in such an application.

References

[Amirbayat and Hearle 1989] Amirbayat, J. and J. Hearle (1989). The anatomy of buckling of textile fabrics: Drape and conformability. *Journal of Textile Institute 80*, 1 (1989), pp. 51-70.

[Baraff and Witkin 1998] Baraff, David and A. Witkin (1998). Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM Press, pp. 43-54.

[Breen et al. 1994] Breen, D. E., D. H. House and M. J. Wozny (1994). Predicting the drape of woven cloth using interacting particles. *Proceedings of SIGGRAPH 1994* (1994), pp. 365-372.

[Choi and Ko 2002] Choi, Kwang-Jin and H.-S. Ko (2002). Stable but responsive cloth. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (San Antonio, Texas, 2002), San Antonio, Texas, ACM Press, pp. 604-611.

[Eischen et al. 1996] Eischen, J. W., S. Deng and T. G. Clapp (1996). Finite-element modelling and control of flexible fabric parts. *IEEE Computer Graphics and Applications 16*, 5 (1996), pp. 71-80.

[Etzmuß et al. 2001] Etzmuß, Olaf, M. Hauth, M. Keckeisen, S. Kimmerle, J. Mezger and M. Wacker (2001). A cloth modelling system for animated

characters. Tech. rep., Wilhelm-Schickard-Institut für Informatik, Graphisch-Interaktive Systeme, Universität Tü, 2001.

[Jakobsen 2001] Jakobsen, Thomas (2001). Advanced character physics. In *Proceedings of Game Developer's Conference* (San Jose, 2001), San Jose.

[Oshita and Makinouchi 2001] Oshita, M. and A. Makinouchi (2001). Real-time cloth simulation with sparse particles and curved faces. In *Proceedings of Computer Animation 2001* (November 2001), pp. 220-227.

[Rudomin and Castillo 2002] Rudomin, Isaac and J. L. Castillo (2002). Realtime clothing: geometry and physics, 2002, <citeseer.ist.psu.edu/rudomin02realtime.html>.