Cloth Simulation

Elaine Kieran (d1146498)

Gavin Harrison (d1141913)

Luke Openshaw (d1159746)

MSc Computer Animation

NCCA Bournemouth University

Chapter 1 Abstract

This report documents the analysis, design, implementation and results of simulating a clothed computer generated model driven by motion capture data as well as a variety of cloth simulation tests of interaction with static objects. A framework has been implemented for all results to be exported to RIB files for rendering in a RenderMan compliant renderer.

Chapter 2 Contents

Chapter 3 Introduction

When approaching cloth simulation in computer graphics, it is worth noting that the phenomena under consideration are fundamentally chaotic. Each time one puts on a skirt or drapes a table cloth, many of its details look different. It is for this reason that when implementing a cloth simulation for computer graphics, the general aim is to get it to “look right” rather than adhere rigidly to engineering principles and physical properties of cloth.

While the simulation outlined in this report was designed with this in mind, many of the methods used are also physically accurate. The fundamental aim was to get a general cloth model looking as “realistic” as possible, whilst also interacting with its surrounding environment.

Research into previously implemented methods shows the first important step is to decide upon the desired outcome of the simulation. As the following Chapters will explain, a variety of available methods were tested and examined until the most suitable was chosen.

In this way the simulation was built up using the appropriate mixture of physically correct algorithms, and problem specific solutions.

Chapter 4 History of cloth simulation

The study of cloth and its simulation began in the 1930's in the textile industry, when a paper entitled "On the geometry of cloth structure" was published [1]. The computer graphics community took an interest in the 1980's with the publication of several papers on different methods of simulating cloth using computer graphics [3]. Today, both industries approach the same subject, but from different angles. The textile community are concerned with the physical properties of cloth, and the computer graphics community are concerned with creating visibly believable simulations with "realistic" computation times.

The computer graphics approach to cloth modelling has mainly focused on the issue of simulating the complex shapes and deformations of fabric and clothing in 3D. The techniques that have been formulated and studied can be broken up into town main sections; Geometric Techniques and Physically Based Approaches.

4.1 Geometric Techniques

In 1986, a geometric method for modelling cloth hanging from a fixed number of points was introduced by Weil [4]. For this method the cloth is modelled as a 2D grid of 3D points and calculated using a two step process:

1. The points are connected recursively with Catenary curves, and are dealt with depending on whether they hang inside or outside a convex hull. The points that do not hang inside the hull are removed.
2. The second step is a relaxation pass. This pass is to ensure the distance constraints set between the particles are adhered to. This will give a smooth cloth with a realistic cloth-like drape.

Later, other geometric approaches to modelling cloth–like surfaces began to emerge. In the 1990's a number of papers were published on the topic [3].

One such technique was a mixture of geometric and physically based methods. It involved a computational geometry technique to roughly estimate the shape a piece of cloth would have, when hanging from specific constrained points [5].

A wrinkle model was also introduced, again using a mixture of geometric and physically based methods [8]. This technique involved performing dynamic analysis on a small deformable sheet of material and using this analysis to identify characteristic deformations for wrinkling cloth. This information was then used to define a model for cloth. The method proved useful for providing cloth–like wrinkles for use in animations; however it did not prove to be accurate enough for modelling wrinkling cloth.

4.2 Physically Based Approaches

There are three main types of physically based models that have been developed for cloth simulation:

1. Elasticity Based Methods
2. Particle Based Methods
3. Mass – Spring Damper Model

4.3 Elasticity Based Methods

In 1986, along with Weil's geometric approach, another technique for modelling cloth simulation was introduced [3]. This was an Elasticity Based Model, which defined a set of energy functions over a 2D grid of 3D points. The energy contained in this model included the cloth's tensile strain, bending and gravity information. The method is based on the idea of treating cloth as a continuous material, with these energy functions being derived from Elastic Theory [3]. The distance between points and a simple measure of the curvature of this distance are used to calculate the elastic forces in the cloth.

4.4 Particle Based Methods

In 1992, Particle Based Methods were introduced by Breen & House [6]. They had developed a model for cloth drape simulation using an interacting particle system that represented the underlying mechanical structure of cloth. The particles interacted with their adjacent particles and the surrounding environment, by using equations that described associated mechanical connections, represented by energy functions. Further to this, a stochastic gradient descent technique [10] was used as a relaxation method to bring the cloth particles to a stable rest state.

This theory for modelling cloth using Particle Based Methods was built on further, by Eberhardt and Weber [7]. They reformulated the basic energy equations resulting in a set of Ordinary Differential Equations, which were then used to calculate the cloths dynamic behaviour.

4.5 Mass Spring Damper Model

This technique was first introduced in 1988 by Hamann and Parent [11, 3] and was developed further in 1995 by Provot [12]. It consists of cloth being modelled as a grid of particles that are connected by spring dampers.

This technique was used as the basis for the cloth simulation discussed in this report and is developed further in the following section.

Chapter 5 Cloth Model

5.1 Mass Spring Model

The cloth model is represented by a grid of particles of known mass, connected by a series of spring-dampers. There are three main types of spring which relate to the characteristics of cloth:

1. Structural springs: Handle extension and compression and are connected vertically and horizontally.
2. Shear springs: Handle shear stresses and are connected diagonally.
3. Bend springs: Handle bending stresses and are connected vertically and horizontally to every other particle.

The springs are the structural elements of the model and resist the various loads that are applied to the particles. Figure 5.1.1 below illustrates the three types of springs and how they are connected to the particles.

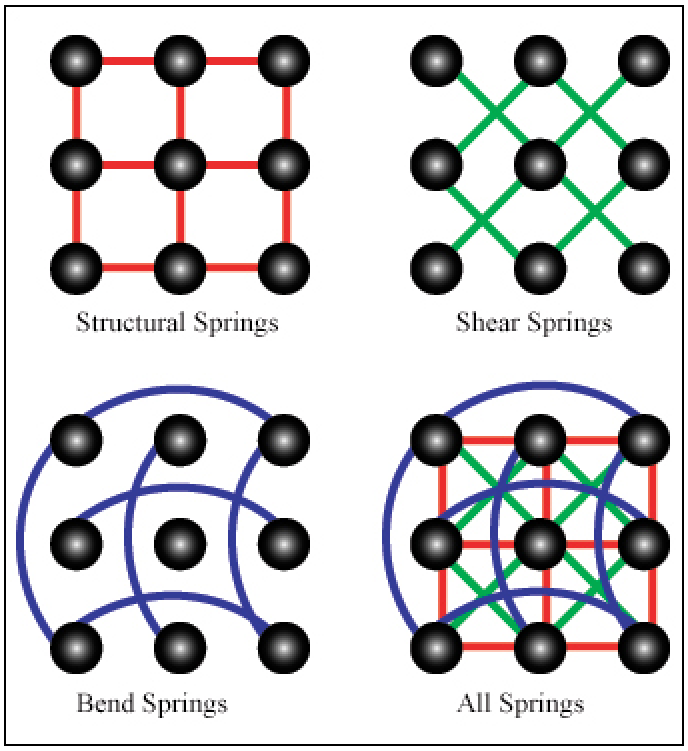


Figure 5.1.1: Three types of springs

When the simulation is initiated, each spring's rest length is set to the original length of the spring. The mass values of the particles and the spring constants are defined by the user. Additional information on simulation initialization is given in Section 7.3.

Once the springs and particles are set up, the simulation is incrementally advanced by integrating Newton's Second Law of Motion. Additional information on integration and Newton's Second Law of Motion is given in Chapter 7 and Chapter 8.

When an environmental force, for example gravity, is applied to the particles in the model over a specified time step, it produces a resulting acceleration for each particle. This acceleration gives rise to a velocity which causes the particle to update its position. The new position of each particle in turn causes a change in length to each connected spring-damper.

Hooke's Law states that the extension of a spring is proportional to the applied force. By applying Hooke's Law with the addition of damping calculations to reduce oscillations, a new force due to the springs can be calculated and applied to the simulation. The combination of the particle and spring forces is integrated with respect to time to provide a new acceleration for each particle. The iterative process of calculating forces and updating positions provides the motion of the cloth object. Additional information on Hooke's Law and the spring-damper force calculations is given in Chapter 7.

Chapter 6 Design Overview

The cloth simulation application can be broken down into elements that define the projects scope. An application manager is in charge of managing the cloth simulation, collision detection and response, motion capture data, and OpenGL specific functionality such as cameras and drawing.

The Cloth class, the FBX class and the Obj Loader class form the core structure of all physical objects in the simulation, instances of which are created from the application manager. In order for physically accurate simulations to occur, all of these classes rely on the Axis-Aligned Bounding Box (AABB) Tree class. The AABB Tree class has the responsibility of encapsulating any instances of the physical object classes in a hierarchy of Bounding Boxes for localized collision detection algorithms to be called. The AABB class therefore creates many instances of the Bounding Box class during the encapsulation process and then utilises these bounding box objects during any collision detection sweeps.

The Cloth class encapsulates the two types of cloth to be simulated in the form of a Skirt class and a Sheet class which inherit the Base Cloth class. These classes rely on the cloth Solver class to calculate the cloth's internal forces and the RenderMan Interface Bytestream (RIB) class for outputting RIB files for rendering in RenderMan.

In regards to collision response, a Response class contains all the functions and attributes required to deal with any individual collision. An instance of the Response class is created for every detected collision allowing only collided particles to be processed for collision response.

Finally, a Camera class is used to allow cameras to be instanced for the purposes of rendering out to RIB files. Instances of this class are the responsibility of the application manager.

Refer to the class diagram in Appendix A for a visual representation of the program design.

Chapter 7 The Cloth Class

7.1 Introduction

During the initial design phase of the project, an early requirement became apparent of the need for two simulation environments:

1. Testing (Sheet)
    a. Simple rectangular cloth
    b. Simple collision object
    c. Ability to control movement of cloth and collision objects
    d. Allow for simple unit tests of key methods
    e. Fast simulation processing and visualisation of results
2. Full (Skirt)
    a. More complex skirt cloth model
    b. Complex collision object
    c. Movement of collision object controlled by FBX data
    d. Allow for unit tests of key methods
    e. Slower simulation processing and visualisation of results

Although the two environments produce different simulations, it was important to maintain one set of the core calculation and response classes.

7.2 Base Class

An important feature of C++ is its support of object-oriented programming design. To minimise the size and complexity of the code base, a parent cloth class was defined. This parent class contains the main attributes and a set of pure virtual methods that each of the child classes has to implement. This application architecture also improves the potential extensibility of additional cloth objects.

7.3 Cloth Initialization

7.3.1 Sheet Class

Cloth initialization for the Sheet class creates a rectangular grid of p*articles* and assigns initial position and force values for each p*article*. All of the p*articles* are contained in a multidimensional vector variable.

It also creates the various types of s*prings* by assigning parameters to each s*pring* and indicating which two *particles* the s*pring* connects. All of the s*prings* are contained in a vector variable. Initialization finally creates the *polygon* list for the collision detection and display. Figure 7.3.1 below, shows an OpenGL view of the Cloth sheet object. Additional images of the OpenGL Cloth sheet can be found in Appendix B.

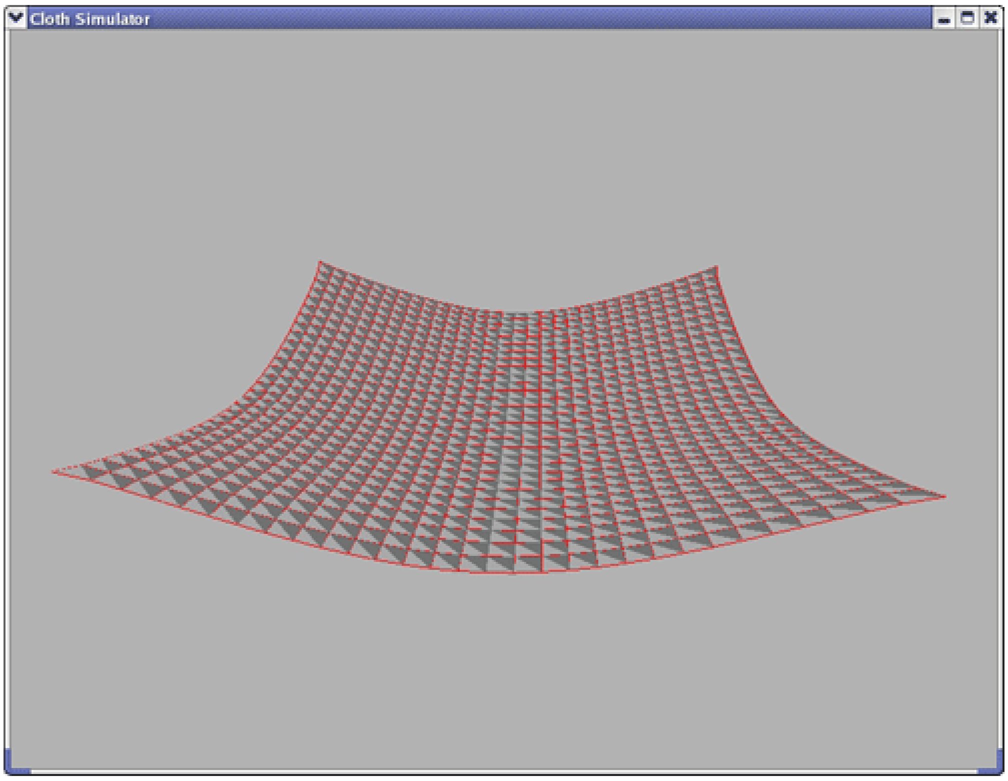


Figure 7.3.1: OpenGL view of Cloth Sheet

7.3.2 Skirt Class

Cloth initialization for the Skirt class creates a circular grid of *particles* and assigns initial position and force values for each *particle*. It also creates the various *springs* by assigning parameters to each *spring* and indicating which two *particles* the *spring* connects. Although the methodology is basically the same for the sheet and skirt, the algorithm for creating the grid of *particles* is more complex for the skirt. This is due to having to define the circular positional information and connecting two edges for the definition of the *springs* and *polygons*. Figure 7.3.2 below, shows an OpenGL view of the Cloth skirt. Additional images of the OpenGL Cloth skirt can be found in Appendix C.

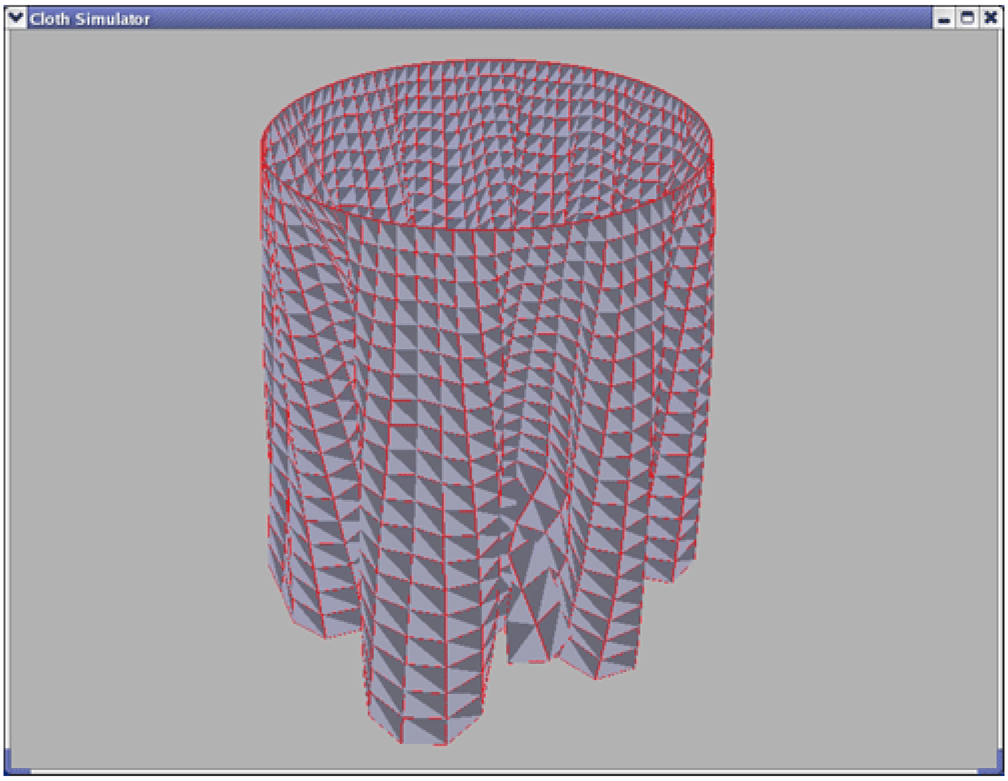


Figure 7.3.2: OpenGL view of Cloth skirt

7.4 Increment Simulation

The increment simulation method calls the relevant force calculation and integration methods to update the position and force values for each *particle* and *spring*. It also calls methods to recalculate the *polygons'* normal and centroid values. Additional information on the integration methods is given in Chapter 8.

7.5 Force Calculations

The force calculation method starts by setting all the force values to zero. Gravity is then applied to each particle using the following formula:

$$Force_{Gravity} = Mass \times Acceleration$$
$$Acceleration = -9.8ms^{2}$$

Equation 7.5.1: Force due to Gravity

The final calculation for the force total is provided by the *springs*. The *spring* force is defined by the following formula [1]:

Hooke's Law states:

$$Force_{Spring} = -k_s\ x$$

k_s = Spring constant

x = Distance

The negative sign indicates that the spring is a restoring force.

A damping force is required for realistic numerical simulation to ensure the springs do not oscillate forever:

$$Force_{Damping} = -k_d v$$

k_d = Damping constant

v = Velocity

A spring and damper force can be combined into one equation for a spring-damper connecting two particles:

$$Force = -k_s(|L| - L_0)\left(\frac{L}{|L|}\right) - k_d(v_1 - v_2)$$

|L| = Length between two Particles

L = Length vector

L_0 = Rest length

v_1 and v_2 = Velocities of Particle 1 and Particle 2 respectively

Equation 7.5.2: Hooke's Law

Applying this formula gives the force value on p*article 1*. This value is simply negated to find the force for *particle 2*.

7.6 Collision Check

The collision checking method simply creates the Axis Aligned Bounding Box (AABB) tree and runs the sweep nodes method within the AABB class. Additional information on Axis Aligned Bounding Boxes is given Chapter 11.

7.7 Cloth Object Display

The display method checks the various flags and displays the relevant data:

- The *cloth* object is displayed as a series of OpenGL triangles

- The *polygon* normals are displayed as OpenGL lines located at the centroid of each *polygon*.
- The *particles* call their respective draw methods
- The *springs* call their respective draw methods.

Additional information on OpenGL display is given in Chapter 14.

7.8 Writing RIB Files

The two writing to RIB methods implement different RenderMan packages:

1. Patches: Iterates through all the *particles* and writes out RenderMan bicubic patches to the RIB file.
2. Subdivs: Iterates though all the *polygons* and writes out a RenderMan subdivision surface to the RIB file.

Additional information on RenderMan is given in Chapter 15.

7.9 Particle Class

The Particle class encapsulates a *particle* object. A *particle* is one of the two main building blocks for the simulation. It has a large number of attributes that store mainly positional, force and flag data. The Particle class only has one method, the draw method, which simply creates an OpenGL point at the relevant coordinates to represent a *particle*.

7.10 Spring Class

The Spring class encapsulates a *spring* object. A *spring* is one of the two main building blocks for the simulation. It has a number of attributes that store distance, property data and pointers to the two *particles* that the *spring* attaches to. The Spring class only has one method, the draw method, which simply creates an OpenGL line between the two referenced *particles* to represent a *spring*. The colour of the line is related to the *spring* type.

Chapter 8 Explicit Integration

Numerical integration is the approximate computation of an integral, using numerical techniques. Integrals, along with derivatives, are the fundamental mathematical objects of calculus [14]. There are several standard integration techniques for solving Ordinary Differential Equations (ODE's) [15]. These ODE's are differential equations in which all dependent variables are functions of a *single* independent variable [15]. According to Newton's laws of motion, any motion of a collection of rigid bodies can be described using a set of 2nd order ODE's, with time being the independent variable.

There are several methods available for implementing cloth simulation systems, but most are based on a system of particle objects linked by springs. It is the motion of these particle objects that is described by the ODE's. The evolution of this model over time is computed using these equations [16].

The various methods of integration for solving these ODE's have varying strengths and weaknesses. As a result, the appropriate method must be implemented to suit the requirements of the system.

As the ODE's describing the cloth model are dependent on time, the integration was used to update the position of the cloth's particles over time [17]. Integration generally involves finding the function from which the derivative function was obtained, and hence a numerical approximation to the function [17].

For example, the acceleration is obtainable using Newton's second law:

$$\text{Force} = \text{Mass} \times \text{Acceleration}$$

Equation 7.10.1: Newton's second law

Performing integration on this ODE will result in the velocity – which itself is a derivative function. Performing integration on the velocity will result in position. Thus:

- Acceleration is the rate of change of velocity over time

- Velocity is the rate of change of position over time

All cloth simulation integration is based on the notion of initial value problems – the starting point is one at which conditions are known at that point in time, and the simulation of this system is forward in nature, through all the time steps.

There are various levels of integration methods available for solving these forward initial value problems:

8.1 Forwards Euler (Explicit)

The Euler algorithm for forwards integration is possibly the simplest method of integrating an ODE. It starts with the original equation, with the forces from time step $t_{(n)}$ contributing to the velocities at time step $t_{(n+1)}$. It is relatively trivial to implement.

$$v_{(n+1)} = v_{(n)} + F_{(n)} \frac{dt}{m}$$

$$x_{(n+1)} = x_{(n)} + v_{(n)} dt$$

where:

$v_{(n)}$ = Current velocity

$x_{(n)}$ = Current position

$v_{(n+1)}$ = New velocity

$x_{(n+1)}$ = New position

$F_{(n)}$ = Current force

dt = Time step

m = Mass

Equation 8.1.1: Explicit Euler

The Euler method is not symmetric. It uses information at the beginning of the time step to obtain the desired information at the end of the time step. This is illustrated in Figure 8.1.1.

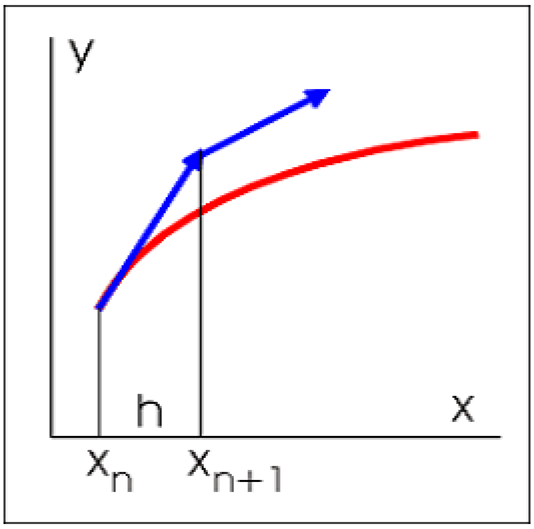


Figure 8.1.1: Explicit Euler function approximation

In the cloth model, the acceleration is the first derivate at the beginning of the time step and results in a vector. This vector is the tangent to the curve that represents the function at that point in time.

Euler does have stability problems. The method is not symmetrical, so if large time steps are used, the estimated path will deviate greatly from the actual path and the simulation will become unstable very quickly. For this reason, Euler requires the caveat of small time steps.

The tangent from the beginning is used to approximate the behaviour of the function over the entire time step; hence Euler's method has a high degree of error.

8.2 Runge-Kutta

Runge-Kutta methods are described by Conte & de Boor as, "methods to obtain greater accuracy than Euler's method, and at the same time avoid the need for higher derivatives, by evaluating the function f(x, t) at selected points on each sub-interval" [7].

These methods are based on taking intervals during the time step to approximate a more accurate answer. Runge-Kutta integration is an extension of Euler integration, and improves the accuracy due to its symmetry with respect to the time interval.

They use the Euler method to get started, with each level of Runge-Kutta performing an Euler step with new values per iteration. The method uses the results from the first Euler integration to calculate the new derivative in between the initial and final times [19].

Runge-Kutta methods are all forward explicit methods that use approximations to take “educated guesses” towards the end point. The benefit of using the Runge-Kutta method is that it allows for larger time steps, however it still does not remove Euler’s instability, it simply reduces it. These methods are also computationally more expensive [3].

8.3 2nd Order Runge-Kutta (RK2)

RK2 is essentially the same as Euler, but considers the derivative at more points in time by making an Euler trial step to the midpoint of the time interval. It then uses the values of (`X, Y`) at the midpoint to make the real step across the time interval [20]. The general form of RK2 is shown in Equation 8.3.1.

$$\begin{aligned} y_{n+1} &\approx y_n + k_2 \\ k_2 &= hf\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\ k_1 &= hf(t_n, y_n) \end{aligned}$$

Equation 8.3.1: General form of RK2

This method takes one sample - the midpoint between the start and end of the time step.

For the cloth simulation, the forces are computed at the start, using the current position and velocity, for half the time step. These values for position and velocity are then stored as temporary mid-point values in the *particle* class. The forces on each *particle* are re-computed using these mid-point values, and the final position and velocity is taken to be a weighted average of these.

The method is more accurate and stable than the Euler method - it is symmetrical and the approximated path will be closer to the actual function path than Euler. However small time steps must still be taken. Figure 8.3.1 below, shows a graphical representation of RK2.

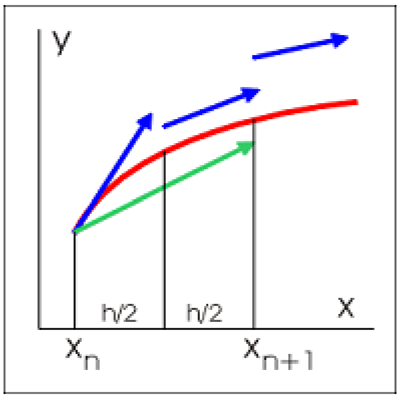
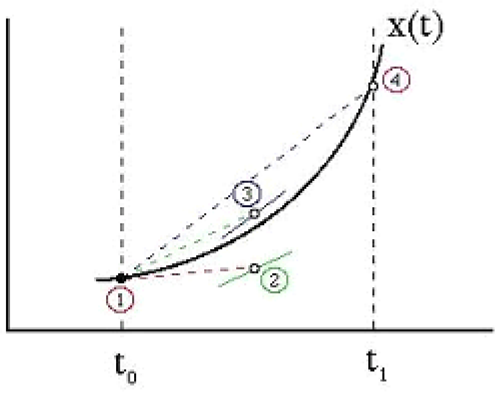


Figure 8.3.1: RK2 function approximation

8.4 4th Order Runge-Kutta (RK4)

RK4 is described as 4th order, because it takes more intervals in between time steps. The process of RK4 is essentially the same as RK2, but uses an expanded Taylor Series approximation. The simulation can use a larger overall time step because the approximation is more accurate. Figure 8.4.1 below, shows the general form of RK4.



$$\begin{aligned}
dx1 &= \Delta t\, v_{x,n} \\
dv_x1 &= \Delta t\, a_x(x_n, y_n, t) \\
dx2 &= \Delta t\, (v_{x,n} + \frac{dv_x1}{2}) \\
dv_x2 &= \Delta t\, a_x(x_n + \frac{dx1}{2}, y_n + \frac{dy1}{2}, t + \frac{\Delta t}{2}) \\
dx3 &= \Delta t\, (v_{x,n} + \frac{dv_x2}{2}) \\
dv_x3 &= \Delta t\, a_x(x_n + \frac{dx2}{2}, y_n + \frac{dy2}{2}, t + \frac{\Delta t}{2}) \\
dx4 &= \Delta t\, (v_{x,n} + dv_x3) \\
dv_x4 &= \Delta t\, a_x(x_n + dx3, y_n + dy3, t + \Delta t) \\
x_{n+1} &= y_n + \frac{dx1}{6} + \frac{dx2}{3} + \frac{dx3}{3} + \frac{dx4}{6} \\
v_{x,n+1} &= v_{x,n} + \frac{dv_x1}{6} + \frac{dv_x2}{3} + \frac{dv_x3}{3} + \frac{dv_x4}{4}
\end{aligned}$$

Figure 8.4.1: General form of RK4

Figure 8.4.2 below, shows the flow chart for the integration process within the application.

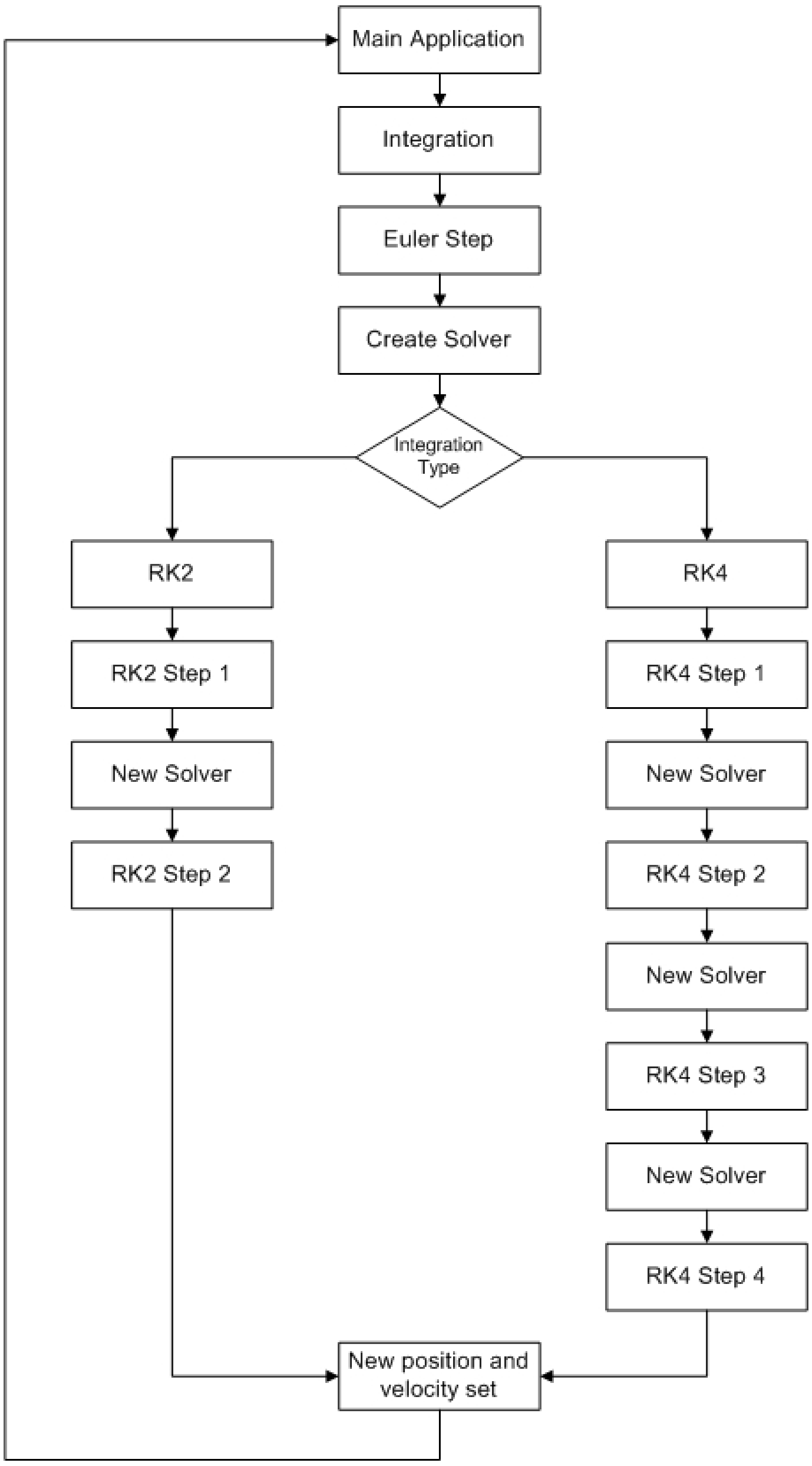


Figure 8.4.2: Flow chart for integration process

Chapter 9 Implicit Integration

9.1 Introduction

All methods previously described have been explicit methods for solving Ordinary Differential Equations (ODE's). The problem with explicit methods is that they are not very useful for solving "stiff" ODEs. A stiff ODE is best described as one that requires very small time steps to remain stable. Baraff [31] describes an example of a very stiff ODE in two dimensions.

Given a particle with position (x(t), y(t)), in order to keep the y-coordinate at zero, a component -ky(t) can be added to y'(t) where k is a large positive constant. As long as k is large enough, the particle will never move too far from y(t) = 0 as -ky(t) will cause y(t) to tend towards zero. Now if the particle is allowed free movement along the x-axis, the differential equation can be described as:

$$\dot{X}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} -x(t) \\ -ky(t) \end{pmatrix}$$

Equation 9.1.1:Particle allowed free movement along the x-axis

Because the particle is strongly attracted to the line y = 0 and less strongly towards the line x = 0, solving the equation far enough ahead in time will result in the particle eventually reaching (0, 0) and staying there upon arrival. Solving the equation using Euler integration with step size h yields:

$$\begin{aligned} X_{new} &= X_0 + h\,\dot{X}(t_0) \\ &= \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + h\begin{pmatrix} -x_0 \\ -ky_0 \end{pmatrix} \\ X_{new} &= \begin{pmatrix} (1-h)x_0 \\ (1-hk)y_0 \end{pmatrix} \end{aligned}$$

Equation 9.1.2: Euler integration with step size h

By looking at the y component of this equation it is possible to see that if $|1 - hk| > 1$ then y_{new} will have an absolute value that is greater than $|y_o|$ and as such Euler's method will never converge to an answer. To ensure this is not the case, the largest step h that can be taken is 2 / k. If k is very large, the steps will have to be very small.

9.2 An Implicit Integration Model

Cloth is a classic example of having to solve stiff ODE's. Initially, Baraff and Witkin [21] was considered as a model for the cloth simulation. Its implicit integration model incorporates the internal cloth forces, constraints, collision detection between cloth and cloth, and cloth and solid objects, and adaptive time stepping. On top of this, the model has proved itself in animations that it has been used in such as Monsters Inc. The only issue with implementing this model is time. Due to its complexity, getting the mathematical model in place is only half of the problem. Implementation specific issues as well as human error means that predicting the length of time required to get a working model into a simulation of choice is a difficult task.

9.3 Force Calculations

At the centre of the model is the structure of the cloth and the way forces are calculated. Rather than using a mass spring model like many cloth simulation frameworks, this model uses the deformation of a triangular element to calculate internal cloth forces. Essentially it acts like a three way spring but seems to be better suited to building non-uniform items of clothing such as a skirt or a t-shirt. This is due to the fact that it is easier to build triangular elements of an arbitrary size and as a result build more interesting items of clothing. A series of particles is therefore split into triangles with their vertices set to a rest position or normal length.

All forces in the model are based on conditions. A condition function C(x) "wants" to be solved for zero. That is, like a spring in a mass spring model acts to pull itself into its natural length, so are the forces calculated to pull a triangle to a state where its condition C(x) is equal to zero. This would normally be the rest state of the cloth. The basic formulation of the forces is as follows:

X_i denotes the position vector of the i^{th} vertex in world space. As such it can be represented as follows:

$$X_i = (x_i, y_i, z_i)$$

Equation 9.3.1: Representation of X_i

Each vertex also has its own coordinates in parametric space (u_i, v_i). uv coordinates are constant for each particle.

W is a function that maps uv coordinates to X, or world space.

$$X = W(u, v)$$

Equation 9.3.2: W function

So for a given triangle of vertices X_I, X_J, and X_K it can be defined that:

$$\Delta X_1 = X_J - X_I$$

$$\Delta X_2 = X_K - X_I$$

$$\Delta u_1 = u_J - u_I$$

$$\Delta u_2 = u_K - u_I$$

$$\Delta v_1 = v_J - v_I$$

$$\Delta v_2 = v_K - v_I$$

From the paper it is known:

$$(W_u, W_v) = (\Delta X_1, \Delta X_2)\begin{pmatrix} \Delta u_1, \Delta u_2 \\ \Delta v_1, \Delta v_2 \end{pmatrix}$$

So it follows:

$$Wu = (\Delta X_1 * \Delta v_2 - \Delta X_2 * \Delta v_1) * (1 / (\Delta u_1 * \Delta v_2 - \Delta u_2 * \Delta v_1))$$

$$Wv = (\Delta X_1 * \Delta u_2 - \Delta X_2 * \Delta u_1) * (1 / (\Delta u_1 * \Delta v_2 - \Delta u_2 * \Delta v_1))$$

Equation 9.3.3: Definition of X_I, X_J and X_K

The condition C(x) can now be calculated as from the paper:

9.3.1 Stretch Force

$$C(x) = a\begin{pmatrix} \|W_u(X)\| - b_u \\ \|W_v(X)\| - b_v \end{pmatrix}$$

Equation 9.3.4: Stretch Force

Where a is the triangles area and bu and bv are normally set to 1. They can however be changed to slightly lengthen a portion of a garment such as a sleeve.

9.3.2 Shear Force

The condition for shearing is calculated by finding the inner or dot product Wu^TWv:

$$C(x) = aWu(x)^TWv(x)$$

Equation 9.3.5: Shear Force

9.3.3 Bend Force

Bending is measured between pairs of adjacent triangles and the condition for bending depends upon the four particles that make up the two triangles. It can be defined as follows:

$$\sin \Theta = (n_1 * n_2) . e$$

$$\cos \Theta = n_1 . n_2$$

Equation 9.3.6: Bend Force

Where n_1 and n_2 are the surface normals of the two triangles and e is a unit vector parallel to the common edge. The condition for bending is simply $C(x) = \Theta$ which results in a force that counters bending.

9.4 Energy

Energy can now be calculated for any given condition by:

$$E_C = k/2 * C(x)^TC(x)$$

Equation 9.4.1: Energy

Where k is the selected stiffness constant.

For every particle i that condition C depends on, the force vector can be defined by calculating the change in energy E_C with respect to the position of the particle:

$$f_i = - \partial E_C / \partial x_i$$
$$= -k * \partial C(x) / \partial x_i * C(x)$$

Equation 9.4.2: The force vector

Finally, the derivative matrix K needs to be constructed out of a series of 3 x 3 matrices K_{ij} for every pair of particles i and j that C depends on.

$$K_{ij} = \partial f_i / \partial x_j = -k \; (\; (\partial C(x)/\partial x_i) \; * \; (\partial C(x)^T \; / \; \partial x_j) \; + \; (\partial^2 C(x) \; / \; \partial x_i \partial x_j) \; * \; C(x) \;)$$

Equation 9.4.3: Derivative matrix K

The full derivation of K can be found in Appendix D.

9.5 Verlet integration

Verlet integration is another method of integration that is suitable for cloth simulation because it is quite stable, especially when dealing with enforced boundary conditions [23]. In this method, there is no explicit velocity term which can cause corruption in the position values.

In terms of speed, Verlet integration is almost as fast as Euler to compute. It is also more accurate - under the right conditions, it is 4th order accurate [22].

The disadvantages with Verlet integration are its relative complexity compared to the explicit methods, and its poor use of varying time steps [23, 24].

9.6 Adaptive time steps

One method to improve the computation time of the RK2 method is to implement adaptive time stepping. This method analyses the vector between the old position and the new proposed position to see if it is within certain bounds. If the vector is deemed to be too small, the time step may be increased. If the vector is deemed to be too large, i.e. the distance between the old position and the new position too great, the time step is decreased. This ensures that the integration step size is maximised whilst remaining stable.

In order to implement adaptive time steps, the minimum and maximum bounds must be specified. The initial method tried was to base the value of the time step size on the magnitude of the force generated. This force is the value that will act on a particular particle, and the time step was changed accordingly. This method was not satisfactory, as its results were too unpredictable.

A second approach was taken to implement adaptive time steps, by basing the value of the time step on the deviation of a spring from its rest length. If the deviation from this value was too great, the spring was deemed to be stretching too far and the time step was decreased. If the deviation was too small, the spring could afford to be stretched further without adverse affect on the stability of the system, and so the time step could be increased. This method proved satisfactory, however, the RK4 method was selected for the integration calculations as it proved to be accurate and stable when handling larger time steps. For debugging purposes, it was decided not to include the adaptive time stepping in the final model.

Chapter 10 Motion capture data and FBX

10.1 FBX SDK

Motion capture data was integrated into the cloth simulator using Alias®'s FBX Software Development Kit. The importance of the FBX system to this cloth simulation is that the *.fbx* file format contains all the information for a model and the transformations that drives its motion. There are five categories of classes for the FBX SDK:

10.2 Object management

The class KFbxSdkManager is responsible for creating, managing and destroying most object types. There can be only one instance created at any given time. Any attempt to create another will result in the object having a value of NULL.

10.3 Scene description

The scene description classes are responsible for the description of the static elements in the scene such as a polygon mesh that a model is made up of and the animation data that is used to transform these elements over time.

10.4 Scene import and export

These classes are the interface between the scene description classes and the .fbx file format. For this simulation, the KfbxImporter class is used to import the .fbx files into the program for rendering in the OpenGL pipeline and for collision detection.

10.5 Tools

The tool classes are used to process animation data. For this simulation, the KfbxGeometryConverter is required to convert the human model from NURBS and patches to polygons.

10.6 Utilities

The utility classes contain all the data types used by the FBX SDK such as KFbxVector4 and KFbxMatrix.

10.7 FBX Overview

The FBX SDK overview document [2005] describes the scene description as follows: "The FBX SDK holds a scene description in a tree structure. The position of each node is expressed in coordinates relative to its parent. Every node has an attribute and an array of takes of animation data. The attribute describes the content of each node. Each take of animation data describes modifications to the state of the node over time". In order for the motion capture data and the corresponding model to be incorporated into an OpenGL environment a number of functions need to be performed during initialization.

10.8 The Process

Firstly, an FBX SDK manager object is created and then an importer object is created to import the FBX file into the manager. Once a scene object has been created the scenes status is set to MUST_BE_LOADED. The scenes status is only MUST_BE _LOADED during initialization and instructs the FBX interface to import the FBX file into memory and pass the FBX interface the following initialization instructions:

- Convert the model from NURBS or patches into polygons by passing the ConvertNurbsAndPatch() function the manager object and the scene object.
- Specify the "take" that you wish to use. An FBX file allows for more than one motion capture sequence to be stored and called for the one model.
- Initialize the frame period.

This process is outlined in Figure 10.8.1 over the page.

After successful initialization the FBX manager is ready to receive the polygon mesh and transformation data extracted at each given time frame. When the GetScene() function is called, the simulation recursively loops through all the nodes that make up the FBX object. Since the only data being extracted from the file is the mesh data for the purposes of rendering and collision detection, and marker data for the purpose of tracking the rotations of the hips, the only nodes that are retrieved are the nodes of type eMesh and eNull.

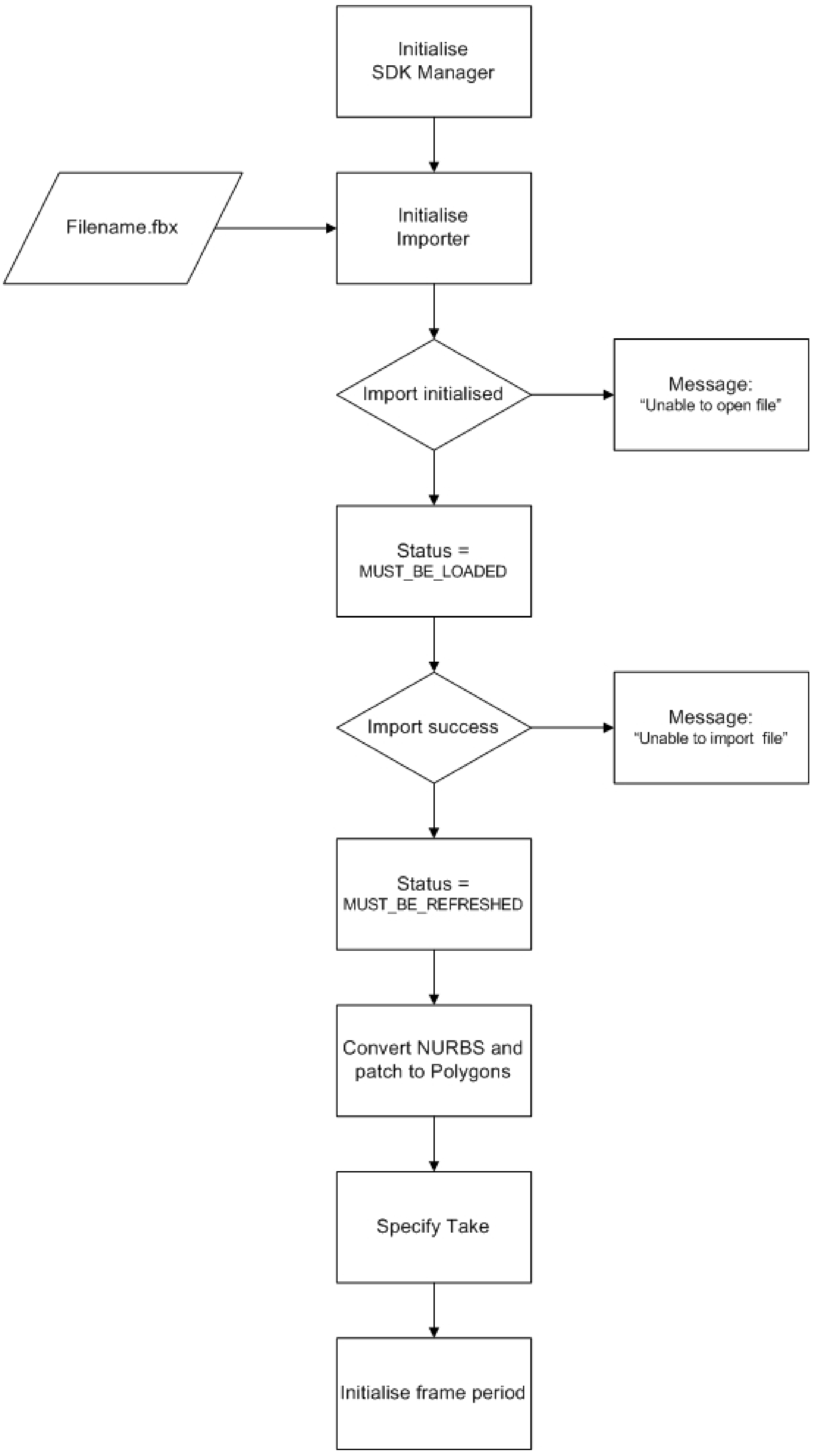


Figure 10.8.1: Flow chart for FBX initialization

A mesh node consists of control points or vertices that make up the individual polygons in the FBX model. Each mesh control point in the FBX file has a matrix that holds its global position for any given point in time. These control points are copied into an array so that they may be passed to deformation functions that interpolate the vertex positions at any given time. The two transformation computation functions are:

- ComputeShapeDeformation: This function deforms the vertex array with the shapes
- ComputeLinkDeformation: This function deforms the vertex array with the links

The result of calling these functions is the array of vertices that make up the polygon mesh in their new positions at the current time frame.

In order for this positional information of vertices to be useful for the cloth simulator, they need to be in a format that is easily understood by the rest of the system. FBX uses its own vector type (KFbxVector4) and in order to keep the FBX interface as a modularized component of the system, the vertex array must be converted into the same vector data type used by the rest of the system. Using the KfbxVector4 data type everywhere in the system would make the entire system dependent on the inclusion of the FBX SDK header file and is not an appropriate solution. A function has been developed that converts the vertex array into the standard Vector3D data type used in the rest of the system. Once the array of vectors has been converted, it can be transformed into a list of 3 or 4-sided polygons so that axis aligned bounding boxes can be created for collision detection and the mesh can be used for rendering.

10.9 Motion Tracking the Mocap Model

One of the challenges presented by placing clothing onto an animated model is constraining the cloth to the model realistically. An item of clothing such as a t-shirt or a poncho does not need to be fastened to a model as it can just be draped over the model and rely on the collision detection of the collar and sleeves to keep it fastened. For a skirt, there are three options that can be considered:

10.9.1 Collision Response

The same theory can be applied as with the t-shirt and poncho example and the skirt can be tightened until there is contact with the model above the hips. Because the width of the hips is greater than the waist the skirt will naturally rest on top of the hips in a realistic manner.

While in theory this approach can produce the most realistic results, it is the least robust. The collision response to any collision detection needs to be robust enough to deal with a permanent "rubbing" of the particles against the character mesh. While the collision response and detection is capable of dealing with this type of situation, there would still need to be a condition that prevented the skirt from sliding up the model during large movements.

10.9.2 Fastening Particles

The top rings of particles that form the cloth can be fastened to a series of polygons around the waist of the model.

Fastening particles in this way would also produce realistic, if slightly different results. However, a number of issues need to be addressed on the implementation. Firstly, the number of particles to be fastened needs to be deduced, and to which vertices they will be constrained. Since the polygon structure of different models varies infinitely, this would not allow the skirt to be ported between different models without manually defining the new constraint points.

10.9.3 Motion Tracking

The translation and rotations of the hips of the model can be tracked using the data from either the characters markers situated inside the character mesh or a series of polygons around the waist. These translations and rotations can then be applied to the top ring of fixed particles that make up the waist band of the cloth.

Choosing between using the marker information at the hip joints of the model and the polygons around the waist of the model to track the translations and rotations for the skirt proved to be fairly simple. Tracking the rig markers is a far easier exercise than tracking polygon vertices as the rig information is stored in the FBX file. Since the marker tracking method proved to be extremely accurate, it was the natural choice.

10.10 Tracking the Hip Markers

Each FBX file has marker information stored for various points on the rig and mesh. In this case, the two markers that make up the hips and the marker on the lower spine are the points of interest. Unfortunately, there is no way of directly extracting the tracking markers required and so as markers are encountered they are tested to see if they are required and stored if they are. From these three points, local hip rotations and translations can be calculated. The process is as follows:

```
Calculate the midpoint m of the two hips marker h1 and
h2: (h1 + h2 / 2).
Calculate the vectors vX and vY by subtracting m from h2
and m from the marker on the lower spine l respectively.
Take the cross product of vX and vY to find the
perpendicular vector vZ.
```

Translation tracking is simply a matter of ensuring that the centre of the ellipse of fixed particles is set to the midpoint *m*. The cloth particles are initialised relative to *m* and any changes to translation are tracked by adding the difference between the midpoints position from the last calculation to the fixed particle positions.

For rotations, initial calculations were made by deriving the relative rotations between *vX* and the world space x-axis, *vY* and the world space y-axis and *vZ* and the world space z-axis respectively. This is done by taking the arctan2 of the necessary vector components. To do this, three points are needed to be compared against the midpoint of the two hip markers. The first two are *h2* and *l* which are in the direction of the normalized *vX* and *vY* respectively. The third point, *g*, must be in the direction of the normalised vector *vZ.*

```
XRotation = arctan2(m.y - g.y, m.z - g.z)
YRotation = arctan2(m.y - g.y, m.z - g.z)
ZRotation = arctan2(m.y - h2.y, m.x - h2.x)
```

Here, arctan2 takes the arctangent of the two parameters and decides on the correct orientation for which of the two parameters will be the denominator and which will be the numerator. Hence, which will be taken as the opposite component and which will be taken as the adjacent component. These angles can then be passed to a function that builds a rotation matrix for the three axes in world space. By multiplying the ellipsoid ring of fixed particles by these rotation matrices, the waist band of the cloth should follow the rotations of the hips. This method is called the fixed angle approach.

This proved to be an elementary mistake as closer inspection of the method showed that it was a completely flawed approach to the problem. The angle between each local axis needs to be calculated for every axis in world space. While rotations around the y-axis will always be correct (assuming the character stays in an upright position) the rotations for the x and z-axis will only be correct if the character's hips have a particular orientation. It is easy to overlook this issue if the tests being undertaken have the character situated in only one orientation, in this case, facing the camera.

A more correct method is achieved by using direction cosines. Rotation information is calculated by taking the relative angles between the three vectors *vX*, *vY* and *vZ* and *all three* of the x, y and z-axis of world space. This is calculated by taking the dot product of each of the three normalised vectors *vX*, *vY* and *vZ* with the unit vector of each of the world space axis.

```
XRotationWithX = vX . Vector(1.0, 0.0, 0.0)
XRotationWithY = vX . Vector(0.0, 1.0, 0.0)
XRotationWithZ = vX . Vector(0.0, 0.0, 1.0)

YRotationWithX = vY . Vector(1.0, 0.0, 0.0)
YRotationWithY = vY . Vector(0.0, 1.0, 0.0)
 YRotationWithZ = vY. Vector(0.0, 0.0, 1.0)

YRotationWithX = vZ . Vector(1.0, 0.0, 0.0)
ZRotationWithY = vZ . Vector(0.0, 1.0, 0.0)
```

```
ZRotationWithZ = vZ . Vector(0.0, 0.0, 1.0)
```

By building a single matrix with these values, the correct rotation of each of the three reference vectors can be calculated with respect to all three of the world space axis. Multiplying each particle in the row of fixed particles by this matrix calculates the correct rotation to allow the waist band to match the rotations of the hips.

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{bmatrix} XRotationWithX & YRotationWithX & ZRotationWithX \\ XRotationWithY & YRotationWithY & ZRotationWithY \\ XRotationWithZ & YRotationWithZ & ZRotationWithZ \end{bmatrix} \bullet \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

It is worth mentioning that in both the fixed angle and the direction cosine method, the angles represented are the absolute angles of the hip rotations. This is obvious as the calculations of angles are relative to the x, y and z-axis of world space, but this does have implications for position storage of the particles. Each particle must store its position in world space with no rotations applied at each frame by storing the particle positions between the translation and rotation processes. Rotations are then applied to these positions and the result stored as a separate vector.

It is also important to note that for the rotation calculations of the ellipse of fixed particles to be correct, the midpoint of the ellipse, *m*, must be subtracted from each fixed particles position before rotation and then added again after rotation. This effectively means that the rotations of the ellipse of fixed particles occur around the origin of world space (0, 0, 0). The particles are then transported back to their positions relative to the midpoint of the hips *m*.

Figure 10.10.1 below shows the OpenGL view of the FBX character with the cloth skirt.

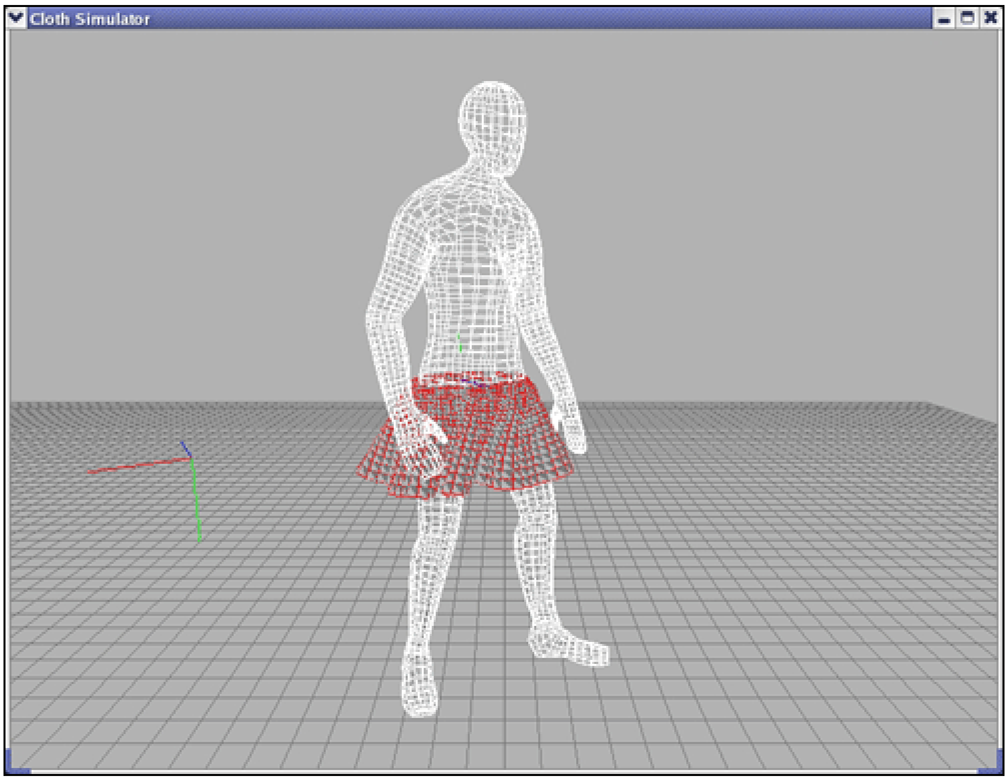


Figure 10.10.1: OpenGL view of the FBX character with cloth skirt

10.11 Motion Capture Data

The motion capture was taken at AccessMocap, Bournemouth Media School. The data cleanup and FBX models were provided by Andy Cousins (Mocap Producer).

Chapter 11 Collision Detection

11.1 Introduction

In order to keep computation time to a minimum, the collision detection framework utilized Axis Aligned Bounding Boxes (AABB) to break the polygon mesh of an object into manageable groups of polygons. The definition of an AABB is simply as the name suggests: A box with edges aligned to the x, y and z-axis of a coordinate system that is bound by a minimum and maximum set of coordinates. By building an AABB hierarchical tree, a series of leaf AABB's , with creation conditions defined by specified heuristics, will hold all of the polygons in a mesh.

11.2 The Process

The process of generating an AABB tree for a given piece of geometry (defined by polygons) is recursive. A root AABB is defined as a bounding box encompassing all of the polygons and is calculated by looping through all of the polygons in the mesh and determining the minimum and maximum values of the x, y and z- axis. Once this has been calculated, the longest axis of the box is calculated and the polygons are split into two groups on either side of this axis with their minimum and maximum values calculated to form two more bounding boxes. These are called child bounding boxes. This process moves recursively down all paths of the tree until specified heuristics are satisfied. The heuristics are:

- The maximum tree depth is reached. This is the number of "generations" of children that are created.
- A bounding box contains the minimum number of polygons allowed.

For any given bounding box, if the heuristics pass, the bounding box becomes a leaf bounding box, i.e. it has no children.
Figure 11.2.1 and Figure 11.2.2 below show a flowchart for the AABB tree creation and the AABB tree hierarchy, respectively.

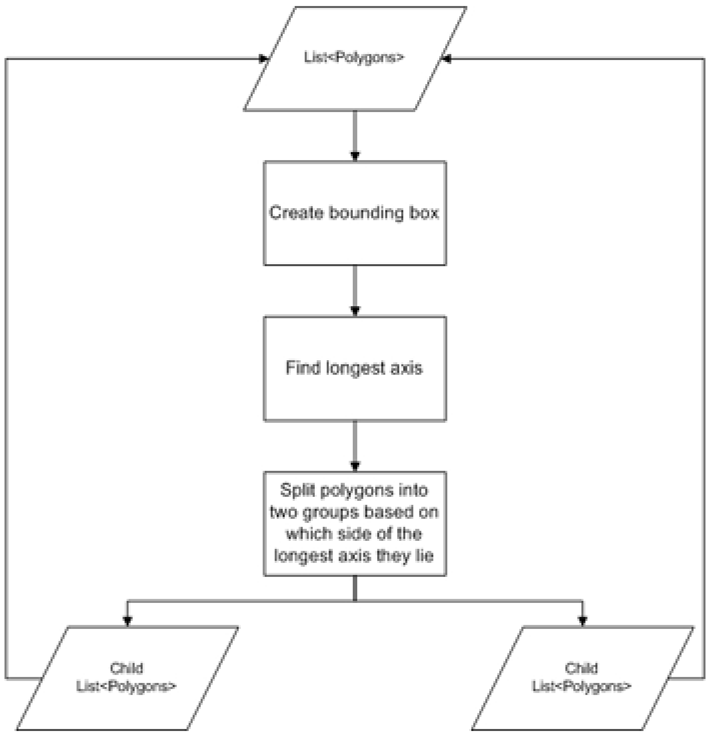


Figure 11.2.1: Flow chart of AABB tree creation

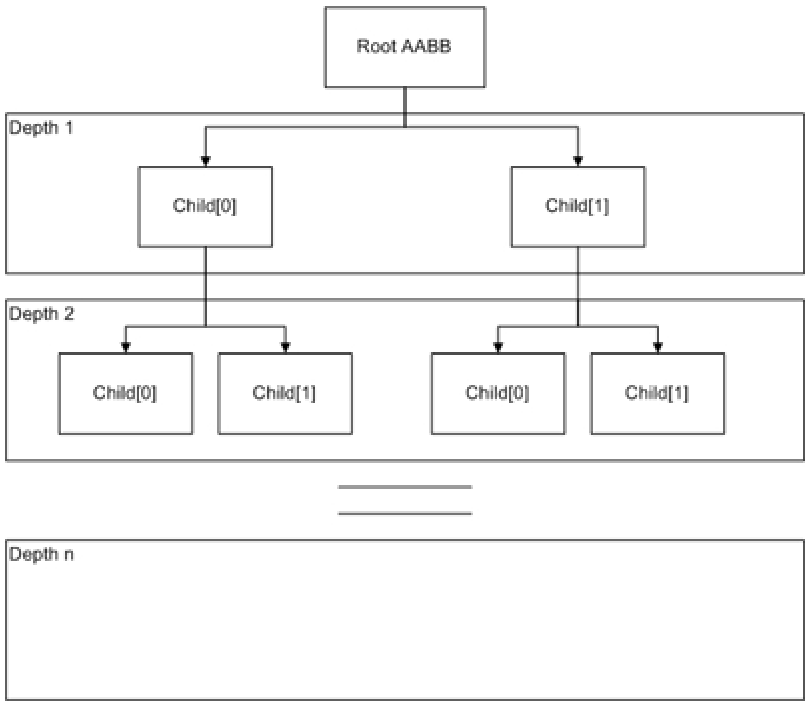


Figure 11.2.2: AABB tree hierarchy

Once the bounding boxes have been created, collision detection can be calculated. Regardless of the collision detection method, the underlying functionality of the bounding boxes with respect to collision detection remains the same. For every step in the animation, check to see if an object has moved into the root bounding box. If it hasn't then there has been no collision. If it has, then check to see if the object is in either of the root nodes' child bounding boxes. The process continues recursively all the way down the trees hierarchy until either the object does not move into any more bounding boxes, or the object has moved into a leaf bounding box. Once in a leaf bounding box, collision can be calculated against all polygons in that bounding box. In this way, collision detection calculations are localized. Figure 11.2.3 below shows the bounding boxes on the FBX character.

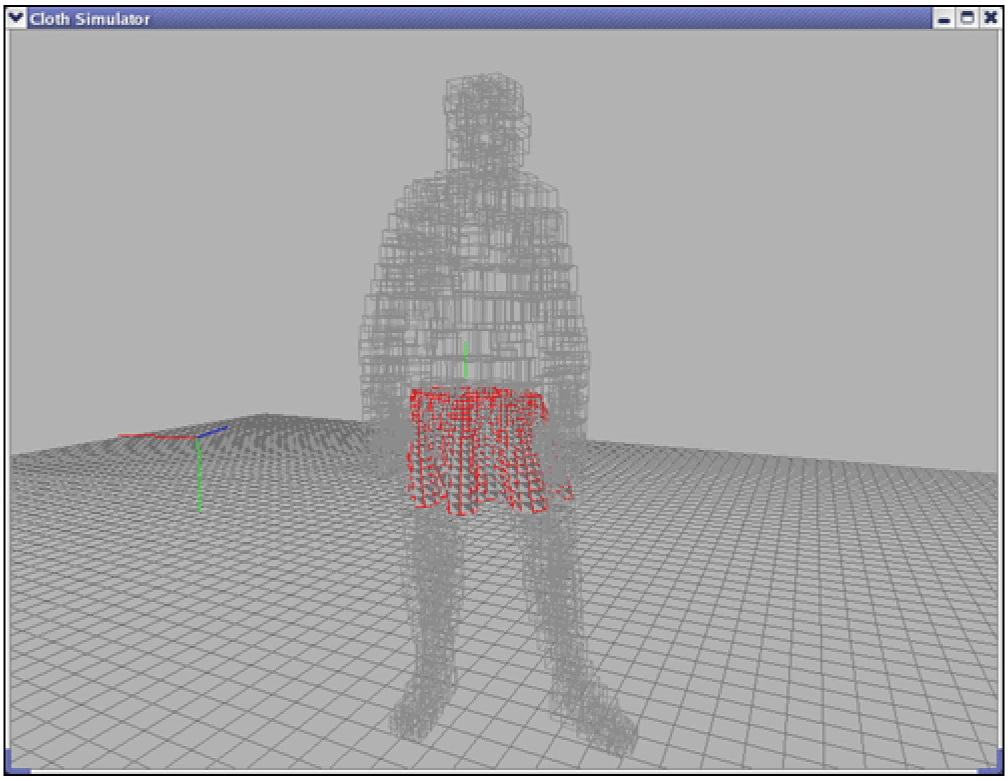


Figure 11.2.3: Bounding boxes on the FBX character

11.3 Collision Detection Algorithms

Two methods for collision detection were researched for this implementation.

11.3.1 Method 1

The collisions being dealt with are between a cloth model and a solid object or a cloth model and itself. The first method simply checks to see if a particle (cloth particle) passes through a given polygon. This method is outlined in various collision detection techniques on the internet and described in Hill [2].

Assuming that it has been determined that a particle has breached a leaf bounding box node according to the procedure outlined in Section 11.2 on bounding boxes, for every polygon in this bounding box, a test needs to be made to see if the particle will pass through any of these polygons during a specified time period *t*.

By taking a particles original position and its particles new position (as calculated for the current time step *t*), a line, or ray, can be drawn between the two points as a representation of the path that it will travel. This line can be described parametrically as:

$$a = b + ct$$

Equation 11.3.1: Ray line

Where *a* is the points new position, *b* is its original position, *c* is the ray direction of the particle and *t* is a time step. Since *a*, *b* and *t* are known, the ray direction *c,* can be calculated by rearranging the equation. *c* is significant as it means a collision can be determined if the dot product of c and a polygons surface normal is less than zero.

$$n \cdot c < 0$$

Equation 11.3.2: Normal test

Where n is the polygons surface normal.

For a specific polygon, it is known that given any point M that lies on the polygons surface, and the polygons surface normal n, an intersection point P will have to satisfy the equation of the plane defined as:

$$n \cdot (P - M) = 0$$

Equation 11.3.3: The plane equation

If a collision has been detected, a collision has occurred at an unknown time t_{hit} and the collision point can be defined as:

```
P = b + ct_hit
This can then be substituted into Equation 11.3.3 to
form:
N . (b +  ct_hit  - M) = 0
Rearranging this to solve for t_hit yields:
t_hit  = (n . (M – a)) / n . c
```

Now t_{hit} has been solved, it can be substituted and the intersection point can be determined.

11.4 Method 2

Bigliani and Eischen [3] describe a method that not only takes into account a given particles current and future positions, but also the position of the other two particles associated with the triangular element containing the colliding particle.

Figure 11.4.1 below, shows two triangles, ABC and DEF at a point k in time. Figure 11.4.2, depicts the same scenario except that the position of point D in DEF can be seen at time k+1.

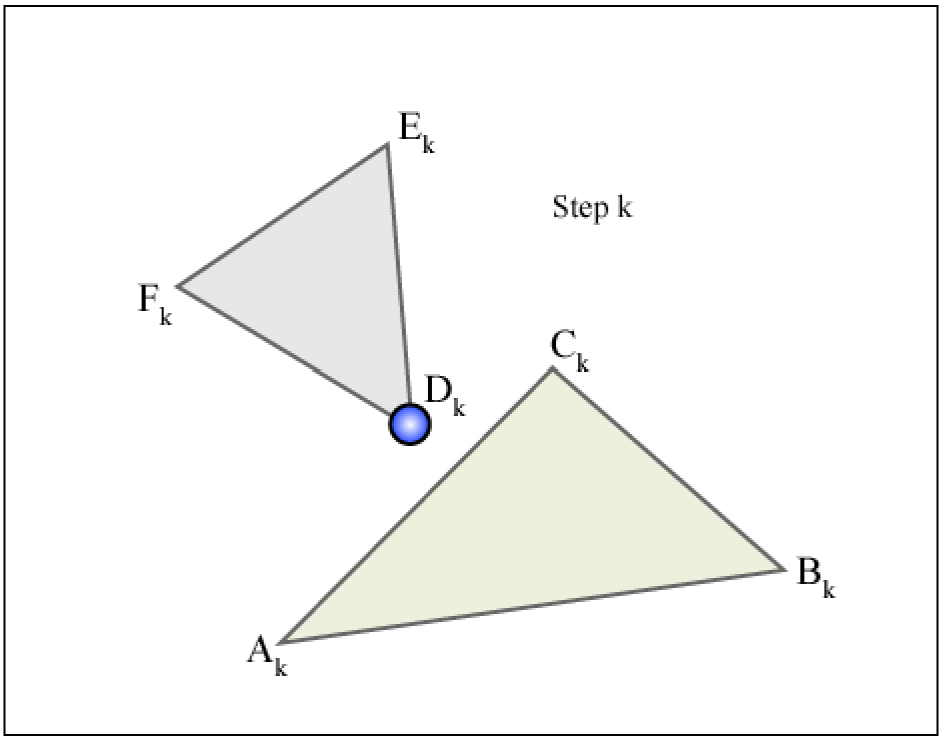


Figure 11.4.1: Triangles DEF and ABC at time k

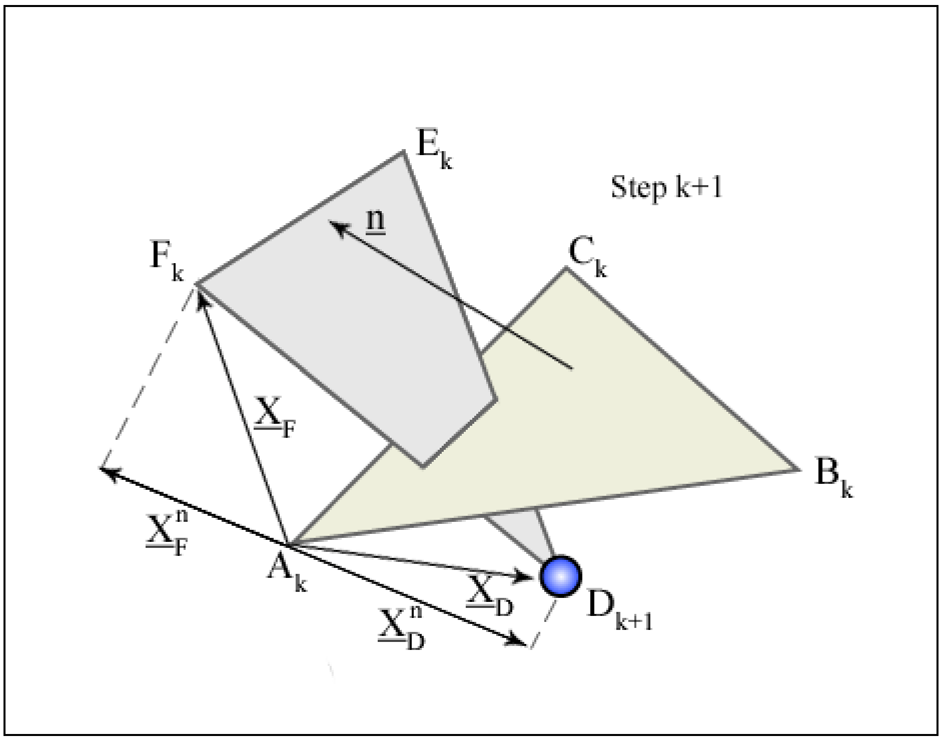


Figure 11.4.2: Point D at time k + 1

By calculating $X_F = F_k - A_k$ and $X_D = D_k+1 - A_k$ a collision can be detected if $n \cdot X_F$ and $n \cdot X_D$ are of opposite sign. Either the particle D has crossed the plane of triangle ABC between time k and k + 1, or the edge DF of triangle DEF intersects the plane of the triangle ABC.

As with method 1, the equation of the plane ABC can be defined as $n \cdot (P - A) = 0$ where P is a point of intersection, A is a vector pointing to vertex A and n is the normal to the surface. Bigliani and Eischen define P, A and n as follows:

$$P = xi + yj + zk$$
$$A = x_A i + y_A j + z_A k$$
$$n = ai + bj + ck$$

Equation 11.4.1: Definition of P, A and n

Thus the equation of the plane can be rewritten as:

$$a(x - x_A) + b(y - y_A) + c(z - z_A) = 0$$

Equation 11.4.2: The plane equation

From Figure 11.4.2, the line r between point F at time k and point D at time k+1 can be defined as a function of a parameter t. Unlike method 1, the ray of intersection is calculated between the intersecting points destination at time k+1 and one of the other vertices that form the triangle at time k, rather than the intersecting points origin and destination. Once the position of point D at time k+1 is known, the line r describes the new direction of one of the edges of the triangle DEF.

Vector r = li + mj + nk has the components:

$$l = x_D - x_F$$
$$m = y_D - y_F$$
$$n = z_D - z_F$$

Equation 11.4.3: Vector components

The parametric equations for the coordinate points on r are:

$$x = lt - x_F$$
$$y = mt + y_F$$
$$z = nt + z_F$$

Equation 11.4.4: Parametric equations

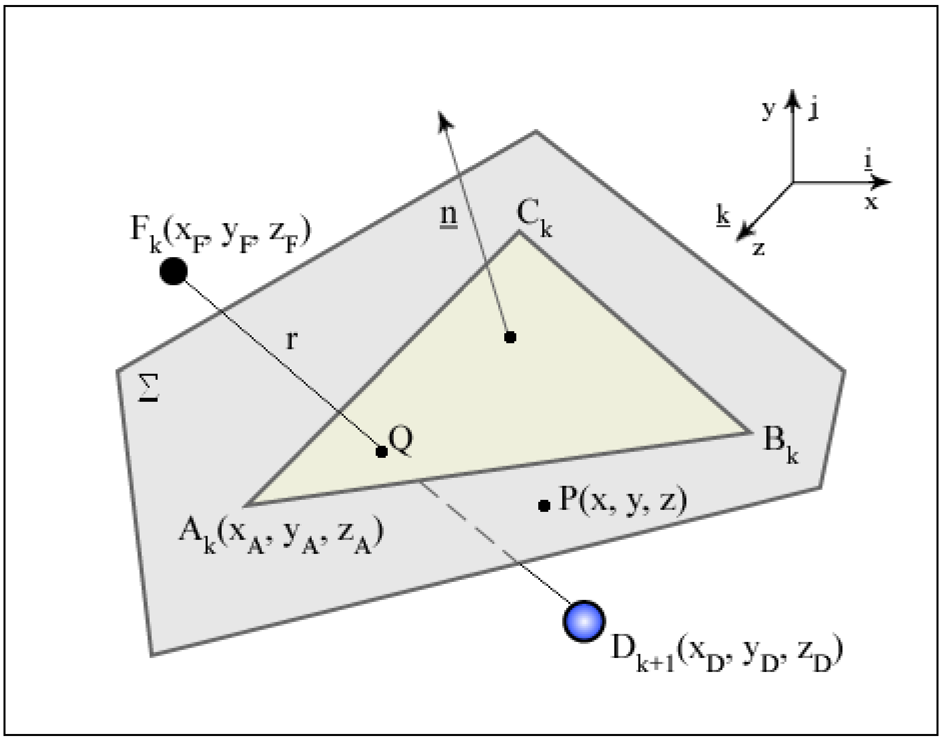


Figure 11.4.3: Line r, the ray of intersection

As with method 1, time t can now be determined at which intersection occurs at point Q by substituting the equation of the line r into the equation of the plane.

$$t_Q = - (n \,.\, (F - A) / al + bm + cn)$$

Equation 11.4.5: Substitution of equations

The coordinates of the intersection can now be determined by substituting this time into the parametric equations defined in Equation 11.4.4:

$$x_Q = lt_Q + x_F$$
$$y_Q = mt_Q + y_F$$
$$z_Q = nt_Q + z_F$$

Equation 11.4.6: Coordinates of intersection

Now that the point of intersection has been derived, the nature of the intersection must be determined. Firstly a check needs to be made to see if the collision point Q is equal to any of the vertices of the triangle ABC, meaning the collision has been with one of the vertices of the triangle. Assuming now, that the collision has not been with a vertex of triangle ABC, the problem can now be projected into two dimensions.

The intersection point Q and the triangle ABC is projected onto a two-dimensional coordinate system parallel to the three dimensional coordinate plane that has a surface normal most closely aligned with the surface normal of triangle ABC. Determining whether the intersection point Q is within the triangle ABC (particle-to-triangle collision) or if the intersection point Q is outside the triangle ABC (edge-to-edge collision) is now relatively simple. By making the point Q the origin of the new two-dimensional coordinate system, a check to see how many times the x-coordinate plane intersects the triangle ABC yields where the point Q lies. As can be seen in Figure 11.4.4, one intersection means the point Q lies inside the triangle ABC (a) and any other result means the point Q lies outside of the triangle ABC (b) and (c).

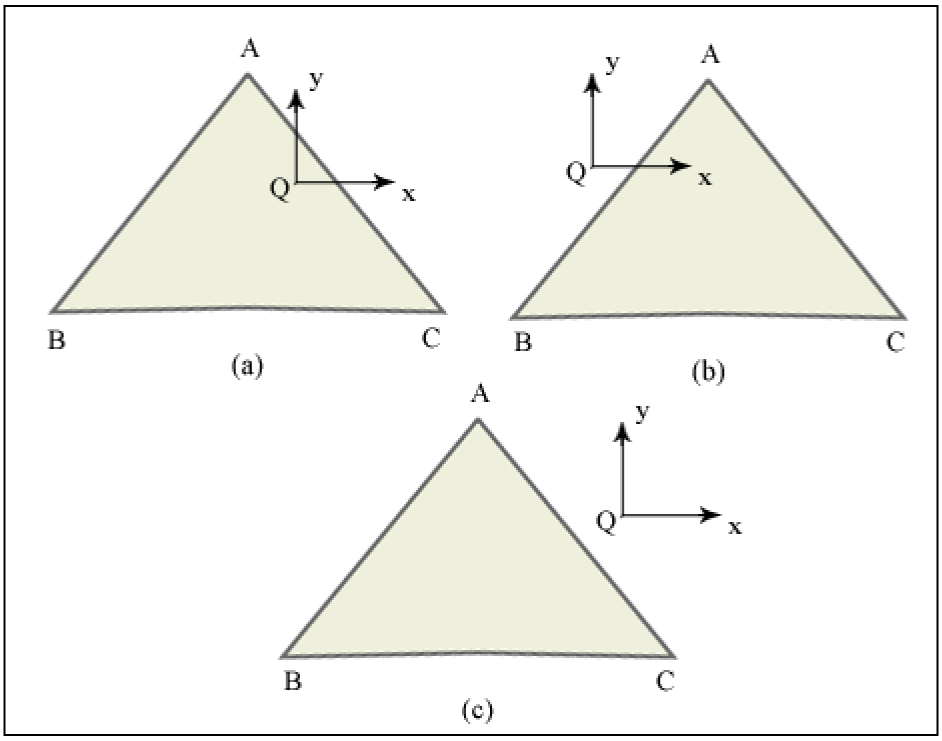


Figure 11.4.4: 2D Projection of intersection point Q

The useful thing about this method when considering collision detection for cloth is that, once a collision has been detected, it can determine whether the collision is between particle-particle, particle-triangle or particle-edge. If the line considered for computing the collision is FD, then the algorithm can also check for edge-edge and triangle-triangle collision.

Chapter 12 Collision Response

12.1 Introduction

Collision detection and response are fundamental to the visual outcome of any cloth simulation. They are separate issues, but rely on the accuracy of each other and for this reason, it is important to make the distinction between the two.

Collision detection is a computational geometry problem, determining whether two or more objects have intersected, and where these intersections have occurred. However, collision response is a physics problem, dealing with the energy, forces and motion of two or more objects after they have collided [25].

Implementing collision detection and response as part of the cloth model was crucial to the final appearance of the fabric. When a collision is detected, the velocity, and therefore positions of the particles involved become altered, which in turn affects the movement and drape of the fabric. Collision detection is an incredibly important part of any cloth simulation, as cloth in the "real world" will not penetrate objects or itself. Also, depending on the physical properties of the cloth, during movement, collisions will determine its behaviour.

When simulating cloth, it is important to consider that there are a wide variety of fabrics, available, each with different physical properties. These physical properties all cause the cloth to behave differently when it collides with an object. Satin is very smooth due to the linear direction of the fibres. It also has a low degree of friction and so will slide off any object it comes into contact with. Wool however, has a high degree of friction, with its coarser fibres generally going in different directions. This will cause wool to stick to objects as opposed to sliding off them, especially if those objects themselves have a high degree of friction [26].

12.2 Response Considerations

The cloth model selected for the application is based on the mass-spring model, which is a physically based model. The chosen collision response techniques have to provide a realistic response to the non-linear problem of the cloth drape and manipulation [3].

It was also necessary that the chosen response method kept the model stable for a wide range of parameters. The parameters could be changed in order to demonstrate the various fabric types, such as satin and wool, and the different collision types, i.e. both cloth on cloth collisions and cloth on object collisions. It was also necessary to implement an algorithm that would not interfere with the physical behaviour of the model, through creating local instabilities or constraining the natural evolution of the fabric shape [26].

Once the collision detection is carried out, the collision response method gets called. It is based on the colliding particle, and the polygon it collides with, which could be on the cloth, the FBX model or on the static object model. Using this information, there are various methods available for dealing with collision response. When deciding on which collision is the most appropriate, there are several questions that need to be asked in relation to the simulation:

- Is it rigid body or soft body dynamics that are being simulated?
- If dealing with cloth, this is a soft body. However, the simulation may be dealing with cloth colliding with a rigid body. In this case, it is important to decide if the rigid body is static or if it will have any movement during the simulation.
- How will the simulation deal with time? When there is a collision, will the simulation back up to the collision point or deal with the collision at the next time step?
- Is the appropriate response force going to be Kinematic, Dynamic or Impulse [27].

A Kinematic response involves moving the colliding particle to the surface with which it has collided. Its motion component, usually normal to the surface that has caused the collision, is negated. This will produce an inelastic response with no bounce. If the response is to be bouncy, the velocity component which is normal to the surface is negated.

A Dynamic Penalty method involves introducing a penalty force to deal with the collision. Usually in this method a spring is inserted between the particle and the collision point to stop collisions. However, this method can introduce stiff equations into the system, and so small time steps are needed to avoid any instabilities.

The impulse method involves assuming the separation velocity of the particle is a normal component of the collision velocity [27]. This gives rise to the equation:

```
Vs = e * Vn.
```

Equation 12.2.1: Impulse method assumption

This value is a function of linear velocity and rotational velocity. The normal force gets calculated at the collision point, and so this results in the rotational velocity and the linear velocity being equal to the separation velocity.

12.3 Response Implementation

The methods that were considered for the cloth model as part of this project are described below. Two of these methods have been implemented for the final model.

1. The first is for cloth collisions with a static rigid body, and will result in a new velocity for the colliding particle.
2. The second method is for cloth collisions with a rigid body that has movement during the simulation, and this method also results in a new velocity for the colliding particle.
3. The third method is for cloth collisions with a rigid body with little or no movement during the simulation, and will results in a new position for the colliding particle.
4. The fourth method is for cloth collisions with a rigid body, with its movement values disregarded, which will result in a new position for the colliding particle.

12.4 Method 1

Cloth - static rigid body collision response calculating a new velocity.

This method is very effective when the cloth is moving and collides with a static object.

It is based on the theory of correcting the velocity of the colliding particles, by essentially reflecting the velocity from the collision to carry the particle back in the opposite direction.

It takes in a particle, X, with a velocity V. The particle is moving towards a plane P with normal N. Two more vectors also need to be calculated to represent motion parallel and tangential to the collision normal. The collision normal is simply the normal to the plane [28].

```
              Vn = normal of velocity
              Vt = tangent of velocity
              V = velocity of particle
    N = Normal to the plane and collision normal.
            Kr = restitution co-efficient

                  Vn = (N . V)N
                   Vt = V - Vn
             NewVelocity = Vt - Kr Vn
```

Equation 12.4.1: Collision response Method 1

The restitution coefficient will determine the amount of the normal force, Vn, which is applied to the resulting force. If this Kr value is 1, the collision will be totally elastic, however, if the value of Kr is 0, the particle will stick to its collision point.

12.4.1 Results

This method provides excellent results when the cloth is colliding with a stationary object. However, because it does not take into consideration any information about the colliding object, such as its velocity, then it is not effective for collisions between cloth and moving polygons.

12.5 Method2

Cloth - moving rigid body collision response, calculating a new particle velocity.

This method is used in order to deal with moving cloth colliding with a moving rigid body. The principle behind this method is the transference of momentum from the rigid body onto the cloth. The velocity with which the cloth is hit, i.e. the velocity of the rigid body, is vital in computing the correct resultant velocity for the cloth's response movement. The energy dissipated from the mechanical damping of the springs, or from the friction due to the surface contacts, must also be accounted for when using this method.

Momentum is defined to be the mass of an object multiplied by the velocity of the object [28]. This collision response method is based on the Momentum Conservation Law**:**
"Within some problem domain, the amount of momentum remains constant; momentum is neither created nor destroyed, but only changed through the action of forces." [28]

The algorithm is based on Newton's Laws of Motion, and it is relatively complex. Dealing with momentum is more difficult than dealing with mass and energy because momentum is a vector quantity, so it has both magnitude and direction [28]. It must also be conserved in all three directions at the same time.

The appropriate response velocity for the colliding particle is computed using the information about the particle before and after the collision, and the information about the polygon it has collided with, which is in the form of a triangle, before and after collision. Before the response is calculated, basic assumptions are made about the simulation:

- The collision line is considered to be the normal line to the surface of the triangle at the collision point [3].
- Because the triangle is attached to the whole network of triangles, its mass M can be assumed to be much greater than that of the particle, so the triangle is forced to keep its velocity after the collision [3].

The relationship between the velocity of D, the colliding particle, and the collision point Q, before and after the collision are expressed using the following formulas:

```
       V_dk_a = -eV_dk_b + (1 - e) V_qk_b
                 V_qk_a = V_qk_b


       K = triangle normal before collision.
```

```
D = colliding particle
Q = collision point on the triangle surface.
V_dk_b = component of velocity of D along direction K
before collision
V_dk_a = component of velocity of D along direction K
after collision
V_qk_b = component of velocity of Q along direction K
before collision
V_qk_a = component of velocity of Q along direction K
after collision
E - restitution coefficent.
```

Equation 12.5.1: Collision response Method 2

The restitution coefficient accounts for energy dissipated upon collision. The algorithm calculates the information about Q, the collision point on the surface of the triangle, based on the triangles information. A local co-ordinate system is defined for that polygon, using ABC, the triangle's vertices.

The position and velocity of ABC is used to define (i, j, k, w), the vectors of the local co-ordinate system. These values are then used to calculate the angular velocity of the triangle:

```
AngVel = (velj . k) * i + (velk . i) * j + (veli . j) * k
```

Equation 12.5.2: Angular velocity

The angular velocity of the triangle is then used to calculate the velocity of the collision point on the triangle:

```
Vel Q = VelA + AngVel x (Q - A)
```

Equation 12.5.3: Velocity of the collision point.

These values for angular velocity and the velocity of Q are calculated for each collision at each time step. Finally these values are used to calculate the new velocity of the particle after its collision.

```
Vel D (after) = VelD (before) - VelDk + VelQk
```

Equation 12.5.4: New velocity after collision

Each time a collision is detected, the collision flag is set in the particle. A reference to the colliding polygon is also set in the particle and after all the collision checks are done, the collision response method is called. Each particle is checked, and if the collision flag is set, a response object is created for this particle. The object calls the appropriate response methods and after the new velocity for the particle is computed, the collision flag is set back to false. This value for velocity is then used during the next time step, resulting in the new position being calculated for the particle being moved away from the colliding object instead of further into it.

This method also takes into account a tangent velocity for the collision. This value can be used when calculating the effect of viscous damping or friction on the forces acting on the cloth [3].

12.5.1 Results

Due to the nature of the simulation, the velocities and momentum of the cloth particles occasionally became too great for the collision response to adequately provide a resultant velocity that would pull the cloth from the model.

In order to work around this issue, the following method was attempted. From observations of the cloths behaviour, it was found that when the particles were colliding with the polygons head on, the velocities of the cloth particles were too great to be overcome by the new calculated velocity. To solve this issue, a method was introduced to calculate the dot product of the vector of the particle going into the polygon, with the surface normal of the polygon. If small, the value meant that the collision was head on, and so the velocity of the polygon pushing out was increased by a large number to counteract the velocity of the cloth particle coming in.

The result appeared to have the desired effect, however this value had to be tested against other cloth parameters to simulate the chosen fabric type.

12.6 Method 3

Cloth – rigid body collisions, with position correction.

This method of collision response is based on the idea of calculating a new position for a colliding particle, using the distances between particles and the polygons, and the corresponding normals.

The model was set up with a maximum threshold distance, which is the maximum distance a particle can be to the surface before a collision occurs. If the particle gets closer than this threshold, the particle is sent back in the direction of the collision normal.

First the normal component of the collision is calculated:

```
Rn = N * (t – d)
```

Equation 12.6.1: Normal component of collision

Next, the method approximates a model for friction, by calculating the tangential component, through scaling the tangential part of the particles velocity:

```
d = New Pos – Initial Position
Dt =  d – N * (d . N)
Rt = -cf * Dt
```

Equation 12.6.2: Tangential component of particles velocity

Finally the new position is calculated using these values:

```
New Pos += Rn + Rt
d = distance of the particle to the surface.
Rn = normal component of the collision
Rt = tangent component of the collision
N = surface normal
T = given threshold
Dt = tangent part of particle movement vector
Cf = friction parameter
```

Equation 12.6.3: New position calculation

12.6.1 Results

Correcting the position without integrating the simulation a second time to get a smooth corresponding velocity, resulted in popping of the particles and reduced the overall visual effect of the skirt.

12.7 Method 4

Relaxation using Verlet Integration and Jacobian or Non-Linear Gause-Siedel Relaxation methods [29].

This method was not implemented as part of this project, however it provides an interesting way of dealing with collisions, and is appropriate to include in this section. Cloth simulation models that implement this type of collision response use Verlet integration as the integration method. Additional information on Verlet integration can be found in Section 9.5.

The Non-Linear Gause-Siedel Relaxation method works on the following principle. It uses constraints to project the particle out of the object it has collided with. This involves moving the point out perpendicular to the collision surface. This method ensures the distance constraints between the particles are satisfied, and that cloth particles do not penetrate any object they collide with. The relaxation technique works on each particle, translating it to a new position and satisfying its constraints. This is repeated until all the constraints are satisfied to within some threshold. This is carried out for a set number of iterations until the relaxation threshold is reached [30].

It can be likened to inserting infinitely stiff springs between the particle and the collision surface. These springs are so strong and suitably damped that instantly they will attain their rest length zero [29]. After an integration step, the particles positions could potentially have broken the distance threshold to the surface. Therefore, in order to get the particles back to a position that is correct within the threshold, they are projected onto the equation condition that describes the infinite springs. This is done by either pushing the particles together, or away from each other directly, based on their deviation from the threshold. Each set of particles is iterated through and this method is carried out for each set locally. This procedure of relaxation is carried out for a set number of iterations, until eventually all the sets of particles in the model are back within their threshold and the model is stable [29].

Verlet integration is a finite difference method for integrating equations of motion. This integration method is perfect for working with the Jacobian relaxation, as there is no explicit velocity term to affect the data. This results in it being far more stable when particles are transferred to a new position, than an integration method that includes velocity. As a result, the collision response is more stable [30].

However, this technique does have its downsides. The Verlet integration requires two time steps during initialization hence the initial conditions are crucial, and using this relaxation technique can also result in slightly rubbery cloth, as the process ripples through the cloth.

12.8 Collision Prediction

When using a velocity correction method, it was felt that the simulation would benefit from having the collisions predicted in advance. Anticipating collisions a step ahead of the current time step enables the new velocity to take effect to prevent the cloth from penetrating the object. Implementing collision prediction effected not only collision detection and response, but also the integration methods.

This method was developed by having two integration steps in each time step. The first integration step calculated the new position and velocity values for the cloth, which was then passed into the collision detection and response.

The second integration cycle calculated what the values would be at dt + 1, the next time step. These values were finally passed into the collision detection to predict if a collision would occur at the next step. If it was found that a collision would occur, the appropriate collision response was calculated.

Finally, a method to update each particle would get called after all these calculations were made, to assign the appropriate values to the particles. The following rules were applied during the assignment:

1. If a particle is found to have no collisions either during this time step or predicted for the next, then it is assigned the velocity values from the first integration step.

2. If a particle is found to have collisions during this time step, then it is assigned the velocity values from the current step, collision response calculations.
3. If a particle is found to have a predicted collision then its velocity is set to the predicted response velocity, in order to prevent the collision from occurring at the next time step.

12.9 Test Environment

A test environment was built to accurately test the collision detection and the appropriate collision response methods. It consisted of a sphere with its associated AABB tree and a Cloth Sheet object. The collision detection and response classes were unchanged from the main cloth simulation.

Initially the test environment was used to test the response from collisions between the cloth and the polygons of the sphere. The method was implemented for correcting the velocity of the cloth particles based on the collision normal and the components of the cloth velocities, as described in Section 12.4. Results for this test found that collision response only worked when the sphere was static.

To improve on these initial tests and to account for a moving rigid body, it was necessary to implement a response method that took account of the momentum of the sphere, as described in Section 12.5.

The test environment allowed for greater control over collision testing by:

- Using simple collision objects with low polygon counts
- Adding control over the movement of the objects
- Providing visual feedback
- Speeding up the simulations

Figure 12.9.1 below shows the flow chart for collision response.

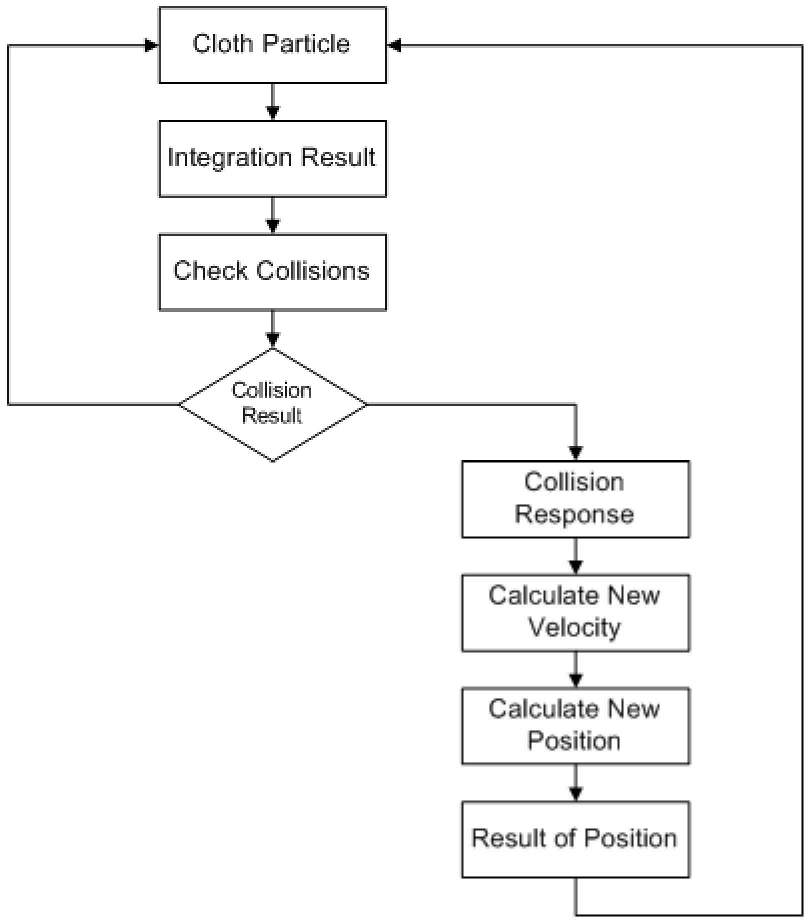


Figure 12.9.1: Flow chart for collision response

Chapter 13 Polygon Models

13.1 Obj Loader Class

The collision response test environment initially used a polygon sphere constructed from vertex data gathered from Maya™ using a simple MEL script. In order to further test the collision algorithms with more complex models, it was necessary to construct an Obj Loader class.

The Wavefront® .obj ASCII file format allows Maya polygonal data to be exported as a text file. The following list outlines some of the data types included:

- Vertex data
  - Geometric vertices (v)
  - Texture vertices (vt)
  - Vertex normal (vn)
- Elements
  - Point (p)
  - Line (l)
  - Face (f)
- Grouping
  - Group name (g)
  - Object name (o)

A square polygonal face that measures one unit along each side would be described in an .obj file as follows:

```
v 0.000000 1.000000 0.000000
v 0.000000 0.000000 0.000000
v 1.000000 0.000000 0.000000
v 1.000000 1.000000 0.000000
          f 1 2 3 4
```

The Obj Loader class consists of a number of switch statements and string parsing methods for loading an .obj file into various data arrays. One caveat for the models exported from Maya™ is that they must only consist of triangular or quadrilateral faces. This is due to the polygon definition used within the cloth simulation application. Figure 13.1.1 below shows the cloth sheet colliding with a polygon model imported from an .obj file.

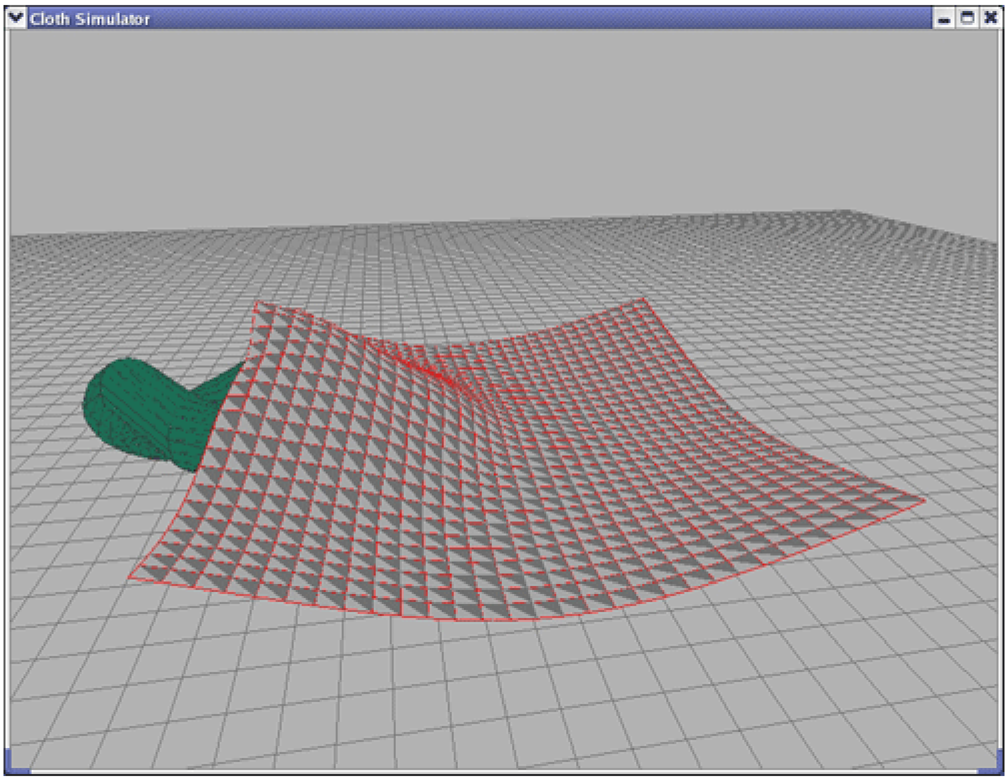


Figure 13.1.1: Cloth sheet and polygon model

The Obj Loader class also contains a method for rendering the polygon model as a RenderMan subdivision mesh. Additional information on RenderMan subdivision meshes can be found in Section 15.5.

Chapter 14 OpenGL Display

14.1 Camera Class

Although OpenGL has standard tools for setting up a camera in a scene, it was necessary to produce a Camera class for the application. This class is used for the OpenGL and RenderMan graphics pipelines. Figure 14.1.1 below shows the general form of the camera in its default position [2].

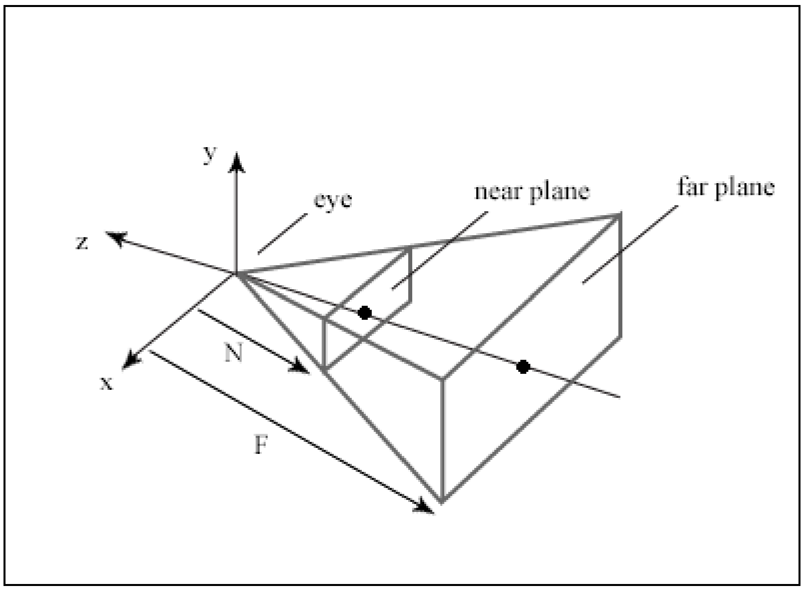


Figure 14.1.1: The camera in its default position

The eye is positioned at the origin of an axis with the pyramid aligned to the z-axis. The rectangular pyramid defines the viewing volume with its opening set to the viewing angleθ. The near plane and the far plane are defined perpendicular to the axis of the pyramid. Two rectangular windows are formed where these planes intersect the pyramid and the aspect ratio of the windows can be set in the application. Any points within the viewing volume are projected on to the view plane and any points outside are clipped off.

The shape of the camera's view volume is contained in the projection matrix. The transformations of the camera away from the default position are part of the modelview matrix. The modelview matrix is the product of the matrix M that expresses the modelling transformations applied to objects and matrix V that expresses the transformations that position the camera. Figure 14.1.2 over the page shows a coordinate system attached to the camera to define its orientation [2].

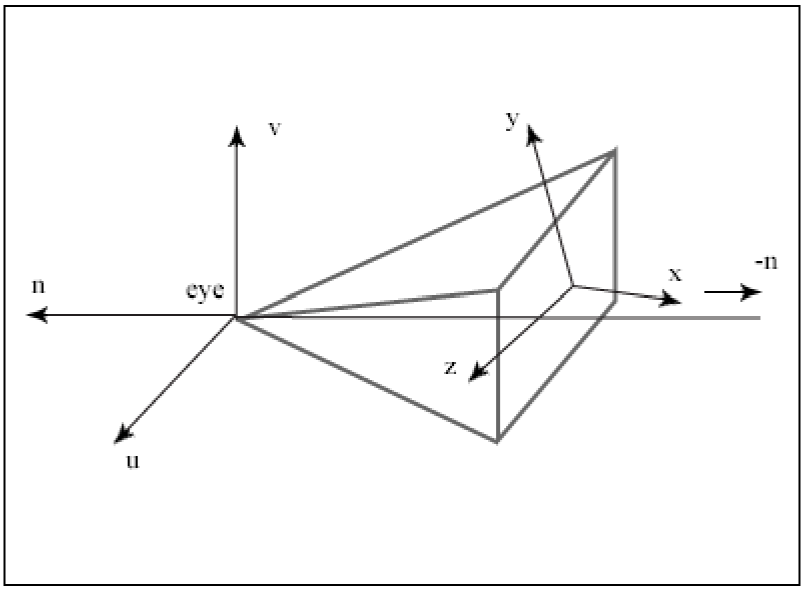


Figure 14.1.2: Attaching a coordinate system to the camera

The three axes of the coordinate system are usually defined by the vectors u, v and n. The camera looks down the negative n-axis, the v-axis points upwards and the u-axis points to the right.

The Camera class has a series of methods to create and manage the modelview matrix, projection matrix and shape of a camera. It also has methods for transforming the camera in space. Although position is fairly trivial to describe, orientation proves more difficult. Figure 14.1.3 below, uses the aviation terms to help specify orientation.

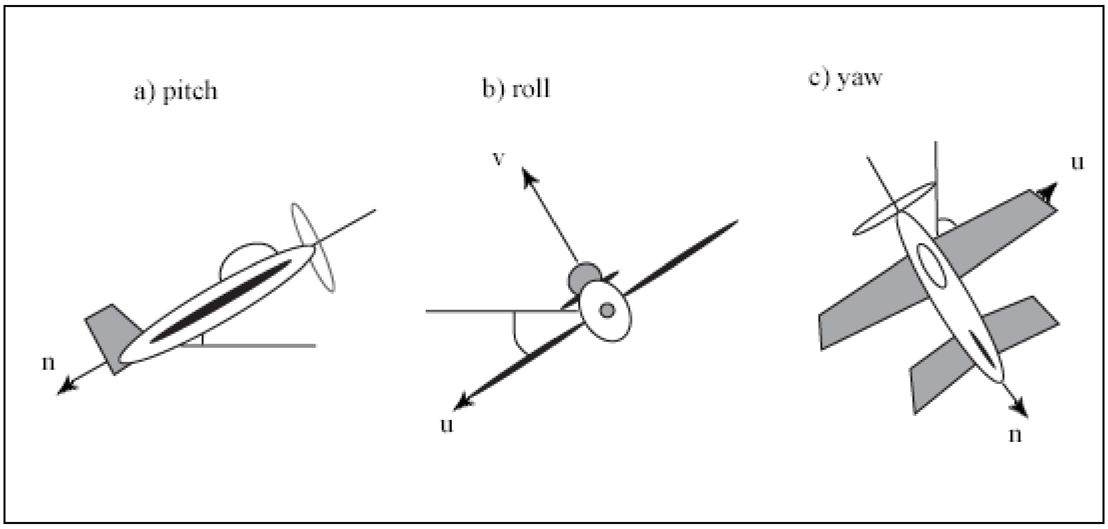


Figure 14.1.3: A plane's orientation relative to the "world"

- Pitch: The angle between the horizontal plane and its longitudinal axis, which involves a rotation about the u-axis.
- Roll: The angle between the horizontal plane and its latitudinal axis, which involves a rotation about the n-axis.
- Yaw: The angle between the vertical plane and its longitudinal axis, which involves a rotation about the v-axis.

A new camera is instantiated by providing:

- *Eye*: The location of the camera.
- *Look*: The point that the camera is looking at.
- *Up*: The up direction of the camera.
- Information pertaining to the shape of the camera.

```
n must be parallel to the eye vector,
∴ n = eye − look
u is perpendicular to up,
∴ u = up × n
v is perpendicular to u and n,
∴ v = n × u
Normalize u, v and n
```

It is worth noting that *u* points to the right as viewed down the negative *n*-axis.

As well as these three rotations a “slide” method was also implemented. This allows the camera to move along its own axes providing three additional degrees of freedom.

14.2 Cloth Object

The cloth object is rendered as a series of polygons in the OpenGL view. The Cloth class implements a simple draw method that iterates though the cloth polygons displaying them as OpenGL triangles. Additionally, the draw method can display the cloth particles, the three different spring types and the polygon normals. The normals are displayed as OpenGL lines, where the endpoints of each line are defined as the centroid of the polygon and the top of the normal.

14.3 FBX Object

The FBX object is rendered as a series of lines in the OpenGL view. The FBXInterface class implements a simple draw method that iterates through all the FBX polygons displaying them as one continuous OpenGL line loop.

Chapter 15 RenderMan Rendering

15.1 RibExporter Class

The RibExporter class contains methods for writing RenderMan Interface Bytestream (RIB) file commands. The *Open* method opens a stream interface to a RIB file whose filename is related to the current frame of the simulation. If the rendering flag is enabled, each frame of the cloth simulation produces a separate RIB file, which is essentially just a text file. Once the relevant information has been added to the RIB file, the file stream is closed.

The main C++ application scripts contain a render function that sets the key RIB file attributes. Some of these include the file format, camera information, light sources, colours and calls to RIB writing methods within the Cloth and FBX Interface classes. A much reduced version of a RIB file is shown in Appendix E, highlighting its major features. Additional information on the Cloth and FBX Interface RIB writing methods is given below.

15.2 Camera Class

One of the main reasons for implementing a Camera class was to ensure that the simulation scene viewed in the OpenGL window could be rendered out using Pixar's RenderMan. The Camera class includes a method to write out the camera information to the RIB file.

The RenderMan specification requires both the position of the camera and the projection of the camera view onto the image. Fortunately there are numerous similarities between RenderMan and OpenGL when it comes to defining cameras and rendering parameters.

OpenGL requires the view angle, aspect ratio and near and far clipping planes to define the perspective projection. This information can be used directly in the RIB file with the appropriate commands.

15.2.1 Clipping

Firstly, the clipping data is written to the RIB file using the following command.

```
Clipping nearplane farplane
```

Clipping sets the near and far clipping planes that are used by RenderMan to optimise the perspective projections. This command is important as the mathematics involved in perspective divides can waste floating point accuracy and the default values should be overridden [13].

15.2.2 Projection

The projection data is written to the RIB file using the following command.

```
Projection type parameterlist
```

Projection specifies the type of camera projection and where on the projection plane objects in the scene will appear. A common parameter supplied to this command is the *field of view*, which is the same value as that used for the OpenGL scene. Specifying a different *field of view* has the effect of zooming the image [13].

15.2.3 Scale

The scale command is written to the RIB file using the following command.

```
Scale sx sy sz
```

This command is very important in the context of converting the OpenGL camera information to RenderMan. This is due to the direction of the z-axis. The positive z-axis for OpenGL points out of the screen, whilst the positive z-axis for RenderMan points into the screen. Therefore to convert the handedness of the data, a scale of negative one in the z-axis is required.

15.2.4 ConcatTransform

The ConcatTransform command is written to the RIB file using the following command.

```
ConcatTransform matrix
```

ConcatTransform premultiplies the *matrix* parameter into the current transformation matrix [13]. The matrix supplied to this command contains the same sixteen floating point numbers as the OpenGL modelview matrix.

15.3 RenderMan Polygons

The RenderMan specification implements two types of polygons – convex and concave. Convex polygons fall under the heading of *polygons* and can be defined by the condition that every vertex can be connected to every other vertex by a line that stays within the polygon [13]. Concave polygons fall under the heading of *general polygons* and can contain indentations or holes.

In addition to the polygon packaging style, RenderMan also provides polyhedra where sets of polygons share vertices. Polyhedra provide the benefit of improved database descriptions due to the reduction of redundant vertex information of describing individual polygons. Figure 15.3.1 below, shows examples of each of the four types of polygonal primitives.

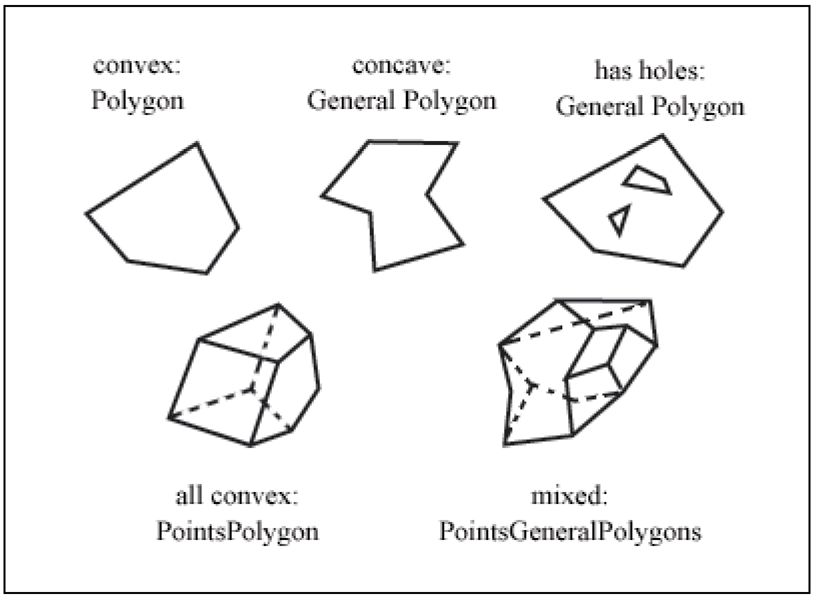


Figure 15.3.1: Examples of each of the four types of polygonal primitives

Vertex information for a polygon is stored in a parameter list which is an array of floating-point 3D vertex coordinates. For example, a square would have an array of twelve floating-point numbers. Various polygon types were used in the FBX Interface class and are discussed in section 15.7.

The Polygon command is written to the RIB file using the following command:

```
Polygon parameterlist
```

The PointsPolygons command is written to the RIB file using the following command:

```
PointsPolygons nverts vertids parameterlist
```

15.4 RenderMan Parametric Patches

Parametric patches are commonly utilised as the primitives for most RenderMan scenes [13]. They are four sided and have two pairs of opposing sides called parametric directions. Patches are curved and hence they do not suffer from polygonal approximation artefacts.

RenderMan implements three general primitives – bilinear patches, bicubic patches and non-uniform rational B-splines (NURBS). Bicubic patches are defined by the cubic curves on each edge, characterized by their basis functions [13]. There are several basis functions built into RenderMan and additional functions can be added. Vertex information for a patch is stored in a parameter list in a similar manner to that for polygons.

The Patch command is written to the RIB file using the following command:

```
Patch type parameterlist
```

15.5 RenderMan Subdivision Meshes

Subdivision meshes combine some of the good properties of several of the different surface types. They are smooth when compared to polyhedral meshes and they don't have the limitation of having to be rectangular. Subdivision meshes also require far fewer points than polyhedral meshes to produce a good quality smooth surface fit [13].

The SubdivisionMesh command is written to the RIB file using the following command:

```
SubdivisionMesh scheme nverts vertids tags nargs intargs
floatargs parameterlist
```

15.6 Cloth Object

The Cloth class has two methods for writing the Cloth object information to the RIB file, one for bicubic patches and the other for a subdivision surface. The bicubic patch construction algorithm simply loops through the cloth particles grouping sets of sixteen vertices. The subdivision surface construction algorithm is more complex as the vertex information requires an additional sorting pass before writing the data to the RIB file.

15.7 FBX & Obj Objects

The FBX and Obj objects can be written to the RIB file as Polygons, PointsPolygons or Subdivision surfaces. During the RIB construction process, the application passes a list of polygons to the RibExporter class that implements methods for each of the surface types. Figure 15.7.1 shows the FBX character rendered as polygons and as a subdivision surface.

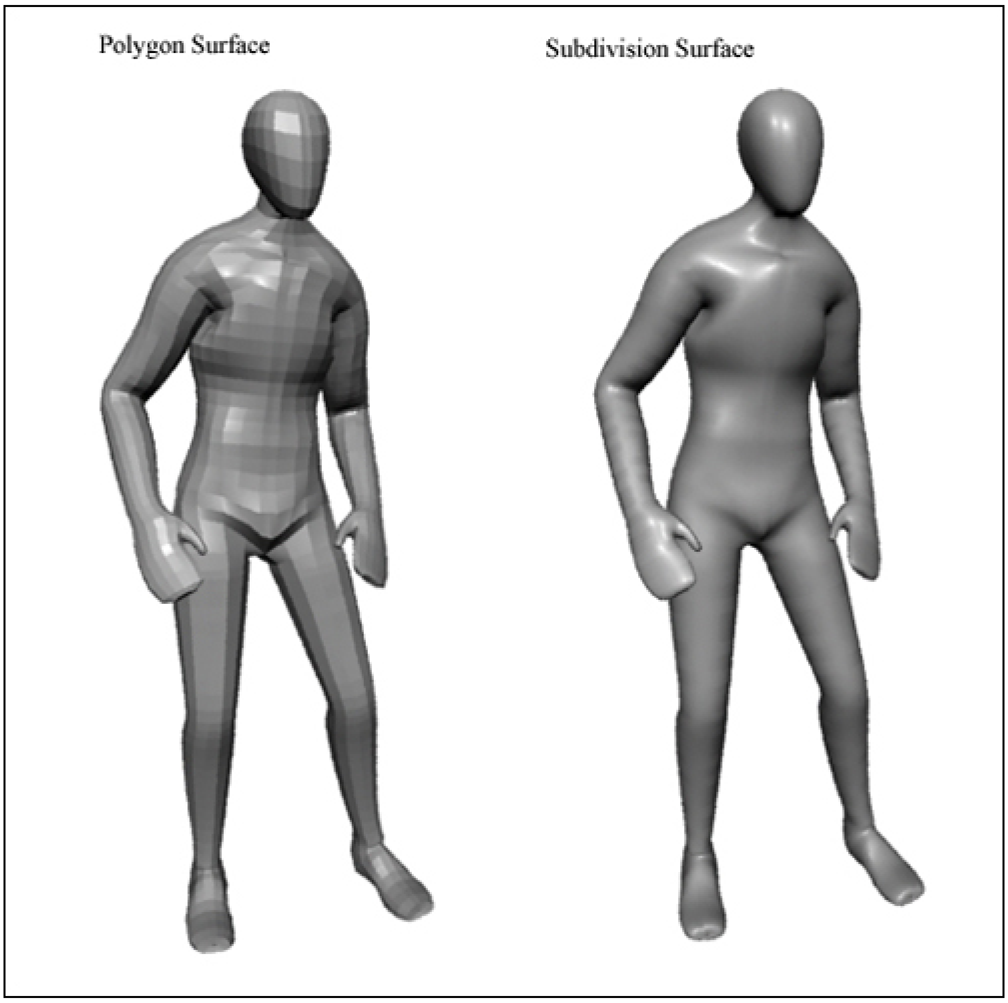


Figure 15.7.1: FBX character rendered using polygons and subdivision surfaces

15.8 Rendering Process

From the RIB file example shown in Appendix E, it is possible to see placeholders for certain key parameters, for example:

```
Surface "__SURFACE1__" __PARAMETERS1__
```

These placeholders were replaced during the rendering process by running a bash script. This method of rendering enabled multiple shaders, opacity and matte information to be changed without having to re-run any simulations. Shadow mattes for the skirt simulation were also rendered efficiently using a similar process. The bash scripts can be found in Appendix F.

Figure 15.8.1 below shows a RenderMan render of the FBX character and cloth skirt.

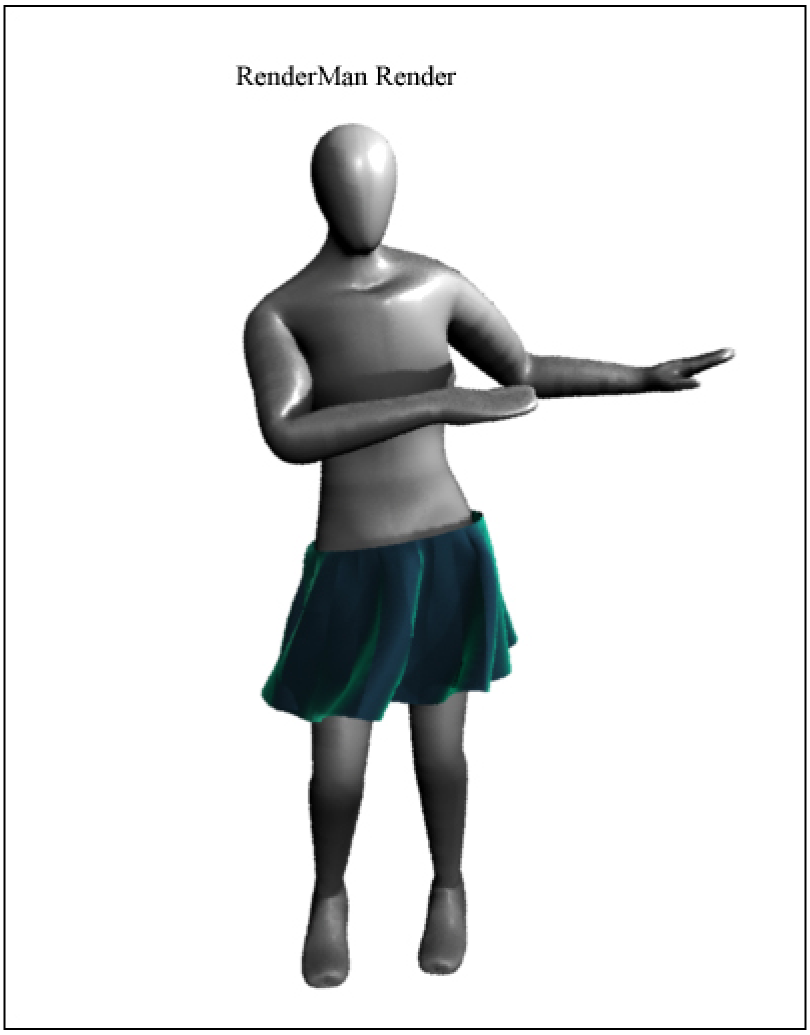


Figure 15.8.1: RenderMan render of the FBX character and cloth skirt

Chapter 16 Pipeline & Project Management

16.1 Project Management

In order to produce a completed project on time, it was important to look at some of the main theories of project management.

Project management in the modern sense began in the early 1960's, where businesses realised the benefits of organising work around projects and the need to co-ordinate and communicate this work between departments and professions. The United States space program was one of the first organisations to employ project management.
Project management can be summarised using the following diamond shown in Figure 16.1.1 below.

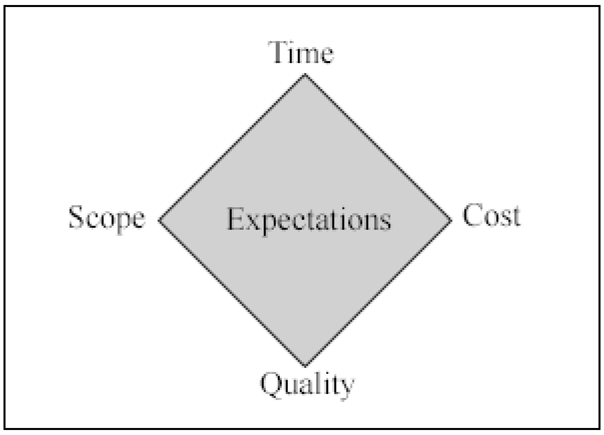


Figure 16.1.1: Project management diamond.

A project goes through four phases during its life. These phases are briefly discussed with regard to this project.

16.1.1 Project definition

The main goal for the project was to produce a simple C++ and OpenGL application to simulate cloth. Some of the objectives and critical success factors set out at the start of the project included implementing:

- A mass spring model for the cloth
- Various integration methods
- Collision detection and response
- The use of motion capture data

- A RenderMan rendering pipeline.

16.1.2 Project Initiation

A number of systems were put in place before any work commenced. These included timescales and deadlines for each group member, a set of project requirements, version control and unit testing of the code base, and strategies for dealing with unforeseen problems. A large amount of research was also undertaken before any coding was started in earnest.

16.1.3 Project Control

To ensure that the project stayed on track, regular meetings, progress reporting and code reviews were held. This process quickly highlighted any potential issues and provided reasoned solutions to any problems. It also allowed all members of the group to remain familiar with the code base. There were numerous occasions where an issue was quickly resolved by brainstorming and coding as a group. This was one of the major strengths of the project.

16.1.4 Project Closure

This simply involved final review meetings, commenting of the code base and production of the simulation renders.

Employing a number of key project management doctrines allowed the project to progress successfully, be monitored to minimise any potential problems, and keep the group members motivated. Some of the systems that were put in place are described below.

16.2 Version Control

In essence, version control manages changes to information over time. It is a well established tool for programmers who often make changes to software source files and then need to revert to previous versions. It also allows multiple programmers to work on the same source files and then enables future integration of the data.

Subversion [9], which is a free / open-source version control system, was selected for this project. It works by placing a tree of files into a central repository. The repository acts much like a file server, except that it remembers every change made to a file or folder.

At the start of the project, a Subversion central repository was created. The basic empty folder structure for the project was imported into the repository. Each member of the group then exported the project from the repository to create their own working copy. At regular intervals during the development process each member would update their local copy, make any necessary amends and then submit these changes back to the repository. Hence, Subversion uses a "copy-modify-merge" model.

At the start of the project there was an issue with the repository locking and corrupting. This meant that the entire repository had to be re-built and each group member had to export a new working copy. Through investigation, it was discovered that the repository uses a series of database files to store the log messages that are committed each time a user submits a change. When one of these database files exceeds one Megabyte of data, a new database file is created. Due to the file permissions system on Linux, if a member of the group caused a new database file to be created, it would have file read / write permissions pertaining to that member. The next time any group member attempted to run an update, Subversion would throw an exception and the repository would be corrupted.

As the group did not have root access on the server, it was not possible to implement Subversion's own server process to overcome this issue. However, a solution was found by creating a Bash script that could be run every time data was submitted to the repository. This basic script would check the database files in the repository created by the group member running the script and update any file permissions to be read / writable to everyone. The bash script is shown in Appendix F. Version control proved to be an invaluable tool for this project.

16.3 Data Backups

To reduce the risk of losing important data, daily backups of the Subversion repository and each group members' working copy were made onto a portable USB hard drive. Coupled with version control, this process helped to ensure the integrity of the data.

16.4 Communication & Tasks

Communication and strategic direction are key areas that can affect the success of a group project. Some of the processes that were implemented to ensure that each group member was kept informed and motivated are discussed in the next four sections.

16.5 Project Plan

At the start of the project, the group held a brainstorming and application design meeting. After the initial application design had been agreed, a Gantt chart was produced showing the tasks and timescales. This visual aid was important for keeping the project on track. The Gantt chart had a few minor revisions as the project progressed, but did not differ substantially from the initial plan. The final project Gantt chart is shown in Appendix G.

16.6 Daily Meetings

Every day during the project, the group would have a quick informal meeting to discuss what was being worked on, any potential issues and to update the project plan as necessary. There was however a deliberate effort to ensure that these meetings were kept short and to the point.

16.7 Code Reviews

Regular code reviews were held during the development lifecycle of the project. These reviews helped to ensure the quality of the code and allowed all members of the group to have a shared understanding of the application.

16.8 Pair Programming

Towards the end of the project, the relatively recent practise of pair programming was used. This involves two software developers working side-by-side at one computer, collaborating on the same algorithm, design or test. This process improved the shared understanding of the application, often produced higher quality code and sped up bug fixing.

16.9 Unit Testing

Unit testing was used increasingly during the project to isolate separate components of the application. One of the main areas where it was used was in testing the collision detection and response classes. Developing and debugging these classes with the FBX data, skirt cloth object and collisions enabled would have been a slow process. To overcome these difficulties a unit test was developed by implementing the following.

- Sphere class: A polygon sphere was created in Maya™ from Alias® and a simple MEL script was used to obtain the vertex information. The vertex data was used in the Sphere class constructor to create a polygon sphere. This method of creating the sphere object was used before the Obj Loader class was created. The Sphere class also had basic methods for drawing and moving the sphere in the OpenGL window.
- Simple Cloth Object: A simple square cloth object was defined with considerably less polygons than the skirt object.

By using the simple polygon objects it was possible to provide real-time visual feedback. By drawing the axis aligned bounding boxes, colour coding the collision points and from the movement of the cloth, it was possible to quickly test and implement various methods in the collision classes. Another important feature of unit testing is its ability to separate interface from implementation. By testing the collision classes by this means, it is possible to minimise the dependencies within this complex area of the application.

Chapter 17 Cloth Shaders

17.1 Introduction

As part of the project, several RenderMan shaders were created to texture the cloth models. This was necessary to enhance the visual appearance of the cloth, and to give the viewer a clue as to the type of fabric the model was simulating. There are several parameters that can be changed within the cloth model, which result in behaviour that would be typical of a particular type of cloth. These include increasing the springs to a high degree of stiffness, which will cause the cloth to behave like wool, however if the cloth is given a high degree of elasticity, it behaves more like Lycra™.

The shaders created for the cloth include two types of wool, satin, velvet, Lycra™, and tartan. The following sections detail the structure of the RenderMan shaders.

17.2 Wool 1

The appearance of the first wool shader is to simulate a stiff, knitted skirt. Woollen textures are slightly more complex than woven fibres [32]. The knit is visible through the pattern, so to achieve this effect, the shader is composed of two parts; procedural pattern generation, and an imported texture file.

The texture file is simply a magnified image of a swatch of woven wool. It was converted to the RenderMan .tx file format, which is standard for RenderMan texture files. It was then used as a bump map to give the impression of knitted fabric.

The stripe pattern is generated based on the s and t coordinates of the current point being shaded. These coordinates are multiplied by a frequency value which is then used to generate a noise value (separate ones for the s and t direction). These values generate the stripes and are then used to blend between the colours of the stripes on the skirt.

The two sets of stripes are generated (horizontal and vertical) and then added together. The stitching texture is then multiplied by this value to give it the textured appearance. Figure 17.2.1 below shows the wool 1 shader.

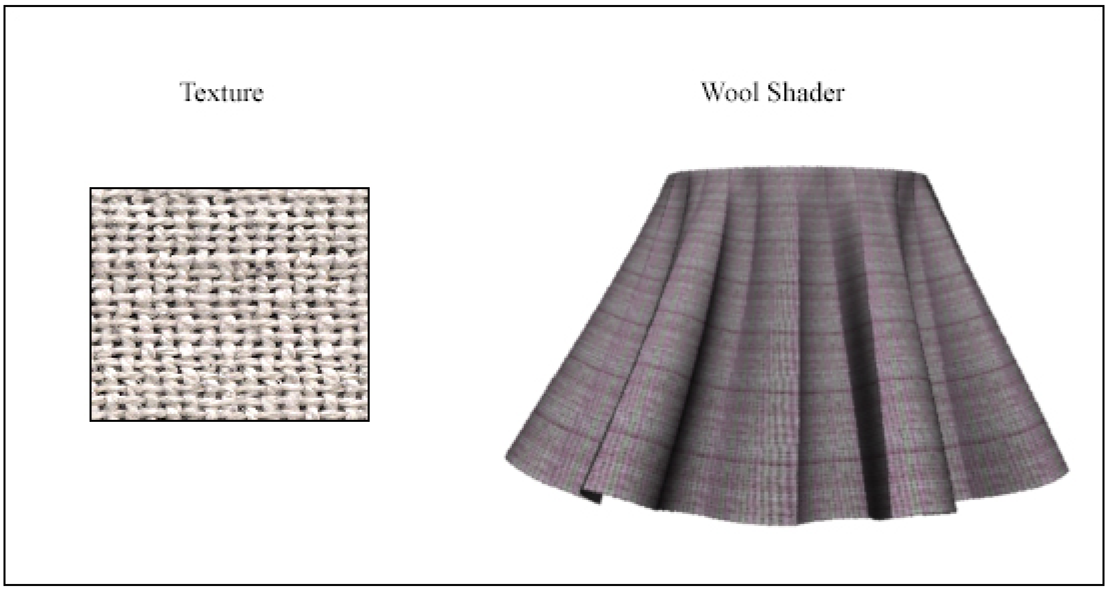


Figure 17.2.1: Wool 1 Shader

17.3 Wool 2

The second wool shader is based on the heavy woven wool fabric that is used in tailoring suits and skirts. It also takes in a texture file, however this time it is taken in as float values instead of colour values. These float values are used in the bump mapping to generate the appearance of threads.

The colour of the threads is generated using two mix functions, and the colours are mixed based on pseudo-noise values generated by dividing the t component by the s component or the s component by the t component.

The resulting value is multiplied by a high frequency number and then used to generate the noise value. The two colours from the mix functions are multiplied together along with the float value from the texture file. Figure 17.3.1 below shows the wool 2 shader.

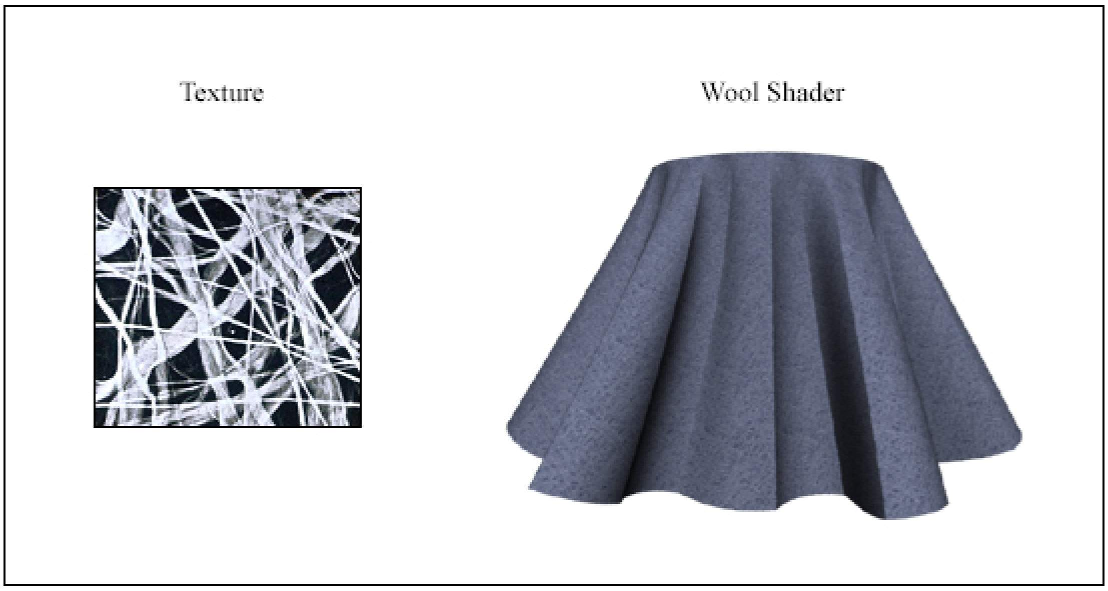


Figure 17.3.1: Wool 2 shader

17.4 Tartan

The tartan shader is generated using the same technique as the method used to create the stripe pattern on the wool shader. The s and t coordinates of the current point being shaded are multiplied by a large frequency value, and are then used to generate a noise value. This is performed separately for the s and t directions, and these values are used to generate the different stripes and blend between the colours. Figure 17.4.1 below shows the tartan shader.

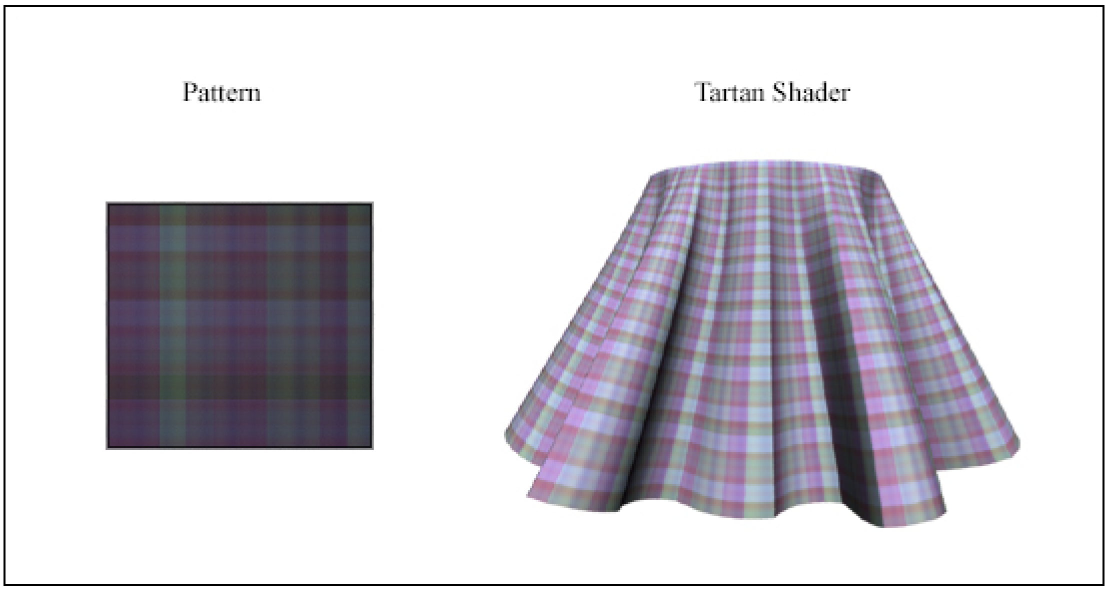


Figure 17.4.1: Tartan shader

17.5 Satin

Satin fabric appears very smooth and tactile. It is recognizable because of its light reflectance properties; it is highly reflective, and if it is blended from fibres of different colours, its appearance can change colour in the light [33].

The Satin shader is a modified version of the anisotropic shader from Essential RenderMan [34]. Anisotropic surfaces reflect their light in one direction and so an important part of the shading calculations are based on the lights in the scene. It utilises an illuminance loop, which loops through each light in the scene for the current point on the cloth being shaded. This then calculates the shine of the fabric based on the reflectance properties of satin material.

Anisotropic materials reflect the light according the direction of their threads. In satin, all the threads are facing the same direction, so the resulting light reflection is all in the same direction. This gives satin its smooth appearance. In order to reproduce this in a shader, the lights in the scene need to be broken down and each light needs to be considered separately. Every time around the illuminance loop, the angle of the light hitting the cloth and its reflected ray is calculated, by computing the dot product of the incident ray and the ray towards the viewer.

The direction of these threads is considered to be the u direction. When the material is at a 90 degree angle to the light, the brightness will be at its greatest. Therefore, the size of the angle between the incident ray and the surface normal is included so the areas of higher reflectance can be calculated. The size of this angle is then also used to blend between different colours to generate a specific shine colour. Figure 17.5.1 over the page shows the satin shader.

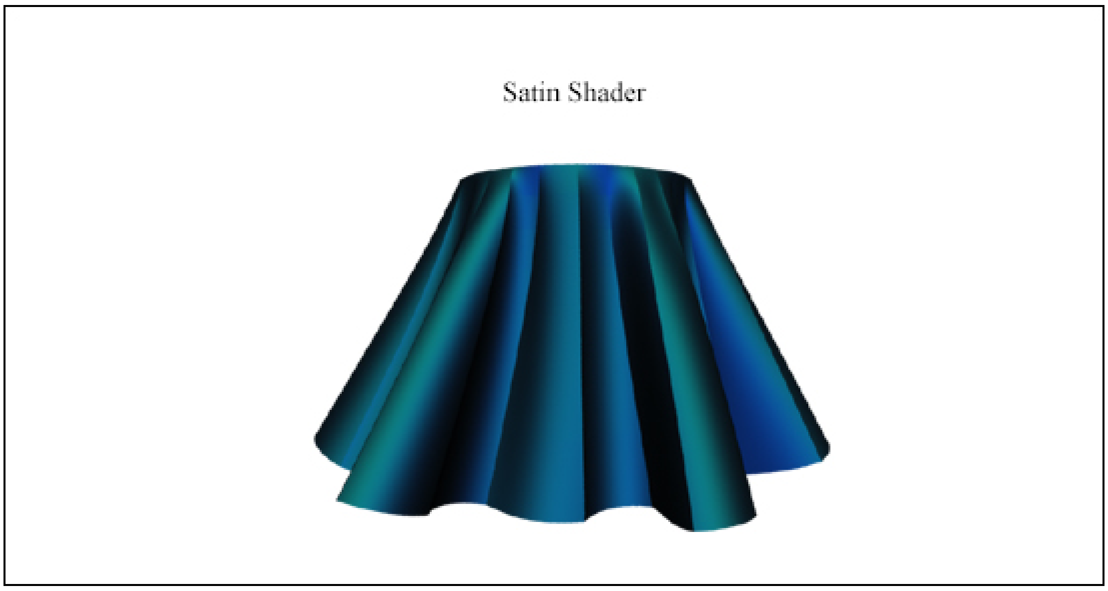


Figure 17.5.1: Satin shader

17.6 Lycra™

Lycra™ fibres are incredibly elastic and are always blended with other fibres to create Lycra™ fabric. It is durable and stretchy, and often the elastic fibres in the cloth cause shininess due to the light reflecting off them [32].

In order to simulate this fabric, the shader was broken into two parts. It is composed of a texture file, used to generate a subtle bump map, which is then in turn used in an illuminance loop to calculate the reflectance.

The texture file is simply a cross hatching to give the appearance of microscopic stitching. The information read in from this file is then used to create a bump map, which calculates a new value for the surface point PP and its normal. These values are then passed into an illuminance loop.

The illuminance loop takes the dot product of the new normal from the bump map and the vector in the direction of the viewer. This is then used to calculate the reflectance angle. When the light shines on areas that are specific values from the texture file, they change colour between the fabric colour and the colour of the underlying elastic threads. These colours are parameterized and can be passed in to the shader by the user. Figure 17.6.1 below shows the Lycra™ shader.

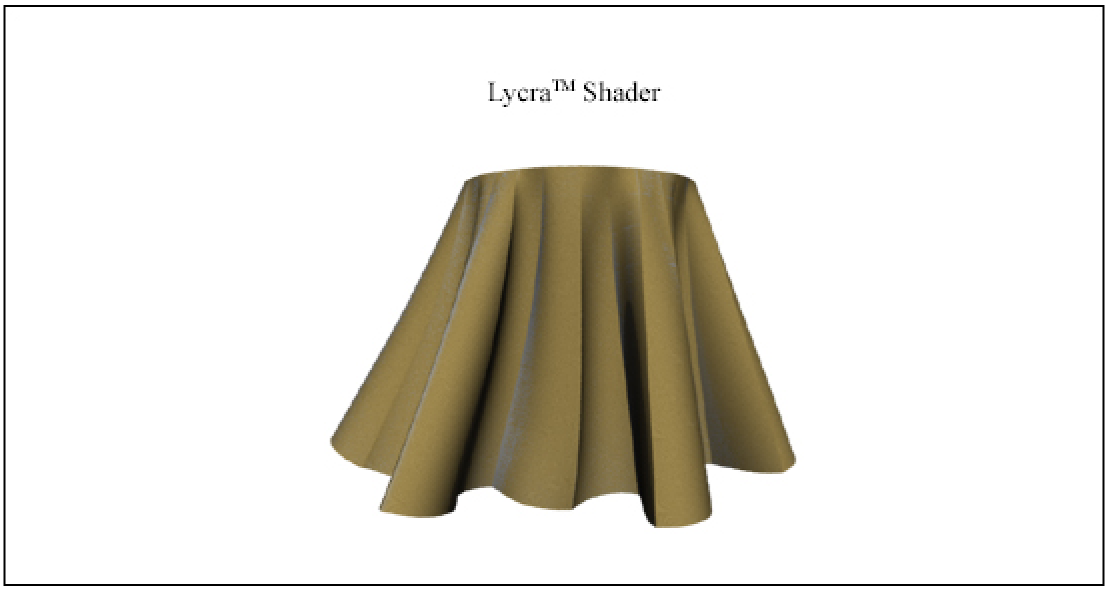


Figure 17.6.1: Lycra shader

17.7 Velvet

Velvet is a rich soft fabric, in which the cut threads are evenly distributed, with a short dense pile that gives its distinctive feel [35]. Velvet can be made from any fibre, and the sheen from the threads depends on their orientation.

The surface shader for velvet is based on the orientation of the light to the surface and the way it is reflected. Despite the threads being cut evenly, the surface of velvet is bumpy, so the light is scattered. However the treads are smooth which means the falloff of light is smooth with the edges being very soft.

In order to produce this effect with a shader that will have a noticeable sheen depending on its orientation to the light, an illuminance loop must be used to get the values of each light in the scene.

Initially, the derivative of the surface point P along the u direction is found and normalized. This value is then used in the illuminance loop, to calculate the angle between the fibres along the u direction and the light reflectance vector towards the user. This angle is then multiplied by a reflectance value and a highlight value resulting in the sheen value [36].

Next the scattering is calculated using the angle between the surface normal and the vector in the direction of the viewer. The size of this angle determines how much scatter the viewer can see and is added to the sheen value. Figure 17.7.1 shows the velvet shader.

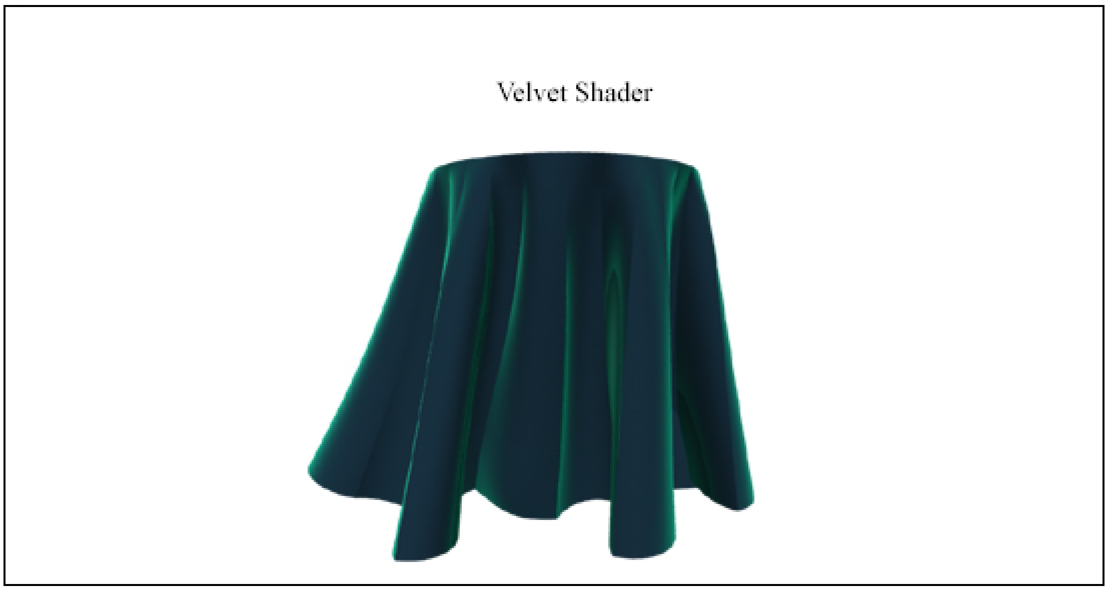


Figure 17.7.1: Velvet shader

Chapter 18 Conclusion

The cloth simulation project undertaken was a success. A motion capture driven CG model wearing a skirt was successfully simulated in an OpenGL environment and exported to RIB files for rendering in a RenderMan compliant renderer. After two weeks of independent research, the entire application was implemented in six weeks. Two successful renders of a stable clothed CG model have been produced as well as two test renders of a sheet interacting with a static arbitrary object.

The application has scope for extension and improvement. A more robust collision framework could be implemented to eradicate any cloth / model intersections; a user interface could be developed as a front end for more flexible interactivity with the cloth simulation, motion capture driven model and the computer generated environment; and aerodynamics and the ability to stitch panels of cloth together would allow for the creation of more interesting items of clothing. Finally, an implicit integration solver would allow for more dynamic FBX data to be used during simulation due to its increased stability.

Excellent group dynamics, communication and project management meant that milestones were routinely achieved and weekly reviews and pair programming allowed for efficient and effective problem solving. Ultimately, hard work and motivation from all members of the team has resulted in a highly successful project.

Chapter 19 Acknowledgements

The group are indebted to and would like to thank Professor John Vince, Jon Macey, Professor Jian J Zhang, Jian Chang, Andy Cousins, Shane Aherne and Mark Doherty.

Chapter 20 Bibliography

[1] PIERCE, F. T. *On the Geometry of Cloth Structure*. In Journal of the Textile Institute, 28: T45 – T97, 1937.

[2] HILL, F.S., 1990. *Computer Graphics using OpenGL, Second Edition*. Prentice Hall.

[3] HOUSE, D.H. AND BREEN, D.E., 2000. *Cloth Modelling and Animation*. A K Peters.

[4] WEIL, J. *The Synthesis of Cloth Objects*. Computer Graphics Proceedings (SIGGRAPH) 20(4): (1986), 49-53.

[5] RUDOMIN, I.J. *Simulating cloth using a mixed Geometric – Physical method*. Ph.D. Thesis, University of Pennsylvania, 1990.

[6] BREEN, D.E., HOUSE, D.H. *A Physically Based Particle Method of Woven Clot*h. The Visual Computer 8(5-6): (June 1992), 266.

[7] EBERHARDT et al. *A Fast, Flexible Particle System Model for Cloth Draping*. IEEE Computer Graphics and Applications, 16(5): (September 1996), 52–59.

[8] KUNII, T.L. AND GOTODA, H. *Singularity, Theoretical Modelling and Animation of Garment Wrinkle Formation Process*. The Visual Computer, 6960: (1990), 326–336.S

[9] COLLINS-SUSSMAN, B., FITZPATRICK, B.W. AND PILATO, C.M. *Version Control with Subversion*. Available from: http://svnbook.red-bean.com/en/1.0/svn-book.html [Accessed 20 July 2005].

[10] BREEN, D.E., HOUSE, D.H. AND GETTO, P.H. *On the Dynamic Simulation of Physically-Based Particle System Models*. Third Eurographics Workshop on Animation and Simulation Proceedings (Cambridge, UK, September 1992).

[11] HAUMANN, D.R. AND PARENT, R. *The Behavioural Test Bed. Obtaining Complex behaviour from Simple Rules*. The Visual Computer 4: 332- 347, 1998.

[12] PROVOT, X. *Deformation Constraints in a Mass Spring Model to Describe Cloth Behavior*. Proceedings Graphics Interface (1995), 146-154.

[13] APODACA, A.A. AND GRITZ, L., 2000. *Advanced RenderMan*. Morgan Kaufmann.

[14] WEISSENSTEIN, E.W. *Numerical Integration*. Available from: http://mathworld.wolfram.com/NumericalIntegration.html [Accessed 19 August 2005].

[15] FITZPATRICK, R. *Integration Methods*. Available from: http://farside.ph.utexas.edu/teaching/329/lectures/node59.html [Accessed 5 August 2005].

[16] THALMANN, N. AND VOLINO, P. *Comparing Efficiency of Integration Methods for Cloth Simulation*. Computer Graphics International, 2001.

[17] PARENT, R. *Computer Animation Algorithms and Techniques*. Morgan Kaufmann.

[18] CONTE, S.D. AND DE BOOR, K.W., 1980. *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill Higher Education.

[19] WANG, M. *Physically Based Cloth Simulation*. Available from: http://www.booyah.com/article07-dx9.html [Accessed 5 June 2005].

[20] LANDU, R. *Computational Physics at Oregon State University*. Available from: http://www.physics.orst.edu/~rubin/nacphy/ComPhys/DIFFEQ/mydif2/node5.html [Accessed 4 June 2005].

[21] BARAFF, D. AND WITKIN, A. *Large steps in cloth simulation*. In Computer Graphics (SIGGRAPH 1998 Proceedings). ACM Press, 43-54.

[22] DUMMER, J.*A. Simple Time-Corrected Verlet Integration Method.* Available from: http://www.gamedev.net/reference/articles/article2200.asp [Accessed 2 June 2005].

[23] KAUPPILA, M. *Numerical integration (Verlet vs. Implicit)*. Available from: http://forum.europeanservers.net/cgi-bin/v.eur?910861 [Accessed 27 May 2005].

[24] Unknown. *Verlet Integration*. Available from: http://www.answers.com/topic/verlet-integration [Accessed 27 May 2005].

[25] BOURG, D.M. *Five Steps to Adding Physics-Based Realism to Your Games*. Available from: http://www.linuxdevcenter.com/pub/a/linux/2001/11/01/physics.html?page=last [Accessed 24 August 2005].

[26] SEYAM, A.M. *Textile World: Technology Weaving*. Available from: http://www.textileworld.com/News.htm?CD=1294&ID=3416 [Accessed 24 August 2005].

[27] PARENT, R. *Collision Response*. Available from: http://www.cse.ohio-state.edu/~parent/classes/888/Cloth/CollisionResponse.html [Accessed 23 August 2005].

[28] BENSON, T. *Conservation Of Momentum, Glenn Research Centre*. Available from: http://www.grc.nasa.gov/WWW/K-12/airplane/conmo.html [Accessed 23 August 2005].

[29] JAKOBSEN, T. *Advanced Character Physics*. Available from: http://www.gamasutra.com/resource_guide/20030121/jacobson_02.shtml [Accessed 23 August 2005].

[30] GARBER, M. *Real Time Cloth Simulation*. Available from: http://www.maxgarber.com/projects/cloth/ [Accessed 26 August 2005].

[31] BARAFF, D. *Physically Based Modelling; Implicit Methods for Differential Equations*. Available from: http://www.pixar.com/companyinfo/research/pbm2001/notesd.pdf [Accessed 23 August 2005].

[32] UNKNOWN. *BBC bitesize – Design and Technology*. Available from: http://www.bbc.co.uk/schools/gcsebitesize/design/textiles/fibresrev5.shtml [Accessed 24 August 2005].

[33] UNKNOWN. *Osgood Textiles Satin*. Available from: http://www.fabrics.net/amysatin.asp [Accessed 23 August 2005].

[34] STEPHENSON, I., 2003. *Essential RenderMan*. Springer.

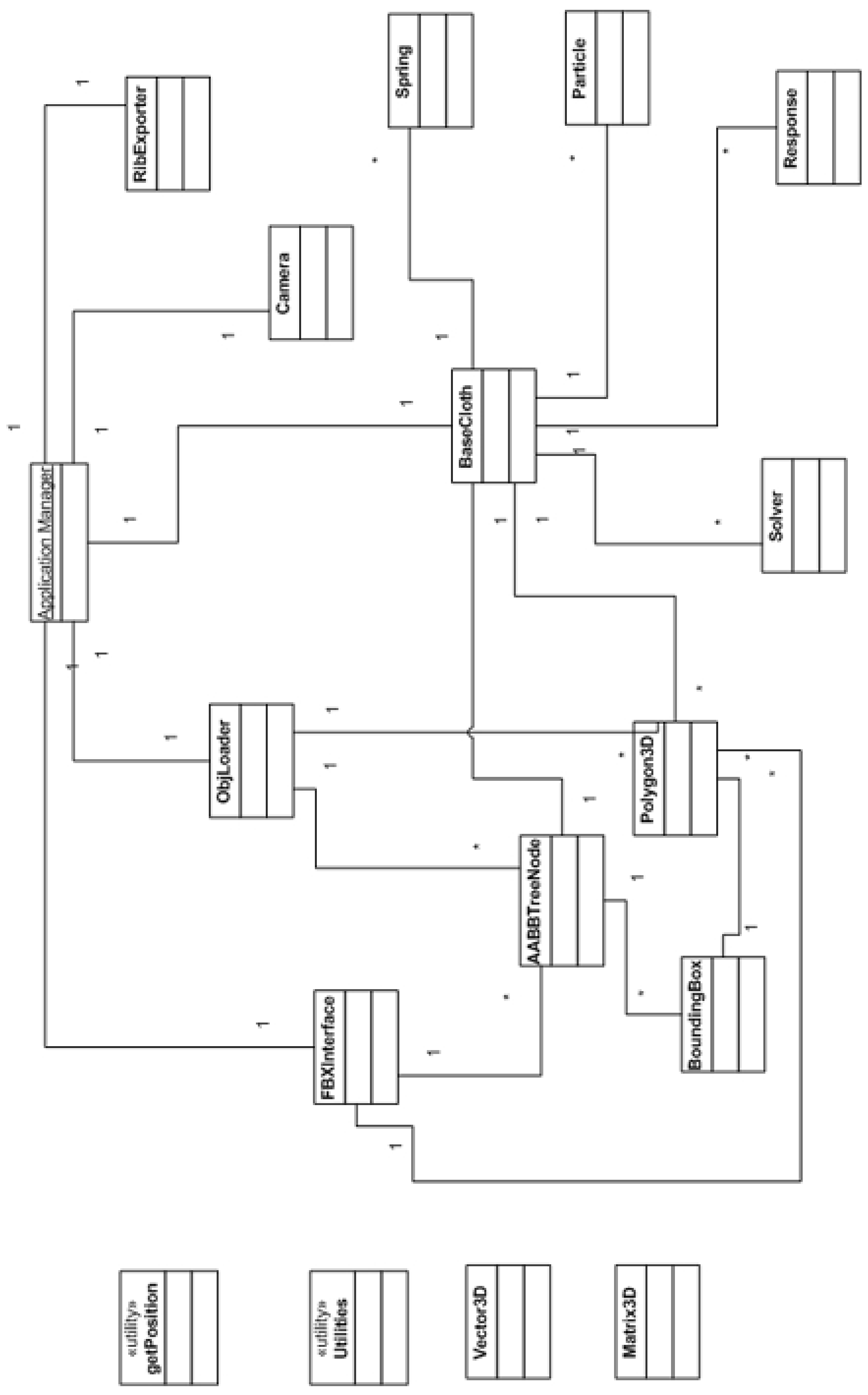
[35] UNKNOWN. *Wikipedia – Velvet*. Available from: http://en.wikipedia.org/wiki/Velvet [Accessed 23 August 2005].

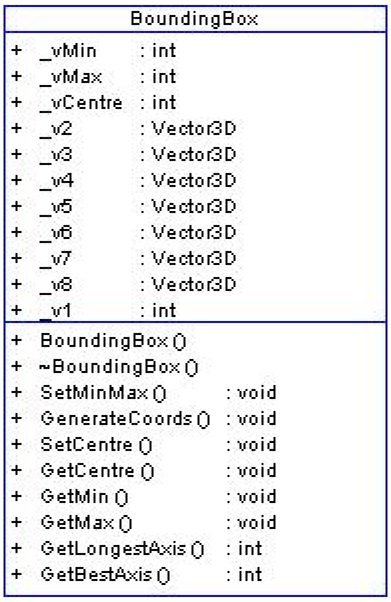
[36] WESTIN, S.H. *Stephen H. Westin Shaders*. Available from: http://www.renderman.org/RMR/Shaders/SHWShaders/ [Accessed 19 August 2005].

Chapter 21 Appendix A

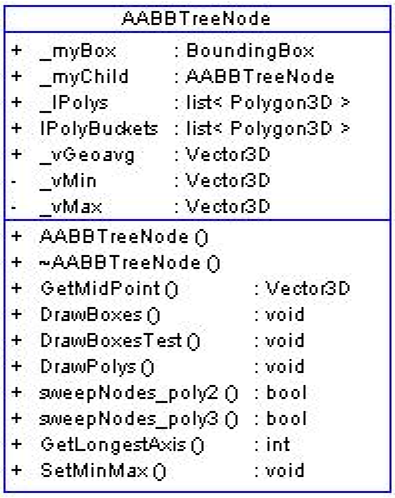
21.1 Class Diagram

The details for each class are shown in the following four pages.

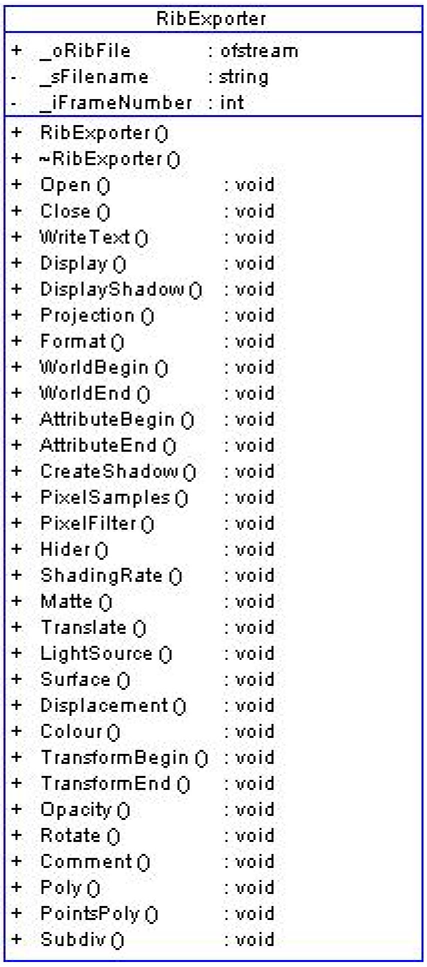




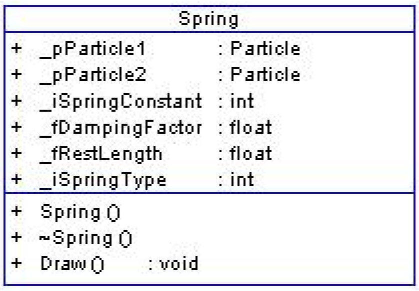
```
BoundingBox
+ _vMin      : int
+ _vMax      : int
+ _vCentre   : int
+ _v2        : Vector3D
+ _v3        : Vector3D
+ _v4        : Vector3D
+ _v5        : Vector3D
+ _v6        : Vector3D
+ _v7        : Vector3D
+ _v8        : Vector3D
+ _v1        : int
+ BoundingBox ()
+ ~BoundingBox ()
+ SetMinMax ()          : void
+ GenerateCoords ()     : void
+ SetCentre ()          : void
+ GetCentre ()          : void
+ GetMin ()             : void
+ GetMax ()             : void
+ GetLongestAxis ()     : int
+ GetBestAxis ()        : int
```



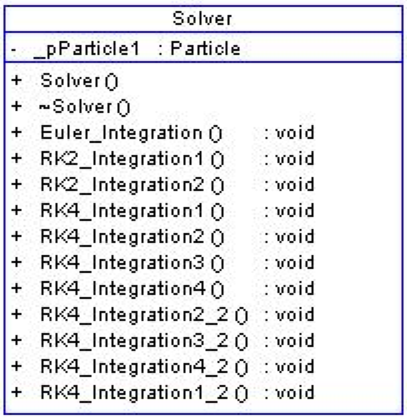
```
AABBTreeNode
+ _myBox          : BoundingBox
+ _myChild        : AABBTreeNode
+ _lPolys         : list< Polygon3D >
+ lPolyBuckets    : list< Polygon3D >
+ _vGeoavg        : Vector3D
- _vMin           : Vector3D
- _vMax           : Vector3D
+ AABBTreeNode ()
+ ~AABBTreeNode ()
+ GetMidPoint ()          : Vector3D
+ DrawBoxes ()            : void
+ DrawBoxesTest ()        : void
+ DrawPolys ()            : void
+ sweepNodes_poly2 ()     : bool
+ sweepNodes_poly3 ()     : bool
+ GetLongestAxis ()       : int
+ SetMinMax ()            : void
```



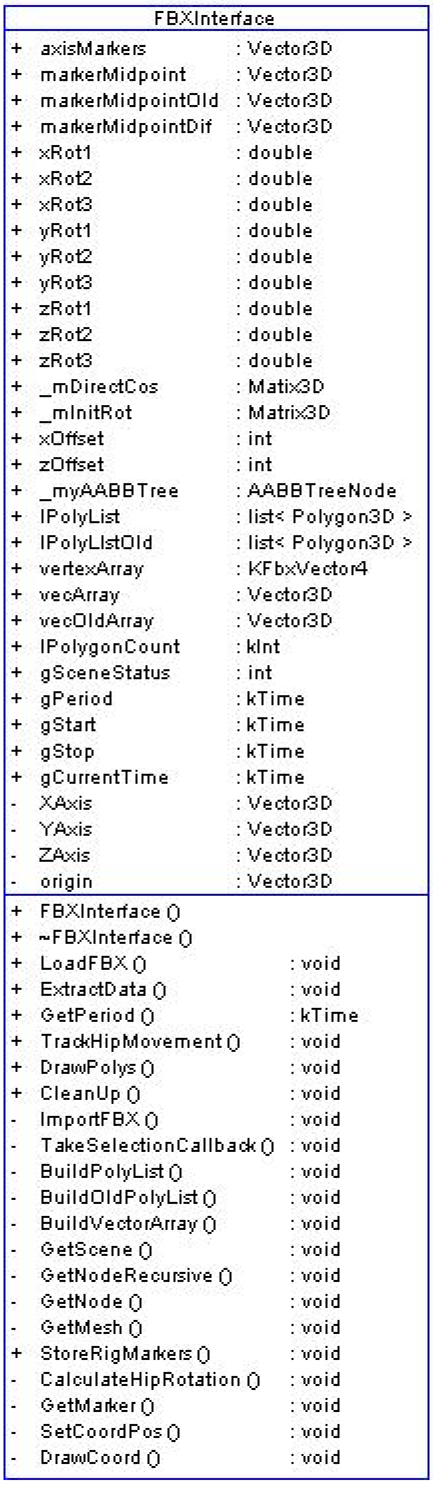
```
RibExporter
+ _oRibFile         : ofstream
- _sFilename        : string
- _iFrameNumber     : int
+ RibExporter ()
+ ~RibExporter ()
+ Open ()               : void
+ Close ()              : void
+ WriteText ()          : void
+ Display ()            : void
+ DisplayShadow ()      : void
+ Projection ()         : void
+ Format ()             : void
+ WorldBegin ()         : void
+ WorldEnd ()           : void
+ AttributeBegin ()     : void
+ AttributeEnd ()       : void
+ CreateShadow ()       : void
+ PixelSamples ()       : void
+ PixelFilter ()        : void
+ Hider ()              : void
+ ShadingRate ()        : void
+ Matte ()              : void
+ Translate ()          : void
+ LightSource ()        : void
+ Surface ()            : void
+ Displacement ()       : void
+ Colour ()             : void
+ TransformBegin ()     : void
+ TransformEnd ()       : void
+ Opacity ()            : void
+ Rotate ()             : void
+ Comment ()            : void
+ Poly ()               : void
+ PointsPoly ()         : void
+ Subdiv ()             : void
```



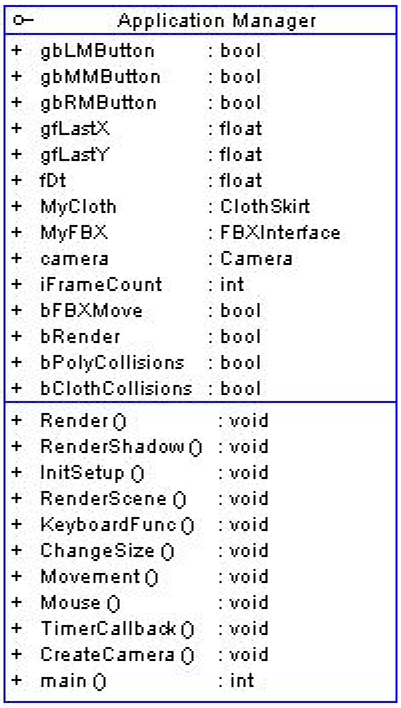
```
Spring
+ _pParticle1         : Particle
+ _pParticle2         : Particle
+ _iSpringConstant    : int
+ _fDampingFactor     : float
+ _fRestLength        : float
+ _iSpringType        : int
+ Spring ()
+ ~Spring ()
+ Draw ()       : void
```



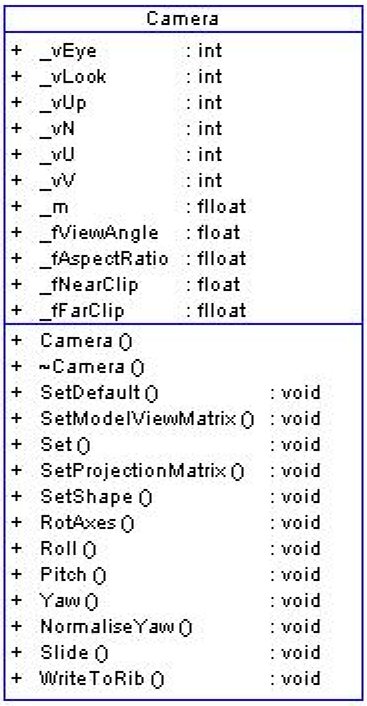
```
Solver
- _pParticle1   : Particle
+ Solver ()
+ ~Solver ()
+ Euler_Integration ()       : void
+ RK2_Integration1 ()        : void
+ RK2_Integration2 ()        : void
+ RK4_Integration1 ()        : void
+ RK4_Integration2 ()        : void
+ RK4_Integration3 ()        : void
+ RK4_Integration4 ()        : void
+ RK4_Integration2_2 ()      : void
+ RK4_Integration3_2 ()      : void
+ RK4_Integration4_2 ()      : void
+ RK4_Integration1_2 ()      : void
```



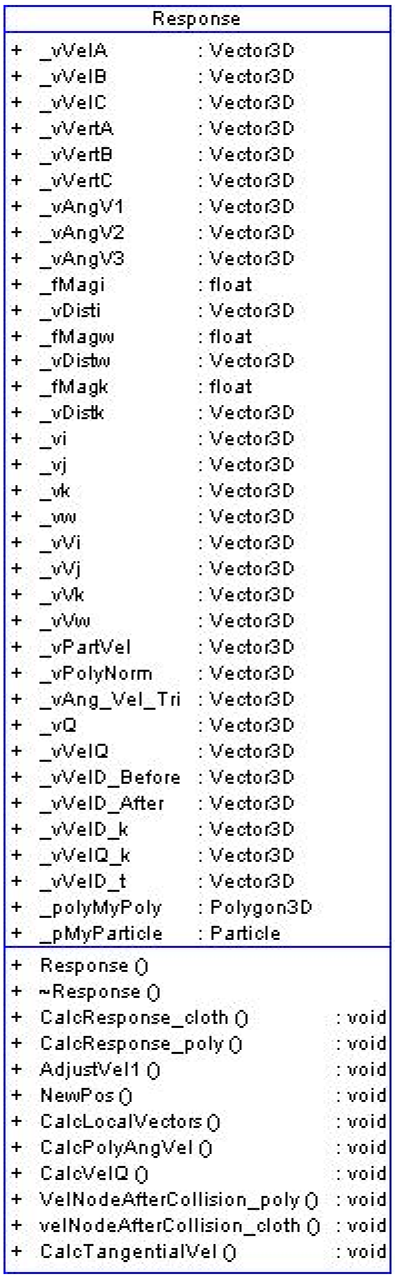
```
FBXInterface
+  axisMarkers           : Vector3D
+  markerMidpoint        : Vector3D
+  markerMidpointOld     : Vector3D
+  markerMidpointDif     : Vector3D
+  xRot1                 : double
+  xRot2                 : double
+  xRot3                 : double
+  yRot1                 : double
+  yRot2                 : double
+  yRot3                 : double
+  zRot1                 : double
+  zRot2                 : double
+  zRot3                 : double
+  _mDirectCos           : Matix3D
+  _mInitRot             : Matrix3D
+  xOffset               : int
+  zOffset               : int
+  _myAABBTree           : AABBTreeNode
+  lPolyList             : list< Polygon3D >
+  lPolyListOld          : list< Polygon3D >
+  vertexArray           : KFbxVector4
+  vecArray              : Vector3D
+  vecOldArray           : Vector3D
+  lPolygonCount         : kInt
+  gSceneStatus          : int
+  gPeriod               : kTime
+  gStart                : kTime
+  gStop                 : kTime
+  gCurrentTime          : kTime
-  XAxis                 : Vector3D
-  YAxis                 : Vector3D
-  ZAxis                 : Vector3D
-  origin                : Vector3D

+  FBXInterface ()
+  ~FBXInterface ()
+  LoadFBX ()                    : void
+  ExtractData ()                : void
+  GetPeriod ()                  : kTime
+  TrackHipMovement ()           : void
+  DrawPolys ()                  : void
+  CleanUp ()                    : void
-  ImportFBX ()                  : void
-  TakeSelectionCallback ()      : void
-  BuildPolyList ()              : void
-  BuildOldPolyList ()           : void
-  BuildVectorArray ()           : void
-  GetScene ()                   : void
-  GetNodeRecursive ()           : void
-  GetNode ()                    : void
-  GetMesh ()                    : void
+  StoreRigMarkers ()            : void
-  CalculateHipRotation ()       : void
-  GetMarker ()                  : void
-  SetCoordPos ()                : void
-  DrawCoord ()                  : void
```



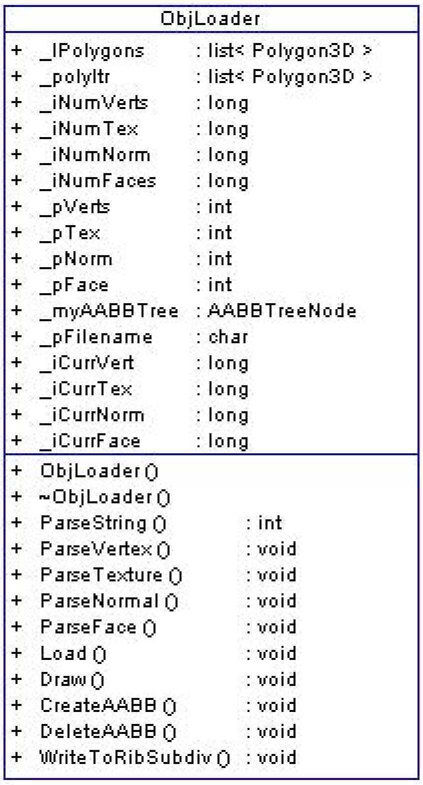
```
Application Manager
+  gbLMButton        : bool
+  gbMMButton        : bool
+  gbRMButton        : bool
+  gfLastX           : float
+  gfLastY           : float
+  fDt               : float
+  MyCloth           : ClothSkirt
+  MyFBX             : FBXInterface
+  camera            : Camera
+  iFrameCount       : int
+  bFBXMove          : bool
+  bRender           : bool
+  bPolyCollisions   : bool
+  bClothCollisions  : bool

+  Render ()           : void
+  RenderShadow ()     : void
+  InitSetup ()        : void
+  RenderScene ()      : void
+  KeyboardFunc ()     : void
+  ChangeSize ()       : void
+  Movement ()         : void
+  Mouse ()            : void
+  TimerCallback ()    : void
+  CreateCamera ()     : void
+  main ()             : int
```



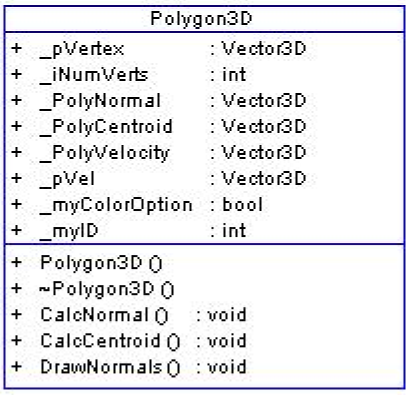
```
Camera
+  _vEye          : int
+  _vLook         : int
+  _vUp           : int
+  _vN            : int
+  _vU            : int
+  _vV            : int
+  _m             : flloat
+  _fViewAngle    : float
+  _fAspectRatio  : flloat
+  _fNearClip     : float
+  _fFarClip      : flloat

+  Camera ()
+  ~Camera ()
+  SetDefault ()              : void
+  SetModelViewMatrix ()      : void
+  Set ()                     : void
+  SetProjectionMatrix ()     : void
+  SetShape ()                : void
+  RotAxes ()                 : void
+  Roll ()                    : void
+  Pitch ()                   : void
+  Yaw ()                     : void
+  NormaliseYaw ()            : void
+  Slide ()                   : void
+  WriteToRib ()              : void
```



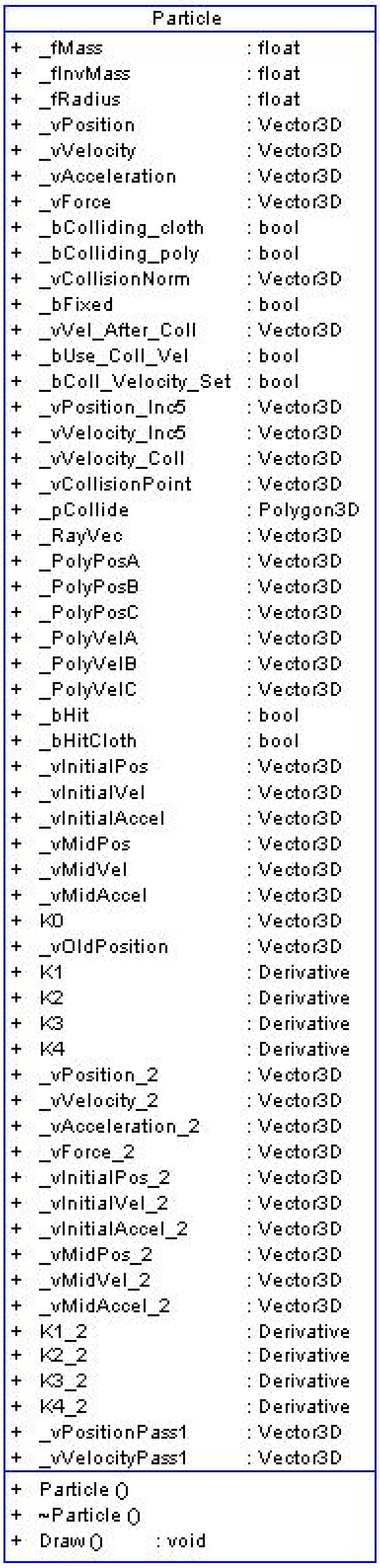
```
Response
+  _vVelA               : Vector3D
+  _vVelB               : Vector3D
+  _vVelC               : Vector3D
+  _vVertA              : Vector3D
+  _vVertB              : Vector3D
+  _vVertC              : Vector3D
+  _vAngV1              : Vector3D
+  _vAngV2              : Vector3D
+  _vAngV3              : Vector3D
+  _fMagi               : float
+  _vDisti              : Vector3D
+  _fMagw               : float
+  _vDistw              : Vector3D
+  _fMagk               : float
+  _vDistk              : Vector3D
+  _vi                  : Vector3D
+  _vj                  : Vector3D
+  _vk                  : Vector3D
+  _vw                  : Vector3D
+  _vVi                 : Vector3D
+  _vVj                 : Vector3D
+  _vVk                 : Vector3D
+  _vVw                 : Vector3D
+  _vPartVel            : Vector3D
+  _vPolyNorm           : Vector3D
+  _vAng_Vel_Tri        : Vector3D
+  _vQ                  : Vector3D
+  _vVelQ               : Vector3D
+  _vVelD_Before        : Vector3D
+  _vVelD_After         : Vector3D
+  _vVelD_k             : Vector3D
+  _vVelQ_k             : Vector3D
+  _vVelD_t             : Vector3D
+  _polyMyPoly          : Polygon3D
+  _pMyParticle         : Particle
+  Response ()
+  ~Response ()
+  CalcResponse_cloth ()              : void
+  CalcResponse_poly ()               : void
+  AdjustVel1 ()                      : void
+  NewPos ()                          : void
+  CalcLocalVectors ()                : void
+  CalcPolyAngVel ()                  : void
+  CalcVelQ ()                        : void
+  VelNodeAfterCollision_poly ()      : void
+  velNodeAfterCollision_cloth ()     : void
+  CalcTangentialVel ()               : void
```



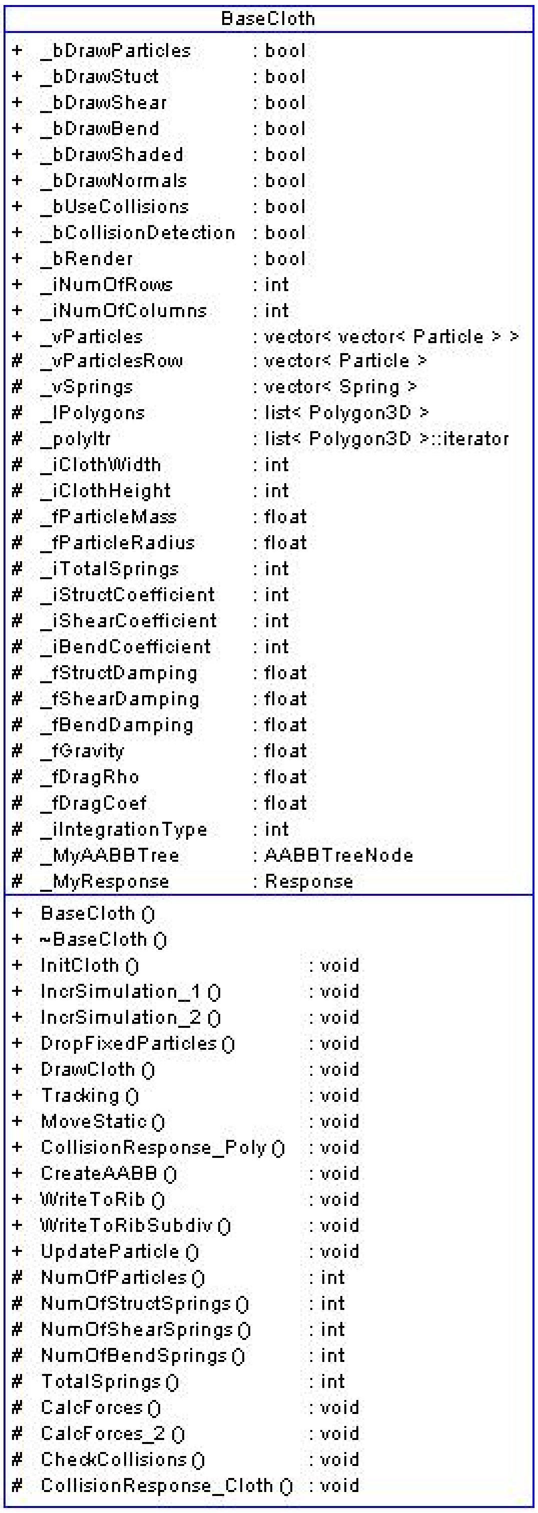
```
ObjLoader
+  _lPolygons        : list< Polygon3D >
+  _polyItr          : list< Polygon3D >
+  _iNumVerts        : long
+  _iNumTex          : long
+  _iNumNorm         : long
+  _iNumFaces        : long
+  _pVerts           : int
+  _pTex             : int
+  _pNorm            : int
+  _pFace            : int
+  _myAABBTree       : AABBTreeNode
+  _pFilename        : char
+  _iCurrVert        : long
+  _iCurrTex         : long
+  _iCurrNorm        : long
+  _iCurrFace        : long
+  ObjLoader ()
+  ~ObjLoader ()
+  ParseString ()            : int
+  ParseVertex ()            : void
+  ParseTexture ()           : void
+  ParseNormal ()            : void
+  ParseFace ()              : void
+  Load ()                   : void
+  Draw ()                   : void
+  CreateAABB ()             : void
+  DeleteAABB ()             : void
+  WriteToRibSubdiv ()       : void
```



```
Polygon3D
+  _pVertex           : Vector3D
+  _iNumVerts         : int
+  _PolyNormal        : Vector3D
+  _PolyCentroid      : Vector3D
+  _PolyVelocity      : Vector3D
+  _pVel              : Vector3D
+  _myColorOption     : bool
+  _myID              : int
+  Polygon3D ()
+  ~Polygon3D ()
+  CalcNormal ()      : void
+  CalcCentroid ()    : void
+  DrawNormals ()     : void
```



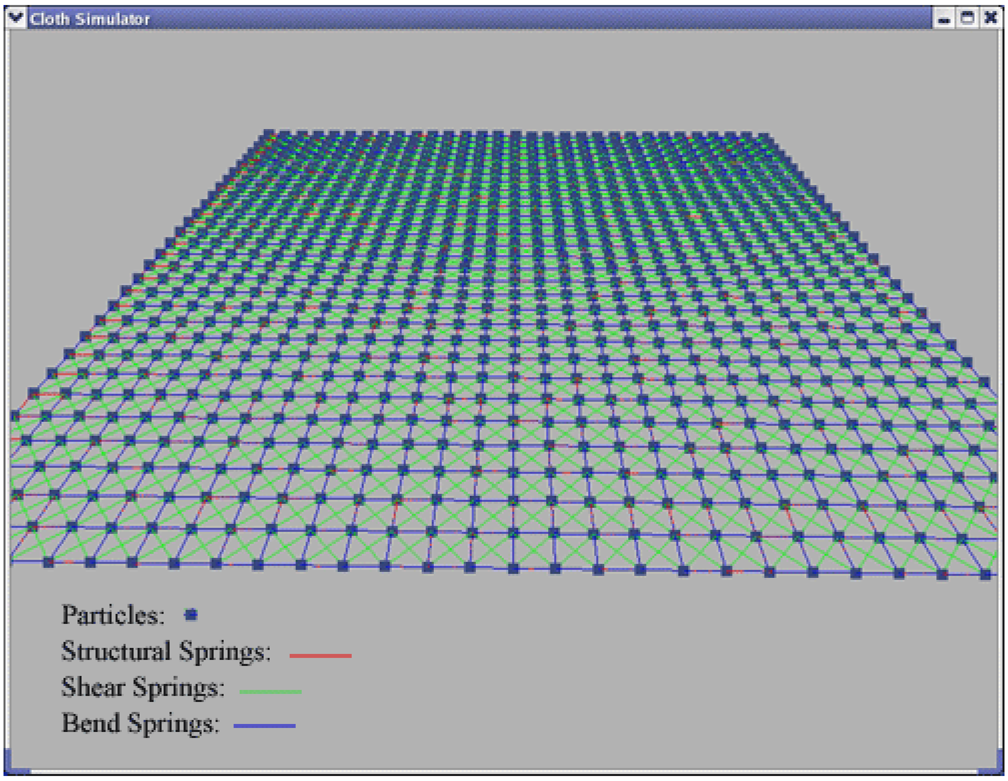
```
                    Particle
+  _fMass                 : float
+  _fInvMass              : float
+  _fRadius               : float
+  _vPosition             : Vector3D
+  _vVelocity             : Vector3D
+  _vAcceleration         : Vector3D
+  _vForce                : Vector3D
+  _bColliding_cloth      : bool
+  _bColliding_poly       : bool
+  _vCollisionNorm        : Vector3D
+  _bFixed                : bool
+  _vVel_After_Coll       : Vector3D
+  _bUse_Coll_Vel         : bool
+  _bColl_Velocity_Set    : bool
+  _vPosition_Inc5        : Vector3D
+  _vVelocity_Inc5        : Vector3D
+  _vVelocity_Coll        : Vector3D
+  _vCollisionPoint       : Vector3D
+  _pCollide              : Polygon3D
+  _RayVec                : Vector3D
+  _PolyPosA              : Vector3D
+  _PolyPosB              : Vector3D
+  _PolyPosC              : Vector3D
+  _PolyVelA              : Vector3D
+  _PolyVelB              : Vector3D
+  _PolyVelC              : Vector3D
+  _bHit                  : bool
+  _bHitCloth             : bool
+  _vInitialPos           : Vector3D
+  _vInitialVel           : Vector3D
+  _vInitialAccel         : Vector3D
+  _vMidPos               : Vector3D
+  _vMidVel               : Vector3D
+  _vMidAccel             : Vector3D
+  K0                     : Vector3D
+  _vOldPosition          : Vector3D
+  K1                     : Derivative
+  K2                     : Derivative
+  K3                     : Derivative
+  K4                     : Derivative
+  _vPosition_2           : Vector3D
+  _vVelocity_2           : Vector3D
+  _vAcceleration_2       : Vector3D
+  _vForce_2              : Vector3D
+  _vInitialPos_2         : Vector3D
+  _vInitialVel_2         : Vector3D
+  _vInitialAccel_2       : Vector3D
+  _vMidPos_2             : Vector3D
+  _vMidVel_2             : Vector3D
+  _vMidAccel_2           : Vector3D
+  K1_2                   : Derivative
+  K2_2                   : Derivative
+  K3_2                   : Derivative
+  K4_2                   : Derivative
+  _vPositionPass1        : Vector3D
+  _vVelocityPass1        : Vector3D
-------------------------------------------
+  Particle ()
+  ~Particle ()
+  Draw ()         : void
```



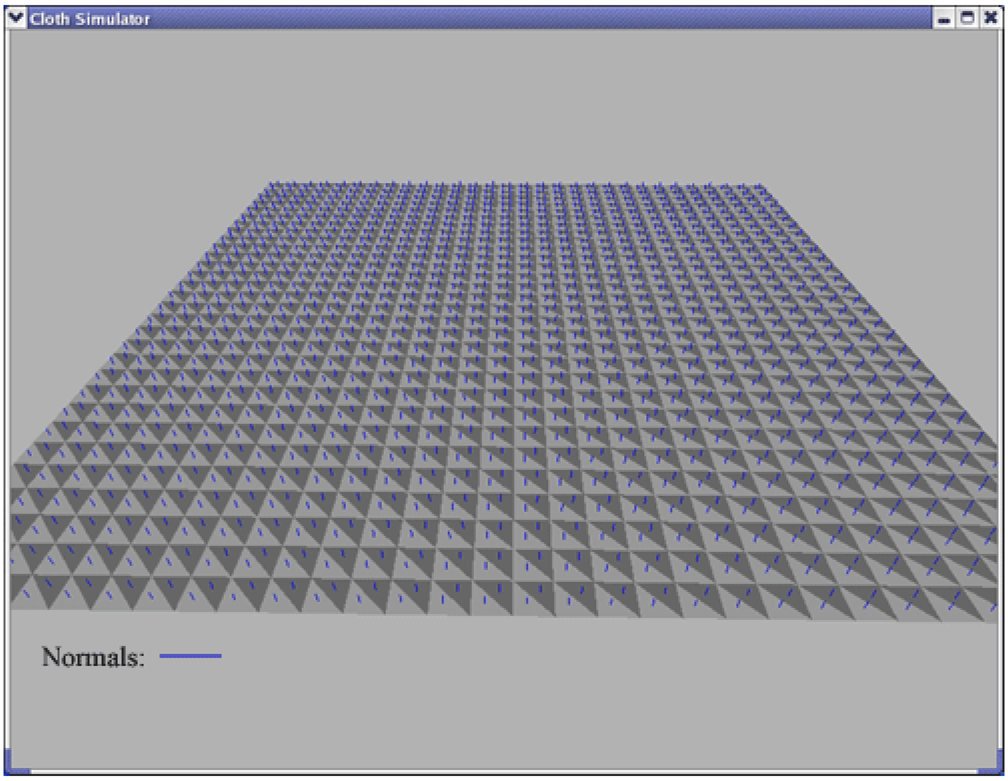
```
                    BaseCloth
+  _bDrawParticles        : bool
+  _bDrawStuct            : bool
+  _bDrawShear            : bool
+  _bDrawBend             : bool
+  _bDrawShaded           : bool
+  _bDrawNormals          : bool
+  _bUseCollisions        : bool
+  _bCollisionDetection   : bool
+  _bRender               : bool
+  _iNumOfRows            : int
+  _iNumOfColumns         : int
+  _vParticles            : vector< vector< Particle > >
#  _vParticlesRow         : vector< Particle >
#  _vSprings              : vector< Spring >
#  _lPolygons             : list< Polygon3D >
#  _polyItr               : list< Polygon3D >::iterator
#  _iClothWidth           : int
#  _iClothHeight          : int
#  _fParticleMass         : float
#  _fParticleRadius       : float
#  _iTotalSprings         : int
#  _iStructCoefficient    : int
#  _iShearCoefficient     : int
#  _iBendCoefficient      : int
#  _fStructDamping        : float
#  _fShearDamping         : float
#  _fBendDamping          : float
#  _fGravity              : float
#  _fDragRho              : float
#  _fDragCoef             : float
#  _iIntegrationType      : int
#  _MyAABBTree            : AABBTreeNode
#  _MyResponse            : Response
------------------------------------------------------
+  BaseCloth ()
+  ~BaseCloth ()
+  InitCloth ()                    : void
+  IncrSimulation_1 ()             : void
+  IncrSimulation_2 ()             : void
+  DropFixedParticles ()           : void
+  DrawCloth ()                    : void
+  Tracking ()                     : void
+  MoveStatic ()                   : void
+  CollisionResponse_Poly ()       : void
+  CreateAABB ()                   : void
+  WriteToRib ()                   : void
+  WriteToRibSubdiv ()             : void
+  UpdateParticle ()               : void
#  NumOfParticles ()               : int
#  NumOfStructSprings ()           : int
#  NumOfShearSprings ()            : int
#  NumOfBendSprings ()             : int
#  TotalSprings ()                 : int
#  CalcForces ()                   : void
#  CalcForces_2 ()                 : void
#  CheckCollisions ()              : void
#  CollisionResponse_Cloth ()      : void
```

Chapter 22 Appendix B

22.1 OpenGL View of the Cloth Sheet's Springs

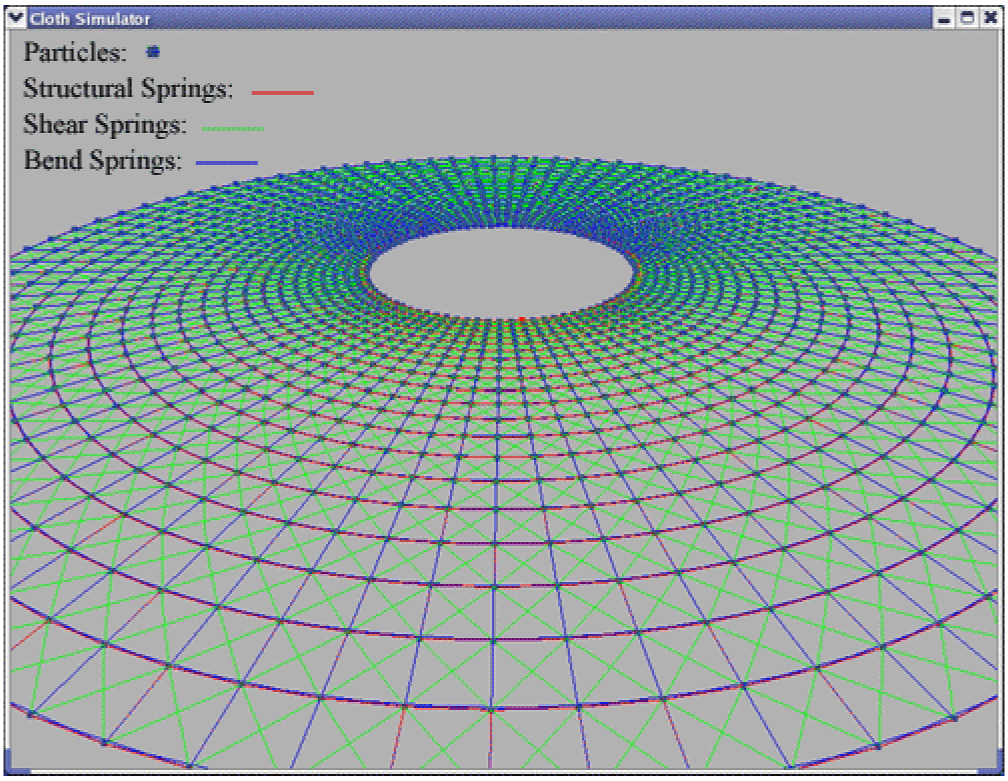


22.2 OpenGL View of the Cloth Sheet's Polygons

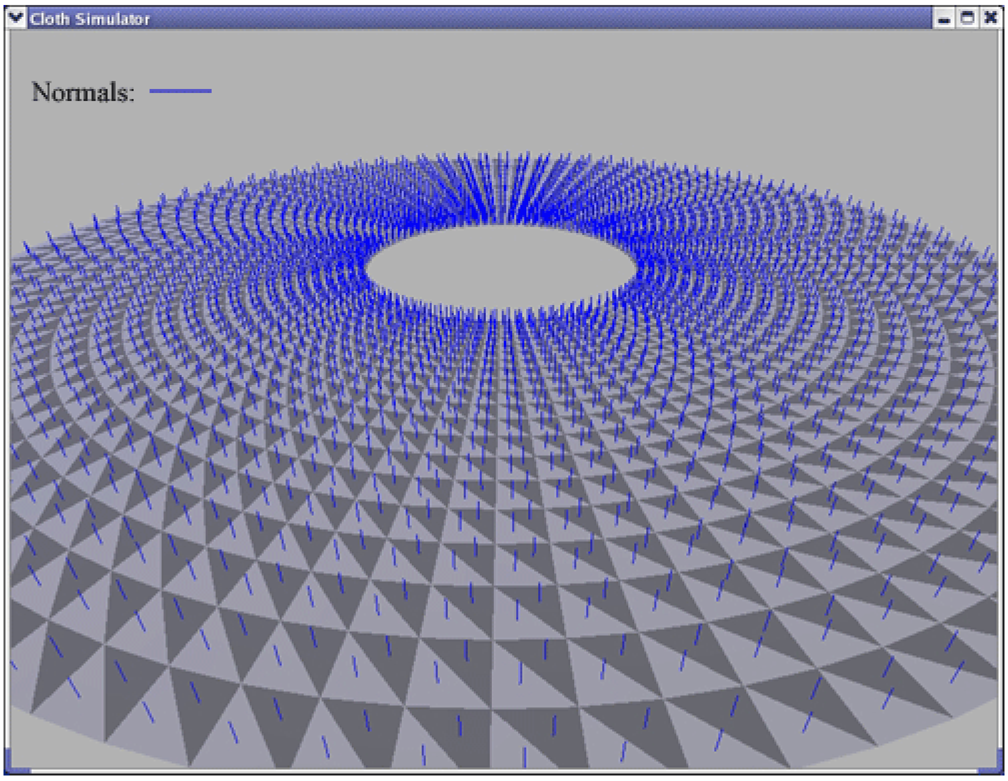


Chapter 23 Appendix C

23.1 OpenGL View of the Cloth Skirt's Springs



23.2 OpenGL View of the Cloth Skirt's Polygons



Chapter 24 Appendix D

24.1 Derivations for implicit integration

An element Kij is a 3x3 matrix in the K^{3nx3n} matrix which is derived for each of the three *pairs* of vertices in a triangular element and represents the change in the force at vertex i with respect to the position of vertex j.

So:

$K_{ij} = \partial f_i / \partial x_j = -k ((\partial C(x)/\partial x_i) * (\partial C(x)^T / \partial x_j) + (\partial^2 C(x) / \partial x_i \partial x_j) * C(x))$ (equation 1)

If:

$f_i = (f_{ix}, f_{iy}, f_{iz})$

$x_j = (x_j, y_j, z_j)$

Then:

$K_{ij} = \partial f_i/\partial x_j = [\partial f_{ix}/\partial x_j, \partial f_{ix}/\partial y_j, \partial f_{ix}/\partial z_j, \partial f_{iy}/\partial x_j, \partial f_{iy}/\partial y_j, \partial f_{iy}/\partial z_j, \partial f_{iz}/\partial x_j , \partial f_{iz}/\partial y_j, \partial f_{iz}/\partial z_j)$ (equation 2)

Given the energy function which is derived from conditional functions:

$E_C = k/2 * C(x)^T C(x)$

$= k/2 * \sum^n_{i=1} C^{(i)}(x)^T C^{(i)}(x)$

$= k/2 * \sum^n_{i=1} (C^{(i)}_n)^2 + (C^{(i)}_n)^2$ (equation 3)

Now for all the triangles M that contain the vertex i:

$f_i = - (\partial E_C/\partial x_i) = -k * \sum_M [(\partial C^{(M)}_u / \partial x_i)*C^{(M)}_u + (\partial C^{(M)}_v / \partial x_i)*C^{(M)}_v]$ (equation 4)

For any other triangle, the Kth triangle, that does not contain the ith vertex:

$\partial C^{(K)}/\partial x_i = 0$

Now if N is the index of all the triangles which contain both vertex i and j:

$K_{ij} = -k * \sum_N [(\partial C^{(N)}{}_u/\partial x_i) * (\partial C^{(N)T}{}_u/\partial x_j) + (\partial C^{(N)}{}_v/\partial x_i) * (\partial C^{(N)T}{}_v/\partial x_j) + (\partial^2 C^{(N)}{}_u/dx_i dx_j) * C^{(N)}{}_u + (d^2 C^{(N)}{}_v/\partial x_i \partial x_j) * C^{(N)}{}_v]$ (equation 5)

To obtain the derivatives defined in equation 5:

$$C^{(I)}(x) = a * \begin{pmatrix} C^{(I)}{}_u(x) \\ C^{(I)}{}_v(x) \end{pmatrix}$$

$$C^{(I)}(x) = \begin{pmatrix} a\,(Wu^T Wu)^{1/2} - ab_u \\ a\,(Wv^T Wv)^{1/2} - ab_v \end{pmatrix}$$

$\partial C^{(I)}{}_u(x)/\partial x_i = \partial[\ a(Wu^T Wu)^{1/2} - ab_u\ ] / \partial x_i$

$= a * \partial[\ a(Wu^T Wu)^{1/2}\ ] / \partial x_i$ (equation 6)

Using the chain rule:

$= (½) * a * (1/(Wu^T Wu)^{1/2}) * \partial(Wu^T Wu)/\partial x_i$

$W_u{}^T W_u$ can be solved from equation BLAH.

$W_u{}^T W_u = (\ 1 / \Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1\ )^2 * [\ \Delta v_2{}^2 \Delta x_1{}^T \Delta x_1 - 2\Delta v_1 \Delta v_2 \Delta x_1{}^T \Delta x_2 + \Delta v_1{}^2 \Delta x_2{}^T \Delta x_2]$

Now, recall that each triangle has three vertices defined as X_I, X_J and X_K. This means that for $\partial C^{(I)}u / \partial x_i$, $\partial C^{(I)}v / \partial x_j$, $\partial^2 C^{(I)}u / \partial x_i x_j$, $\partial^2 C^{(I)}v / \partial x_i x_j$there are three cases to be solved for.

Solving for $\partial C^{(I)}u / \partial x_i$:

Case 1: x_i refers to X_I in the triangle:

It is known:

$\Delta X_1 = X_J - X_I$

$\Delta X_2 = X_K - X_I$

Since the partial derivation is for X_I:

$\partial \Delta x_1 / \partial x_i = \partial x_1 / \partial X_I = [-1, 0, 0, 0, -1, 0, 0, 0, -1]$

$\partial \Delta x_2 / \partial x_i = \partial x_2 / \partial X_I = [-1, 0, 0, 0, -1, 0, 0, 0, -1]$

Therefore:

$\partial(Wu^T Wu) / \partial x_i = (1 / \Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1)^2 * [-2\Delta v_2^2 \Delta x_1 + 2\Delta v_1 \Delta v_2 \Delta x_2 + 2\Delta v_1 \Delta v_2 \Delta x_1 - 2\Delta v_1^2 \Delta x_2]$

Substituting equation 6 we have solved for $\partial C^{(I)}u / \partial x_i$ for X_I:

$\partial C^{(I)}u / \partial x_i = a(1 / (Wu^T Wu)^{1/2}) * (1/ (\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1)^2) * ((-\Delta v_2^2 + \Delta v_1 \Delta v_2)\Delta x_1 + (-\Delta v_1^2 + \Delta v_1 \Delta v_2)\Delta x_2))$

Case 2: x_i refers to X_J in the triangle:

Since the partial derivation is for X_J:

$\partial \Delta x_1 / \partial x_i = \partial x_1 / \partial X_J = [1, 0, 0, 0, 1, 0, 0, 0, 1]$

$\partial \Delta x_2 / \partial x_i = \partial x_2 / \partial X_J = [0, 0, 0, 0, 0, 0, 0, 0, 0]$

Substituting equation 6 we have solved for $\partial C^{(I)}u / \partial x_i$ for X_I:

$\partial C^{(I)}u / \partial x_i = a(1 / (Wu^T Wu)^{1/2}) * (1/ (\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1)^2) * (\Delta v_2^2 \Delta x_1 - \Delta v_1 \Delta v_2 \Delta x_2)$

Case 2: x_i refers to X_J in the triangle:

Since the partial derivation is for X_K:

$\partial \Delta x_1 / \partial x_i = \partial x_1 / \partial X_K = [\ 0, 0, 0, 0, 0, 0, 0, 0, 0\ ]$
$\partial \Delta x_2 / \partial x_i = \partial x_2 / \partial X_K = [\ 1, 0, 0, 0, 1, 0, 0, 0, 1\ ]$

Substituting equation 6 we have solved for $\partial C^{(I)}u / \partial x_i$ for X_K:

$\partial C^{(I)}u / \partial x_i = a(1 / (Wu^T Wu)^{1/2}) * (1/ (\ \Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1)^{\ 2}) * (\Delta v_1^{\ 2} \Delta x_2 - \Delta v_1 \Delta v_2 \Delta x_1)$

Solving for $\partial^2 C^{(I)}u / \partial x_i x_j$:

For Case 1 of $\partial C^{(I)}u / \partial x_i$:

$\partial C^{(I)}u / \partial x_i = a(1 / (Wu^T Wu)^{1/2}) * (1/ (\ \Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1)^{\ 2}) * ((-\Delta v_2^{\ 2} + \Delta v_1 \Delta v_2)\Delta x_1 + (-\Delta v_1^{\ 2} + \Delta v_1 \Delta v_2)\Delta x_2))$

For simplification, this can be rewritten as:

$d = [\ a / (\ \Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1)^{\ 2}\ ]$
$e(x) = [\ (-\Delta v_2^{\ 2} + \Delta v_1 \Delta v_2)\Delta x_1 + (-\ \Delta v_1^{\ 2} + \Delta v_1 \Delta v_2)\Delta x_2\ ]$

Therefore:
$\partial C^{(I)}u / \partial x_i = (d / (Wu^T Wu)^{1/2}) * e(x)$

Using substitution:

$\partial^2 C^{(I)}u / \partial x_i x_j = (\partial / \partial x_j) * (\partial C_u / \partial x_i)$
$= (\partial / \partial x_j) * d * e(x) * (Wu^T Wu)^{-1/2} + (d / (Wu^T Wu)^{1/2})^{*}\ (\partial e(x_1) / \partial x_j)$
$= (-1/2) * (Wu^T Wu)^{-3/2\ *}\ d * e(x) * (\partial (Wu^T Wu) / \partial x_j) + (d / (Wu^T Wu)^{1/2\ *}$ $(\partial e(x_1) / \partial x_j)$

To derive $(\partial e(x_1) / \partial x_j)$

$(\partial e(x) / \partial x_j) = \partial [\ (-\Delta v_2^{\ 2} + \Delta v_1 \Delta v_2)\Delta x_1 + (-\ \Delta v_1^{\ 2} + \Delta v_1 \Delta v_2)\Delta x_2\ ] / \partial x_j$

This in turn needs to be solved for three cases:

Case 1.1: where x_j is X_I

$\partial \Delta x_1 / \partial x_j = \partial \Delta x_2 / \partial x_j = [-1, 0, 0, 0, -1, 0, 0, 0, -1]$

We can now calculate:

$(\partial e(x) / \partial x_j) = (\Delta v_1 - \Delta v_2)^2$

Case 1.2: where x_j is X_J

$\partial \Delta x_1 / \partial x_j = [1, 0, 0, 0, 1, 0, 0, 0, 1]$
$\partial \Delta x_2 / \partial x_j = [0, 0, 0, 0, 0, 0, 0, 0, 0]$

We can now calculate:

$(\partial e(x) / \partial x_j) = (\Delta v_1 \Delta v_2 - \Delta v_2^2)$

Case 1.3: where x_j is X_K

$\partial \Delta x_1 / \partial x_j = [0, 0, 0, 0, 0, 0, 0, 0, 0]$
$\partial \Delta x_2 / \partial x_j = [1, 0, 0, 0, 1, 0, 0, 0, 1]$

$(\partial e(x) / \partial x_j) = (\Delta v_1 \Delta v_2 - \Delta v_1^2)$

Finally, Case 2 and Case 3 of $\partial C^{(I)}u / \partial x_i$ can be solved for $\partial^2 C^{(I)}u / \partial x_i x_j$ in the same way.

Chapter 25 Appendix E

25.1 Reduced Version of a Simulation RIB File

```
#=======================================================
# Script: ribFiles/sheet_807.rib
# Author: Rib generated from Cloth Simulator
# Description: Parameters and bicubic patches for
# cloth simulation object
#=======================================================


#=======================================================
# Initialise display parameters
#=======================================================
Display "./renders/sheet_807.tiff" "file" "rgba"

#=======================================================
# Initialise camera parameters
#=======================================================
Clipping 1 24000
Projection "perspective" "fov" [60]
Scale 1 1 -1
ConcatTransform [ 0.995054 0.0616784 0.0778465 0 0 0.783798 -0.621009
0 -0.0993196 0.617937 0.779921 0 -7.10949 0.412747 -187.418 1 ]
Format 720 576 1

#=======================================================
# The geometry and lighting
#=======================================================
WorldBegin

LightSource "distantlight" 1 "intensity" [ 0.8 ] "lightcolor" [ 1.0
1.0 1.0 ] "from" [ 20 40 60 ] "to" [ 0 0 0 ]

LightSource "shadowspot" 2 "shadowname" [ "__SHADOWMAP__" ]
"intensity" [ 40000 ] "coneangle" [90] "conedeltaangle" [1]
"lightcolor" [ 1.0 1.0 1.0 ] "from" [ 170 250 0 ] "to" [ 0 0 0 ]

TransformBegin

AttributeBegin

Color [0.7 0.7 0.7]

Surface "__SURFACE1__" __PARAMETERS1__

Opacity [__OPACITY1__]

Matte __MATTE1__

#=======================================================
# Obj Object
#=======================================================
SubdivisionMesh "catmull-clark" [ 3 3 3 ………………………
```

```
AttributeEnd

AttributeBegin

Color [0.21 0.43 0.57]

Surface "__SURFACE2__" __PARAMETERS2__

Opacity [__OPACITY2__]

Matte __MATTE2__

#=======================================================
# Cloth Object
#=======================================================
SubdivisionMesh "catmull-clark" [ 3 3 3 ……………………


AttributeEnd

AttributeBegin

Color [0.21 0.43 0.57]

Surface "__SURFACE3__" __PARAMETERS3__

Opacity [__OPACITY3__]

Matte __MATTE3__

#=======================================================
# Environment Object
#=======================================================
PointsPolygons [ 3 3 3 ……………………


AttributeEnd

TransformEnd

WorldEnd
```

Chapter 26   Appendix F

26.1   Bash Script to Fix the Subversion Log File Issues

```
#!/bin/bash

##########################################################
# Script: fixSvnLog.sh
# Author: Gavin Harrison <gavin@gavinharrison.co.uk>
# Abstract: Finds all files with $USER and changes their
# file permissions.
##########################################################

# Counter
counter=0

# List all files with the relevant username
for i in $(find /masters/gharrison/subversionRepos/db/ -user $USER -
type f); do

      # Change the file permissions
      chmod 777 $i

      # Increment the counter
      counter=`expr $counter + 1`

done

echo "Number of files amended: " $counter
```

26.2   Bash Script to Render the Sheet Simulations

```
#!/bin/bash

##########################################################
# Script: rmSheet.sh
# Author: Gavin Harrison <gavin@gavinharrison.co.uk>
# Abstract: Opens Renderman RIB files with a filename
# starting with "sheet_" and sets the surface
# parameters, opacity and matte, then renders
#
# Note: Had issues supplying variables to sed, hence the
# hard-coded surface and parameter values
##########################################################

# Counter
counter=0

# List all the filenames starting with "sheet_"
for i in $(ls sheet_*); do

# Get number
number='sheetShadow_'${i:6:3}'.shad'
```

```
  # Open the file, substitute the surface and parameter values,
opacity and matte, then render
  cat $i | sed 's/__SHADOWMAP__/'$number'/g' \
      | sed 's/__SURFACE1__/plastic/g' | sed 's/__PARAMETERS1__//g' \
      | sed 's/__OPACITY1__/1.0 1.0 1.0/g' \
      | sed 's/__MATTE1__/0/g' \
      | sed 's/__SURFACE2__/tartan2/g' | sed 's/__PARAMETERS2__//g' \
      | sed 's/__OPACITY2__/1.0 1.0 1.0/g' \
      | sed 's/__MATTE2__/0/g' \
        | sed 's/__SURFACE3__/plastic/g' | sed 's/__PARAMETERS3__//g'
\
      | sed 's/__OPACITY3__/1.0 1.0 1.0/g' \
      | sed 's/__MATTE3__/1/g' \
      | render

  # Print the count
  echo "Rendered frame: " $counter

  # Increment the counter
  counter=`expr $counter + 1`

# Finished
done
```

26.3 Bash Script to Render the Skirt Simulations

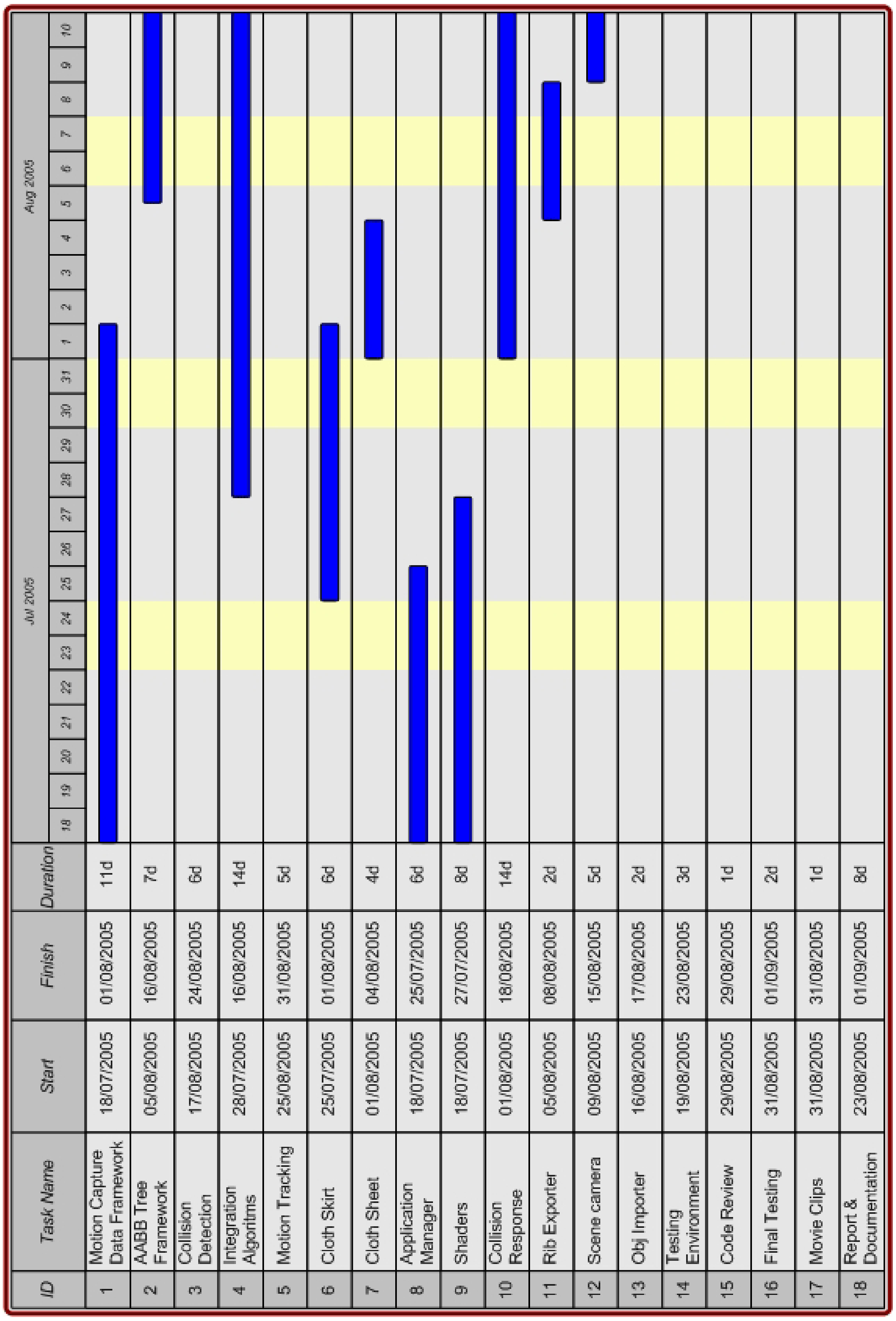
```
#!/bin/bash

##########################################################
# Script: rmSkirt.sh
# Author: Gavin Harrison <gavin@gavinharrison.co.uk>
# Abstract: Opens Renderman RIB files with a filename
# starting with "skirt_" and sets the surface
# parameters, opacity and matte, then renders
#
# Note: Had issues supplying variables to sed, hence the
# hard-coded surface and parameter values
##########################################################

# Counter
counter=0

# List all the filenames starting with "skirt_"
for i in $(ls skirt_*); do

# Get number
number='skirtShadow_'${i:6:3}'.shad'

  # Open the file, substitute the surface and parameter values,
opacity and matte, then render
  cat $i | sed 's/__SHADOWMAP__/'$number'/g' \
      | sed 's/__SURFACE1__/plastic/g' | sed 's/__PARAMETERS1__//g' \
      | sed 's/__OPACITY1__/0.0 0.0 0.0/g' \
      | sed 's/__MATTE1__/1/g' \
      | sed 's/__SURFACE2__/velvet/g' | sed 's/__PARAMETERS2__//g' \
      | sed 's/__OPACITY2__/1.0 1.0 1.0/g' \
      | sed 's/__MATTE2__/0/g' \
        | sed 's/__SURFACE3__/plastic/g' | sed 's/__PARAMETERS3__//g'
\
      | sed 's/__OPACITY3__/0.0 0.0 0.0/g' \
      | sed 's/__MATTE3__/1/g' \
      | render

  # Print the count
  echo "Rendered frame: " $counter

  # Increment the counter
  counter=`expr $counter + 1`

# Finished
done
```

26.4 Bash Script to Render the Skirt Simulation Shadows

```
#!/bin/bash

##########################################################
# Script: rmSkirtShadow.sh
# Author: Gavin Harrison <gavin@gavinharrison.co.uk>
# Abstract: Opens Renderman RIB files with a filename
# starting with "skirtShadow_" and renders
#
# Note: Had issues supplying variables to sed, hence the
# hard-coded surface and parameter values
##########################################################

# Counter
counter=0

# List all the filenames starting with "skirtShadow_"
for i in $(ls skirtShadow_*); do

  # Open the file and render
  cat $i | render

  # Print the count
  echo "Rendered shadow frame: " $counter

  # Increment the counter
  counter=`expr $counter + 1`

# Finished
done
```

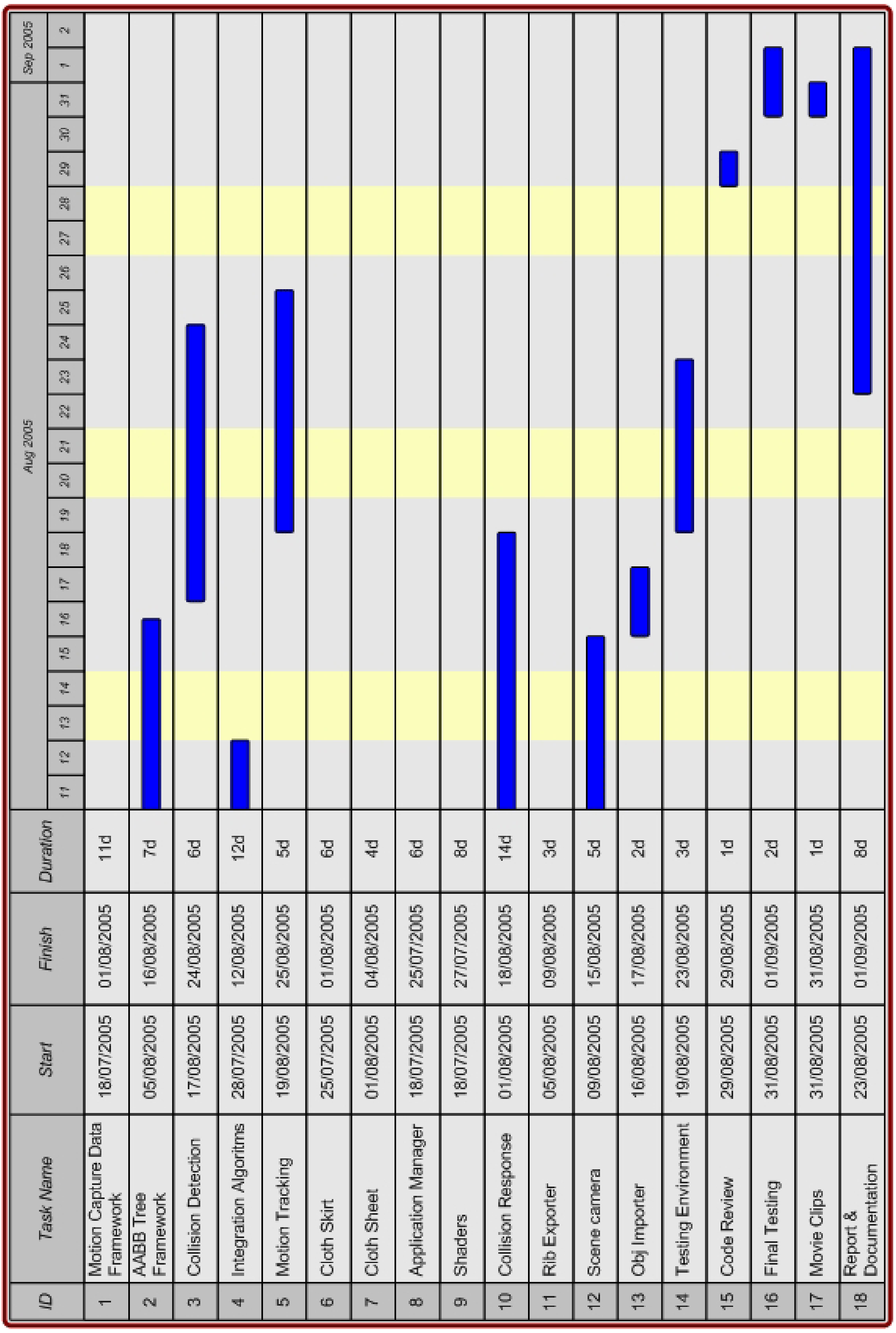
Chapter 27 Appendix G

27.1 Project Plan



ID	Task Name	Start	Finish	Duration
1	Motion Capture Data Framework	18/07/2005	01/08/2005	11d
2	AABB Tree Framework	05/08/2005	16/08/2005	7d
3	Collision Detection	17/08/2005	24/08/2005	6d
4	Integration Algoritms	28/07/2005	16/08/2005	14d
5	Motion Tracking	25/08/2005	31/08/2005	5d
6	Cloth Skirt	25/07/2005	01/08/2005	6d
7	Cloth Sheet	01/08/2005	04/08/2005	4d
8	Application Manager	18/07/2005	25/07/2005	6d
9	Shaders	18/07/2005	27/07/2005	8d
10	Collision Response	01/08/2005	18/08/2005	14d
11	Rib Exporter	05/08/2005	08/08/2005	2d
12	Scene camera	09/08/2005	15/08/2005	5d
13	Obj Importer	16/08/2005	17/08/2005	2d
14	Testing Environment	19/08/2005	23/08/2005	3d
15	Code Review	29/08/2005	29/08/2005	1d
16	Final Testing	31/08/2005	01/09/2005	2d
17	Movie Clips	31/08/2005	31/08/2005	1d
18	Report & Documentation	23/08/2005	01/09/2005	8d

27.2 Project Plan (Continued)



ID	Task Name	Start	Finish	Duration
1	Motion Capture Data Framework	18/07/2005	01/08/2005	11d
2	AABB Tree Framework	05/08/2005	16/08/2005	7d
3	Collision Detection	17/08/2005	24/08/2005	6d
4	Integration Algoritms	28/07/2005	12/08/2005	12d
5	Motion Tracking	19/08/2005	25/08/2005	5d
6	Cloth Skirt	25/07/2005	01/08/2005	6d
7	Cloth Sheet	01/08/2005	04/08/2005	4d
8	Application Manager	18/07/2005	25/07/2005	6d
9	Shaders	18/07/2005	27/07/2005	8d
10	Collision Response	01/08/2005	18/08/2005	14d
11	Rib Exporter	05/08/2005	09/08/2005	3d
12	Scene camera	09/08/2005	15/08/2005	5d
13	Obj Importer	16/08/2005	17/08/2005	2d
14	Testing Environment	19/08/2005	23/08/2005	3d
15	Code Review	29/08/2005	29/08/2005	1d
16	Final Testing	31/08/2005	01/09/2005	2d
17	Movie Clips	31/08/2005	31/08/2005	1d
18	Report & Documentation	23/08/2005	01/09/2005	8d