Soft Body Animation

in Real-Time Simulations

by

Mark A. Sullivan III

S.B. C.S., M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May, 2011
[June 2011]

©2011 Massachusetts Institute of Technology
All rights reserved.

ARCHIVES

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 2 1 2011

LIBRARIES

Author ____
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by ____
Philip Tan
US Executive Director, Singapore-MIT GAMBIT Game Lab
Thesis Supervisor

Accepted by ____
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Soft Body Animation in Real-Time Simulations
by
Mark A. Sullivan III

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2011

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a novel approach for creating deformable object animations. A deformable object can be represented as a discrete lattice of particles, and transforming those particles defines a new state for the represented object. By applying shape matching techniques, we are able to adapt traditional mesh based animations to this representation. We then allow these particles to take place in a soft body physics simulation. By making the particles track positions defined in the animation, soft body tracking of user created animation has been made possible.

Thesis Supervisor: Philip Tan
Title: US Executive Director, Singapore-MIT GAMBIT Game Lab

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

This research was done at the Singapore-MIT GAMBIT Game Lab, and as such, I owe a lot of thanks to the lab and its members. In particular, I would like to thank Philip Tan for his supervision and the chance to do this thesis work at GAMBIT. I also owe special thanks to Andrew Grant, who helped me find this research project, allowed me to explore this technology through the UROP program, and provided frequent feedback for both the technical and written components of this project.

I would like to thank Alec Rivers, who originally developed RealMatter, for allowing me to build off of his project and for providing me with his source code.

I would also like to thank those students who worked on this project through the UROP program (pardon the redundant acronym). Adin Schmahmann helped with some of the early stages of game engine integration. Patrick Rodriguez and Skyler Seto used my design and layout tools and provided valuable feedback. Lauren Cason, Hing Chui, David Kenyon, and Hannah Lawler produced many models used for testing and demos.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction....11
2 Background....13
    2.1 Soft Body Physics....14
    2.2 Nonphysical Animation....15
    2.3 Rigid Body Animation....15
    2.4 Soft Body Animation....16
3 RealMatter....19
    3.1 Soft Body Representation....19
    3.2 Soft Body Simulation....21
    3.3 FastLSM....23
    3.4 Soft Body Rendering....24
    3.5 Fracture Simulation....25
    3.6 Fracture Rendering....26
4 Animation Tracking....29
5 Animation Creation....31
    5.1 Maya....31
    5.2 Forward Solving....31
    5.3 Inverse Solving....34
    5.4 Plugin Design Considerations....39
6 Game Engine Integration....43
    6.1 Unity....43
    6.2 Integration....44
    6.3 User Interface....46
7 Performance....51
8 Conclusion....55
A Appendix....57
    A.1 User Guide....57

List of Figures

Figure 2-1: Rigid body tracking applied to skeletal animation....16
Figure 2-2: Soft body baby with embedded skeleton.... 17
Figure 3-1: Soft body discretization.... 20
Figure 3-2: Mesh reconstruction....25
Figure 3-3: A summary of Muller's triangle splitting algorithm....27
Figure 4-1: Keyframed lattice states.... 30
Figure 5-1: Forward solving.... 32
Figure 5-2: Inverse solving.... 34
Figure 5-3: Transformation not realizable by trilinear interpolation....36
Figure 6-1: Screenshot of Unity editor.... 44
Figure 6-2: Fracture rendering....46
Figure 6-3: Soft body menu bar....47
Figure 6-4: Soft body properties editor....48
Figure 6-5: Lattice view....49
Figure 7-1: Simulation time per frame for different animation schemes....53

List of Tables

Table 5-1: Comparison of forward and inverse animation solving.............................. 39
Table 7-1: Simulation time per frame for different animation schemes 52

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Video games are continually increasing their capacity for realism. Improvements in the field of computer graphics allow for detailed rendering of increasingly complex scenes. However, static renders do not communicate the entire visual experience. Another important class of visual effects includes animation and physics, which specify the visual dynamics of a game. Many modern games utilize physics engines, packages which execute a real-time simulation of rigid body mechanics. This permits objects to react to one another in such a way as to mimic reality.

Current physics techniques in games, however, impose restrictions. If one wanted, for instance, an object which could bend, stretch, or break, then rigid body mechanics are an inadequate representation. Rigid bodies can only translate and rotate. Inclusion of these additional transformation capabilities mandates the use of soft body physics. Soft body physics, however, isn't nearly as pervasive as its rigid counterpart. Real-time techniques have been developed, as discussed in Chapter 2, so perhaps there are other improvements which could be made to foster their inclusion.

To make soft body physics more approachable, a set of tools were developed in this project which allow non-programmers to set up and run these simulations. In game development, allowing anyone on a multidisciplinary team to set up and modify soft bodies in a scene has potential for increasing a team's efficiency over the case where only programmers have such capability. Therefore, ease of use was a large priority for making soft body use more appealing.

The tools described here also address the lack of support existing systems have for controlled animation. These systems work well for simulating soft static objects, but these objects ultimately settle down to their initial structure after being perturbed. If one wanted to simulate a self-actuating soft body, for instance an invertebrate which moves about on its own, or a skeleton guided by the contractions of attached soft muscles, then one would quickly discover this limitation.

The goal of the thesis project was to fit an existing soft body engine with the ability to track poses and animations. Another objective was to create an animation representation which makes use of soft body properties, as current animation representations have a basis in rigid body ideas. This also entailed the development of artist-usable tools to permit creation of animations within this framework.

Chapter 2

Background

2.1 Soft Body Physics

"Soft body physics" is a term which refers to the behavior deformable objects exhibit. Deformable objects are those which change their shape over time, often through transformations such as stretching, squashing, bending, twisting, or fracturing. This is to be contrasted with rigid body physics, where objects always maintain their shape, and only have the transformation affordances of translation and rotation.

Many different approaches to simulation of deformable bodies have been developed [1]. A survey by Gibson and Mirtitch [2] classifies three major types of soft body physics models used for simulation within the computer graphics community.

Continuum models form the first classification. These types of models attempt to model the object as continuous, although solving for these models is ultimately discrete. This type of model is also sometimes called a finite element model. In these methods, the soft body is adaptively discretized, so that the representation of a body is finer at points where large deformations are occurring. Internal forces are computed based on real world models of deformable objects. It is the most physically accurate of all of the proposed model categories. However, its accuracy comes at the cost of speed. While the techniques used in continuum models are popular in applications such as computer aided drafting software, where the stress or thermal flow in a material can be assessed, it is currently too costly for real-time applications.

The next class is composed of mass-spring models. These models approximate soft bodies as a 3D lattice of masses and springs. One might be able to envision how a model like this can approximate a soft body which can be stretched or bent; there's some intuition associated with this model. However, it's not as accurate as a continuum model, and it might not be stable. Standard first order integrators will often cause the system to explode if the lattice is too complicated or the springs are too stiff. This mandates the use of implicit or higher order integrators, which slow down the speed at which this can be simulated.

Finally, there are geometric methods. Whereas the other methods were entirely rooted in the laws of physics, geometric methods are an approximation achieved by manipulating the mesh in part based on geometric arguments instead of purely physical ones. Consequently, this is the least realistic out of these classes. However, the primary application for this work is games. Perfect physical accuracy is not necessary if the result looks visually plausible. So, while this model wouldn't be used for a quantitative simulation, in practice it looks convincing enough for a game. Additionally, these methods have the benefit of being fast enough for real-time simulation. These qualities make this branch of methods ideal for games, and thus the animation framework created for this project was built from a soft body system in this category.

2.2 Nonphysical Animation

Animations are often defined in a purely graphical context, independent from any physics simulation. This type of animation is called "kinematic." Creation of such animation often must be performed by a skilled expert in order to produce convincing motion [3]. Animations are typically defined by creating keyframes, representative snapshots of the animation. If necessary, a sensible interpolation takes place between those keyframes. Since such animations only play back a scripted sequence of frames, purely kinematic animations will not interact or respond to their environment. Despite these limitations, these types of animations are very easily controlled; there is no element of unpredictability during playback, and there do exist qualified individuals to generate these animations.

One of the most prevalent approaches to defining such animations on a deforming character is skeletal animation. In typical skeletal animation, a 3D mesh is fit with a nonphysical skeleton. The vertices on the mesh are given weightings which tell them how to move based on

the transformations of the underlying skeleton. This algorithm which decides the new vertex positions is called skeletal subspace deformation [4]. To define an animation, an artist only has to move and position the skeleton. The rest of the mesh follows along. This greatly reduces the amount of work an artist must do to define an animation, as opposed to manually specifying vertex positions for every frame, while still allowing for expressiveness.

2.3 Rigid Body Animation

When defining animations for a physical object, it is not desirable for the physical entity to simply play the animation back. If it did, then the physics has afforded no new behavior, and this would still be in the realm of kinematic animation. Instead, the animated body should interact with the physics engine to track the animation [5].

Consider the rigid body case. Let's say we have some physical object that we want to track a position and orientation over time. We can model that object as a rigid body, some geometry with a mass and tensor of inertia. Then we can use feedback techniques such as PID control to compute forces and torques over time which we should apply to that body [6]. This will allow the body to track its desired position and rotation while still interacting with the physical world. It has collision geometry so its motion will be appropriately interrupted if a solid object gets in its path. With an appropriately tuned feedback model, the body will additionally be able to respond naturally to gravity, wind, explosive forces, or any other external influences.

If there is some model to be physically animated, it can be represented as a physical skeleton, which can interact with anything in the world. It's very amenable to the nonphysical skeletal representation of animation. So, one can use that same representation to define target transformations for all of the bones. An example of how this can be done can be seen in Figure 2-1. Tracking techniques such as those mentioned above can be used to let the bones track poses, and, using skeletal subspace deformation, the mesh will deform to reflect the changes based on the state of the underlying skeleton. Tracking the animation allows for that animation to be realistically interrupted or affected by interactions with other objects in the game world.

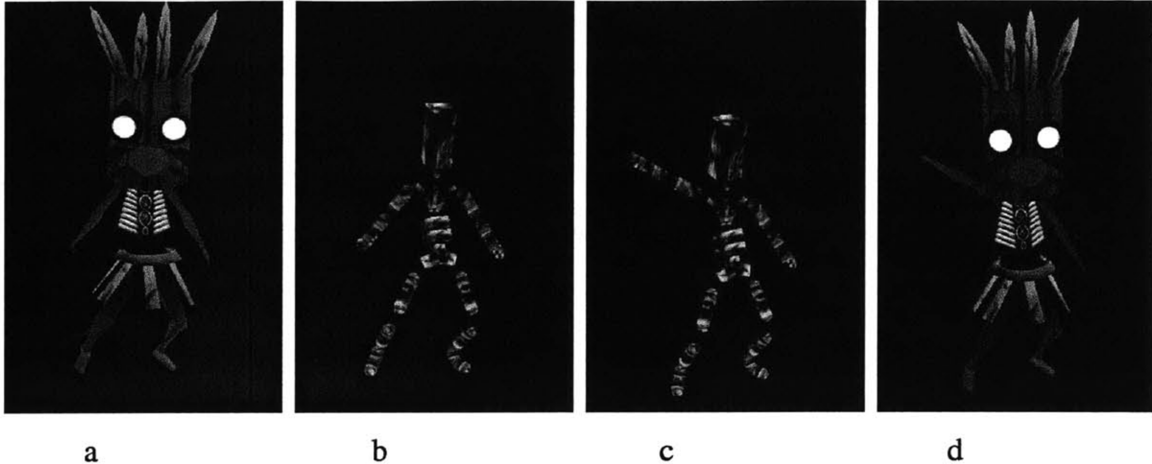


Figure 2-1: Rigid body tracking applied to skeletal animation. In a, we have a humanoid 3D model. It can be converted into a collection of rigid bodies representing the skeleton as shown in b. If that skeleton is told to track an animation, closed loop control can apply forces and torques necessary to transform it into a new pose such as c. And, given c, we can apply the skeletal subspace deformation algorithm to reconstruct the appearance of the mesh in d. The skeleton is exposed to the simulation, but would typically be invisible to the user. This character is from Moki Combat, created at the Singapore-MIT GAMBIT Game Lab.

2.4 Soft Body Animation

Soft body animation techniques have not been explored as much as their rigid and nonphysical counterparts. Nevertheless, there do exist some techniques to define animations for soft bodies.

In Autodesk Maya 2011, a popular 3D modeling and animation tool, the creation and simulation of soft bodies is possible [7]. The soft body is represented as a grid of particles, with a one-to-one mapping between particles and vertices in a mesh. The artist can paint on weights, which dictate how strongly those particles pull themselves to their goal position. The artist can adjust these goal positions by creating a goal object out of either the original or duplicate geometry. The bijection between particles and vertices allows this technique to work, since a deformed version of the mesh will have the same particle-vertex mapping.

NVIDIA PhysX is a physics engine which also supports soft bodies [8]. This tool represents the soft body volumetrically as an approximate tetrahedral lattice, up to a user defined precision. Affecting the motion of the soft body is typically done by the use of fields, time-varying spatial arrangements of forces. The drawback to this approach is that this control is indirect if one wanted to have a self actuating soft body.

Alec Rivers, creator of the geometrically based soft body engine RealMatter, produced a demonstration of an animated soft body baby, seen in Figure 2-2. The baby's skeleton is rigid, but the muscles and skin covering the baby form a soft body lattice. The demonstrated animation is fundamentally rigid. The skeleton tracks an animation, and the skin just follows along. While this does yield interesting effects like the ability to have the skin deform upon contact with objects in the scene, many properties of soft body physics are left unexplored. Artists would have no direct control over what happens to the skin through their animations.

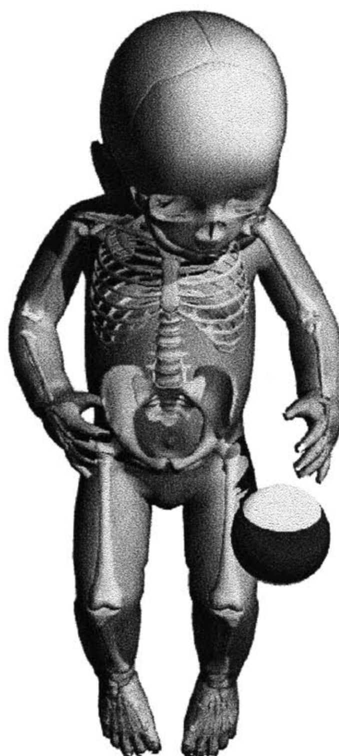


Figure 2-2: Soft body baby with embedded skeleton. Image created by Alec Rivers [9]. The skin of the baby is soft, but the underlying animation is fundamentally rigid, as it is specified by the state of the skeleton.

Soft body transformations are important even in traditional animation. As Lesseter discusses [10], deforming an object with squash and stretch is an important part of emphasizing the animation. Soft body properties can be used to emphasize movement, communicate material properties, and provide continuity to animations. These techniques have existed for much of the history of traditional hand drawn animation, which suggests the value of enabling this type of animation to be defined for simulations of virtual worlds.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

RealMatter

In order to understand this project's contributions, it is necessary to understand the underlying soft body representation and simulation techniques. This project extended RealMatter, a soft body physics library developed by Alec Rivers. The primary simulation technique used, Fast Lattice Shape Matching, is discussed in detail in his paper [9].

3.1 Soft Body Representation

It is necessary to have some way to convert from an artist's specification of a 3D mesh to a representation suitable for physical simulation. This is done by discretizing the mesh into finite rigid elements. Existing physics engines are able to handle rigid bodies, so such a representation will be useful when it comes time to simulate the body.

The mesh is discretized into finite elements by overlaying a rectangular grid on top of the mesh. The dimensions of this grid are parameters which would be set by the user. Each 3D cuboid which forms an element of the grid is known as a cell. Some subset of those cells is chosen to be our representation. In particular, those cells which intersect with the surface polygons, as well as optionally those which are on the inside of the mesh for closed meshes where the inside is well defined, are selected. This creates a blocky representation of the original mesh.

The cells will be useful as rendering elements; other objects are used for collision detection and simulation. These objects are called particles. Once there is a set of cells that represent the mesh, particles are placed at each vertex of these cells. Each cell is a cuboid, so it

will border eight particles. Particles will be shared by adjacent cells; duplicate particles won't be created in a single location. The collection of all these particles is known as the lattice. This process can be seen in Figure 3-1.

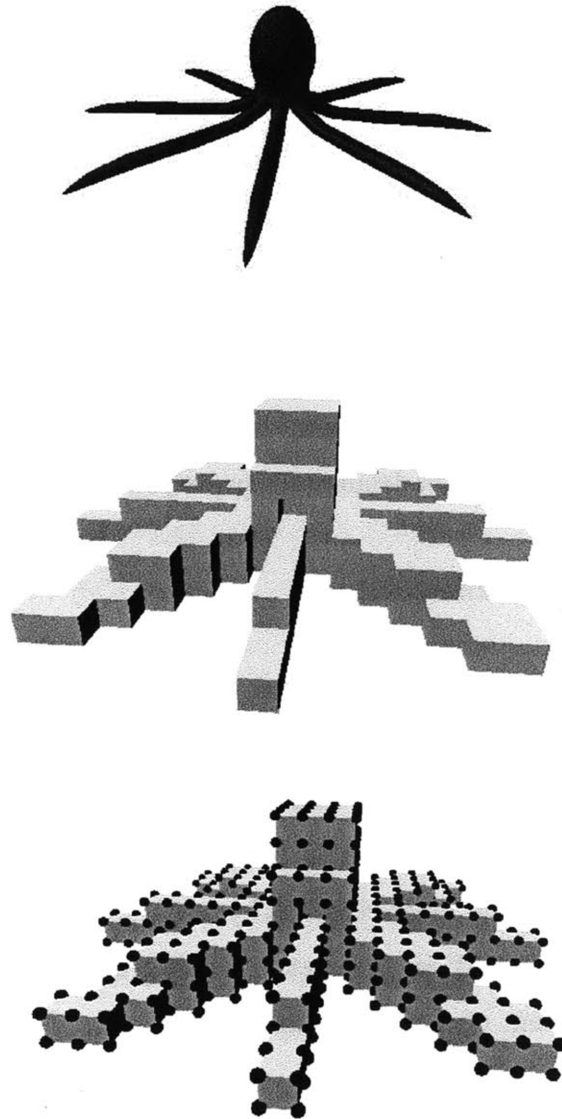


Figure 3-1. Soft body discretization. Given a mesh such as this octopus, top image, we are able to select some set of cells to serve as a discrete approximation, middle image. Finally, we take the corners of those cells and place particles there, bottom image.

3.2 Soft Body Simulation

Simulation only relies on the physical representation of the mesh, the particle lattice. Reconstruction back to a mesh will be done at a later step. Here, the soft body must both compute internal restorative forces and react to externally applied ones. Out of the technique classifications put forth by Gibson and Mirtitch [2], the techniques used here would be classified as geometric [9].

Every particle i has some associated neighborhood N_i. These are all of the particles reachable via an adjacent cell edge or diagonal, so each neighborhood may have up to 27 particles including itself. There will be fewer particles per neighborhood along the boundaries. The simulation is given a parameter w. This parameter is the "region half-width." It will be used to specify the size the particle "regions." Every particle has an associated region R_i, which is a set of particles. If w is 1, then R_i equals N_i. If w is 2, then R_i includes all those particles in the w=1 case, as well as all of their neighbors, and so forth for higher w.

Intuitively, a region is a portion of the mesh which will track a rigid shape. If w is small, and consequently regions are small, then perturbations on one side of the lattice won't immediately be detected on the other side, since a small region here would not span the entire lattice. If we had a large w which did span the entire lattice, then information about perturbations would travel through the lattice in a single simulation step, and the body would more closely model a rigid body. In this way, w controls how soft or rigid a soft body is.

The goal of shape matching is to approximate every region of particles as though they have been transformed by some rigid transformation applied to their original positions, a translation and a rotation. We can then move those particles towards those positions corresponding to that rigid transform. This will give an object a tendency towards a nondeformed state. Assume that every particle i has a starting position $\overrightarrow{x_i^0}$ and a current position $\overrightarrow{x_i}$. All summations are over all particles belonging to the current region. We can then approximate a rigid body transform for the set of particles as follows.

$$\overrightarrow{x_{cm}^0} = \frac{\sum_i m_i \overrightarrow{x_i^0}}{\sum_i m_i}$$

$$\overrightarrow{x_{cm}} = \frac{\sum_i m_i \overrightarrow{x_i}}{\sum_i m_i}$$

$$\overrightarrow{q_i} = \overrightarrow{x_i^0} - \overrightarrow{x_{cm}^0}$$

$$\overrightarrow{p_i} = \overrightarrow{x_i} - \overrightarrow{x_{cm}}$$

$$\boldsymbol{A}_{pq} = \sum_i m_i \overrightarrow{p_i} \, \overrightarrow{q_i}^T$$

Equation 3-1

A_{pq} contains the rotational information required, and it can be extracted via a polar decomposition.

$$\boldsymbol{S} = \sqrt{\boldsymbol{A}_{pq}^T \boldsymbol{A}_{pq}}$$

$$\boldsymbol{R} = \boldsymbol{A}_{pq} \boldsymbol{S}^{-1}$$

Equation 3-2

We now have, for every region, an approximate translation and rotation from its initial state. Each particle in that region is assigned a goal position g_i, the position where a rigid transformation applied to that region would place it.

$$\overrightarrow{g_i} = \boldsymbol{R}\left(\overrightarrow{x_i^0} - \overrightarrow{x_{cm}^0}\right) + \overrightarrow{x_{cm}}$$

Equation 3-3

Since a particle may belong to many regions, its final goal position is a weighted average of the assigned goal positions from all of the regions to which it belongs. Nonuniform weighting may be used to resolve boundary condition problems and is discussed by Rivers [9].

Given a particle's current position and goal position, the output of this algorithm is an updated velocity of the particle, given by, for time step h:

$$\overrightarrow{v_i}(t+h) = \overrightarrow{v_i}(t) + \alpha \frac{\overrightarrow{g_i}(t) - \overrightarrow{x_i}(t)}{h}$$

Equation 3-4

Since particles are rigid bodies, they can interact with any objects within a standard rigid body physics engine. The particles can be simulated in the rigid world along with any other rigid objects, and then corrective soft body velocities can be applied using the techniques above.

Structuring a simulation cycle like this allows for easy integration of soft body properties into complete and well developed rigid body physics engines.

3.3 FastLSM

An important optimization involves exploiting the redundancy of shape matching among multiple regions in a dynamic programming styled approach [9]. This approach is called fast lattice shape matching, or FastLSM.

We often need to sum up values for all of the particles in some region. Consider the calculation of the center of mass of each region, necessary for calculating its translational component. Since the width in each dimension is $2w+1$ in a non-boundary case, the cost is $O(w^3)$ per region. So, the naïve approach isn't very practical. However, we can exploit repetition by breaking down the summation as follows. Assume every particle xyz has some value v_{xyz}, and we want to sum over all of those values in every region.

$$X_{xyz} = \sum_{i=x-w}^{x+w} v_{iyz}$$
$$XY_{xyz} = \sum_{j=y-w}^{y+w} X_{xjz}$$
$$SUM_{xyz} = \sum_{k=z-w}^{z+w} XY_{xyk}$$

Equation 3-5

We want to compute X, XY, and SUM for each particle in the lattice. Intuitively, we are computing the sums one dimension at a time so that we can reuse that sum for the next higher dimension. There is more redundancy we can exploit. When traversing a dimension to compute one of these sums, we can reuse the sum just computed as follows.

$$X_{xyz} = X_{(x-1)yz} - v_{(x-w-1)yz} + v_{(x+w)yz}$$
$$XY_{xyz} = XY_{x(y-1)z} - X_{x(y-w-1)z} + X_{x(y+w)z}$$
$$SUM_{xyz} = SUM_{xy(z-1)} - XY_{xy(z-w-1)} + XY_{xy(z+w)}$$

Equation 3-6

So, in a non-boundary case, we have only 6 floating point operations, and, factoring in boundary cases, we have an O(1) amortized cost per particle. With this tool, the rigid transform approximated by each region can be quickly calculated. A similar computation is done for the rotation, discussed by Rivers [9], and the result is a simulation speed asymptotically independent of w.

3.4 Soft Body Rendering

The techniques outlined so far can are able to generate a physical representation of a mesh, but it is still necessary to return to the mesh representation for rendering.

The process of reconstructing the mesh involves trilinear interpolation. At initialization, every mesh vertex lies inside of a cell. Based on this starting position, we can define a set of eight weights for each vertex, one corresponding to each of the particles enclosing its occupied cell.

We can index the eight particles around a given cell by a Boolean triplet ijk, where 000 is the particle of minimum lattice index along x, y, and z, and any of i, j, or k is 1 if the x, y, or z index is increased, respectively. If a vertex has a position $\vec{c}$ within that cell normalized to a unit cube, then its weights associated with each of its surrounding particles are given by

$$w_{ijk} = \big(i\,(\vec{c}.\mathrm{x}) + (1-\mathrm{i})(1-\vec{c}.\mathrm{x})\big) * \big(j\,(\vec{c}.\mathrm{y}) + (1-\mathrm{j})(1-\vec{c}.\mathrm{y})\big) * \big(k\,(\vec{c}.\mathrm{z}) + (1-\mathrm{k})(1-\vec{c}.\mathrm{z})\big)$$

Equation 3-7

We only need to compute those weights once. From then on, at every rendering step, the vertex is to be positioned at the weighted sum of the positions of those surrounding particles.

$$v_i = \sum_{ijk} w_{ijk} x_{ijk}$$

Equation 3-8

This process can be seen in Figure 3-2.

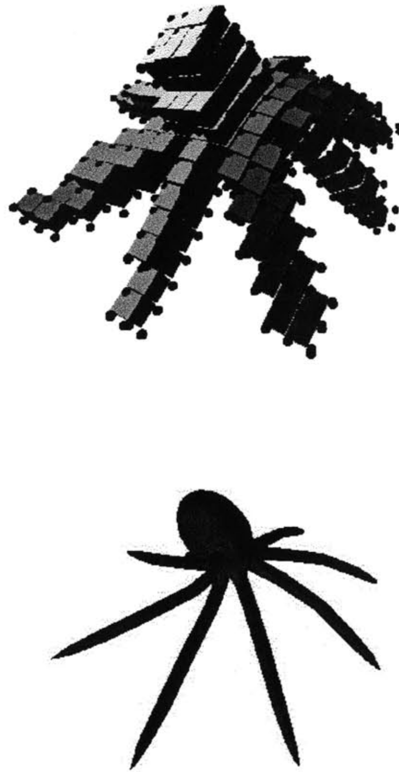


Figure 3-2. Mesh reconstruction. After simulating, an updated set of particle positions and velocities is produced. The particles and (reconstructed from this data) cells are shown in the top figure. Using the trilinear interpolation techniques described, the state of the mesh can be determined.

3.5 Fracture Simulation

In addition to those types of deformations which don't modify the underlying structure of the soft body, there is also support for fracturing. Fracturing is the process by which a physical body can rip apart, either partially or totally, disconnecting portions of the lattice.

A fracture can be automatically triggered in one of two ways. In either case, every cell in the lattice examines its immediate neighbors, those cells which differ in index by one along a single coordinate. First, for each such pair of cells, the difference between their center of mass separation in the current pose and the rest pose is computed. If this distance exceeds a certain threshold, a fracture between these cells is triggered. Second, each cell will have had some approximate rigid rotation assigned to it as using the shape matching procedure outlined earlier

based on the positions of its surrounding particles. If the magnitude of the rotational difference between two neighboring cells exceeds a certain threshold, then a fracture is triggered.

Once a fracture is triggered, several changes need to be made to the lattice. Since those two cells that are being split should no longer be attached, they should not share particles. Therefore, the four particles along the shared face of the two particles are duplicated. One copy of each particle is assigned to each of those cells. Other neighboring cells that also share any of these particles are assigned the copy assigned to that cell on their side of the intersection plane. Given these changes in the particle lattice, regions, which were dependent on particle neighborhoods, must now be recomputed.

While each of these fractures is a fairly small, local one, many fractures may occur in the object, either in a single or across several time steps. When these fractures accumulate, the object may exhibit larger scale fractures. If the particle lattice may become disconnected entirely, the object will break apart into multiple pieces.

3.6 Fracture Rendering

Once a soft body has been fractured in simulation, the mesh used for rendering must be changed as well. The triangles along the fracture boundary surface must be split, and an interior surface must be generated for proper visualization of the fracture. The technique used is described by Müller [11].

Using this technique, the surface triangles are never modified. Instead of splitting the surface triangles along the face of intersection, each triangle along the boundary are assigned to one of the sides of the fracture boundary. Vertices which were once shared between formerly adjacent triangles are duplicated and assigned to either side, so that those triangles can become completely independent. A closing interior surface is then generated on each side of the mesh. The algorithm is summarized concisely by Alec Rivers in the RealMatter source code in Figure 3-3.

Triangle Splitting Summary
1. Generate directed graph of nodes. 1a. Go through the set of polygons and test them against face f. 1b. If they intersect, generate nodes n1 and n2 either when an edge of t intersects f or when an edge of f intersects t. 1c. Generate a directed edge between n1 and n2, according to the rule n1->n2 iff [nt x (n2 - n1)].nf > 0; nf is normal of face with respect to c1. 1d. Connect coincident nodes from the node without an outgoing edge to the one with one. 1e. Generate four nodes at the corners of f. 1f. Walk along the edges of f in the positive direction, connecting sequential nodes iff the first node has no outgoing edge yet 1g. Detect cycles, pruning branch edges and unused points 2. Extend the nodes outward to connect to the surface mesh. (That is, close the mesh.) 2a. Assign to each node a vertex of the surface mesh. 2b. Move each pair of nodes n1, n2 generated by triangle t towards each other by one third of the segment length (to make sure the triangulation generates everything). 2c. Delete all consecutive nodes on the formed cycles that refer to the same surface vertex. 2d. Use the original locations of the nodes on f to compute a 2D trianglulation using ear-cutting. 2e. Span the generated triangles across the assigned surface vertices. 2f. For each generated triangle t, create a duplicate triangle t' with reversed orientation. Assign t to c1 and t' to c2.

Figure 3-3. A summary of Müller's triangle splitting algorithm.

At a high level, this algorithm determines all triangle intersections along the face at the boundary between the two cells being split. A node is generated everywhere a triangle edge intersects the face, as well as everywhere a face edge intersects a triangle. These nodes are connected in a graph if they were generated from the same triangle, occupy the same position, or are adjacent along the border of the face. Cycles in the graph are detected to determine surfaces which need to be closed. This cycle forms a polygon, which can then be tessellated using ear cutting [12]. These projected triangles can then be mapped back onto a 3D surface using the nodes which generated them, each of which will generally be mapped to one of the vertices of the triangle that generated it.

Chapter 4

Animation Tracking

The techniques described so far are able to simulate a soft body which will always tend towards its initial rest configuration. This allows for simulation of soft inanimate objects, but it requires modifications to allow for tracking of a dynamic configuration. Fortunately, these changes are not extensive.

The simulation techniques described compute the target positions of particles. In the process, an approximate rigid body transformation is calculated for each region. However, while the particles are arranged in a rectangular grid in the rest pose, there is no mathematical necessity for this. What has thus far been treated as the "initial" pose, allowing us to compute the invariant center of mass and rotation properties, can now be treated as a dynamic "target" pose. This will dirty the invariants, and they will require recalculation every time the target pose changes, but this modification alone is sufficient to allow soft bodies to track dynamic poses.

Thus, to define an animation for a soft body, the data we need is a keyframed series of poses. Each pose is a specification of the lattice state, the positions of all of the particles that frame. The poses are to exist along some time line, so that each pose also has some associated number signifying its temporal location within the animation. An animation should also have some sort of framerate, the speed at which the animation timeline is traversed.

Given this information, at any point in time, it can be determined where the animation exists on the animation timeline. This time will either be on a keyframe or between two of them, assuming the animation either loops or terminates upon reaching the end. The target particle positions can then be set to an interpolation over the timeline of the particle positions specified by the keyframes between which the current frame lies, as seen in Figure 4-1.

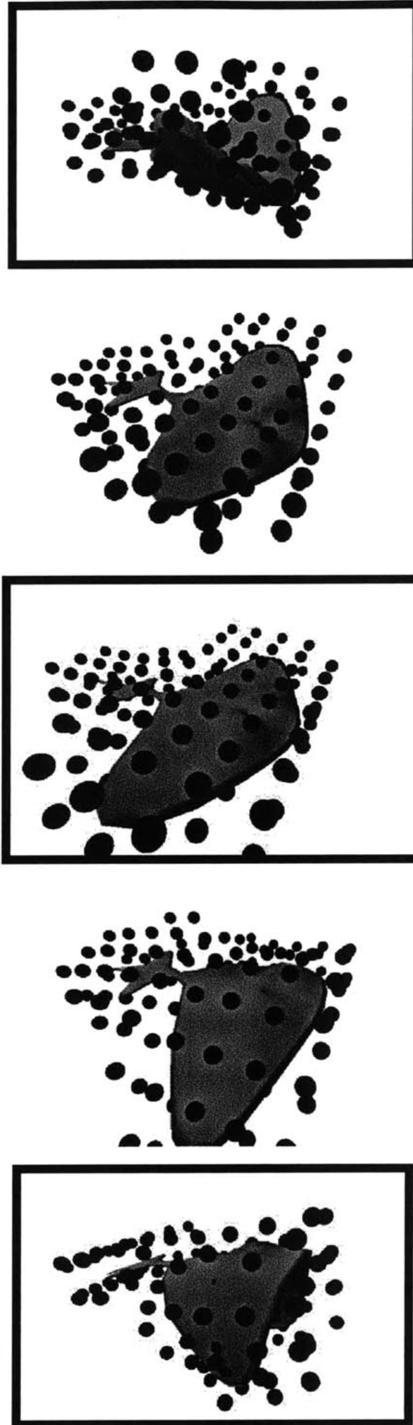


Figure 4-1. Keyframed lattice states. By defining the state of the particle lattice at keyframes (boxes), one can interpolate between those to solve for the target animation at any intermediate point on the animation timeline.

Chapter 5

Animation Creation

5.1 Maya

Autodesk Maya is a popular 3D modeling and animation program. This software provides a wide variety of tools used to create content for media such as movies or video games. In addition to simply using the packaged functionality, Maya permits user scripting. Developers can create plugins to provide new modeling and animation functionality. These plugins can then easily exist within the Maya interface as if they were packaged scripts.

Because of its scripting support, it was practical to build the tools for animation creation into Maya. The alternative would be creating an animation tool as a standalone piece of software. Using Maya means that the core functionality necessary for animation already exists; Maya has many advanced animation features which can work in tandem with custom plugins.

5.2 Forward Solving

The first set of Maya scripts is based on the idea termed here as forward solving. This nomenclature here parallels traditional skeletal animation. In forward kinematics [13], the user specifies the animation representation, the joint angles, directly. Similarly, here the user specifies the positions of the lattice particles at several keyframes along a time line. This is exactly the type of data required during simulation. These tools also permit the user to preview

the effect of their lattice deformations on the target mesh in real-time. An illustration of this technique can be seen in Figure 5-1.

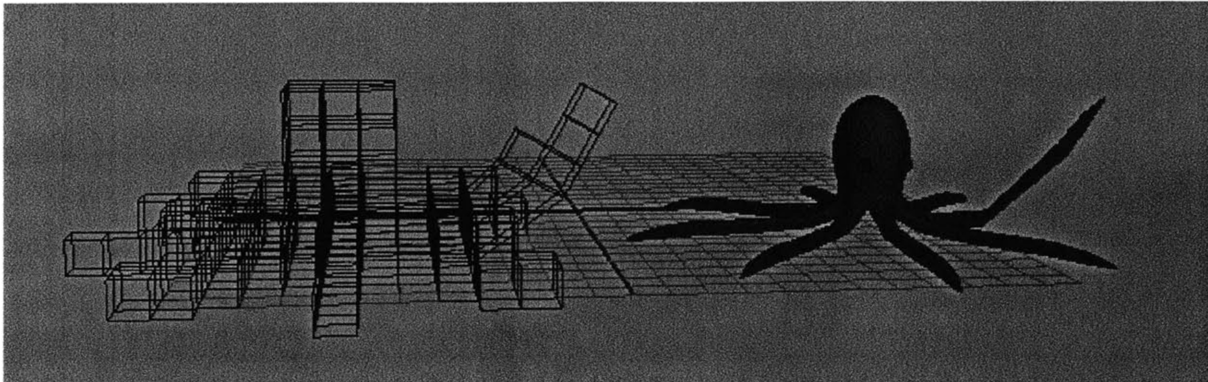


Figure 5-1. Forward solving. The user specifies the state of the particle lattice (left), and the effect if that deformation is interactively previewed (right).

The first step for the user is to generate a representation of the particle lattice in Maya. From the toolbar created for this project, the user must click the "Generate" icon, and then they are asked to select a .rmb file. This file contains all of the data necessary to create the soft body lattice: the offset between the origin of the lattice and the origin of the mesh, the spacing between the particles, the scaling that is applied to the target mesh, and a list of lattice indices which are occupied by particles. This file type is generated by scripts written for the target game engine, Unity. Once the user selects the file, a Maya representation of the particle lattice is generated. In this representation, the lattice is a simple mesh: a set of vertices, edges, and polygons. There is a vertex associated with each particle of the mesh, placed at a position determined by its lattice index. Edges are placed between adjacent vertices, that is, particles whose lattice index in a single dimension differs by one. These edges are purely for visualization purposes, so that the shape of the soft body is apparent.

Once the generation step is complete, the user has a set of Maya vertices, serving as a representation of the particle lattice, which they are able to manipulate. Users then have the optional step of importing a preview mesh. This step allows users to interactively preview the appearance of a mesh given a particular particle configuration. The vertices of the preview mesh are positioned based on the same interpolation techniques used to determine the vertex positions of a mesh from particle positions in simulation. Each mesh vertex is initially contained within a rectangular cuboid of vertices representing particles. As the set of particle vertices deforms, the positions of the mesh vertices are determined by a trilinear interpolation of its surrounding

particle-vertices. Creation of a preview mesh is not a required step; it has no bearing on the animation that will be produced. It is designed purely as a usability feature. To enable the preview, the user clicks the "Import Mesh" button and selects the Maya file that contains the mesh corresponding to the particle lattice. The preview mesh updates in real-time as the particle lattice is deformed.

Once the user has finished initialization, he or she can then define the animation. The user does so by manipulating the vertices of the lattice mesh and storing its state at several keyframes. The ability to define an animation this way is already present in Maya, and is one of the reasons why the soft body lattice was chosen to be represented as a mesh within Maya. The full Maya toolkit is available to the user to create animations. For instance, one common way to define animations is through the process of skeletal animation. To do this in Maya, an artist creates a skeleton and assigns it to a mesh. As the skeleton deforms, the mesh follows according to the skeletal subspace deformation algorithm [4]. Since the particle lattice is represented as a mesh, Maya can naturally use this object to participate in skeletal animation, a technique most animators are familiar with. Similarly, the entire Maya toolbox can operate on the representation of the soft body. Regardless of the techniques used to create deformations, the user will do so at several keyframes, positions on the animation timeline. The particles are linearly interpolated between these keyframes.

After creating the animation, the user is able to export it to a format usable by the soft body simulation. The user clicks the "Export" button, and is presented with a dialog box. It asks for the destination of the animation file and the desired framerate. When the user clicks export, the script iterates through the keyframes and writes out the frame numbers and the state of the particle lattice. These are written out to a .txt file.

This technique has several strengths and weaknesses, which are summarized in comparison to an alternative technique later in Table 5-1. Giving the user direct control of the particles means that they are able to modify the purest form of the animation definition. Additionally, since the soft body is represented as a mesh, Maya's built in tools work well with it. However, this technique does have its drawbacks. Artists are accustomed to being able to modify a mesh directly. Here, they can only modify the mesh via the particles, which act as control points. While mesh modification via control points is not unheard of [14], this may be unintuitive or against their training. Additionally, it's possible that the soft body properties will

change as the simulation is iterated upon. For instance, a lattice might become more detailed if a higher resolution is needed, or simplified for performance reasons. Since the animation is defined purely by the state of the lattice, changing the lattice means that the animation is lost. Therefore, this approach is not resilient to iteration within the simulation.

5.3 Inverse Solving

The next set of scripts seeks to address some of the failings of forward solving. In forward solving, the user specifies the state of the particle lattice, and the preview solves for what the resultant mesh will look like. In what will be termed here as inverse solving, the user specifies the state of the mesh, and then scripts solve for what the state of the particle lattice should be to best approximate that mesh state. An illustration of this technique is shown in Figure 5-2.

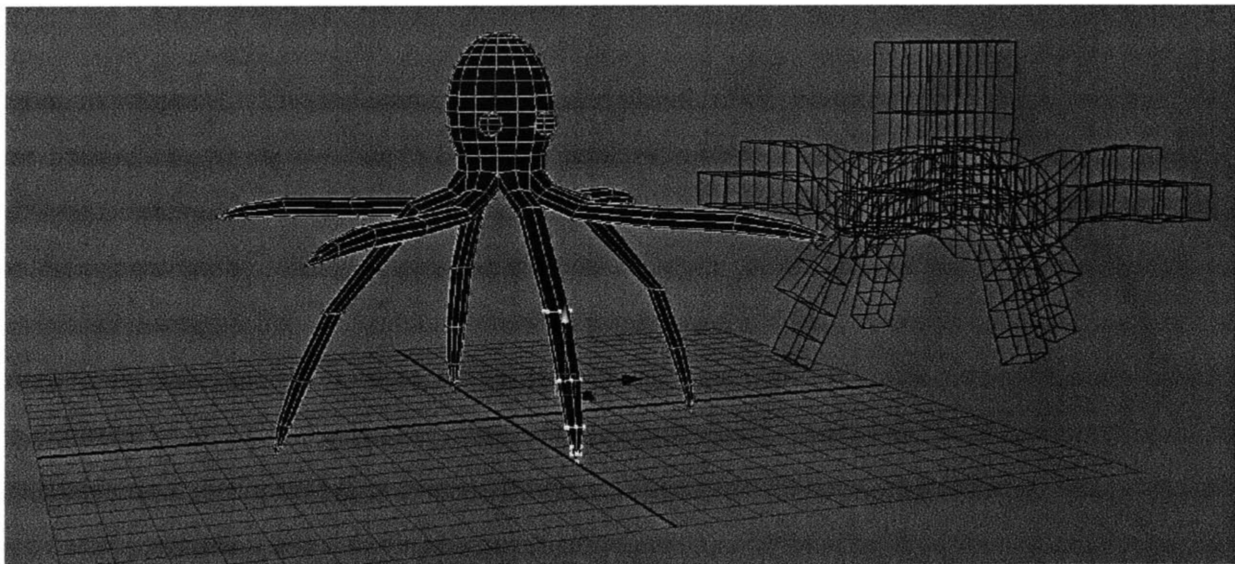


Figure 5-2. Inverse solving. The user specifies the state of the mesh (left), and the best fit particle lattice is previewed upon user request (right).

Inverse solving has an analogous relationship to inverse kinematics as forward solving had to forward kinematics. In each inverted case we define an animation by setting a target. In this case, the state of the mesh is the target, and we find the corresponding particle lattice state. In inverse kinematics, the positions of joints are the target, and the rotations of the joints are solved for [13].

The first step for the user is to import the mesh for which they wish to define animations into the scene. They are prompted with a dialog window to select the Maya file to import. The mesh is then added to the scene, and the initial state of the mesh is saved. This data is maintained to ensure that the lattice, which is generated from the mesh's original state, can be correctly matched to the deformed mesh.

The artist is then able to animate the mesh as desired. At this stage, the user is working entirely with the mesh, so he or she can deform the geometry in any way using Maya's tools. One restriction, however, is that geometry can't be added or removed. That is, the set of vertices, edges, and polygons in the mesh must remain unchanged, although those elements themselves will transform. This is not an unusual constraint, but it should be noted.

At some point before exporting the animation, the user needs to import the appropriate particle lattice. This is done similarly to the forward solving case; the user clicks the "Generate" icon and selects the appropriate .rmb file. The lattice is represented as a mesh as before. The user does not need to make any modifications to the lattice mesh to create animations. The purpose of the lattice mesh is so that the user will be able to preview the state of the particle lattice that the inverse solver determines is the best fit for the mesh deformations.

When the animation is ready to be exported, the user clicks the "Export" button and is presented with a similar dialog as with the forward solving case. The user specifies the output file and framerate, and then clicks the export button. Scripts now solve for the state of the lattice at each frame, giving the user a preview of each frame as it is computed. The output file is of the same format as in the forward solving case; it consists of particle positions at each of the keyframes.

When forward solving, the input is a set of particle positions, and trilinear interpolation is used to determine the positions of the mesh vertices. However, this mapping is not invertible. Some configurations of vertices may have no associated configuration of particles which will yield the necessary deformation. As an example, consider a set of vertices in a line parallel to one of the coordinate axes, in this case the x axis, as shown on the left portion of Figure 5-3. Now suppose that, after a user deformation has been applied, one of the vertices is pulled away from the line, as shown in the right portion of Figure 5-3.

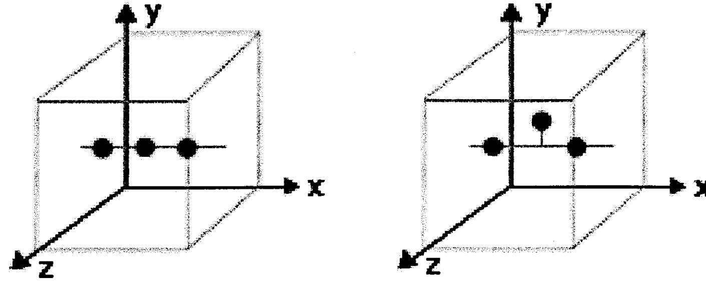


Figure 5-3. Transformation not realizable by trilinear interpolation. If the initial configuration of vertices lie in a line parallel to one of the coordinate axes as shown on the left, then a configuration of vertices where the points are not linear as shown on the right is not realizable.

All points on the left side of the figure lie on some line given parametrically by:

$$x = at + x_0$$

$$y = y_0$$

$$z = z_0$$

Equation 5-1

Applying Equation 3-7 to determine the weight for an arbitrary point on this line, we see that, since every weight is linear in x and x is linear in t, every weight is linear in t, so each weight takes the form:

$$w_i = b_i t + c_i$$

Equation 5-2

Equation 3-8 can then be applied to reconstruct the deformed positions of the points on the line, where $\vec{v_i}$ is the position of the ith particle of the enveloping cell.

$$\vec{x}' = \sum_{i=0}^{7} w_i \vec{v_i}$$

Equation 5-3

This is the sum over the product of values linear in t and constants, so the resulting sum $\vec{x}'$ is thus linear in t. One can thus conclude that any deformation on such a line must also result in a line, and so deformations such as the one shown in Figure 5-3 are not realizable by any configuration of particles.

Conversely, the positions of the set of particles may leave the system underconstrained, so that there are an infinite number of particle positions which would yield a certain vertex configuration. Consider a single vertex in an isolated cell, whose neighboring cells also contain

no vertices. With this configuration, the particles which enclose the vertex only affect that single vertex, so no constraints are introduced by their membership to neighboring cells. Even if it is assumed that the surrounding particles will form a cuboid, there is no information with regards to the scale or rotation of that cuboid, and as a result there are infinitely many possible particle configurations which would result in the correct placement of that vertex.

So given that the trilinear interpolation does not have a simple inversion, approximations can be used. This is done by associating a linear transformation with each cell. That is, the algorithm will try to find a translation, scale, and rotation that, if applied to the original positions of the vertices within the cell, will yield a least squares best fit. The particles can then be moved into positions so as to create this linear transformation for the mesh vertices by applying that same transformation to the particles. A single particle may be a corner for several cells, so the actual position to which it is set is the average of its calculated position for all of the cells to which it belongs.

To execute this approach, it is necessary to be able to approximate a best-fit linear transformation. That is, there is a set of vertices within a cell, their positions in the undeformed mesh, and their positions in the deformed mesh. A similar computation is already done during the shape matching phase of the soft body physics simulation, and this technique is given by Müller [15]. This technique seeks to find a rotation and scaling matrix **A** and translation vectors $\vec{t}$ and $\vec{t_0}$ that, when applied to the initial positions of the vertices $\overrightarrow{x_i^0}$, yields a least squares best fit to the deformed positions of the vertices $\vec{x_i}$. This approximated state of the lattice $\tilde{\vec{x_i}}$ is given by:

$$\tilde{\vec{x_i}} = \boldsymbol{A}\left(\overrightarrow{x_i^0} - \vec{t_0}\right) + \vec{t}$$

Equation 5-4

Müller shows that $\vec{t}$ and $\vec{t_0}$ are given by the deformed and initial center of mass of the vertices, respectively.

$$\vec{t_0} = \overrightarrow{x_{cm}^0} = \frac{\sum_i m_i \overrightarrow{x_i^0}}{\sum_i m_i}$$

$$\vec{t} = \overrightarrow{x_{cm}} = \frac{\sum_i m_i \vec{x_i}}{\sum_i m_i}$$

Equation 5-5

From here, the scaling and rotation matrix **A** can be determined as follows:

$$\vec{q_i} = \vec{x_i^0} - \vec{t_0}$$
$$\vec{p_i} = \vec{x_i} - \vec{t}$$
$$\boldsymbol{A}_{pq} = \sum_i m_i \vec{p_i}\,\vec{q_i}^T$$
$$\boldsymbol{A}_{qq} = \left(\sum_i m_i \vec{q_i}\,\vec{q_i}^T\right)^{-1}$$
$$\boldsymbol{A} = \boldsymbol{A}_{pq}\boldsymbol{A}_{qq}$$

Equation 5-6

This is computed for every cell at every frame. Since $\boldsymbol{A}_{qq}$ only depends on initial conditions, that matrix is only computed once per cell, and it is cached for use with subsequent frames.

It may be the case, however, that this approach fails when considering the set of vertices within a single cell. Some cells may have no vertices within them, so the result here is completely undefined. A cell with a single vertex contains information about the cell's associated translation, but nothing about rotation or scale. Even if we have an arbitrary number of particles, if they are all coplanar, then there is no data about the scale along that plane's normal axis. Fortunately, detection of this problem is simple; A_{qq} will not exist. That is, it is computed as a matrix inverse, and the matrix from which it is computed will be singular if there is insufficient information to compute the linear transform. If this case is detected, a larger vertex pool is used. The vertex pool is expanded by including the vertices within the neighboring cells. If this is still singular, we continue this process until we are able to compute A_{qq}, in which case we proceed, or our vertex pool contains the whole mesh, in which case this technique fails. This extreme type of failure can only happen if the entire mesh is planar.

The strengths of this approach address the weaknesses of the forward solving case. This comparison is summarized in Table 5-1. Artists are used to animating a mesh directly, and this technique permits them to do so. Additionally, this technique is resilient to new iterations of the soft body. If a soft body is made to be more or less detailed, then the lattice can be imported again while leaving the mesh animation intact. The artist can then export the animation again, and then the animation will be defined for the new lattice. One downside to this technique is that all details might not be preserved. The artist is permitted to place vertices in positions not realizable by any lattice configuration, so those details may be smoothed over during the shape

matching. Additionally, when using the forward solving approach, the artist is given a real-time preview of the effect on the mesh. Similarly, exporting is nearly instantaneous. The matrix computations here are more computationally intensive than a simple linear interpolation, so a real-time preview of the lattice as the mesh is deformed is not possible on modern machines. Similarly, exporting takes on the order of a few seconds. Even given the disadvantages, the usability and resilience make this a powerful technique.

Forward Solving	**Inverse Solving**
=Artist specifies the state of the particle lattice	=Artist specifies the state of the mesh
+Results can be previewed in real-time	-Results take on the order of seconds to preview
+The artist deals with the animation representation directly	-The artist has only indirect control of the actual exported animation
-Manipulating particles is not the way artists are trained to animate	+Artists can animate using the techniques in which they are trained
-Animations are defined for the particle lattice, which may change	+Animations are defined for the model, resilient to changes in the particle lattice structure

Table 5-1. Comparison of forward and inverse animation solving.

5.4 Plugin Design Considerations

In Maya, scripts can be executed in response to user activation, usually clicking a button. Additionally, the data created in the scripts is temporary and lost when the Maya session is restarted. These restrictions can be circumvented by using what Maya calls plugins. Maya maintains a directed acyclic graph to represent the scene. This scene contains nodes, which may, for instance, represent transformations or geometry. Plugins are able to define new custom nodes one can insert into the scene graph.

An objective of the forward solving implementation was to permit an interactive preview of the state of the target mesh in response to the state of the particle lattice. By extending

Maya's MPxLocatorNode, the new class, trilerpNode, is able to receive a callback whenever the scene needs to be redrawn. This will be triggered whenever the particle lattice is modified, or whenever the user navigates the animation timeline. It will also be triggered, however, if the camera within the scene moves. To avoid unnecessary recomputation, the state of the particle lattice is cached, and the target mesh is only updated if the particle lattice has changed since the last update. This also protects against an infinite loop of draw calls. Updating the target mesh forces a redraw, so there is another draw callback triggered.

When running scripts, they execute in a temporary Python environment. Restarting Maya means restarting the Python environment, so any data managed only by the scripts is lost. For this purpose, the rmDataNode, which extends Maya's MPxNode, is created. Nodes are permitted to maintain persistent attributes. Therefore, data which might have otherwise been lost can be stored. This includes the lattice spacing, mesh scaling, mesh offset, mapping between vertices and particles, and for inverse solving, the initial state of the mesh. Much of this data is read from the .rmb file and cannot be inferred from the state of the scene. These nodes can't natively store arbitrary data, however. One data type they can store is a string. Fortunately, Python's pickling module allows data to be converted to and from a string representation, so this method of storage is perfectly sufficient. So this means that when Maya is closed and restarted, the data can be extracted from this node, and the artist can proceed exactly from where they stopped, requiring no extra data files, just the Maya scene file.

It is also the case that, if a new scene is loaded without restarting Maya, then the Python environment is unchanged. This means that it must be possible to distinguish this scene's data from temporary Python data. In the case of the data previously mentioned, that is solved by reloading the data from the rmDataNode at the start of every script, which is not terribly expensive. Extra care must be taken during the inverse solving, however. A_{qq} is stored for each cell. This means that the Python environment must store some data structure of cells and matrices. This was done by creating a custom cell datatype. These cells have pointers to one another, as they keep track of their neighbors. This type of data is not readily pickled into string form, so it only lives in the Python environment and not in the data storage node. It is desirable to avoid recomputing this when necessary, though. This is done by including a special boolean attribute on the data node, needCellRecompute. This attribute is configured so that it is not persistent, it resets to true every time a scene is reloaded. In this way, this attribute can be

consulted before proceeding with the set of cells in the Python environment. If recomputation is necessary, then that is done, and `needCellRecompute` is set to false. This ensures that the script does not mistake residual cell lattice data from another scene as the data from this scene.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Game Engine Integration

In addition to the project's goal of adding new functionality, creating a soft body physics engine capable of allowing soft bodies to track dynamic poses, it was also a goal to improve the usability of the library. The user interface design was already talked about for the animation tools, but the details of how one sets up and runs a simulation are thus far unaccounted for.

As received, RealMatter was purely a code library, with a full C++ implementation and a partial C# implemention. Setting up and running a simulation was thus easy to do as a programmer, but one of the primary applications of this work is video game creation. Artists or designers, people who do not necessarily have a programming background, should be able to create and modify soft bodies in a game without programmer intervention.

6.1 Unity

This project involved the integration of the soft body physics into the game engine and development tool Unity, created by Unity Technologies. This integration was done with Unity v3.3. Unity has several features which made it a good candidate for integration. It is designed for 3D games, which is necessary to showcase extent of the capabilities of the soft body physics. It has a visual scene editor. This means that any user, even without programming knowledge, can assemble and modify a scene. It has NVIDIA's PhysX rigid body physics engine built in. The soft body physics engine was developed to cooperate with the rigid physics engine, which enables interaction between soft and rigid bodies. Unity supports the export of builds to multiple platforms, including Windows, Mac OSX, web, iOS, Android, XBox 360, and Wii.

Unity also supports several languages for scripting. Among these languages is C#, of which a partial implementation of RealMatter existed. The full C++ library is indirectly usable through dlls, but this would restrict the set of target platforms substantially. Web deployment was a goal of this project, so the integration was performed with C#. Finally, Unity's editor can be modified. New buttons and menus can be created, which could aid in a more seamless integration of the soft body interface with existing interface elements. A glimpse of the Unity Editor can be seen in Figure 6-1.

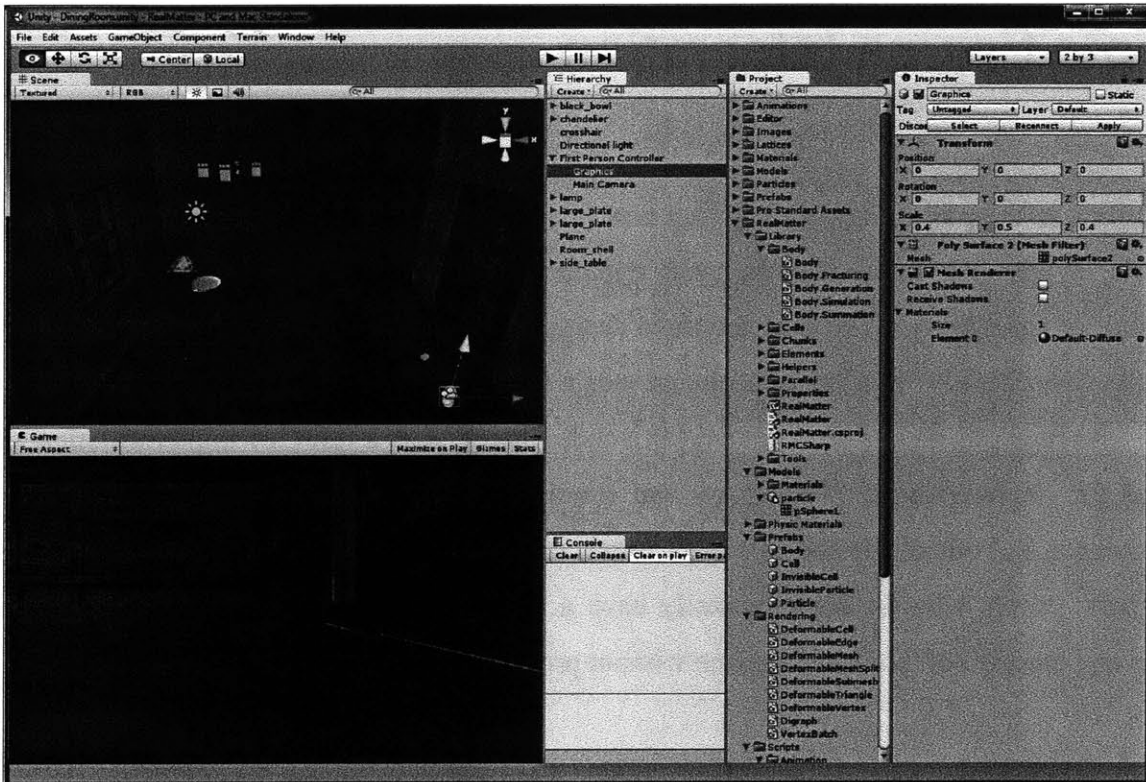


Figure 6-1. Screenshot of Unity editor. The top left window allows for placement and manipulation of objects within the scene. The right pane allows for numerical adjustment of object properties, as well as the attachment of scripts, as Unity uses a component based object model.

6.2 Integration

In order to simulate soft bodies in Unity, there must be some information exchange between Unity and RealMatter. For a given body, every soft body simulation step takes the positions and velocities of all of the particles in its lattice and produces new velocities. With these as the only

necessary inputs and outputs, bodies and particles are the only objects which require synchronization.

Any object in Unity which is to be a soft body should have the `BodyScript` component attached. This component maintains a reference to the RealMatter body object. When the game starts, this script is responsible for initializing the RealMatter body. Every `BodyScript` has several parameters the user can control. These include the lattice spacing, region half-width *w*, and restorative coefficient *alpha*. This script is responsible for taking the input mesh, as specified by the object to which the script is attached, and creating a particle lattice.

When a body is built, a lattice of particles is created. For every one of these particles, an object is created within Unity. This object will have an attached `ParticleScript`. Similar to bodies, each `ParticleScript` maintains a reference to the RealMatter particle with which it is associated. Every particle is a small chunk of the soft body which should be treated as locally rigid, so every particle also has a Unity `RigidBody` component. This component informs Unity that this object should take part in the physics simulation. This enables automatic updating of its position and velocity according to the particle's interactions with the world. In order to allow for collisions to take place, every particle is also given a `SphereCollider`. Collisions with other colliders will be appropriately handled by Unity's physics engine.

Once simulation has started, `BodyScript` receives a callback after every Unity physics step. This script first loops through all of the rigid bodies associated with all of the particles and feeds those positions and velocities into RealMatter. Once this is done for all particles, the RealMatter library is used to advance the soft body one time step. The script then reads the updated velocity of every particle and assigns it to each associated rigid body. This occurs every physics time step, usually once per rendering frame.

While `BodyScript` is responsible for simulation updates, the `MeshShape` script is responsible for the rendering updates. When initialized, this script precomputes the weights for each vertex used in the trilinear interpolation technique for computing the state of a mesh from the state of the particle lattice. It receives a callback every rendering step, and updates the vertex positions accordingly. Vertex normals are updated using Unity's built in functionality for doing so.

Care was taken with the design to allow for proper rendering of fractures. When a mesh is fractured, vertices are split as triangles are severed at the boundary, and new triangles must be

drawn at the exposed surface. The user is able to specify a rendering material to be applied to the interior surfaces. New vertices and triangles are able to be dynamically added as necessary, as seen in Figure 6-2.

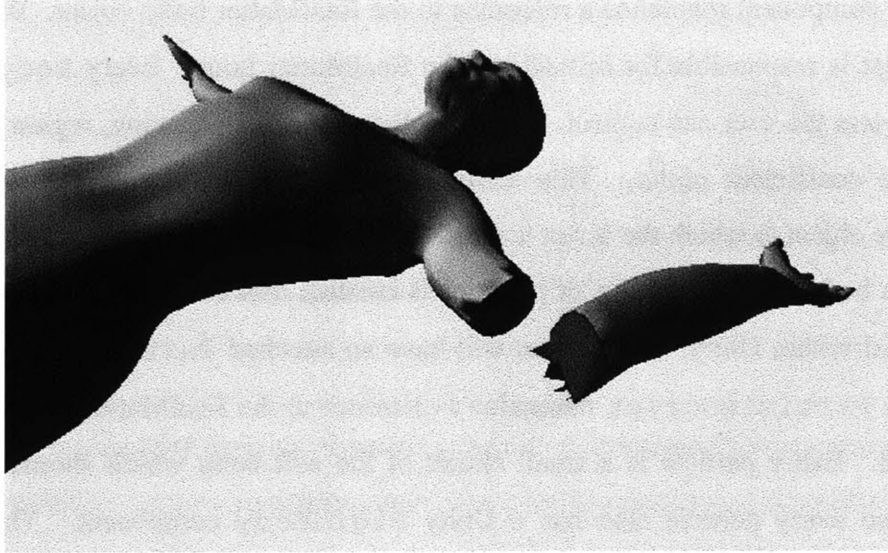


Figure 6-2. Fracture rendering. The user is able to specify a material for exposed surfaces, and that is used when new vertices and triangles are created at a fracture boundary.

6.3 User Interface

Unity has support for modifying its user interface. In this project, several modifications were implemented to allow for easy access to the soft body functionality.

Once the user has placed an object into the scene using the usual tools Unity provides, making it a soft body only requires a few clicks. The user need only select the object and choose "Soft Body" from the component section of the top menu bar, Figure 6-3.

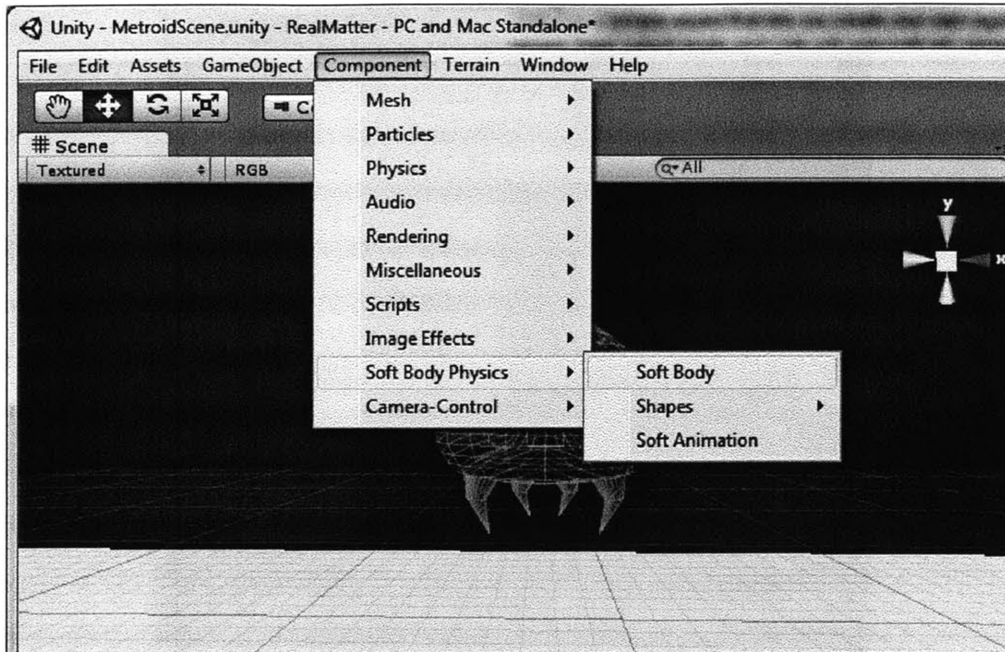


Figure 6-3. Soft body menu bar. Turning a 3D model into a soft body can be done by placing it into the scene and selecting this option from the menu bar.

The necessary soft body components are then attached once this button is clicked. The soft body defaults to reasonable parameters, but they are all exposed to the user through the interface shown in 6-4.

The Unity representation of the particle lattice can then be built in the editor by clicking the "Rebuild Body" button. This is done manually so that the body isn't generated before the user has a chance to tune the parameters. It's possible that the user has a large model and wants a large lattice spacing - if the lattice were automatically generated with the default parameters, then a very large number of particles might need to be created, and the software would appear to hang. "Rebuild Body" should be clicked any time a change of properties would bring about a change in the lattice, for instance by changing the particle spacing or mesh scale. The computed particle lattice is then visible in the editor upon selection as seen in Figure 6-5.

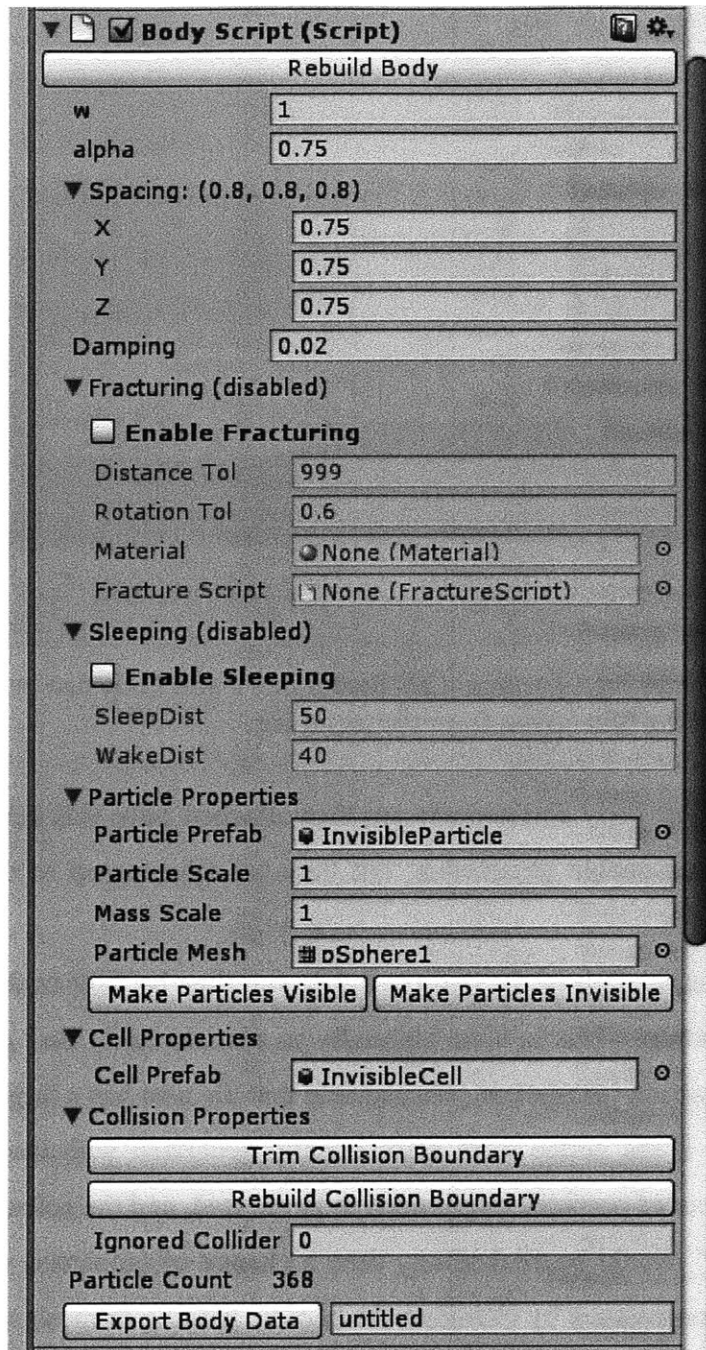


Figure 6-4. Soft body properties editor. The user is given full access to all of the properties used in the creation and simulation of the soft body from this interface. The interface here is fully expanded; it is collapsed by default to allow for easy access to common operations.

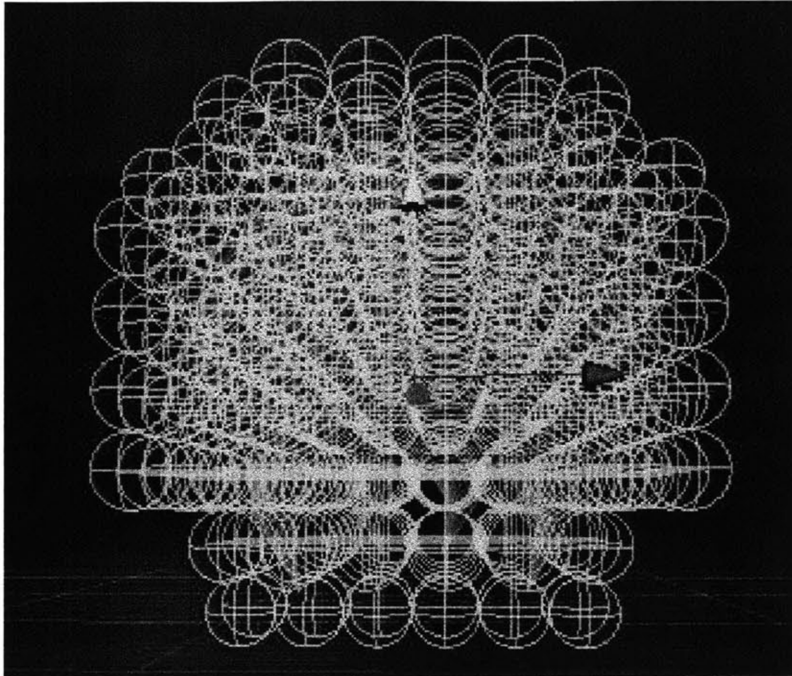


Figure 6-5. Lattice view. When selecting a soft body, the lattice representation can be seen in the editor.

Any of these particles are individually or collectively selectable. Since Unity treats them as rigid bodies, they can handle other rigid body operations such as the addition of joints. One or more particles can be fixed to a rigid body or the environment in this way.

From here, the user is able to hit Unity's "Run" button, the usual way of starting a game, and the simulation will work as desired, making use of the soft body properties specified by the user. Interaction with all other game objects happens automatically.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Performance

While the focus of this project was not explicitly performance engineering, the intended real-time applications of these tools mandates that the simulation can run at a reasonable speed. RealMatter was engineered to run at real-time rates [9]. The results in the final product here are slower than those reported in the paper for several reasons. Soft bodies here are engineered to take part in collision detection with rigid bodies in the world and with other soft bodies, whereas those in the paper only collide with static geometry. Additionally, this code was written in C#, whereas the results in that paper were achieved using a C++ implementation. This decision was made so that the game would be playable on web pages, as execution of the dlls which would have been necessary to execute the faster C++ implementation is not permitted on web pages for security reasons. Despite these tradeoffs, the software still performs in real-time, while supporting more complete collision detection as well as facilitating distribution.

Soft body animation is reasonably expensive. Every frame that the target pose is updated, some set of what would otherwise be invariants must be recomputed. In particular, each region will in general have a new center of mass and configuration from which a rotation is computed. In the case where the animation is to be played back completely smoothly, this computation takes place every frame. This cost is high, but it was mitigated by parallelization. Another solution is only changing the soft body's target pose at keyframes. The velocity driven motion of the particles will inherently introduce smoothness, so this solution may be sufficient in

many cases. Since this requires far more infrequent recomputation of invariants, the average time spent per frame is far less.

The primary parallelization technique utilized C# threadpools. There are many computations which occur independently over some collection, such as particles or regions, so these types of jobs are evenly divided and assigned to threads, resuming computation once all threads signal themselves as terminated. The simulation has a serial backbone, so this type of computation can only be done at certain junctions, but it does produce measurable improvements, especially with animations which update every frame.

These statistics were taken from a desktop with two Intel Xeon E5462 processors running at 2.80 GHz. Each processor has four cores, so this machine has eight cores total. The computer has an NVIDEA GeForce 8800 GT graphics card.

These measurements were done on a scene with a soft body containing 420 particles, a reasonably high particle count. The amount of time spent per frame was computed when the soft body had no animation, animated updating its target only at keyframes, and animated updating its target every frame. The provided animation provided a new keyframe once per second. The parallelization had the largest impact on continuously updating animation. In order to emulate usage in applied scenarios, these frame times include the time spent on rigid body simulation and rendering.

Threads	Unanimated Frame Time (ms)	Smooth Animation Frame Time (ms)	Keyframe Only Animation Frame Time (ms)
1	16.7	49	17.4
2	15.9	31.2	16
3	15.9	26.9	16
4	15.9	26.1	16
5	15.9	25.4	16
6	15.9	25.6	16
7	16.1	24	16
8	16	24.5	16.2

Table 7-1. Simulation time per frame for different animation schemes. This is the amount of time it takes to simulate for one frame on a test scene using three different animation schemes. This performance is seen as a function of the number of parallel threads.

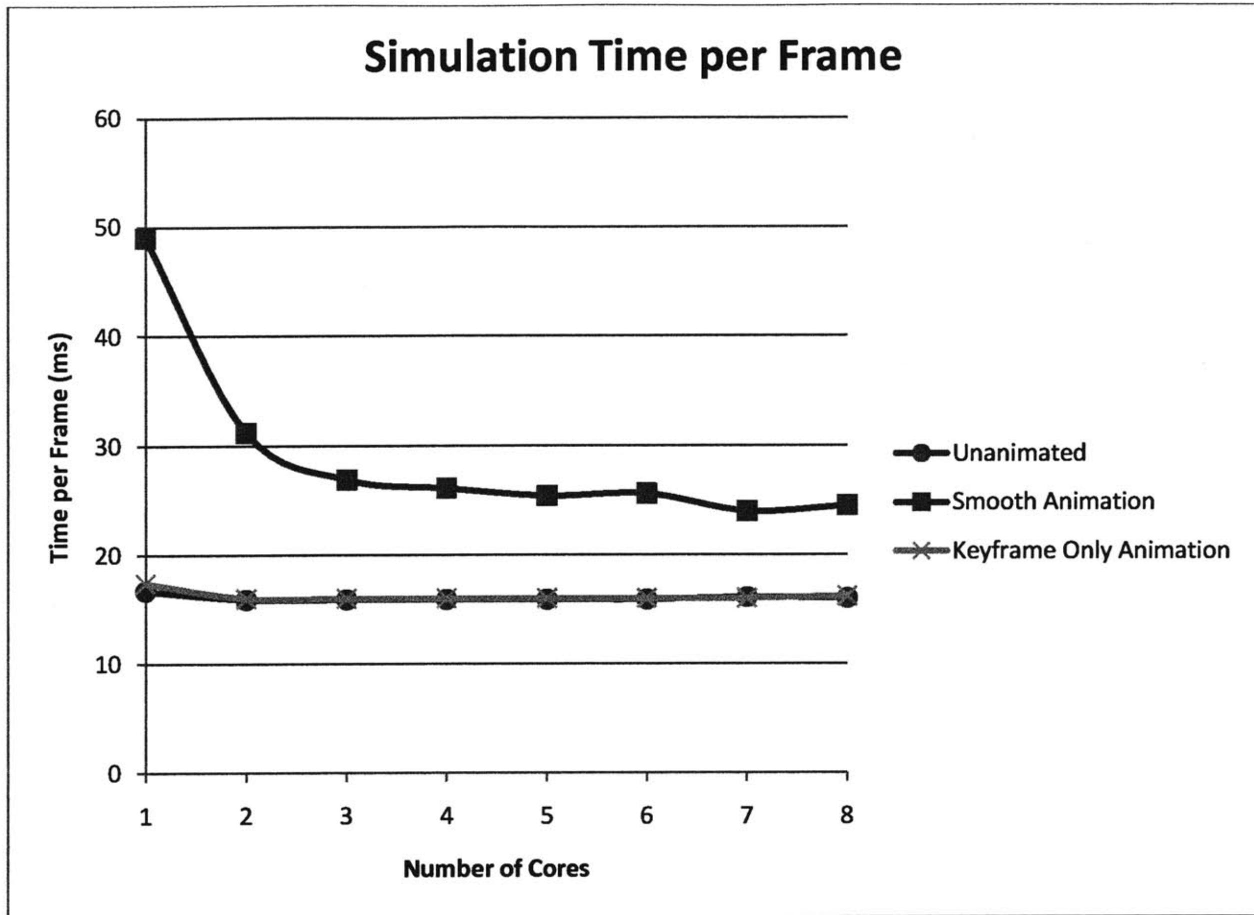


Figure 7-1. Simulation time per frame for different animation schemes. Note that an unanimated object has nearly identical performance to keyframe only animation.

The performance of the Maya animation creation tools was measured as well. In both the case of forward and inverse solving, the same mesh, consisting of 690 vertices, and particle lattice, consisting of 420 particles, were used. Deforming the mesh with forward solving, each update averaged at 22.6ms. This corresponds to approximately 44 updates per second, a rate which would readily be considered real-time. Using inverse solving, computing the first frame, which takes care of initialization overhead, averages at 2309ms. Subsequent calculations averaged at 815ms per frame. So while this implementation of inverse solving is not able to be run in real-time, an artist with a comparable machine and a similar model should be able to preview poses within a second, and export entire animations on the order of a few seconds. Performance in each of these cases was satisfactory and not as much of a concern as in the simulation. Were performance here more of an issue, these algorithms could have been implemented in faster C++ plugins rather than Python for some boost.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Conclusion

Real-time physics simulation has become prevalent in many modern games. However, rigid body dynamics are far more developed than soft body dynamics. Even with the development of real-time simulation of deformable objects, we are still left with animation techniques such as skeletal animation, which work well with rigid bodies, but fail to easily express the deformation affordances of soft bodies.

The particle driven animation techniques presented here seek to address this problem by creating an animation representation which is both natural for simulation and allows for the full range of non-rigid deformations which can be simulated by particle driven simulation. Additionally, animations created using traditional 3D animation techniques can be adapted to soft bodies using the inverse solving techniques described here. While designed for a particular soft body physics engine, it is likely that these animation techniques could be adapted to other systems which represent soft bodies as a set of discrete particles.

These animation tools, along with a simulation framework, were successfully designed and developed. It is our hope that these tools will make it so that even those without a programming background, such as many artists or designers, will be able to create soft body simulations and animations on their own when creating a game.

There is still plenty of room for similar future work. A soft body's collision geometry is currently represented as a sphere at the site of each particle. This is only an approximate reflection of the body's actual shape. If the body undergoes large deformations, then the particles may separate sufficiently and create holes in the object's collision geometry.

Additionally, the current implementation of the soft body simulation is run entirely on the CPU. More extensive parallelism may be possible if the algorithms can be adapted to run on a GPU.

The author hopes that the techniques developed here, and the corresponding tools which have been created, will be used by others to create games which showcase interesting physics. The tools are eventually intended for a general release. In Summer 2011, a team of undergraduate interns at the Singapore-MIT GAMBIT Game Lab will be making a game to showcase the new technology developed here. It is the hope of the author that this will be the first of many projects to use this software.

Appendix

A.1 User Guide

What follows is the user guide that will be distributed with the tools described. These instructions explain what the tools do, and how a user would accomplish common operations.

Unity-RealMatter User Guide

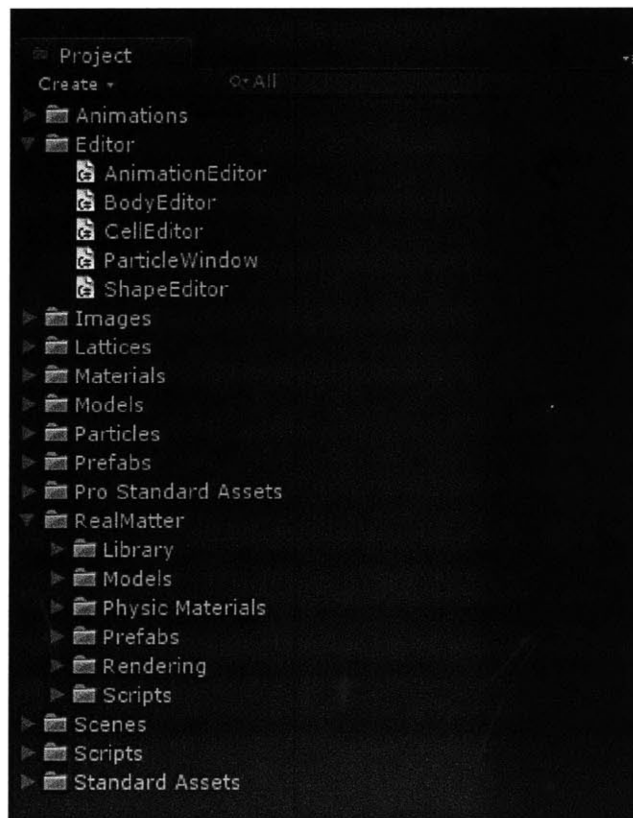
What are these tools?

These tools allow for the simulation of soft bodies within your Unity game. "Soft body physics" is a bit of an umbrella term, which may include everything from cloth to hair physics. The type of soft body physics here primarily refers to rubbery volumes. The soft bodies created with these tools are able to interact with existing colliders and rigid bodies in your Unity scene.

These Unity plugins allow for generation and simulation of soft bodies in a game. This document assumes that the user is familiar with the Unity editor. These scripts were designed for Unity 3. In addition, a set of tools were created for Autodesk Maya 2011 x64; these are not necessary for basic soft body simulation, but they allow for creation of animations usable by the Unity tools.

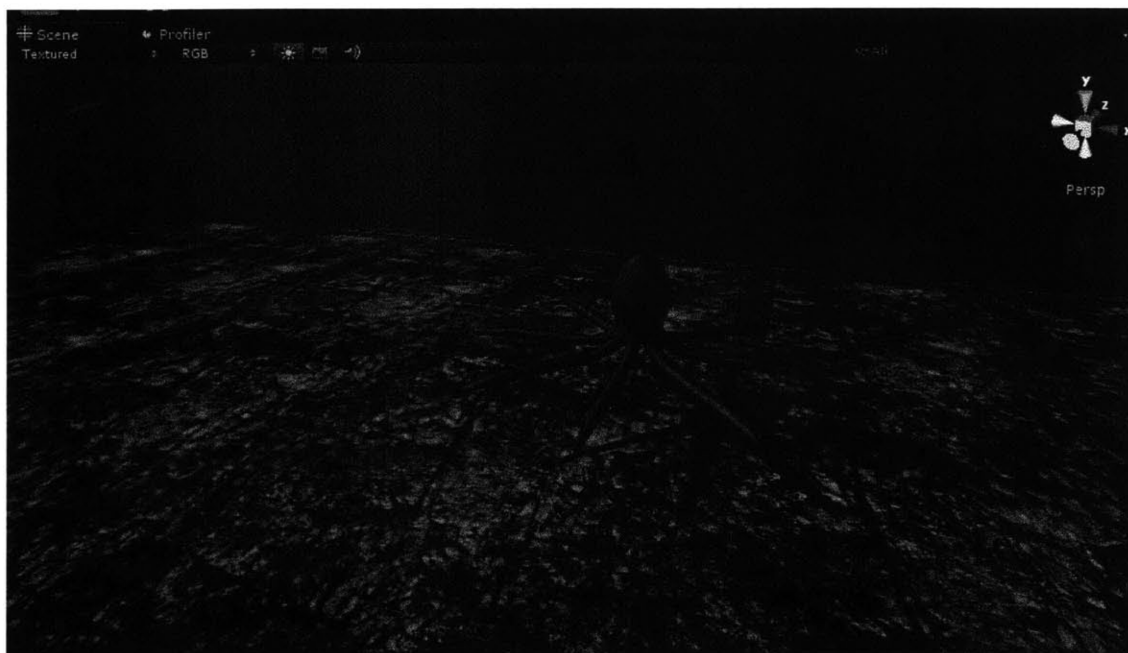
Unity Tools Installation

The relevant files, which are bundled in a Unity package, extract to the folders ./Assets/Editor and ./Assets/RealMatter in your project directory. Those in the Editor folder include augmentations to the Unity user interface. Those in the RealMatter folder are needed for simulation.

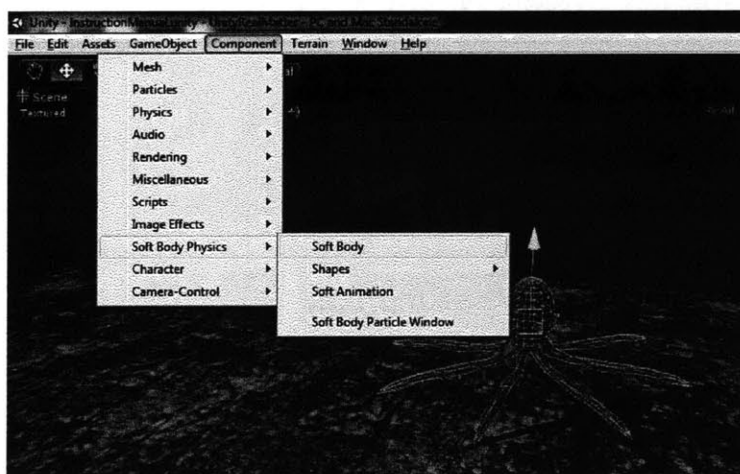


Creating Soft Bodies

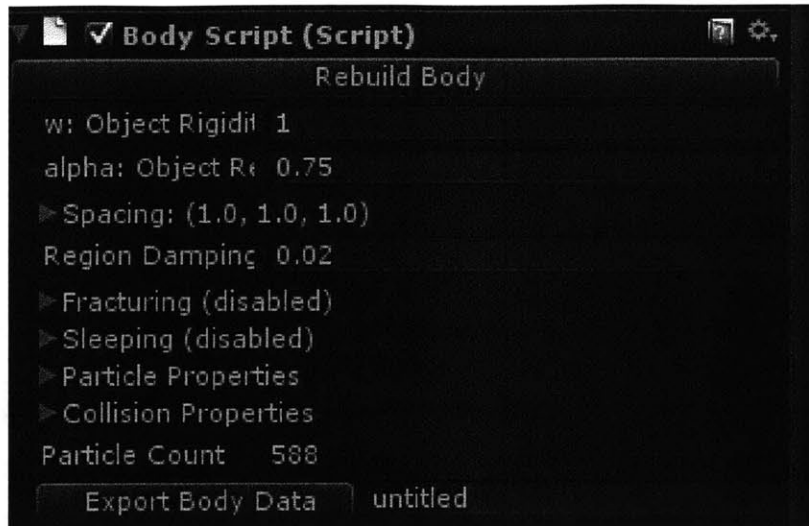
Let us suppose you have a scene with an object that you want to simulate as a soft body. As an illustration, I will use an octopus model. This model must be a prefab of a model with ONE mesh. This can be done in modeling software, and it will be discussed later how to ensure this is the case in Maya.



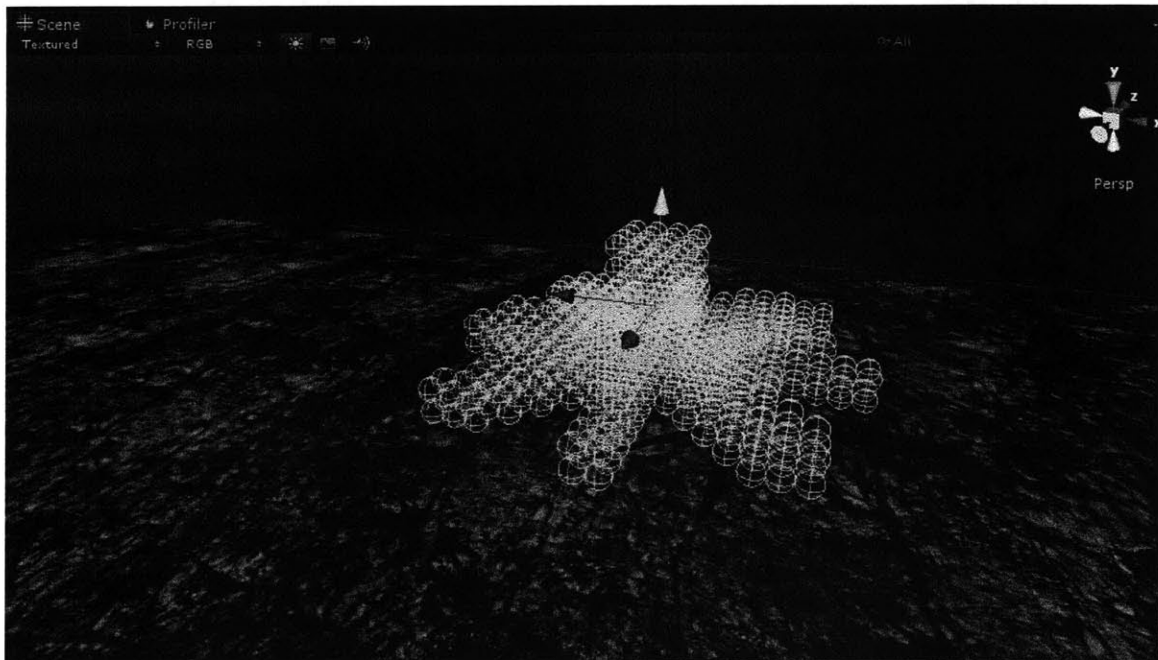
To make an object a soft body, select the object in the editor, and, from the main menu, select Component->Soft Body Physics->Soft Body. Select 'Add' if it prompts you that adding a component will lose the prefab parent.



Now, our object will have a Body Script component which appears in the inspector pane when selected. This is the primary way that an object's soft body behavior will be modified. This will also add a Mesh Shape script, and remove any collider or rigid body attached to this body.



Whenever the object's soft body representation should be generated, the user should hit "Rebuild Body." In this example, I will click that button without making any parameter changes. If you are trying this with your own model, you may want to rescale your model if it is very large. Clicking "Rebuild Body" discretizes the mesh into unit cubes by default, and this may generate a very large number of particles for a large mesh.



When the object is selected after clicking "Rebuild Body," it should look something like this. You can now hit play to start the game, and you should have a functional soft body!

Each of those spheres is the collision geometry for the particle. Every particle is a rigid body, and every one, by default, has a spherical collision geometry. In the hierarchy, these are parented to the soft body.

Soft Body Properties

The soft body might not behave as you want right away. It might be too squishy or too rigid, the lattice may be too coarse or too fine, the collision geometry might not be perfect. Fortunately, there are ways to customize your soft body to make it behave the way you'd expect. With the soft body selected, there are several parameters which can be changed from the inspector.

Rebuild Body

This button creates the particle lattice. Whenever a change is made to the body which mandates regeneration of the body (initial creation, changing the object's scale, or changing the lattice spacing), this button must be manually clicked. This is partly because of the consequence associated with the button. It is a safeguard against the user inadvertently generating too many particles. Additionally, rebuilding the lattice is somewhat destructive, since all joints and modifications to the collision geometry will be lost, since the lattice is now completely different.

w: Object Rigidity

Default value: 1. This is an integer which affects the stiffness of the lattice. Soft bodies are simulated by approximating regions of particles as locally rigid. This number specifies exactly how local that computation is. When w is 1, every particle looks at its nearest neighbors. When w is 2, every particle looks at its neighbors and second neighbors.

alpha: Object Restitution

Default value: .75. This is a floating point number which specifies the restoring force the lattice is allowed to apply. Intuitively, one can think of it like a spring constant. At low alpha, an object will have a harder time maintaining its shape. At high alpha, an object will be more effective at tracking its target pose. Alpha should be less than 1 to ensure stability.

Spacing

Default value: (1,1,1). This is a 3-tuple of floating point numbers. When the mesh is discretized, this value specifies how coarse or fine that discretization is. A smaller spacing means smaller voxels, and thus more particles. If this value is modified, the user must click "Rebuild Body" for their changes to take effect. Note that Unity does not support anisotropic scaling of spherical collision geometry, so, if different numbers are used for different dimensions of spacing, spherical collision geometry will default to having a diameter of the largest value.

Region Damping

Default value: .02. This is a floating point number which specifies damping to be applied by the simulation. Smaller damping means that objects will move faster, but may become unstable.

Fracturing>Enable Fracturing

Default value: False. This is a Boolean which specifies whether or not an object should be allowed to fracture.

Fracturing>Fracture Distance Tolerance

Default value: 999. This is a floating point number which specifies the translational tolerance for triggering a fracture. A smaller number means that it's easier to trigger a fracture by squashing or stretching the object.

Fracturing>Fracture Rotation Tolerance

Default value: .6. This is a floating point number which specifies the rotational tolerance for triggering a fracture. A smaller number means that it's easier to trigger a fracture by bending the object.

Fracturing>Fracture Material

Default value: None. This is an object of type Material which specifies the material used to generate the closing surface when a fracture occurs. That is, what material will be used to 'fill in' the inside of an object.

Fracturing>Fracture Script

Default value: None. This is a script which inherits from FractureScript. The abstract class FractureScript can be found at ./Assets/RealMatter/Scripts/FractureScripts/FractureScript.cs. The script here will receive a callback whenever a fracture occurs. It might be used to trigger functional events (you broke the object so unlock a door) or graphical effects (create a particle effect at the fracture site).

Sleeping>Enable Sleeping

Default value: False. This is a Boolean which specifies whether or not a soft body should fall asleep if you are far enough away from its center. When an object is asleep, it stops being simulated so performance will improve. If poor thresholds are set, however, an object may appear to suddenly freeze and become noninteractable.

Sleeping>Distance which Triggers Sleep

Default value: 50. This is a floating point number which specifies how far away you must be from an object for it to fall asleep.

Sleeping>Distance which Triggers Waking

Default value: 40. This is a floating point number which specifies how close you must be to an object for it to wake up from sleep. This should be smaller than the sleeping distance.

Particle Properties>Particle Prefab

Default value: InvisibleParticle. This is a prefab which is instantiated at the site of each of the particles. By default, it is an invisible particle with spherical collision geometry. Different particles can be placed here to, for instance, change the material of a particle,

change its collision geometry, or turn off gravity. Note that particles do not have an associated rotation, so be careful if using nonspherical collision geometry. To use the existing particle as a template, it is fount at ./Assets/RealMatter/Prefabs/InvisibleParticle.prefab.

Particle Properties>Particle Scale

Default value: 1. This floating point number specifies any additional scaling that should be applied to the collision geometry of the particles. Rebuilding the body is not necessary if this value is changed, its results are interactively previewed.

Particle Properties>Mass Scale

Default value: 1. This floating point number specifies the scale that should be applied to the mass of the particles. When using the default prefab, particles have mass 1, so this directly specifies the mass of each particle.

Particle Properties>Particle Mesh

Default value: pSphere1. This Mesh specifies what should be used to visualize particles without a mesh. This mesh is found at ./Assets/RealMatter/Models/particle/pSphere1. For instance, InvisibleParticle, the default particle prefab, has no built in Mesh, so visualization is undefined without this.

Particle Properties>Make Particles Visible

This button can be used to visualize all of the particles. The mesh placed at each particle is given by Particle Mesh above. This is handy because, if we can draw particles, than we can easily select particles by clicking on them or box-selecting them, which will make things like placing joints or modifying collision geometry easy.

Particle Properties>Make Particles Invisible

This button destroys the visualization of all of the particles, as created by the previous button.

Collision Properties>Trim Collision Boundary

This is a common way to remove some of the excess particles at the fringes of the lattice. This button removes the collision geometry for all particles which do not have all of their neighbors, so it preserves collision geometry for those particles in the center of a volume. This does not work well for objects with long and skinny parts, since it may trim all of those collision geoms.

Collision Properties>Rebuild Collision Boundary

This button is a nice way to fix your mistake if you realize you trimmed too much. It adds a collider back to every particle in the lattice.

Collision Properties>Ignored Collider Count

Default value: 0. This integer specifies how many colliders the soft body should ignore. Specifying a nonzero value for this will allow the user to add that many game objects to a list. The soft body will then not collide with those objects during simulation.

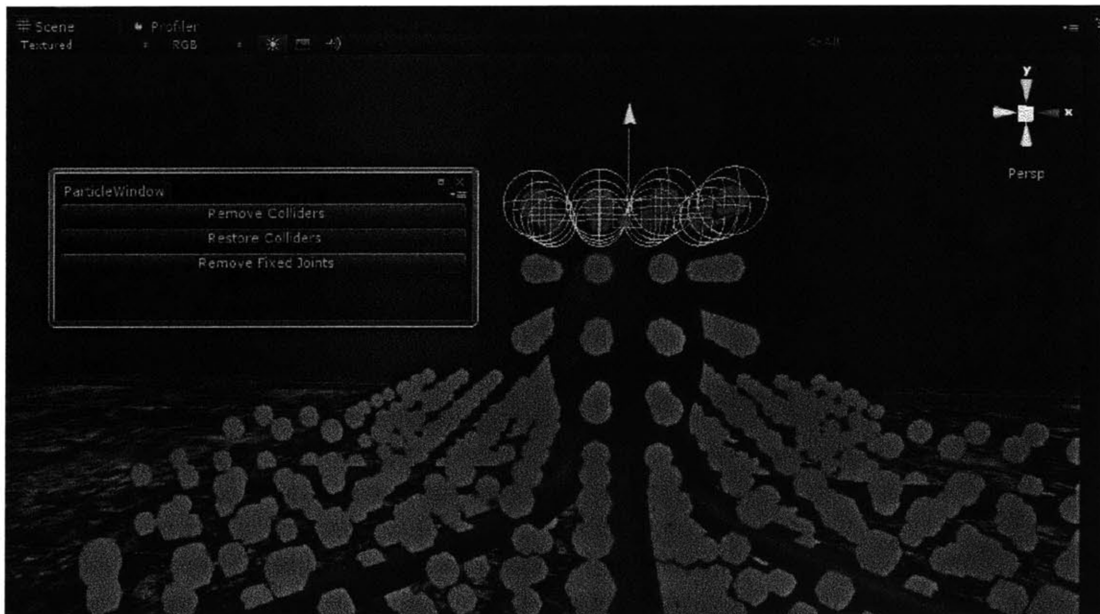
Particle Count

This value is immutable by the user, but displays how many particles are in the current lattice.

Export Body Data

This is used by the soft body animation tools. This exports the current body configuration information. When the body is named and this button is clicked, a file is generated in ./Assets/Lattices/*.rmb. This RealMatter body file is a text file with the following format. All pieces of data are new line delimited, and all tuples are comma delimited. The first piece of data is the center of the mesh, as measured from the center of the particle lattice, that particle with index (0,0,0). Next, it has the spacing of the lattice. Next, the scale applied to the mesh. Finally, it lists the 3D index of each particle in the particle lattice.

Many of these properties affect the whole body, you may find yourself wanting to make local changes. Modifying each particle individually can be a pain, so some common operations are made easy by ParticleWindow. This window can be launched from Component>Soft Body Physics>Particle Window, or Window>Particle Window. It is primarily designed for use when the particles have been made visible in the body.



Remove Colliders

This removes the collision geometry from any selected particles. This allows for finer control than the bulk operation "Trim Collision Boundary."

Restore Colliders

Restores collision geometry to any selected particles, the inverse of the previous button

Remove Fixed Joints

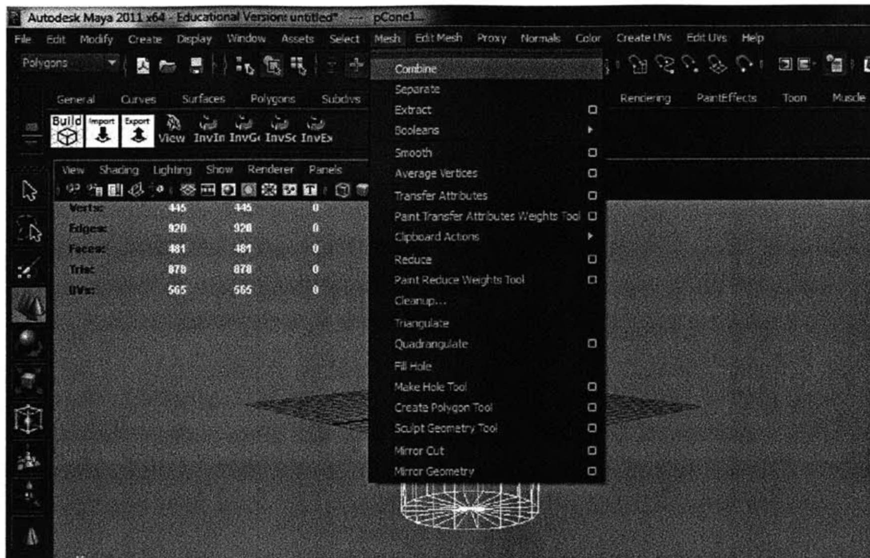
Parts of a soft body can be fixed into place by highlighting them and adding a fixed joint. Unity lets you do this in bulk, but it does not easily let you remove them. Here, you can remove those joints by selected the target particles and clicking this button.

Modeling

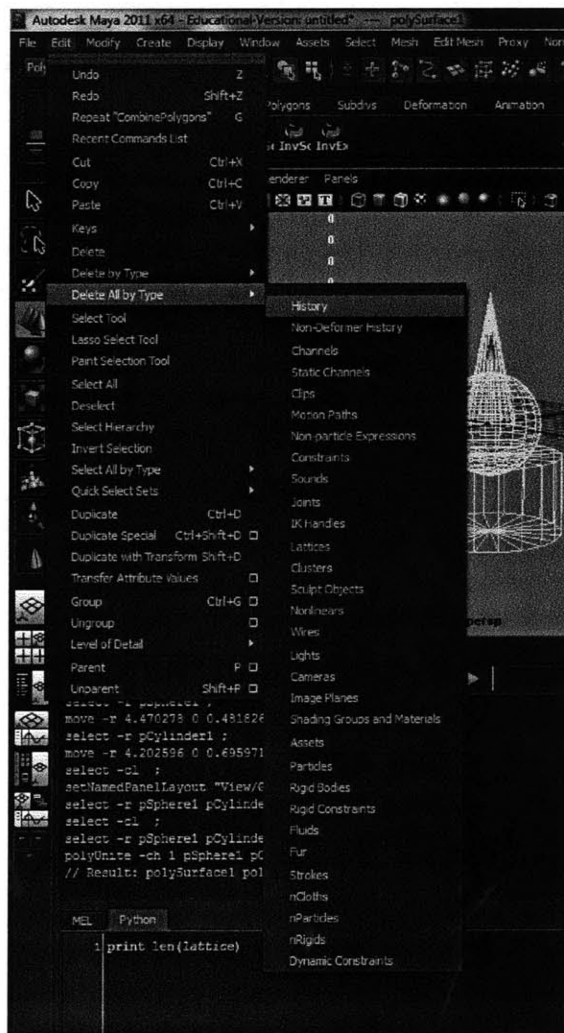
Models for the soft body generation must be a single mesh. Here are steps to ensure that happens in Maya 2011 64-bit (other versions of Maya may work as well but this was the development and testing environment).

You can model the object as usual, using compound meshes if necessary. I recommend saving a working version of your file, since this process will destroy your modeling history and hierarchy.

First, select all of the objects in your scene. Then, with "Polygons" as your active menu set, select Mesh>Combine.



Next, click Edit>Delete All By Type>History.



If done properly, there should only be one mesh left when you view the Hypergraph window. You can now save the Maya file to your Unity Assets folder.

Maya Animation Tools Installation

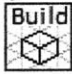
In order to create soft body animations, you must place the files rmDataNode.py and trilerpNode.py into you Maya plugin directory, <Maya Install>/bin/plug-ins/. These plugins can be loaded via Window>Settings/Preferences/Plug-in Manager. I recommend selecting Auto load for the previous two scripts.

You'll also want to install the RealMatter shelf. To do that, place the file shelf_RealMatter.mel into the directory <Maya Workspace>/2011-x64/prefs/shelves. You may need to restart Maya, but once you do, you should see the RealMatter shelf.

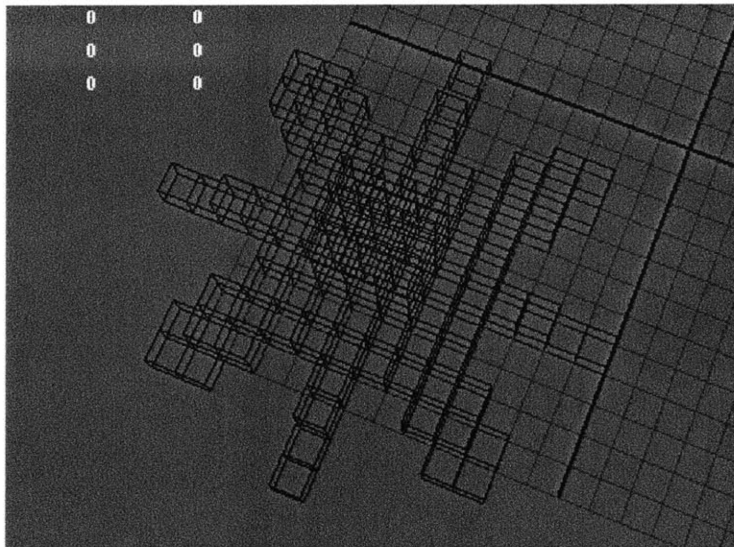
There are two ways to specify soft body animations, forward and inverse.

Forward Animations

When you define a 'forward animation,' you will specify a keyframed sequence of poses on the particle lattice. First, you will have to import the particle lattice. You can configure and export the lattice in Unity as discussed before, but as a reminder, once the body is configured, click "Export Body Data" on the Body Script inspector window.

Click Build 

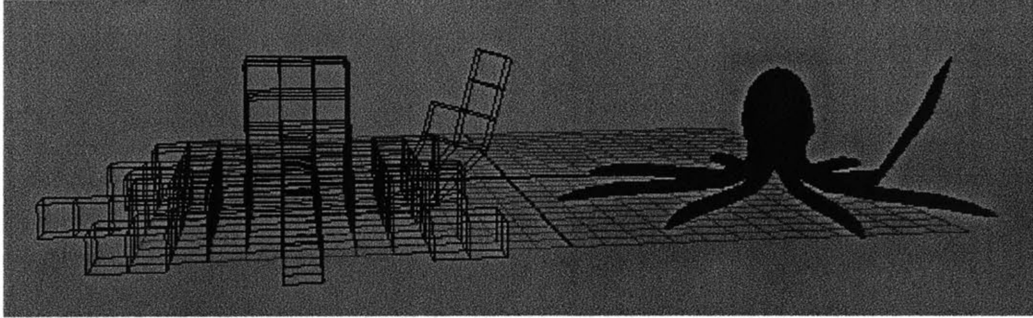
to load this file into Maya. You will be prompted to locate the lattice file, which is saved by default to <Unity project directory>/Assets/Lattices. This script reads in the particle data, and creates a vertex at the site of every particle.

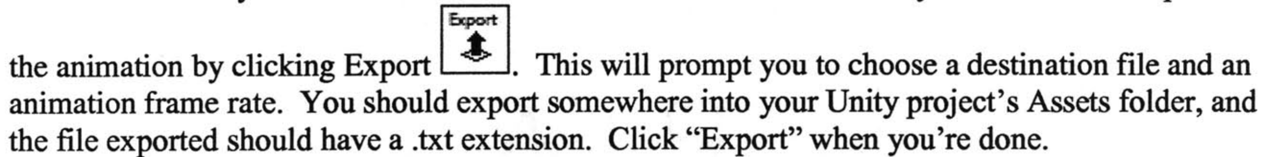


You can animate this particle lattice as you would any other mesh. You can pose it and set keyframes. Since it is just a collection of vertices and edges, you use existing Maya tools, like skeletal rigging, or you can manually position the vertices at keyframes.

You may want to preview the effect a particle lattice deformation has on its target mesh. To do that, click Import

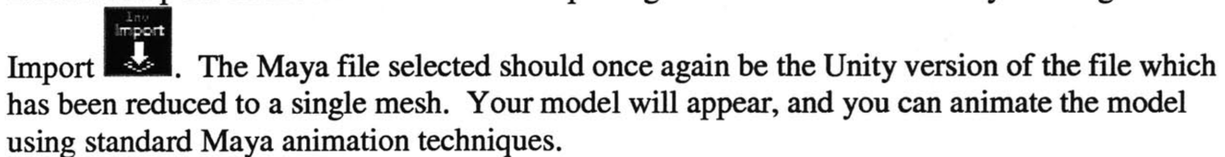
. This step is optional but quite useful. You may do this at any point, though I'd recommend doing it after building. You will be prompted to select the Maya file that generated the lattice. The one you select should be the one saved for Unity outlined above, the one that has been reduced to a single mesh. The preview mesh will generate inside the lattice originally, but it might be easier to work with if the mesh is translated off to the side, that is up to the user. The mesh will interactively update as the lattice is deformed.

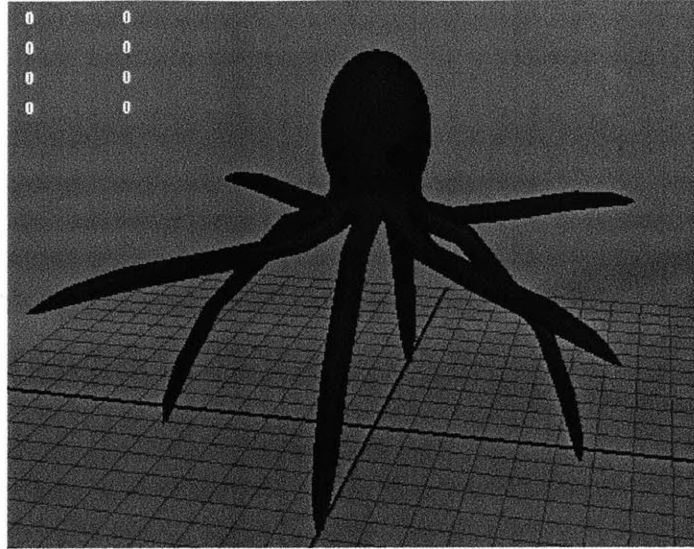


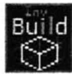
Once all of the keyframes have been set and the animation is finished, you will want to export the animation by clicking Export . This will prompt you to choose a destination file and an animation frame rate. You should export somewhere into your Unity project's Assets folder, and the file exported should have a .txt extension. Click "Export" when you're done.

To edit this animation in the future, simply reopen the Maya file in which this animation was created. Note that if the particle lattice changes in Unity, you WILL need to recreate the animation if you use forward animation. If this is a concern, inverse animation may be a better choice.

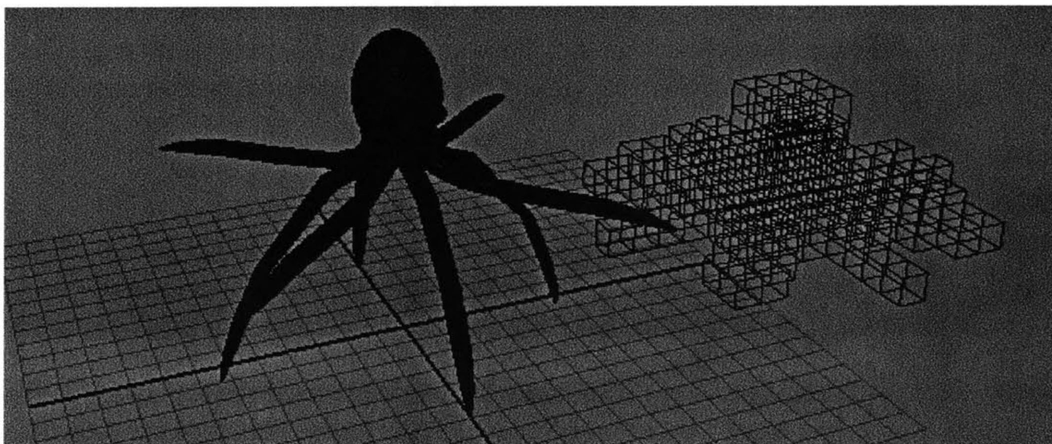
Inverse Animations

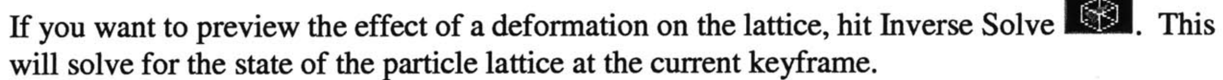
If you specify an inverse animation, you will animate the mesh directly, and then the soft body data, that is the state of the particle lattice, will be reconstructed from the mesh deformations. The first step for an inverse animation is importing the mesh. This is done by clicking Inverse Import . The Maya file selected should once again be the Unity version of the file which has been reduced to a single mesh. Your model will appear, and you can animate the model using standard Maya animation techniques.



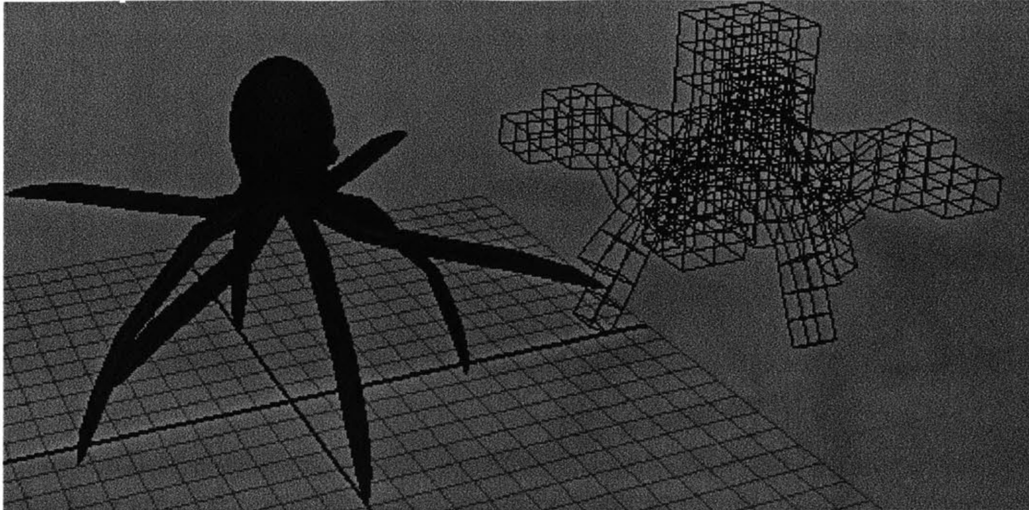
This next step may be done at any point before exporting, but as early as right after you import the model. You must specify the target particle lattice. This is done by clicking Inverse Build 

. This will generate the particle lattice. It will be in the lattice's rest pose initially. Similar to the forward solving case, you can drag the lattice off to the side so that it's easier to distinguish the two objects. If you ever want to change the particle lattice, for instance, perhaps you changed the spacing of the lattice in Unity, then you can delete the lattice object and hit Inverse Build again, selecting the new lattice.

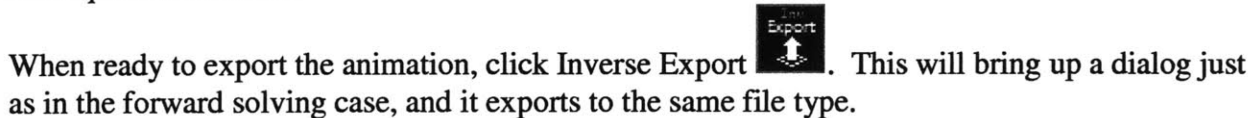




If you want to preview the effect of a deformation on the lattice, hit Inverse Solve. This will solve for the state of the particle lattice at the current keyframe.

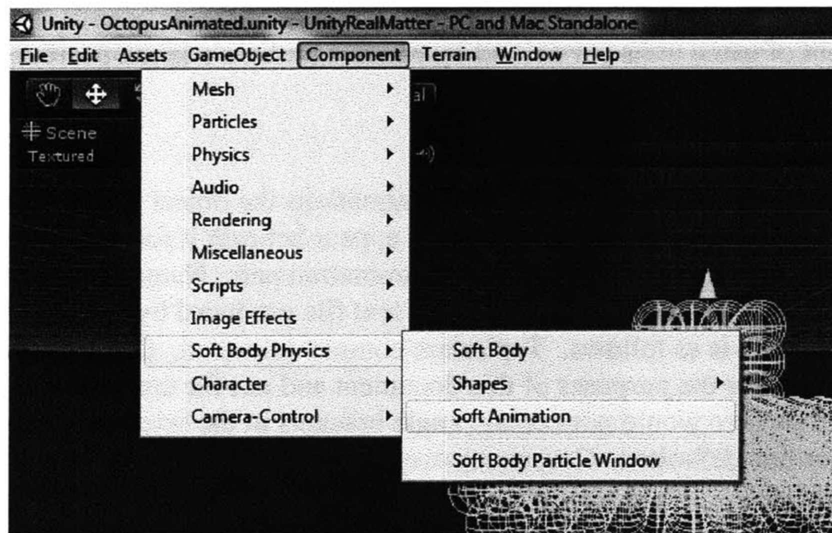


This operation will take longer the first time you do it in a scene, but it will speed up for subsequent executions.

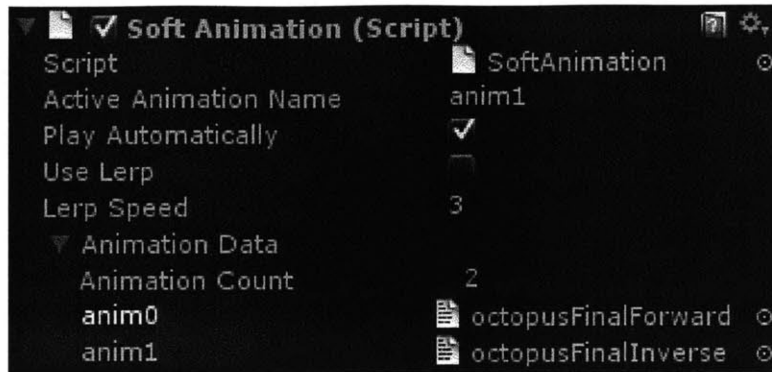
When ready to export the animation, click Inverse Export . This will bring up a dialog just as in the forward solving case, and it exports to the same file type.

Animation Simulation

In order to play back an animation, you will need to have a soft body to which you will add the Soft Animation component. This can be added from the main menu, Component>Soft Body Physics>Soft Animation.



The soft body will then have a Soft Animation component attached to it, which can be viewed in the inspector pane when the soft body is selected.



Active Animation Name

This string specifies the name of the animation which is currently playing. If the game is not running but we are in editor view, then this name corresponds to the animation which will be played if Play Automatically is checked.

Play Automatically

This Boolean specifies whether or not the object should be animating when the game starts. If this is checked, then the animation named Active Animation Name will be played.

Use Lerp

This Boolean specifies whether or not the tracked particle lattice should have a cap on target particle velocities. This is false by default, but may be useful for smoothing animation transitions.

Lerp Speed

The amount of game units per second by which a particle's target position is allowed to change when Use Lerp is active.

Animation Data

Animation count specifies the number of animations the object should have attached to it. Setting this value dictates how many fields appear beneath it for animation entry. The fields which appear beneath it are a name:animation pair. Names may be set to anything the user wants. The animation should be a text file produced by the Maya tools. The format of the file is as follows. Tuples are comma delimited. Comments here, after '//' characters, are for the purposes of this document and not for the file itself. Strings within angle brackets here would not contain angle brackets in the actual file. Each of these lines is new line delimited. Comments may be present on their own lines, preceded by a # character.:

particle //Specifies the animation type – for these purposes, it should always be particle

fps <FPS count float> //Specifies the animation speed

interpolate <1 or 0 boolean> //Specifies if the tracked animation should interpolate between keyframes. Always true with the Maya tools.

declaration //For readability, specifies that the next set of tuples declares the particle lattice

<tuple0> <tuple1> <...tuplen> //Specifies integer tuples, each specifying an index into the particle lattice.

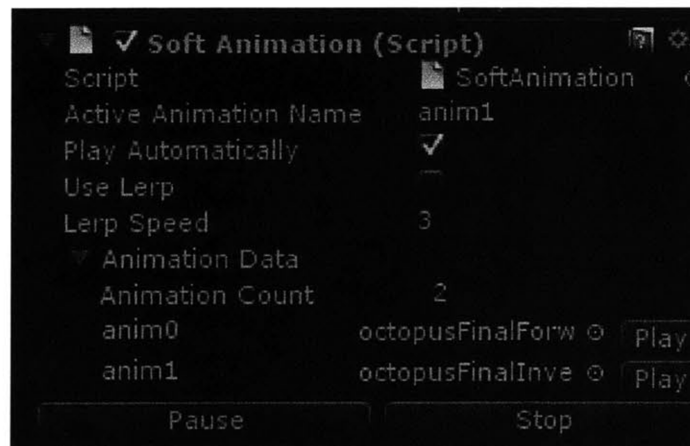
frames //For readability, specifies that the next set of tuples declares the keyframes

<frame number int> <tuple0> <tuple1> <...tuplen> //Specifies the frame number, followed by the position of each particle at that frame. Particle order should be identical to the line following 'declaration'

An example of a simple animation file can be seen below.

```
particle
#Default framerate of the animation
fps .5
#interpolate between frames?
interpolate 1
#Specify the particle indices and their order
declaration
0,0,0 0,0,1 0,1,0 0,1,1 1,0,0 1,0,1 1,1,0 1,1,1
#Now specify animation frames
frames
0 -.5,-.5,-.5 -.5,-.5,.5 -.5,.5,-.5 -.5,.5,.5 .5,-.5,-.5 .5,-.5,.5 .5,.5,-.5 .5,.5,.5
1 -1,-1,-1 -1,-1,1 -1,1,-1 -1,1,1 1,-1,-1 1,-1,1 1,1,-1 1,1,1
2 -1,-1,-1 -1,-1,1 -.5,1,-.5 -.5,1,.5 1,-1,-1 1,-1,1 .5,1,-.5 .5,1,.5
```

The interface within the inspector changes slightly when the game is running.



Each animation has a "Play" button next to it. This will make the soft body begin playing the corresponding animation immediately. "Pause" pauses the current animation, but changes to "Resume" so that the animation can be resumed where it left off. "Stop" freezes the current animation in place, and resuming the animation will bring it back to its beginning.

THIS PAGE INTENTIONALLY LEFT BLANK

References

[1] A. Nealen, M. Müller, R. Keiser, E. Boxerman, M. Carlson. Physically Based Deformable Models in Computer Graphics, *Computer Graphics Forum*, Vol. 25, Issue 4, pp 809-836. December 2006.

[2] S. F. Gibson., B. Mirtitch. A Survey of Deformable Models in Computer Graphics. Technical Report TR-97-19, Mitsubishi Electric Research Laboratories, Cambridge, MA, November 1997.

[3] Terzopoulos, D., Platt, J., Barr, A., Fleischer, K. Elastically Deformable Models. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, pp 205-214, 1987.

[4] J.P Lewis, M. Cordner, N. Fong. Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, pp 165-172, July 2000.

[5] Hahn, James K. Realistic Animation of Rigid Bodies. *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, pp 299-308, 1988.

[6] Isaacs, Paul M., Cohen, Michael F. Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions, and Inverse Dynamics. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, pp 215-224, 1987.

[7] Autodesk Maya 2011 Online Help. Autodesk, Inc., 2010, <http://download.autodesk.com/us/maya/2011help/index.html>.

[8] NVIDIA PhysX SDK 2.8 Documentation. NVIDIA Corporation, Santa Clara, CA, 2008

[9] A. Rivers, D. James. FastLSM: Fast Lattice Shape Matching for Robust Real-Time Deformation. ACM SIGGRAPH 2007 Papers, San Diego, CA, August 2007.

[10] Lasseter, J. Principles of Traditional Animation Applied to 3D Computer Animation. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, pp 35-44. 1987.

[11] M. Müller, M. Teschner, M. Gross. Physically-Based Simulation of Objects Represented by Surface Meshes. *Proceedings of Computer Graphics International (CGI)*, Crete, Greece, pp 26-33. June 2004.

[12] ElGindy, H., Everett, H., and Toussaint, G. T., (1993) Slicing an Ear Using Prune-and-Search. *Pattern Recognition Letters*, Volume 14, Issue 9, pp 719–722. 1993.

[13] Welman, C. Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation. PhD Thesis, Simon Fraser University. 1993.

[14] T. Sederberg, S. Parry. Free-Form Deformation of Solid Geometric Models. *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, pp 151-160, August 1986.

[15] Müller, M., Teschner, M., Gross, M. Meshless Deformations Based On Shape Matching. *ACM Transactions on Graphics,* Volume 24, Issue 3, New York, NY, pp 471-478. July 2005.