Meshless Deformation for Real-Time Soft Tissue Simulation

Alex Henriques*
Graphics Group, Department of Computer Science
University of Auckland, Auckland, New Zealand

Supervised by

Burkhard Wuensche

Contents

1 Introduction

Soft tissue refers to soft, flexible parts of the body such as muscles, tendons, organs, and skin. In medical applications, realistic real-time modeling of soft tissue is an important and difficult challenge. The most frequently used method for realistic simulation is finite element methods (FEM), but FEM is seldom fast enough for real-time simulation. When speed is more important mass-spring systems are most frequently used, but at the cost of significant accuracy.

In this report we apply a new method to soft tissue modeling, recently developed by Müller et al. [Müller et al. 2005]: "Meshless Deformations Based on Shape Matching." Meshless deformation is an efficient, easy to use, unconditionally stable method of dynamically simulating deformable objects. It allows fewer degrees of freedom than mass-spring systems, but is potentially significantly more efficient. Thus meshless deformation would be most suited to soft tissue simulations where speed and visual plausibility are important, but accurate physical modeling of complex biological structures is not.

In section 2, we give a quick overview of the traditional methods of soft tissue simulation, and discuss their advantages and disadvantages. Section 3 introduces the meshless deformation technique in more detail, and section 4 discuss our improvements to the technique. In section 5 we present our analysis of the soft tissue simulation capabilities of meshless deformation. Section 6 describes the interaction techniques available in the application we have developed. Finally, section 7 summarizes our results, and section 8 concludes.

2 Soft Tissue Simulation

Most methods of soft tissue simulation can be broadly categorized into one of three types: mass-spring systems, finite element methods, and finite difference methods.

Mass-spring systems model the object as a set of particles connected by springs. Each particle can be connected by spring to many other particles, so it is often difficult to decide and specify which particles are connected to which, and with what stiffness. Mass-spring systems are generally quite efficient and are suitable for real-time applications, but are not as physically accurate as finite element methods. Further, stability can often only be guaranteed with small timesteps or heavy damping constants, and even then springs with a high stiffness coefficient pose problems [Meyer et al. 2001]. Some other applications of mass-spring systems include facial animation and cloth modeling [Noh and Neumann 1998].

The finite element method (FEM) involves the discretization of a mesh into simpler components, or *finite elements*. For each element type, behaviour is characterized by the loads and responses at discrete nodes. The solution at each element is then extended over the entire object by interpolating element nodal values [Felippa 2001]. FEMs are generally very accurate and are used when physical accuracy is important, e.g. in mechanical and aeronautical engineering simulations. They are not very fast however, and it is difficult to model complex scenes with FEM in real-time.

The finite difference method (FDM) is related to FEMs, but approximates the differential equation itself rather than the solution to the differential equation [Wikipedia 2006]. FDMs are generally used for fluid dynamics, while FEMs are more broadly used for all types of structural mechanics analysis.

3 Meshless Deformation

"Meshless Deformations Based on Shape Matching" is a recently developed technique for dynamically simulating deformable objects [Müller et al. 2005]. The underling model is geometrically–as opposed to physically–motivated. It is unconditionally stable, does not require any pre-processing, and is simple to compute.

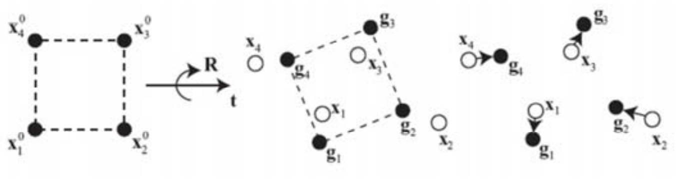


Figure 1: First, the original shape $\mathbf{x}_i^0$ is matched to the deformed shape $\mathbf{x}_i$. Then, the deformed points $\mathbf{x}_i$ are pulled towards the matched shape $\mathbf{g}_i$ (adapted from [Müller et al. 2005]).

3.1 The Technique

Meshless deformation treats each object as a *point cloud*, or set of points, with no connectivity information required. To understand the basic idea, let the initial configuration of points be $\mathbf{x}_i^0$, and the deformed configuration of points at some later time be $\mathbf{x}_i$. As a set of unconnected particles, each $\mathbf{x}_i$ responds to gravity and collisions, but no force acts to retain the overall object's shape. Meshless deformation's solution is to take the initial configuration $\mathbf{x}_i^0$, then move and rotate it as closely as possible onto the actual configuration $\mathbf{x}_i$ (see figure 1). The rotated version of the initial configuration is now the set of *goal positions* $\mathbf{g}_i$ which minimize the least squares distance to actual positions. Each particle is pulled towards its goal position after each timestep, retaining the object's initial shape.

The fundamental equation that finds the optimal transformation from $\mathbf{x}_i^0$ to $\mathbf{g}_i$ is that of "absolute orientation": given coordinates of a set of points as measured in two different Cartesian coordinate systems, find the optimal transformation between them [Horn 1987]. To find this optimal transformation, the following sum is minimized.

$$\sum_i w_i \left(\mathbf{R}(\mathbf{x}_i^0 - \mathbf{t}_0) + \mathbf{t} - \mathbf{x}_i\right)^2$$

where $\mathbf{R}$ is a pure rotation matrix. In meshless deformation, $\mathbf{t}_0$ is the centre of mass of the initial configuration, and $\mathbf{t}$ is the centre of mass of the actual configuration. Müller et al. extend this equation by adding linear and quadratic matching; $\mathbf{R}$ is replaced by a linear deformation matrix $\mathbf{A}$, or a quadratic deformation matrix $\widetilde{\mathbf{A}}$. Thus, the goal positions can be not only a rotated version of the initial configuration, but a stretched, sheared, bent and twisted version. To produce a tendency towards the original undeformed state in linear and quadratic matching, $\mathbf{R}$ is combined with $\mathbf{A}$ or $\widetilde{\mathbf{A}}$ to produce a final deformation matrix $\mathbf{F}$.

$$\mathbf{F} = \beta\widetilde{\mathbf{A}} + (1-\beta)\,\mathbf{R}$$

where β is a user defined constant between 0 and 1. Low β indicates a tendency mostly towards the rigid matched state, while high β indicates a tendency towards the quadratic match. The last important constant is α, which defines how quickly each point moves to its goal position. When $\alpha = 1$, each point moves precisely to its goal position every timestep.

In summary, meshless deformation effectively transforms the original object by a matrix representing stretch, shear, bend and twist to find the closest match to the deformed object, then pulls the deformed object towards the goal positions represented by the transformed object.

3.2 Clusters

The primary disadvantage of meshless deformation is simple. Goal positions are calculated by transforming the object with at best a quadratic deformation matrix, so goal positions can only ever be a quadratically deformed version of the initial object. Figure 2 visualizes all 27 possible deformation modes. Physical expressiveness may seem high, but significant limitations become apparent with objects more complicated than cubes and beach balls. These limitations are with respect to one of two things: higher order deformation and local deformation. Consider two common objects as examples. A slithering snake might have two bends in it, which requires at least cubic deformations during animation, so it cannot be deformed with global quadratic equations. As a second example consider a sweatshirt with a hood. When using global deformations raising or lowering the hood is impossible to perform without bending the entire object.

To extend meshless deformation for local and higher order deformation, Müller et al. divide the set of particles into overlapping clusters, each with its own deformation modes and matrix. An entity consisting of multiple interacting clusters has a much greater range of deformation than an entity consisting of only one cluster. This is not a complete solution however, as we discuss next.

3.3 Evaluation

The advantages of meshless deformation are clear: it is fast, and very easy to set up and tweak. The primary disadvantage of meshless deformation is that modes of deformation are quite limited. Clustering increases freedom, but is generally only well suited to objects with a small number of subparts, each of which deform at most quadratically. The only way to model more complex objects like cloth is to divide them into many fine grained clusters. But this is extremely inefficient and not very accurate – methods like mass-spring systems would be more suitable.

4 Improvements

4.1 Surface Area Preservation

Meshless deformation matches a goal configuration to the deformed point cloud as closely as possible. However, the goal configuration matched frequently has greater or lesser volume than the original object, which is generally undesirable. To preserve volume, meshless deformation scales the deformation matrix such that the goal configuration's volume is identical to the original object's volume. The problem with such blind scaling is that when for example a force squashes the object along one dimension, the volume of the goal configuration pre-scaling can be very small. To preserve volume the scaling factor must be very large to compensate, and the other two dimensions are scaled up drastically in response. The situation is depicted in figure 3.

4.1.1 Possible Solutions

If an airtight balloon filled with water were thrown gently at a wall, the volume of water inside would remain constant. But the balloon would not behave as in figure 3, because of resistance to *surface area* stretch. Clearly then, a method to constrain surface area is needed. Some algorithms use mesh-based explicit surface area preserving forces [Teschner et al. 2004a]. For meshless deformation, possible solutions include the following:

1. Limit forces applied to objects. If the vertices are not subject to large forces, they will not move so far out of their original configuration that blind volume-preservation scaling will produce such extreme surface area changes.

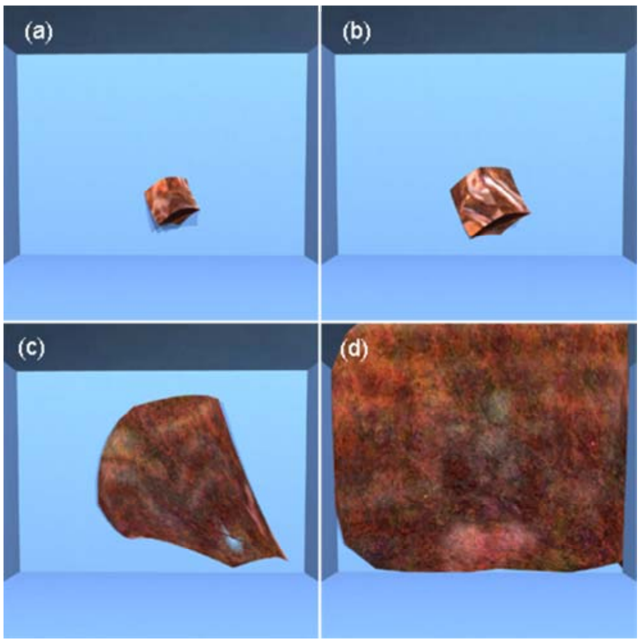


Figure 3: Volume preservation without surface area preservation results in unrealistic surface expansion as a cube is pushed into a wall.

2. Limit the maximum velocities of vertices. As with 1, if the vertex velocities are constrained to within a maximum, extreme configurations will be more difficult to produce.

3. Limit α and β. If α is large, the vertices will return quickly to their goal positions, lessening the likelihood of extreme configurations being produced. If β is small, the tendency of the cube to return to an undeformed state will override the quadratic transformation if it matches an extreme configuration.

4. Have vertices propagate a constraint force through to adjacent vertices.

5. Limit the transformation matrix somehow so that it doesn't match extreme configurations.

1, 2, and 3 used in various combinations are quite successful in combating this problem. 4 is an interesting option that could be experimented with. But these methods require tightly regulated parameters, so by definition cannot be unconditionally stable. 5 looks like the best option.

The simplest way to constrain surface area using 5 is to cap the Frobenius norm of the linear deformation matrix $\mathbf{A}$.

$$||\mathbf{A}||_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2$$

Drastic increases in surface area are caused by large stretch or shear values, which are the contributors to $||\mathbf{A}||_F^2$. Therefore, by limiting $||\mathbf{A}||_F^2$, we limit stretch and shear. If we want to cap the amount of quadratic deformation for visual reasons, the following methods can also be trivially extended from $\mathbf{A}$ to $\widetilde{\mathbf{A}}$. In the subsequent computations, we use the term $||\mathbf{A}||$ as shorthand for the Frobenius norm.

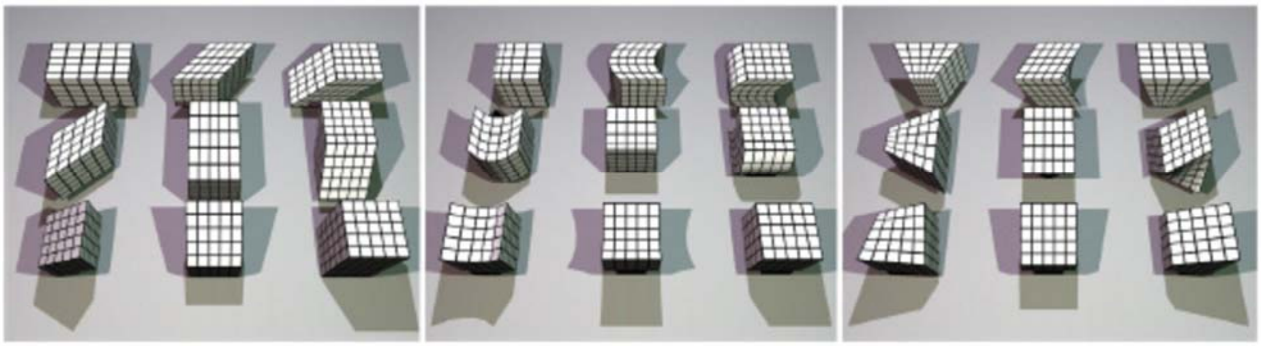


Figure 2: Visualization of all 3×9 deformation modes (adapted from [Müller et al. 2005]).

4.1.2 Methods of clamping

There are several ways to clamp $||\mathbf{A}||$; here are three.

1. Let rows of $\mathbf{A}$ be termed $\mathbf{r}_i$. If any $||\mathbf{r}_i||^2$ exceeds a user selected c_{max}, scale $\mathbf{r}_i$ by x such that $||x\mathbf{r}_i||^2 = c_{max}$.

2. Cap the magnitude of $\mathbf{A}$ at c_{max}. To do this, if $||\mathbf{A}||^2 > c_{max}$ update $\mathbf{A}$ as
$$\mathbf{A} \leftarrow \gamma\mathbf{A} + (1-\gamma)\mathbf{R}$$
where γ is derived from the quadratic equation
$$||\gamma\mathbf{A} + (1-\gamma)\mathbf{R}||^2 = c_{max}.$$
The final matrix $\mathbf{F}$ is then calculated as:
$$\begin{aligned}\mathbf{F} &= (\gamma\mathbf{A} + (1-\gamma)\mathbf{R})\beta + \mathbf{R}(1-\beta)\\ &= \gamma\beta\mathbf{A} + \beta\mathbf{R} - \gamma\beta\mathbf{R} + \mathbf{R} - \beta\mathbf{R}\\ &= \gamma\beta\mathbf{A} + (1-\gamma\beta)\mathbf{R}.\end{aligned}$$
So γ ends up being applied as a simple beta modifier.

3. As a cheaper imitation of 2, simply set
$$\gamma = \frac{c_{max}}{||\mathbf{A}||^2}.$$

The first method works well, but restricts deformation along each axis regardless of deformation in the other axes. The second and third methods on the other hand restrict the sum of deformations along all axes, so maximum deformation along one axis prevents further deformation along the other axes. The appropriate method would seem to depend on the physical properties of the object. Visually we could not distinguish between methods 2 and 3.

4.1.3 Further Extensions

These three methods solve the blow-up problem well, but introduce a slight problem with visual plausibility. A soft object falling to the ground will flatten to the point where the deformation magnitude ϕ is capped, then deformation will jerk to a stop. To solve this we suggest a "soft" cap rather than a hard one. This would take the form of a monotonically increasing function f such that for an intermediate threshold c and a maximum threshold m,

$$f(\phi) = \begin{cases} \phi & \text{if } \phi \le c \\ < m & \text{if } \phi > c \end{cases}$$

In other words, an object with deformation magnitude ϕ exceeding the soft cap c will have ϕ reduced towards c. To prevent unrealistically large deformations, m indicates a hard cap below which ϕ will always be reduced. Here is an example function:

$$f(\phi) = \begin{cases} \phi & \text{if } \phi \le c \\ m - \left(\frac{c}{\phi}\right)(m-c) & \text{if } \phi > c \end{cases}$$

4.2 Inversion

Recall that the central equation to be minimized in meshless deformation is

$$\sum_i w_i \left(\mathbf{R}(\mathbf{x}_i^0 - \mathbf{t}_0) + \mathbf{t} - \mathbf{x}_i\right)^2$$

Müller et al. present the most referenced solution to this problem (referred to as that of absolute orientation) derived in [Horn 1987]. In his paper however Horn mentions that the $\mathbf{R}$ obtained may be a reflection, rather than a rotation, in cases where reflection provides a better fit.

In traditional photogrammetric applications of the absolute orientation problem, the data may seldom be corrupted enough to produce a reflective $\mathbf{R}$. When applied to physical objects undergoing large deformations however, the vertices can frequently be deformed enough that the optimal $\mathbf{R}$ is a reflection. The inverted object produced is an unacceptable result for homogeneous objects, because it would require massive self-penetration.

4.2.1 Determinant Cube Root Solution

Müller et al. do not specifically mention what to do when a reflective $\mathbf{R}$ is produced. The only related comment is made when discussing volume preservation of the linear transformation matrix $\mathbf{A}$:

> To make sure that volume is conserved, we divide $\mathbf{A}$ by $\sqrt[3]{det(\mathbf{A})}$ ensuring that $det(\mathbf{A}) = 1$.

When $det(\mathbf{A})$ is negative, $\sqrt[3]{det(\mathbf{A})}$ is also negative. The subsequent division results in an $\mathbf{A}$ that produces a non-inverted, volume preserving goal position configuration. This configuration is obtained however by a simple reflection of each optimal position through the origin. $\mathbf{A}$ no longer describes a minimization of goal position with respect to vertex position. Thus the goal positions will tend to be far away from their respective vertex positions, and the integration step will produce large velocities. The result is a blowup.

When taken literally, the method deals with an inverted goal match by producing a blowup. If $\sqrt[3]{det(\mathbf{A})}$ is constrained to its absolute value, the method results in a stable, inverted object configuration. Neither result is acceptable.

4.2.2 Modified R Extraction Solution

Rather than make a modification to the transformation matrix after $\mathbf{R}$ has been calculated, a modified algorithm is proposed by Umeyama [Umeyama 1991] that strictly produces an optimal rotation matrix $\mathbf{R}$. Implementing this modification involves only a simple addition to the singular value decomposition solution method of Arun et al. [Arun et al. 1987].

This method solves the inversion problem, but only partially. While $\mathbf{R}$ will always be a rotation, $\mathbf{A}$ may still contain a reflection (assuming the absolute value of $\sqrt[3]{det(\mathbf{A})}$ is used). The final transformation matrix $\mathbf{F} = \beta\mathbf{A} + (1-\beta)\mathbf{R}$ then will always have a tendency towards a non-inverted configuration. But with β close to 1, the tendency will be slow, and may produce physically implausible results. Ideally $\mathbf{A}$ would calculated in a manner that never produced reflections, however it is unclear how this would be done.

4.3 Constraining Vertices

If a box is stuck to the floor, or a beam is attached to the wall, or two objects are hinged at a joint, we face the problem of constraining the involved vertices. In most physically based models this is easy to do: simply hold constant the positions of the constrained vertices. In this section we a) explain why this method often does not produce satisfactory results, and b) explore alternative methods.

4.3.1 Holding Positions Constant

In mass-spring and finite element systems, displacing a vertex causes local effects that then propagate outwards to effect the deformation of the entire object. Thus, if the positions of some vertices are not updated, a local effect will propagate outwards in a continuous manner, hopefully producing realistic results. In meshless deformation however, if we do not update vertex positions, the local effect does not propagate outwards realistically (see figure 4).

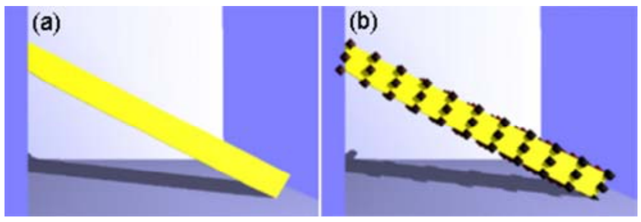


Figure 4: A beam is attached to the left wall. The points in (b) represent the goal positions.

While the constrained vertices stay in place, the overall shape of the object does not take this into account. Instead the goal positions are simply calculated as a rotated version of the beam, and the constrained vertices, by virtue of their static positions, are at a sudden discontinuous displacement from their goal positions.

It seems that in order to produce a continuous deformation, we would like the goal positions of the constrained vertices to conform to the actual vertex positions. The next two sections discuss ways of doing this.

4.3.2 Constrained Vertices of Infinite Mass

Goal positions are calculated to minimize the mass-weighted least squares distance between themselves and their respective vertices. If we give the constrained vertices infinite mass, then the least squares minimization will produce goal positions at zero distance from their respective constrained vertices. This will produce a constrained, yet continuous deformation. Our tests used a beam attached to a wall, parallel to the floor (see figure 5).

The problems faced by this solution are immediately apparent. Under normal conditions (figure 5 (a)), the beam simply remains rigid and does not deform. With gravity five times normal and an extreme $\beta = 0.99$, the beam droops down with some realism (see figure 5 (b)), however the extreme conditions mean this result is of limited use. Further, the beam's response to interactive spring forces is not physically plausible (see figure 5 (c)-(d)). When we pull a bar up by its corner we expect it to largely retain its cross-sectional shape. Our tests instead show the beam's cross section flatten out in a strange manner.

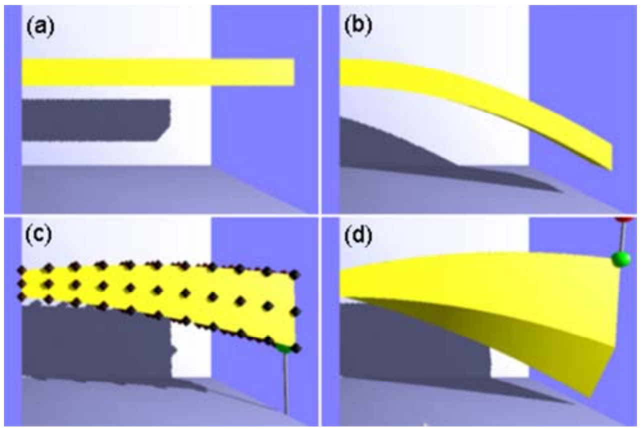


Figure 5: (a) Beam with constrained vertices of infinite mass, under normal conditions of $\beta = 0.50$, $g = -10m.s^{-2}$. (b) As a, but with extreme conditions $\beta = 0.99$, $g = -50m.s^{-2}$. (c) An edge of the beam is pulled down under normal conditions. (d) A corner of the beam is pulled up under extreme conditions.

Why the disappointing results? Recall the goal positions are calculated using the matrix

$$\mathbf{F} = \beta\widetilde{\mathbf{A}} + (1-\beta)\widetilde{\mathbf{R}}.$$

If we fix at least three linearly independent coplanar vertices, the only possible rotation $\widetilde{\mathbf{R}}$ that will match is rotation by 0, i.e. the identity matrix, leaving the object in its original position. (It is easy to see that any other rotation will move at least one of these vertices from its original position). Thus $\widetilde{\mathbf{R}}$ will give $\mathbf{F}$ the tendency to keep the entire object rigid and in its original position.

With $\beta = 0.50$, this tendency is strong enough that the entire bar remains immobile. We might expect the bar to form a halfway-position, where the $\widetilde{\mathbf{A}}$ component has a large quadratic bend, and the $\widetilde{\mathbf{R}}$ component keeps immobility. In reality, what happens though is if $\widetilde{\mathbf{A}}$ matches a particular quadratic bend, the combination $\mathbf{F}$ will be somewhere between $\widetilde{\mathbf{A}}$'s match and identity. Next timestep, $\widetilde{\mathbf{A}}$ (as a least squares minimization) will match the previous combination $\mathbf{F}$, effectively halving the deformation matched by $\widetilde{\mathbf{A}}$ in the previous timestep. In this inverse exponential manner, the tendency will always be strongly towards the original configuration. Only with the extreme conditions in figure 5 (b) can this inverse exponential tendency be overcome, and a more realistic deformation produced.

We can also examine the behaviour produced by subjecting the object to spring forces. Suppose we have the situation in figure 5 (c). The goal positions here are stable, so the system is in equilibrium with its deformation defined by an unchanging $\mathbf{F}$. As only one vertex is being pulled, we know that $\mathbf{F}$ must be the combination of $\widetilde{\mathbf{R}}$ and the quadratic $\widetilde{\mathbf{A}}$ matched to an $\mathbf{F}$ including the changed position of the sole vertex being pulled. This means to create the deformation in (c) a very strong spring force is needed; one that effectively doubles the stable deformation by its displacement of a single vertex.

The expected behaviour in figure 5 (c) might be that the top face of the beam followed the bottom face down, keeping a beam shape. Viewed from the perspective of the above analysis, we can see why this does not happen. Recall that in order to produce a stable configuration, the displacement of a single vertex needs to double the deformation of the stable configuration. Imagine the vertex in (c) being dragged down a long way. The bottom face may match to a continuous curve, bending down towards the displaced vertex. But why would the top face move? None of its vertices have, so a least squares minimization would create as close a match as it could to the unmoved top face (along with the moved bottom vertex). Thus we see the results in (c), where the top face moves little while the bottom face is dragged in a curve downwards.

What of the even more unusual deformation seen in (d)? As a combination of high gravity, high β and a spring force, it shows the nature of the limited modes of quadratic deformation available. Each edge is following the same curve defined in $\mathbf{F}$, different in end appearance only because of each edge's different x and z values.

4.3.3 Matrix Constraints

It makes sense at this point to look at the mathematical form $\mathbf{F}$ must take if it is to constrain a certain class of vertices. First, recall from section 3.1 that the final deformation matrix $\mathbf{F}$ is calculated as

$$\mathbf{F} = \beta\widetilde{\mathbf{A}} + (1-\beta)\widetilde{\mathbf{R}}$$

where $\widetilde{\mathbf{A}}$ is the 3×9 quadratic transformation matrix, and $\widetilde{\mathbf{R}}$ is the rotation transformation matrix $\mathbf{R}$ extended to 3×9 by filling columns 4 to 9 with zeros. Each goal position $\mathbf{g}_i$ is calculated as

$$\mathbf{g}_i = \mathbf{F}\widetilde{\mathbf{q}}_i + \mathbf{x}_{cm}$$

where $\mathbf{x}_{cm}$ is the object's centre of mass, $\mathbf{q} = \mathbf{x}_i^0 - \mathbf{x}_{cm}^0$ ($\mathbf{x}_i^0$ is the vertex's original position and $\mathbf{x}_{cm}^0$ is the object's original centre of mass), and

$$\widetilde{\mathbf{q}} = \begin{pmatrix} q_x & q_y & q_z & q_x^2 & q_y^2 & q_z^2 & q_xq_y & q_yq_z & q_xq_z \end{pmatrix}^T$$

Suppose then that we want to constrain all vertices in the plane of a particular y. For all such vertices, $\mathbf{g}_i = \mathbf{x}_i^0$ and the following constraints on $\mathbf{F}$ apply.

$$\begin{aligned}
\mathbf{x}_i^0 &= \mathbf{F}\widetilde{\mathbf{q}}_i + \mathbf{x}_{cm}\\
\mathbf{q}_i + \mathbf{x}_{cm}^0 &= \mathbf{F}\widetilde{\mathbf{q}}_i + \mathbf{x}_{cm}\\
\mathbf{F}\widetilde{\mathbf{q}}_i &= \mathbf{q}_i - \left(\mathbf{x}_{cm} - \mathbf{x}_{cm}^0\right)\\
\mathbf{F}\widetilde{\mathbf{q}}_i &= \mathbf{q}_i - \Delta cm
\end{aligned}$$

To find explicit constraints, we solve the above equation for several points on the given y plane. Let $\mathbf{F}$ be the 3x9 matrix

$$\mathbf{F} = \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} & \mathbf{g} & \mathbf{h} & \mathbf{i} \end{pmatrix}$$

For the first constraint, select

$$\mathbf{q}_i = \begin{pmatrix} 0\\ y\\ 0 \end{pmatrix}$$

for a constant y and

$$\begin{aligned}
\mathbf{F}\widetilde{\mathbf{q}}_i &= \mathbf{q}_i - \Delta cm\\
\mathbf{b}y + \mathbf{e}y^2 &= \mathbf{q}_i - \Delta cm\\
\mathbf{b} &= \frac{\widetilde{\mathbf{q}}_i - \Delta cm - \mathbf{e}y^2}{y}
\end{aligned}$$

For the second constraint, select

$$\mathbf{q}_i = \begin{pmatrix} x\\ y\\ 0 \end{pmatrix}$$

for some arbitrary x and

$$\begin{aligned}
\mathbf{F}\widetilde{\mathbf{q}}_i &= \mathbf{q}_i - \Delta cm\\
\mathbf{a}x + \mathbf{d}x^2 + \mathbf{g}xy + \mathbf{b}y + \mathbf{e}y^2 &= \mathbf{q}_i - \Delta cm\\
\mathbf{a}x + \mathbf{d}x^2 + \mathbf{g}xy &= \begin{pmatrix} x\\ 0\\ 0 \end{pmatrix}\\
\mathbf{a} + \mathbf{d}x + \mathbf{g}y &= \widehat{\mathbf{X}}\\
\mathbf{d} &= \mathbf{0}\\
\mathbf{a} &= \widehat{\mathbf{X}} - \mathbf{g}y
\end{aligned}$$

where $\widehat{\mathbf{X}}$ is the unit x vector. For the third constraint, select

$$\mathbf{q}_i = \begin{pmatrix} 0\\ y\\ z \end{pmatrix}$$

for some arbitrary z and

$$\begin{aligned}
\mathbf{F}\widetilde{\mathbf{q}}_i &= \mathbf{q}_i - \Delta cm\\
\mathbf{c}z + \mathbf{f}z^2 + \mathbf{h}yz + \mathbf{b}y + \mathbf{e}y^2 &= \mathbf{q}_i - \Delta cm\\
\mathbf{c}z + \mathbf{f}z^2 + \mathbf{h}yz &= \begin{pmatrix} 0\\ 0\\ z \end{pmatrix}\\
\mathbf{c} + \mathbf{f}z + \mathbf{h}y &= \widehat{\mathbf{Z}}\\
\mathbf{f} &= \mathbf{0}\\
\mathbf{c} &= \widehat{\mathbf{Z}} - \mathbf{h}y
\end{aligned}$$

where $\widehat{\mathbf{Z}}$ is the unit z vector. For the last constraint, select

$$\mathbf{q}_i = \begin{pmatrix} x\\ y\\ z \end{pmatrix}$$

with $x = z$ and

$$\begin{aligned}
\mathbf{F}\widetilde{\mathbf{q}}_i &= \mathbf{q}_i - \Delta cm\\
\mathbf{a}x + \mathbf{b}y + \mathbf{c}z + \mathbf{e}y^2 + \mathbf{g}xy + \mathbf{h}yz + \mathbf{i}xz &= \mathbf{q}_i - \Delta cm\\
\mathbf{a}x + \mathbf{c}z + \mathbf{g}xy + \mathbf{h}yz + \mathbf{i}xz &= \begin{pmatrix} x\\ 0\\ z \end{pmatrix}\\
\mathbf{a} + \mathbf{c} + \mathbf{g}y + \mathbf{h}y + \mathbf{i}x &= \begin{pmatrix} 1\\ 0\\ 1 \end{pmatrix}\\
\mathbf{i} &= \mathbf{0}
\end{aligned}$$

This defines the constrained form of F as

$$\mathbf{F} = \begin{pmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} & \mathbf{e} & \mathbf{f} & \mathbf{g} & \mathbf{h} & \mathbf{i} \end{pmatrix}$$

with:

$$\begin{aligned}
\mathbf{a} &= \widehat{\mathbf{X}} - \mathbf{g}y \\
\mathbf{b} &= \widehat{\mathbf{Y}} - \mathbf{e}y - \frac{\Delta cm}{y} \\
\mathbf{c} &= \widehat{\mathbf{Z}} - \mathbf{h}y \\
\mathbf{d} &= \mathbf{0} \\
\mathbf{e} &= free \\
\mathbf{f} &= \mathbf{0} \\
\mathbf{g} &= free \\
\mathbf{h} &= free \\
\mathbf{i} &= \mathbf{0}
\end{aligned}$$

This gives three independent vectors, three **0** vectors, and three dependent vectors. This is the form the desired **F** must take. How are the free variables to be decided though? The naïve approach (see figure 6) is to leave **e**, **g**, and **h** as they are from the normal calculation of **F**.

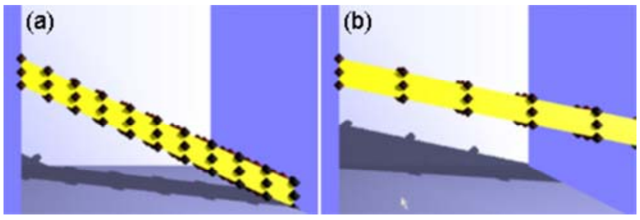


Figure 6: Beams attached to a wall with constrained deformation matrix.

Two problems present themselves. In figure 6 (a), we see the deformation is continuous, but having applied the matrix constraints to **F**, there is no longer any tendency to return to the original shape. Given a velocity, the beam's unconstrained vertices will float off to infinity, deforming further and further. In figure 6 (b), we see that **F** no longer conserves volume–the beam has stretched out from the wall with no corresponding reduction in cross-sectional area.

To introduce a tendency to return to the original shape, we note that if two deformation matrices preserve the correct constraints, then so will any convex combination of the matrices. If we set $\widetilde{\mathbf{R}}$ to identity, it will necessarily preserve the constraints. So instead of applying the above process to **F**, we can apply it to $\widetilde{\mathbf{A}}$ instead, and the constraints will be preserved. Unfortunately, this just produces the same results as in the previous section, "Giving Constrained Vertices Infinite Mass." We are left unsure how to obtain the desired results.

4.4 Specifying clusters

An entity's deformation behaviour of course depends on how it is divided into clusters. One method of division was suggested in [Müller et al. 2005]:

> We regularly subdivide the space around a given surface mesh into overlapping cubical regions. For each region, we generate one cluster with all the vertices contained in this region.

In our implementation of this method a cluster resolution can be specified, along with an overlap factor. For example, a cord could be given a cluster resolution $X = 1$, $Y = 5$, $Z = 1$ (where Y is the cord's length axis). This would give the cube 5 overlapping cuboid clusters along the Y axis. The degree to which the cuboids overlap can be set by a separate parameter.

When clusters are not regularly spaced, this is inadequate. A more powerful method would allow the user to specify the precise location and size of the cuboids at the modeling phase. (Any shape other than cuboids could conceivably be used, though a "triangle-soup" mesh would make cluster creation less efficient.)

When even greater control is required, arbitrary cuboid definition is not adequate either. For example, in trying to apply meshless deformation to facial animation we needed to divide a human face into clusters corresponding to facial muscle groups. Creating triangle-soup meshes to capture the precise 3-dimensional area of each cluster would be very difficult. Our final solution used the 3d modeler Maya, custom properties, and a custom exporter.

Each model in Maya consists of an arbitrary number of meshes. In our solution each mesh is either a visual mesh or a cluster mesh (or both). Visual meshes define the visible triangular mesh, while cluster meshes define which vertices belong to which cluster. In general to specify a cluster, the user duplicates the appropriate part of a visual mesh, then sets the duplicate mesh to be a cluster. With highly detailed meshes, selecting an appropriate part of the mesh to duplicate can be a good deal of work–often triangle by triangle selection. Nevertheless, for fine control over cluster division, it is unclear what other option there is. A paint or brush type tool has potential, but this did not seem to be immediately supported by Maya.

One problem with the method as described is that usual physical tetrahedralization based on the visual mesh does not work, because physical vertices have already been specified in the model. Tetrahedrons are necessary for collision detection, but it is not clear how to tetrahedralize from a physical surface cluster mesh that provides no volumetric information. The only general solution appears to be the creation of a thin tetrahedral "skin."

5 Analysis of Simulation Capabilities

We expect soft tissue to respond to environmental forces and user interaction with visual plausibility. The aim is efficiency, so the results need not be physically accurate. We carry out three tests:

1. Simulation of a simple jelly cube subjected to various user interactions. This is meshless deformation's strength, so we expect visually plausible results.

2. Simulation of a soft L-shaped bar. This tests the capabilities of clusters in a simple situation, and we expect results that are reasonably plausible.

3. Simulation of a soft tissue patch of skin. This tests the ability of meshless deformation to simulate complex and high order deformations.

The four basic modes of deformation possible are stretch, shear, bend and twist (see figure 2 for all 3×9 deformation modes). We tested the real-time rigid, linear, and quadratic deformation of a deformable cube subjected to user interaction (see figure 7). Visual plausibility was good; the cube behaved as one would expect a tough jelly cube to behave.

For the next test, we took an object with more complex deformation: a soft L-shaped bar. We would expect to be able to bend both sections of the bar together or apart, but with one cluster this is not possible (see figure 8 (a)-(b)). Also, when set to deform quadratically behaviour is not always plausible. The bar twists and bounces

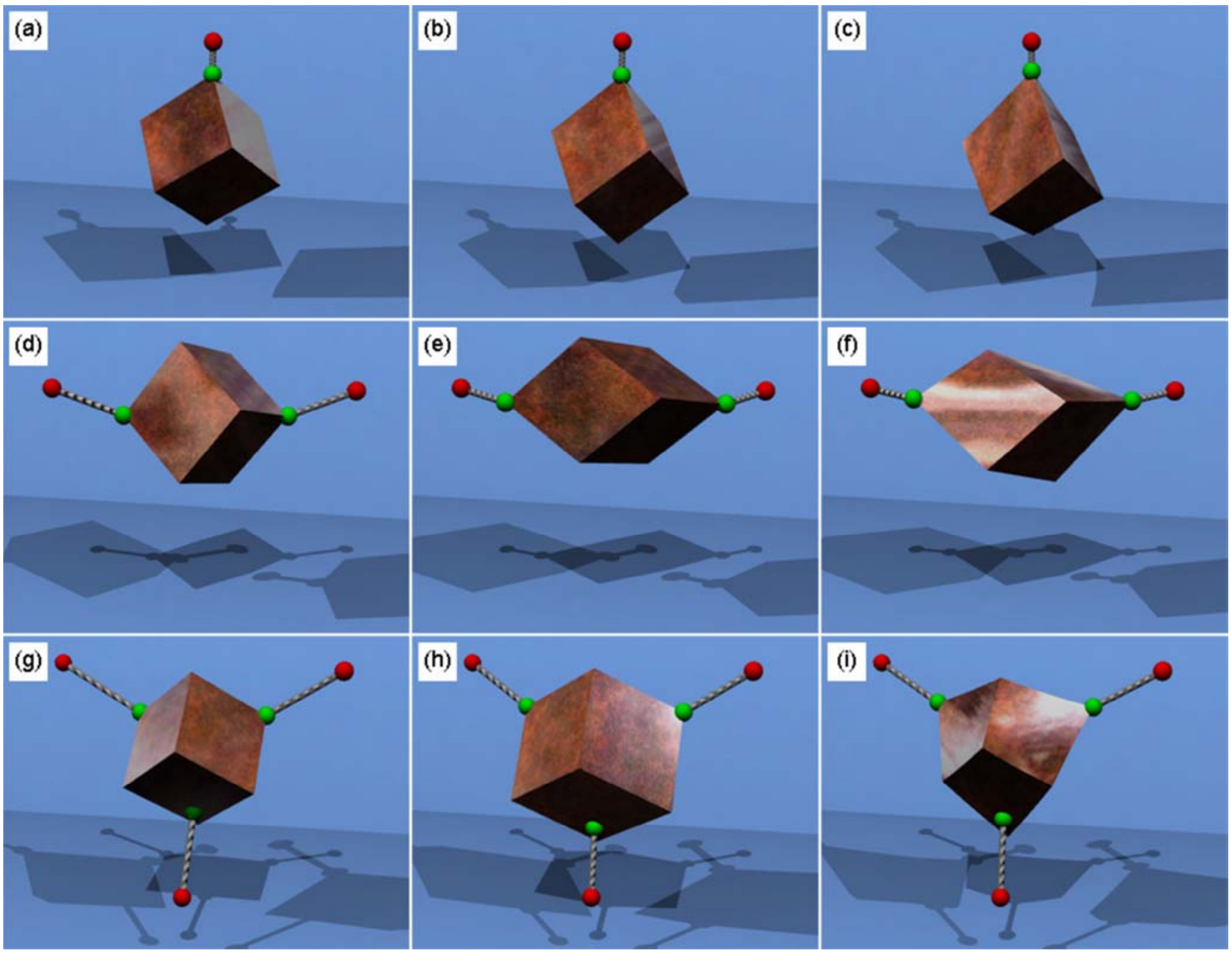


Figure 7: Rigid, linear and quadratic deformations of a cube undergoing user interaction.

strangely (see figure 8 (c)). Some of this can be mitigated by allowing quadratic bend, but not twist, matching. Generally however, bar-like behaviour is only achieved when deformations are very close to rigid, which of course precludes bending.

This should be an ideal time to use clustering; one cluster for each branch of the L-bar. As discussed in section 4.4, we specify two cuboid regions to enclose each cluster. Results are plausible (see figure 9), however where clusters meet at the corner of the bar, deformation is uneven when the bar is pulled straight (figure 9 (c)). Visual plausibility would probably improve with more vertices around the joint. Visual plausibility is good then, but not perfect.

Objects tested up to now have been simple, with few modes of deformation. For the next test, we see if we can simulate something more complex: a section of skin. We expect to be able to pick an arbitrary point on the skin and lift it up while the surrounding skin deforms plausibly. The bottom surface of the skin is attached to the ground. (Settings are $\alpha = 1.0$, $\beta = 0.6$. Note that α could be set below 1.0 to allow the picked point to be dragged away from the object easily, but without also dragging surrounding points, this would not help visual plausibility much.)

With a skin patch of one cluster, picking simply attempts to move and deform the entire skin patch, and not much happens (see figure 10 (a)). When the skin patch is divided into 2×2 clusters deformation is more plausible but still limited, with the dividing lines between clusters quite visible (see figure 10 (b)-(d)). Dividing the skin patch into 5×5 clusters significantly increases visual plausibility, and to quite a satisfactory level (See figure 11).

While our tests show that meshless deformation *can* be used to simulate skin in a visually plausible manner, by no means is it efficient. To simulate plausible pick response within for example a 5cm range of error, we need to divide the skin into clusters with length on the order of 5cm. Further, each cluster must have at minimum somewhere on the order of 20 vertices to deform plausibly. All this to produce behaviour less accurate than a mass-spring system with *one node* per 5cm. From an efficiency standpoint, it seems infeasible to use meshless deformation to simulate localized deformation. By nature, meshless deformation describes limited modes of deformation over a whole object, and thus localizing the deformation necessarily requires inefficiently creating many little objects (clusters).

To summarize, meshless deformation can plausibly simulate stretch, shear, bend and twist over a whole object, and with clustering, individual parts of an object. Localized deformation however requires clustering too fine grained to be efficient.

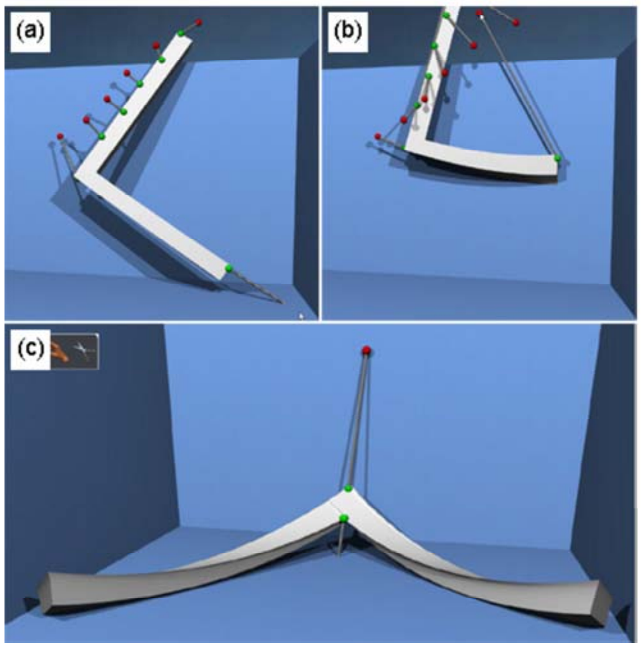


Figure 8: (a)-(b) show an attempt to bend the bar, while (c) shows some strange deformation behaviour as the bar is pulled apart.

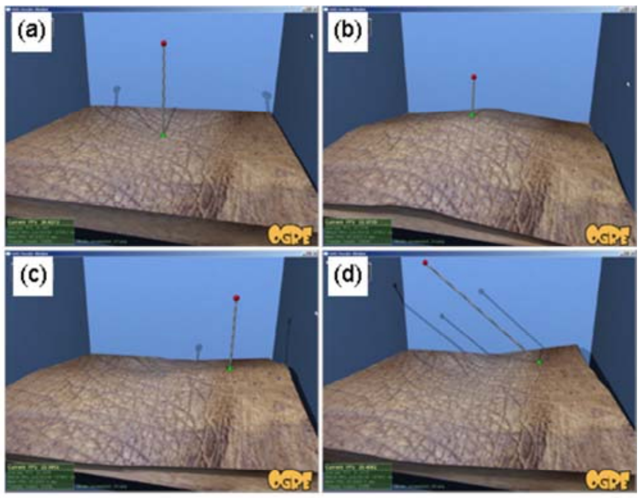


Figure 10: (a) pick on a single cluster skin patch, (b)-(d) picks on 2×2 cluster skin patch.

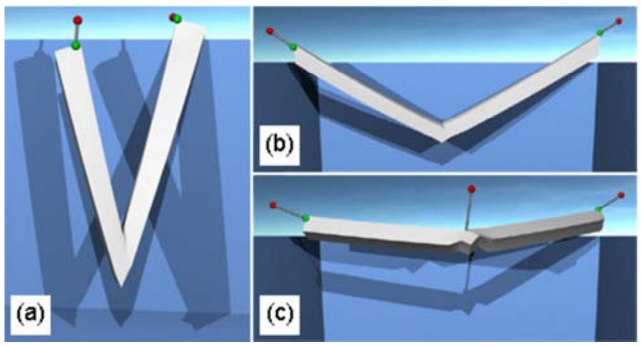


Figure 9: two clustered l-bar bending

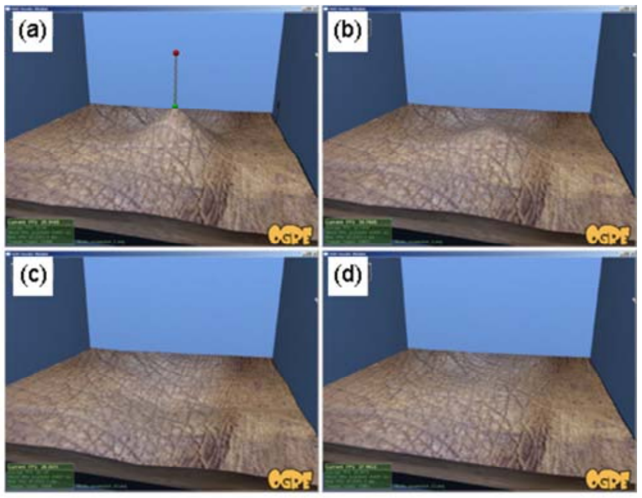


Figure 11: Behaviour of a 5×5 cluster skin patch in response to a user pick.

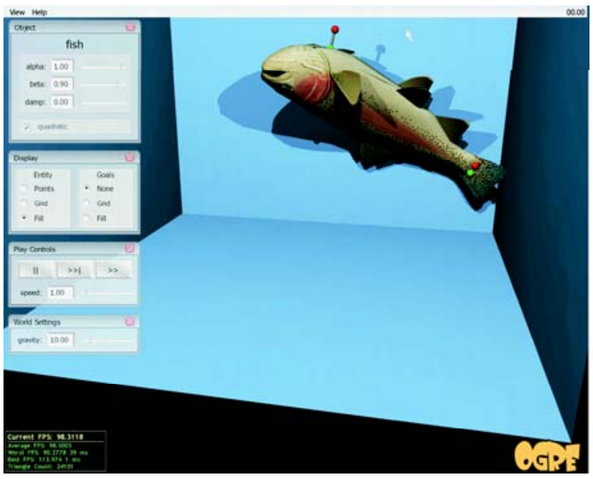
6 Interaction Techniques

Interaction techniques are an important part of creating an immersive and believable environment. In applications such as virtual surgery, advanced interaction techniques can even be a necessary part of the simulation. In this section we introduce techniques for picking, constraining, pushing and cutting objects simulated using meshless deformation based on shape matching.

6.1 Picking

The main function of the picking mode is to grab objects and manipulate them with a spring force. The user can press the left mouse button to grab an object vertex, then drag the mouse around to control the direction of the spring force acting on that vertex. The spring force acts towards the position of the red sphere, i.e. the cursor. When the user moves the mouse, the red sphere moves along a plane facing the user. The mousewheel can move the red sphere away from (mousewheel up) or towards (mousewheel down) the user. This moves the red sphere's plane of movement away from or towards the user, while keeping the plane's normal unchanged.

While dragging a spring force around, the user can release the left mouse button to stop the force and release the spring. Alternatively, the user can click the right mouse button to lock the force (i.e. the red sphere) in place. The user can then move around, change modes, or create a new spring force, while the original spring force remains in position. This makes it easy to "fix" an object in a deformed position (see figure 12 for an example).To remove a locked in spring force, the user can click and drag on the red sphere to regain control of it then release the left mouse button, or press a key to remove all spring forces from every object.



fish

alpha: 1.00

beta: 0.90

damp: 0.00

quadratic ✓

Entity: Fill; Goals: None

speed: 1.00

gravity: 10.00

```
Current FPS: 98.3118
Average FPS: 98.3003
Worst FPS: 90.2778 39 ms
Best FPS: 113.974 1 ms
Triangle Count: 24939
```

Figure 12: A deformed model of a trout fixed using two locked pick points.

6.2 Pushing

The main function of the pushing mode is to move objects by pushing them. A solid sphere follows the user's cursor in the same manner as the red sphere of the active spring force does in the picking mode above. Any objects colliding with the sphere undergo collision response forces. This is designed to mimic the user pushing objects around with his hand.

6.3 Cutting

The main function of the cutting mode is to cut objects into separate pieces. The cursor turns into two cylinders designed to mimic a cutting instrument, e.g. a pair of scissors. To cut an object, the user moves the "scissors" to the appropriate position relative to the object, then holds down the left mouse button to begin the cutting process. The two "blades" of the scissors move closer together, and when they meet, every object the scissors intersect is severed along the plane of the scissors, creating two new separate objects.

To change the orientation of the scissors, the user can move the scissors towards him with mousewheel down, away from him with mousewheel up, or he can rotate the scissors about the y axis by holding down shift and dragging the left mouse button up or down.

6.3.1 Cutting Implementation

The method we implement splits an object in two along a plane. (The plane is derived from the orientation of the cutting tool used in the application). This simplifies the general cutting problem somewhat, as (a) we do not have to deal with partial cuts, and (b) the internal surface revealed by the cuts is always planar.

First, we define a *sever* operation which, taking an object o and a cutting plane c, removes all of o in c's positive halfspace and neatly seals up the exposed cross-section. The *cut* operation then consists of two *sever* operations: $sever(o,c)$ and $sever(o_{clone},-c)$, where o_{clone} is a clone of o and $-c$ is c with normal reversed.

The first step of *sever* is to separate o's triangles into categories. Triangles with $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$ in the positive halfspace of the cutting plane are discarded. Triangles with $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$ all in the negative halfspace of the cutting plane are kept. The remaining triangles straddle the cutting plane, and are cut along c to obtain a clean edge. These triangles have either exactly one or exactly two vertices in c's negative halfspace. The former kind are shortened to produce the clean edge; the latter kind are cut to form two subtriangles (see figure 13).

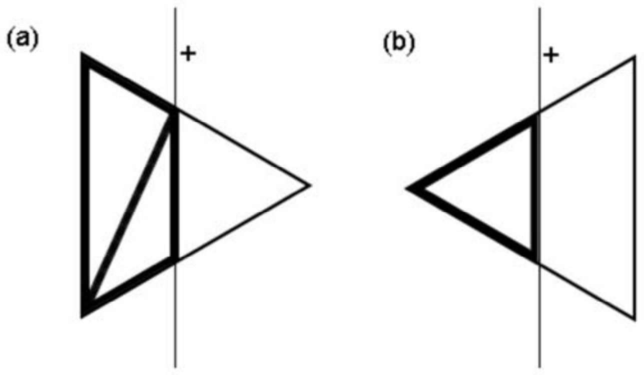


Figure 13: Triangles are made flush with the cutting plane's surface by (a) division into two smaller triangles, or (b) shortening.

When this process is carried out over every triangle, a neat edge all around the cutting plane is produced. Figure 14 shows the results for an axis-aligned cutting plane; figure 15 for a diagonal cutting plane.

The next step is to seal the exposed cross-section. A surface is created by triangulating the newly created vertices touching the cutting plane with a delaunay triangulation algorithm (see figure 16).

The triangles tend to be irregularly shaped (see figure 16) because only vertices around the edge of the surface are fed into the algorithm. With no vertices in the centre, each triangle needs to span edge to edge. An improvement to our method would add new

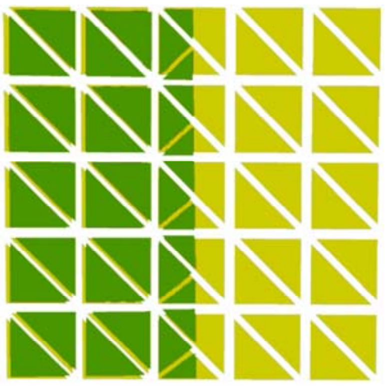


Figure 14: An overlay shows the result of a cut down the centre of the object.

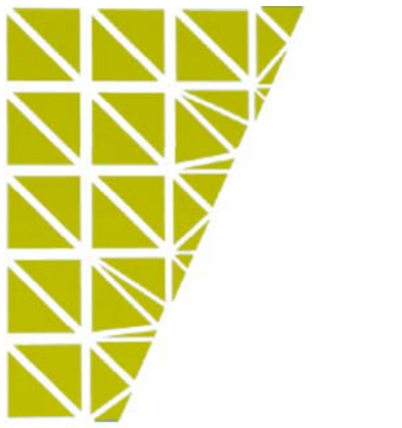


Figure 15: Triangulation around a diagonal cutting plane.

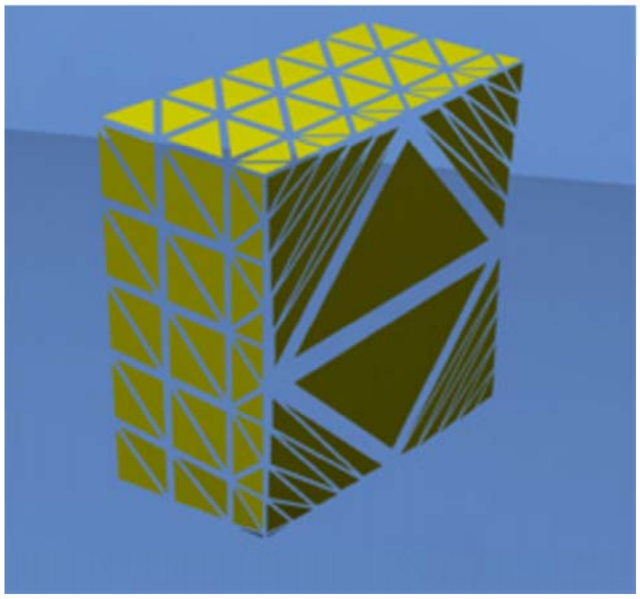


Figure 16: After a cut, the exposed internal hole is sealed up with a delaunay triangulation.

vertices inside the edges before running the delaunay triangulation algorithm, resulting in more consistently sized and shaped triangles.

After triangulation is performed, the object is tetrahedralized and divided into clusters again. Another possibility would be to keep what was left of the old tetrahedrons and clusters, but possible cluster degeneracy would need to be dealt with.

6.3.2 Problems

Many new triangles are created from the modification of triangles straddling the cutting plane. When two of these triangles are adjacent, they will share at least one vertex with the same position. If both triangles reference the same vertex, normals will be calculated to display a smooth surface. If both triangles reference two different vertices with identical positions, calculated normals can show an edge between the two triangles. This should be taken into account when creating the new triangles, but currently we give each new triangle unique vertices, so the surface can potentially look rough when it should be smooth. This is not trivial to fix, however. Each triangle would need to communicate with adjacent triangles somehow to find out which vertex to share. Or alternatively, upon vertex construction that position could be checked for already existing vertices, perhaps against an ordered list or heap of previously added vertices. This would lessen efficiency, which is an important consideration when cuts are carried out in real-time.

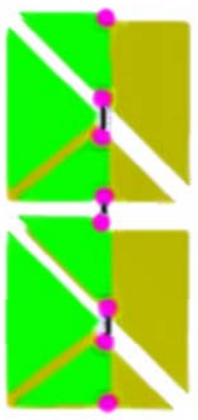


Figure 17: Vertices resulting from new triangle creation are duplicates. Each pair of vertices linked by a line should be merged into one vertex.

A problem with that method though is that where new vertices are created along a previously existing edge, we would like to duplicate vertices in order to preserve the edge. Slightly rough surfaces are more visually appealing than abnormally smooth normals around otherwise sharp edges. It seems to avoid this each triangle would have to check with adjacent triangles, and only share new vertices with triangles it already shared an edge with, i.e. triangles with which it shared unique, not duplicated vertices. For now though, it is very difficult to see the non-smoothed triangulations, so we leave the implementation as is.

A requirement of the delaunay triangulation algorithm is that the input is free of duplicate vertices (i.e. vertices with near identical positions). To achieve this, we simply create a separate list of duplicated cutting surface vertices, ensuring every new vertex added to the list has a unique position. Delaunay triangulation is performed on this separate list. The edges of the surface produced, consisting of duplicate vertices, are thus sharp (as they probably should be).

Another problem is that at times, the delaunay triangulation produced by the algorithm we used is incorrect (see figure 18). This is likely a problem with the particular implementation, so experimentation with other implementations is necessary for a more robust result.

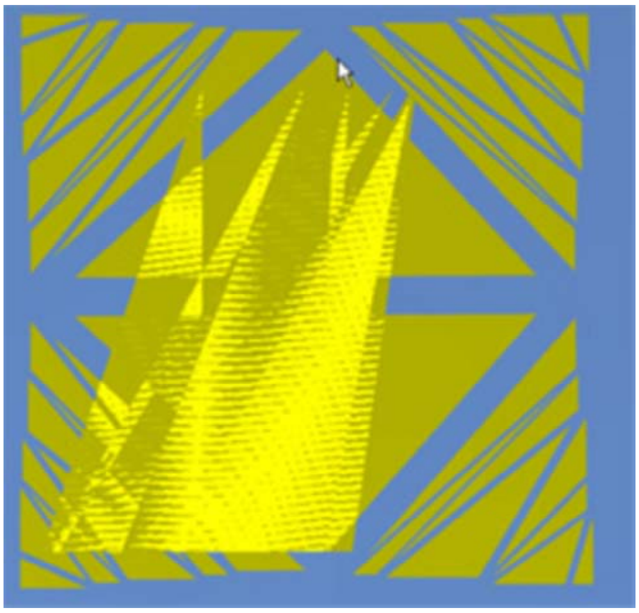


Figure 18: Delaunay triangulation produces an incorrect result.

We used delaunay triangulation of a 2D surface, however, the surface we wished to triangulate was actually a planar surface in 3D. To circumvent this, we converted the 3D coordinates of the vertices into 2D coordinates, relative to the cutting plane. Feeding these coordinates into the algorithm then, we should get back triangle indices that can be immediately used with the 3D vertices. Unfortunately the particular implementation we used required that the input 2D points be sorted in order of increasing x. To preserve the mapping from 2D to 3D we created a data structure consisting of a 2D coordinate and an index into the 3D vertices' array, where the 3D vertex indexed is mapped to the 2D coordinate. The array of these data structures is then sorted into order of increasing x. The triangle results of the delaunay triangulation index into this array, from which we can extract the correct index into the 3D vertex array.

6.4 Collision

Several types of methods are available detecting and responding to collisions between deformable objects. These include bounded volume hierarchies, stochastic methods, distance fields, spatial subdivision, and image-space techniques [Teschner et al. 2004b].

Our application uses spatial hashing [Teschner et al. 2003] and penetration depth estimation [Heidelberger et al. 2004] techniques. We found that collision detection was a performance bottleneck however. No "best way" to perform collision detection for deformable objects has been decided on yet, and future research will improve this area.

7 Results

We have implemented meshless deformation based on shape matching with our improvements into an basic environment suitable for virtual surgery or games. The user may pick, push or cut deformable objects in real-time. Simple objects with limited modes of deformation are simulated best, while objects composed of simple subcomponents are simulated well with clusters. Objects with a very high number of deformation modes, such as cloth, can not be simulated efficiently [Rubin 2006].

Usability. Informal user testing indicates that our environment and all our interaction techniques were intuitive and easy to use. The ability to push, pull, fix and cut deformable colliding objects significantly increased user enjoyment.

Ease of implementation. We found meshless deformation relatively easy to implement and integrate into the 3D rendering engine Ogre. There are only two main differences between current 3D engines and what is required for deformable object simulation. Firstly, rigid objects have static sharable meshes, while deformable objects require updates to individual vertex positions every timestep on their own mesh instance. Secondly, collision detection and response is a much slower, more difficult task for deformable objects.

Performance. Our environment is comparatively fast: We can simulate dozens of simple 32 tetrahedron objects with collisions in real-time and unconditional stability (see figure 20). Suitable speed for simple virtual surgery applications could be achieved by optimising our algorithms and/or implementing them on the GPU. However where deformable behaviour is less important, for example in most games, we predict it will be at least several years before deformable object simulation is the best marginal use of processing power.

Tweakability. The "gooeyness" and stiffness of each object can be easily modified using the α and β parameters. Further collision-response parameters can also be tweaked. The strength of surface area preservation can be specified with a force response curve. Volume preservation is automatic, but can be adapted to use a force response curve as well.

Disadvantages. The primary disadvantage of our environment is the lack of robust local deformation. For complex virtual surgery applications which require plausible localized deformation of an arbitrary region, our environment is less suitable. Also, even when simulation is visually plausible, it is usually not physically accurate.

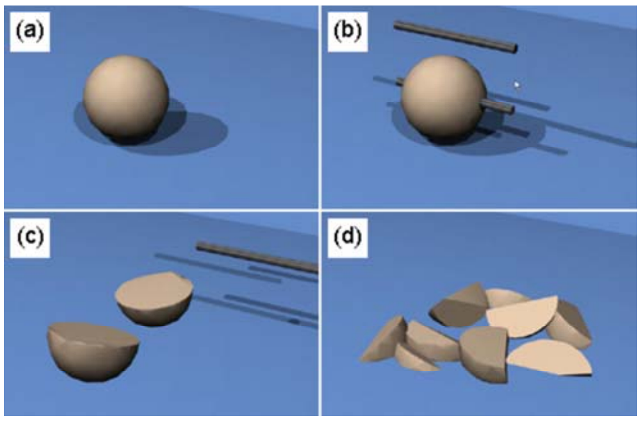


Figure 19: Cutting an object: (a) during cut, (b) immediately after cut, (c) two resulting halves have rolled apart, (d) after further cuts.

8 Conclusion

We have presented an environment in which we implemented an improved algorithm for meshless deformation based on shape matching. Our environment includes several novel interaction techniques that allow users to interact with objects in an intuitive and realistic manner. We have also implemented collision detection and

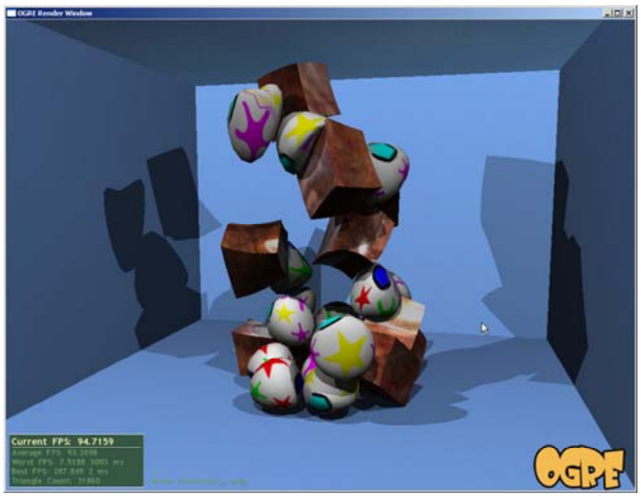


Figure 20: Large scale simulation of deformable objects.

response for deformable objects based on spatial hashing and accurate penetration depth estimation techniques. Informal user testing indicates that users find our environment significantly more enjoyable and immersive than a comparable rigid body physics environment.

Our environment handles manipulation of simple objects with visual plausibility, though not physical accuracy. Entities that deform as a whole along quadratic modes such as stretch, shear, bend and twist are modeled best. More complex entities such as intestines need to be split up into many clusters, decreasing efficiency and visual plausibility. Localized deformations are not currently possible.

Performance is adequate for simple circumstances, but more work probably needs to be done for more realistic situations. With no optimization effort, twenty low tetrahedron cubes and spheres were able to be simulated with collision detection at an interactive rate. However with optimization and implementations on the GPU or specialized physics processors, performance could be significantly increased.

In summary, we believe that the implemented techniques have promising potential as applied to a virtual surgery simulator, games, or any other environment where speed and immersive interactions are required but physical accuracy is not important.

9 Future Work

The major problem limiting meshless deformation's use in virtual surgery applications is the lack of robust local deformation, and the difficulty of using clusters to achieve even limited local deformation. One avenue of investigation might be to integrate a mass-spring system, which is usually disabled, but where user picks activate mass-spring behaviour in the pick's local region. Perhaps mass-spring areas around a partial cut or incision could similarly be activated. For larger cuts, but not complete severances, a method of dynamically partitioning new clusters may be possible that would allow "flapping" behaviour, similar to a tennis ball nearly cut in half with both halves "talking" like a mouth.

References

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*. Elsevier Science Publishers, Amsterdam, North-Holland, 3–10.

ARUN, K., HUANG, T., AND BLOSTEIN, S. 1987. Least-squares fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence 9*, 5, 698–700.

BULKA, A., 2003. Model GUI Mediator.

BURBECK, S. 1992. Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC). *online article, http://st-www. cs. uiuc. edu/users/smarch/st-docs/mvc. html (up on 13/4/04)*.

DOMPIERRE, J., LABBE, P., VALLET, M., AND CAMARERO, R. 1999. How to subdivide pyramids, prisms and hexahedra into tetrahedra. *8th International Meshing Roundtable*, 10–13.

EDDIE, C. Crazy Eddies GUI FAQ. URL: `http://www.cegui.org.uk/wiki/index.php/What_is_CEGUI`.

EDDIE, C. Crazy Eddies GUI main website. URL: `http://www.cegui.org.uk/wiki/index.php/Main_Page`.

FELIPPA, C. 2001. Introduction to Finite Element Methods. *Department of Aerospace Engineering Sciences and Center for Aerospace Structures, University of Colorado Boulder*.

HEIDELBERGER, B., TESCHNER, M., SPILLMAN, J., AND MUELLER, M. DefColStudio. URL: `http://graphics.ethz.ch/~brunoh/defcolstudio.html`.

HEIDELBERGER, B., TESCHNER, M., KEISER, R., MULLER, M., AND GROSS, M. 2004. Consistent penetration depth estimation for deformable collision response. *Proceedings of Vision, Modeling, Visualization VMV04, Stanford, USA*, 339–346.

HORN, B. 1987. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America A 4*, 4, 629–642.

MEYER, M., DEBUNNE, G., DESBRUN, M., AND BARR, A. 2001. Interactive animation of cloth-like objects in virtual reality. *The Journal of Visualization and Computer Animation 12*, 1, 1–12.

MÜLLER, M., HEIDELBERGER, B., TESCHNER, M., AND GROSS, M. 2005. Meshless deformations based on shape matching. *ACM Trans. Graph. 24*, 3, 471–478.

NOH, J., AND NEUMANN, U. 1998. A survey of facial modeling and animation techniques. Tech. rep., USC Technical Report, 99–705.

RUBIN, J. 2006. A framework for interactive and physically realistic cloth simulation. 780 project report, University of Auckland, Feb. `http://www.cs.auckland.ac.nz/~burkhard/Reports/2005_SS_JonathanRubin.pdf`.

SPILLMANN, J., AND TESCHNER, M. Contact Surface Computation for Coarsely Sampled Deformable Objects.

TESCHNER, M., HEIDELBERGER, B., MUELLER, M., POMERANETS, D., AND GROSS, M., 2003. Optimized spatial hashing for collision detection of deformable objects.

TESCHNER, M., HEIDELBERGER, B., MULLER, M., AND GROSS, M. 2004. A versatile and robust model for geometrically complex deformable solids. *Computer Graphics International, 2004. Proceedings*, 312–319.

Teschner, M., Kimmerle, S., Zachmann, G., Heidelberger, B., Raghupathi, L., Fuhrmann, A., Cani, M.-P., Faure, F., Magnetat-Thalmann, N., and Strasser, W. 2004. Collision detection for deformable objects. In *Eurographics State-of-the-Art Report (EG-STAR)*, Eurographics Association, Eurographics Association, 119–139.

Umeyama, S. 1991. Least-squares estimation of transformation parameters between two point patterns. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 13*, 4, 376–380.

Wikipedia, 2006. Finite element method — wikipedia, the free encyclopedia. [Online; accessed 21-October-2006].

A Critical Comments

Müller et al. write [Müller et al. 2005],

> The versatility of the approach in terms of object representations that can be handled, the efficiency in terms of memory and computational complexity, and the unconditional stability of the dynamic simulation make the approach particularly interesting for games.

What role then might deformable objects in general – and meshless deformation in particular – play in games? Rigid body physics is starting to become common in games. In Half-Life 2, rigid body physics allows the player to pick up objects and stack them on top of eachother, or use a "gravity gun" to grab and fire objects at enemies. In Oblivion, some traps involve logs rolling down a slope to crush enemies. Such examples make it easy to see how rigid body physics can be an important, though perhaps not integral, part of gameplay. What about soft body physics though?

Cloth, liquids and particle effects are widespread and useful applications of soft body physics. Meshless deformation however applies only to discrete objects with limited modes of deformation; a beach ball, rather than cloth, liquid or a particle. A bouncing beach ball, jelly, or a giant rubber ducky would look more realistic modeled with soft body physics than with rigid body physics. But this small extra visual realism would hardly add to gameplay. Indeed with respect to one's normal activities in the real world, it is hard to think of a situation where objects undergo significant and visible soft body deformation. A cushion, perhaps? A trampoline? Maybe a child would enjoy playing with stretchy objects, but we find it hard to think of a mainstream game application where meshless deformation could be important. With networked games certainly bandwidth is too expensive to send every deforming mesh to every client each timestep, so deformation calculation would need to be client-side, and could not impact gameplay.

Then there are the performance issues. Meshless deformation's per-particle simulation is significantly more computationally expensive than rigid body physics' per-entity simulation. Precomputed bounded volume hierarchies (BVHs) allow for efficient rigid body collision detection; BVHs would have to be (partially) recomputed every timestep for use with soft body physics [Teschner et al. 2004b].

Memory usage is important too. In a real-time environment, the number of unique models allowed onscreen is limited by memory constraints. Several instances of a rigid model can be rendered onscreen simultaneously with little memory overhead. This is because the same model can be sent through the graphics pipeline each time–only the transformation matrices etc. differ. Deformable objects though generally allow vertex by vertex changes, thus each instance requires its own space in memory. If the choice is multiple instances of a rigid object versus one instance of a deformable object, the former may prove more attractive.

To summarize, we predict neither meshless deformation nor any similar technique will achieve widespread use in games for at least several years. Any benefit would be dubious, and costs significant.

B Collision

Several types of methods are available detecting and responding to collisions between deformable objects. These include bounded volume hierarchies, stochastic methods, distance fields, spatial subdivision, and image-space techniques [Teschner et al. 2004b].

We examined several methods for collision detection and response, all of the spatial subdivision and hashing type described in [Teschner et al. 2003]. *Spatial hashing* implicitly subdivides $\mathbf{R}^3$ into small grid cells. A hash function then maps 3D grid cells to a hash table. For example, suppose we want a list of all vertices intersecting a particular tetrahedron. The hash entry of each grid cell partly or wholly inside the tetrahedron is simply checked for vertices, and any vertices found are checked fully for actual intersection. This reduces the set of vertices tested from every vertex to a small number of vertices near the tetrahedron (assuming no hash collisions). Using this spatial subdivision, we experimented with a variety of methods.

B.1 Vertex/Triangle intersections

This method aims to test whether each vertex is colliding with, i.e. inside the bounds of, any entity.

First each vertex is hashed to a grid cell. Then, each triangle's bounding box is calculated, giving a cube of grid cells to iterate through. For each vertex in each grid cell the triangle's bounding box intersects, we can check for collisions between the triangle and the vertex. This turns out to be difficult, however, as figure 21 demonstrates.

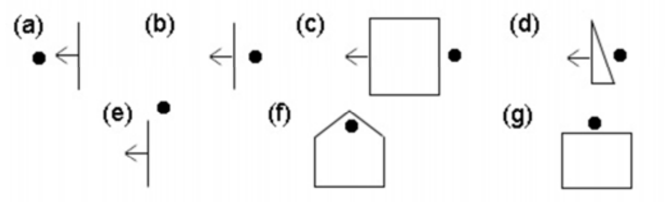


Figure 21: Different vertex positions with respect to collision triangles.

There are three general possibilities. The vertex may be in: 1. The triangle's positive halfspace (thus non-colliding); 2. The triangle's negative halfspace, inside the triangular prism extended along the triangle's normal; or 3. The triangle's negative halfspace, *outside* the triangular prism extended along the triangle's normal.

1. In figure 21 a), the vertex is in the triangle's positive (?) halfspace so clearly does not intersect. The remaining situations are not quite so simple.
2. In figure 21 b), the vertex is just behind the triangle in its negative halfspace and extruded triangular prism, so one may think it safe to assume an intersection. But how far is just behind? That the vertex and triangle occupy the same hash location implies that it is likely the vertex is inside the triangle's bounding grid cells, and thus reasonably close. But there is no assurance that this is the case; the hash function may "collide" and map two grid cells in entirely different locations to the same hash location. Thus the colliding vertex could be halfway across the room, and certainly not intersecting, e.g. c). To solve this,

the distance from the vertex to the plane of the triangle could be calculated, with collisions only occurring if the vertex is below a set distance from the plane perhaps dependent on the size of the triangle. This clearly won't work either, however; a long thin object may produce false negatives, and a short object may produce false positives, e.g. d).

3. In the final situation, the vertex is in the triangle's negative halfspace, but outside the triangle's extruded prism, e.g. e). This provides even more problems than the previous situation. In general it is impossible to tell whether the vertex is "colliding" in any meaningful sense with the entity purely from its relationship with the triangle; f) and g) are both possibilities that cannot easily be ruled out.

So it is in general impossible to tell whether a vertex is intersecting an object based on its position relative to any single triangle but what about methods involving more than one triangle?

At the extreme of methods involving more than one triangle, we could say a vertex is colliding if and only if it is in the negative halfspace of every triangle in the entity. This is a nonstarter though in general it will only correctly detect collisions for convex objects (figure 22), and even then it requires testing every vertex against every triangle an unacceptably slow proposition.

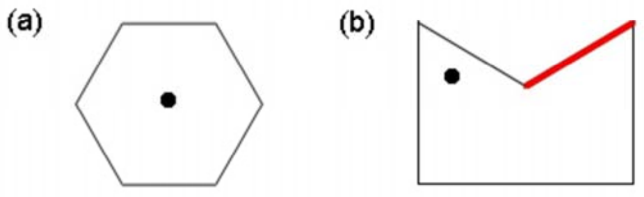


Figure 22: In the convex object A, an intersecting vertex is within the negative halfspace of every triangle, so collides. In the non-convex object B, the vertex is intersecting, but is not in the negative half-space of the bold red triangle, so the method fails.

Ignoring efficiency problems for the moment, the above method suggests there might be a way to detect intersection even in non-convex objects by counting the number of triangles the vertex is within the positive or negative half space of. A quick counter example in figure 23. shows why this cannot work. Any particular triangle can be split into an arbitrary number of subtriangles, showing that any intersecting vertex, intersecting or not, is within an arbitrary number of positive/negative halfspaces.

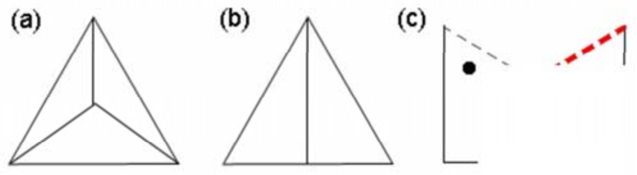


Figure 23: In A, a triangle is split into 3 subtriangles. In B, into 2 subtriangles. C shows that the vertex is within the positive/negative halfspace of an arbitrary number of triangles, depending on how many triangles different portions of the mesh contain.

So we cannot get anywhere with pure halfspaces. How about with the "extruded prism" halfspaces? No matter how many times a triangle is split into subtriangles, a vertex sitting on top of it will always remain sitting on only one subtriangle (assuming the vertex is not on an edge).

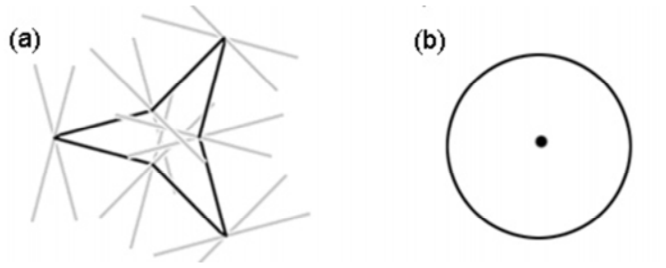


Figure 24: A. Black lines denote a star shaped object. Grey lines are the extrusions of the black lines along their normal. B. A vertex intersecting a circle.

Figure 24 shows an object and diagrams the prism halfspaces. (Figure 24 is in 2d. The extruded prisms here correspond to the area formed by a line sweeping along its perpendicular axis. For the 3d case, imagine you are looking down on the top of a star shaped prism. The black lines show the triangles with horizontal normals, while the lid and floor of the prism are flat on the page. In the 3d case, each intersecting vertex is in exactly two additional negative prism halfspaces compared to the 2d case; that of the lid triangle directly above the vertex, and that of the floor triangle directly below it.)

Here we can see that if a vertex is in the very middle of the star, it is contained in zero prism halfspaces (two floor and ceiling - for the 3d case). Move the vertex into one of the adjoining areas, and it can be in one, two, or three negative prism half spaces. In figure 24 B, clearly the vertex is in the same number of negative prism halfspaces as there are triangles. So in general, an intersecting vertex can be in an arbitrary number of negative prism half spaces.

What about positive prism halfspaces? In figure 24 a), if a vertex is in any positive prism halfspaces, then it is not intersecting. However, the converse does not necessarily hold; a vertex can be non-intersecting without being in any halfspaces at all. This is only a problem if intersecting vertices can be outside all halfspaces. In the 2d case, this is clearly possible. In the 3d case, assuming the star is a prism, all intersecting vertices are within the prism halfspaces of the floor and ceiling. Suppose however that floor and ceiling triangles meet at the centre of the star. If the floor centre point is raised up slightly, and the ceiling centre point lowered slightly, a vertex in the very centre of the star object will be outside all halfspaces. This is show in figure 25, where the red area is inside the star yet outside all halfspaces (assuming the sides of the star are shaped as in figure 24).

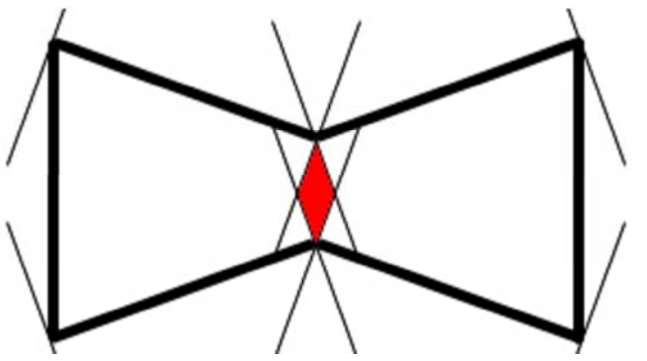


Figure 25: An internal area outside all triangle halfspaces.

So in general, a vertex outside all prism halfspaces can be either intersecting or nonintersecting. Further, a vertex inside an arbitrary number (possibly including zero) of negative prism halfspaces can be either intersecting or nonintersecting. If a vertex is in a positive prism halfspace though, is that a guarantee it is nonintersecting? For the examples given so far (which include non convex objects), yes. But not in general. The property of the examples given so far that makes them exhibit this property is a kind of pseudo convexity for any surface triangle, no extrusion along the positive direction of its normal intersects the object. It's easy to create a counterexample to show this property does not always hold.

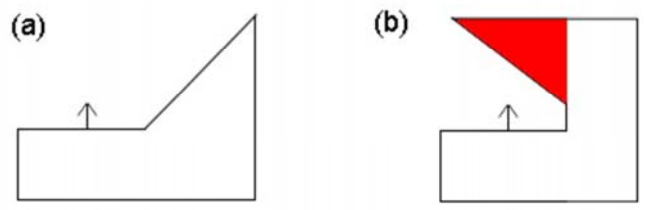


Figure 26: Two different types of convexity.

Figure 26 (a) shows that the objects so far, while non convex, are pseudo convex (not a technical term, ??) with respect to extrusions of their surface. Figure 26 (b) shows an object where a vertex can be in the positive prism halfspace of a surface triangle, yet still intersecting. To see that the number of positive prism halfspaces a vertex lies within is arbitrary, note that the surface extruded in (b) could be morphed into an arbitrary number of subtriangles, all of which have normals pointing directly at the intersecting vertex (the surface would be a kind of bowl shape).

To sum up, in general:

- A vertex outside all prism halfspaces can be either intersecting or non-intersecting.
- A vertex inside any number of negative prism halfspaces can be either intersecting or non-intersecting.
- A vertex inside any number of positive prism halfspaces can be either intersecting or non-intersecting.

We are faced with abject failure then; there does not seem to be any general way to determine intersection of non convex objects purely from (prism) halfspaces. The discussion suggests another possible solution however. Recall the situation where a vertex is in the negative halfspace of a surface triangle.

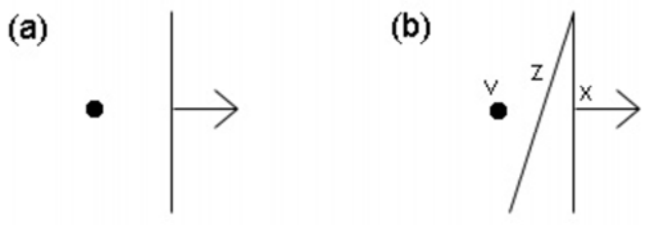


Figure 27: a) an intersecting vertex in the negative halfspace of a triangle; b) a non-intersecting vertex in the negative halfspace of a triangle.

As discussed, there is no way to detect the intersecting state of the vertex from (prism) halfspaces alone. Consider the difference between figure 27 (a) and (b) though. The vertex v in (b) is not intersecting because between it and the triangle under consideration x, there is another triangle z facing towards the vertex. In general if there are no additional triangles between v and x, then v must be intersecting. If there are additional triangles between v and x, then v is intersecting if and only if the number of additional triangles z is even. For proof of this, note that no ray passing through an object can consecutively pass through two triangles facing the same direction from the ray's perspective. The intersecting/nonintersecting state of v then must alternate for each additional z between v and x. If there are zero additional triangles z, then v is intersecting. One z, v is non-intersectiong; two z, v is intersecting; and so on.

Suppose there are multiple triangles z_i, $i = 1 \ldots n$ between v and the midpoint of x. Let the triangles be ordered by distance from v, i.e. z_0 is the first triangle a ray from v to the midpoint of x passes through. Now note that between v and z_0 must be a clear path (if there were a triangle t between v and z_0, then z_0 would have been calculated to be t). One might think that if for example we are iterating through all triangles x to detect an intersection for v, any x with triangles z between x and v can be skipped, because collision can always be detected during the iteration process when the x is equal to the previous z_0. It turns out this isn't the case. While there must be a clear path between any z_0 and v, that path need not necessarily be from v to z_0's midpoint. See figure 28 for an example.

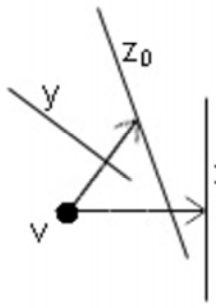


Figure 28: A ray from v to z_0's centre point passes through y.

Here a ray cast from v to x's midpoint passes through z_0. However, a ray from v to the midpoint of z_0 is not a clear path; it passes through y.

This tells us that for any vertex/triangle pair (v,x), we can detect a collision by counting the number of triangles z between v and x (but we can't skip x assuming that a later triangle will detect the collision). If v is in x's positive halfspace, then an odd number of z's implies collision, an even number non-collision. If v is in x's negative halfspace, then an even number of zs implies collision, an odd number non-collision.

This approach is similar to for each vertex v, casting a ray to a point p known to be outside the object you wish to detect collisions for. If the number of triangles intersected by the ray is even, there is no intersection; odd, there is an intersection. See figure 29 for an example.

The problem with the above approach is that in a naïve implementation, to detect all triangle collisions for a particular ray requires the ray be tested against every triangle in the object - an $O(vt)$ proposition (where v is the number of vertices and t the number of triangles). Spatial hashing would improve the efficiency of the method, in that it would limit the triangle/ray intersection tests performed to those triangles inside the same hash cells as the ray passes through. Given that a ray could be quite long however, and pass through many objects before it is guaranteed to be at a non-intersecting point, this is not necessarily as efficient as we would

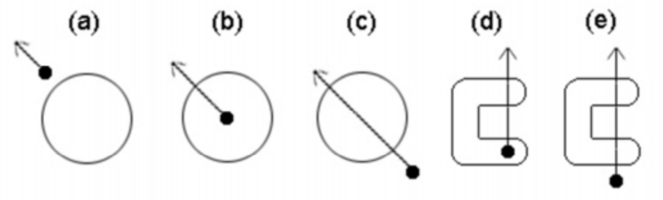


Figure 29: (a) 0 triangles are intersected and there is no collision; (b) 1 triangle is intersected and there is a collision; (c) 2 triangles are intersected and there is no collision; (d) 3 triangles are intersected and there is a collision; (e) 4 triangles are intersected and there is no collision.

like. We would like it if, for a vertex triangle pair (v,t) there were some way to get a smaller superset of the triangles intersecting a ray from v to t.

One situation in which it might be possible to get a small superset of the triangles intersecting a ray from v to t is when v occupies the same grid cell as the part of the bounding box of t. If this is the case, then one might think that any triangles between t and v also occupy the grid cell of v, so we'd have an acceptably short list of triangles to test intersection with. This turns out not to be true – see figure 30

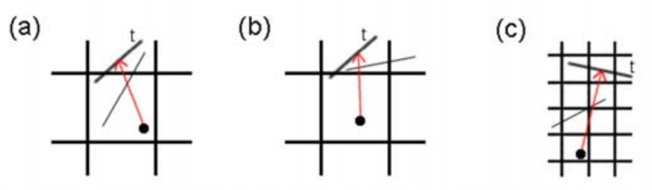


Figure 30: (a) the cell occupied by v provides a complete list of the triangles between v and t, from which we can detect collisions; (b) the cell occupied by v does not contain the triangle between v and t, but one of the cells occupied by t does; (c) neither of the cells occupied by t or v contain an intermediate triangle.

Figure 30 (a) and (b) suggests a complete list of triangles between v and t might be obtained by the union of all triangles in the cells occupied by either v or t. In (c) however this is shown to be wrong; the triangle between v and t is entirely disjoint from the cells occupied by v and t.

So it seems that from any triangle t occupying the same voxel as a vertex v, we cannot get an accurate list of all the potential triangles between t and v. Returning to the original problem, we wanted to, for a given object o and vertex v, find a ray from v to outside o that minimizes the number of cells traversed (or the number of triangles tested for intersection). This often translates to finding the closest triangle t to v. Then a ray r is cast from v to t's midpoint, and using an efficient voxel traversal technique [Amanatides and Woo 1987], all the grid cells intersected by r have their triangles tested for intersection with r. Then collision is detected based on the orientation of t and the number of triangles intersected by r.

The question then is how to find a triangle close to v. This is basically the same method as casting a ray from v to an arbitrary point outside the object. The only difference is the point outside the object is selected to be a point infinitesimally beyond the midpoint of a triangle (the direction dependent on the triangle's orientation). If a triangle t occupies the same grid cell as v then, it is likely that the ray from v to t will be short. It may be long in cases with very long triangles (similar to figure 30 (c)), or in cases of hash collision, when a triangle occupying the same hash cell as v is very far away in grid cell terms. (Though note the ray traversal process can be stopped at the first triangle it actually intersects. If no grid cells before t's contain triangles, the ray has simply traversed through many empty voxels. If grid cells before t's contain triangles, but the ray does not intersect them, there will be many failed intersection tests and efficiency will suffer.)

If there are no triangles occupying v's grid cells, we could choose a number of methods to deal with it.

1. Instead of a triangle, cast the ray to a point known to be outside the object under consideration, e.g. a point under the floor.
2. Cast the ray to a triangle selected at random.
3. Randomly select an empirically determined number of triangles, and cast the ray to the closest one.

In the end, selecting a triangle close to v when possible will be faster than casting a ray to a point completely outside the object, as the ray will have to pass through fewer grid cells on average. It must be noted however that when detecting collision from a ray to a triangle, collision will be detected correctly only if the mesh topology of the object is correct. To see why, look at figure 31.

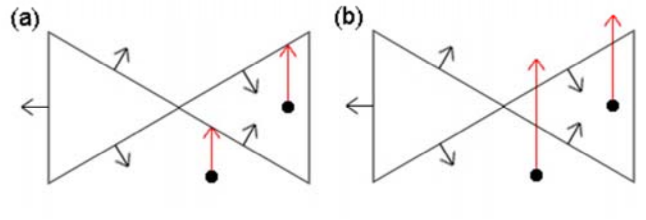


Figure 31: A triangle mesh deformed such that surface normals are not as expected.

In figure 31 (a), vertex collision is detected by comparisons with triangles. Take the non-intersecting vertex, let's call it v. A ray r is cast from v to the closest triangle t. There are no triangles between v and t, and t's normal is pointing in the same direction as r, so we conclude v is intersecting. Do the same thing with the intersecting vertex, and we conclude that it is not intersecting. Both results are wrong. In figure 31 (b) on the other hand, triangle normals and mesh topology are irrelevant, as only the number of triangles the ray passes through matters. Even with a deformed topology with strange triangle normals, collision will be correctly detected. As we cannot guarantee that mesh topologies will not become deformed in the above manner, the speed advantages of checking for collision by comparing vertices to triangles is probably not worth compromising the robustness of collision detection.

So we conclude that the only useful way of detecting collision between vertices and triangles seems to be to cast a ray from the vertex to a point outside the object, and count the number of triangles intersected. But there is still a small problem to be worked out.

If a ray passes through an edge connecting two triangles as in figure 32 (a), a naïve algorithm would calculate two intersections one for each triangle. (If the algorithm only calculated intersections inside triangles, rather than at their edges, the algorithm would calculate zero intersections.) However, in (a) we clearly want only one intersection detected. One might suggest that only one intersection at any particular point in the ray be allowed, but in figure 32 (b) that would lead to the calculation of one intersection when we want two (or zero). A solution to this is to divide the triangles a ray intersects along an edge or corner into two sets: those with normals facing

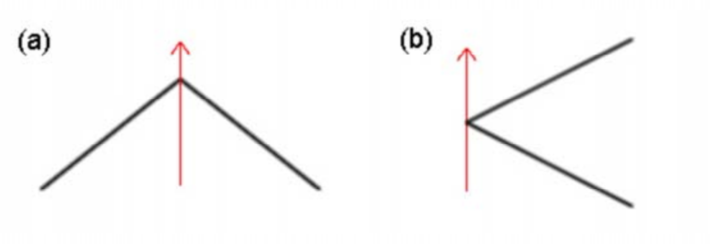


Figure 32: Rays are cast through triangle edges, causing problems with intersection counting.

the same direction as the ray (within 180 degrees), and those facing the opposite direction. Then we count only one intersection in each set. Applying this, in (a) then there are two triangles in one of the sets, so one intersections is calculated. In (b) there is one triangle in each set, so two intersections are calculated, as desired.

The method then seems reasonable robust. For now we just note that while this method can detect collisions, a separate method is required for collision response and to find the penetration depth and direction. One such method is discussed later. Next we move onto other methods of collision detection and response that we investigated.

B.2 Triangle/Triangle intersection

In this method, each triangle is spatially hashed to grid cells in the same manner as first described. Then for each triangle, its grid cells are tested for other triangles. If two triangles share the same grid cell, they are tested for intersection. So we have two intersecting triangles, and we need to respond to those collisions. In figure 33, (a) shows two such intersecting triangles, and (b) shows the kind of situation we'd like to end up with.

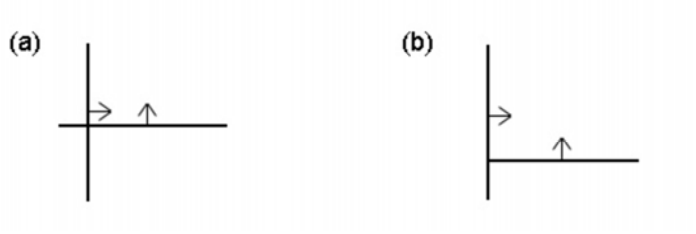


Figure 33: (a) two triangles are colliding; (b) the collision is resolved.

One way to handle response is to exert a force and/or displacement on each triangle along the other triangle's normal, such that next timestep the triangles are no longer intersecting. Suppose we displace both triangles by the same amount. As both triangles are responding to the force then, we need to find the moment in time where intersection stops.

Let one triangle v be denoted by the vertices v_1, v_2 and v_3 and the normal n_v, and let the other triangle u be denoted by the vertices u_1, u_2 and u_3 and the normal n_u. To make things easier we look at the vector r traveled by v relative to u: $r = n_v - n_u$. We want to find the distance multiple d that each v_i needs to travel along r to reach u's plane. That is, we want to find the d such that $v_i + d\mathbf{r}$ is on the plane of u.

As figure 34 shows, translating the triangle v by $d\mathbf{r}$ will move the triangle out of intersection. To ensure that the triangle is completely out of intersection, we need to calculate d for every point of v, then use the maximum d(a d other than the maximum would leave the point with maximum d still intersecting). This d may overshoot however (see figure 35).

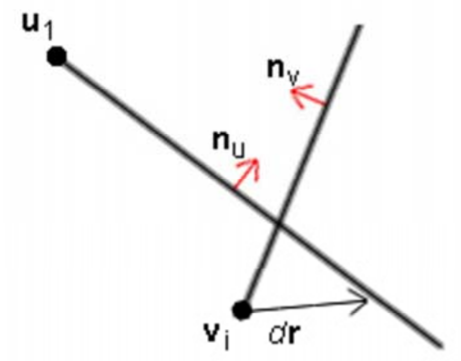


Figure 34: Collision resolution vectors of colliding triangles.

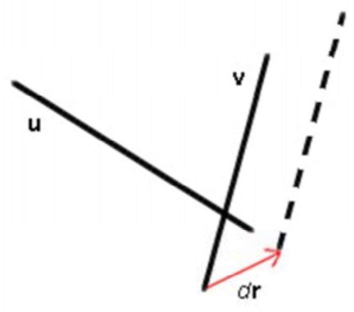


Figure 35: Moving triangle v by $d\mathbf{r}$ moves v too far out of intersection.

The solution is to calculate a dv and a du, where du is the distance along $\mathbf{r}_u$ that u needs to move to no longer be intersecting v. To sum up then, we have:

$$d = min(max(dv_1, dv_2, dv_3), max(du_1, du_2, du_3))$$

Once we have the final d, we set

$$v = v + d\mathbf{n}_u$$
$$u = v + d\mathbf{n}_v$$

Note that this results in equal movement by both triangles. If one object were stiffer than the other, the method could easily be adapted so that the triangles of stiff objects moved comparatively less when intersecting with triangles of less stiff objects.

Implementing this method, we find that it works well but for one major problem. Intersections between triangles are resolved nicely, but when penetration is large, there can be many triangles inside another object, i.e. colliding, but not intersecting any other triangles. See figure 36 for an example.

Here collisions of the red intersecting triangles would be resolved, but the triangles completely inside the other object will not be detected as colliding. (And if they were detected as colliding, we would need a completely different method of collision response to determine how to resolve their collisions.)

The effect of this is to produce plausible collision detection and response, but only when objects are moving very slowly. If objects move slowly, nearly all colliding triangles will be near the surface of the other object, thus also intersecting its triangles. In our first effort at implementation, we accidentally tested each triangle against the other object's nearby triangles multiple times. This was

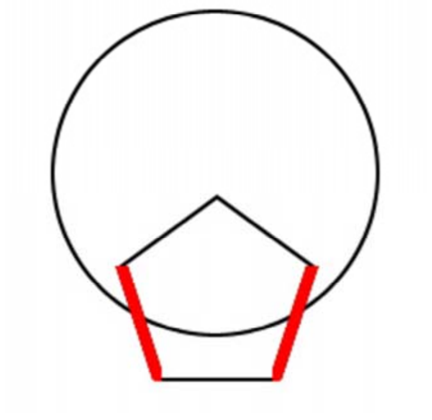


Figure 36: Collision response only affects penetrating triangles, and ignores internal colliding triangles.

effectively the same as running the above algorithm multiple times, where after each step of collision detection/response formerly colliding but not intersecting triangles would be brought closer to the surface, and eventually detected as intersecting. Unfortunately such repetition still only worked on reasonably slow moving objects, and was quite inefficient compared to methods investigated in following sections.

In summary, this triangle/triangle collision detection and response method produced plausible results, but only inefficiently and with slow moving objects.

B.3 Edge/Triangle Intersection

In this method each triangle is hashed to the grid cells it intersects. Then for each edge of each triangle, we test intersecting grid cells using a fast voxel traversal technique [Amanatides and Woo 1987]. See figure 37 for an example.

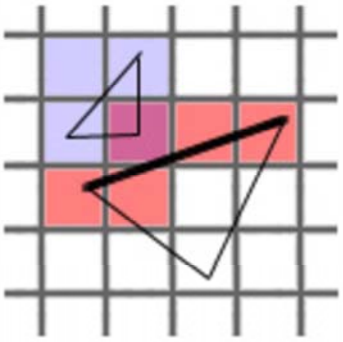


Figure 37: Two triangles are hashed to grid cells for collision detection.

Here the blue triangle has been hashed to the blue coloured grid cells. We are iterating through edges, and the current red coloured edge is part of the magenta coloured triangle. Using a voxel traversal technique, each grid cell that the edge intersects is tested for other triangles. In this case, we see that the edge passes through a grid cell containing the blue triangle. The red edge is then tested for intersection with the blue triangle (in this example there is no intersection so the test returns a negative).

When we find that an edge e is intersecting a triangle t, we classify one vertex of e as colliding, and the other vertex as non-colliding, based on whether the vertex is in the positive halfspace of t (non-colliding) or the negative halfspace of t (colliding). After this process is complete, we should have a situation like the one shown in figure 38.

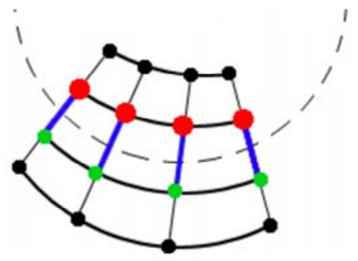


Figure 38: Two objects are tested for collision. Intersecting edges are calculated. Vertices on the inside end of an intersecting edge are classified as colliding; vertices on the outside end as non-colliding.

From here we could exert response forces on the colliding vertices, however that would leave the unclassified (black) yet colliding vertices inside the object with no response force similar to the problem we had with triangle/triangle collisions. Instead we note that the non-colliding green vertices form a kind of protective shield such that all unclassified vertices adjacent to (red) colliding vertices must be unclassified colliding vertices. So to classify all colliding vertices correctly, we can execute a graph search with classified (red) colliding vertices as the starting set, growing to encompass any adjacent unclassified vertices, but terminating at classified (green) non-colliding vertices. See figure 39 for an example.

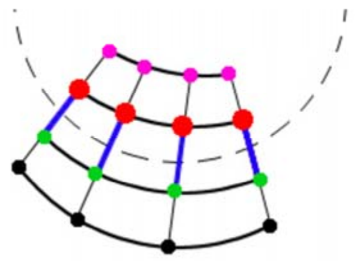


Figure 39: All colliding vertices are classified. Red vertices are colliding "border points", adjacent to non-colliding vertices. Magenta vertices are internal colliding points.

We classify the original colliding vertices, that is the vertices connected to an intersecting edge, as "border points". From here we seem to be in a good position to calculate response forces for all colliding vertices. There are three problems however.

1. Mere edge collision will not necessarily classify all border points correctly. See figure 40 for an example. One vertex is detected as a colliding border point, when in reality both points are non-colliding.

   One possible solution to this is to let the non-colliding classification override the colliding classification. Thus the edge in figure 40 will have intersection detected twice. Each intersection will classify one vertex as colliding and the other as non-colliding, but each intersection will give opposite classifications. If we let non-colliding override colliding, both vertices will end up classified as non-colliding, as desired. But

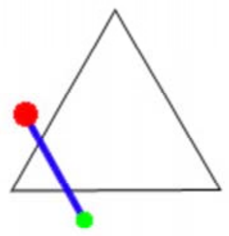


Figure 40: An edge passes completely through another object, resulting in incorrect collision classification.

this still does not solve all cases the two vertices above may actually be colliding, if the triangular object is thought of as a hole in a larger object. In general it is quite difficult (especially with several objects at once) to correctly determine whether a vertex is colliding, and if it is colliding whether it is a border point or an internal colliding point. This is not necessarily a show-stopper however, as troublesome cases such as those just described are not the norm.

2. The correct partitioning of colliding versus non-colliding vertices requires that the classified non-colliding vertices completely separate the unclassified non-colliding vertices from the border vertices. Otherwise, when the graph search takes place, far too many vertices will be detected as colliding. Figure 41 is an example. Here the rightmost edge is missed somehow when intersections were being tested for. Now when we do the graph search starting from the red border vertices, the green non-colliding vertices no longer block the path to the outer black non-colliding vertices. This results in all but the green non-colliding vertices being classified as colliding an unacceptable result.

   But why was that one intersecting edge that caused the problem not detected as intersecting? In theory it should always be detected, but in practice we found that when objects intersected very slightly, with edges almost parallel to the surfaces of other objects, very often one edge of dozens would be missed, rendering collision detection and response for that time step useless. Thus the method as described is not very robust at all, and depends on every non-colliding point adjacent to every border point being detected and classified. If even one is missed, the entire process collapses.

   It was suggested to us that there may be some stochastic way to keep partitioning consistent and correct, but it is difficult to see how an algorithm might distinguish between an unclassified colliding vertex, and an unclassified vertex that should have been classified as non-colliding.

3. Assuming the vertices do get partitioned correctly, we now know: (a) which vertices are colliding, (b) which vertices are not colliding, and (c) which intersecting edges correspond to which border points. We now need to decide how response forces will be calculated. Ideally response forces would be similar to figure 42 (a).

   These kind of responses are not trivial to compute, however. In the next section we describe an algorithm where response forces are first calculated for border points based on their intersecting edges, then all the colliding vertices adjacent to border points have their response forces calculated based on the response forces of their adjacent border points. This was repeated in the manner of a breadth-first search, where at each step the current set of colliding vertices had their response forces calculated from the response forces of the previous set of colliding vertices (which at step one were border points). Tried here however, this method results in response forces similar to figure 42(b).

   There are other possible ways to calculate response forces, but given the other two problems with this method of collision detection, we decided to move on to an entirely different and more promising method instead.

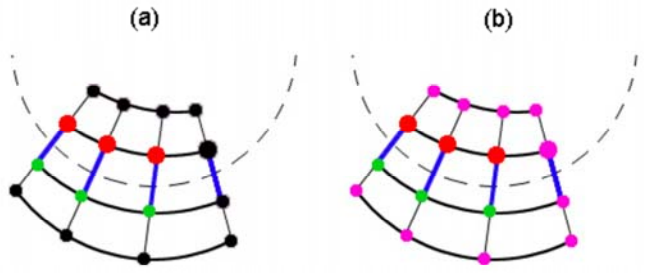


Figure 41: A colliding edge is missed, and non-colliding vertices are incorrectly partitioned as colliding vertices.

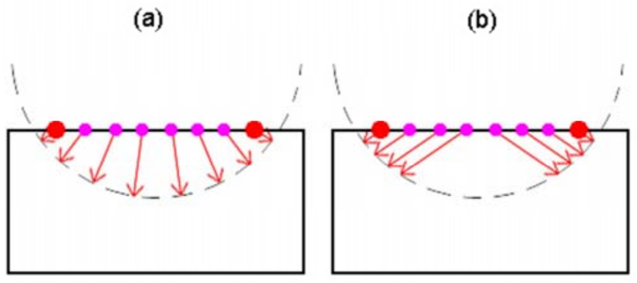


Figure 42: (a) Ideal response forces. (b) Response forces created using a consistent penetration depth estimation technique.

B.4 Tetrahedralization Collision

Giving up on triangle mesh based collision detection, we decided to tetrahedralize the objects for a more robust result. The technique we used is similar to the one used by Müller et al. [Müller et al. 2005]. It uses spatial hashing with tetrahedrons, edges, and vertices [Teschner et al. 2003], and consistent penetration depth estimation [Heidelberger et al. 2004]. For collision response, we tried forces proportional to penetration depth, depth squared, depth cubed, with different constant multiplications, and so on. Results were good, but stacked objects would not come to a rest, instead tending to jitter. This kind of behaviour with pure response forces is apparently typical, and more complicated methods are required to deal with it. One method for contact surface computation looked promising [Spillmann and Teschner n. d.], but we did not have time to implement it.

C Tetrahedralization

3d models are input to the program in Ogre's .mesh format, which defines a surface mesh. This is not adequate, for two reasons.

1. Surface meshes can be highly detailed, but provide no additional deformation realism over less detailed meshes. We

would like to use a low detail mesh for physical simulation, attached to a high detail visible mesh.

2. The collision detection method we use requires a tetrahedral mesh.

So we would like to create a low detail tetrahedral mesh for physical simulation and collision detection and response, attached to a high detail visual mesh for display. The tetrahedralization algorithm we used is described by Dompierre et al. [Dompierre et al. 1999]. The basic technique is to spatially partition the entity's bounding box into cuboid regions at a specified resolution. Each cube is split into 5 tetrahedrons. Then each tetrahedron that has its midpoint outside the entity is discarded. There are three further levels of optimization to create a tetrahedralization more consistent with the surface mesh, which we do not discuss here.

To implement the separation between physical and visual vertices, each entity has two separate Vertex lists: one for physical vertices and one for visual vertices. The entity's triangles index into the visual array, while its tetrahedrons index into the physical array. Similarly, each cluster has separate lists of indices into its entity's physical and visual vertices. Physics, collision detection and response, and the meshless deformation algorithm all use only the physical vertices. The only exception is that the final deformation matrix is applied to both physical and visual vertices, keeping the visual vertices "attached."

D GUI

The GUI was based on Crazy Eddie's GUI System (CEGUI) [Eddie n. d.b]. From CEGUI's website:

> Crazy Eddie's GUI System is a free library providing windowing and widgets for graphics APIs / engines where such functionality is not natively available, or severely lacking. The library is object orientated, written in C++, and targeted at games developers who should be spending their time creating great games, not building GUI sub-systems!

CEGUI natively supports Microsoft DirectX, OpenGL, Ogre3D and the Irrlicht engine [Eddie n. d.a]. We started this project using CEGUI with raw OpenGL, then later migrated without difficulty to Ogre3D.

CEGUI comes with three default widget sets (look-and-feels): WindowsLook, TaharezLook, and Vanilla. Figure 43 shows a screenshot of the Vanilla GUI skin from CEGUI's website. We decided to use a different skin though - a custom one borrowed from DefCol Studio [Heidelberger et al. n. d.]. Figure 44 shows the entity properties window we created using this skin.

To create the actual GUI layout, we used CEGUI's official WYSIWYG layout editor, CELayoutEditor. As one would expect, this program allows the user to select a GUI component, place it in a document, and graphically move it into the desired position. The position and size of each element can be given in either absolute pixel values, or values relative to the parent component, where e.g. 0.0 is the left side of the parent, and 1.0 is the right side. The user can also modify properties such as alpha transparency, border visibility, and whether a close button exists or not. Some properties are not displayed in CELayoutEditor however (e.g. a panel's background visibility). To change these properties, one has to manually edit the XML file defining the layout, or hard code the settings in the application.

Most of the GUI was made using the DefCol Studio windows, but we made our own button images for switching modes between picking, pushing and cutting.

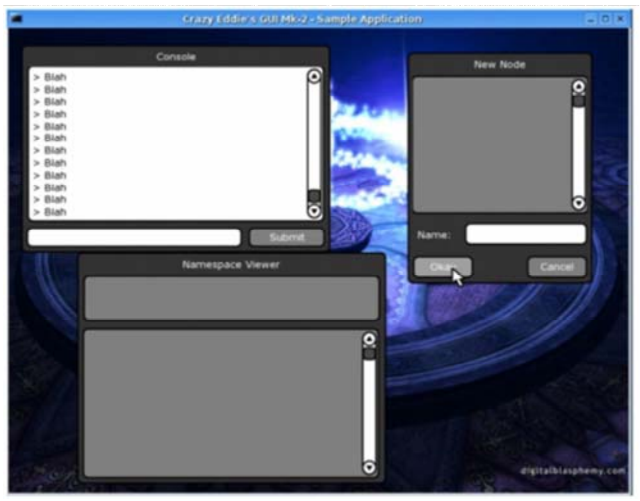
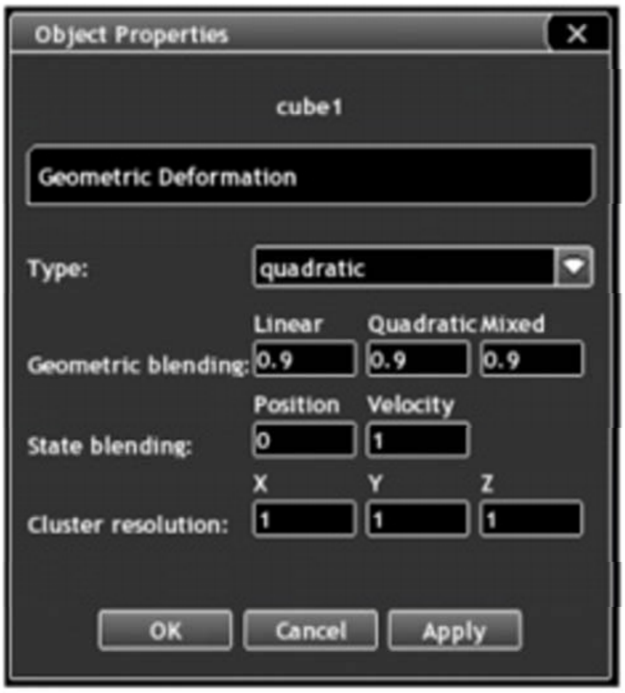


Figure 43: The Vanilla CEGUI skin.



cube1

Geometric Deformation

Type: quadratic

	Linear	Quadratic	Mixed
Geometric blending:	0.9	0.9	0.9
	Position	Velocity	
State blending:	0	1	
	X	Y	Z
Cluster resolution:	1	1	1

Figure 44: The entity properties window, created with DefCol Studio's skin.

D.1 Object-Oriented Design

To separate the GUI code from the main application code, we used the Model GUI Mediator (MGM) design pattern [Bulka 2003]. MGM has a similar goal to Model View Controller [Burbeck 1992], except it is more suited for use with off-the-shelf event based GUI systems like CEGUI. From [Bulka 2003]:

> The Model GUI Mediator pattern is actually 2 classic patterns overlaid on each other: Mediator and Observer. A mediating "view" class mediates all communication between the GUI component and model. The model communicates with the mediating view(s) using observer.

Figure 45, borrowed from the paper, shows the general structure of the design pattern.

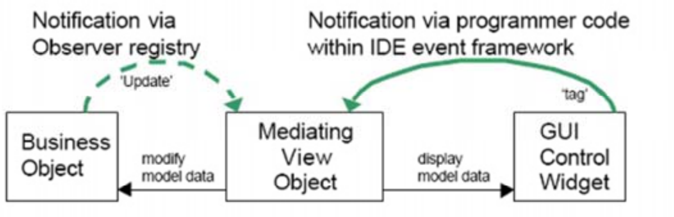


Figure 45: The Model GUI Mediator design pattern.

Implemented in our project, MainApp and DeformableEntity were the Business Objects, EntityUIMediator and GlobalUIMediator were the Mediating View Objects, and CEGUI took care of the GUI Control Widget part.

MainApp and DeformableEntity implement the interface *Model*. The *Model* interface defines a set of attributes and a set of observers. Whenever a *Model*'s attribute is changed using *Model*::setAttr(), the *Model* calls *Observer*::update() to inform each of its *Observer*s that the attribute in question has changed. (Attributes are identified by integer enumerations.) The *Observer*::update() method, implemented by a *UIMediator*, then updates the part of the GUI corresponding to the changed attribute. Figure 46 diagrams the relationships.

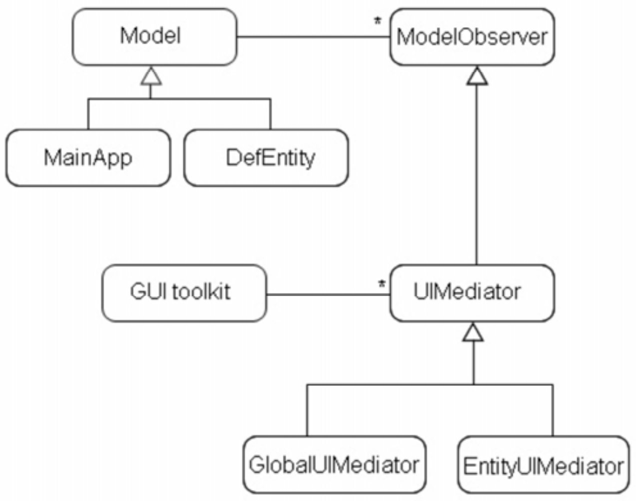


Figure 46: UML diagram of object relationships.

Conversely, if the user changes an attribute using the GUI, the *UIMediator* that has signed up to receive events from that particular GUI element, receives an event, in for example EntityUIMediator::editboxHandler(). The *UIMediator* then calls *Model*::setAttr() to change the appropriate attribute in the model. As the above paragraph explains, this then triggers the *UIMediator*'s update method. This might seem like a waste - the user has changed the GUI himself, so the *UIMediator*'s update method needn't be called to set the GUI to the same state again. Suppose though that two GUI elements display the same attribute of a *Model*, for example a slider and an edit box displaying the strength of gravity. When the user moves the slider and *UIMediator*'s event handler calls *Model*::setAttr, instead of the handler needing to update the edit box itself, every GUI element depending on the modified attribute is automatically updated in *UIMediator*::update (which is called by *Model* when the attribute in question is modified).

Thus the interaction between GUI and application is strictly limited to the following:

1. If the GUI is modified, the *UIMediator* receives an event then updates the corresponding attribute in its *Model*.

2. If a *Model*'s attribute is modified, the *Model* informs its observing *UIMediator*s, which then update the corresponding GUI elements.

As the GUI is limited to setting attributes of models, one cannot directly link the press of a GUI button to a game action. Nor can any *Model* implementations attach an action to a particular attribute being modified. This difficulty can be handled in three different ways.

1. The *Model* simply retrieves the required attribute afresh every frame. This works with for example DeformableEntity's alpha attribute. Each frame, the alpha attribute is retrieved afresh and used, thus if the alpha attribute is modified, the new value is automatically used next frame.

   The next two options address the situation where the behaviour of a *Model* implementation depends not only on an attribute's current value, but also on its previous value (or just on whether the attribute has changed or not). This kind of situation arises for example with MainApp's INTERACT_MODE attribute. If the attribute has not changed, then we do not want to deactivate and reactivate the current manipulator. So we need some way of testing whether the attribute has changed since last frame or not. In general, there are two ways to handle this:

2. We can retrieve the attribute, and test it against other variables for consistency. If the attribute is inconsistent with other variable values, the attribute must have changed, so behaviour can proceed accordingly. It might also be possible to ascertain the attribute's previous value from other variables. For example, every frame MainApp tests the INTERACT_MODE attribute. If INTERACT_MODE is set to PICK, then the current manipulator should be set to the picking manipulator. If it is not, then we know INTERACT_MODE has changed, and we further know what its previous value was, based on what the current manipulator is.

3. If 2 isn't possible – that is, if other variables do not provide adequate information as to the changes, the *Model* implementation can provide a flag attribute. For example, if the cluster resolution of DeformableEntity is changed, it has no way of knowing what the previous cluster resolution was. At best, it only knows what the total number of clusters currently is and should be. Thus each frame DeformableEntity checks a

CLUSTER_RES_CHANGED flag. If it is set, DeformableEntity regenerates the appropriate clusters, then desets the flag.