

DETC00/DAC-14489

IMPLEMENTATION AND ANALYSIS OF A MECHANICS SIMULATION MODULE FOR USE IN A CONCEPTUAL DESIGN SYSTEM

Johan Jansson

Faculty of Design, Engineering and Production
Delft University of Technology
Jaffalaan 9
NL-2628 BX, Delft
The Netherlands
email: j.jansson@io.tudelft.nl
phone: +31 15 2789321

Imre Horváth

Joris S. M. Vergeest

Faculty of Design, Engineering and Production
Delft University of Technology
Jaffalaan 9
NL-2628 BX, Delft
The Netherlands
email: i.horvath@io.tudelft.nl, j.s.m.vergeest@io.tudelft.nl

ABSTRACT

Previously, we have described the theory of a general mechanics model for non-rigid solids (Jansson, Vergeest, 2000). In this paper, we will describe and analyze the implementation, i.e. algorithms and analysis of their time complexity. We will reason that a good (better than $O(n^2)$, where n is the number of elements in the system) time complexity is mandatory for a scalable real time simulation system. We will show that, in simplified form, all our algorithms are $O(n \lg n)$. We have not been able to formally analyze the algorithms in non-simplified form, we will however informally discuss the expected performance. The entire system will be empirically shown to perform slightly worse than $O(n \lg n)$, for a specific range and typical input. We will also present a working prototype implementation and show it can be used for real time evaluation of reasonably complex systems. Finally we will reason about how such a system can be used in the conceptual design community as a simulation of traditional design tools.

NOMENCLATURE

$O(g(n))$ For a given function $g(n)$, we denote by $O(g(n))$ the set of functions
 $O(g(n))$ $\{f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$.

This is an asymptotic upper bound on $f(n)$.

$\Theta(g(n))$ For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions

$\Theta(g(n)) = \{f(n) : \text{there exists positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \}$.

This is both an asymptotic lower and upper bound on $f(n)$. $f(n) = \Theta(n)$ implies $f(n) = O(n)$.

$T(n)$ We denote a function $T(n)$ if it represents the execution time of an algorithm with an input of n elements.

INTRODUCTION

Computer aided design has had a large impact on many fields. It has grown from non-existence some decades ago, to being virtually ubiquitous, at least in industrial environments. However, while there for sure has been development of the paradigm, there still remains certain obstacles before it can truly replace traditional design, especially in the conceptual phase.

While existing CAD systems allow precise designs to be formulated, the conceptual stage is still largely unsupported. We can say that existing tools support static designs well, but do not support dynamic designs. It would be good to have computer support of the whole process, thus enabling all the advantages a computer based system offers.

Requirements for a conceptual design system have been discussed in several papers (Wiegers, et. al., 1999). The two key requirements which apply for this specific domain, conceptual shape design, can be stated as such:

1. Natural interaction methods - interaction with virtual objects should be experienced as interaction with real objects.
2. Rapid feedback and evaluation - interaction should not be hindered by poor resolution and time lags

Our approach to meeting these requirements is to create a simulation of the mechanics of a domain of reality which applies to conceptual shape design, and to execute this simulation in real time, to make it appear as if the time flow in the simulation matches the time flow of an observer.

Instead of explicitly creating mathematical models for each specific design tool and method, we create a general model in which we indirectly can model such tools and methods, i.e. shape tools from hard materials and apply them to soft materials.

While such a model may appear complex to attempt, from a mathematical viewpoint, and computationally expensive to evaluate, there are two relieving circumstances which make it relatively simple:

1. We will restrict ourselves to solids, and not incorporate liquid or gas behavior.
2. Metric accuracy is not a requirement for conceptual design systems.

STATE OF THE ART

The basic problem presented in this paper is by no means new, it has been approached by both the computer graphics, virtual reality, and structural physics communities. It has also to some extent already been approached by the industrial design community.

General simulation of the mechanics of solids has been developed mainly by and for those interested in durability studies of cars and other semi-rigid structures. A method known as the Finite Element Method (FEM) has become prevalent in such simulations (Grandin, 1986). While this method is not tied specifically to a physical model, but is a mathematical method, it has been developed for the purpose of solving certain kinds of models, and thus can be used informally as an identifier for the entire simulation method. Such physical models of solids typically employ some sort of basic element, a plate or tetrahedron, which is used to build up more complex shapes. The union of all separate behaviors and their interactions then form the behavior of the entire shape.

Kang and Kak (Kang, Kak, 1996) have employed the FEM solution method to create a CAD system. The system presents the designer with an initial physical shape, represented by an FE shape. The designer can then utilize a force-input interface, in

their case a four-sensor plate, to deform the shape. This system could presumably be generalized to allow arbitrary input methods, and thus essentially meet our requirements.

Terzopoulos, Platt, Barr and Fleischer with colleagues have attacked this problem from a computer graphics point of view (Terzopoulos, et. al., 1987). They employ an energy theory (which is also the base of FEM) to compute restorative forces on shapes, and force fields to compute collision forces. While their approach is similar to a FEM approach, they use a finite difference method to solve the systems. Their approach could also presumably be used to meet our requirements, it is a general method for approximatively simulating physical systems.

These solution approaches have much in common, but unfortunately share two important detrimental (to our goal) properties: complexity of treatment and complexity of evaluation. Complexity of treatment makes them difficult to grasp, analyze and develop further, as well as implement. Complexity of evaluation makes them unsuited for real time evaluation. Kang and Kak solved this issue by using a two-resolution FEM, a two stage process where at first a coarse shape deformation is performed, the general shape is then fixed, and the detailing stage starts. While this appears to be an excellent approach for their system, it appears not to scale well to general systems.

CONCRETE AIMS

We have outlined our abstract goal: to simulate the mechanics of traditional design methods and tools in real time. We now need to specify how to actually reach such a goal.

There will be two parts in a solution: a theoretical mechanics model and an algorithm evaluating the model. Applying the previously stated requirements, we can say that the model must be general enough to create a realistic virtual environment, and the algorithm must be able to evaluate the model in real time.

We need to define what is actually meant by "real time", or "rapid feedback and evaluation" to be able to specify these requirements. In computer systems science, "real time" means being able to guarantee that a specific task will be completed in a specific time interval. We will use a similar definition, but applicable to simulation and humans.

We will say that a simulation is "real time" if there exists a fixed ratio between simulation time flow and real time flow, and feedback update is above a fixed frequency.

For humans to be able to interact with such a simulation, this ratio and frequency must be within a certain interval. While it is not possible to quantitatively and generally find what these intervals must be, we can specify a rough estimate based on our interactions with simulators. We can say that the ratio must be above 1/10 and the frequency above 10 Hz. When we subsequently mention "real time", these numbers will implicitly be included. For optimal interaction, the ratio should be 1 and the frequency as high as possible.

While this definition of a real time requirement is enough to guarantee an interactive simulation, it is impossible to satisfy for arbitrary simulation input. Therefore, we need to specify a scalability requirement. With such a requirement, we can determine for which input range the simulation will be interactive.

Since our implementation will be an algorithm, we can simply apply the existing metric for algorithmic time scalability, we will call it asymptotic time complexity. It classifies algorithms by determining the asymptotically dominating time function. the mathematical treatment is similar to that used in Taylor series expansion.

We will say that for a simulation algorithm to be scalable enough, it must be a better algorithm than the class $O(n^2)$ of algorithms. An algorithm which is $O(n^2)$ will, for 10 times larger input, require 100 times as much evaluation time.

We have already described a theoretical model for the mechanics of non-rigid solids in a previous paper (Jansson, Vergeest, 2000). This model meets our theoretical requirements. This paper will describe how the model can be implemented, as well as show that the implementation can meet our implementational requirements. We will also go back to our original discussion in the introduction, and show how it is possible to implement traditional modeling operations in this complete system.

SUMMARY OF THEORY OF THE MECHANICS MODEL

The base of the theoretical model is the assumption that the interaction of idealized aggregates of atoms is isomorphic to the interaction of single atoms. Since there exists at least approximations of the atomic interaction forces, we can thus anchor the model reasonably well in established physics.

The model consists of two primitives; the object and the connection. The object represents an aggregate of atoms, it can be viewed as a piece of mass with a certain spherical volume, while the connection represents the forces between two such objects. We use a connection primitive instead of a globally defined force function to ease local customization of forces. Geometry is represented by a configuration of objects, and all forces, such as collision forces, internal strain forces etc., are represented by inter-atomic forces, through connections. As inter-atomic forces are negligible for large distances, we destroy a connection when two objects move too far apart. Symmetrically, we create a new connection when two objects come within the close vicinity of each other. We can also use globally defined forces when appropriate, such as for a true gravitation simulation.

Definitions

Object An object o is a set of parameters $\{ \vec{p}, \vec{v}, m, r, k, t, d, u, C \}$.

C set of connections connected to the object

d damping constant

m mass

\vec{p} position

r radius

\vec{v} position

t tolerance force

u friction constant

Connection A connection c is a set of parameters $\{ l, k, t, b, u, o_1, o_2 \}$.

b damping constant

k Hooke spring constant

l nominal length

o_1, o_2 objects comprising the connection

t tolerance force

u friction constant

Connection Forces As the friction force depends on the other components of the bond force (which define the normal force of the friction equation), we need to further subdivide the bond force. For clarity, we simply use the logical components we have already defined:

$$\vec{F}_C = \vec{F}_b + \vec{F}_d + \vec{F}_f \quad (1)$$

\vec{F}_b original bond force definition

\vec{F}_d damping force

\vec{F}_f friction force

To simplify notation, we define the subscript o as we have done before, and a new subscript p as the opposite object in the connection. We iterate over the connection set C of the object:

$$\vec{F}_b = \sum_{i=0}^{|C_o|} -k_c(\|\vec{p}_o - \vec{p}_p\| - l_c) \frac{\vec{p}_o - \vec{p}_p}{\|\vec{p}_o - \vec{p}_p\|} \quad (2)$$

We define the relative velocity of the two objects in the connections as two components, one parallel to the connection, and one orthogonal:

$$\vec{v}_{\parallel} = \frac{(\vec{v}_o - \vec{v}_p) \cdot (\vec{p}_o - \vec{p}_p)}{\|\vec{p}_o - \vec{p}_p\|^2} (\vec{p}_o - \vec{p}_p) \quad (3)$$

$$\vec{v}_{\perp} = (\vec{v}_o - \vec{v}_p) - \vec{v}_{\parallel} \quad (4)$$

Then:

$$\vec{F}_d = \sum_{i=0}^{|C_o|} -b_c(\vec{v}_i) \quad (5)$$

The sum of these two forces could be called “contact force” in certain contexts. They form the normal force in the friction definition,

$$\vec{F}_N = \vec{F}_b + \vec{F}_d \quad (6)$$

We can now define the friction force:

$$\vec{F}_f = \sum_{i=0}^{|C_o|} -u_c \vec{F}_N \frac{\vec{v}_i}{\|\vec{v}_i\|} \quad (7)$$

IMPLEMENTATION STRUCTURE

The main advantage of our model is its simplicity. There exists only two primitives, orthogonal to one another. All actions occur through forces, there are no assumptions made about energy minimization, which normally form the basis of similar methods, such as FEM etc.

We need to keep one set of objects, and one set of connections. The connections form a graph over the objects. The configuration of the system forms the initial state. As time is necessarily discretized by numerical solution, there exists a calculation cycle:

1. Perform vicinity calculation and create/destroy connections.
2. Calculate connection forces on the objects, also globally defined forces if they exist.
3. Calculate new positions according to the forward Euler method.
4. Start over.

ALGORITHMS AND TIME COMPLEXITY ANALYSIS

We can identify two calculations in our cycle (if we disregard global forces): connection force calculation and vicinity calculation.

The algorithm for calculating the connection forces is trivial. If we look at the connection force definition, we see that it is only dependent on the attributes of the two objects the connection

consists of. Since we maintain the set of connections, we can simply iterate through the set once, and calculate the forces. Thus our connection force algorithm is $O(n_c)$, where n_c is the number of connections in the system.

The vicinity calculation however, is more complex. It is equivalent to a collision detection calculation for a system of spheres. We have a set S of spheres, and want to find all pairs which intersect.

The brute force algorithm compares every sphere with every other sphere. It iterates through S once, and for every sphere iterates through S again. Thus this algorithm is $O(n_s^2)$, where n_s

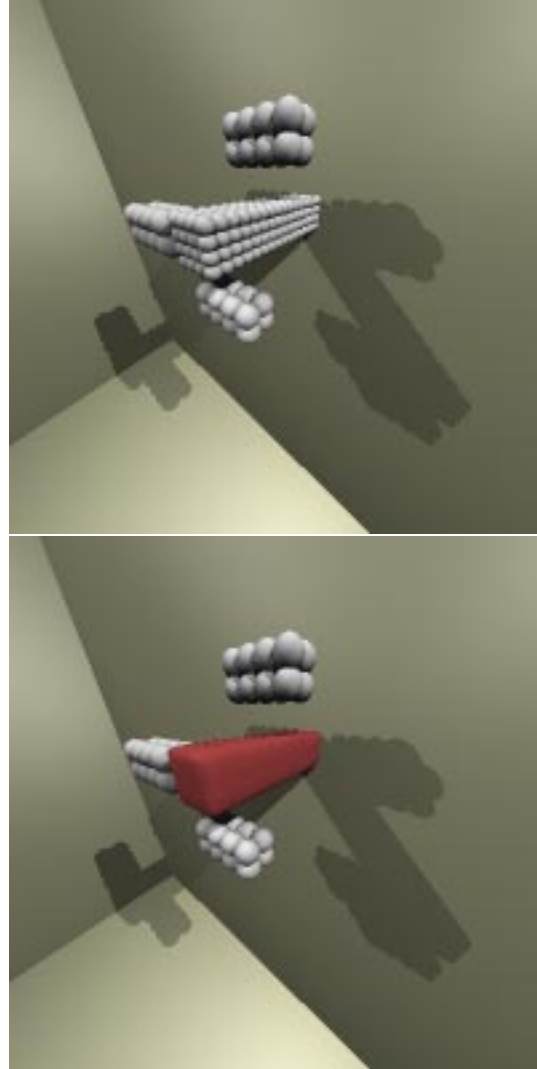


Figure 1. Here we can see our material, which is a long bar fixed at one end to a wall, and our manipulators. Since we initially know the appearance of the shape, we can also create a boundary representation which we can display.

is the number of spheres in S . As we have stated, this is not good enough. We need to find a better algorithm.

A standard approach toward problems such as this, i.e. visibility problems, collision problems, etc. employ hierarchic bounding space partitioning. This means that all the relevant geometry is recursively partitioned into a tree structure. One test at the root node is then enough to disregard a significant part of the space, and thus a significant part of the geometry. This then recurses until a leaf is reached, which either contains geometry, or is empty. The algorithm thus consists of two parts: construction of the partition tree, and examination. While an n -ary tree could be used, such a structure could be represented as a binary tree. Therefore, to simplify treatment, we will only consider only a binary tree.

Our case however, is somewhat special. Mathematically, the term partition means dividing a set into pairwise disjoint subsets, of which the union forms the original set. It is this property which allows the method to perform correctly. For some configurations of geometric entities with a spatial extension (i.e. not points), it is impossible to find a partitioning plane which does not intersect the entities. This is normally solved by dividing the particular entities into two. This is not a tempting solution for our problem however, as it will destroy the ease of treatment of spheres. Instead, when a sphere spans a partition plane, we add it to that node, so that leaves will not be the only holders of geometry. Thus we have violated the partition definition. If we take this into account when we examine the tree however, we can still enforce correctness; we need to treat spanning spheres as a special case, and the rest of the spheres as truly partitioned.

CONSTRUCTION The construction algorithm needs a set S of spheres, which it will construct a tree from. At every node, it partitions S around a partition plane, and thus constructs two new sets, S_1 and S_2 . All spanning spheres are added to the node, as described. It then recurses twice, with S_1 and S_2 as the S of the children. For simplicity's sake, we partition the set fully, so that every leaf either is empty, or contains exactly one sphere. Every sphere exists only once in the tree, either in a leaf, or if it spans, in an internal node.

There are many ways a partition plane could be selected. However, as the partitioning method will affect the height of the tree, and thus potentially the time complexity, it is imperative that we partition correctly. If we partition naively, we could imagine a pathological case where we get a linear tree, and thus will have accomplished nothing with our algorithm. Instead, we have to try to find a method of finding a good partition plane, which guarantees a good tree. The optimal partitioning would split every S in exactly half, thus making the tree complete, and thus guaranteeing a tree height of $\lg n$.

Before we consider how we could achieve such a method, or a similar good enough method, we have to consider how much

time we are allowed to spend on the partitioning. We know that the complexity of partitioning an unsorted set around a known plane is $\Theta(n)$. If we can match this with our imagined optimal method, our construction algorithm will have a time complexity of $O(n \lg n)$, which is very good. Fortunately, there exists a median selection algorithm, which thus would suit our purposes, and which is $\Theta(n)$. It is based on quicksort, and is sometimes called quickselect (Cormen, Leiserson, Rivest, 1998).

Since our tree will be complete, time complexity analysis will be simple. The only thing which could cause potential problems are the spanning spheres. However, to retain simplicity, we can still imagine that they are also inserted into the appropriate partition, as tokens. We will during execution ignore these tokens, but take them into account during this analysis.

Since we have a complete tree, and we split the number of elements in each node exactly in half, we have the familiar recurrence:

$$T(n) = 2T(n/2) + \Theta(n)$$

Which by case 2 of the master theorem is $\Theta(n \lg n)$ (Cormen, Leiserson, Rivest, 1998).

EXAMINATION The examination algorithm will test one particular sphere s against the tree. It will return all spheres which intersect with s . We begin at the root node of the tree. If the node is a leaf, it is either empty, and thus no collision has taken place, or it contains exactly one sphere, which we test s manually against. If the node is not a leaf, it contains a plane which partitions its space. Depending on which side of the partition plane s is, we select the appropriate child and recurse down it. If s spans the partition plane, it exists in both partitions, so we must recurse down both children. Finally, if the node contains any spanning spheres, we test s manually (with brute force) against them.

For this time complexity analysis, we will disregard the two spanning possibilities. Later on, we will instead reason about how often they happen.

Time complexity of the non-spanning s examination algorithm is dependent on the height of the tree; it starts at the root, and follows children until it reaches a leaf. Since we have shown that the height of the tree is $\lg n$, the time complexity of this algorithm is thus $\Theta(\lg n)$, or $\Theta(n \lg n)$ for testing all spheres against the tree.

SPANNING We have made two simplifications, which are really one. We have assumed that any given sphere will never span a partition plane. It is trivial to show this is not true; assume that all spheres share the same position and radius, all spheres will thus span the first partition, and we will end up with only

one set, which will result in our original brute force algorithm. It is difficult to make a rigorous proof of an average case, at least we have not been able to do so. We can instead reason about what normal input will look like, and how likely spanning will be.

We select the orientation of the partition planes according to a straight cycle through the axes. This means that every node in the tree will be reasonably cuboid.

Normal input will most likely consist of several aggregates of spheres, representing disjoint shapes. We might have several rigid tools, and several working materials. This means that the considered space will be mostly empty. A partitioning of this space can then put all the different aggregates in differing children at various levels, and thus a collision test between spheres of the same aggregate will be able to disregard the other aggregates. Even such a self-collision test need not include many spanning spheres. The distance between spheres in an aggregate will normally be large enough so that the spheres do not intersect, or barely intersect. This means that a cubic aggregate for example, can safely be partitioned into two, with only one slice of it spanning the plane. We can also see that a collision between two aggregates can be seen as a self-collision between one larger aggregate, and thus the same discussion will apply for such a case.

Unfortunately, this is not completely true for the partitioning algorithm we have described. While partitioning at the median sphere is theoretically appealing, in practice it will avoid all empty space, and actually guarantees the partition plane cuts through spheres every time! This need not be fatal, but will degrade performance.

We have also implemented a variation of the partitioning algorithm, where we simply partition space into half at every cut, while still cycling orientation of the plane. This will result in essentially a simulation of an octree. This variation has produced better results, at least for our test data. It is however difficult to treat this theoretically, as the geometry of the tree will be completely input-dependent.

We will later show some timing results of various inputs, where we will see that while time growth is worse than $O(n \lg n)$, it is not much worse, and at least initially much better than $O(n^2)$. We will also see that the "half-partition" algorithm performs closer to $O(n \lg n)$ than the "median-partition" algorithm.

TYPES OF ELEMENT Since we have two types of element, of which one, the connection, is dependent on the other, the object, we need to consider how this dependence behaves. Again, we can contrive a worst case, where all objects share the same position and radius, which will mean all objects intersect all others, and we will get n_o^2 connections, where n_o is the number of objects. However, this will not be the norm. When constructing aggregates, an object should only initially be connected to its immediate neighbors, mimicking how atomic lattices are con-

structed in nature. This means that a given object can only initially be connected to a constant number of other objects, resulting in a linear dependence between the number of connections and objects.

We now have collisions left to treat. While it is possible that a system can degenerate into our pathological case we have described, or a similar case, by being significantly compressed for example, it would be a very rare occurrence. Instead, we can reason as we have before, that a collision will simply produce a larger aggregate, while still reasonably maintaining the "lattice property".

PRACTICAL IMPLEMENTATION NOTES Normally when a hierarchic space partitioning structure is used, it is created to represent a static system, and one active agent which examines the system several times. Thus the construction of the tree is allowed to be reasonably expensive, while the examination needs to be cheap. In our case, the system is dynamic, so to use a method such as this, we will need to recreate the tree every time the system changes, i.e. once every cycle.

We have seen that the time complexity for construction of the tree is at least as good as that for the examination. This is slightly deceptive however, as we have not considered space usage, or space complexity. Allocating space in the computer memory is a very expensive operation, so since the construction algorithm allocates space for every node, in addition to performing some arithmetic operations, it should be a much slower algorithm than examination.

This must not necessarily be the case however. Normally we would create a tree, destroy it when we are finished, and then recreate it again the next cycle. This is not a very efficient solution. Instead, we can create a pool of nodes. We redefine our node allocation algorithm to first check if there are any free nodes in the pool, and only if not, really allocate. Symmetrically, instead of destroying a node when we're finished with it, we put it in the pool. This will mean the pool will at most contain the number of nodes of the largest tree we have yet created.

While this is not an issue when using static memory allocation schemes, where all memory for the program is allocated in advance, it is a very useful method when using dynamic memory allocation. The difference between the two schemes is that with dynamic memory allocation, one need not know the upper bound of used memory in advance.

PARALLELIZATION

Creating algorithms with good time complexity is a necessary first step, but this in itself may not be enough to achieve the execution speed needed. We must also make sure the constants hidden by the O notation are as small as possible. This can be achieved partly by good programming, but an additional effective method is parallelization. If an algorithm can be parallelized

effectively, and there exists good enough hardware solutions, we can halve the execution time of a particular algorithm simply by doubling the amount of processors in the computer hardware.

To be able to parallelize an algorithm, we need a way of partitioning the calculations. The most straightforward way is to partition the data set, and simply apply the algorithm to the partitions separately. This is not always possible however.

Parallelizing the connection force calculation is straightforward. We partition the connection set into p partitions, and then apply our connection force algorithm to them all in parallel.

Since the examination computation depends on the space partitioning computation, they cannot be computed in parallel, but must be done sequentially. However, they can in themselves be parallelized.

Parallelization of the space partitioning can be done in at least two ways. As partitioning is already inherent in the tree structure, it is possible to simply use that partitioning. However, this presents an unnecessary artificial restriction on partition size balance. We have instead chosen to partition the object set (which has been represented as S , the set of spheres in the space partitioning discussion) independently for this purpose, thus allowing an arbitrary partition size. So we simply partition S into n subsets S_1 to S_n , and apply space partitioning to each separately to produce trees T_1 to T_n .

Examination is done in a similar fashion. We must use the same subsets S_1 to S_n , and perform parallel examination of these sets against trees T_1 to T_n . For every s in a subset S_i , we perform examination on trees T_i to T_n . This is because examination of s on trees T_1 to T_{i-1} has implicitly been performed by the examination of subsets S_1 to S_{i-1} on trees T_1 to T_{i-1} . This will not perform more examinations than a non-partitioned examination, what we are doing is partitioning the calculations, and not only the input.

While no timing results for parallel vs. sequential performance have been included in the paper, we have made some rough timings indicating a 70% performance increase, with a 10% idle time for execution on two processors vs. one, on our specific system, partitioning the data sets into two. This can be considered quite good, as there is necessarily some locking overhead, as well as memory access congestion.

IMPLEMENTING MODELING OPERATIONS

The most common and general modeling operation we can identify is arbitrary deformation. Given a specific shape, we want to deform it into some other, topologically homeomorphic shape. Bending or compressing a piece of clay with our hands is an example of such an operation.

Thus, we can construct our first simple modeling scenario. We will have one or more more or less soft shapes, which will represent our material, as well as several semi-rigid shapes, which will represent our hands or fingers.

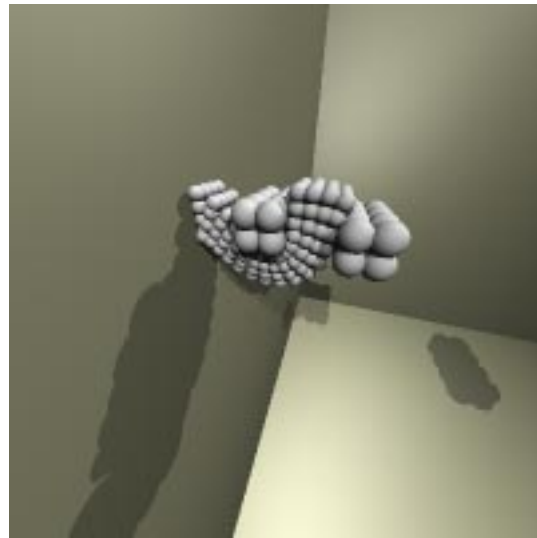


Figure 2. The manipulators are applied for a bending operation.

This scenario can be directly implemented in our system. However, since we, during development of this system, have not had access to advanced enough input devices to be able to sample hand/finger positions, and perhaps even give force feedback, we have been forced to make a simplistic representation. We instead introduce three very basic manipulators into the scenario. They are simple blocks which are navigable through keyboard input (See figure 1). We can convert b-rep shapes into a representation in the system, thus creating the initial shape of the material. We can also artificially fix a certain part of the shape, as it is difficult to fix it with our manipulators, while also deforming it. Thus we can create arbitrary deformations, by manipulating an imported shape with our "hands" (See figures 3 and 2). To further ease modeling, we have created an artificial operation, which we can call "re-normalization". We create an event, that when triggered, sets the nominal length of all connections to the current length. This would correspond to something like heating rubber, and then letting it cool, i.e. some sort of recasting into the current shape.

With real input devices, we could simply introduce an aggregate of objects which covers the volume of the sampled hand, and move it around artificially. It would thus follow the path of the real hand, while still being able to feedback forces, as the individual objects of the aggregate would still know what force was applied on them. Force feedback, at least optimal such, requires a very high update frequency however, so this may be a goal not achievable in some years. One-way input appears to be implementable today. We have done some preliminary testing using a 5 DOF digitizing arm as input, and have achieved real time interaction. This will have to be further investigated.

There are two other common operations which also can be

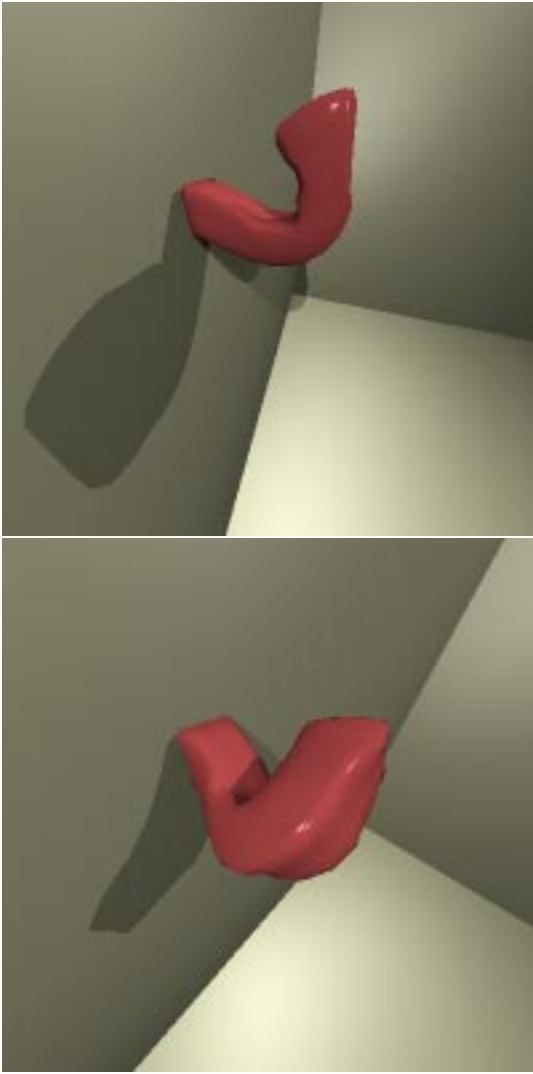


Figure 3. By applying tools, we can perform arbitrary deformations on the material. We are however limited by the simplicity of our manipulators, and the interface.

directly implemented. These are adhesion and fracturing. Since we have full control over the parameters of each individual object and connection in the system, we can describe aggregates which are prone to fracturing, or prone to stick to other aggregates. These are not artificial operations in any way, but are properties of the inter-atomic forces. We could however also create artificial entities in the system which could ease these operations. We could for example create an artificial glue aggregate, which is simply static, but makes any objects which touch it more adhesive, etc (See figures 4 and 5 for a fracturing demonstration).

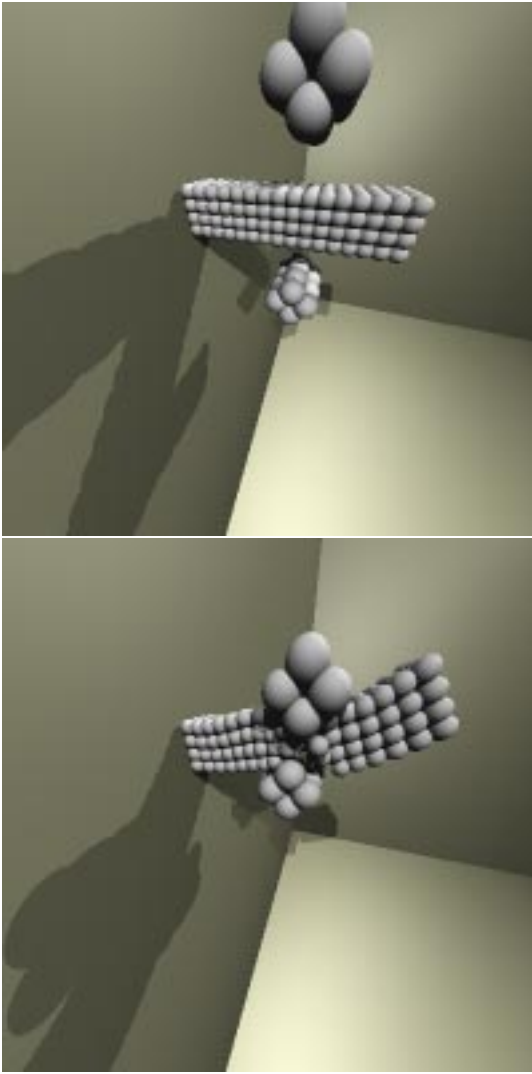


Figure 4. We can perform cutting by clamping a piece of the material, thus creating stress between objects pushed aside by our clamping.

RESULTS

We have performed a suite of experiments, evaluating the time growth of the entire system, with the exception of the visualization subsystem. We created a test data set based on an aggregate consisting of 425 objects and 3883 connections. We started with only one aggregate for the first data point, and then added more for the following. The aggregates were configured irregularly in space, and given differing velocities. We visually verified that the configuration went through both uniform and ununiform spatial distribution during the interval, to ensure that no particular best or worst case was prevalent. We collected 30 timings for every data point, and calculated an average and an uncertainty. We performed such a suite for both the "median-

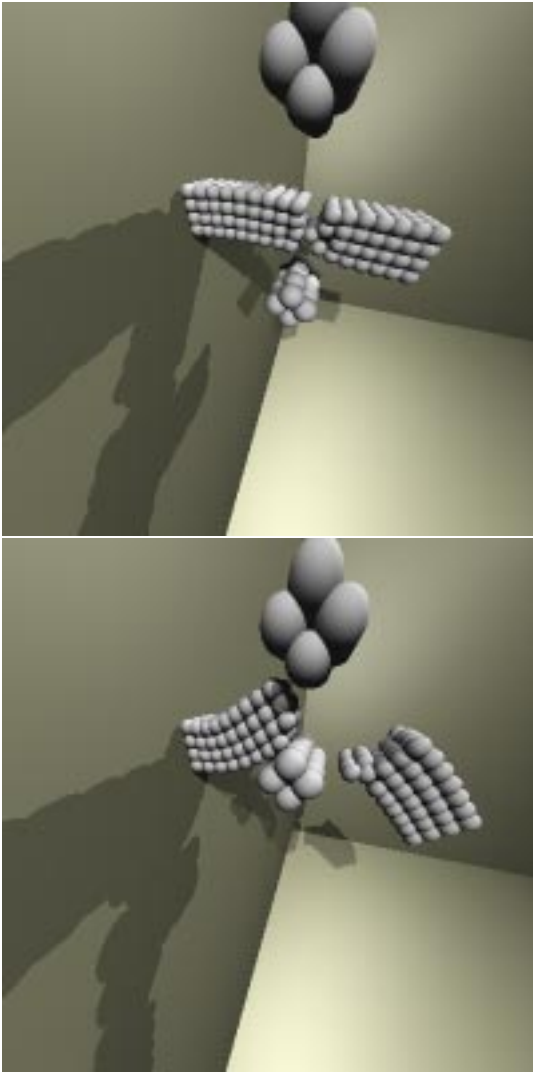


Figure 5. We have now clamped the material. The first image shows a failed attempt, with only partial severing. The second image shows a successful severing.

partition” algorithm and the ”half-partition” algorithm. All tests were performed on a PC-AT computer with two Intel Celeron processors at 450 MHz, running BeOS R4.5.

In our graphs (See figure 6), we have plotted our timing results as well as two reference functions: $g(n) = an^2$ and $h(n) = bn \lg cn$. We have fit the functions to (0,0) and the first data point (roughly, both algorithms almost share the same point) by selecting the appropriate constants. Since $h(n)$ has two constants, it is not possible to make a unique fit, but we have chosen fitting constants so that the curve does not interfere with the other data. These curves are intended as a growth reference, and not as an absolute measurement reference.

It is not possible to say anything conclusive about asymptotic time complexity from this experiment, since we cannot produce infinite data points, nor try all different inputs. We can however note tendencies in the growth, which at least hold for roughly the interval and type of input we’ve examined. We can see that the half-partition algorithm has a better growth than the median-partition algorithm.

We can use $f(2n)/f(n)$ as a simple metric. For large n , any $\Theta(n^2)$ function should produce a value slightly below 4, while a $\Theta(n \lg n)$ function should produce a value slightly above 2. We can make a rough interpolation of the graphs, and compare $f(7000)/f(3500)$ for both graphs. For the half-partition graph, we have roughly $9/4 = 2.25$, while the median-partition graph produces roughly $12/4 = 3$.

Due to the way we handle partition spanning, we know we have $\Theta(m^2)$ components in our time function, where m is the number of spanning objects of a partition plane (spheres in our previous discussion). If m/n (n is the total number of objects) is constant, the algorithm will be $\Theta(m^2)$. It is difficult to say anything about m/n however. We need to create a statistical model of a set of spheres with an arbitrary (or somehow constrained) spatial configuration, and see how many spheres we intersect with a partitioning plane in the average case. We have not been able to create such a model however.

Aside from a time complexity requirement, we had a fixed requirement for the ratio of simulation time to real time and the feedback update frequency. We required the ratio to be above 1/10 and the frequency to be above 10 Hz. As 425 objects is enough for a complex enough simulation (all shown samples use less objects than this), we can analyze this data point. The computation time was ca. 0.22s for 10 Euler iterations, which covered 0.034s of simulation time. Our ratio is roughly 0.15, which meets the requirement. 0.22s per update gives us a feedback update frequency of ca. 4.5 Hz, which does not meet our requirement. Given twice as fast hardware, which is affordable today, this second requirement can easily be met however.

CONCLUSION

We have described an implementation of a mechanics model for solids, as well as a prototype application of it in a conceptual design system. We have shown several examples of how to implement modeling operations on top of this model. We have also demonstrated that real time evaluation and interaction of relatively complex simulation configurations are possible with existing, affordable hardware.

In our algorithm analysis, we have reached the conclusion that vicinity calculation is the most dominant task. We have examined two variants of a hierarchic binary space partitioning algorithm, one which partitions space at the median of all considered objects, and one which simply partitions space in half. Unfortunately, we have not been able to state a theoretical bound

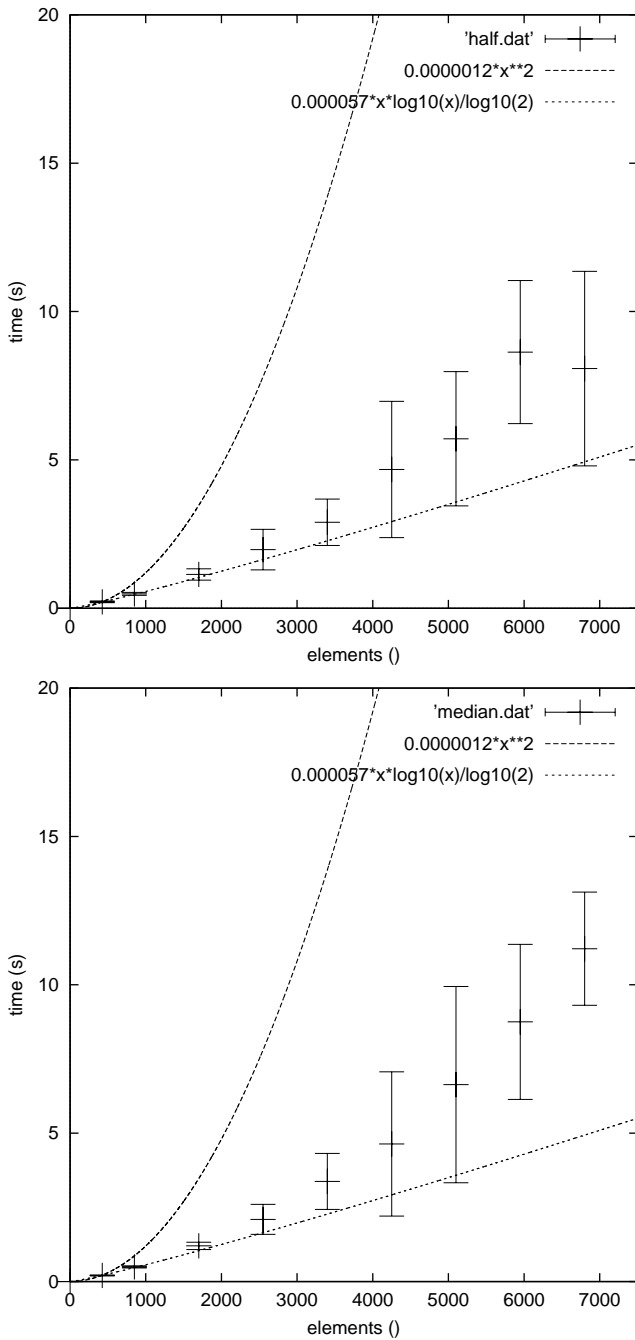


Figure 6. Graphs of the timing results for the algorithms.

on either of these algorithms, but have only been able to analyze simplified versions, and make informal discussions. However, we have also performed an empirical experiment, showing that the "half-partition" variant appears to perform slightly worse than $\Theta(n \lg n)$, and the "median-partition" variant somewhere in between $\Theta(n \lg n)$ and $\Theta(n^2)$, for at least the tested interval and

data sets.

While both algorithms appear to be usable for our interval, and perhaps further, it would be good to find an algorithm with a strict theoretical bound significantly better than $O(n^2)$ for the average case. For our specific problem, collision detection of spheres, this might prove a hard task however. It would require a statistical representation of the average distribution of a configuration of arbitrarily sized spheres. Even so, it is not certain such an algorithm exists.

The design-related conclusions we can draw are that a system such as this appears to be a feasible conceptual design system approach. We have been able to implement several modeling operations, operated through a, theoretically, very natural interface. Further research will have to show possible weaknesses, and if it can be practically used for real design tasks.

ACKNOWLEDGMENT

This research has been performed as part of the Integrated Concept Advancement (ICA) project, at the Delft University of Technology.

REFERENCES

- Cormen, H., Leiserson, C., Rivest, R., *Introduction to Algorithms*. MIT Press (1998).
- Fuchs, H., Z. M. Kedem, and B. F. Naylor, "On Visible Surface Generation by A Priori Tree Structures". *SIGGRAPH 80 (1980)*, pp. 124-133.
- Grandin Jr., H., *Fundamentals of the Finite Element Method*. Macmillan Publishing (1986).
- Hollerbach, J. M., Cohen, E., Thompson, W., Freier, R., Johnson, D., Nahvi, A., Nelson, D., Thompson, T. V. "Haptic Interfacing for Virtual Prototyping of Mechanical CAD Designs". *Proceedings of the ASME Design Engineering Technical Conferences (1997)*.
- Jansson, J., Vergeest, J. S. M., "A General Mechanics Model for Systems of Deformable Solids". *Proceedings International Symposium On Tools and Methods for Concurrent Engineering 2000 (2000)*.
- Kang, H., Kak, A., "Deforming Virtual Objects Interactively in Accordance with an Elastic Model". *Computer Aided Design (1996)*.
- Terzopoulos, D., Platt, J., Barr, A and Fleischer, K. "Elastically Deformable Models". *SIGGRAPH Vol 21 (1987)* pp. 205-214.
- Wiegers, T., Horvath, I., Vergeest, J. S. M., Opiyo, E. Z., Kuczog, G., "Requirements for highly interactive system interfaces to support conceptual design". *CIRP99 (1999)*.