Datalogisk Institut
KØBENHAVNS UNIVERSITET

Contact Graphs in Multibody Dynamics Simulation

Kenny Erleben

Technical Report no. 2004/06
ISSN: 0107-8283
CR subject Classification I.3.5 I.3.7

DIKU
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

Contact Graphs in Multibody Dynamics Simulation

Kenny Erleben
Department of Computer Science
Copenhagen University
Universitetsparken 1
DK-2100 Copenhagen
kenny@diku.dk

Technical Report DIKU-TR-04/06

Abstract

In rigid body simulation contact graphs are used detecting contact groups. Contact graphs provide an efficient underlying data structure for keeping information about the entire configuration and in this paper we extend their usage to a new collision detection phase termed "Spatial-Temporal Coherence Analysis". This paper will review contact graphs and demonstrate the performance impact in a typical constraint based multibody simulator.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation;

Keywords: Multibody Dynamics, Contact Graphs, Contact Groups, Contact Analysis, High Level Control

1 Introduction

Historically contact graphs are used for splitting objects into disjoint groups that can be simulated independently. Contact graphs are frequently mentioned between people working with rigid body simulation, as can be seen by searching through the archives of comp.graphics.algorithms, but they are often not formally described in the literature, for instance [Mirtich 1998] uses the word "contact group" but nowhere is it explained. Most of the time people just mention the idea of using contact groups to break down contact force computations into smaller independent problems [Anitescu et al. 1998; Coutinho 2001]. The benefit of doing this is so obvious and not many people would spend a lot of time on explaining it. To our knowledge [Mirtich 2000] is the first advanced attempt on using contact groups for other things than contact force computations, and the first use of the word "graph" appeared in [Hahn 1988], where a contact graph is used to properly back-up penetrating objects in the simulation, in our opinion this is the first example of a primitive time-control algorithm using contact graphs. Recently [Guendelman et al. 2003] developed a shock propagation algorithm for efficient handling of stacked objects, which uses a contact graph. The contact graph in [Guendelman et al. 2003] is constructed in a different manner than described in this paper.

Today simulators do exploit contact groups for breaking down the computations into smaller independent problems, for instance the Open Dynamics Engine (ODE) (v 0.035) and Vortex (v 2.0.1) from CMLabs compute contact groups, which they call islands and partitions respectively, however they do not store an actual graph data structure as the one we propose in this paper.

Alternatives to contact graphs are not very surprisingly neither mentioned or talked about, the closest thing to an alternative appears to be putting the contact-matrix into block-form as briefly described in [Barzel 1992]. This is again as far as we know an idea that is not well described in the computer graphics literature. In comparison with the contact graph approach the "block-form" matrix approach is limited to contact force and collision impulse computations and can not be used for anything else in the simulator.

Lastly we feel that contact graphs are a good companion for our rigid body simulator module design see [Erleben 2001; Erleben and Sporring 2003] and as such they are a step further in the direction toward a more standardized and powerful framework.

The contact graph algorithm we present in this paper is part of the Spatial-Temporal Coherence (STC) analysis module. The algorithm shows that STC analysis is scattered in between the other phases of the collision detection engine. We use contact graphs for caching information, such as contact points. The cached information can be used for to improve run time performance of a rigid body simulator. Several speed up methods are presented, these fall into two categories, the first is real speed-ups due to improvements of simulation algorithms, the second is due to changes of the properties of the mechanical system, which alters the physical system, but still produces plausible results. Our main focus is computer animation and not accurate physical simulation.

2 The Contact Graph

A contact graph consists of a set of nodes, where a node is an entity in the configuration, such as a rigid body or a fixed body. However a node can also be a virtual entity, that is something which does not have a physical influence on other entities in the configuration. For instance trigger volumes, i.e. volumes placed in the physical world, which raises events, when other "physical" objects move in and out of them. Nodes could also be something without a shape, for instance a timer event. Another example of a shapeless node could be logical rules, such as grouping of configuration objects and/or filters.

The node types are easily divided into three categories: Physical nodes, Container nodes, and Logical nodes. Table 1 the node types and their respective categories together with the symbolic notation we use. The physical nodes are those nodes representing entities which can physical interact with each other. Rigid bodies, fixed bodies, scripted bodies (see [Erleben and Henriksen 2002]) and link bodies are

Category	Type	Symbol
Physical	Rigid Bodies	(R)
	Fixed Bodies	(F)
	Link Bodies	(L)
	Scripted Bodies	(S)
Container	Composite bodies	(C)
	Multibodies	(M)
Logical	Logical Rules	(A)
	Trigger Volumes	(V)
	Timers	(T)

Table 1: Node types.

all physical objects. Rigid bodies can be rigidly attached to each other to form a composite body, a composite body is therefore a "container" type. Link bodies and joints can form a jointed mechanism, which is called a multibody or articulated figure, a multibody is therefore also a "container" type. Logical rules, trigger volumes and timers are all nodes which do not have a physical meaning, they are used for generating events, which have logic consequences in the sense that an end user reacts to them, and constraining physical interactions to a specified set of objects. We call all such kind of nodes logical nodes.

When objects interact with each other, contact information are usually computed and cached. It is particular easy to use the edges in the contact graph for storing information of interactions between objects. Edges are also useful for keeping structural and proximity information.

All the edge types are listed in Table 2. An edge between

Category	Type
Logical	Rule vs. Rigid
	Rule vs. Fixed
	Rule vs. Scripted
	Rule vs. Link
Structural	Rigid vs. Composite
	Link vs. Multi
Geometrically	Trigger vs. Rigid
	Trigger vs. Fixed
	Trigger vs. Scripted
	Trigger vs. Link
Physically	Rigid vs. Rigid
	Rigid vs. Fixed
	Rigid vs. Scripted
	Rigid vs. Link
	Fixed vs. Link
	Scripted vs. Link
	Link vs. Link

Table 2: Edge types.

a logical rule and a physical object, means that the logical rule applies to the physical object. This sort of edge is static in the sense that it is defined by an end user prior to the simulation.

An edge between a composite body and a rigid body tells that the rigid body is part of a composite body. This kind of edge give us structural information about how the composite body is build, and it is a static edge i.e. defined prior to simulation by an end user

An edge between a multibody and a link indicates that the link is part of the multibody. Again the edges are static giving us structural information about a multibody.

An edge between a physical object and a trigger volume indicates that the physical object has moved inside the trigger volume. This kind of edges can therefore be used to generate trigger volume event notifications. This type of edge is a dynamic edge, meaning that it is inserted and removed dynamically by the collision detection engine during the simulation.

The last type of edges we can encounter are those which tells us something about how the objects in the configuration currently interact with each other. For instance if two rigid bodies come into contact, then an edge is created between them. There are some combinations of edges, which do not make sense such as an edge between two fixed bodies.

In order to access cached information unambiguisly and fast. Nodes and edges must be found in constant time, and edges are bidirectional and uniquely determined by the two nodes they run between. These properties can be obtained by letting every entity in the configuration have a unique index, and letting edges refer to these indices, such that the smallest indexed entity is always known as A and the other as B.

Figure 1 contains a pseudo-code outline of the contact graph data structure. All objects should be inherited from

```
Class Node
  int idx
  Enum {...} type
  List<Edge> edges
Class Edge
  int idxA
  int idxB
Class ContactGraph
  Hashtable<Node> nodes
  Hashtable<Edge> edges
```

Figure 1: Contact graph data structures.

the Node class such that they can be inserted directly into the contact graph.

It is fairly easy to visualize a contact graph. In Figure 2 you can see a small complex example of a contact graph.

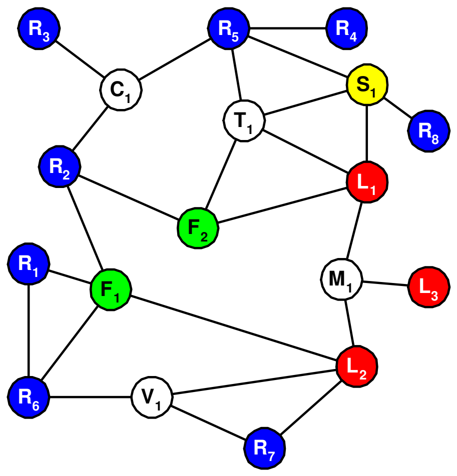

Figure 2: A contact graph example. Symbolic notation is listed in Table 1.

3 The Contact Graph Algorithm

We will now outline how a contact graph can be used in the collision detection pipeline. Notice that although we claim a contact graph to be a higher order contact analysis phase, it is not a phase that is isolated to a single place in the pipeline. Instead it is spread out in between all the other phases, i.e. in between the broad phase, narrow phase and contact determination modules.

In the following subsections we will walk through what happens in the collision detection pipeline step by step.

3.1 Edge Insertion and Removal

The first step in our algorithm is to update the edges in the contact graph, which is done by looking at the results of the broad phase collision detection algorithm. The results of the broad phase collision detection algorithm are an unsorted list of pairs of nodes, where each pair denotes a detected overlap in the broad phase algorithm. Observe that each pair is equivalent to a contact graph edge. We can therefore insert new edges into the contact graph, which we have not seen before. At the same time we can handle all close proximity information, that is detection of vanished, persistent, and new close proximity contacts. This is done by comparing the state of edges with their old state. The pseudo-code in Figure 3 outlines the general idea. The notation e_{old} in

```
$O$ = BroadPhase.getOverlaps()
for all edges $e \notin O$ do
  mark $e$ as vansihed close proximity
  if $touching(e_{old})$ then
    mark $e$ as vansihed touching contact
  end if
  if $obselete(e)$ then
    remove $e$ from $graph$
  end if
next $e$

for all $e \in O$ and $e \in graph$ do
  mark $e$ as persistent close proximity
next $e$
for all $e \in O$ and $e \notin graph$ do
  create edge $e$ in $graph$
  mark $e$ as new close proximity
next $e$
```

Figure 3: Edge insertion and removal.

Figure 3 refers to the "state" of the edge in the previous iteration, it is not a new instance.

The main idea behind removing edges is to avoid the case of edges accumulating to $O(n^2)$ size, so when the nodes between an edge are far apart, and it is unlikely that they come close in the near future, then it is "safe" to remove the corresponding edge. We call this obsolete testing.

In our simulator we apply an heuristic approach to the obsolete testing, by simply requiring that the orthogonal distance along the axes between the $AABB$s of the corresponding nodes, be twice the maximum edge size of the $AABB$s. We favor this test, because it is computational inexpensive, since it only uses a couple of subtractions and comparisons.

3.2 Logical and Coherence Testing

We can perform logical testing and exploit caching, by scanning through all the reported overlaps and remove those overlaps, we do not have or want to treat any further. In this phase logical rules are applied. any kind of logical construct could be used such as: "Ignore all interactions between objects in group X and/or group Y", .

Overlaps with passive objects are also removed, passive objects do not really exist in the configuration, they are merely objects kept in memory in case they should be turned active later on. In this way objects can be preallocated, and there is no penalty in reallocating objects that dynamically enter and leave the configuration during runtime. We refer to objects using the passive/active scheme as being light weighted. The opposite is called heavy weighted and it means objects are explicitly deallocated and reallocated, whenever they are added or removed from the configuration. One drawback of light weighted objects is that there is a penalty in the broad phase collision detection algorithm. Fortunately broad phase collision detection algorithms often have linear running time with very low constants, so the penalty is negligible.

The last screening test is for change in relative placement. Every edge stores a transform, xform$(\cdot)$, indicating the relative placement of the end node objects. If the transform is unchanged then, there is no need to run narrow phase collision detection nor contact determination, because these algorithms would return the exact same results as in the previous iteration. This is illustrated in Figure 4. Notice

```
$N$ = empty set
for each $e \in O$ do
  if not $rule(e)$ then
    continue
  else if $A(e)$ and $B(e)$ are triggers then
    continue
  else if $A(e)$ or $B(e)$ is passive then
    continue
  else if not $xform(e)$  is changed then
    if $touching(e)$ then
      mark $e$ as persistent
    end if
    if $penetration(e)$ and $ShotCircuit$ then
      terminate
    end if
    continue
  end if
  add $e$ to $N$
next $e$
```

Figure 4: Logical and Coherence Testing.

that when objects relative placement is unchanged, then it is tested if objects are in a persistent touching contact.

3.3 Narrow Phase and Short Circuiting

We are now ready for doing narrow phase collision detection and contact determination on the remaining overlaps. Output from these sort of algorithms are typical a set of feature pairs forming principal contacts, PCs and a penetration state. The contact graph edges provide a good place for storing this kind of information. The output of the narrow phase should of course also be cached in the edge, because most narrow phase collision detection algorithms reuse their results from the previous iteration to obtain constant time algorithms. Sometimes the closest principal contact is needed, for instance when using impulse based simulation or estimating time of impact. At this stage it is therefore possible to search the output of the narrow phase for the

the closest principal contact. Next it is tested if any contact state changes occurred, such as if touching or penetrating contact vanishes or is persistent, that is if a contact also were present in the last iteration. If one of the nodes were a trigger volume, then we do not mark touching contact, but rather *in-* and *out-* events of the trigger volume, the same applies to the marking that took place earlier on. The pseudo-code is shown in Figure 5. As can be seen in Figure 5 the output from the narrow phase collision detection, $PCs(e)$, is often used as a seed for the contact determination. This is why the method *addSeed()* is invoked after the narrow phase collision detection has been run. The meaning of the surrounding if-statement will be explained in the next section.

```
for each $e \in N$ do
  NarrowPhase.run($e, PCs(e)$)
  if $penetration(e)$ and $ShotCircuit$ then
    terminate
  end if
  if not only proximity info then
    ContactDetermination.addSeed($e, PCs(e)$)
  end if

  $minPC(e) = \min\{PCs(e)\}$

  if not $separation(e_{old})$ then
    if not $separation(e)$ then
      mark $e$ as persistent touching contact
    else
      mark $e$ as vanishing touching contact
    end if
  else
    if not $separation(e)$ then
      mark $e$ as new touching contact
    end if
  end if
next $e$
```

Figure 5: Narrow phase and short circuiting.

3.4 Contact Determination

Finally we can run the contact determination for all those edges, where their end node objects are not separated. The pseudo-code is shown in Figure 6. Observe the out most if-statement in the pseudocode. In an impulse based simulator it is often not necessary to do a full contact determination, only the closest points are actual needed [Mirtich 1996b], so an end user might want to turn of contact determination completely.

```
if not only proximity info then
  for each $e \in N$ do
    if $A(e)$ and $B(e)$ are physical then
      if not $separated(e)$ then
        ContactDetermination.run($e$)
      end if
    end if
  next $e$
end if
```

Figure 6: Contact Determination.

In the pseudo-code we have chosen to skip contact determination on nodes representing things like trigger volumes. Such entities are merely used for event notification, so there is no need for contact determination.

3.5 The Contact Groups

Now we have completed exploiting the logical and caching benefits we can gain from a contact graph. We are now ready for using the contact graph for its intended purpose: determining contact groups. The actual contact groups are found by a traditional connected components search algorithm, restricted to the union of the list N introduced in Figure 4, and the structural edges. The algorithm works by first marking all edges that should be traversed as "white". Afterwards edges are treated one by one until no more white edges exist. The pseudo-code is shown in Figure 7 and 8. In Figure 7 we have again placed a surrounding if-statement, in order to provide the most flexibility for an end user. Fixed and scripted bodies are rather special, and they behave as if they had infinite mass, such that they can support any number of bodies without ever getting affected themselves. They work like an insulator, which is why we ignore edges from these nodes, when we search for contact groups.

```
if should compute groups then
  for all edges, $e \in N$ do
    $color(e) = white$
  next $e$
  for all edges, $e \in N$ do
    if $color(e) = white$ then
      let $G$ be an empty group
      traverseGroup($e, G$)
    end if
  next $e$
end if
```

Figure 7: Connected components search.

```
algorithm traverseGroup($e$:Edge,$G$:Group)
  $color(e) = grey$
  add $e$ to $G$
  for end nodes, $n \in e$ do
    if not $n$ is fixed or scripted body then
      for all edges, $w \in n$ do
        if $color(w) = white$ then
          traverseGroup($w, G$)
        end if
      next $e$
    end if
  next $n$
  $color(e) = black$
end algorithm
```

Figure 8: Traverse group

Let us look at the contact groups of the example from Figure 2. As can be seen in Figure 9 we have four contact groups A, B, C, and D.

4 The Event Handling

In the pseudo-code we have outlined so far, we have not explicitly stated when events get propagated back to an end user. Instead we have very clearly shown, when and how the events should be detected. Table 3 summarizes the types of events, we have talked about. We can traverse the edges of

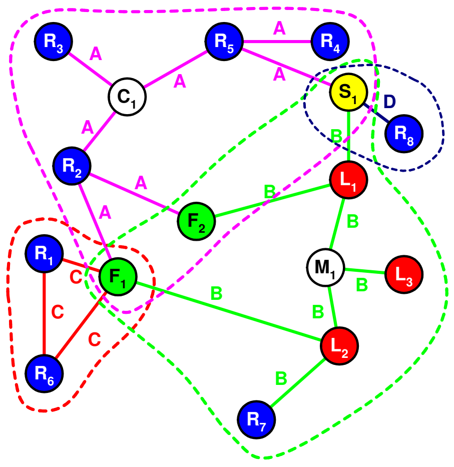

Figure 9: Example contact groups.

In Trigger Volume
Out Trigger Volume
New Touching Contact
Persistent Touching Contact
Vanishing Touching Contact
New Proximity
Persistent Proximity
Vanishing Proximity
Timer Tick

Table 3: Event types.

the graph, and simply generate the respective event notifications for all those edges that have been marked with an event. This is shown in Figure 10. The only problem are

```
for each edge $e \in graph$ do
  if $marked(e)$ then
    for each mark $m \in e$ do
      generateEvent($m, e$)
      $marked(e_{old}) = marked(e)$
    next $m$
  end if
next $e$
```

Figure 10: Event handling.

those edges, we removed due to the obsolete testing. However this can be handled gracefully by only allowing edges to become obsolete, if they were at least marked as vanishing close proximities last time the collision detection query was run.

There is one major subtlety to event handling: Some simulators are based on backtracking algorithms, also called retroactive detection, meaning that they keep on running forward until something goes wrong, and then they backtrack, correct things, and then go forward once again. This behavior could occur many times during the simulation of a single frame, and the consequences is that we might detect events, which are disregarded.

The problem of backtracking can be handled in two ways. In the first solution, events can be queued during the simulation together with a time stamp indicating the simulation time, at which they were detected. Upon backtracking one simply dequeues all events with a time stamp greater than the time the simulator backtracks to. After having dequeued the events one would have to reestablish the "marked" state of the edges at that time, which can be done by scanning through the queue. In the second solution, events are restricted to only be generated, when it is "safe", i.e. whenever a backtrack cannot occur or on completion of the frame computation. The e_{old} state should also only be updated at these places, such that the events that are generated reflect the changes since the last time events were generated.

The second solution would clearly miss events, which the first method might catch, and as the time between event generation gets bigger it will probably miss even more events. It is not because the events that are returned indicate a faulty picture of what has occurred, they merely show the same picture but with less detail. The first solution on the other hand is capable of catching more details at the cost of dynamic memory allocation, something we really would like to avoid.

We favor the second solution, because event notification are most likely to be used in a gaming context, and in such a context a backtracking algorithm would not be favorable. In predicting motion or validating offline simulation event notifications might not even be used, so the less details might not be an issue in such contexts.

As a final remark we should note that overshooting and missing contact transitions will occur with both solutions, due to the fact that the collision detection is only invoked at discrete times during a simulation. From this viewpoint the second solution can be made just as detailed as one wants, by lowering the time-step of the simulator at an added performance degradation.

5 The Spatial-Temporal Coherence Analysis Module

Having outlined how the contact graph should be used in the collision detection pipeline, we can now sketch how a STC analysis module works together with the three other modules in a collision detection engine, that is the broad phase collision detection module, the narrow phase collision detection module, and the contact determination module. Figure 11 illustrates the interaction, from which we see that the STC analysis occurs in three phases, post-broad-phase, post-narrow-phase, and post-contact-determination. We have not used a pre-broad-phase in the STC analysis, but if any initialization is supposed to take place, then a pre-broad-phase analysis would be a good place for doing this.

We have treated the narrow phase collision detection in an one-by-one approach. Other narrow phase collision detection algorithms handles all pairs of overlap at the same time see e.g. [Hubbard 1996; O'Sullivan and Dingliana 1999]. However it is pretty straightforward to modify the pseudocode we have outlined to accommodate this behavior, simply rewrite the *for*-loop in Figure 5 such that invocation of the narrow phase collision detection algorithm occurs before the *for*-loop and works simultaneously on all overlaps.

6 Using Contact Groups

In our opinion there are basically three different ways to exploit the contact groups in rigid body simulation. We will briefly talk about them in the following.

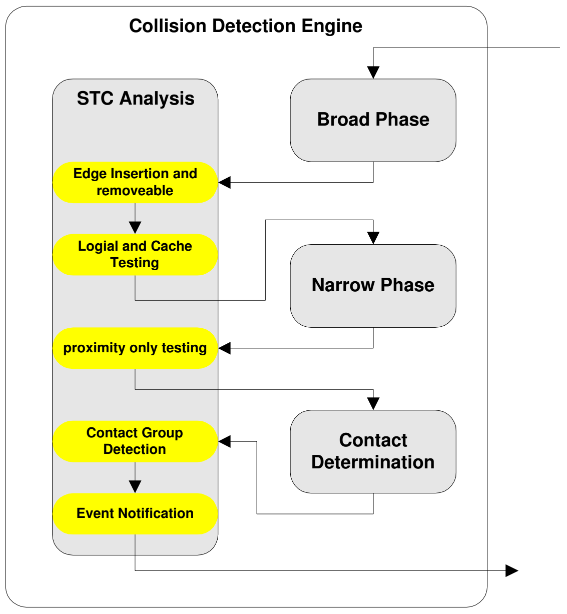

Figure 11: Spatial-Temporal Coherence Analysis Module

Time Warping Traditionally one would backtrack the entire configuration when an illegal state is found such as penetration of two objects.

This is very inefficient, since there might be a lot of objects in the scene, who's motion is completely independent of the two violating objects. The result of backtracking them is simply that one would have to redo the exact same computations as previously, thus wasting computation time.

Knowing the contact groups, one could instead only backtrack those contact groups, where the violating objects belong to, and leave all other contact groups alone. The reader should refer to [Mirtich 2000] for more details.

Subdivision of Contact Force Computation Constraint-based methods for computing contact forces are often NP-hard, so it is intractable to solve large problems, however the contact forces needed in one contact group is totally independent of all the other contact groups. This knowledge can be exploited, and instead of computing the contact forces for all contact groups, the problem is broken down into smaller problems by solving for the contact forces of each contact group separately [Anitescu et al. 1998; Coutinho 2001].

Caching Contact Forces If contact forces from the previous iteration are cached in the contact graph edges, then these forces can be used as initial guess for the contact force computation in the current iteration [Baraff 1992].

The contact graph also holds information about change of relative placement, this can also be exploited, because this means that the contact forces are the same as in the previous iteration, so these can simply be reused. Of course contact forces are dependent on external forces, so one could only exploit this idea, if current absolute placement and external forces "physically" agree with the previous absolute placement. As an example, think of a stack of books on a table: The books do not change placement at all. Similarly for a block sliding down an inclined plane: The total external force on the block is independent of the blocks motion down the plane. An example where this does not hold could be a block sliding down an inclined plane, where the inclination angle decreases as the block slides down the plane.

7 Results

We will elaborate on several speed up methods that relies on or relates to contact graphs. The speed up methods are generally applicable to any kind of rigid body simulator. In order to show the effects we have chosen to extend our own multibody simulator, a velocity based complementarity formulation [Stewart and Trinkle 1996] using distance fields for collision detection, with the speed ups. Example code is available from the OpenTissue Project [OpenTissue n. d.]. Our simulator was originally developed for medical applications to simulate skeleton bone movements, performance was not a concern but accuracy were. In this paper we will focus on performance speed up only. For this reason we have chosen a semi-implicit fixed time stepping scheme with a rather large time-step, 0.01 second, for a medical application we would have used a fix-point time stepping scheme and done a convergence analysis to determine the time-step.

Using distance fields for collision detection has one major drawback with the above mentioned time stepping scheme, during the "fake" position update objects tend to be deeply penetrating, in these cases a large number of contact points will be generated, the consequence will be a performance degradation due to the large number of variables that must be resolved. Performance improvements are therefore particular important even though real-time simulation is out of our grasp.

We have done several performance measurements and statistics on 120 spheres falling onto an inclined plane with the word "DIKU" engraved upon it. The configuration is shown in Figure 12. The total duration of the simulation is 10 seconds. In Figure 13 measurements of the brute force method is shown, i.e. without using contact graph. Observe that the number of variables and real-life time per iteration are increasing until the point where the spheres settle down to rest and then the curves flatten out.

In comparison Figure 14 shows how the curves from Figure 13 change when a contact graph is used. Notice that the number of variables per contact group is much smaller than in the brute force method, also observe the impact on the real-life duration curves. Table 4 shows how the total real-life running time in seconds is affected by using a contact graph to divide a simulation into independent contact groups. However one can do even better in the following we will explain 7 more speed up methods we have successfully used.

An obvious improvement comes from ignoring contact groups where all objects appear to be at rest, we call such objects sleepy objects, and we determine them by tracking their kinetic energy, An object is flagged as sleepy whenever its kinetic energy have been zero within numerical threshold over a number of iterations specified by an user. If a contact group only contains sleepy objects the group is completely ignored. Contact graph nodes provide the means for tracking the kinetic energy. Since this speed up is computationally inexpensive we have invoked it in all of the measurements given in Table 5. The speed up could have a potentially disastrous effect on a simulation if the scheme for tagging sleepy objects is not well-picked. To greedy an approach could leave objects hanging in the air, to lazy an approach would result in no performance gain.

We exploit contact graph edges for caching contact points and contact forces. Cached contact points are used to skip

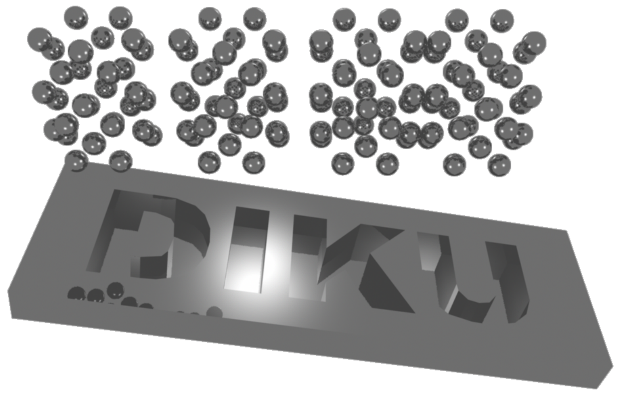

Figure 12: 120 Falling Spheres onto inclined plane with engravings.

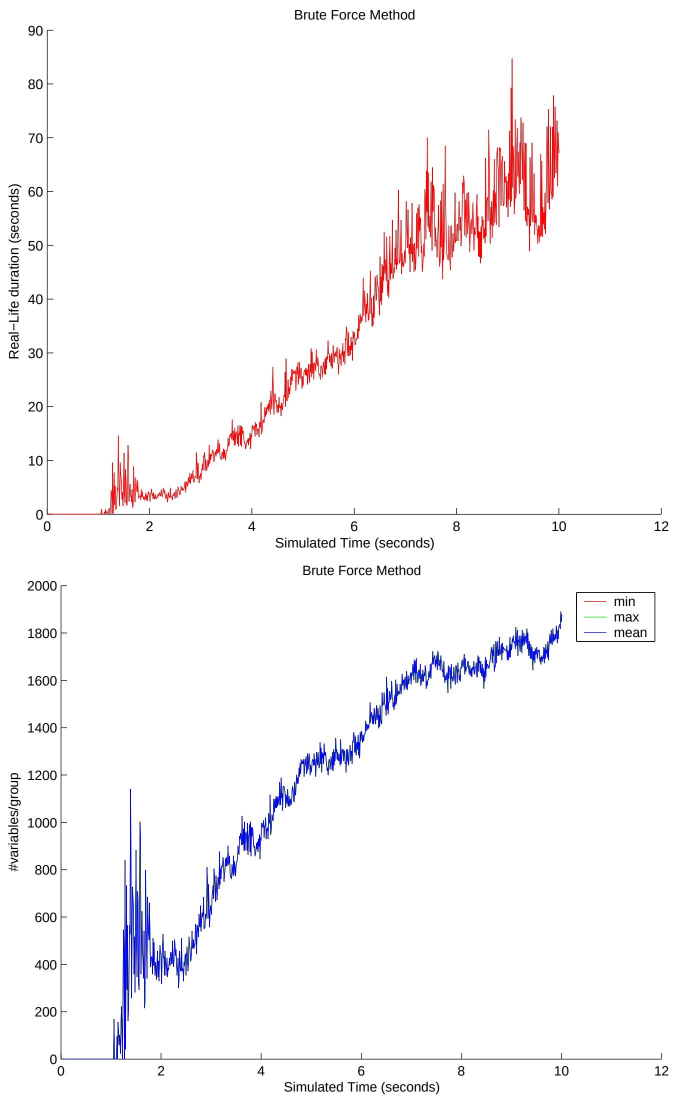

Figure 13: The Brute Force Method. In this case there is only one large contact group.

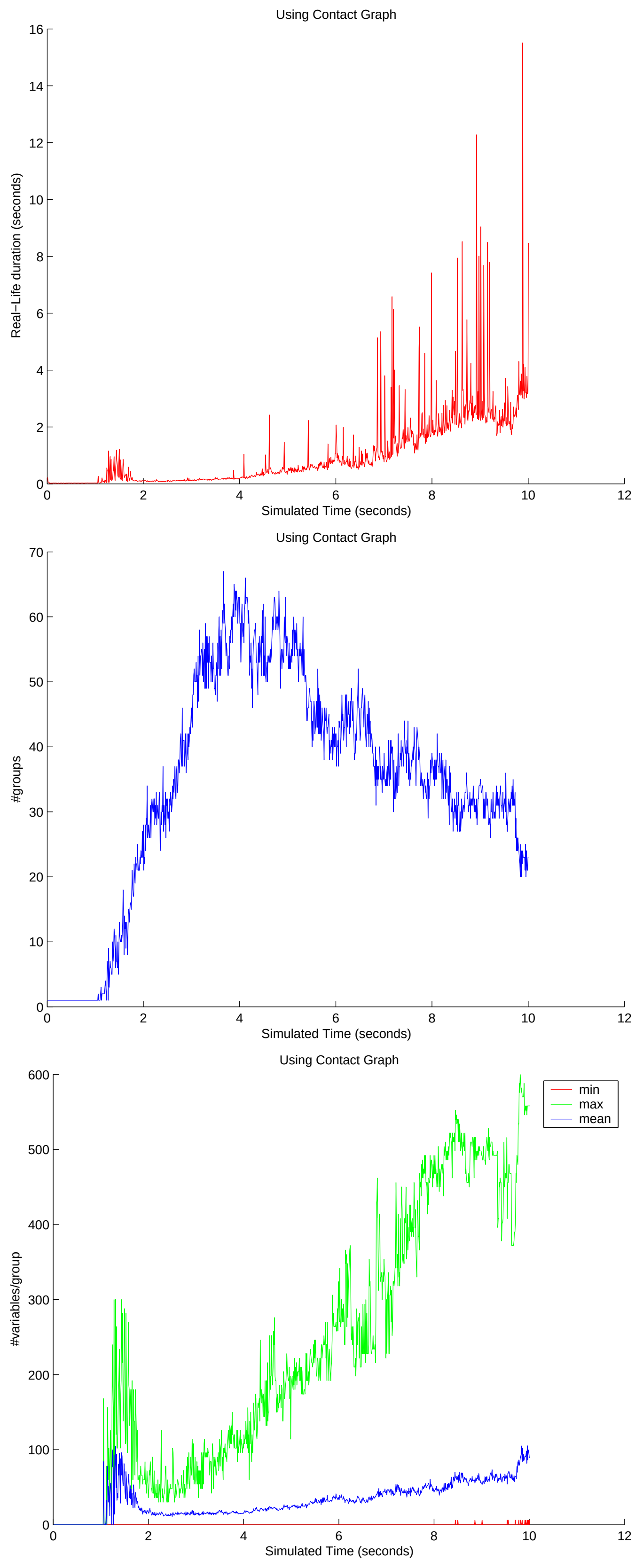

Figure 14: The performance impact of using contact graph to determine independent contact groups.

	Time (secs)
Brute Force Method	28424
Contact Graph	1011.4

Table 4: The performance effect of dividing simulation into independent contact groups.

narrow phase and contact determination whenever two incident objects of a contact edge are at absolute rest, cached contact forces are used to seed the iterative LCP solver, we use Path from CPNET [Path n. d.]. Hopefully the LCP solver is able to converge much more rapidly. From here on we call this speed up "caching".

A further speed up can sometimes be obtained by limiting the number of times the iterative LCP solver is allowed to iterate, currently we change the limit from the default value of 500 to 15, as a consequence the motion is altered but still looks plausible. The speed up has nothing to do with contact graphs, but it is interesting to examine in combination with the other speed ups we apply, see Table 5. We refer to this speed up as "tweaking".

Another speed up we use is to reduce the number of contacts between two objects in contact, the reduction is applied to objects, that are deeply penetrating. During the reduction all contacts are pruned except the single contact of deepest penetration. Again the contact graph edges provide a convenient storage. We have named this speed up "reduction". Reduction have an effect on the motion of the objects, we believe that it is actually more correct, because intuitively the deepest point of penetration better resembles the idea of using the minimum translational distance as a separation measure. Besides theoretically reduction should give a decrease in the number of variables used in the complementarity formulation.

Inspired by the speed up of detecting independent contact groups, further subdivision into groups that could be simulated independently seems feasible. An idea to further subdivide is to prune away sleepy objects from those contact groups containing both non-sleepy and sleepy objects. We refer to this speed up as "subgrouping". The idea is to think of the sleepy object as a fixed object during the computation of the contact groups. This speed up will of course have a drastic impact on the motion of the objects, however from a convergence theory point of view the effect should vanish as the time step goes to zero. Because what we have done is to interleave the simulation of subgroups by one frame. Other subgrouping/sleepy object schemes can be found in [Barzel 1992; Schmidl 2002]. To help objects settle down and become sleepy faster it intuitively seems to be a good idea to let the coefficient of restitution fall to zero the more sleepy an object gets, meaning that sleepy objects are sticky objects. Currently we simply set the coefficient of restitution to zero whenever at least one of the incident objects are sleepy. The speed up is referred to as "zeroing". In the same spirit a linear viscous damping term is added to the motion of all objects in the simulation, the intention is to slow down objects making them less willing to become non-sleepy. We call this "damping". The contact graph is used for the subgrouping and zeroing. For instance subgrouping is done by setting the edge color to black in the pseudo-code of Figure 7 whenever the incident objects both are sleepy.

The last method we have applied consist of setting the inverse mass and inertia tensor to zero for all sleepy objects. The main intuition behind this is to "force" sleepy objects to stay sleepy. We have named this "fixation", and it has a dramatic impact on the simulation, object motion is visually altered as can be seen from ♠ in Figure 15). Fixation only makes sense to apply when subgrouping is used, otherwise the iterative LCP solver have to solve for contacts between two fixated objects.

Table 5 contains performance measurements of all sensible combinations of the previously mentioned speed up methods. Figure 15 shows motion results of four selected combinations: ♢,♡,♣, and ♠ from Table 4. These are compared to the motion of the brute force method. The four combinations were picked because they resembles the best performance whenever a new speed up were used. Observe that the resulting motion diverges more and more from the brute force method the more speed ups that are used. Especially ♠ is different, during the last seconds objects actually fly up in the air. This is due to constraint stabilization correcting large errors in the simulation. In Figure 16 a comparison is done between the performance statistics of the four selected combinations ♢,♡,♣, and ♠. Observe that the plots of the first three combinations, ♢, ♡, and ♣, are similar to those shown in Figure 14. The fourth combination, ♠, has very different plots for the real-life duration and variables per group plots, these appear to be nearly asymptotically constant.

8 Discussion

It is obvious from Table 5 that the prober combination of the speed ups is capable of producing a speed up factor of $\frac{28424}{135} \approx 210$. It is difficult to describe the impact on the resulting motion, however it is clear that using Contact Graphs, Caching Contact Forces and Sleepy Groups do not change the motion of the brute force method, but all other speed ups we presented change the physical properties and as a consequence motion is altered as can be seen in Figure 15. Especially the reduction and the subgrouping have great impacts on the motion. The motion do however in the authors opinion still look plausible.

The tagging of sleepy objects can have a rather drastic impact on the simulation such as leaving objects hanging when they should not. For instance in the simulation shown in Figure 17 near the K-letter, a bunch of spheres land on top of each other, while the top-most spheres rumbles of the top, the bottom-most sphere is kept in place and prohibited from gaining kinetic energy, at the end of the simulation a single sticky sleepy sphere can be seen on the inclined plane. We have to be careful not making general conclusions based on the measurements in this paper, since only one configuration have been examined.

It is clear though that contact graphs are a valuable extension to a multibody simulator, they can be used for more than finding independent groups of objects. Even in the welhm of physical accurate simulation a speed up factor of the order of 20-30 is not unlikely, disregarding accuracy completely the speed up factor can be increased by an order of magnitude.

Using more speed ups does not always imply better performance, in some cases one speed up cancels the effect of another. For instance using caching seems to make tweaking needles, the cached solutions results in fast convergence, only

Cache	Tweak	Reduce	Zero	Damp	Subgroup	Fixate	Time
+	-	-	-	-	-	-	658.499
-	+	-	-	-	-	-	589.523
+	+	-	-	-	-	-	952.519
-	-	+	-	-	-	-	712.567
+	-	+	-	-	-	-	◇ 624.274
-	+	+	-	-	-	-	895.157
+	+	+	-	-	-	-	1157.27
-	-	-	+	-	-	-	840.673
+	-	-	+	-	-	-	771.285
-	+	-	+	-	-	-	1205.34
+	+	-	+	-	-	-	575.343
-	-	+	+	-	-	-	1246.65
+	-	+	+	-	-	-	965.894
-	+	+	+	-	-	-	599.258
+	+	+	+	-	-	-	688.454
-	-	-	-	+	-	-	578.171
+	-	-	-	+	-	-	578.339
-	+	-	-	+	-	-	♡ 528.633
+	+	-	-	+	-	-	533.934
-	-	+	-	+	-	-	788.291
+	-	+	-	+	-	-	890.056
-	+	+	-	+	-	-	789.72
+	+	+	-	+	-	-	1093.42
-	-	-	+	+	-	-	766.438
+	-	-	+	+	-	-	627.623
-	+	-	+	+	-	-	736.771
+	+	-	+	+	-	-	663.246
-	-	+	+	+	-	-	608.286
+	-	+	+	+	-	-	956.992
-	+	+	+	+	-	-	1273.16
+	+	+	+	+	-	-	730.885
-	-	-	-	-	+	-	700.215
+	-	-	-	-	+	-	541.854
-	+	-	-	-	+	-	785.367
+	+	-	-	-	+	-	561.064
-	-	+	-	-	+	-	855.264
+	-	+	-	-	+	-	569.153
-	+	+	-	-	+	-	523.856
+	+	+	-	-	+	-	618.052
-	-	-	+	-	+	-	1077.2
+	-	-	+	-	+	-	761.245
-	+	-	+	-	+	-	760.623
+	+	-	+	-	+	-	903.259
-	-	+	+	-	+	-	767.546
+	-	+	+	-	+	-	1329.69
-	+	+	+	-	+	-	949.809
+	+	+	+	-	+	-	535.479
-	-	-	-	+	+	-	515.74
+	-	-	-	+	+	-	♣ 460.561
-	+	-	-	+	+	-	703.067
+	+	-	-	+	+	-	578.634
-	-	+	-	+	+	-	863.636
+	-	+	-	+	+	-	576.862
-	+	+	-	+	+	-	612.122
+	+	+	-	+	+	-	502.618
-	-	-	+	+	+	-	625.918
+	-	-	+	+	+	-	569.84
-	+	-	+	+	+	-	550.011
+	+	-	+	+	+	-	702.223
-	-	+	+	+	+	-	958.467
+	-	+	+	+	+	-	871.314
-	+	+	+	+	+	-	958.066
+	+	+	+	+	+	-	643.04
-	-	-	-	-	+	+	253.577
+	-	-	-	-	+	+	222.027
-	+	-	-	-	+	+	747.439
+	+	-	-	-	+	+	176.69
-	-	+	-	-	+	+	383.162
+	-	+	-	-	+	+	264.526
-	+	+	-	-	+	+	134.679
+	+	+	-	-	+	+	147.884
-	-	-	+	-	+	+	259.678
+	-	-	+	-	+	+	313.898
-	+	-	+	-	+	+	171.455
+	+	-	+	-	+	+	172.426
-	-	+	+	-	+	+	410.996
+	-	+	+	-	+	+	306.881
-	+	+	+	-	+	+	1115.21
+	+	+	+	-	+	+	174.302
-	-	-	-	+	+	+	191.925
+	-	-	-	+	+	+	273.825
-	+	-	-	+	+	+	176.747
+	+	-	-	+	+	+	168.905
-	-	+	-	+	+	+	186.134
+	-	+	-	+	+	+	311.081
-	+	+	-	+	+	+	179.803
+	+	+	-	+	+	+	♠ 134.721
-	-	-	+	+	+	+	363.336
+	-	-	+	+	+	+	296.197
-	+	-	+	+	+	+	704.94
+	+	-	+	+	+	+	176.358
-	-	+	+	+	+	+	222.81
+	-	+	+	+	+	+	276.723
-	+	+	+	+	+	+	181.782
+	+	+	+	+	+	+	137.757

Table 5: Comparison of various combinations of performance speed-up methods. “+” means enabled, “-” means disabled.

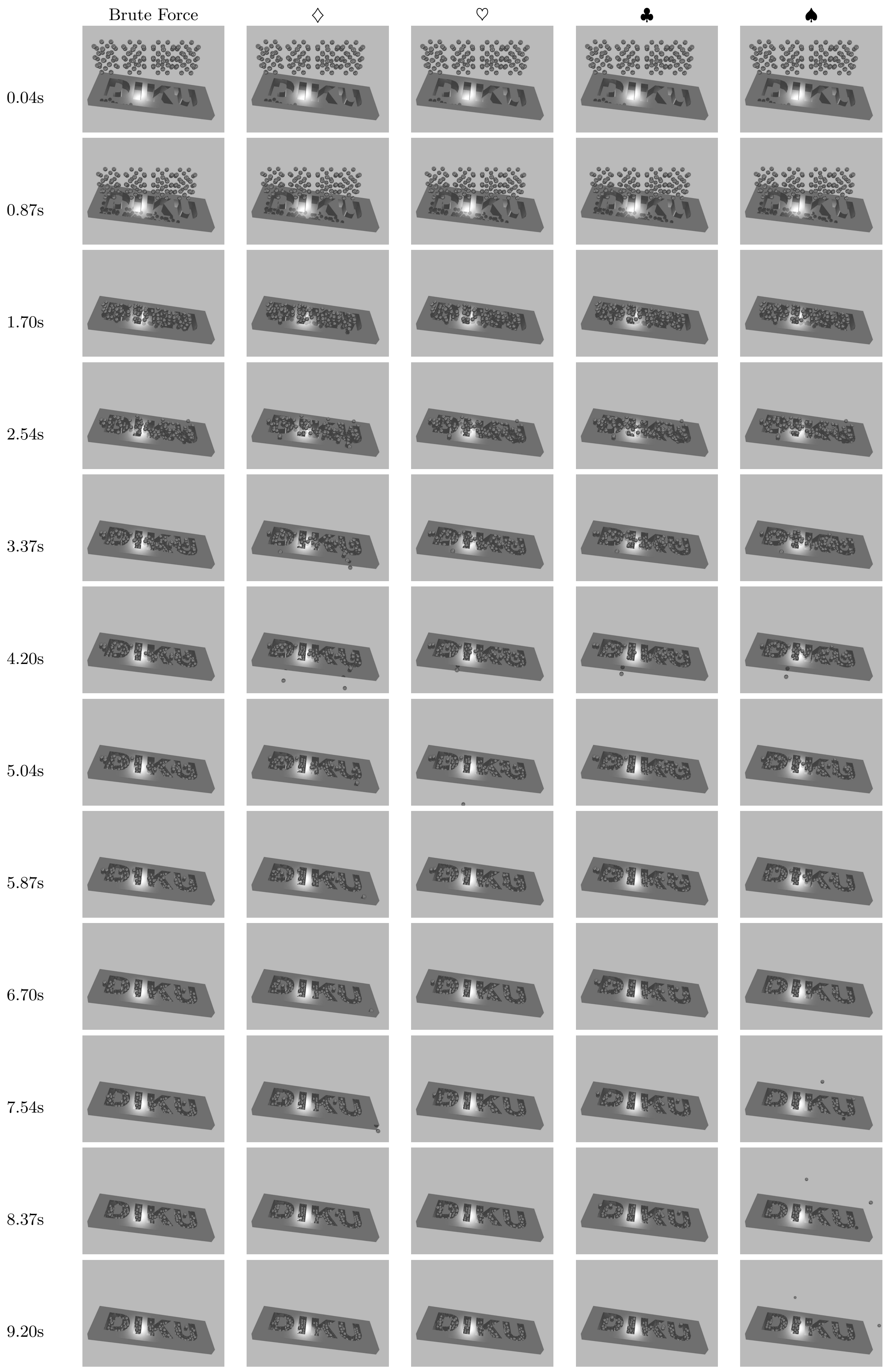

Figure 15: Motion Results of selected combinations of speed up.

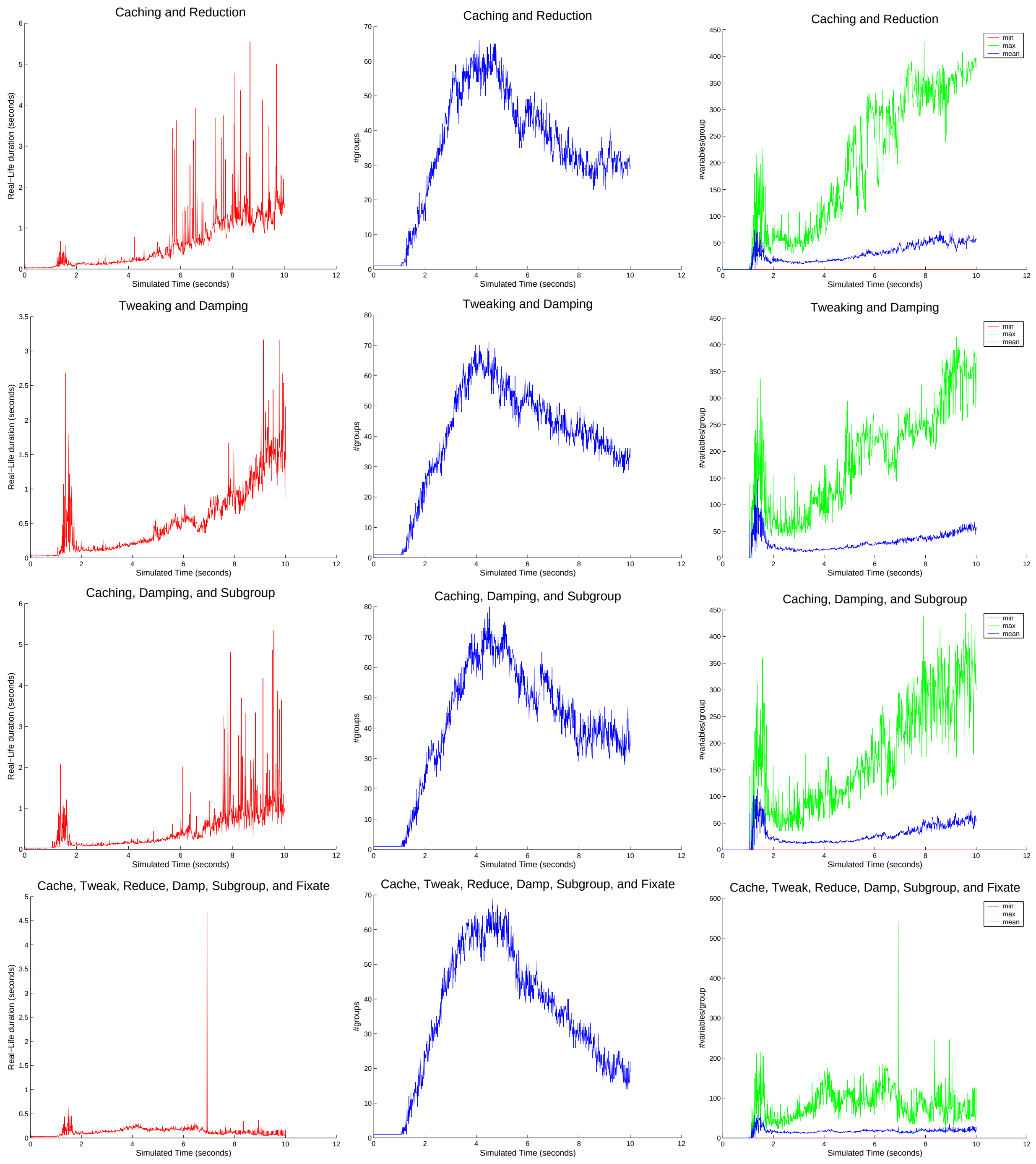

Figure 16: Performance measurements on the four selected combinations of speed ups.

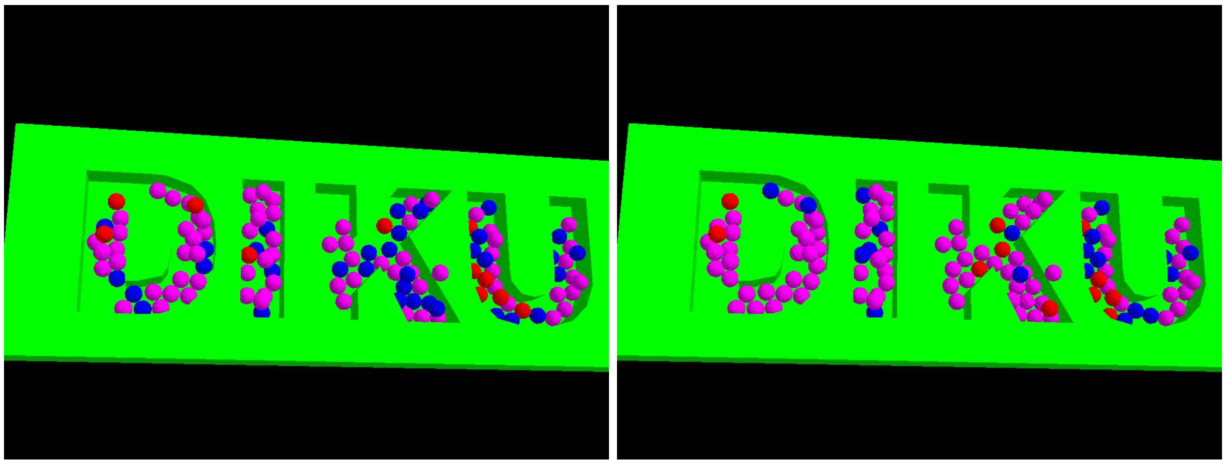

Figure 17: Figure showing sleepy object hanging in the air. Purple means sleepy, red moving, blue absolute rest, green fixed. Frame grabs of simulation at time 9.8 secs and 9.87 secs using zeroing, damping, subgrouping and fixation.

seldom is the upper iteration limit dictated by the "tweaking" reached.

Our experiments indicate that a promising avenue for high-performance simulations is a combination of subgrouping and fixation, however from the resulting motions shown in Figure 15 it is also clear that this is far from trivial to device such a scheme.

We believe that contact graphs also can be exploited to determine when a paradigm switch should occur in a hybrid simulation as described in [Mirtich 1996a; Mirtich 1996b]. In [Mirtich 1996a] it is suggested to track the contact state, i.e. the relative contact velocity, and use this information to determine when a contact should be solved by a constraint based method or by an impulse based method. Contact edges provide a perfect place for storing this tracking information, they also provide one with the possibility to taking neighboring contacts into consideration. In [Mirtich 1996b] large stacks of objects is shown to be infeasible to simulate with an impulse based method, examining the size of contact groups and their structure, might give a clue to switch from an impulse based method to a constraint based method when lots of objects settle into resting contact upon each other.

Our numerical experiments clearly indicates that sleepy objects are a promising strategy, it therefore seems promising to look into better methods for more quickly making objects become sleepy and stay sleepy. For instance to preprocess the complementarity formulation with a sequential collision method truncating impulses [Chatterjee and Ruina 1998] and successfully applied to sequential collision resolving [Guendelman et al. 2003], The novelty would be to extend the ideas to simultaneously contact resolving.

References

ANITESCU, M., POTRA, F. A., AND STEWART, D. E. 1998. Time-stepping for three-dimensional rigid body dynamics. *Comp. Methods Appl. Mech. Engineering*.

BARAFF, D. 1992. *Dynamic simulation of non-penetrating Rigid Bodies*. PhD thesis, Cornell University.

BARAFF, D. 1994. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics 28*, Annual Conference Series, 23–34.

BARZEL, R. 1992. *Physically-based Modelling for Computer Graphics, a structured approach*. Academic Press.

CHATTERJEE, A., AND RUINA, A. 1998. A new algebraic rigid body collision law based on impulse space considerations. *Journal of Applied Mechanics*.

COUTINHO, M. G. 2001. *Dynamic Simulations of Multibody Systems*. Springer-Verlag.

ERLEBEN, K., AND HENRIKSEN, K. 2002. Scripted motion and spline driven motion. Technical Report 02/18, Department of Computer Science University of Copenhagen, August.

ERLEBEN, K., AND SPORRING, J. 2003. Review of a general module based design for rigid body simulators. Unpublished, draft version can be obtained by email request.

ERLEBEN, K., 2001. En introducerende lærebog i dynamisk simulation af stive legemer, Maj.

GUENDELMAN, E., BRIDSON, R., AND FEDKIW, R. 2003. Nonconvex rigid bodies with stacking. *ACM Transaction on Graphics, Proceedings of ACM SIGGRAPH*.

HAHN, J. K. 1988. Realistic animation of rigid bodies. In *Computer Graphics*, vol. 22, 299–308.

HUBBARD, P. M. 1996. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics 15*, 3, 179–210.

MIRTICH, B., 1996. Hybrid simulation: combining constraints and impulses.

MIRTICH, B. 1996. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley.

MIRTICH, B. 1998. Rigid body contact: Collision detection to force computation. Technical Report TR-98-01, MERL, March.

MIRTICH, B. 2000. Timewarp rigid body simulation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 193–200.

OPENTISSUE. http://www.diku.dk/forskning/image/research/opentissue/.

O'SULLIVAN, C., AND DINGLIANA, J., 1999. Real-time collision detection and response using sphere-trees.

PATH. PATH CPNET Software, http://www.cs.wisc.edu/cpnet/cpnetsoftware/.

SCHMIDL, H. 2002. *Optimization-based animation*. PhD thesis, Univeristy of Miami.

STEWART, D., AND TRINKLE, J. 1996. An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction. *International Journal of Numerical Methods in Engineering*.