1.5

Collision Detection of Deformable Volumetric Meshes

Kenny Erleben, Department of Computer Science and Jon Sporring, 3DLab, School of Dentistry, University of Copenhagen

kenny@diku.dk
sporring@lab3d.odont.ku.dk

Collision detection of deformable objects is challenging, because these objects may change shape. A deformable object can be the shape of a ball during play, a mattress under pressure, or in a movie fantasy, the magical substance *Flubber*. Deformations can even be so extreme that an object self-intersects or that topological changes occur, like when the shape of a cloud and a droplet of water separates from a stream flow. In this article, we consider narrow phase collision of solid deformable objects. Self-intersection is therefore considered as self-collision, i.e., an error state that the simulator should compensate for by calculating a collision response. While topological changes may be easily integrated in the proposed algorithm, they are not considered in this article.

For both static and deformable objects, efficient collision detection algorithms rely on coarse to fine representations of objects' spatial configuration (i.e., shape). For static objects, this representation may be calculated in advance, but efficient collision detection of deformable objects requires that these structures be updated continuously. Most successful algorithms are based on Axis-Aligned Bounding Box (*AABB*) trees [Larsson01], [van den Bergen97], where techniques from static objects have been augmented with efficient updating of data structures. However, these algorithms do not support self-intersection.

In this article we will present a method of doing collision detection on organic tissues modeled as volumetric tetrahedron meshes based on a specialization of the BucketTree algorithm [Ganovelli00]. We will extend and specialize the BucketTree algorithm in order to accommodate our needs in a dynamic mandibular motion and knee joint simulator project. In our application, we are simulating the movement of the lower jaw under the forces of the muscles in order to study the physiology of the human head. A typical illustration of the data we are working on is given in Figure 1.5.1.

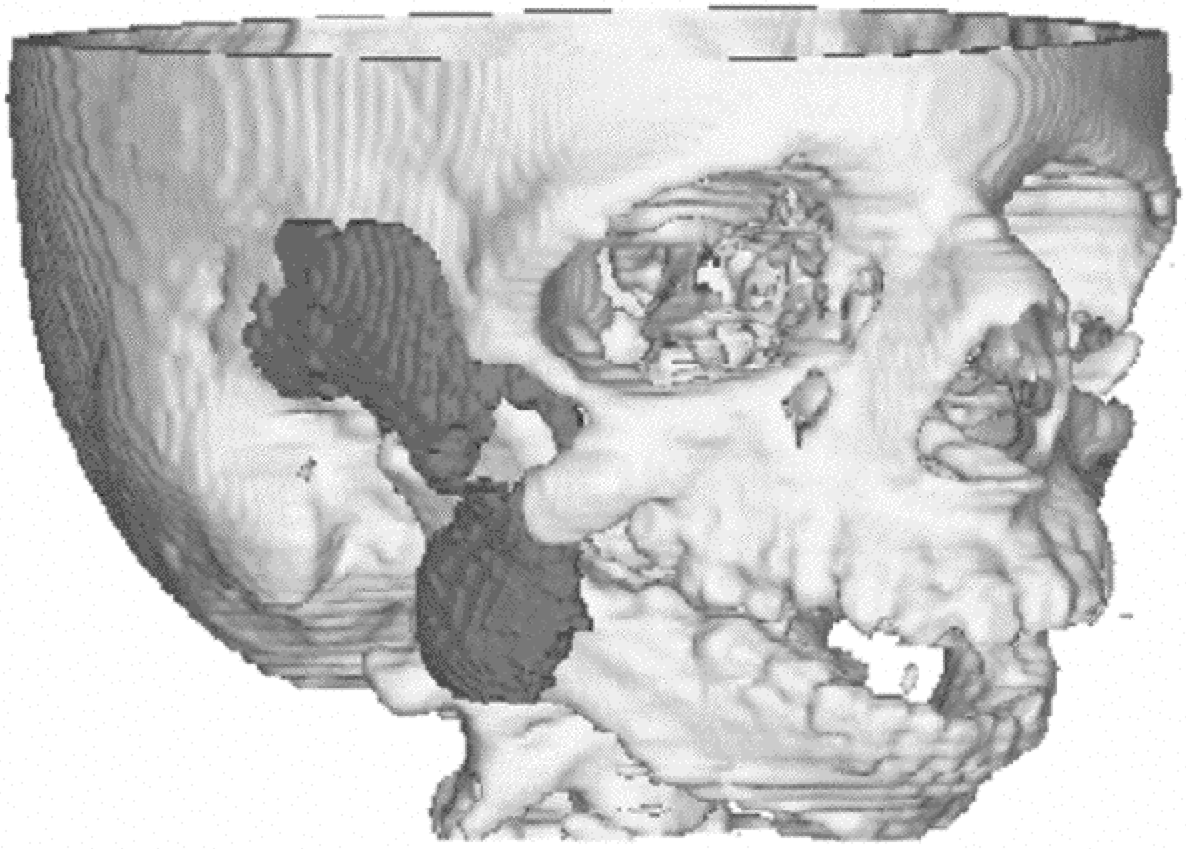

FIGURE 1.5.1 *A cranium with two sets of muscles. © Ole Fogh Olsen, IT-C 2003.*

Previous Work

In [van den Bergen97], initial *AABB* trees are built in local object coordinate systems. The trees are built using classical top-down splitting methods, and the entire *AABB* tree is updated in a bottom-up fashion ensuring that parent *AABB*s enclose their children *AABB*s. During runtime, the *AABB* trees are moved, causing the local defined *AABB*s to become Orientated Bounding Boxes *(OBB)*. A variation of the separation axis overlap test method is used to deal efficiently with this problem.

Self-intersections are treated in [Volino98] by organizing the boundary of an object into a hierarchical representation of subsurfaces, each of which has no self-intersections.

The BucketTree algorithm, [Hirota02] and [Ganovelli00], is a different approach to updating a spatial data structure, which is easily extended to more complex scenarios. In the BucketTree algorithm, a fast method is used to remap the "primitive" into the spatial data structures, instead of updating the spatial data structure at every iteration.

In [Larsson01] a top-down method is used for building *AABB*s that handle deformation of arbitrary vertex positioning of meshes. Mesh connectivity is analyzed when splitting parents to children. Further, in order to minimize the number of *AABB* updates, the method uses a top-down update approach while doing a tandem traversal, but when at a predefined depth, the method uses a bottom-up update method on the remaining subtree.

Other algorithms recursively search for the "contour of collision" between two meshes and build the "zone of inter penetration" [Joukhadar96], [Joukhadar98], [Joukhadar99], and [Sundaraj00]. These algorithms exploit the topology of the faces in a mesh in order to perform a fast search, given an initial collision pair. Our BucketTree does not assume or exploit any topological information about the volumetric mesh, and therefore is solving a more general problem.

The Organic Tissue

The simulator we wrote is for simulating organic tissue in a surgery simulator. We model organic tissue with a volumetric mesh (i.e., a mesh of small volumes connected to each other by neighboring faces). We will assume that the volumes are convex polyhedra of identical structure. The simplest polyhedra are a tetrahedron mesh, which will be the focus here, but our ideas extend to the general case.

Tetrahedra in a mesh share triangular faces. We say that two tetrahedra sharing a face are neighbors, as illustrated in Figure 1.5.2.

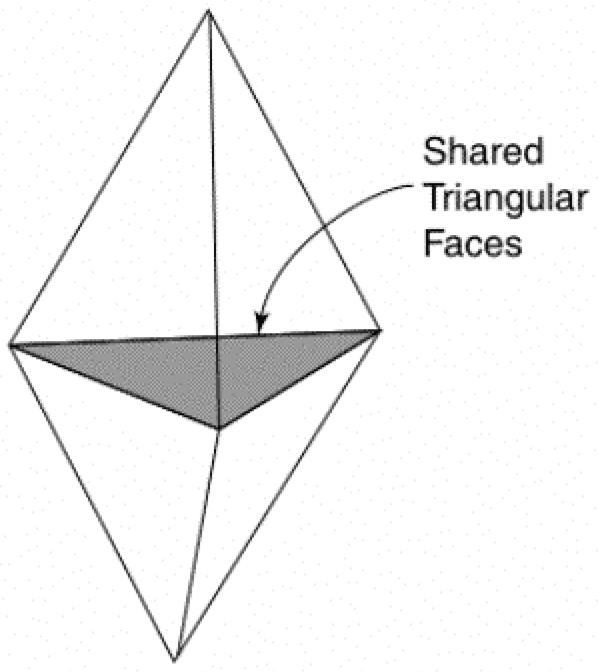

FIGURE 1.5.2 *Two neighboring tetrahedra with a shared triangular face.*

Conversely, the tetrahedra that have triangle faces without a neighboring tetrahedron must lie on the surface of the tissue. Thus the surface of the tissue is made up completely by the triangles of the tetrahedra without a neighbor. The information about a tetrahedron mesh does not depend on the deformation of the object. Because of this, it may be precomputed and obtained in constant time during runtime.

Physical simulation of organic tissue requires that we are capable of detecting penetration of a surface of one tissue against the volume of another tissue, where a test for self-intersection implies testing the surface of an object against its own volume [Hirota01]. We want to test surfaces against volumes, not surfaces against surfaces. The reason we do this is that typical energy simulations, such as Finite Element Models [Zienkiewicz00], apply forces to the nodes in volume elements, and testing against volumes appears to yield the simplest collision response in general.

A sketch of the collision detection we are using for a simple organic tissue simulator is seen in the following pseudo code.

Listing 1.5.1

```
Algorithm simulate(...)
  while not finished do
    deformation and/or movement
    test tissues against each other
    test tissues for self-intersections
    Process reported penetrations
  end while
End algorithm
```

The details of this algorithm will be discussed in the rest of this article.

Creating a BucketTree

Most organic tissue is nonrigid, the exception being the skeleton and the teeth. The tissue will deform during simulation because of the pressure of applied tools or because of movement of the body parts. For fast collision detection, we use a hierarchical representation of the spatial extent of each object. For three-dimensional objects, we use octrees. Octrees are a recursive subdivision of a cube into eight smaller, nonoverlapping cubes such that the entire volume of the mother cube is covered. The process in 2D is illustrated in Figure 1.5.3.

The smallest cubes (squares in Figure 1.5.3) are called leaves or nodes, and the leaves are joined in tuples of eight until the root is reached. The advantage of this tree is that the levels of the tree represent the objects at the corresponding resolution. In other words, we may test collision in a coarse to fine manner. First we compare the roots of two objects, and if their bounding boxes overlap, we may recursively refine the collision search down through the two trees.

In the BucketTree algorithm, the leaves are denoted buckets. If the tree has l-levels, then the number of leaves is 8^{l-1}, where we use the convention that the root is level 1. At the root level, the tree is a single cube of size $w \times h \times d$. At the mth level a cube has size:

$$\frac{w}{2^{m-1}} \times \frac{h}{2^{m-1}} \times \frac{d}{2^{m-1}}. \tag{1.5.1}$$

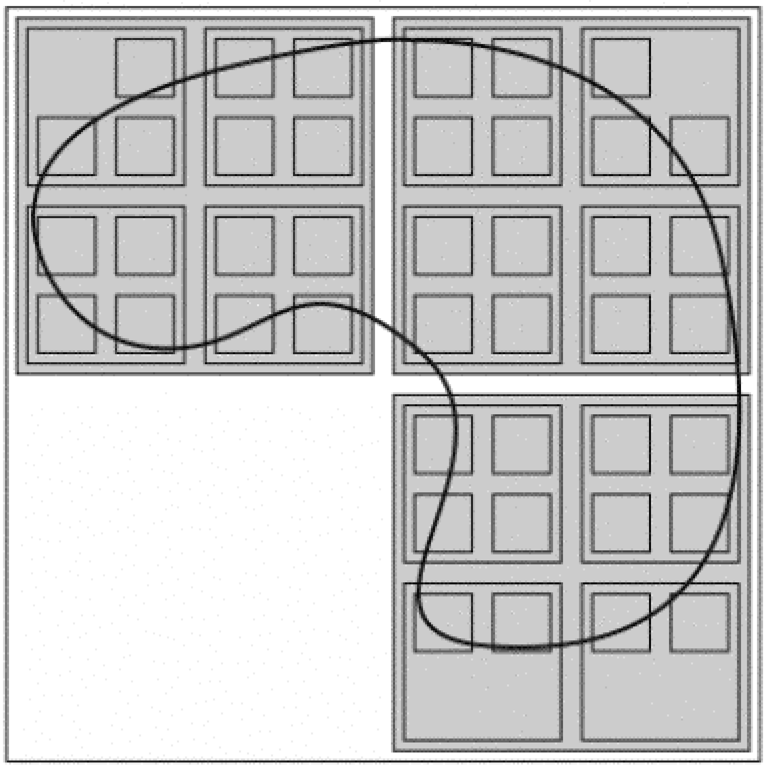

FIGURE 1.5.3 *A quadtree representation of a flat object. Each rectangle is recursively subdivided into four rectangles completely covering the mother square. The subdivided rectangles are visualized slightly smaller in the figure for illustration purposes.*

All boxes in the tree have the same orientation and, before simulation is started, they are all axis-aligned. Although unnecessary, we have decided to keep the tree axis-aligned throughout the simulation, because it makes the mapping easier as discussed in the following section. This means that all nodes in the tree have a corresponding *AABB*. At the lowest level, the buckets actually form a grid structure of *AABB*s.

When creating the tree, we face two problems: picking the size of the root bucket and determining the number of levels in the tree. The size of the root $w \times h \times d$ must be big enough to accommodate extreme deformations. When working with organic tissues, there is a maximum bound on how extreme these deformations can become. We may estimate the values of w, h, and d, such that the tree always will be able to contain the deforming tissue. Alternatively, w, h, and d may be recomputed before mapping takes place [Ganovelli00], (i.e., the entire bucket tree is scaled to fit the deformed object). Increasing the number of levels, l, implies that the fine scale precision of the collision detection is increased, however, there is a trade-off between the number of levels and the number of tetrahedra in each bucket. This is discussed in more detail in the Computational Complexity section of this article.

The size of the tetrahedra chosen to model objects is a direct indicator of the volumetric precision needed for graphical modeling. We assume that the organic tissue to be simulated uses tetrahedra of almost similar size; hence the size of the buckets must be proportional in size. Each bucket only overlaps with a small number of tetrahedra. Let n be an indicator for the number of tetrahedra to overlap with a single bucket and let s be an estimate of the average edge size of the tetrahedra, then

$$sn \approx \frac{w}{2^{l-1}} \approx \frac{h}{2^{l-1}} \approx \frac{d}{2^{l-1}} \tag{1.5.2}$$

and we may choose value of l as

$$l \cong 1 + \log_2 \frac{\max(w, h, d)}{sn} \tag{1.5.3}$$

using $\log_m(k) = \log(k)/\log(m)$. To calculate s, we first estimate the mean tetrahedron volume in the mesh, $\bar{V} = \sum_{i=1}^{N} \frac{V_i}{N}$, where N is the number of tetrahedra, and where the volume of a tetrahedron is calculated by [Zienkiewicz00, App. E] [O'Rourke93]

$$V_i = \frac{1}{6} \begin{vmatrix} 1 & x_0 & y_0 & z_0 \\ 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \end{vmatrix}, \tag{1.5.4}$$

using $\vec{\mathbf{p}}_j = (x_j, y_j, z_j)^T, j \in \{0..3\}$ for the four nodes of the tetrahedron and $|\cdot|$ as the determinant operator. Then we calculate the corresponding mean radius, as if the tetrahedra had been spheres,

$$\bar{r} = \left(\frac{3\bar{V}}{4\pi}\right)^{1/3}, \tag{1.5.5}$$

and use $s = 2\bar{r}$. Because we want to ensure that n is a lower limit on the number of tetrahedra overlapping with each bucket in the mean, we use,

$$l = \left\lfloor 1 + \log_2\left(\frac{\max(w, h, d)}{2\bar{r}n}\right)\right\rfloor, \tag{1.5.6}$$

where the floor operator, $\lfloor a \rfloor$, is the largest integer not greater than a. Note that during simulation, the objects are deforming dynamically, and the size of each bucket is not guaranteed to be similar to n.

The full octree is defined solely by the four constants w, h, d, and l, except for the global position of the root. The constants w, h, and d define the size of the root. Starting with the root cube, the cubes are subdivided into exactly eight children until l levels have been produced.

Mapping Tetrahedra into Buckets

Each bucket can be assigned a unique triplet $\mathbf{T} = (T_x, T_y, T_z)$ identifying its spatial position. Let the bucket with lowest, leftmost, and back-facing coordinates be $\vec{o} = (o_x, o_y, o_z)^T$, where ^{T} is the transpose operator making the row vector into a column vector and vice versa. Hence, for a point in space, $\vec{p} = (p_x, p_y, p_z)^T$, we can find the bucket in which it lies by calculating $\mathbf{T}$ as,

$$\mathrm{T} = \left\lfloor 2^{l-1}\left(\frac{p_x - o_x}{w}, \frac{p_y - o_y}{h}, \frac{p_z - o_z}{d}\right)\right\rfloor \qquad (1.5.7)$$

where $\lfloor \cdot \rfloor$ is generalized to vectors by applying the operation element-wise.

To map a tetrahedron to the BucketTree, we consider the four points of the tetrahedron, $\vec{\mathbf{p}}_0$, $\vec{\mathbf{p}}_1$, $\vec{\mathbf{p}}_2$, and $\vec{\mathbf{p}}_3$. The lowest, leftmost, and back-facing corner point $\vec{\mathbf{p}}_{\min}$ and highest, rightmost, and front-facing corner point $\vec{\mathbf{p}}_{\max}$ define an *AABB* around the tetrahedron. Mapping $\vec{\mathbf{p}}_{\max}$ and $\vec{\mathbf{p}}_{\min}$ into the grid gives us two triplets $\mathbf{T}_{\min}$ and $\mathbf{T}_{\max}$, from which we can compute an interval of buckets, that covers the tetrahedron using the following constraints,

$$\begin{aligned} \mathrm{T}_{\min,x} &\le n_x \le \mathrm{T}_{\max,x} \\ \mathrm{T}_{\min,y} &\le n_y \le \mathrm{T}_{\max,y} \\ \mathrm{T}_{\min,z} &\le n_z \le \mathrm{T}_{\max,z}\,. \end{aligned} \qquad (1.5.8)$$

When we map a tetrahedron to the BucketTree, we simply add the tetrahedron to the buckets that fulfil the constraints. All nonempty buckets are stored in a list, such that they may easily be cleared at a later stage. We also update a time stamp on the nonempty buckets. The time stamp will allow us to easily find the parts of the BucketTree that are nonempty after the object has deformed. A sketch of the algorithm is found in the following.

Listing 1.5.2

```
Algorithm addToBuckets(Q:Tetrahedron, B:List )
  ($\vec{P}_{min}$, $\vec{P}_{max}$) = AABB(Q)
  $T_{min}$ = map($\vec{P}_{min}$)
  $T_{max}$ = map($\vec{P}_{max}$)
  for all $T_{min} < T$ and $T < T_{max}$ do
    selfIntersection(Q, T)
    if Q on surface then
      surface(T) = true
    end if
    add Q to bucket T
    stamp(T) = now
    add bucket T to B
  next T
End algorithm
```

If a bucket contains more than one tetrahedron, then the object may be self-intersecting. We only test the surface of the newly added tetrahedron against nonneighboring tetrahedra that already have been added. So if the tetrahedron to be added has neighbors to all four sides, then no self-intersection test is performed. Neighboring tetrahedra are likely to be mapped into the same buckets because their *AABBs* probably overlap. A fast implementation of the notion of neighbors, such as a list of pointers to the neighbors, decreases computation time considerably. A sketch of the self-intersection algorithm is shown as in Listing 1.5.3.

Listing 1.5.3

```
Algorithm selfIntersection(Q:Tetrahedron, T:Triplet)
      if Q is on surface then
        S = extract surface faces of Q
        for all tetrahedra W∈T do
          if Q and W not neighbors then
            for all faces F∈S do
              if overlap(F, W) then
                report(F, W)
            next F
          end if
        next W
      end if
    End algorithm
```

Updating the BucketTree

Having explained the details of how a single tetrahedron is mapped into the BucketTree, we can now explain how all the tetrahedra are corrected or remapped at each iteration of the simulator. First we clear all nonempty buckets from the previous iteration. Next, we loop over all the tetrahedra, and add each tetrahedron one by one to the buckets. The final step is to propagate information up the octree (i.e., from the buckets to their parents and so on). This will be discussed in the section about Colliding Bucket Trees. In the remap algorithm, for each iteration of the simulator, the shape of the object is possibly changed and the octree is changed accordingly by recalculating (remapping) the overlap of the geometric primitives (the tetrahedra) and the octree.

Listing 1.5.4

```
Algorithm remap(M:TetraMesh, B:List)
  clear all buckets in B
  for all tetrahedra Q∈M do
    addToBuckets(Q, B)
  next Q
  updateFlags(B)
End algorithm
```

With the algorithm updateFlags, two flags are maintained in all the nodes of the octree: time stamp and the existence of surface in or below the node. These values are maintained recursively up the tree, starting from the buckets (leaves).

Listing 1.5.5

```
Algorithm updateFlags(B:List)
  for each b∈B do
    p = parent(b)
    propagate = true
    while propagate do
      if p = NULL then
        propagate = false
      else if stamp(p) = stamp(b) then
        if surface(b) and not surface(p) then
          surface(p) = surface(b)
        else
          propagate = false
        end if
      else
        stamp(p) = stamp(b)
        surface(p) = surface(b)
      end if
  next b
End algorithm
```

During the propagation, the time stamps of all the ancestors of all the nonempty buckets are updated. This way we know which nodes in the BucketTree contain parts of the volumetric mesh.

Time stamps prove useful because we do not have to reset them in those parts of the BucketTree that have become empty due to deformation. During a collision query, these parts will have old time stamps and are therefore ignored.

The worst case upper bound on the number of nodes in the BucketTree where where time stamps need to be updated is identical to the number of nonleaves in the BucketTree, $\frac{8^{l-1}-1}{7}$.

The Coordinate Sorting Strategy

By associating an *AABB* with each node in the BucketTree, it is realized that improved performance can be obtained [Ganovelli00] by adopting a coordinate sorting strategy. Storing the coordinates of all the *AABB*s of the tetrahedra in three coordinate lists, one for the *x*-, *y*-, and *z*-coordinates, allows us to exploit coherence when sorting, and sorting may be performed in linear time by using insertion sort. The idea is to avoid the multiplications and divisions needed for computing the triplet values identifying the buckets. The updating strategy is similar to the flipping technique usually used in the implementation of sweep-and-prune in broad phase collision detection [Cohen et al., 1994], [Baraff, 2001]. Our strategy is as follows: first we sort the *AABB*s of all the tetrahedra, then we update the interval values of the *AABB*s of the nodes in the octree, and finally we compute the tetrahedra bucket indices.

We calculate the width, height, and depth of the BucketTree by sorting the coordinate lists and finding the minimum and maximum coordinate values, such that the BucketTree precisely encloses the tetrahedron mesh:

$$\begin{aligned} w &= x_{\max} - x_{\min} \\ h &= y_{\max} - y_{\min} \\ d &= z_{\max} - z_{\min} \end{aligned} \tag{1.5.9}$$

and we set $\vec{o}$ to be the minimal point,

$$\begin{aligned} o_x &= x_{\min} \\ o_y &= y_{\min} \\ o_z &= z_{\min}. \end{aligned} \tag{1.5.10}$$

To recalculate the *AABB*s of the nodes in the octree, we use three lists of coordinates: *x*-, *y*-, and *z*-values, denoting the interval of the nodes. These are the interval lists. We may now represent the *AABB*s for each node in the octree as pointers into the interval lists. Hence, during simulation, only the interval lists need to be updated in order to update the *AABB* of all the nodes in the octree. The algorithm `updateIntervalLists` has the coordinates of the *AABB* of each node in the octree are stored as references into interval lists.

Listing 1.5.6

```
Algorithm updateIntervalLists()
  X[0] = o.x
  for i = 1 to $2^{l-1}$ do
    X[i] = X[i - 1] + (w / $2^{i-1}$)
  next i
  Y[0] = o.y
  for i = 1 to $2^{l-1}$ do
    Y[i] = Y[i - 1] + (h / $2^{i-1}$)
  next i
  Z[0] = o.z
  for i = 1 to $2^{l-1}$ do
    Z[i] = Z[i - 1] + (d / $2^{i-1}$)
  next i
End algorithm
```

In contrast to [Ganovelli00], we store the coordinates of the *AABB* of the tetrahedra together with the respective indices, (t_x, t_y, t_z). In this manner, we can compute all the triplets, *T*, of the BucketTree in a linear manner. This is done by looping through the coordinate lists and the interval lists simultaneously, as illustrated in the algorithm `updateBucketIndices`. This routine performs fast computation of the triplets of the nodes in the BucketTree. The comparison of two floats is performed instead of the multiplication and division used in Equation 1.5.1.

Listing 1.5.7

```
Algorithm updateBucketIndices( A:CoordinateAxe, X:Values)
  i = 0
  while value(e)• X[i + 1] then
    i = i + 1
  end while

  for each endpoint e∈A do
    while value(e) ≥ X[i + 1] and i + 1 < $2^{l-1}$ do
      i = i + 1
    end while
    index(e) = i
  next e
End algorithm
```

All that remains is to map the tetrahedra into the buckets. Notice that the update requires no multiplications or divisions, only a comparison of two float values and an increment of an integer value for the loop.

Colliding BucketTrees

When two objects are colliding (i.e., their root nodes overlap), we use a traditional tandem traversal on the two BucketTrees. However, the traversal may be pruned heavily because we are only interested in nodes that contain surface elements of the tetrahedron mesh. Therefore, only pairs of nodes are considered where at least one of the nodes has surface primitives. This is illustrated in the algorithm `overlap`. The overlap between two nodes or *AABB*s is only considered when either contains the respective object surface.

Listing 1.5.8

```
Algorithm overlap( TreeA:BucketTree, TreeB:BucketTree)
  A = root(TreeA)
  B = root(TreeB)
  push (A, B) onto queue
  while queue is nonempty do
    (A, B) pop from queue
    if stamp(A) not now or stamp(B) not now then
      continue
    end if
    if not surface(A) and not surface(B) then
      continue
    end if
    if not overlap(A,B) then
      continue
    end if
```

```
      if A and B both buckets then
        intersectionTest(A, B)
        intersectionTest(B, A)
        continue
      end if
      pick(A, B)
      if A is picked then
        descend(A, B)
      end if
      if B is picked then
        descend(B, A)
      end if
    end while
  End algorithm
```

Similarly, we only consider those possibilities containing surface primitives, when we descend in the traversal as illustrated in Algorithm `descend`: the search is terminated if neither overlapping octree nodes contains the respective surface.

Listing 1.5.9

```
Algorithm descend(A, B:Nodes)
  for all children C∈A do
    if C is nonempty then
      if surface(B) then
        push (C, B) onto queue
      else if surface(C) then
        push (C, B) onto queue
      end if
    end if
  next C
End algorithm
```

The actual intersection testing is similar to testing for self-intersection. However, this time we can overlook the neighborhood testing because we are testing primitives from two different BucketTrees. Testing if the surface of one object intersects with the volume of another is accomplished with the `intersectionTest` algorithm.

Listing 1.5.10

```
Algorithm intersectionTest(A, B)
  if surface(A) then
    S = extract surface faces of A
    for each face F∈S do
      for all tetrahedra W∈B do
        if overlap(F,W) then
          report(F, W)
        end if
      next W
    next F
  end if
End algorithm
```

For the actual overlapping testing between a triangle and a tetrahedron, we currently use *GJK* [Gilbert88] and [van den Bergen99], because it is generally applicable to any kind of convex primitive. As such, GJK allows us to support any kind of convex volumetric mesh. There are probably other choices that would give us better performance, but this remains for future research.

The way we have described BucketTree in this article suffers from two known artifacts: multiple reporting and reporting of neighboring primitives.

Because several buckets might cover a tetrahedron, it is possible that the same pair of triangle and tetrahedron can be reported multiple times during the tandem traversal.

Imagine we have a bucket in *TreeB* that overlaps two buckets in *TreeA*. If a tetrahedron is mapped to both the buckets in *TreeA*, and we have an intersecting triangle in the bucket of *TreeB*, then this intersection will be reported twice during the tandem traversal. Fortunately, it is fairly easy to adopt a bookkeeping strategy, similar to a sparse matrix representation, in order to get rid of the redundancy [Press99].

Because we work with volumetric meshes, it is reasonable to expect that the number of surface elements that intersect a volume element is very low. If we store a list of respective colliding tetrahedra in each triangle, then the testing for a reoccurrence of an intersection between a triangle and tetrahedron is nearly constant in computational complexity. Of course, we need to keep track of those triangles that were intersecting, so we can clear their lists of colliding tetrahedra from a previous query. The resulting algorithm, `report`, is shown which updates the collision and intersections list at each simulation step.

Listing 1.5.11

```
Algorithm report(A:Triangle,B:Tetrahedron)
  if collision(A) is empty then
    add A to intersections
  end if
  if B ∉ collision(A) then
    add B to collision(A)
  end if
End algorithm
```

If the overlap-testing algorithm between a triangle and a tetrahedron reports contact as an intersection, then self-intersections will be reported between triangles and tetrahedra that share a node. This is unfortunate, but we can remedy it. Either we shrink the triangle and tetrahedron by a small threshold value or we redefine the neighborhood relation, such that two tetrahedra are considered to be neighbors if they share at least a single node.

Computational Complexity

The algorithm is designed for large meshes; therefore, it is imperative that the computational complexity is low. The complexity of calculating the collision between two objects is $O(n^2/8^l)$ [Gottschalk00, p. 35], where n is maximum number of tetrahedra mapped to any bucket, and l is the number of levels in the largest BucketTree of the two objects. In the dominating situations of the tissue simulation we are performing, where two objects have small penetrations along large areas, we find that the average running time of the collision detection is $O(n^2 8^l)$, where the increase in performance is due to the pruning capability of the algorithm. The computational complexity of self-intersections is $O(n^2 8^{2l})$, corresponding to all tetrahedra mapped into a single bucket. However, typically, average running time is $O(n 8^l)$.

For our preliminary performance measurements we used V-shaped volumetric meshes. Self-intersections were measured by letting the legs of a V-shape oscillate, and intersections were measured by dragging two V-shapes through each other.

Figure 1.5.4 shows time measurements of two intersecting V-shapes of varying size. In the left graph (A) measurements was carried out on two V-shapes that had 800–1200 intersections in the right graph (B) the number of intersections is 3200–4800.

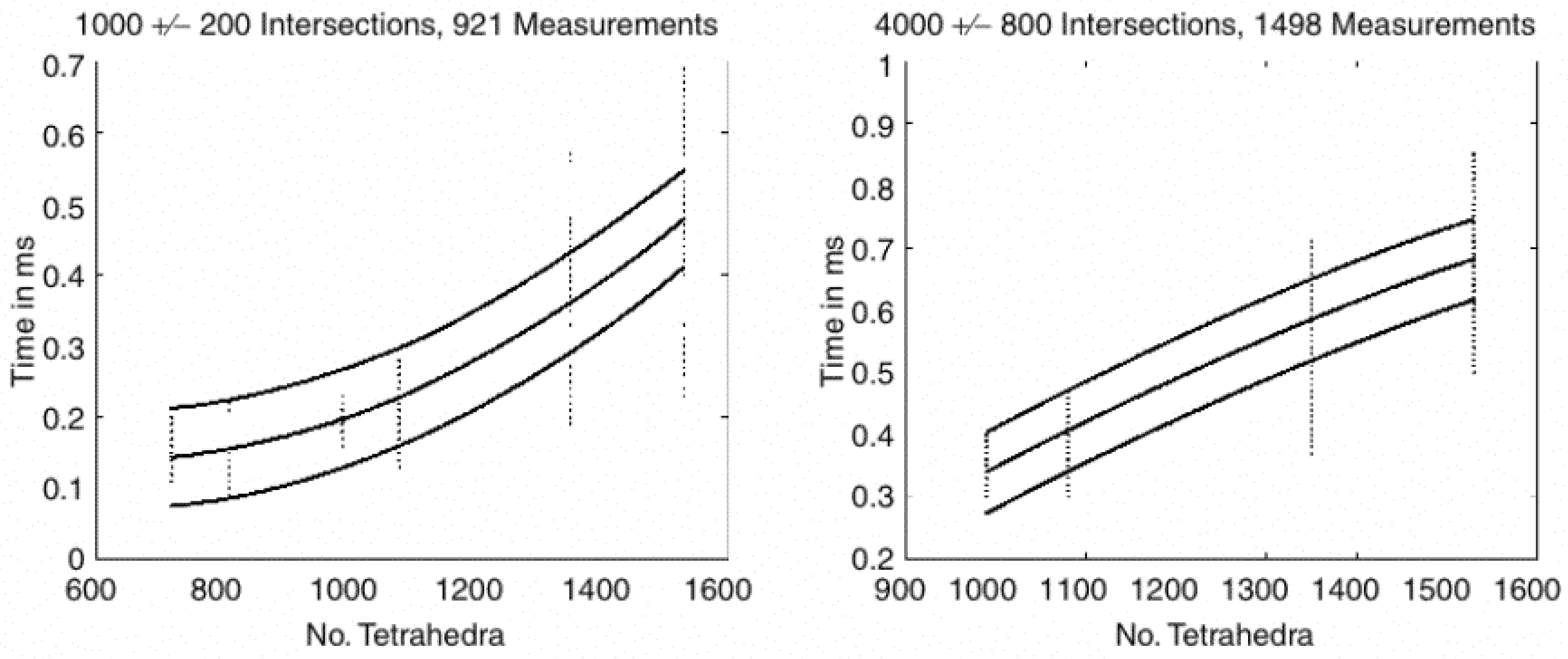

FIGURE 1.5.4 *Query time of two intersecting V-shapes of varying size. Top and bottom line show one standard deviation from middle line, which is a quadratic fit.*

Figure 1.5.5 shows how one can expect the query time to increase as a function of the number of intersections. In these measurements we used two V-shaped meshes with 600–1400 tetrahedra.

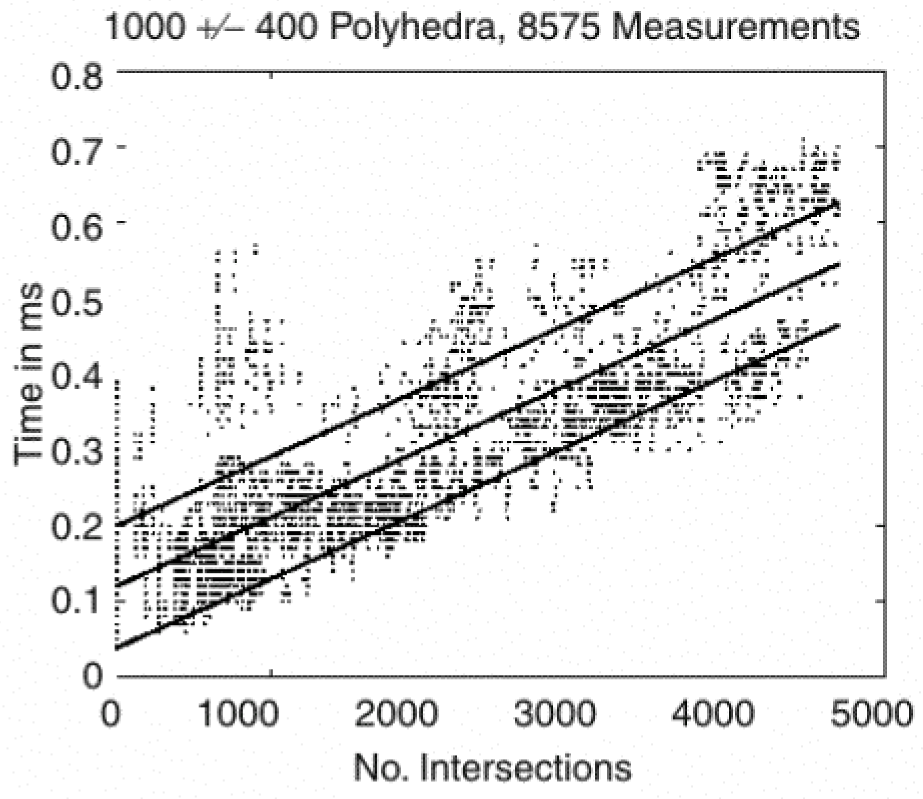

FIGURE 1.5.5 *Query time of two intersection V-shapes as a function of the number of intersections. Top and bottom line show one standard deviation from middle line, which is a linear fit.*

Figure 1.5.6 shows time measurements of a self-intersecting V-shape of varying size. The measurements was carried out on a V-shape that had 800–1200 intersections.

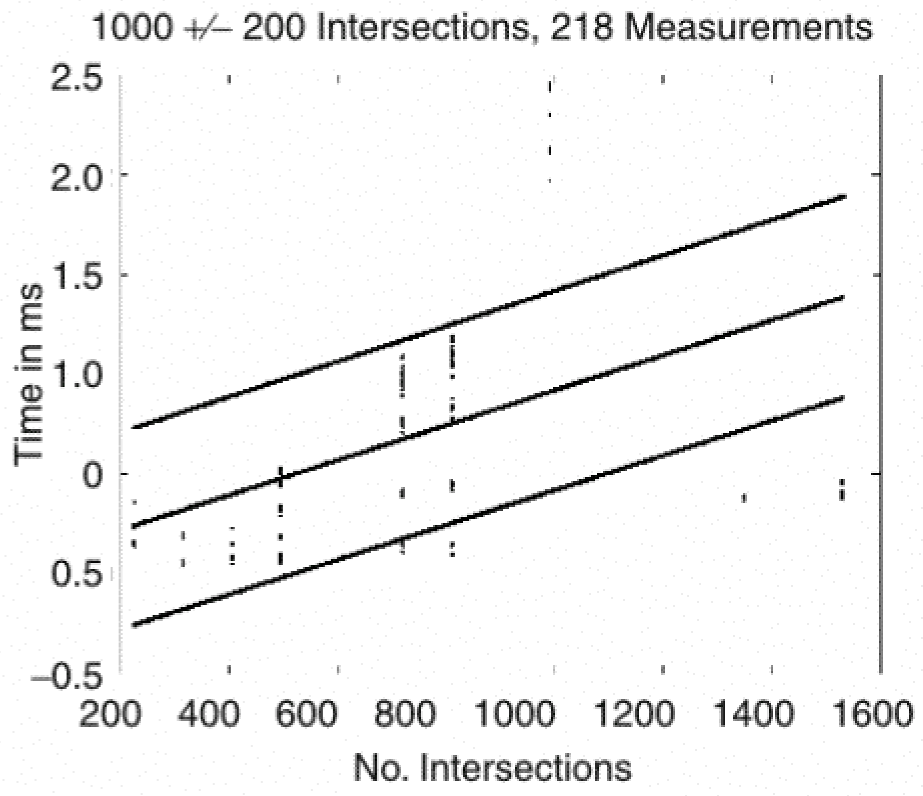

FIGURE 1.5.6 *Query time for self intersecting V-shape of varying size. Lines for linear fitting with one standard deviation are shown.*

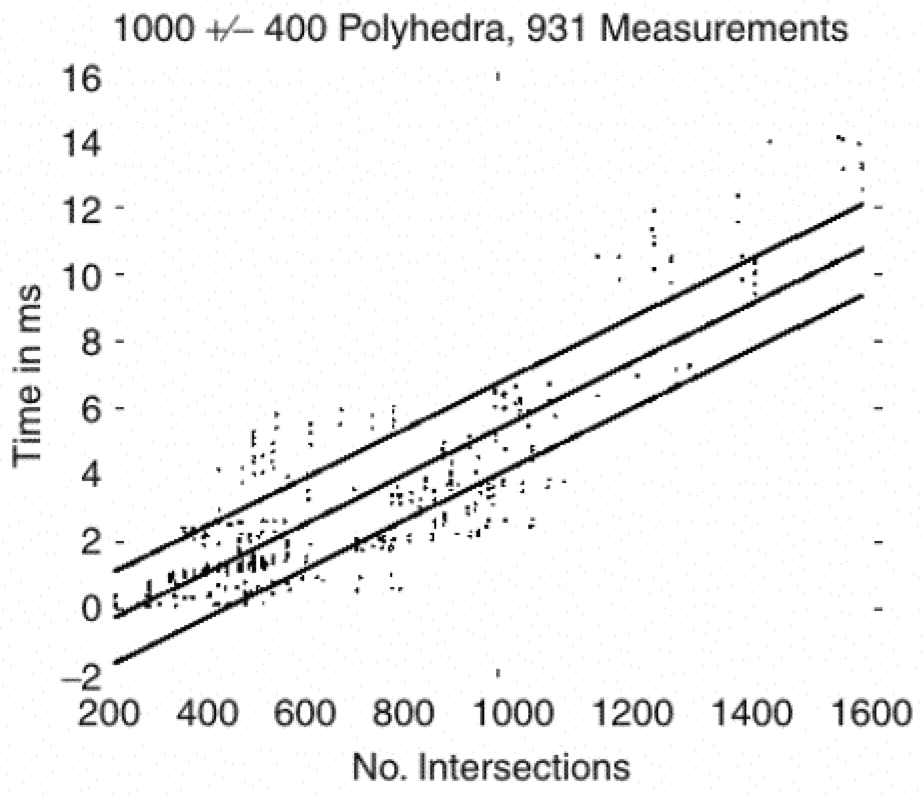

FIGURE 1.5.7 *Query time of self-intersecting V-shape as a function of the number of self-intersections. Lines for linear fitting with one standard deviation are shown.*

Figure 1.5.7 shows how one can expect the query time to increase as a function of the number of self-intersections. In these measurements we used a V-shaped mesh with 600–1400 tetrahedra.

All measurements were performed on a machine with a 2.4-MHz Xeon processor, having 4GB ram, and using Red Hat Linux 8.0.

Conclusion

The entire update and self-intersecting process in our method is similar to Mirtich's hierarchical hash tables [Mirtich96]. The main differences in our method lie in the number of levels, the cell size of each level, and the missing concept of "resolution." There also appears to be some overlap with the implementation briefly discussed in [Hirota, 2002].

With respect to the original BucketTree algorithms [Ganovelli00], we have made specific improvements.

Nodes in the BucketTree are flagged as being either empty, containing surface, or not containing surface. The flags at buckets are propagated up through the tree, and the information can be used to speed up the tandem traversal at collision. The flagging is done on the fly during the remapping procedure in each iteration of the simulator.

Self-intersections are efficiently detected during remapping by testing newly added surface primitives against previously remapped primitives.

Coordinate sorting proves to be a fast way to compute remapping and also allows for dynamic rescaling of the BucketTree to fit any kind of deformation.

We conclude that the present algorithm is very effective for detecting collisions between large meshes over large contact areas.

Our intention is to extend the BucketTree algorithm further by exploiting the geometry in a manner similar to [Joukhadar96], [Joukhadar98], [Joukhadar99], and [Sundaraj00]. The main idea is to use our BucketTree to find initial penetrating primitives and then switch to a recursive search method capable of exploiting the topology information of neighbors in the tetrahedron mesh.

For our current simulator, the major bottleneck appears to be GJK. We are currently testing alternative algorithms for doing the actual primitive collision testing to see the impact on the performance.

References

[Baraff01] Baraff, David, "Physical based modeling: Rigid body simulation," SIGGRAPH 2001 Course Notes, Pixar Animation Studios.

[Cohen94] Cohen, J., M. Ponamgi, D. Manocha, and M. C. Lin, "Interactive and Exact Collision Detection for Large-Scaled Environments," Technical Report TR94-005, Department of Computer Science, University of N. Carolina, Chapel Hill, 1994. Available online at http://www.cs.unc.edu/~lin/papers.html.

[Ganovelli00] Ganovelli, F., J. Dingliana, and C. O'Sullivan, "Buckettree: Improving collision detection between deformable objects," In Spring Conference in Computer Graphics (SCCG2000), pages pp. 156–163, Bratislava, 2000.

[Gilbert88] Gilbert, E., D. Johnson, and S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," IEEE Journal of Robotics and Automation, 4:193–203, 1988.

[Gottschalk00] Gottschalk, S., "*Collision Queries using Oriented Bounding Boxes,*" PhD thesis, Department of Computer Science, University of N. Carolina, Chapel Hill, 2000.

[Hirota02] Hirota, G., "*An Improved Finite Element Contact Model for Anatomical Simulations,*" PhD thesis, University of N. Carolina, Chapel Hill, 2002.

[Hirota01] Hirota, G., S. Fisher, A. State, C. Lee, and H. Fuchs, "An implicit finite element method for elastic solids in contact," In Computer Animation, Seoul, South Korea, 2001.

[Joukhadar98] Joukhadar, A., A. Deguet, and C. Laugier, "A collision model for rigid and deformable bodies," In IEEE Int. Conference on Robotics and Automation, Vol. 2, pages pp. 982–988, Leuven (BE), 1998.

[Joukhadar99] Joukhadar, A., A. Scheuer, and C. Laugier, "Fast contact detection between moving deformable polyhedra," In IEEE-RSJ Int. Conference on Intelligent Robots and Systems, Vol. 3, pages pp. 1810–1815, Kyongju (KR), 1999.

[Joukhadar96] Joukhadar, A., A. Wabbi, and C. Laugier, "Fast contact localisation between deformable polyhedra in motion," In IEEE Computer Animation Conference, pages pp. 126–135, Geneva (CH), 1996.

[Larsson01] Larsson, T. and T. Akenine-Möller, "Collision detection for continuously deforming bodies," In Eurographics, pages 325–333, 2001.

[Mirtich96] Mirtich, B. "*Impulse-based Dynamic Simulation of Rigid Body Systems,*" PhD thesis, University of California, Berkeley, 1996.

[O'Rourke93] O'Rourke, Joseph, *Computational Geometry in C*, Cambridge University Press, New York: p. 26, 1993.

[Olsen97] Olsen, O. F., and M. Nielsen "Multi-scale gradient magnitude watershed segmentation," In Bimbo, A. D., editor, ICIAP '97—9th Int. Conf. on Image Analysis and Processing, volume 1310 of Lectures Notes in Computer Science, pages 6–13. Springer, 1997.

[Press99] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical recipes in C. Cambridge University Press, 2nd edition, 1999.

[Sundaraj00] Sundaraj, K. and C. Laugier, "Fast contact localisation of moving deformable polyhedras," In IEEE Int. Conference on Control, Automation, Robotics and Vision, Singapore (SG), 2000.

[van den Bergen97] van den Bergen, Gino, "Efficient collision detection of complex deformable models using AABB trees," Journal of Graphics Tools, 2(4):1–13, 1997

[van den Bergen99] van den Bergen, Gino, "A fast and robust GJK implementation for collision detection of convex objects," Journal of Graphics Tools, 4(2):7–25, 1999

[Volino98] Volino, P. and N. M. Thalmann, "Collision and self-collision detection: Efficient and robust solutions for highly deformable surfaces," Technical report, MIRALab, 1998.

[Zienkiewicz00] Zienkiewicz, O. and R. Taylor, "The Finite Element Method: The basis," volume 1. Butterworth-Heinemann, 5th edition, 2000.