Algorithms and Data Structures for Haptic Rendering: Curve Constraints, Distance Maps, and Data Logging

Dan Morris
Stanford University Robotics Lab
Computer Science Department
Stanford, CA 94305-9010
dmorris@cs.stanford.edu

ABSTRACT

In this paper, we describe three novel data processing techniques used for haptic rendering and simulation:

- We present an approach to constraining a haptic device to travel along a discretely-sampled curve.
- We present an approach to generating distance maps from surface meshes using axis-aligned bounding box (AABB) trees. Our method exploits spatial coherence among neighboring points.
- We present a data structure that allows thread-safe, lock-free streaming of data from a high-priority haptic rendering thread to a lower-priority data-logging thread.

We provide performance metrics and example applications for each of these techniques. C++-style pseudocode is provided wherever possible and is used as the basis for presenting our approaches. Links to actual implementations are also provided for each section.

Categories and Subject Descriptors

H.5.2 [Information Interfaces and Presentation]: User Interfaces – Haptic I/O; H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems

General Terms

Algorithms, Human Factors

Keywords

Haptics, haptic rendering, virtual fixtures, distance maps, synchronization, threads, voxelization, flood-filling, kd-tree, curve constraints

1. INTRODUCTION

Applications incorporating haptic feedback are subject to significant performance constraints; it is generally accepted that an application needs to sustain a 1kHz haptic update rate before sampling effects become perceptible.

This stringent computation-time limitation requires careful consideration of the design and implementation of preprocessing, rendering, and data streaming techniques. In this paper, we present three techniques for optimized haptic data processing, each in an individual section of the paper. Section 2 will discuss the implementation of a haptic curve constraint, or "virtual fixture", using kd-trees. Section 3 will discuss the rapid (offline) generation of exact signed distance fields for surface meshes. Section 4 will discuss a threaded data structure for lock-free streaming of data from a high-priority haptic rendering thread to a lower-priority disk-interaction thread.

2. HAPTIC CURVE CONSTRAINTS

2.1 Background

Haptic devices generally provide a user with three or six degrees of freedom. Haptic feedback, however, offers the possibility of dynamically reducing the effective degrees of freedom available within the device's workspace via virtual constraints.

Non-penetration constraints associated with surfaces are extremely common and are used in nearly every haptic simulation involving interaction with rigid objects, but other types of constraints have been applied using haptic devices as well. Abbott et al [1] propose "virtual fixtures" to assist in dexterous manipulation; the goal is to reduce the degrees of freedom involved in a complex task and/or to restrict a device's motion to a "safe" portion of the workspace. This may be particularly suitable for robotic surgery applications in which an actuated master can assist the surgeon by restricting the movement of the slave. The authors discuss multiple types of fixtures, including a "guidance virtual fixture" (GVF), which is a constraint associated with a 3D curve. Garroway and Hayward [2] constrain the user to an analytic curve to assist in editing a spatial trajectory.

In both of these cases, it is assumed that the closest point on the curve to the current haptic probe position and/or the distance to that point are readily available, either by analytic computation or by explicitly tracking the progress of the haptic probe along the curve.

2.2 Discretized Curve Constraints

For some applications, particularly those where constraints can be dynamically added and removed, it may be necessary to constrain a user to a curve beginning at an arbitrary starting point, or to recover when the constraint has been significantly violated. It is thus necessary to rapidly find the closest point on a curve to the current haptic probe position.

In addition, analytic representations are not always available for curves; curves are most generally represented as discretely-sampled, ordered point sets rather than analytic functions. This is particularly useful, for example, for haptic training applications (e.g. [3,4,5]), in which one might use previously-recorded trajectories as teaching examples.

We thus provide a rapid method for finding the closest point on a discretely-sampled curve to a current probe position. We also present an approach to tracking the constraint position on a curve when the haptic device may deviate from the constraint and approach other points on the curve to which it should not become constrained. Tying the haptic device to this constraint position by a virtual spring will provide a general-purpose curve constraint.

A curve is assumed to be represented as a series of N points, each of which stores its 3-dimensional position, an index into a linked-list or flat array that stores the N points in order, and its arcposition along the curve (the curve runs from arcposition 0.0 to arcposition 1.0). The curve is not required to have a uniform sampling density. Each point p_i (for $i \neq 0$ and $i \neq (N-1)$) is implicitly part of two line segments, $[p_{i-1} \rightarrow p_i]$ and $[p_i \rightarrow p_{i+1}]$. For clarity, I will provide C++ pseudocode of the relevant data structures and computations throughout this section, beginning with the representation of the curve and sample points. The 'vector' class is assumed to represent a 3-dimensional vector and to support the expected operators.

```
struct curvePoint {
  vector pos;
  unsigned int index;
  float arcpos;
};

struct curve {
  unsigned int N;
  curvePoint* points;
};
```

All of these points are then placed in a standard kd-tree [6] (a 3d-tree in this case). A kd-tree stores a point set and provides efficient retrieval of the subset of points that lie within a bounding rectangle. This can be generalized at minimal cost to return the approximate nearest K neighbors to a given test point. We will assume that our kd-tree provides the following function, which returns the *K* points closest to *testPoint*:

```
void search(
       vector testPoint,
       int K, curvePoint* points);
```

At each timestep at which a haptic constraint force is requested, we use this interface to find the N closest points to the device position p_{dev}. N is chosen empirically; higher values of N require more computation time but reduce the occurrence of incorrect forces resulting from sparse sampling of the curve. Figure 1 demonstrates this problem and illustrates why using N=1 does not generally give correct results.

```
// Get the points closest to the device
vector pdev = getDevicePosition();
curvePoint neighbors[N];
myKDTree.search(pdev, N, neighbors);
```

The N returned points are sorted by index, and for each returned point p_i we build the two associated line segments ($[p_{i-1} \rightarrow p_i]$ and $[p_i \rightarrow p_{i+1}]$) and insert them into an ordered list of candidate line segments that might contain the closest point to our haptic probe. This ordering reduces redundancy; we now have a maximum of (but generally less than) 2N line segments to search. We can compactly represent each line segment as its first index, so we can store the candidate set as an ordered, non-redundant list of indices:

```
// Sort the candidate line segments by index
```

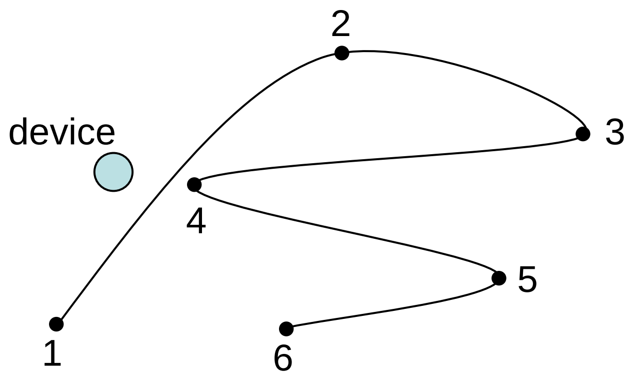

Figure 1. The device should be constrained to the segment between vertices 1 and 2, but sparse sampling of the curve places it closer to vertex 4 than to either of these vertices. This motivates the use of a broader nearest-neighbor search to handle this case properly.

```
std::set<unsigned int> candidateSegments;
for(unsigned int i=0; i<N; i++)
  candidateSegments.insert(neighbors[i]);
```

Now for each of those candidate segments, we compute the smallest distance between our device position p_{dev} and the segment, using the approach presented (and available online) in [7]. We assume we have a function `distanceToSegment` that takes a test position and a segment defined by the indices of its two endpoints and returns the corresponding distance and point of closest approach (as a t-value, where 0.0 is the segment start and 1.0 is the segment endpoint). We find the segment with the smallest distance to the haptic device point:

```
// Find the point of closest approach among
// all candidate segments

struct distanceRecord {
  int segmentIdx;
  float t;
  float distance;
};

float shortestDistance = FLT_MAX;
distanceRecord closest;
std::set<unsigned int>::iterator iter;

// Loop over all candidate segments
for(iter=candidateSegments.begin();
  iter != candidateSegments.end(); iter++) {

  int index = *iter;
  float t;

  // What's the smallest distance to this
  // segment?
  float distance =
    distanceToSegment(pdev,index,index+1,t);
  distanceRecord dr(index,t,distance);

  // Is this the smallest distance
  // we've found so far (to any segment)?
  if (distance < shortestDistance) {
```

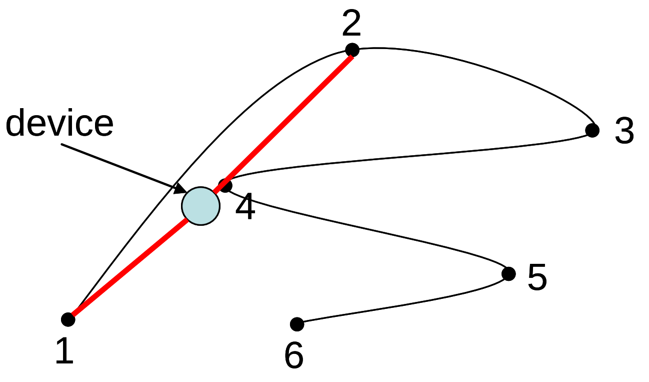

Figure 2. The device passes through vertex 4 on its way between vertices 1 and 2, but should still be constrained to segment [1,2] to guide the user along the correct curve shape.

```
    closest = dr;
    shortestDistance = distance;
  }

}
```

For most curves, it is now sufficient to simply apply a constraint force pulling the device toward the closest point on the closest segment with stiffness $k_{constraint}$:

```
// Generate a constraint force pulling
// the haptic device toward the closest
// point on the curve

vector start =
  myCurve.points[closest.segmentIdx].pos;
vector end =
  myCurve.points[closest.segmentIdx +1].pos;
vector closestPoint =
  start + (end - start) * closest.t;
vector force =
  kconstraint * (closestPoint - pdev);
```

This approach, however, fails in the case illustrated in Figure 2. Here, due to normal deviation from the constraint path (resulting from limited stiffness), the device passes through vertex 4 on its way between vertices 1 and 2, but should still be constrained to segment [1,2] to guide the user along the correct curve shape. This can be handled by a modification to our distance-computation function, which takes into account the *arcdistance* of the point to which the haptic device was most recently constrained. Essentially, when choosing the closest point on the curve, we want to penalize points that are far from the test point both in Euclidean distance and in arclength.

We assume that the distance computation function is provided the arcposition of the point to which the device was previously constrained (or a flag indicating that this is a new constraint and there is no previous state, in which case the distance returned is just the usual Euclidean distance). For each segment we process, we find the closest point on that segment to the haptic device and compute the corresponding Euclidean distance as usual. We then take the absolute difference in arcposition between this point and the previous constraint point, multiply it by an empirically-selected penalty factor, and return this weighted "score" as our distance value in the above routine (this pseudocode replaces the distance computation in the above routine):

```
// known from our previous iteration
float previousArcPos;

float distance = distanceToSegment(pdev,
  index,index+1,t);

// Find the arcposition of the closest
// point of approach on this segment
float newArcPos =
  (myCurve.points[index].arcpos*t)
  +
  (myCurve.points[index+1].arcpos*(1.0-t));

// Find the arcdistance between this test
// point and my previous constraint position
float arcidst =
  fabs(previousArcPos - newArcPos);

// Weight our 'distance' value according to
// this arcposition.
distance = distance +
  arcidst * ARC_PENALTY_WEIGHT;
```

Higher values of ARC_PENALTY_WEIGHT maximally eliminate "jumping" along the curve (Figure 2). However, inappropriately high values may cause friction-like effects as the user rounds sharp corners in the curve and is prevented from "jumping" around corners when he *should* be allowed to move to subsequent segments. We have found this effect to be imperceptible for a wide range of values of ARC_PENALTY_WEIGHT (see Section 2.3).

2.3 Implementation and Results

The above algorithm was implemented in C++ using the public-domain kd-tree available in [8], a Phantom haptic device [9], and the CHAI 3D libraries for haptics and visualization [10]. Curves were generated according to [5], with 2000 points. N (number of nearest neighbors to search) was set to 100, with the arc penalty weight set to 1.0.

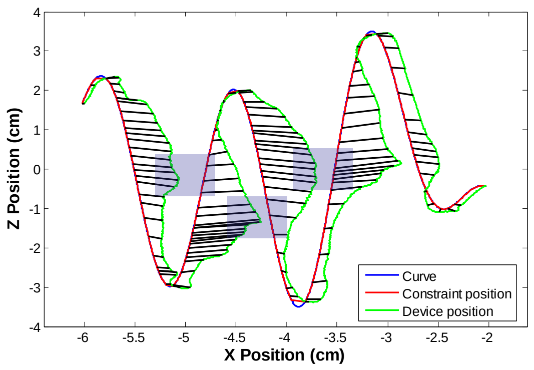

Figure 3. Black lines indicate correspondences between device position (green) and constraint position (red). The highlighted areas show regions where the device approached a region on the curve that was distant in terms of arclength and was thus appropriately constrained to the current curve segment, despite being physically closer to the "incorrect" segment.

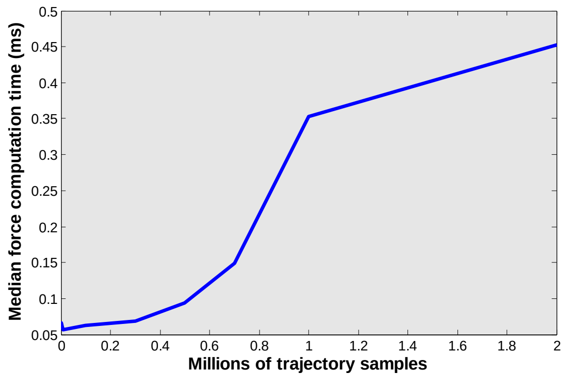

Figure 4. Increase in computation time with increasing trajectory size, N (number of neighbors used) fixed at 100. The increase is approximately linear, but even with two million samples, the computation time is well under 1ms.

Figure 3 demonstrates the robustness of our approach. We see the actual path of the device in green, constrained by force vectors (indicated in black) to the curve. We see several regions (highlighted in blue) where the device very closely approaches a region of the curve that is distant from the current constraint position in terms of arclength, and the constraint position correctly remains on the current region of the curve.

For the constant values presented above, mean computation time per haptic iteration on a 1GHz Pentium 4 was 0.2ms, well below the accepted perceptual threshold of 1ms per haptic computation. Figure 4 shows the dependence of computation time on the number of samples in the trajectory for a fixed N (number of neighbors used in constraint search). We see that even with very large trajectories (up to two million samples), computation time is well below 1ms. Figure 5 shows the dependence of computation time on N (number of neighbors used in constraint search) for a fixed trajectory size. Although this increase is also approximately linear, there is a much more expensive constant factor associated with increasing N, since this increases the number of floating-point distance computations.

As a final point, we note that our distance-computation function generates the closest point returned from each point/segment comparison, so this is the only part of our overall approach that would need to be modified to represent line segments as Bezier curve segments or other interpolation functions.

An implementation of the algorithm discussed here is included as part of our "haptic mentoring" experimental platform [5], available at:

http://cs.stanford.edu/~dmorris/haptic_training

3. DISTANCE MAP GENERATION

3.1 Terminology

For an object **O** on $\Re^n$ and a set of points **P** on $\Re^n$, the *distance field* is defined as the smallest distance from each point in **P** to a point on **O**. The distance metric is generally Euclidean distance, but any symmetric, non-negative function satisfying the triangle inequality can serve as a distance metric. The *distance map* is the distance field annotated with the *position* of the closest point on

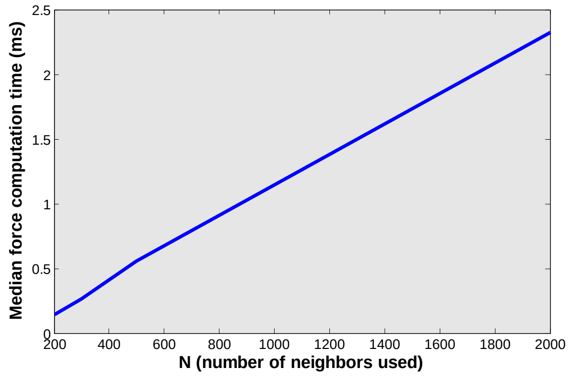

Figure 5. Increase in computation time with increasing N (number of neighbors used), trajectory size fixed at 2000.

O for each point in **P**. When **O** is an orientable surface (a surface that partitions $\Re^n$ into two subspaces), the sign of the stored distance at a point indicates the subspace in which that point lies (in particular, this sign is often used to indicate whether a point is inside or outside a closed surface **O**). The *distance transform* takes a set of points **P** and an object **O** and annotates **P** with a distance map on **O**. The closely-related *closest-point transform* takes a set of points **P** and an object **O** and annotates each point in **P** with the location of the closest point on **O**, without distance information. The closest-point transform is computed by definition whenever a distance map is generated.

3.2 Background

The distance map is an implicit object representation with extensive applications in computer graphics, for example in physical simulation [13] and isosurface generation [14].

Distance maps have also been applied in haptics, to search for collisions between a haptic tool and the environment [15], to provide constraint forces when navigating a volume [16], and to constrain a surface contact point to the boundary of a region [17].

Several methods have been proposed for computing distance fields, distance maps, and closest point transforms. Many applications in computer animation use the approximate but extremely efficient Fast Marching Method [18]. [19] proposes a method based on Voronoi regions and local rasterization, and provides an open-source implementation [20]. More recently, approaches have emerged that use parallel graphics hardware to accelerate distance field computation [21].

We propose an alternative method for generating exact distance maps from point sets to triangle meshes that leverages bounding-box structures, which are already generated as a preprocessing step for many interactive applications in haptics and graphics.

3.3 Distance Map Generation

The following procedure assumes that we are given a list of points **P**, preferably sorted in an order that promotes spatial coherence (this is generally the case in practice, where regular voxel grids are used as the point set and sorting is trivial). We are also given a set of triangles **M**, which represent one or more logical objects in a scene.

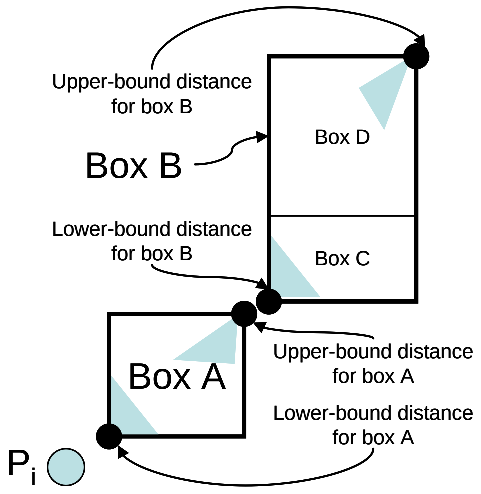
Upper-bound distance for box B
Box B
Box D
Box C
Lower-bound distance for box B
Box A
Upper-bound distance for box A
Lower-bound distance for box A
P_i

Figure 6. Distance transformation for point P_i. If we've processed Box A before we process Box B, we will not descend to Box B's children, because Box B's lower-bound distance is greater than Box A's upper-bound distance.

We further assume that a bounding-volume hierarchy has been built on **M**. A strength of this approach is that it leverages common bounding-volume techniques, which are used in a variety of existing applications in haptics and graphics. Without loss of generality, we will assume that the hierarchy is composed of axis-aligned bounding boxes (AABB's). Further details on the construction and characteristics of AABB trees can be found in [22].

The general approach to finding the closest point on **M** to a point $\mathbf{P_i}$ in **P** is to descend the AABB tree, computing lower and upper bounds on the distance to each box we descend, and tracking the lowest upper bound $\mathbf{d_{lu}}$ we've encountered so far (the lowest "guaranteed" distance). If the lower bound for a box is farther from $\mathbf{P_i}$ than $\mathbf{d_{lu}}$, we can skip this box (see Figure 6). Using this culling approach and exploiting spatial coherence among subsequent points in **P** by selectively mixing breadth-first and depth-first examination of our bounding volume hierarchy, we can build distance maps in a manner that is both efficient and heavily parallelizable.

In the following pseudocode, we assume without loss of generality that the AABB tree representing our triangle mesh is in the same coordinate frame as our point list; in practice coordinate transformations are performed before distance computation begins. We also assume for clarity of terminology that the list of points **P** is a series of voxel locations (this is the case when computing the distance transform on a regular grid), so we refer to the $\mathbf{P_i}$'s as "voxels" and locations on the surface **M** as "points".

```
// A simple AABB tree hierarchy

// A generic tree node maintaining only a
// parent pointer.  This pseudocode avoids
// pointer notation; all links within the
// tree and all references to AABBNode's in
// the code should be read as pointers.
struct AABBNode { AABBNode parent; };

// A structure representing a bounding box
// and pointers to child nodes.
struct AABBox : public AABBNode {

  // the actual bounding box
  vector3 xyzmax, xyzmin;

  // my children in the AABB tree
  AABBNode left, right;
}

// A structure representing a leaf node
struct AABBLeaf : public AABBNode {
  triangle t;
}

// The inputs to our problem

// The Pi's
std::list<vector3> voxels;

// The triangle set M, pre-processed into
// an AABB tree
AABBox tree_root;

// All the boxes we still need to look at
// for the current voxel.  This may not be
// empty after a voxel is processed; placing
// nodes here to be used for the next voxel
// is our mechanism for exploiting spatial
// coherence.
std::list<AABBNode> boxes_to_descend;

// The smallest squared distance to a
// triangle we've seen so far for the
// current voxel...
//
// We generally track squared distances,
// which are faster to compute than actual
// distances.  When all is said and done,
// taking the square root of this number
// will give us our distance value for this
// voxel.
float lowest_dist_sq = FLT_MAX;

// The point associated with this distance
vector3 closest_point;

// The tree node associated with the closest
// point.  We store this to help us exploit
// spatial coherence when we move on to our
// next voxel.
//
// This will always be a leaf.
AABBNode closest_point_node;

// The lowest upper-bound squared distance
// to a box we've seen so far for the
// current voxel.
float lowest_upper_dist_sq = FLT_MAX;

// Process each voxel on our list, one
// at a time...

std::list<vector3>::iterator iter =
  voxels.begin();

while (iter != voxels.end) {
  // Grab the next point
  vector3 v = (*iter);

  // Now we’re going to find the closest
  // point in the tree (tree_root) to v...
  //
  // See below for the implementation of
  // find_closest_point.
  find_closest_point(v);

  // Now output or do something useful
  // with lowest_dist_sq and closest_point;
  // these are the values that should be
  // associated with v in our output
  // distance map...
  do_something_useful();

  // So it’s time to move on to the next
  // voxel.  We’d like to exploit spatial
  // coherence by giving the next voxel
  // a "hint" about where to start looking
  // in the tree.  See the explanation below
  // for what this does; the summary is that
  // it seeds 'boxes_to_descend' with a
  // good starting point for the next voxel.
  seed_next_voxel_search();

}

// Find the closest point in our mesh to
// the sample point v
void find_closest_point(vector3 v) {

  // Start with the root of the tree
  boxes_to_descend.push_back(tree_root);

  while(!(boxes_to_descend.empty)) {
    AABBNode node =
      boxes_to_descend.pop_front();
    process_node(node,v);
  }

}

// Examine the given node and decide whether
// we can discard it or whether we need to
// visit his children.  If it’s a leaf,
// compute an actual distance and store
// it if it’s the closest so far.
//
// Used as a subroutine in the main voxel
// loop (above).
void process_node(AABBNode node, vector3 v){

  // Is this a leaf?  We assume we can get
  // this from typing, or that the actual
  // implementation uses polymorphism and
  // avoids this check.
  bool leaf = isLeaf(node);

  // If it’s a leaf, we have no more
  // descending to do, we just need to
  // compute the distance to this triangle
  // and see if it’s a winner.
  if (leaf) {

    // Imagine we have a routine that finds
    // the distance from a point to a
    // triangle; [7] provides an optimized
    // routine with a thorough explanation.
    float dsq;
    vector3 closest_pt_on_tri;

    // Find the closest point on our
    // triangle (leaf.t) to v, and the
    // squared distance to that point.
    compute_squared_distance(v,leaf.t,
      dsq,closest_pt_on_tri;

    // Is this the shortest distance so far?
    if (dsq < lowest_dist_sq) {

      // Mark him as the closest we’ve seen
      lowest_dist_sq = dsq;
      closest_point = clost_pt_on_tri;
      closest_point_node = node;

      // Also mark him as the "lowest upper
      // bound", because any future boxes
      // whose lower bound is greater than
      // this value should be discarded.
      lowest_upper_dist_sq = dsq;
    }

    // This was a leaf; we’re done with him
    // whether he was useful or not.
    return;
  }

  // If this is not a leaf, let’s look at
  // his lower- and upper-bound distances
  // from v.
  //
  // Computing lower- and upper-bound
  // distances to an axis-aligned bounding
  // box is extremely fast; we just take
  // the farthest plane on each axis
  float best_dist = 0;
  float worst_dist = 0;

  // If I'm below the x range, my lowest
  // x distance uses the minimum x, and
  // my highest uses the maximum x
  if (v.x < node.box.xyzmin.x) {
    best_dist += node.box.xyzmin.x - v.x;
    worst_dist += node.box.xyzmax.x - v.x;
  }

  // If I'm above the x range, my lowest x
  // distance uses the maximum x, and my
  // highest uses the minimum x
  else if (v.x > node.box.xyzmax.x) {
    best_dist += v.x - node.box.xyzmax.x;
    worst_dist += v.x - node.box.xyzmin.x;
  }

  // If I'm _in_ the x range, x doesn't
  // affect my lowest distance, and my
  // highest-case distance goes to the
  // _farther_ of the two x distances
  else {
    float dmin =
       fabs(node.box.xyzmin.x - v.x);
    float dmax =
      fabs(node.box.xyzmax.x - v.x);
    double d_worst = (dmin>dmax)?dmin:dmax;
    worst_dist += d_worst;
  }

  // Repeat for y and z...

  // Convert to squared distances
  float lower_dsq = best_dist * best_dist;
  float upper_dsq = worst_dist * worst_dist;

  // If his lower-bound squared distance
  // is greater than lowest_upper_dist_sq,
  // he can't possibly hold the closest
  // point, so we can discard this box and
  // his children.
  if (lower_dsq > lowest_upper_dist_sq)
    return;

  // Check whether I'm the lowest
  // upper-bound that we've seen so far,
  // so we can later prune away
  // non-candidate boxes.
  if (upper_dsq < lowest_upper_dist_sq) {
    lowest_upper_dist_sq = upper_dsq;
  }

  // If this node _could_ contain the
  // closest point, we need to process his
  // children.
  //
  // Since we pop new nodes from the front
  // of the list, pushing nodes to the front
  // here results in a depth-first search,
  // and pushing nodes to the back here
  // results in a breadth-first search.  A
  // more formal analysis of this tradeoff
  // will follow in section 3.4.
  boxes_to_descend.push_front(node.left);
  boxes_to_descend.push_front(node.right);

  // Or, for breadth-first search...
  // boxes_to_descend.push_back(node.left);
  // boxes_to_descend.push_back(node.right);
}
```

When we've finished a voxel and it's time to move on to the next voxel, we'd like to exploit spatial coherence by giving the next voxel a "hint" about where to start looking in the tree. We expect the node that contains the closest point to the next voxel to be a "near sibling" of the node containing the closest point to the current voxel, so we'll let the next voxel's search begin at a nearby location in the tree by walking a couple nodes up from the best location for this voxel.

The constant TREE_ASCEND_N controls how far up the tree we walk to find our "seed point" for the next voxel. Higher values assume less spatial coherence and require more searching in the case that the next voxel is extremely close to the current voxel. Lower values assume more spatial coherence and optimize the case in which subsequent voxels are very close, while running a higher risk of a complete search.

Section 3.4 discusses the selection of an optimal value for TREE_ASCEND_N.

```
void seed_next_voxel_search() {

  // Start at the node that contained our
  // closest point and walk a few levels
  // up the tree.
  AABBNode seed_node = closest_point_node;
  for(int i=0; i<TREE_ASCEND_N; i++) {
    if (seed_node.parent == 0) break;
    else seed_node = seed_node.parent;
  }

  // Put this seed node on the search list
  // to be processed with the next voxel.
  boxes_to_descend.push_back(seed_node);

}
```

In summary, for each voxel in $\mathbf{P}_i$ we track the lowest upper-bound distance that we've found for a box as we descend our AABB tree, and discard boxes whose lower-bound distance is larger. When we reach a leaf node, we explicitly compute distances and compare to the lowest distance we found so far. We exploit spatial coherence when processing a voxel by first searching a small subtree in which we found the closest point for the previous voxel.

3.4 Implementation and Results

The approach presented here was evaluated in the context of generating internal distance fields (finding and processing only voxels that lie inside a closed mesh) during the process of voxelization. *Voxelizer* is an application written in C++ that loads meshes and uses a flood-filling process to generate voxel representations of those meshes, optionally including distance fields. Both the flood-filling and the distance-field generation use the public-domain AABB tree available in CHAI [10].

To evaluate the suitability of our approach and the benefit of our exploitation of spatial coherence, we generated voxel arrays and

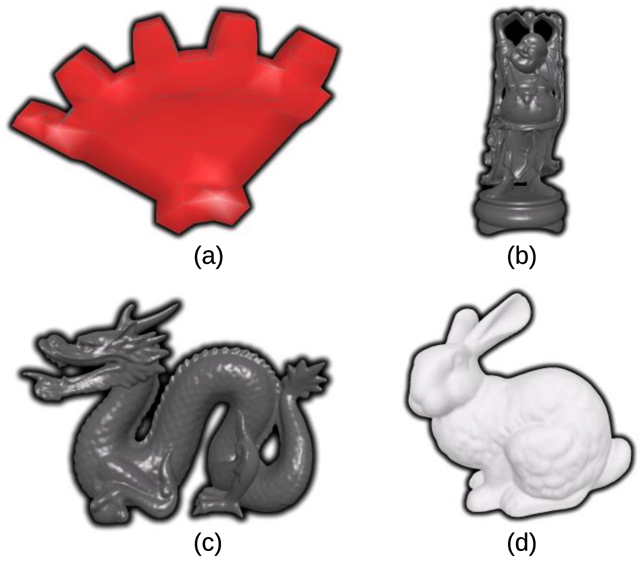

Figure 7. Meshes used for evaluating distance map computation. (a) Gear: 1000 triangles. (b) Happy: 16000 triangles. (c) Dragon: 203000 triangles (d) Bunny: 70,000 triangles.

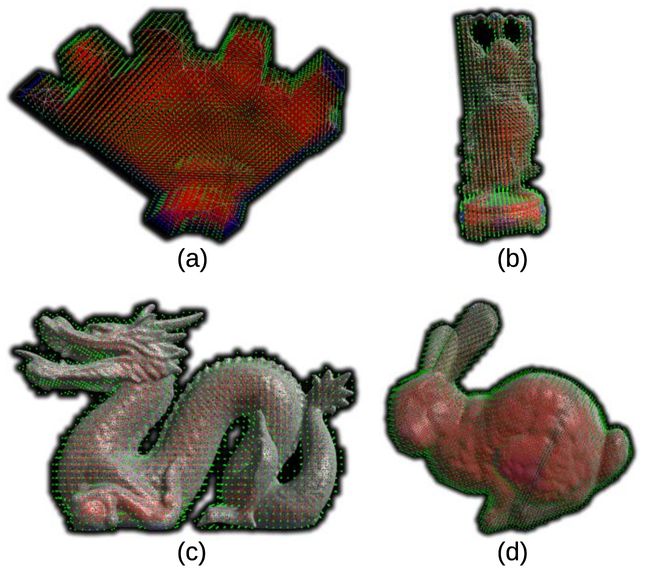

Figure 8. The same meshes displayed in Figure 7, after using the *voxelizer* application to identify internal voxels (voxel centers are in green for surface voxels and red for internal voxels) by flood-filling. The long axis resolution in each case here is 50 voxels.

distance fields for a variety of meshes (Figures 7 and 8) at a variety of voxel densities and a variety of values for TREE_ASCEND_N (see above). Furthermore, at each parameter set, we generated distance fields using both depth- and breadth-first search. The following sections discuss the performance results from these experiments.

Overall Performance

Table 1 shows the computation time for flood-filling and distance-

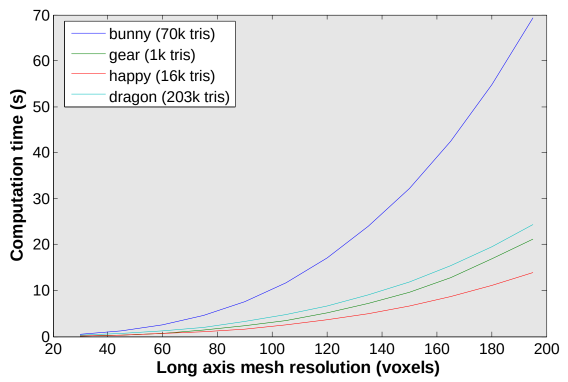

Figure 9. Performance of our distance-map computation approach on all four meshes at a variety of mesh resolutions.

field generation for each of the four test meshes at a variety of resolutions. The voxel arrays generated represent surface and internal voxels only; the full distance field for voxels outside the mesh is not generated. "Long axis resolution" indicates the number of voxels into which the longest axis of the mesh's bounding-box is divided; voxels are isotropic so the resolutions of the other axes are determined by this value.

We note that for small resolutions, on the order of 30 voxels, times for distance computation are interactive or nearly interactive, even for complex meshes. We also note that in general, distance computation represents the significant majority of the total time required to perform the combined flood-filling and distance-field generation (on average, distance-field generation represents 86% of the total time).

Figure 9 shows the dependence of computation time on long axis resolution for all four meshes. As expected, all meshes display an exponential increase in computation time as voxel resolution increases, but even at very high resolutions, computation time is

Mesh	Triangles	Long axis resolution	Voxels	Total time (s)	Distance time (s)
bunny	70k	30	7168	0.736	0.683
bunny	70k	75	95628	6.107	5.282
bunny	70k	135	529024	29.033	25.258
bunny	70k	195	1561728	82.341	71.585
gear	1k	30	4156	0.144	0.117
gear	1k	75	54270	1.751	1.383
gear	1k	135	286813	9.228	7.282
gear	1k	195	829321	27.137	21.387
happy	16k	30	2020	.13495	.1177
happy	16k	75	25308	1.387	1.208
happy	16k	135	132910	6.132	5.261
happy	16k	195	381120	16.956	14.48
dragon	203k	30	2550	0.494	0.47
dragon	203k	75	31674	3.158	2.859
dragon	203k	135	164061	11.839	10.558
dragon	203k	195	468238	30.13	26.633

Table 1. A comparison of flood-filling and distance-computation times for all four meshes at a variety of voxel resolutions.

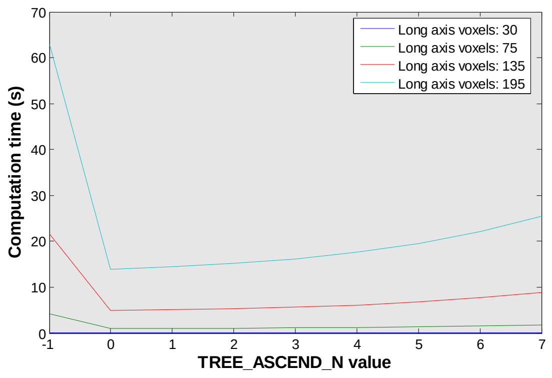

Figure 10. Performance benefit of exploiting spatial coherence and optimal value selection for TREE_ASCEND_N (results shown here are for the "happy" mesh). A value of -1 indicated that spatial coherence was not exploited at all. A value of 0 indicated that the "hint" node was the leaf node (a single triangle) that contained the shortest distance for the previous voxel.

tractable for preprocessing applications (only above one minute for one of the four meshes and only above a long axis resolution of 180 voxels).

Spatial Coherence

To analyze the benefit of exploiting spatial coherence in distance-map computation, and to identify the optimal value of TREE_ASCEND_N (the number of tree levels to step up in generating a "hint" location for the next voxel's distance search), voxel arrays and distance fields were generated for all four meshes with various values of TREE_ASCEND_N. Figure 10 shows the results for the "happy" mesh (this mesh was chosen arbitrarily; results were similar for all four meshes). A TREE_ASCEND_N value of -1 indicated that spatial coherence was not exploited at all; i.e. every distance search started at the top of the tree. A value of 0 indicated that the "hint" node was the *leaf* node (a single triangle) that contained the shortest distance for the previous voxel.

Exploiting spatial coherence yields five-fold improvement in performance (a reduction in distance field time from 62 seconds to 13 seconds) for the largest resolution shown in Figure 10. This corresponds to the difference between TREE_ASCEND_N values of 0 and 1. Further increasing TREE_ASCEND_N does not further improve performance; it is clear in Figure 10 that zero is the optimal value. This is equivalent to assuming that locality extends as far as the closest triangle; it isn't worth searching neighboring AABB nodes as well before searching the whole tree.

Depth- vs. Breadth-First Search

To compare the use of depth- and breadth-first distance search, voxel arrays and distance fields were generated for all four meshes using each approach. Figure 11 shows the results when using the optimal TREE_ASCEND_N value of 0. Depth-first search is consistently better, but by a very small margin.

When spatial coherence is not exploited – which serves as a surrogate for the case in which the point set is not sorted and does not provide strong spatial coherence – depth-first search performs

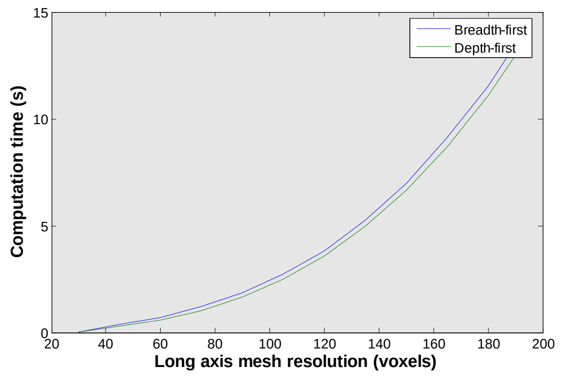

Figure 11. Comparison of depth- and breadth-first search for the "happy" mesh using a TREE_ASCEND_N value of 0 (optimal).

significantly better. This is illustrated in Figure 12, which shows results for the "happy" mesh at various resolutions with no assumption of spatial coherence.

Implementation Availability

A binary version of this application, with documentation and the models used in these experiments, is available online at:

http://cs.stanford.edu/~dmorris/voxelizer

Voxelizer is currently used to generate the voxel meshes used in [23]; distance fields are used to shade voxels based on their distances to anatomic structures.

Future work will include leveraging the obvious parallelism available in this approach; voxels are processed nearly independently and could easily be distributed across machines with a nearly linear speedup. Furthermore, the simple nature of the computations performed here makes this suitable to parallelization across simple processing units, such as those available on commercial GPU's, which have been successfully used to process AABB-based collision queries by [24]. We would also like to explore the performance impact of using other bounding-volume hierarchies (e.g. oriented-bounding-box trees and sphere trees), which fit trivially into our framework.

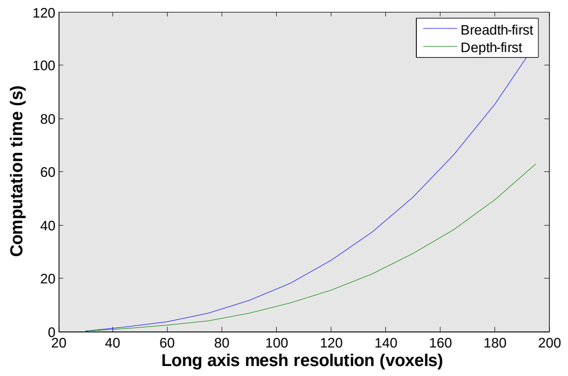

Figure 12. Comparison of depth- and breadth-first search for the "happy" mesh using a TREE_ASCEND_N value of -1 (no exploitation of spatial coherence).

4. HAPTIC DATA LOGGING

4.1 Background

It is conventionally accepted that a user will begin to notice discretization artifacts in a haptic rendering system if the system's update rate falls below 1kHz. Furthermore, as a haptic application's update rate falls, the system becomes more prone to instability and constraint violation. With this in mind, it is essential that designers of haptic software structure applications to allow high-bandwidth, low-latency generation of haptic forces.

There are two relevant implications of this requirement. First of all, haptic computation must run on a thread that allows computation at 1kHz. This is non-trivial on single-CPU systems running non-real-time operating systems, which typically have thread timeslices of 15ms or more. In other words, naively sharing the CPU among a haptic application thread and other application or system threads will not nearly provide the necessary performance. Boosting thread and process priority is a simple solution that is offered by common OS's, but indiscriminately boosting thread priority can prevent other application tasks (e.g. graphic rendering) and even critical operating system services from running. Common solutions to this problem include using dual-CPU PC's, boosting thread priority while manually ensuring that the persistent haptic loop will yield periodically, and/or using hardware-triggered callbacks to control the rate of haptic force computation.

Additionally, this stringent performance constraint means that "slow" tasks (those that require more than one millisecond on a regular basis) cannot be placed in the critical path of a haptic application. Graphic rendering, for example, is often computationally time-consuming and is generally locked to the refresh rate of the display, allowing a peak throughput of approximately 30Hz on most systems (lower if the graphical scene is particularly complex). For this reason, nearly all visuohaptic applications decouple graphic and haptic rendering into separate threads.

Disk I/O is another task that incurs high latencies (often over 10ms), particularly when bandwidth is high. For a haptic application that requires constantly logging haptic data to disk – such as a psychophysical experiment involving a haptic device – it is essential to place blocking disk I/O on a thread that is distinct from the haptic rendering thread.

Using this common scheme, data synchronization between a haptic thread (which collects position data from the haptic device, computes forces, and sends forces to the device) and a "slow" thread (handling graphics and disk I/O) can become a bottleneck. Traditional locks allow the slow thread to block the haptic thread, and if the locked region includes a high-latency operation, the haptic thread can stall for an unacceptable period. Many applications are able reduce the data exchanged among threads to a few vectors or small matrices, and forego synchronization entirely since the probability and impact of data conflicts are rare.

Data logging tasks, however, cannot take this approach. Even small errors resulting from race conditions can place data files in an unrecoverable state. Furthermore, the high bandwidth of data flow increases the probability of conflicts if data queued for file output is stored in a traditional linked list. We thus present a data structure that allows lock-free synchronization between a producer thread and a consumer thread, with the constraint that the consumer thread does not need to access data immediately after the data are produced. The only synchronization primitive required is an atomic pointer-sized write, provided by all current hardware. This structure does not address sleeping; it's assumed that the producer never sleeps (it's a high-priority loop). Periodically waking the consumer – who might sleep – is a trivial extension.

We present this approach in the context of a haptic application, but it's equally applicable to other applications with similar threading structures, for example neurophysiological and psychophysical experiments. For example, the implementation discussed here is used by the software presented in [11], which is used in the experiments presented in [12].

4.2 Data Structure

The data structure presented is labeled a "blocked linked list" (BLL). The BLL is a linked list of blocks of data records; the list's head pointer is manipulated only by the consumer, and the list's tail pointer is manipulated only by the producer. The BLL is initialized so that the head and tail pointers point to a single block. In pseudocode:

```
struct bll_record {
  // the relevant data structure is defined
  // here; in practice the BLL is templated
  // and this structure is not explicitly
  // defined
};

struct bll_block {

  // the data stored in this block
  bll_record data[BLOCK_SIZE];

  // how many data records have actually
  // been inserted?
  int count=0;

  // conventional linked list next pointer
  bll_block* next=0;

};

struct BLL {

  // conventional linked list head/tail ptrs
  bll_block *head,*tail;

  // initialize to a new node
  BLL() { head = tail = new bll_block; }

};
```

The BLL offers the following interface:

```
// This function is called only by the
// producer (haptic) thread to insert a new
// piece of data into the BLL.
void BLL::push_back(bll_record& d) {

  // If we've filled up a block,
  // allocate a new one.  There's no
  // risk of conflict because the
```

```
    // consumer never accesses the tail.
    if (tail->count == BLOCK_SIZE) {

      bll_block* newtail = new bll_block;
      newtail->next = tail;

      // After this, I can never touch
      // the old tail again, since
      // the consumer could be using it
      tail = newtail;

    }

    // Insert the new data record
    tail->data[count] = d;
    count++;

  }

  // This function is called only by the
  // consumer (logging) thread to flush
  // all available data to disk
  void BLL::safe_flush() {

    // If the tail pointer changes during
    // this call, after this statement,
    // that's fine; I'll only log up to
    // the tail at this instant.  I can't
    // access 'tail' directly for the rest
    // of this call.
    bll_block* mytail = tail;

    // If there are no filled blocks, this
    // loop won't run; no harm done.
    while(head != mytail) {

      // Dump this whole block to disk or
      // perform other high-latency operations
      fwrite(head->data,
        sizeof(bll_record),BLOCK_SIZE,myfile);

      // Increment the head ptr and clean up
      // what we're done with
      bll_block oldhead = head;
      head = head->next;
      delete oldhead;

    }

  };
```

The central operating principle is that the `push_back` routine only accesses the current tail; when the tail is filled, a new block becomes the tail and this routine never touches the old tail again. The `safe_flush` routine flushes all blocks *up to but not including* the current tail. If the current tail changes during this routine's execution, it may leave more than one block unflushed, but it will not conflict with the producer's `push_back` routine.

These two routines comprise the important components of the data structure; required but not detailed here are additional initialization routines and a "tail flush" routine that flushes the current tail block and can be run when the producer is permanently finished or has downtime (the pseudocode above never flushes the last, partially-filled block). The BLL also presents an O(N) routine for safe random element access by the consumer thread, allowing access to elements up to but not including the head block.

4.3 Implementation and Results

A template-based, C++ implementation of this data structure is available at:

http://cs.stanford.edu/~dmorris/code/block_linked_list.h

This implementation was used in [5], [11], and [12], and introduced no disk latency on the high-priority haptic/experiment threads.

`BLOCK_SIZE` is a performance variable; in practice it is also templated but it need not be the same for every block. Higher values improve bandwidth on the consumer thread, since larger disk writes are batched together and allocated memory is more localized, but may result in larger peak latencies on the consumer thread (due to larger writes). Higher values of `BLOCK_SIZE` also increase the latency between production and consumption. A BLOCK_SIZE value of 1000 was used in [5], [11], and [12].

ACKNOWLEDGMENTS

Support was provided by NIH LM07295 and the AO Foundation.

The "dragon", "bunny", and "happy" models were obtained from the Stanford 3D Scanning Repository [25]. The "gear" model was obtained from the TetGen examples page [26].

REFERENCES

[1] Abbott, J., Marayong, P., and Okamura, A. Haptic Virtual Fixtures for Robot-Assisted Manipulation. *12th International Symposium of Robotics Research (ISRR)*, October 2005.

[2] Garroway, D. and Hayward, V. A Haptic Interface for Editing Space Trajectories. Poster presented at *ACM SIGGRAPH & EuroGraphics Symposium on Computer Animation*. August 2004.

[3] Williams, R.L., Srivastava, M., Conaster, R., and Howell, J.N. Implementation and Evaluation of a Haptic Playback System. *Haptics-e*, Vol. 3, No. 3, May 3, 2004.

[4] Feygin, D., Keehner, M., and Tendick, F. Haptic Guidance: Experimental Evaluation of a Haptic Training Method for a Perceptual Motor Skill. *Proceedings 10th IEEE Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, March 2002.

[5] Morris, D., Tan, H.Z., Barbagli, F., Chang, T., and Salisbury, K. Haptic Training Enhances Force Skill Learning. *IEEE World Haptics*, Tsukuba, Japan, March 2007.

[6] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (Sep. 1975), 509-517.

[7] Schneider, P. and Eberly, D.H. *Geometric Tools for Computer Graphics*. Morgan-Kauffman, 2003. Relevant source:

http://www.geometrictools.com/Foundation/Distance/Wm3DistVector3Segment3.cpp http://www.geometrictools.com/Foundation/Distance/Wm3DistVector3Triangle3.cpp

[8] Mount, D.M. and Arya, S. ANN: A library for approximate nearest neighbor searching. *CGC 2nd Annual Fall Workshop on Computational Geometry*, 1997. Available at http://www.cs.umd.edu/~mount/ANN .

[9] Massie, T.H., and Salisbury, J.K. The PHANTOM Haptic Interface: A Device for Probing Virtual Objects. *Symp. on Haptic Interfaces for Virtual Environments*. Chicago, IL, Nov. 1994.

[10] Conti, F., Barbagli, F., Morris, D., and Sewell, C. CHAI: An Open-Source Library for the Rapid Development of Haptic Scenes Demo paper presented at *IEEE World Haptics*, Pisa, Italy, March 2005.

[11] Morris, D. TG2: A software package for behavioral neurophysiology and closed-loop spike train decoding. Technical documentation, 2006. Available at http://cs.stanford.edu/~dmorris/projects/tg2_description.pdf

[12] Ojakangas, C.L., Shaikhouni, A., Friehs, G.M., Caplan, A.H., Serruya, M.D., Saleh, M., Morris, D.S., Donoghue, J.P. Decoding movement intent from human premotor cortex neurons for neural prosthetic applications. Journal of Clinical Neurophysiology, December 2006, Volume 23, Issue 6, p577-584.

[13] Fisher, S. and Lin, M. Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields. *IROS* 2001.

[14] Varadhan, G., Krishnan, S., Sriram, T,, and Manocha, D. Topology Preserving Surface Extraction Using Adaptive Subdivision. *Eurographics Symposium on Geometry Processing*, 2004.

[15] McNeely, W.A., Puterbaugh, K.D., and Troy, J.J. Voxel-Based 6-DOF Haptic Rendering Improvements. *Haptics-e*, vol. 3, 2006.

[16] Bartz, D. and Guvit, O. Haptic Navigation in Volumetric Datasets. *Second PHANToM Users Research Symposium*, Zurich, Switzerland, 2000.

[17] Kim, L., Sukhatme, G., and Desbrun, M. A haptic rendering technique based on hybrid surface representation. IEEE Computer Graphics and applications, March 2004.

[18] Sethian, J.A. A fast marching level set method for monotonically advancing fronts. In Proc. Nat. Acad. Sci., volume 93 of 4, pages 1591-1595, 1996.

[19] Mauch, S. Efficient Algorithms for Solving Static Hamilton-Jacobi Equations. PhD thesis, 2003.

[20] Closest Point Transform (open-source software): http://www.acm.caltech.edu/~seanm/projects/cpt/cpt.html

[21] Sud, A., Otaduy, M., and Manocha, D. DiFi: Fast 3D Distance Field Computation Using Graphics Hardware. *Eurogrpahics 2004*.

[22] Cohen, J.D., Lin, M.C., Manocha, D., and Ponamgi M. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scaled Environments. *Proc. ACM Symposium on Interactive 3D Graphics*, pp. 189-196, 1995

[23] Morris, D., Girod, S., Barbagli, F., and Salisbury, K. An Interactive Simulation Environment for Craniofacial Surgical Procedures. Proceedings of MMVR (Medicine Meets Virtual Reality) XIII, Long Beach, CA, January 2005. Studies in Health Technology and Informatics, Volume 111.

[24] Thrane, N. and Simonsen, L.O. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, Denmark, 2005.

[25] http://graphics.stanford.edu/data/3Dscanrep/

[26] http://tetgen.berlios.de/fformats.examples.html