Analysis of collision detection using Implicit Sphere Tree in haptic interaction environments for maxillofacial surgery applications

Laurens Le Jeune[1], Wout Swinkels[1], Yi Sun[2], Constantinus Politis[2], and Luc Claesen[1]

[1]Faculty of Engineering Technology, Hasselt University - UHasselt, Diepenbeek, Belgium
[2]OMFS IMPATH Research Group, Department of Imaging and Pathology, Faculty of Medicine, KU Leuven, and Oral and Maxillofacial Surgery, University Hospitals Leuven, Leuven, Belgium

***Abstract*—Facial malformations such as overbite or underbite can be treated by maxillofacial surgery where the mandibula (i.e. the lower jaw) and/or maxilla (i.e. the upper jaw) is repositioned to obtain a preferential occlusion. These treatments need to be carefully planned prior to the actual operation. The planning of such surgery is an intensive and time consuming procedure in which the experience and knowledge of the surgeon plays a major role. As part of a CAD-based surgery planning, haptic interaction using 3D virtual models of the mandibula and maxilla, haptic feedback can make planning and surgery much more efficient. Haptic devices bridge the gap between the virtual environment and the physical world. An important step in such haptics based workflow is the collision detection between the two models at an update frequency of at least 1000 Hz. In this paper the possibility of the Implicit Sphere Tree algorithm as a collision detection algorithm tailored for this specific application is presented.**

***Index Terms*—Haptic interaction, Collision detection, octree, Implicit Sphere Tree, orthognathic surgery**

I. Introduction

Haptic technology is the technology that allows a user to interact with a virtual object. Where Virtual Reality (VR) usually is associated with extremely realistic graphics, it can be augmented with haptic technology where the haptic device, controlled by the user, interacts with the virtual object. This device provides a force feedback (haptic feedback) to the user based on the users actions in the virtual world. A general consensus is that in order to achieve a realistic touch of a solid virtual surface, an update frequency of at least 1 kHz is mandatory [1]. This poses a lot of challenges, such as the development of collision detection algorithms that can handle the desired update frequency of more than 1 kHz.
Collision detection has already been researched for a few decades. The work of Hubbard [2] for example dates from 1995. The main goal is to efficiently calculate the points of intersection (if any) between two virtual objects, for example two polygon meshes (e.g. two 3D structures built from triangles). Typically, this ought to be done at a high update frequency (at 1 kHz), but without compromising on the resolution, which means that a lot of primitives, triangles in this case, are required. If the number of primitives becomes larger, however, the number of calculations increases too.
A straightforward way to tackle this problem is to take each primitive in one object and compare it to each primitive in the other object. This is a very inefficient way to find collisions, as the problem is proven to have a time complexity of $O(n^2)$ [3]. For the application in this paper, each object consists of more than 100,000 primitives. With an update frequency of 1 kHz this requires at least 10^{10} comparisons of triangles. This is too much to be practically useful. An efficient data structure is necessary to represent the objects, and an efficient algorithm is needed to quickly and accurately detect collisions between the meshes.

The methods presented in this paper are intended to be used in pre-operative orthognathic surgery decision making to enable haptic interaction between virtual jaw models. In order to be able to "feel" all contact points during virtual occlusion finding, a high level of detail is required. This results in large data structures. The work presented in this paper builds on the data structure and supplementary algorithms presented by E. Ruffaldi et al. [4], *the Implicit Sphere Tree*, and apply it to the problem of occlusion finding in virtual models of the mandibula and maxilla. The data structure used is a hybrid mixture of regular octrees [5] and spherical octrees [6].
In this paper, the implementation and performance of the implicit sphere tree are presented for this specific application. Section II describes the implementation, design choices and virtual environment for the software development. In section III the performance is presented, and the obtained results are discussed. Section IV provides conclusions.

II. Implicit Sphere Tree

In order to generate Implicit Sphere Trees (IST), several calculation steps have to be taken. First, the given 3D data structure must be voxelized. Voxelization means that a cubic bounding box, enclosing the object to be voxelized, is divided into smaller cubes until a pre-defined cubic size, the voxel size, is reached. Next, the given voxels are efficiently

represented in an octree (Fig. 1) [5]. An octree is an extension of a binary tree concept enabling more efficient data structures to organize and efficiently search objects in a 3D space. The octree forms the basic data structure that the IST is built upon. During collision detection, the algorithm will traverse the octree while simultaneously constructing a sphere-based representation of the octree. These spheres can be used for fast approximation of contact points upon collision because it is easier to calculate the intersection between two spheres, this geometric primitive is rotation invariant, compared to bounding boxes, which are rotation variant.

The implementation of the application has been realized in C++, using the open-source framework CHAI3D [7] in Visual Studio. This framework, in conjunction with previous results [8] provides a sturdy backbone upon which the IST can be implemented. Finally, in this paper it is assumed that the initial 3D data structure is presented in a STL file format.

A. Voxelization

The voxelization algorithm is inspired by an application of the Image Synthesis Group at the Trinity College Dublin [9]. To voxelize a 3D triangular mesh, a data structure called *Spatial Hashing* is used. This hash is built by first determining the longest edge (l) of the smallest possible box containing the entire object. The length of this edge serves as the length of the 3D cube that composes the hash. Once the required voxel size s (the length of the voxels edges) is given, the number of voxels n in each Cartesian direction can be found:

$$n = \left\lceil \frac{l}{s} \right\rceil \tag{1}$$

The result of (1) is incremented until n equals a power of 2, which makes further calculations easier (II-B). When n is found, a 3D array -the Spatial Hashing- with n elements in each direction is initialized. These elements, small cubes with edge length s, are called voxels. Each triangle in the STL-file is then placed in this hash according to a specific procedure: First, all voxels with the potential to intersect the triangle are selected. This is done by calculating the bounding box of the triangle and search for all voxels contained by this volume. Then, each of these voxels is checked for intersection with the triangle. This is done with the triangle-cube-intersection algorithm [10]. Now, each voxel can be classified using the method of [11], which distinguishes 4 types of voxels: *Free space*, *Interior*, *Surface* and *Proximity*. The voxels that intersect with a triangle are Surface voxels. Proximity voxels, the voxels within a range of approximately *2s* from the outer surface of the object, can easily be found using the surface voxel as well as the triangle it intersects with. Since the triangle originates from an STL-file, its normal -given in the STL-file- can be used to determine which voxels neighboring the surface voxel are proximity voxels. Currently, there is no need to distinguish between *Interior* and Exterior (*Free space*) voxels, as these are not necessary for collision detection and are therefore classified as *Undefined*.

B. Octree

In order to rapidly search through all available data, the voxelized structure is turned into an octree. In principle, an octree can be viewed as a cube divided in half along each Cartesian direction. This provides 2^3=8 equal cubes that each enclose 1/8th of the original cube. The original cube is called the parent and the smaller cubes are the children. Both, the parent and the children, are considered tree nodes. To further build the tree each of these children can once again be divided into 8 new cubes until the desired tree depth and/or accuracy is reached. The nodes on this level are called the leaf nodes. To transform the voxelization into an octree, it is necessary that the number of voxels on each axis is a power of 2. The voxels of the Spatial Hashing always correspond to the smallest cubes of the octree, the leaf nodes. Equation (2) can then be used to calculate the amount of levels in the octree to completely fit the Spatial Hashing. As recommended in [4], the added parameter L (level) will play a large role in determining the characteristics of each level in the octree. For L going from 0 to *maxL* in the tree, let L=*maxL* be the highest level consisting solely of the root node. Let L=0 be the lowest level of the tree, consisting of the leaf nodes which correspond to a voxel.

$$log_2(n) = \text{maxL} \tag{2}$$

It is fairly intuitive to build the octree bottom-up, starting from L=0. By grouping 8 adjacent voxels in a 2×2×2 fashion, these 8 voxels can be transformed into nodes, the children of one node at L=1. When all voxels are processed this way, all the nodes of L=1 can be grouped 2×2×2 to become children of nodes at L=2. This way, the octree can be built one layer at a time. There are a few weaknesses in this strategy, though. A first weakness is that the possibility exists that a far larger than necessary number of voxels in the Spatial Hashing is created by requiring each axis to be a power of 2. As it happens to be, all voxels added by increasing n to a power of 2 are *Undefined*, and per definition of no use. A second weakness is that for the octree, the only nodes of interest are the nodes containing a relevant voxel (*Surface* or *Proximity*). All other nodes saved in the octree contribute to unnecessary calculations and memory consumption. But these issues can be, to a certain degree, resolved with an easy optimization.

C. Optimized octree

A simple and intuitive way to optimize the octree, is to exclude any voxel that is not relevant. Starting bottom up, each group of voxels is monitored. A group that does not contain any relevant voxels will always result in a parent node one level higher that does not lead to any relevant voxels. This node is flagged as being empty, and the group of voxels are not represented explicitly in the data structure. One level higher, any group consisting of 8 empty nodes should not be saved, and the node on that level enclosing these 8 empty nodes should then be flagged as empty. This process is repeated until the top of the tree is reached. If executed correctly, for $L \geq 1$, all nodes in the tree contain at least some relevant information. On L=0, it is possible that some *Undefined* nodes grouped with

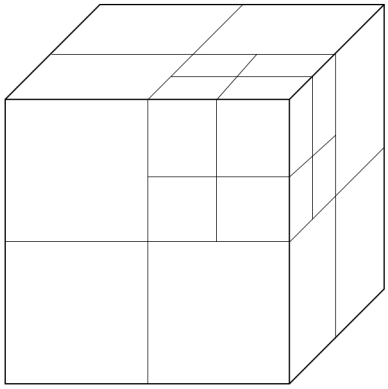

Fig. 1. Basic octree construction. Each node is divided in 8 child nodes.

relevant nodes are saved. While not completely eliminating every unnecessary node in the tree, the number of unwanted nodes in the octree will be greatly reduced which solves the second issue described above. Indirectly, this also partially invalidates the first issue, as almost none of the unnecessarily added *Undefined* voxels will remain saved in the octree. This also allows further optimization by adapting the algorithm in such a way that it is no longer necessary to require n to be a power of 2. When given n' conforming to (1) but not a power of 2, let n'' be n - n'. As it is known all voxels beyond n' are *Undefined*, these n'' voxels can be extrapolated onward from n'. In this way, the Spatial Hashing only needs to save n' voxels instead of n voxels, again reducing the memory consumption for larger data structures.

D. Implicit Spheres

During collision detection, the octree is traversed while simultaneously constructing a spherical representation of the octree. This spherical representation is the IST, which consists of implicit spheres. These spheres are called implicit because the spheres are not explicitly calculated and stored during precalculation but are derived from the current state of the octree whenever needed. In order to do this, it is necessary to first define an efficient way to derive a sphere from an octree(node). Based on [4], two methods are implemented and explored. The first method, the simple method, is based on (3) in [4] and the logic of the octree. Equation (3) gives the minimal radius r to enclose a cube with an edge length x. The center of the sphere matches the center of the cube.

$$r = \frac{\sqrt{3}}{2}x \tag{3}$$

Since the edge lengths of the child nodes of a node are $x/2$, the radii of the spheres enclosing those child nodes are half the radius of r. The offset of the centers of those spheres from their root is $s{\times}2^{L-2}$ according to [4]. However, in this application, this formula is found to be incorrect and must be adapted to $s{\times}2^{L-1}$. Each of the child nodes then must be offset this distance on each of the Cartesian axes.
The second method looks to optimize the collision detection by reducing the necessary amount of collision checks. In order to do this, the radius of an implicit sphere has to be minimized according to the child nodes occupation. If, for example a node only has one relevant child node, the enclosing sphere only needs to enclose this individual child, not the entire eight child nodes. The various situations are discussed in [4] and can be implemented by using a lookup table where the correct radius and offset for each situation can quickly be found. Fig. (2) illustrates this by showing the L=2 implicit sphere occupation for a 32×32×32 octree. Since the total volume of the IST decreases by implementing the lookup table, fewer intersections will occur, meaning the amount of collision checks reduces.

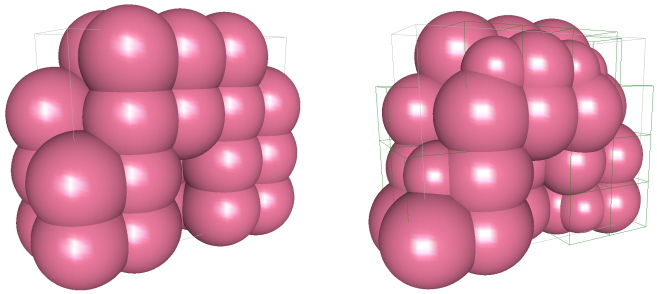

Fig. 2. Level 2 implicit sphere occupation for a 32×32×32 octree following the simple method (left) and the optimized method (right). Notice that on the right side the enclosed volume is remarkably smaller than the volume on the left side. The grey lines accentuate the bounding box of the original figure, the green lines give the skeleton of level 2 of the octree based on data from [12].

E. Collision detection

Now that a method has been explained to calculate the IST from a given octree, it is possible to implement collision detection according to [4]. Computing whether two spheres collide requires only one condition (4) to be fulfilled. Let r_1 and r_2 be the radii of the spheres 1 and 2, and C_1 and C_2 be the centers of those spheres. If this condition is fulfilled, it is certain that the two spheres intersect.

$$r_1 + r_2 \geq |C_1C_2| \tag{4}$$

Since this is easier and faster than calculating collisions between cubes, collisions are calculated using the IST's derived from the octrees.
In order to calculate whether or not two objects collide, the implicit spheres enclosing the root node of those objects octrees are first calculated. If the two spheres collide, the next level is checked. By comparing each root node child of the first octree with each root node child of the second octree using implicit spheres, all colliding branches are found. For each of these child nodes, their colliding children can again be found by comparing its children to those of the node it collides with.

In this way, using a depth-first search algorithm, pairs of colliding leaf nodes are found. Since each leaf node correlates with one voxel, colliding voxels can be found. Using this algorithm, one distinct advantage directly stands out: Each

implicit sphere is only calculated if its parent sphere collides with another sphere, which means it is only calculated if it has the potential of colliding. This means the number of calculated spheres during runtime is minimal, which minimizes the overall calculation time. On the contrary, if the entire sphere tree had to be calculated each time a collision check was executed, the overhead would significantly increase.

III. Results

The procedure described in section II is used to derive the IST for both the upper and lower jaw model with respectively 119,110 and 106,696 triangles. A representation of the used lower jaw model is depicted in Fig. 3. The results in the following sections are the results obtained for the lower jaw model, however the derived conclusions are also applicable for the upper jaw model.

Fig. 3. STL-version of the lower jaw used for testing the data structures and collision detection based on data from [12].

A. Spatial Hashing

Figure 4 gives the surface voxel count in function of the voxel size and figure 5 gives the calculation time to voxelize the jaw in function of the voxel size. When examining the construction of the spatial hashing, it can be concluded that there is an exponential growth in the calculation time and the number of used voxels. After examining this growth it becomes clear that the relationship between either voxel amount and calculation time is cubic. This means that the time complexity is $O(n^3)$ and that the growth is a function of n^3, where n is the number of voxels in one dimension. Equation (5) for calculating the number of voxels m, given n from (1), also highlights this relationship.

$$m = n^3 = \left(\frac{l}{s}\right)^3 \tag{5}$$

Following this observation, a new conclusion arises: Voxelizing an object will always be a trade-off between efficiency and accuracy. Efficiency is inversely proportional to m for a given object: every voxel needs to be calculated and stored. Therefore, if m increases, the efficiency will decrease. Accuracy however is inversely proportional to s. If s decreases, the accuracy will increase (smaller cubes can more easily be used to mimic complex shapes) but m will also increase, decreasing efficiency. It is impossible to maximize both simultaneously. Figure 6 illustrates the voxelized lower jaw for different voxel sizes.

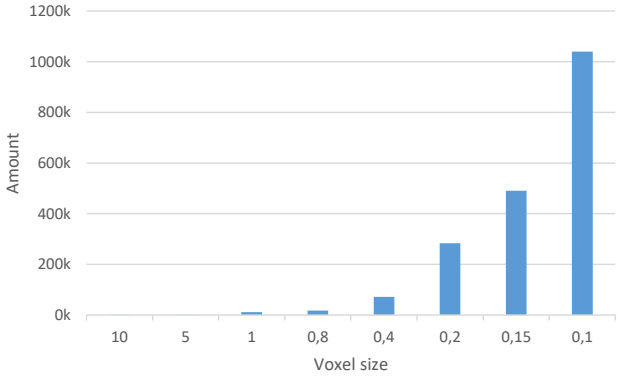

Fig. 4. Number of surface voxels in function of the chosen voxel size. Note: For the voxel sizes 10, 5 and 1 the number of surface voxels (92, 397 and 11120) is too small to show on a linear scale.

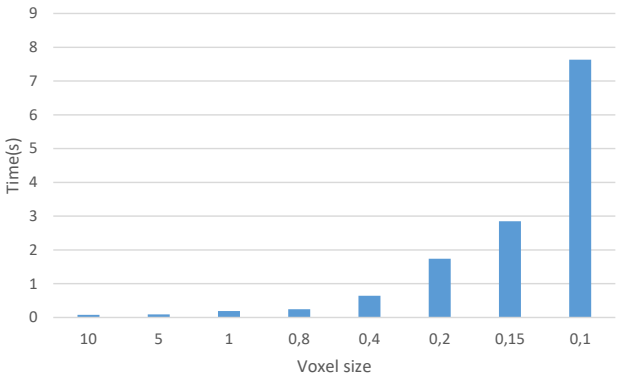

Fig. 5. Calculation time in function of the chosen voxel size. Note: For the voxel sizes 10, 5 and 1 the number of surface voxels (92, 397 and 11120) is too small to show on a linear scale.

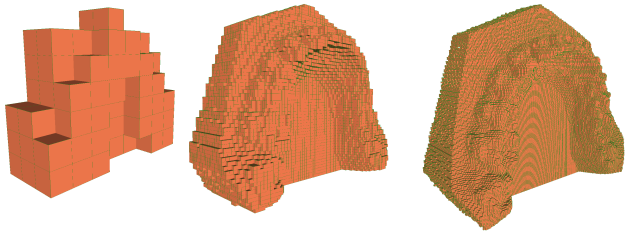

Fig. 6. Jaw voxelization, with a varying accuracy based on the voxel size. Left: Voxel size = 10. Middle: Voxel size = 1. Right: Voxel size = 0.4 based on data from [12].

B. Octree

When the object is voxelized the next step, structuring the data, is taken by converting the spatial hashing into an octree.

Figure 7 and figure 8 provide some insight concerning the calculation time and octree size in function of the voxel size. Fundamentally, the same cubic trend as discussed in III-A also appears during the octree construction. When looking at the calculation time for the octree, it becomes clear that not only the voxel size but also the spatial hashing size defines the necessary calculation time. While a slightly smaller s (adding a few more voxels) may result in a slight relative increase in time (relative to times with the same spatial hashing size), the real jumps in time duration occur when the spatial hashing increases in size. This can be explained by the fact that while not all voxels are saved as leaf nodes in the tree, all voxels are visited at least once to check whether it is a relevant voxel or not. So, while perhaps only 10% of the voxels are relevant, all voxels are visited once, which has a significant impact on the calculation time. The minimum total amount of nodes relative to the amount of nodes when all voxels are used for the octree construction can easily be estimated without measurements since a level L will at least contain as much nodes as (L - 1) divided by 8. For voxel size 1 this would result in 12700 nodes instead of the actual 14600, indicating that many nodes do not contain 8 relevant children.

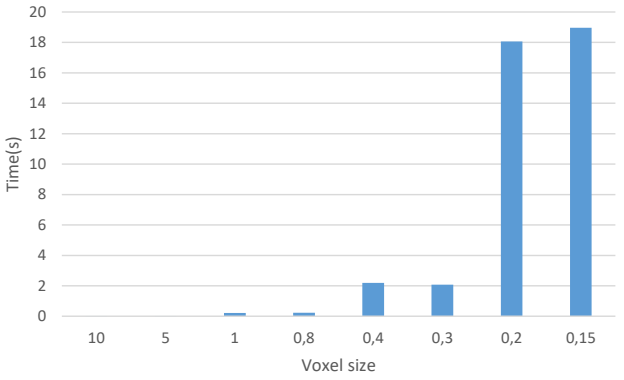

Fig. 7. Octree calculation time in function of the voxel size.

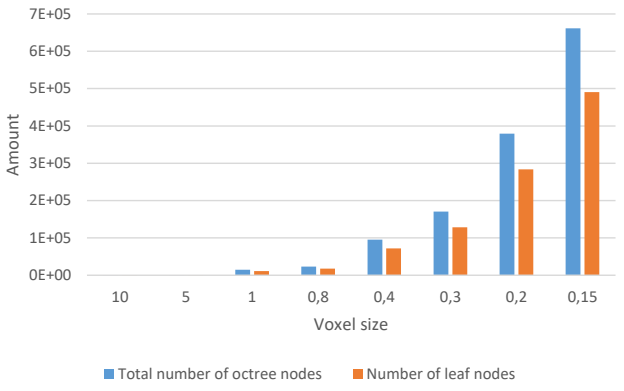

Fig. 8. The number of nodes in the entire octree and the number of leaf nodes for a certain voxel size.

C. Implicit Spheres

When the octree is constructed, it can be converted into an IST. In the results (Fig. 9) concerning the implicit spheres, two observations stand out. First, the optimized implicit spheres result in a significant decrease in contained volume compared to the simple implicit spheres. The volumes are calculated with respect to a unit volume defined as $V=\frac{\sqrt{3}}{2} \times \pi \times 2^{3(L-1)} \times s^3$. This is the volume of a sphere enclosing a voxel. Note however that while the volume difference seems very large on L=7, in that case it only concerns one sphere (i.e. the root sphere). Optimized spheres detect fewer false positive collisions than simple spheres therefore there is a greater chance that the simple sphere still needs to execute 8 extra collision checks while none of its children collide.
Another important observation is that the calculation time of the simple sphere is significantly less compared to that of the optimized sphere. This indicates that collision checks with optimized spheres are far more time consuming than those of simple spheres. Because of this, optimized spheres must result in a smaller number of false collision checks to compensate for the increased time delay. Figure 9 visualizes the volume difference between the simple sphere implementation and the optimized spheres on the other hand.

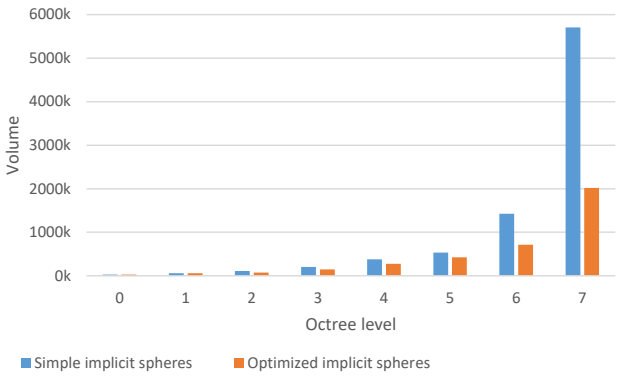

Fig. 9. Total volume of all implicit spheres on each level.

D. Collision detection

Finally, after constructing the IST, collision detection can be performed. Therefore an IST is created for both models, the lower and upper jaw. The dept-first search algorithm is used as the tree traversal algorithm. Both models are loaded into the CHAI3D platform. The model of the lower jaw is fixed while the upper jaw model descent upon the lower jaw model. In this case, the lower jaw size is 73.0×27.5×60.9 and a step is the displacement with a value of 1 in any Cartesian direction. Since an STL file contains no scale information, these values need to be compared to the real size of the lower jaw. Figure 10 provides a graph with the refresh rates in function of the voxel size. When the haptic refresh rates measurements are examined, it becomes clear that the optimized spheres in fact do not reach the desired

update frequency. Simple spheres are more efficient when the distance has decreased to a certain point (between 20 and 25). Because the simple spheres are larger, an intersection occurs faster. Even though in most cases this only means a small number of additional collisions checks are needed, this still creates a higher update frequency compared to the optimized implementation. When the distance between the objects decreases, the number of collision checks becomes so large that the sub-optimal delay in optimized sphere calculations negates the advantage of having to do less checks.
Furthermore, it is also clear that the current haptic feedback rates do not meet the desired 1000Hz when the objects are too close to each other. In the worst case, the frequency is as low as 50 Hz for optimized spheres, and 100Hz for simple spheres. Partially, however, this is because the code is not fully optimized. The worst case always occurs right before collision, because then the largest number of spheres need to be checked for a potential collision without actually having a collision. Certainly, for the optimized sphere tree there is a lot of room for improvement, but it does not end there. A couple of effective optimizations can be:

- Save the IST. Currently, the IST is constructed during collision detection. This has one clear advantage: It does not need to be saved. This increases memory efficiency, but greatly reduces the calculation speed of collision detection. If the IST is saved after construction in a preprocessing step, the only thing left to do would be to traverse the sphere trees and the on the fly construction could be neglected. Since the octree, as argued in [4], is only needed due to the complexity of the independent sphere tree construction, it is also possible to just remove the octree from the memory once the IST has been saved. This option is not explored in [4] but has a lot of potential.
- A more efficiently implemented lookup table. This would result in faster collision checks for optimized spheres, however it will also result in a memory usage increase. The lookup table also is very suitable for possible hardware implementations.

A final remark concerning the optimization of the collision detection is that optimized implicit spheres have far more room for improvement than simple implicit spheres. This suggest that while simple spheres are more efficient now, optimized spheres have the potential to still swiftly surpass that efficiency.

IV. Conclusion

The potential of the Implicit Sphere Tree is examined as part of a CAD-based surgery planning. Although the desired update frequency of 1000 Hz is not met, as great accuracy is required for the medical application, the gained insights show that there is still room for improvement. The implemented data structures however are a decent foundation for further research. The potential of the IST suggests that further optimization and study might cause it to reach the 1000 Hz evaluation requirement. The conducted research also opens the path to a new idea, namely to save the IST once the octree is fully constructed. This too provides more possibilities to make the IST viable for haptic feedback in highly accurate environments.

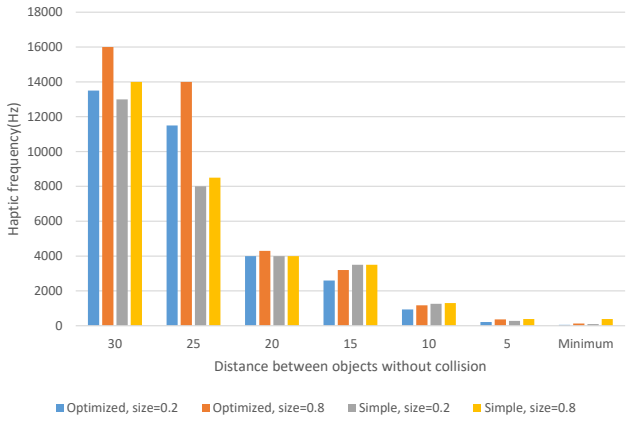

Fig. 10. Haptic refresh rate (= haptic frequency) for collision detection between objects with voxel size 0.8 or 0.2. Collision detection was executed with either simple or optimized implicit spheres.

References

[1] W. R. Mark, S. C. Randolph, M. Finch, J. M. Van Verth, and R. M. Taylor, II, "Adding force feedback to graphics systems: Issues and solutions," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 447–452. [Online]. Available: http://doi.acm.org/10.1145/237170.237284

[2] P. M. Hubbard, "Collision detection for interactive graphics applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 3, pp. 218–230, Sep. 1995. [Online]. Available: https://doi.org/10.1109/2945.466717

[3] R. Weller, *New Geometric Data Structures for Collision Detection and Haptics*. Springer; EuroHaptics Society, 2013.

[4] E. Ruffaldi, D. Morris, F. Barbagli, K. Salisbary, and M. Bergamasco, "Voxel-based haptic rendering using implicit sphere trees," *Symposium on Haptics Interfaces for Virtual Environment and Teleoperator Systems 2008 - Proceedings, Haptics*, pp. 319–325, 2008.

[5] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 19, no. 2, pp. 129 – 147, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0146664X82901046

[6] C. Tzafestas and P. Coiffet, "Real-time collision detection using spherical octrees: Virtual reality application," in *IEEE Int. Work. on Robot and Human Communication*, 1996, pp. 11–14.

[7] F. Conti, F. Barbagli, R. Balaniuk, M. Halg, C. Lu, D. Morris, L. Sentis, J. Warren, O. Khatib, and K. Salisbury, "The chai libraries," in *Proceedings of Eurohaptics 2003*, Dublin, Ireland, 2003, pp. 496–500.

[8] N. Pirotte, C. Vranken, W. Swinkels, L. Claesen, Y. Sun, and C. Politis, "Haptic collision detection on highly complex medical data structures," *Proc. 10th Int. Congress on Image and Signal Processing, BioMedical Engineering and Informatics IEEE CISP-BMEI 2017*, pp. 3B–7–3B–12, 2017.

[9] I. S. G. T. C. Dublin, "Sphere-tree construction toolkit," Online.

[10] D. Kirk, *Graphic Gems 3*. Academic Press Inc., 1994.

[11] W. A. McNeely, K. D. Puterbaugh, and J. J. Troy, "Six degree-of-freedom haptic rendering using voxel sampling," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 401–408. [Online]. Available: http://dx.doi.org/10.1145/311535.311600

[12] C. Politis and Y. Sun, afdeling Mond-, kaak- en aangezichtschirurgie, UZ-Leuven.