uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Naim El-Far

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Ph.D. (Computer Science)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Haptic Rendering of Highly Detailed Point-based Virtual Models

TITRE DE LA THÈSE / TITLE OF THESIS

Nicolas Georganas

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Keyvan Hashtrudi-Zaad

Dorina Petriu

Jochen Lang

Emil Petriu

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Haptic Rendering of Highly Detailed Point-Based Virtual Models

Naim R. El-Far

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the PhD degree in Computer Science

Ottawa-Carleton Institute of Computer Science
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-50726-1
Our file Notre référence
ISBN: 978-0-494-50726-1

To my mother.
To the memory of father.

Acknowledgment

In omnibus gratias agite.

I offer my eternal gratitude to Dr. Nicolas Georganas, my supervisor, for taking me under his wing and supporting my education at the University of Ottawa both morally and financially. It has truly been an honor and a privilege to work under the supervision of who I am not alone in considering a major contributor to human knowledge.

An equal debt of gratitude is owed to my head coach, Coach Denis Piché, for being an older brother, a confidant, and an advisor, in addition to being a great football coach and a greater person whom I am proud to call my friend.

I also thank both Dr. Georganas and Coach Piché for their understanding that I led a dual life at the University of Ottawa; one as a graduate student, and another as a football player. Without Dr. Georganas' and Coach Piché's cooperation, this wonderfully rewarding dual life could not have been possible.

This section would be incomplete without thanking Dr. Abdulmotaleb El Saddik for his help throughout my career at the DISCOVER lab. Dr. Abed has been a friend, an advisor, and an invaluable source of knowledge whose help was essential in bringing my pursuit of a PhD to fruition.

My colleagues at the DISCOVER lab are also owed a debt of gratitude, especially Jilin Zhou and Francois Malric. My thanks to Jilin for keeping me company in the lab till the early hours of the morning, testing my algorithms for me, and serving as a sounding board for many of my ideas. My thanks, also, to Francois for being the devil's advocate and ever the skeptic who has unknowingly ensured that I be ready for any criticism imaginable of my

work. Thanks are also due to Francois, in his capacity as our lab manager, for more than five years of professionalism in dealing with my never-ending requests.

Last, but not least, my sincerest thanks to those who have contributed to this work indirectly and from behind the scenes; my family and my girlfriend. An eternal and infinite debt of gratitude is owed to my mother, whom I dedicate this work to, for her support, encouragement, and never-ending prayers. I also thank my brother and sister, Fayez and Dalia, for their blind faith and their unconditional support which have often given me strength and fortitude. And I thank my girlfriend Nadine who has been a source of joy in my life since the day I had first met her, as well as a tireless and ever-understanding partner who has run my errands, cooked and brought me food, and did a lot of my chores for me while I was in my "bunker" working on this thesis.

Abstract

In this work, we present a comprehensive approach to haptically render highly-detailed point clouds without first creating their corresponding polygonal mesh. We say that our approach is comprehensive because it addresses: kinesthetic and tactile rendering; static and dynamic models; collision detection and force response; force shading; deformation and stiffness; and friction. These features compromise the majority of the haptic interactions possible while using a 3-degrees-of-freedom haptic device, which is the target device for our algorithms. Furthermore, we look in this work at height fields and redefine them as a special case of point clouds for which we present a specialized haptic rendering approach that includes all the features already mentioned for our general purpose approach.

Our work relies on redefinitions of what a point cloud's surface is; for the purposes of collision detection, we look at it as a collection of touching, if not slightly overlapping, axes-aligned bounding cubes; and for the purposes of force response and haptic effects rendering, we look at the surface as a neighborhood of points where each point knows its immediate neighbors. Our collision detection algorithms are novel, and our force response algorithms are loose adaptations to point clouds of standard constraint-based approaches.

Our work is motivated by, and largely designed for, models that are the result of scanning real-life objects in 3D. These models are of great practical use in a large number of fields ranging from arts to manufacturing, and from entertainment to medicine. The scanning technology (laser or contact) is of no consequence to our work, but what is relevant is the high-density point cloud that typically results from 3D scans. When possible, we also make use of our a priori knowledge of the path the scanner takes to sample the real-life object in a novel approach to compile neighborhood information.

Finally, this work will demonstrate, through experimental results, its effectiveness in conveying haptic information, its speed even when all haptic algorithms run in a single thread on a single processor, and its insensitivity to the size of the input point clouds; factors that make the case for our approach's adaptation in place of mesh reconstruction techniques.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1. Haptic Rendering, Virtualization, and Point-Based Models

Haptic rendering is a well researched field with many approaches, algorithms, and workflows available today for use in applications ranging from prototyping and design, to training and e-learning. Most of today's approaches, however, presuppose the use of graphic models that are polygonal. This is to be expected not only because polygons are the most prevalent graphics representation technique today, but also because most collision detection (CD) algorithms used in haptics are based on their graphics counterparts (which assume polygonal representation). Add to the mix the fact that most force response (FR) algorithms – FR being another essential component of haptic rendering – also presuppose polygonal representation, and you quickly realize that polygons are intimately intertwined with everything haptic and graphic rendering today.

Consider the desk on which you may be sitting or the wall at which you might stare if distracted by a thought from this work. Such flat objects could easily be represented in graphics with polygons. The same is not true, however, for more complex objects such as the twirling vase in the corner of the room or the statuette on your bookshelf. Undoubtedly, those objects could be represented with polygons but, if that were to be done manually using some design software, the process could quickly get tedious.

Enter virtualization of real-life objects. To virtualize (or digitize) an object is to import it from the physical realm to the virtual one typically using 3D scanners that capture the surface information of real-life models in order to describe them in the virtual world. 3D scanning removes all "manual" labor from the virtualization process save moving the scanner around the object and, later, maybe some user-assisted post-scan processing. The

catch comes in the model format that the laser scanners produce; a large number (sometimes in the millions) of unstructured and independent points in 3D space corresponding to surface points scanned. The collection of these points is called a point cloud, a point set, or a point-based model, and the denser this point cloud is (i.e. the more points it contains per unit volume), the more details it captures from its real-life counterpart. Figure 1 shows an example of the whole process; a desktop laser scanner samples a real-life model of an alien head for import into the virtual world which happens in the form of the point cloud shown on the computer screen in the background of the figure.

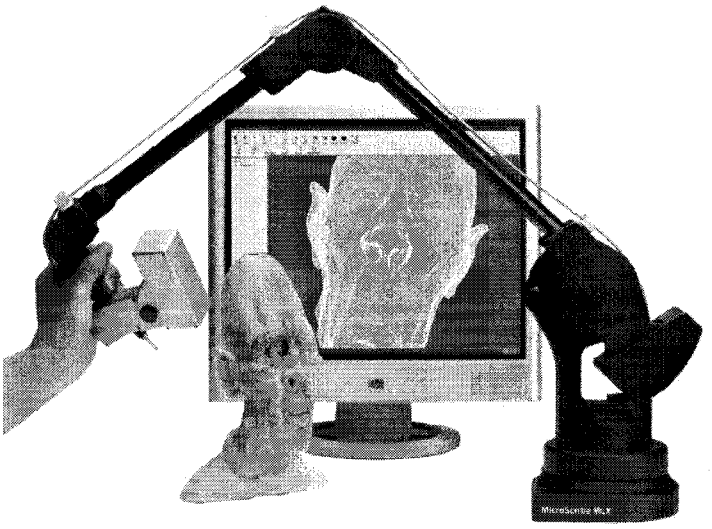

Figure 1: The Microscribe MLX scanning a real-life object to produce a point cloud model (image from Immersion Corp's website).

Once the point cloud has been imported into the virtual world, it must be rendered so that it can be displayed haptically and/or graphically. How to haptically render a point cloud in its native format is the focus of this work.

Today, common practice is to build a polygonal mesh from the point cloud and then haptically render that mesh using available techniques. While this approach produces good results, it requires lengthy preprocessing and its accuracy in recreating the real-life object from its point cloud representation largely depends on the algorithms used to reconstruct the

surface of the input point cloud. [1], [2], and [3] survey the many algorithms in use today for reconstructing polygons from point clouds pointing out the common problems with each; problems like aliasing, under-sampling, and redundancy. Figure 2 shows a model in point cloud form and in polygonal form.

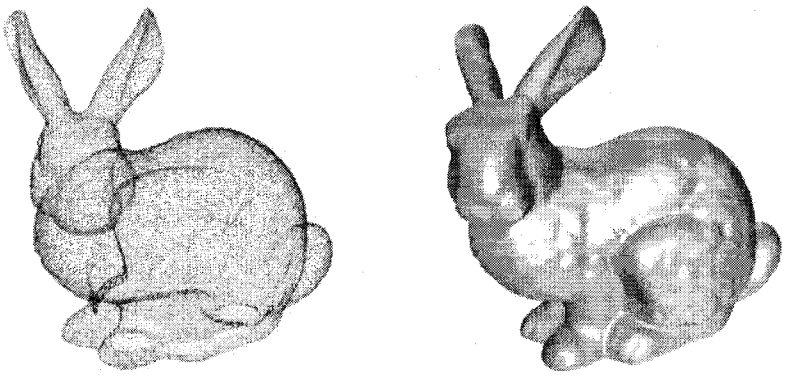

Figure 2: The Stanford Bunny model in point cloud form (left) and in polygonal form (right).

Furthermore, and to the best of our knowledge at the time of this writing, there is no comprehensive work in the literature today that provides an alternative to this approach. More specifically, there is very little work in haptically rendering point clouds without translating them into a different representation first.

The problems with reconstructing surfaces from point clouds, as well as the lack of point cloud rendering algorithms in the literature, are two motivations for our work.

1.2. Applications that Use Point-Based Models

The use of point clouds has been gaining momentum in recent years thanks in large part to highly accurate 3D scanning: a widespread technology used in fields such as manufacturing, engineering, education, and multimedia.

In the automotive and aerospace industries, for example, 3D scanning is used by manufacturers in a variety of ways: to import clay models into CAD (computer-aided

design) software effortlessly and rapidly; to import scaled models into finite element analysis software to perform aerodynamics studies; and to import various parts and models into CAD software for the purposes of analysis and reverse engineering.

Education is also an area in which 3D scanning technologies are used extensively. Art schools are using laser scanners to digitize their students' hand-made work for later manipulation in specialty software. Engineering schools are teaching their students the principles of design and analysis by importing real-life objects into simulated environments. And medical schools are presenting their students with high-resolution 3D models that can be accessed online to examine and learn from.

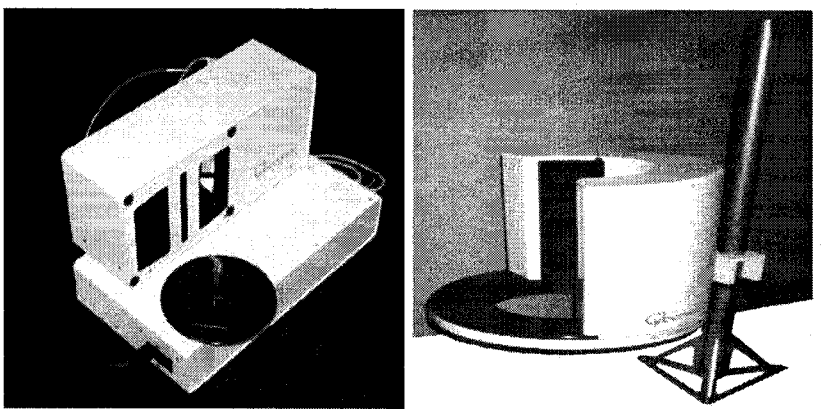

Figure 3: Left: The Cyberware 7G Ear Impression Scanner. Right: The Cyberware Below-the-Knee 3D scanner. (Images from Cyberware's website).

Furthermore, in the medical field, 3D scanning technology is used nowadays to design and create custom orthotics, orthodontics, and prosthetics more accurately than ever before. Figure 3 shows two Cyberware scanners one for capturing ear impressions for the design of prosthetic ears and hearing aids, and another for digitizing residual limbs in amputees for the purpose of designing highly customized sockets for prosthetic devices. Both scanners

produce high resolution models that ensure accurate design parameters when the time comes to physically build the prosthetics.

Multimedia applications also make great use of 3D scanning technologies: scanners are used to capture texture that is too tedious to create from scratch in a design application; they are used to accurately capture an actor's face for importing into CGI (computer-generated imagery) software where special effects may be added as desired with little inconvenience or risk to the actor; and also, 3D scanners are used to quickly import small-scale hand-made models into a virtual environment where they could be scaled and edited to become parts of a realistic computer-generated scene in a movie for example. This technique is called mocking-up. Figure 4 shows just such a practice with a rather famous model.

Figure 4: A Polhemus hand scanner being used in digitizing a model of the All-Terrain Armored Transport vehicle of Star Wars fame. (Image from the Polhemus website).

Finally, we mention the role that 3D scanning has in heritage preservation. Archiving efforts such as the 3D Examination of the Mona Lisa project [4] undertaken by Canada's National Research Council in association with the Louvre in Paris, and the Digital Michelangelo project [5] led by a team from Stanford University, are examples of archiving endeavors that forever have saved classical art in digital form for study and appreciation by future generations.

Figure 5: Images from the NRC's 3D Exploration of the Mona Lisa Project [4]. Left to right: The portrait being scanned by the NRC high-resolution color laser scanner; the scanned image; and a depth image of the Mona Lisa showing the surface's texture.

Figure 6: Michelangelo's David being scanned during the course of the Digital Michelangelo Project [5].

1.3. Haptics and Point-Based Models

Consider the clay models imported into a virtual environment by a designer at a car manufacturer. Instead of editing the imported model using the keyboard, mouse, and maybe even a digital pen and tablet, imagine the utility of using a haptic device to perform editing on the model in the virtual world analogous to sculpting the clay model in the real world. Armed with the ability to quickly undo, save, restore, copy, and share one's work, haptic sculpting is largely advantageous over its real-life counterpart.

Consider also the art student who just scanned her drawing into a virtual environment. She can experiment with haptic coloring, shading, and painting without risking the work she has

completed up to this point. Our fellow student also benefits from all the features of having her work in digital form that the car designer above benefits from.

Finally, imagine the increased richness of a museum patron's experience if he gets to touch the David statue via a haptic device since touching the real-life model is next to impossible. Or a patron's experience if she gets to trace the Mona Lisa via a haptic device and feel the texture of the centuries old canvas first-hand without putting the portrait at risk.

Making the case for the value added by haptics to any interaction with point-based models is making the argument for the value added by haptics to any haptic application; not only is it often useful and enriching, but it is just plain "cool".

1.4. Motivation and Problem Statement

As discussed above, point-based models are of great utility in a plethora of applications ranging from art to aerospace design. We also made the case for how much this utility could be increased if we were able to load point-based models into hapto-visual environments where they could be edited or explored haptically as well as visually.

For any model to be loaded into a hapto-visual environment, it must be haptically rendered, and as the state-of-the-art stands today, haptic rendering of point clouds is done indirectly by first creating polygonal meshes from these point clouds; there is no comprehensive direct approach available today to haptically render a point set. Not only that, but constructing polygons from point clouds is not without its drawbacks [1][2][3].

The above three arguments are the main motivation behind this work in which we present an A-to-Z approach to haptically render a point cloud in its native format. We aim to convey all the information available in the point cloud, be it tactile or kinesthetic, in a stable, speedy, accurate, and robust fashion.

Stated formally, the problem that we tackle in this work is that of devising a haptic rendering approach (full with collision detection and force response) for use with highly-detailed point-based models representing real-life objects, built using 3D scanners, and loaded into an environment haptically interacted with using a 3 degrees-of-freedom (3-DOF) stylus-based haptic device (e.g. the SensAble Omni shown in Figure 7).

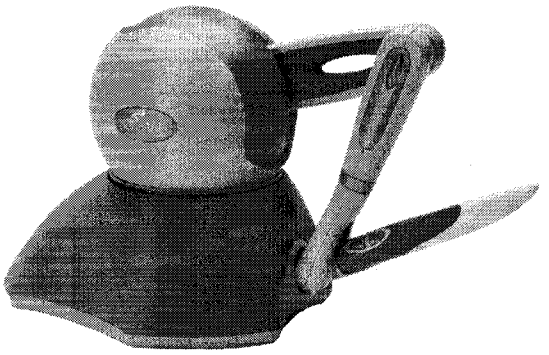

Figure 7: The Phantom Omni is a stylus-based haptic device. (Image from SensAble's website).

Formally, we set the following goals for our haptic rendering approach listed in no particular order:

- High fidelity: All the information present in the point cloud is to be rendered. This includes both kinesthetic (i.e. model geometry) and tactile (i.e. surface properties) information.
- No explicit surface reconstruction: Surface information must be deduced, not matter how implicitly, from the point cloud in order to haptically render it but we avoid all explicit surface reconstruction efforts to differentiate our work clearly from approaches that first build polygonal meshes before haptically rendering the model.
- No assumption of a multi-machine architecture: Some applications run haptic rendering, graphic rendering, and environment management code in two threads running on two

different machines, one dedicated solely to haptics. This is done in cases where haptic rendering is too expensive to run inside the main application loop while still maintaining the recommended 1 kHz run rate necessary for the interaction to remain stable. Our approach assumes a single machine architecture, which brings us to our next point.

- Strict worst-case runtimes: Our algorithm must not have a worst-case runtime slower than 1 ms (in order to maintain the 1 kHz haptic loop rate).
- Runtimes largely insensitive to model complexity: In order to consistently respect the 1 ms worst-case runtime limit, and keeping in mind that a given point cloud could potentially contain millions of points, our algorithm's complexity must be largely insensitive to model complexity.
- A stable and high quality haptic experience: Running under 1 ms is no guarantee of a stable haptic experience. Our algorithm must also produce force vectors that are temperate and accurate.
- Support for multiple models interacted with simultaneously: Our algorithms must place no limit on the number of models loaded simultaneously (assuming enough memory resources exist).

Finally we mention that, although this work is primarily concerned with point clouds the resolution of which is no worse than 2.5 mm (for reasons that will be explained later), and that result from 3D scans of real-life objects, it does not exclude arbitrary point-based models. In presenting the main facets of our algorithms in this work, we address how applicable each particular algorithm is to lower resolution point clouds, discussing modifications or alternatives that better suit generic point clouds.

1.5. Scope

In order to keep this work focused on solving our problem statement and adhering to the goals outlined in the previous section without shooting off on tangents, we make two assumptions in this thesis about the point-based models input to our algorithms.

We assume that the given model is a coherent and congruent point cloud that has all points in the same coordinate system and placed correctly in relation to each other. We explicitly make this assumption because of the nature of model acquisition via 3D scanning which does not produce these well-formed point clouds without some significant post-scan processing. [6] and [7] discuss the aforementioned processing which is outside the scope of our work. The reader should rest assured, however, that all commercial scanners we have come across over the course of our research come with their own proprietary software that produces the desired well-formed point clouds with little to no involvement from the scanner's operator. Furthermore, freely-available software libraries (resulting from the works in [6] and [7]) exist to handle raw scans and create well-formed clouds out of them should, for whatever reason, commercial software not be available to achieve the same purpose.

Throughout this thesis, we also make the assumption that the point clouds we are dealing with are highly detailed ones. How we define a highly detailed point cloud is covered in section 1.6, but we quickly mention here that unless otherwise noted, we assume that the point clouds handled are of a resolution equal to or better than 2.5 mm. Although this may seem to exclude many point clouds from candidacy for use with our algorithms, the reader will quickly realize that this is not the case when considering that most commercial laser scanners are capable of scanning objects at resolutions as good as 0.015 mm (see Table 1 page 28). As a matter of fact, the 2.5 mm mark that we set is quite relaxed.

Besides the abovementioned assumptions, we explicitly state that we do not pay special attention to the memory requirements of any algorithms presented in this work. This is not to say that we assume infinite memory resources (we will show that our algorithms actually run very well on mid-range machines), but rather that we do not concern ourselves in this work with the examining the memory footprint of our algorithms.

1.6. Definitions

Before proceeding any further with the details of our work, we define some technical terms that will be repeated throughout this thesis. The terms are listed in order of generality from most to least.

- Haptics: The term is derived from the Greek *hapteshthai* meaning "of or relating to the sense of touch". Haptics refers to the study of the physical perception and/or manipulation of objects through the sense of touch. In the strictest sense, the act of "touching" could be made by humans and/or machines, and the objects being "touched" could be real and/or virtual. In the context of this work, haptics refers to the feedback humans experience when using hardware transducers (i.e. haptic devices) to touch virtual objects.
- Virtual environment: A locale that exists in effect but not in reality; a scene that exists in software.
- Hapto-visual virtual environment: A virtual environment that is perceived by the user via his or her senses of touch and sight. Audition may be implied.
- Hapto-Audio-Visual Environment (HAVE): A virtual environment that is perceived by the user via his or her senses of touch, audition, and sight.
- Graphic rendering: The process of generating a visual representation of a described model. The model description typically includes geometry, viewpoint, texture, lighting, and

shading information. Graphic rendering translates this information into visual forms that are displayed on a computer screen or any other visual display.

- Haptic rendering: The process of generating a haptic representation of a described model. The model description typically includes geometry (kinesthetic) information and surface properties (tactile information). To haptically render a scene one needs: a collision detection algorithm that understands the virtual objects' representation; a force response algorithm to compute interaction forces; and control algorithms to communicate force and position information to and from the hardware.
- Point cloud: Also known as a point set, or a point-based model, a point cloud is a set of three-dimensional point coordinates corresponding to the outlines or surface features of an object. Point clouds are a common way to represent highly-detailed models, specifically these models' surfaces. Point clouds could also represent the interior of objects although this is not often the case. Point clouds are the typical product of 3D scanners (laser or contact) used to virtualize (digitize) real life objects in typically high resolutions.
- Vertex: A point on the surface of a model. A point in the point cloud.
- Point cloud density: A measure of the resolution of a point cloud. A point cloud's density is defined as the number of points the point cloud contains divided by the volume of the tightest fitting axes-aligned model-bounding box.
- Point cloud resolution: The maximum distance separating any two immediate neighbors in the point cloud.
- High-resolution point cloud: Also known as a dense point cloud or a highly-detailed point cloud, a high-resolution point cloud is a point cloud the resolution of which is equal to

or better than 2.5 mm. The reason we set the definition threshold at 2.5 mm is closely related to the fact that even a low-end 3D scanner can achieve resolutions better than 2.5 mm.

- Highly-detailed model: A model the representation of which provides enough information to render it at a resolution better than or equal to 2.5 mm.
- Depth resolution of a 3D scanner: The threshold along the scanning axis under which a scanner cannot discriminate between two distinct points. If, for example, the depth resolution of a laser scanner is 0.1 mm, that means that two points less than 0.1 mm apart along the scanning axis are seen as being at the same depth by the laser scanner. Note that the scanner's resolution along the other two axes is dependent on the method it moves (automatically in case of platform-mounted scanners or manually in case of hand-held scanners) in those axes and independent of the scanner itself. Typically, however, it is closely related to the scanner's depth resolution in case of platform-mounted scanners to ensure high-resolution data in all dimensions, not only in the scanning axis' dimension.
- Implicit function representation: A method to represent model surfaces by mathematical functions. For example, an implicit function representation of a sphere is of the form $f(x, y, z) = 0$ where (x, y, z) is a point on the sphere's surface. Implicit function representations are another method besides point clouds used to describe highly-detailed models.
- Collision detection (CD): In the context of virtual environments, collision detection is the process in which an environment's objects are examined to see if and/or when and/or where they come into contact with one another. The stylus's virtual representation coming into contact with a virtual object is an example of a collision that we need to detect in order to haptically render a scene. In this work, all references to collision detection imply real-time

CD (as opposed to offline collision detection which is used for example in non-interactive animation sequence rendering).

- Force response (FR): Also known as collision response. In the context of virtual environments, collision response is what occurs after a collision is detected. In the example of a stylus representation coming into contact with a virtual brick wall (collision detection), the haptic device's actuators providing stiff force feedback is a collision response.
- Height field: Also known as a height map, a height field is a 2.5-dimensional model representation that assigns a height value (the 0.5 dimension) to points in 2-dimensional space. Height fields are, by definition and by construction, uniformly partitioned (i.e. each two immediately neighboring points along one of the two 2D axes are separated by a fixed distance). Height fields are commonly used in representing texture in both graphic and haptic rendering applications.

1.7. Contributions

In this thesis, we claim the following contributions to the field of haptic rendering:

- A novel algorithm to test for collisions between an arbitrary but dense point cloud, representing a virtual model, and a haptic probe.
- A novel algorithm to test for collisions between a dense point cloud, representing a high resolution height field, and a haptic probe.
- An adaptation of the common god object/proxy approach to force response for use with point clouds.
- An adaptation of standard force shading algorithms for use with arbitrary point clouds not necessarily of a high resolution.

- An adaptation of standard methods in the literature to add surface properties (e.g. friction, deformation, texture) to haptic models represented by dense point clouds.
- A proposal to use information about the path a 3D scanner takes in scanning a real-life object to determine neighborhood information in the scanner's output point cloud.

1.8. Publications Arising from this Thesis

The following papers have preceded this work:

- Naim R. El-Far, Nicolas D. Georganas, Abdulmotaleb El Saddik, "A Collision Detection Algorithm for Point-Like Haptic Interactions in Highly Detailed Virtual Environments" IEEE VECIMS 2007, Ostuni, Italy. (Published).
- Naim R. El-Far, Nicolas D. Georganas, Abdulmotaleb El Saddik, "Collision Detection and Force Response in Highly detailed Point-Based Hapto-Visual Virtual Environments" IEEE/ACM DS-RT 2007, Crete, Greece. (Published).
- Naim R. El-Far, Nicolas D. Georganas, Abdulmotaleb El Saddik, "An Algorithm for Haptically Rendering Objects Described By Point Clouds" IEEE CCECE 2008, Niagara Falls, Ontario, Canada. (Published).

1.9. Organization

We start out in Chapter 1 (current chapter) by introducing point clouds, haptic rendering, and how both can be used together in various real-life applications. Chapter 1 also gives the reader an idea of how point clouds are built via 3D scanning and how typically dense these clouds are. This all serves as motivation for the rest of the thesis.

Chapter 2, our review of the state-of-the-art, will present the basic concepts of how models are represented graphically and haptically with special focus on highly-detailed models, the primary interest of this work. We also present current approaches to perform collision

detection (in graphics and haptics) and haptic force response, as well as present techniques that add visual and haptic surface properties to models.

Chapter 3 presents most of our contributions; in it, we present our approach to haptically render high-resolution point clouds. Collision detection and force response are first discussed, and then we move on to force shading, deformation, texture, and friction. Finally we present our experimental results.

In Chapter 4 we present our specialized approach to specifically render point-based height fields. Again, as we did with our general approach in Chapter 3, we discuss collision detection, force response, and all the surface properties and model features listed above.

We conclude this work in Chapter 5 with possible future directions as well as closing notes.

Chapter 2. Literature Review

In this chapter, we will review current techniques used to represent virtual models before we direct our attention to previous works in the three major sub-fields of haptic rendering: Collision detection; collision response; and control algorithms.

2.1. Model Representation

The phrase "model representation" in the literature has more often than not implied graphic model representation. There is very little work on representing haptic models since it is a much more recent problem than its graphic counterpart, and also because haptic model representation has been able to borrow successfully from the graphics domain.

2.1.1. Representing Graphic Models

When viewing a virtual object, two sets of information are relayed to you: the object's shape, and the color of every visible pixel on its surface. The former is described by geometry, and the latter by surface properties that result in a given pixel being a certain color given the viewpoint, the lighting, and the pixel's intrinsic color.

2.1.1.1. Shape Information

[8] identifies five major graphic model geometry representation techniques. They are: polygon soups, polygon meshes, implicit representations, halfspace intersections, and constructive solid geometries. Below is a summary of what [8] says about each.

Polygon soups: A polygon soup is, as the name implies, simply an unordered collection of polygons; a list of simple geometries with no structure. This representation carries no inherent information beyond the description of its constituent geometries, and thus is not attractive in applications such, as collision detection, where the extra connectivity information can be useful.

Polygon meshes: As opposed to the polygon soup, a polygon mesh has information about the edges connecting neighboring polygons. Figure 8 shows a polygon mesh (where the polygons are triangles) describing a dolphin. By just looking at the figure, one cannot tell if it is a polygon mesh or a polygon soup, however if the model file is inspected, the availability of common edge information would mean that the model is represented by a polygon mesh.

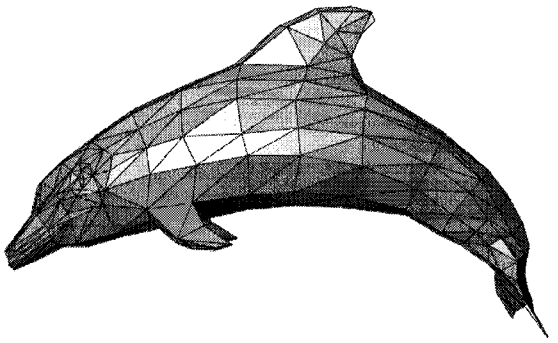

Figure 8: A polygon mesh representation of a dolphin. The polygons in the figure are triangles.

Implicit representation: Polygon soups and meshes list the geometries that make up a model; if a given geometry is not part of the soup or mesh, then it is not part of the model. This is called *explicit representation.* In contrast, we have what is referred to as *implicit representation*, which is a representation describing an object by means of a mathematical expression. An implicit representation is so named because the constituent geometries are not listed, but rather implied by the mathematical function describing them. Typically, an implicit representation is of the form $f : \mathbf{R}^3 \rightarrow \mathbf{R}$ where f is a function that maps a 3D coordinate (x, y, z) to a real number **R** such that all points (x, y, z) where $f(x, y, z) = 0$ are on the surface of the object represented. A sphere for instance is described by the function $f(x, y, z) = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0$ where (x_0, y_0, z_0) is the center of the sphere and r is its radius.

Halfspace intersections: Convex polygons are polygons that have no internal angles greater than 180°. A concave polygon, on the other hand, has at least one. Figure 9 shows a convex polygon and a concave polygon. It is fairly straightforward to see how intersecting halfspaces (or plains) at the edges of the convex polygon can define it. Halfspace intersection is a method used to describe convex polygonal models such as the cube which is the intersection of 6 halfspaces.

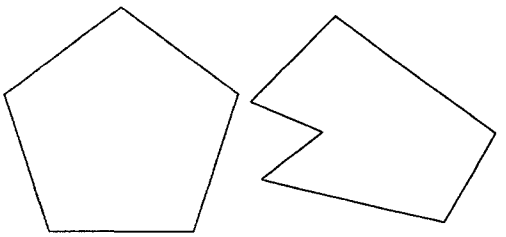

Figure 9: A convex polygon, such as the one on the left, can be represented by halfspace intersections, but a concave polygon, an example of which is on the right, cannot be.

Constructive Solid Geometries (CSG): The CSG approach involves the recursive building of arbitrary models by applying set-theoretic operations to basic geometries. A CSG representation is, in essence, a binary tree with union, intersection, and difference operators as internal nodes, and simple geometries as leaves. Figure 10 shows how a fairly elaborate object can be described by a 3-level binary tree.

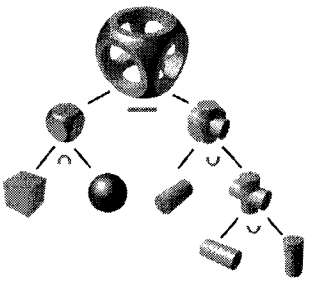

Figure 10: A Constructive Solid Geometry is in essence a binary tree where the internal nodes are set-theoretic operations and the leaves are primitive geometries.

2.1.1.2. Surface Properties

The surface properties of a model determine the color of each visible pixel on the model's surface given the lighting of the model's environment and the viewing angle that the model is seen through. Three major intrinsic properties describe a graphic model's surface: color, material properties, and texture.

In computer graphics, a model's surface color typically is an RGB value that encodes the color of the surface as if it were self illuminating (i.e. if it were a light source and did not reflect extrinsic light). A red sphere is red throughout with every visible pixel exactly the same shade of red. Incidentally, this uniform coloring can also be achieved in computer graphics using an ambient light source.

Material properties of graphic objects determine the amount and type of light reflected, refracted, and/or transmitted. In computer graphics, the common practice is to assign a surface a material property corresponding to how it interacts with incidental light; specular surfaces are surfaces that reflect light much like a mirror, while diffuse surfaces are surfaces that reflect light equally in all directions. Surfaces that refract light change its angle as it passes through the model while surfaces that transmit light are simply see-through surfaces that keep the light angle unchanged.

One of the common denominators between surface color and material properties is that they are typically invariant over a given visual model. A virtual fighter jet may be made of different components but its fuselage will have uniform color (except for the writing) and material properties, as will its wheels, and its translucent cockpit. This invariance along the many polygons that describe the shape of the model makes encoding color and material property information into a visual object's representation an easy task. An object-oriented approach may allow for each polygon or a group of polygons to be assigned a single surface

property as is the case in VRML or X3D (virtual scene meta-languages), or OpenSceneGraph (Scene-oriented libraries built on top of OpenGL), or, another approach would be to procedurally assign polygons surface properties as is in the case in graphics programming using OpenGL.

The third surface property we look at in this section, texture, is a different monster; it typically differs from one point on the object's surface to its neighbor. If we were to represent texture naively as the minute polygonal geometry that it actually is, we would be forced to use a large number of very small polygons that are tedious to create, expensive to store, and burdensome on the graphic rendering pipeline to draw. Enter texture mapping.

Gift wrap is to a blank cardboard box what texture mapping is to a polygonal box of the same shape; instead of drawing the gift wrap patterns on the cardboard box (a tedious process), we wrap it with separate patterned wrap. By the same token, and instead of adding minute geometrical shapes to the otherwise-flat-surfaced polygonal box (also a tedious process), we cover the box's surfaces with an image of the texture we wish to display.

To encode a texture in this manner into a model's graphics representation, we attach to the model the image texture map and ask whatever graphics library we are using to perform the necessary math to map the image correctly onto the object maintaining perspective, among other considerations. Alternatively, procedural texture mapping can overlay a virtual colored image (typically represented as a 1-, 2-, or 3-dimensional array of RGB values) onto our model foregoing the use of an image file altogether. In this approach, the RGB array is included in the model representation.

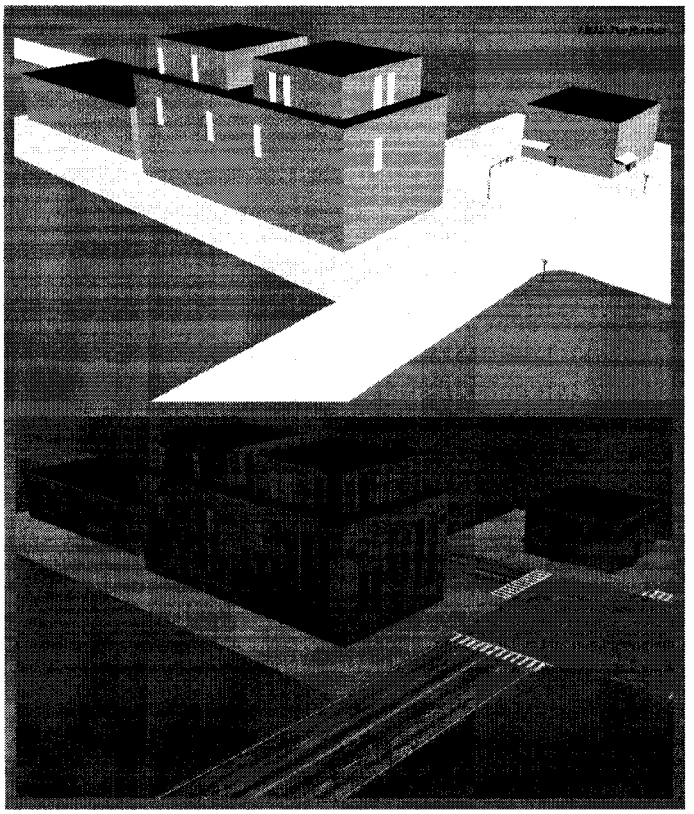

Figure 11: Texture mapping on its own produces static textures that look the same from every angle.

As you can see in Figure 11, texture mapping is not exactly realistic; the texture image is rendered the same 2D way in every angle, and the best we could hope for to create a the illusion of depth is to have some image pixels colored darker than others (note how the darker windows in the small building in Figure 11 appear 3-dimensional).

Height field mapping, also known as bump mapping, adds a third dimension to a visual model's surface without the need to modify the underlying model geometry. As the name implies, height mapping assigns (i.e. maps) a height value (a "bump" with a height relative to the local model surface) to each point on the model's surface. The change in the point's height off the flat polygonal surface creates texture as can be seen in Figure 12. As is the case with texture mapping, bump mapping can be incorporated into a visual model's

representation either by attaching the bump map file separately or by explicitly defining bumps at each point in a procedural manner.

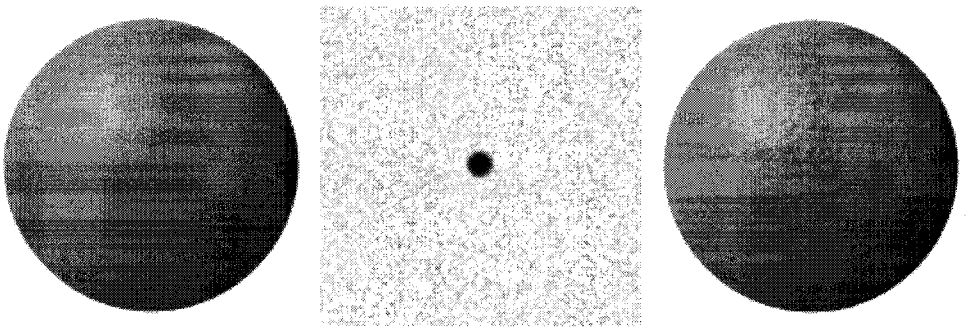

Figure 12: (Left) A non-textured model. (Middle) A bump map. (Right) The same model but bump mapped.

What bump mapping essentially does is it perturbs the surface normals at given points on the surface of the model by moving the surface according to the mapped height. Since surface normals are the main factor in determining lighting effects in graphic environments, this is sufficient to create the perception of 3D textured surface as evident in Figure 12.

2.1.2. Representing Haptic Models

Like a visual model, a haptic model is basically a geometric shape with surface properties. Following we discuss how to represent both geometry and surface information.

2.1.2.1. Shape Information

Describing geometry, in general, is a well researched area that we have surveyed in section 2.1.1. To the best of our knowledge, there is not a haptic-specific geometry description method although maybe there should be; the triangular mesh nature of computer graphics and the triangle-processing features of computer graphics hardware go hand in hand in efficiently rendering graphics. Why not have similar lines of thought in haptic rendering? This however is outside the scope of our work.

2.1.2.2. Surface Properties

[9] divides haptic surface properties into three categories: Stiffness, friction, and texture.

Surface stiffness is a measure of how resistant a surface is to deformation. In other words, it is a measure of how much force should be felt by the user, pressing on a haptic surface using a haptic device, in response to this pressing action. Stiffness is typically constant over the same haptic surface if not the whole haptic model, so it is fairly straightforward to encode into a haptic model's representation. Some haptic API's (e.g. Reachin API [10]) support a descriptive file format that encodes haptic properties (such as stiffness) alongside object geometry. Such approaches rely on extending a markup language (e.g. VRML) for use with a given API as is the case in Figure 13. Efforts have been undertaken to standardize this easy and intuitive haptic property encoding approach through the introduction of haptic markup languages. HAML [11] is such a language.

```
#VRML v2.0 utf8

# The Display node is the root node of the scene-graph
Display {

    # The children field is a collection that may contain
    # many Shapes or other child nodes
    children [

        Shape {

            # The appearance field contains an Appearance node,
            # which has fields for how the object's surfaces
            # should look and feel
            appearance Appearance {

                # The material field contains a Material node,
                # which specifies the color etc.
                material Material {
                    diffuseColor 0.5 1 0
                }

                # The surface field makes the object touchable
                surface FrictionalSurface {
                    stiffness 800
                }
            }
```

```
            # The geometry field specifies the shape and size
            # of the object
            geometry Cylinder {
                height 0.04
                radius 0.03
            }
        }
    ]
}
```

Figure 13: An excerpt from a Reachin VRML file [10]. Note the surface field which assigns a stiffness parameter to an object.

Another approach to encoding stiffness information is to do so programmatically vis-à-vis the haptic API used. All haptic API's encountered in our literature review (see section 2.2.1.3) support this method, an example of which is shown in Figure 14.

We address the details of how stiffness is rendered haptically in section 2.2.2.3.

```
virtual int SetBaseParameters
(
        double center[3]
        double orientation[3]
        double size[3]
        double ExtSpringStiffness
        double IntSpringStiffness
        double ExtDampingFactor
        double IntDampingFactor
        double ExtThickness
        double IntThickness
);
```

Parameters

center[3] Specifies the x, y and z center coordinates of the block in meters in a Cartesian frame.
orientation[3] Specifies the x, y and z orientation of the block in radians.
size[3] Specifies the x, y and z size of the block in meters in a Cartesian frame.
ExtSpringStiffness Specifies the stiffness of the external wall of the block in Newton/Meter. Default value is 1000.
IntSpringStiffness Specifies the stiffness of the internal wall of the block in Newton/Meter. Default value is 1000.
ExtDampingFactor Specifies the damping factor of the external wall. Default value is 1.0.
IntDampingFactor Specifies the damping factor of the internal wall. Default value is 1.0.
ExtThickness Specifies the thickness of the external wall in meters. Default value is 0.01.
IntThickness Specifies the thickness of the internal wall in meters. Default value is 0.01.

Figure 14: The HapticMASTER API is a C++-based API. In this code example from the HapticMASTER API's programming manual [12], the function `SetBaseParameters` defines the geometry and haptic properties of a geometric primitive in the haptic scene.

Friction, the second main surface property in haptic models, is a measure of how much tangential force resists the haptic probe's motion along the haptic model's surface. In 3-DOF haptics, there are two types of frictional forces: static friction and kinetic friction. Static

friction is the opposing force exerted on the haptic probe when the probe is being pushed in a certain direction, but not with enough force to cause it to move. This "sticking" effect is felt as long as the following condition is holding: $|\mathbf{F_t}| \leq \mu_s |\mathbf{F_n}|$ where $\mathbf{F_t}$ is the tangential component of the force being exerted by the probe on the haptic model, μ_s is the static friction coefficient which is a property of the haptic surface, and $\mathbf{F_n}$ is the normal component of the force being exerted by the probe on the haptic model (note that $\mathbf{F_n} + \mathbf{F_t} = \mathbf{F_{tip}}$ where $\mathbf{F_{tip}}$ is the force at the tip of the haptic probe being exerted by the probe on the haptic surface). As was the case with stiffness, static friction is encoded into the haptic model's representation by way of storing its coefficient, μ_s.

Once the tangential force vector $\mathbf{F_t}$ is large enough to break out of the static friction state, then the haptic probe starts moving along the object's surface but with an opposing kinetic friction force $|\mathbf{F_{kf}}| = \mu_k |\mathbf{F_n}|$ where μ_k is the coefficient of kinetic friction, another property of a haptic surface. Again, kinetic friction is encoded into the haptic model's representation by storing one constant which, in this case, is μ_k. The details of rendering friction are covered in section 2.2.2.3.

So far, we have been encoding haptic surface properties quite easily by adding a constant to the haptic model's representation. This is because, typically, a model has the same stiffness and friction coefficients along the same surface, but that is definitely not the case when it comes to a surface texture, where the norm is to have frequent variation along the same surface. We borrow from graphics to tackle the challenges faced by haptic texture rendering, noting that for the same reasons that encoding texture as intrinsic geometries in the model is too expensive to do in graphics, encoding texture as geometry in haptics is also expensive. It

is also unnecessary thanks to various cheaper alternative techniques that achieve the same purpose.

Recall that the goal of texture in graphics is to help determine the color of a pixel on the object's surface which, in effect, enhances the whole model's visual realism. Analogously, the goal of texture in haptics is to help determine the force vector felt at a given position on the object's surface which, in effect, allows us to haptically display non-trivial surfaces thereby enhancing the model's haptic realism. Haptic bump mapping is an often used technique that augments polygonal surfaces with height fields as is the case with graphic bump mapping. [9] and [13] discuss this adaptation to haptics of a common practice in graphics. Whether the bump field is procedurally created or predefined in a separate file, its description is included in the haptic model's representation.

Other than manipulating surface normals via height fields, different research has aimed to simulate texture by manipulating another surface property: Friction. [13] simulated sandpaper by manipulating friction coefficients, and laid the grounds for [14] and [15] which, using the same principles, were able to simulate a wide variety of surfaces. For such methods, a friction coefficient mapping is incorporated into the haptic model representation. The details of implementing texture haptically are covered in section 2.2.2.3.

2.1.3. Special Case: Representing Highly Detailed Models

Recall that in section 1.6, we defined a highly detailed (or high resolution) virtual model as a virtual model the description of which provides enough information to haptically render it at a resolution equal to or better than 2.5 mm. The threshold we place at 2.5 mm is specific to our research and is related to the depth resolutions of 3D scanners in use today. Table 1 shows a myriad of commercial scanners and their depth resolution values all of which are

significantly better than 2.5 mm. As a matter of fact, we were hard pressed to find recent commercial scanners with worse resolutions than 1.0 mm.

Table 1: Commercial 3D scanners and their depth resolutions

Scanner	Depth resolution
Cyberware 3030 MS [16]	0.05 mm – 0.015 mm
Faro LaserScan Arm V3 [17]	0.053 mm
Metris ModelMaker D [18]	0.05 mm
Micrometric Vision CLS 60 [19]	0.05 mm
Micrometric Vision CLS 60 PLUS [20]	0.038 mm
Microscribe MLX [21] (shown in Figure 1)	0.127 mm
NRC 3D Laser Scanner [22]	0.01 – 0.06 mm
Polhemus FastScan [23]	0.1 mm – 0.5 mm
Stanford Large Statue Scanner (manufactured by Cyberware) [24]	0.1 mm

Now that we have given the motivation for our definition of a highly detailed model, we turn to an important characteristic of these models. Note that if information exists to render the model at such high resolutions, then, by definition, the model captures geometric information fine enough that it explicitly describes texture; we need not worry about how to represent the model's shape independently from how to represent its texture. This makes highly-detailed models special and worthy of consideration on their own. (Note that other surface properties, such as stiffness and friction in haptic models, and material properties and color in graphic models, still need to be represented explicitly should they be included in the model representation).

The question we address in this section is how to represent these highly-detailed objects. Consider trivial models that are rare in real life, such as a perfectly flat surface, a perfect sphere, a perfectly smooth cube, etc. Note the keyword "perfect" in the previous descriptions. In a virtual environment, specifying a corner point and a length is enough to

fully describe a cube the faces of which are perfectly flat and smooth, however you will be hard pressed to find such a cube in real life. A real life cube could very well have indentations, grooves, and bumps (no matter how small) on its faces that we might dismiss at first glance but are nevertheless present. Also, consider a perfect sphere that can be easily described by an implicit representation. Such representation would yield the highest possible resolution of the model simply because the infinite points that make up this perfect's sphere surface are described mathematically by the implicit representation that is the sphere's function. Again, though, you will be very hard pressed to find a perfect sphere in real life. What you do find in real life are imperfect objects, which, if you intend to import at a high resolution into a virtual environment, you must also describe their imperfections. Doing so using implicit representations is expensive, prohibitively so most of the time.

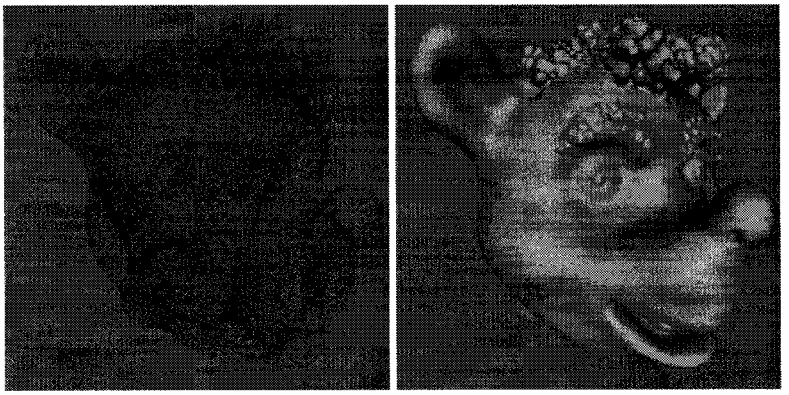

Figure 15: A point cloud is a collection of 3D coordinates corresponding to an object's surface and outlines. The point cloud on the left represents the model on the right [25].

The alternative is to represent objects using point clouds. To virtualize real life objects at a high resolution, 3D scanners are used. The NRC's Mona Lisa model [4] (Figure 5) and the Michelangelo Project's models [5] (Figure 6) are examples of high-resolution laser scans of real life objects. A detailed description of the laser scanning technologies used in the aforementioned two projects is beyond the scope of this work. Suffice it to say, however,

that such devices typically consist of a range or triangulating laser scanners that measure the reflection properties of emitted laser beams in order to determine the scanned object's surface information. The laser scanner is usually mounted on a translation/rotation platform to allow it to move around the scanned object (see Figure 5 and Figure 6) or it could be moved around manually (see Figure 1). In either case, independent scans from different angles are registered in order to coherently transform all the scanned points into a single coordinate system. The result of 3D scanning is, as we have explained in Chapter 1, referred to as a point cloud; a set of 3D point coordinates that correspond to the surface outline and features of the scanned object.

High resolution, or dense, point clouds can consist of hundred of thousands, if not millions, of points. This is to be expected of course because of the high resolution these models are scanned at. Table 2 below shows some models scanned with 3D laser scanners, their physical dimensions, laser depth resolutions these models were scanned at, and the corresponding number of points resulting from the aforementioned scans.

Table 2: Point clouds from real-life models

Model	Physical Dimensions (mm)	Scanning Resolution (mm)	Number of Points Raw/Processed (millions)
Thai Statue (XYZ-RGB model [26])	400 x 80 x 80	0.1	19.4 / 5.0
Vellum Manuscript (XYZ-RGB model [26])	220 x 30 (2.5D model)	0.1	~2.4 / ~2.2
Asian Dragon (XYZ-RGB model [26])	200 x 80 x 90	0.1	6.3 / ~3.6
Dragon (Stanford University CG Lab [26])	N/A	N/A	~2.7 / ~0.6
Happy Buddha (Stanford University CG Lab [26])	N/A	N/A	~4.6 / ~0.5
Digital Michelangelo Project (10 objects) [5]	Varying	≤ 0.25	Varying
NRC's Mona Lisa Scans [4]	N/A	≤ 0.01	N/A

Worth mentioning here is that laser scanning is not the only method of 3D scanning; contact scanning relies on positional input read via a contact head physically touching the real-life model to build the model's point cloud. Also, we note that the raw output of 3D scanners can be noisy so noise reduction techniques are used to smooth out the scanned models and eliminate outliers. The reader can find more on noise reduction, redundancy reduction, and other specifics of 3D scanning in [6] and [7]. As mentioned in section 1.5, most commercial scanners available today come with their own proprietary software that produces a well-formed point cloud from the raw scan data with little involvement from the scanners' users.

Inherently, point clouds are similar to polygonal soups in that they have no organization to the model data; just like a polygon soup is a collection of explicitly unrelated polygons, a point cloud is a collection of explicitly unrelated points. This scattering necessitates a surface reconstruction step in which surface information is deduced from the point cloud typically resulting in a polygonal mesh. Recalling the arguments we have made against representing surface properties (i.e. texture) geometrically, we recognize the shortcomings of representing highly-detailed models as polygonal meshes but most rendering algorithms in use today take only polygonal meshes as inputs, and so surface reconstruction is necessary (this lack of algorithms to directly haptically render point clouds is a prime motivator for this work).

2.1.4. From Point Cloud to Polygonal Mesh

As mentioned in earlier sections and staying in the context of point clouds, surface reconstruction is the process of creating a polygonal mesh representing the surface of a model using the said model's point cloud. Surface reconstruction is a multi-step process: preprocessing readies the point cloud data for input to the main algorithm, performing

operations such as noise reduction, up-sampling, down-sampling, hole-filling, and organization of the point cloud in a data structure; the main algorithm usually deduces point, edge, and face neighborhood information, builds explicit polygons, computes surface normals, and may also infer color and other surface properties if applicable; finally, a post-processing step typically normalizes and organizes the model information to fit a standard format.

Note that the preprocessing step shares functionality (e.g. noise reduction, up- ad down-sampling) already provided by the proprietary software that typically accompanies commercial 3D scanners or by software freely available online [6][7] but other functionality related, for example, to prepping the data for input to the reconstruction algorithms must still be accounted for in preprocessing.

The reconstruction algorithms themselves are outside the scope of this work but we refer the reader to [1], [2], and [3] for recent surveys of such algorithms and to [27], [28], [29], [30], [31], and [32] for the algorithms of most direct relevance to this thesis. Suffice it to say for our purposes here that surface reconstruction techniques typically follow one of the following approaches (all covered in the works cited above): Marching cubes and the related moving least squares approaches; Delaunay triangulation and the related Voronoi diagramming approaches; and surface fitting and approximation approaches.

Below, Table 3 shows the reported run-times of the algorithms in [27] through [32]. The reader will note that the times are quite lengthy for large point clouds.

Table 3: Reported run-times of several surface reconstruction algorithms

Method	Model	Reported time (mins)	Test machine
Projective clustering [27]	Stanford Dragon (~435K points)	4.53 mins on 1 CPU 1.55 mins on 4 CPUs	4-CPU 852, 2.4GHz Opteron processor system with 8GB RAM
Delaunay balls (NormFet) [28] (results reported in [27])	Stanford Dragon (~435K points)	7.78 mins on 1 CPU	4-CPU 852, 2.4GHz Opteron processor system with 8GB RAM
Delaunay triangulation [29]	Stanford bunny (~36K points)	23 mins	SGI Onyx with 512MB RAM
Delaunay triangulations [30]	Stanford dragon (~435K points)	18 mins	PC with 512MB RAM
	1M+ model	53 mins	PC with 512MB RAM
	3.5M+ model	198 mins	PC with 512MB RAM
Medial Access Transformation (Power Crust) [31]	30K	~6 mins	400 MHz Sun
Cyberware proprietary software [32]	500K – 3M	15 – 60 mins	High-end 2008 PC

2.1.5. The Graphics, Collision, and Haptic Scenes

Now that we have introduced the virtual hapto-visual models and how they are represented, we draw the reader's attention to a common practice in designing virtual applications; the separation of the virtual world into independent but synchronized specialty scenes, a practice made possible, in large part, thanks to the limitations of our senses.

The visible world around us, as we experience it, and without getting into philosophical arguments, is a singular world where the same object you see, is the object you touch, and the object you hear (should it somehow make noise). This is not necessarily the case in a virtual environment.

In an audio-visual virtual environment, we typically have a graphic scene and a set of predetermined events that would trigger audio. Besides the virtual scene that we see and hear, there could also be an invisible one, a collision scene. A collision scene in a virtual

environment is a scene that only contains objects that are subject to collision rules. For example, imagine a role-playing-game (RPG) where your character is walking down a corridor with a high ceiling. We can see the ceiling, but if the character can not jump high enough to touch it, we need not bother give it collision consideration. The ceiling is thus a part of the graphic scene, but not the collision scene. On the other hand, an impenetrable wall is a part of both scenes as is the ground on which the character walks.

There are many reasons why there is a distinction between a graphic scene and a collision scene. For one, there is no reason to add an object to the collision scene if there's no chance anything would collide with it. Another reason why we distinguish between the collision and graphic scenes is because of the growing complexity of graphics today. Should we use the geometry that we render to the screen as the geometry we perform physics-related computations on (such as collision detection), then our approach will demand significantly more computing resources.

A third reason why we should separate the graphic and collision scenes is our observation that collision data can be present beyond what is seen on the screen. Consider an RPG character walking backwards into a wall without actually seeing the wall; a collision has to be detected in such a scenario.

Making the distinction between the more complex graphic scene and the simpler collision scene is possible thanks to our ineptitude as humans in visually detecting whether exact collisions have occurred or not, and at predicting the exact outcome of a collision should it occur [8]. This allows programmers some liberties in approximating graphic scene objects in the collision scene (more on this in section 2.2.1). For collision-related purposes, a high-resolution graphic model can be represented by a basic bounding volume often without a

perceived loss in collision accuracy, and with significant savings in computational resources and time.

The above is not to say, however, that there are no reasons for keeping the graphic and collision scenes as one. [8] argues that: some duplication (costing more memory use than needed) is inevitable if the two scenes are separated; synchronizing both scenes in a dynamic environment would be required (extra work); and other logistic considerations (such as limitations on graphic designers) must be taken into account.

Beyond the graphic and collision scenes, we discuss here the haptic scene, which as you might guess, is the scene that contains "touchable" objects. We distinguish between the haptic scene and the collision scene mentioned above by restricting the haptic scene to haptic collision detection which has different constraints than graphic collision detection (discussed later in sections 2.2.1.2 and 2.2.1.3).

Some of the above arguments for and against separating the graphic and collision scenes could be extended to the haptic scene, but one crucial consideration must be kept in mind; humans' sensitivity to visual stimuli is different than their sensitivity to haptic stimuli, and oftentimes, both senses work hand in hand to help the human brain perceive an environment, real or virtual. This is evident by the hundreds of visual illusions (such as the one in Figure 16) that we are susceptible to, almost all of which could be dismissed if we were to haptically examine the illusionary objects. Furthermore, humans visually sample the world at a discrete rate much lower than the rate at which they sample the world haptically. This is evident when comparing the rate movies are played through a projector (24 frames per second) with the rate a haptic scene is rendered to a haptic device (500 – 1000 updates per second). For the reasons mentioned above, a haptic scene cannot get away with the

approximations that the collision scene may allow, and that is why it is seldom separated from the graphic scene, at least in terms of geometry.

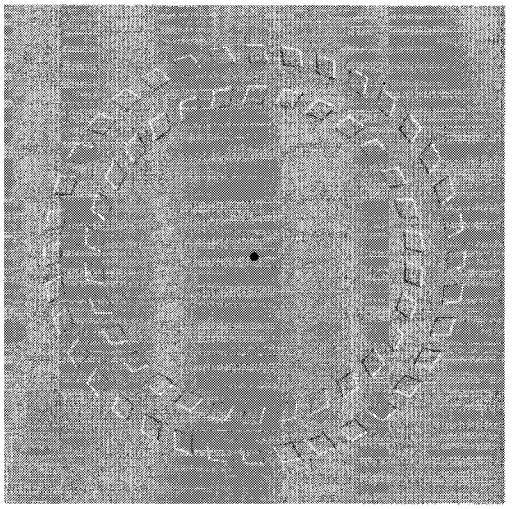

Figure 16: Looking at the dot in the middle of the above figure, move your head forwards and backwards to see the surrounding rings actually rotate. Feeling the rings if they were grooves on the paper would immediately dismiss the illusion.

2.2. Haptic Rendering

Just like the process of graphic rendering essentially takes a graphic representation and displays it on a computer screen, haptic rendering takes a haptic representation and displays it on a haptic device. This is the definition of haptic rendering on an intuitive level. More formally, [33] defines haptic rendering as "the process by which desired sensory stimuli are imposed on the user to convey information about a virtual haptic object".

A hapto-visual application can be conceptually broken down into three major components. The first is the simulation engine which is responsible for determining the virtual environment's behavior throughout the life of the application. By behavior we mean object positioning, events triggered, environment coordination, etc. The second major component of a hapto-visual environment is the rendering block which encompasses algorithms for graphics, haptic, and audio rendering. These algorithms translate the environment as it exists

in the simulation engine into one that could be experienced and perceived by the human user. How this translation is displayed falls into the lap of the third component we point to here which is the display block. The display block includes haptic, video, and audio transducers that physically interface the environment with the human user.

In examining Figure 17, which shows the placement of haptic rendering in a hapto-visual application, the reader might notice bidirectional arrows in the haptic path between the simulation engine and the end user compared to unidirectional arrows in the audio-visual path. This is because haptic interaction is a means for both input and output as opposed to graphics and audio which typically are output media.

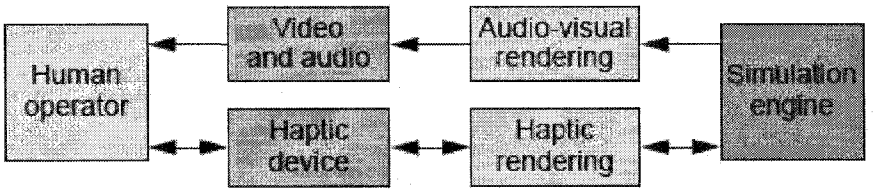

Figure 17: A block diagram of a hapto-visual application showing haptic rendering in context [33].

A haptic-rendering-centered look at a hapto-visual environment is shown in Figure 18 [33]. In sync with visual rendering, haptic rendering is coordinated by the simulation engine albeit at a much higher rate of around 1 kHz as opposed to the graphics update rate of around 30 Hz. The reason for the higher update rate has to do with the workings of a haptic device and how, specifically, it displays hard surfaces. Being a digital device, a haptic display operates in discrete steps, grouped together to form a single haptic loop, rendering force at the end of each aforementioned loop. For the device to seem to display continuous force despite the discrete time stepping it actually goes through, it must have a high loop rate, which has

experimentally been shown to necessarily near 1000 loops every second, or 1 kHz, in order for the device to display hard surfaces.

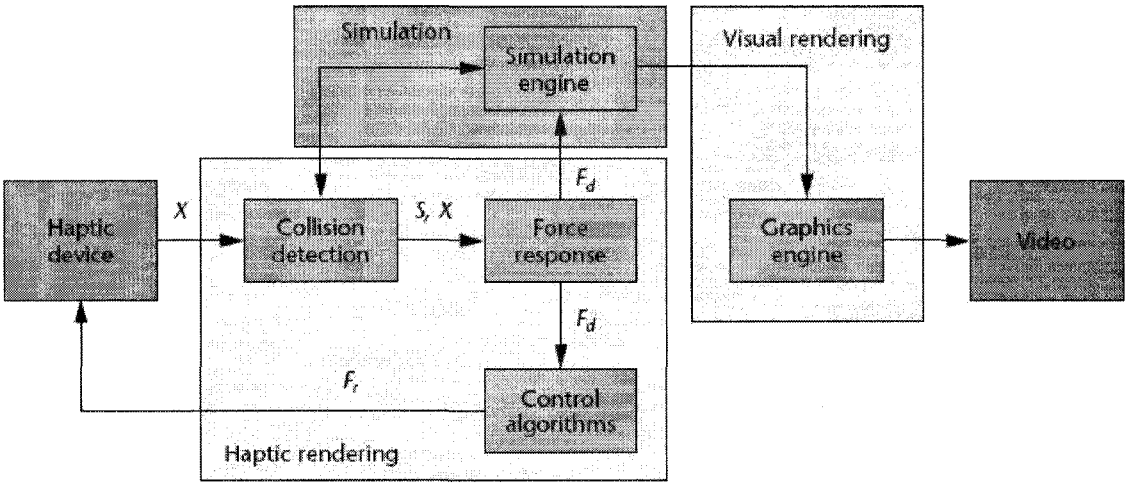

Figure 18: Haptic rendering is a 3-step process involving collision detection, force response, and control algorithms [33].

Each haptic loop, a haptic transducer's position in world space (denoted by X in Figure 18) is communicated to the collision detection algorithm of the haptic rendering block. The collision detection algorithm then determines whether the haptic end-effector (i.e. the single point representation of the end of the haptic device) is in contact with any of the world's objects. If indeed it is (i.e. a collision is detected), a set of contact points or polygons (S) is communicated to the force response algorithm alongside the end-effector's position. The force response algorithm then calculates the ideal reaction force vector (F_d) as per its physics formulas and the surface properties of the polygons in S. The ideal force vector is communicated to the simulation engine (which in turn updates the environment accordingly) and to the control algorithm which, taking into consideration the mechanical limitations of the haptic device, instructs the haptic device to display an actual response force vector (F_r).

Following, we examine the three major components of the haptic rendering algorithm separately: the collision detection algorithm, the force response algorithm, and the control algorithm.

2.2.1. Collision Detection

Collision detection (CD) is the process by which objects in a virtual environment are examined to see if, when, and where they come in contact with one another. This definition makes the importance of CD apparent; it is an integral part of any sort of virtual environment that claims any measure of realism. Be it computer games where a player expects his or her avatar not to walk through a wall, robots that have to autonomously navigate a physical environment, or an animated movie where a character dribbles a basketball and is sure not to have it go through the ground, applications for CD are as numerous as physics-governed virtual (or virtualized) environments.

Demands placed on CD algorithms, however, are different for different applications. The two major considerations in CD design are accuracy and speed.

Typically, for robotic navigation, substituting the geometry of the robot with that of a simpler bounding geometry (such as a cylinder or a box) is accepted for the purpose of real-time collision detection. If the bounding geometry is expected to collide with an object on the robot's path, then it is fair to assume that the robot is on a collision course with that object and hence must navigate away from it. This is an example application where CD accuracy is deemphasized in favor of speed.

On the other hand, consider an animation sequence not unlike those found in animated movies. The accuracy of any CD algorithm in such an application is essential so that the human viewers are not thrown off by an unrealistic event such as a character stepping on thin

air as if it were a step (unless that is what the filmmakers want). Although accuracy is of the essence here, CD speed is not; the movie is rendered once and is simply played back after. This is an example application where CD speed is deemphasized in favor of accuracy.

Other applications require both speed and accuracy. Consider a haptic surgery application that trains the user to perform a given surgical procedure on a virtual patient. In such an application, CD speed is of the essence since the haptic scene is rendered at a very high frequency (typically 1,000 Hz), a fact that sets a hard upper limit on the time necessary to render a haptic frame, and so, by extension, the time necessary to perform CD. Accuracy is also crucial when considering CD algorithms for use in such applications since surgical trainers are intricate simulations where fractions of a centimeter count.

In the following section, we will discuss some issues that we take into consideration when designing a CD algorithm, and then we will proceed to survey collision detection techniques used in graphic and haptic environments.

2.2.1.1. Issues in the Design of Collision Detection Algorithms

As mentioned above, in designing collision detection algorithms, computer scientists have to weigh their run-time performance needs against the CD algorithm's accuracy; generally speaking, the more stringent one factor is, the more relaxed the other is forced to be. In finding the most suitable combination of both, the target application in which the CD algorithm in question will run in must be well defined. This allows the algorithm's designer to make assumptions and lay down ground rules that can improve speed and/or accuracy.

In [8], the author discusses several issues that must be taken into consideration when designing or choosing to implement a CD algorithm. Below, we present our own adaptation of the aforementioned issues:

Internal Object Representation: For a CD algorithm to run within a given application, it must be able to handle the application's internal object representation; If an application were to represent the visual objects in its virtual environment using a polygonal soup, polygonal meshes, or CSG for instance, then the CD algorithm must be able to examine each respective representation, perform operations on it, and return results that in turn could be handled by the calling application. Although this might seem on the surface as a given, an application's choice of object representation could be critical to the performance of the CD algorithm. To illustrate this point, consider an application that requires a Boolean result (i.e. collision/no collision) from its CD component, and that describes its visual objects using an implicit representation. Now contrast the performance of a CD algorithm that is aware of the implicit functions with that of a more general one that perhaps performs distance queries; no matter the complexity and the lengths that the latter algorithm has gone through to perform well, it is evident that the former algorithm is much more suited for the application at hand.

A basic distinction, largely overlooked in [8], is between deformable and rigid bodies. Deformable objects are, as the name implies, objects that change shape (i.e. geometry) at run time in response to some stimulus (e.g. human skin). Rigid bodies on the other hand guard their form regardless of stimuli. Rigid body CD, although a challenge in and of itself, is typically an easier problem than deformable body CD.

Memory Requirements: Closely related to model object representation, memory requirements are a concern when designing a CD algorithm. In non-trivial applications where models could be complex and environments could contain tens of models, how much memory each model requires is a concern for the application's simulation engine, but more

importantly is a concern for the CD algorithm in that the less memory overhead the CD algorithm introduces, the better.

On the surface, it may seem that the best possible scenario is for the CD algorithm not to introduce any memory overhead at all and just use the graphic object to perform collision detection. However, as we have discussed in section 2.1.5, there could be some very good reasons why the graphic models should not be used for collision queries. In such a scenario, the CD algorithm designer must try to minimize the extra memory used to store the collision geometry alongside the graphic one. At the expense of extra computing, CD algorithms could actually reduce memory use below what is necessary by the graphic model through a level-of-detail approach. In such an approach, geometries are loaded at run-time at different levels of detail (i.e. resolutions) depending on environment parameters such as how close the model (or a part of the model) is to the user's area of interest. This level-of-detail approach could be also applied to the graphic scene should it be separate from the collision scene.

Collision Detection Algorithm Specialization: We previously discussed the positive effect that narrowing an application's domain has on the performance of a CD algorithm specifically when assumptions are made about objects' internal object representation. This in essence allows for the specialization, and hence improved performance, of a CD algorithm. Further specialization could be achieved if designing for specific environments. Consider particle systems for instance. [8] correctly asserts that treating such systems as a collection of individual particles is unnecessary for most simulations because a realistic CD algorithm could get away with treating the system as a collection of groups of particles. To illustrate this point, think of simulating dust in a video game; there is no need to treat each individual

dust particle separately when one can achieve an effect just as visually realistic treating the system as a collection of a few groups of several hundred dust particles.

On the other end of the spectrum, think of a world made up of a few simple geometries. No compromises are necessary is a such a world and objects could loaded in their entirety.

Type of Queries and Demands on Accuracy: In [8], the author discusses the different types of queries that CD algorithms must answer. Three distinct types are mentioned. The first, and simplest, is Boolean and answers the question "did a collision occur?". Boolean CD is also referred to as *interference detection* or *intersection testing*. It is, generally speaking, the easiest to implement, the fastest to execute, and may be enough for certain applications. Often, however, more information is needed, such as the location and time of collision. Some applications do not require more than a single point where the collision has occurred (e.g. stylus-based haptic applications), but more complex ones require the entire set of collision primitives (i.e. the *contact manifold*) be returned. The level of accuracy imposed on the contact manifold dictates if a CD algorithm designer could approximate the set in an effort to save precious time. Furthermore, some application may require finding the *penetration depth*, or in other words how far inside one object, another object is. The penetration depth is most commonly represented by a vector, and is required when simulating deformable objects (e.g. human skin) or rigid objects that, because of some external limitation, end up being penetrated (e.g. a haptic probe penetrating a presumably impenetrable haptic wall).

Environment Simulation Parameters: In the spirit of making assumptions about the virtual environment for the sake of improving performance, we look at the "big picture" that is the

entire environment. Environmental parameters that concern CD algorithm designers, the author in [8] states, include the number of objects and their motion.

Strictly speaking, if we were to perform a CD query between each object and each other object in the virtual environment, we need to perform a maximum of $n(n-1)/2$ pair-wise tests where n is the number of objects in the scene. This is in the order of $O(n^2)$; simply too expensive for large values of n.

As for the second environment parameter of interest to a computer scientist designing a CD algorithm, namely the motion of the objects, [8] defines two criteria for classifying motion types: Sequential vs. Simultaneous, and Discrete vs. Continuous.

Real life motion is simultaneous and continuous meaning that at time t all objects in a system could move, and those that do move over a given time interval continuously traveling over every single point in their path. Simulating this in a virtual environment is expensive and problematic. To illustrate this point, consider an object resting on top of another in a virtual environment. A CD algorithm operating on a simultaneous and continuous motion environment by advancing to the next time where collision is calculated to have occurred would advance in infinitesimal time steps when queried about the object resting on top of another. This problem is not without a solution (e.g. advancing at different rates in combination with broad-phase spatial partitioning, an approach that may be inappropriate for densely populated virtual scenes), but nevertheless, it is a problem.

Approximating simultaneous and continuous motion with sequential and discrete motion degrades the accuracy of the CD algorithm, but improves its speed in many cases. Sequential motion dictates moving one object at a time detecting collision as it happens and advancing the simulation only when all objects in a virtual scene have gone through their motion step.

Immediately, sequential motion manifests one of its major shortcomings to the reader who observes that perhaps there are two or more objects that if moved simultaneously would not collide but do if moved sequentially. The advantages however, besides the simplicity of the algorithm that assumes sequential motion as opposed to continuous motion, include ease of undoing motion if collision is detected (since one object is moved at a time), and ease of reporting contact manifolds since they could be built iteratively as the simulation progresses. Discrete motion dictates that time advances in fixed time steps, so in effect objects seem to "teleport" from one location to another as time progresses. Small enough time steps create the illusion of continuous motion, however as the time step grows, one can see how two objects that may have otherwise collided, might simply teleport across from each other without collision being detected between them.

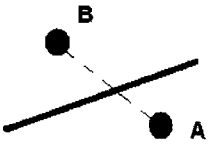

Figure 19: Temporal aliasing occurs if the haptic probe is sampled discretely at points A and B, in effect penetrating the surface of the model without this penetration being detected.

Temporal aliasing in the context of hapto-visual environments is a phenomenon in which a haptic end-effector could penetrate a thin object without that penetration being detected because the world is sampled discretely and not continuously. In Figure 19, at time t_0, the end-effector is at position A and at time t_1, it is at position B. Since the world is not sampled between t_0 and t_1, the end-effector has entered the object and exited it without the collision being detected. Because Vector Field Mapping does not take into account an end-effector's

previous position, only its current position, then there is no way to avoid temporal aliasing without modifying our response algorithm.

At time t_0, the end-effector is at position A and at time t_1, it is at position B. Since the world is not sampled between t_0 and t_1, the end-effector has penetrated the object without the collision being detected.

Speed: We have discussed performance issues throughout this section, all of which, at least implicitly, address speed and accuracy. [8] estimates that typically collision detection cannot take longer than 2 to 5 ms per frame in graphic applications and 1 to 2 ms in haptic ones. This is a strict upper-bound placed by the rates at which a graphic rendering loop and a haptic rendering loop run respectively. This is uncompromising regardless of the complexity of the environment and the number of objects to be tested. It dictates built-in optimization for just about any practically useful CD algorithm. This optimization could be in any and all stages of the algorithm, from the data structure used as input to the data structure returned as output, and everything in between. In a sense, all techniques mentioned before and after in this work (e.g. bounding volumes, spatial partitioning, etc) are optimization techniques, but worth mentioning in and of itself here is what is referred to as *temporal coherency*; a common optimization technique that assumes objects will not teleport as part of the application domain, and hence can proceed with CD only on objects that have moved since the last time step or frame. This is yet another example of a situation where knowing how objects in a given environment act can lead to a better performing CD algorithm.

Robustness: [8] defines robustness as "a program's capability of dealing with numerical computations and geometrical configurations that in some way are difficult to handle." In other words, a robust CD algorithm would return the correct result while a non-robust

algorithm would either return erroneous results or simply crash given complex inputs, degenerate geometry, or other ill-formed data.

[8] insists that numerical discrepancies, resulting from such operations as finite precision computations and geometrical mis-configurations are to be expected, and that a robust enough CD algorithm would be able to gracefully handle their occurrence. More preferably, the author states, a good computer scientist would integrate into all relevant parts of the development process measures to combat the occurrence of such divergences when designing a CD algorithm.

Pragmatic Considerations: The final point we discuss here encompasses pragmatic issues such as development time of both the CD algorithm and the application that ends up using it, the reusability of the CD algorithm for different class geometric representations and other applications, the reusability of the CD algorithm for different complexity, size, and density environments, and its interoperation with other components of the application calling it. Again, the consistent theme of tradeoff balancing is apparent here; investing time and money in a CD algorithm that works flawlessly and performs terrifically for a very small class of applications might not be the wiser choice over investing the same time and money for developing a less specialized algorithm that performs fairly well for a wider class of applications.

2.2.1.2. Collision Detection in Graphic Scenes

Collision detection algorithms in graphic scenes are naturally dependent on the method with which the scene is represented. Implicit representation is probably the easiest and most straightforward object representation for the purposes of collision detection, which simply becomes an algebra problem. Graphics built with Constructive Solid Geometry are different

in that collision detection would involve tests on the primitives (i.e. the leaves in the CSG trees). And halfspaces-intersection representation calls for halfspace collision detection (object-plane intersection).

Of prevalent use, however, in computer graphics is the polygonal mesh and/or soup representation of virtual models as described in section 2.1.1, so most collision detection work has been for these representations.

As mentioned earlier, it is too expensive to check every object in the scene against every other object in order to determine collision, so a prevalent approach in CD is to perform what is called *broad phase collision detection*, followed by *narrow-phase pair processing.* This divide-and-conquer approach removes from consideration objects that are not in close enough proximity, and hence unlikely to collide. Only objects that are determined to be close enough to perhaps collide are further examined in the narrow phase to assert whether they indeed are colliding or not.

Following we review narrow-phase pair process and broad phase collision detection which includes bounding volumes collision detection, bounding volume hierarchies' collision detection, and spatial portioning. We conclude this section by looking at other CD techniques not mentioned elsewhere in our review.

Narrow Phase Pair Processing: Generally speaking, narrow phase CD involves tests on geometric primitives. These tests include closest-point computation, separating axis test, and/or ray-segment tests, and are typically carried out after the broad phase CD algorithms have narrowed down the areas of interest to certain regions in the scene.

One of the most basic and powerful narrow phase CD algorithms is the *closest-point test.* If the two points closest to each other on two different objects are some distance apart, then a

collision could be ruled out. This trivial observation is non-trivial to implement. The problem of finding the two closest points is a minimization problem that could be solved using calculus (e.g. Lagrange multipliers) or using geometric observations. The latter approach is more prevalent in CD literature because virtual models are typically represented geometrically and not mathematically. [8] presents several algorithms for finding the nearest points depending on the polygonal representation used. Closest point algorithms exist for models built using planes, line segments, bounding boxes (axes-aligned and oriented), triangles, tetrahedra, convex polyhedra, etc. The details of such tests will not add to the value of this work so we refer the user to [8] for more information.

Another primitives test is the separating-axis test, which follows from the separating hyperplane theorem; if two convex sets are not intersecting, then there exists a separating hyperplane that has each object on opposite sides of. This theorem could be further qualified as follows: For symmetric objects with a center point, if the sum of the projections of their radii is less than the distance between them, then they are separated. This observation is evident in Figure 20.

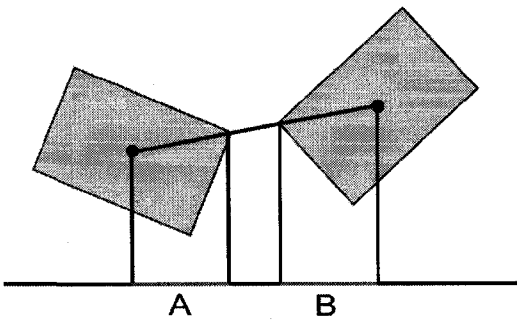

Figure 20: The separating axis theorem states that if the sum of the radii of convex objects' projections (A + B) is less than the distance between their centers (the arrow in the figure), then they are separated.

[8] gives other primitives intersection tests including sphere-plane, box-plane, cone-plane, sphere-box (axis-aligned or oriented), sphere-triangle, sphere-polygon, triangle-box (axis-aligned or oriented), and triangle-triangle.

[8] also surveys lines, rays, and segments intersections tests. The distinction between the three 1-dimensional geometries is that a line is infinite in both directions, a ray in infinite in only one direction, and a segment is finite in both directions. The authors review these geometries' intersection tests against a plane, a sphere, a box, a triangle, a quadrilateral, a cylinder, and a convex polyhedron.

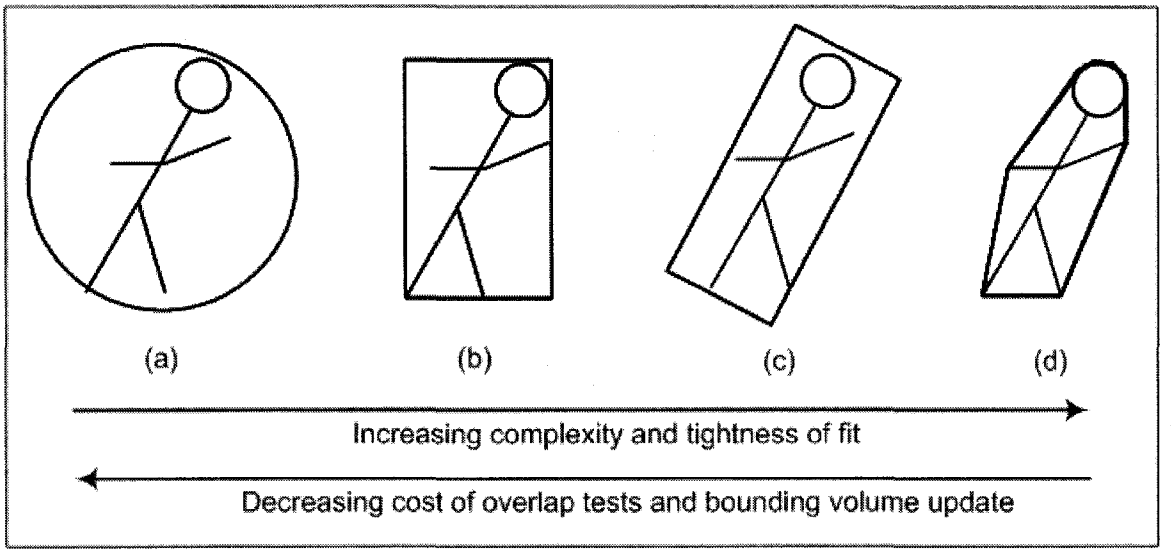

Figure 21: Bounding volumes commonly used in broad-phase CD [34]. (a) Bounding sphere, (b) Axes-aligned bounding box (AABB), (c) Oriented bounding box (OBB), and (d) *k*-discrete orientation polytope (*k*-DOP).

Broad Phase Collision Detection: The above survey covers the major aspects of narrow-phase CD. Broad-phase CD typically involves simpler and faster tests because it is required to run faster but not more accurately. As mentioned before, the whole idea behind a broad phase test is to quickly exclude objects that are not expected to collide with each other because they are relatively far apart.

One way to perform broad-phase CD is to approximate complex objects by simpler bounding geometries that lend themselves to cheaper and faster (but less accurate) intersection tests. Figure 21 shows the most common bounding volumes used to approximate complex geometries.

We note that bounding volumes are typically built in a pre-processing phase and adjusted at run-time. This is to improve run-time performance (it might even be prohibitive to build bounding volumes at run-time) albeit at the expense of extra loading time.

We now present the four most common bounding volumes used in CD: the axes-aligned bounding box, the sphere, the oriented bounding box, and the *k*-discrete orientation polytope.

Axes-Aligned Bounding Box**:** In examining the many bounding volumes used, we start with probably the simplest; the axes-aligned bounding box (AABB). As its name implies, an AABB is a rectangular six-sided box the face normals of which are parallel to the three axes of the given world coordinate system. An AABB could be described in three ways depending on what information is used the most in collision detection tests that use the AABBs. The first representation uses the minimum and maximum coordinate values along each axis, which allows the AABB in question to be described by its two diagonally opposite corner points. The second representation is given by a minimum point (i.e. the corner point with the smallest values in all three axes) and an array holding the three lengths of the box sides along the three major axes. Finally, the third representation of an AABB, similar to the second one, holds the AABB's center point and an array that stores the three half-lengths along the three major axes.

Point-AABB collision tests are straightforward and simple; if the point lies in the three segment projections of the AABB, then it is inside the AABB.

Updating the AABB of an object is also simple but only if the object translates in space. Should the object rotate arbitrarily, however, the AABB would most likely need to be recomputed from scratch at run-time.

Finally, we mention that AABBs are built using one of four methods: By examining the original point set and determining the minimum and maximum values along each axis (tight-fitting AABB); by hill climbing if neighbor information is available (tight-fitting AABB); by bounding the object's bounding sphere (loose AABB); and by using a preset fixed-size AABB that we know will always bound any object in the scene (very loose AABB).

Bounding Sphere: Another common bounding volume is the sphere, which is not only simple to perform intersection tests on, but also invariant to its bounded object's rotation (a rotated sphere is the same sphere). A bounding sphere is efficiently represented with four components: the x, y, and z coordinates of its center point, and a radius. As a matter of fact, the sphere is the most memory-efficient bounding volume in use equaled only by an axes-aligned bounding cube (a special case of the AABB).

Point-sphere intersection tests are simple; we calculate the squared distance between the sphere' center and the point, and then compare the squared distance to the squared radius of the sphere. If the former is less than or equal to the latter, then the point intersects the sphere. Note that in our intersection test, we used the squares of the distances, and not the distances themselves. This is because the square root operation is expensive and not needed since comparing the squares is enough to compare values.

As for how a bounding sphere is constructed, there are several methods. The first computes the object's AABB and then bounds the AABB with a sphere. The second is specific to point clouds and applies statistical analysis to find the object's direction of maximum spread [35];

then, the two farthest points from each other when projected on the axis that is the direction of maximum spread are computed and become the two opposite points on the bounding sphere's surface. Another approach to building a bounding sphere involves iterative refinement; we start with a fully bounding sphere, shrink it so that it no longer bounds all the points and reconstruct a sphere with a different center and radius that does encompass all the object points again. This process is repeated until some termination condition is reached. Finally, we present the Welzl Sphere [36]. Guaranteed to be a minimum bounding sphere, the Welzl sphere is built recursively using the observation that given a minimum bounding sphere and a point outside of that sphere, the new point has to be on the surface of any new bounding sphere. Some shortcomings in Welzl's algorithm were later addressed in [37].

Oriented Bounding Box: A third common bounding volume is the oriented bounding box (OBB). As the name implies, the OBB is a bounding box that is oriented with the object bounded (i.e. it is an AABB relative to the model's coordinate space as opposed to the world's coordinate space). There are many ways to represent an OBB: a collection of eight vertices, six planes, a corner point and three vectors, or a center point plus an orientation matrix and three half-lengths.

Constructing an OBB is also an involved process. There are three minimum volume OBB construction algorithms in the literature (one in [38] and two in [39]) but, by their authors' admission, none of the three is practical to implement, so we are left with approximation approaches. The literature contains several OBB approximation algorithms that can generally be categorized as either covariance-aligned [40] [41] or inertia-aligned [42]. Covairance-aligned OBB approaches work best for hollow models, while inertia-aligned approaches work best for solid ones.

k-Discrete Orientation Polytope: A fourth common bounding volume is the discrete-orientation polytope (*k*-DOP) also known as the fixed-direction hull [43] [44]. [45] defines a *k*-DOP nicely: Given a set of k vectors in 3-space, a discrete orientation polytope is a convex polytope all of whose k facets have outward normals from the given discrete set of *k* vectors. An axis-aligned bounding box is simply a special case of a *k*-DOP for a given set of $k = 6$ normal vectors parallel to the coordinate axes.

The fact that all objects bounded by *k*-DOPs share the same axes, representing a *k*-DOP is simple; we only need to store the min-max intervals for each axis. This also means a cheap intersection test between a point and a *k*-DOP; similar to a point-AABB test, a point-*k*-DOP test checks for point-segment intersection along all intervals of the model.

Keeping in mind that an AABB is a specialized case of a *k*-DOP, building a *k*-DOP simply is a generalization of building an AABB. An 8-DOP for instance is built by finding the min and max values along the four axes specified.

Finally, we mention that *k*-DOPs are obviously invariant to translation, however need rebuilding should their bounded model undergo rotation.

At this point in our literature review, we have looked at primitive tests which are the essence of narrow-phase CD, and looked at bounding volumes which serve to substitute complex model geometries with basic ones for faster broad-phase CD. What we have yet to do is improve on the number of pair-wise tests that we perform, not only the speed of the performed tests. Enter bounding volume hierarchies (BVH) and Spatial Partitioning techniques.

By arranging bounding volumes into BVH's (trees of bounding volumes), CD time complexity is reduced to logarithmic thanks to the logarithmic cost of traversing trees. The

details of arranging bounding volumes in trees are not much different than arranging any kind of data structure into a tree; the same traversal methods (depth-first search, breadth-first search, informed traversal, etc) are valid and the same construction methods (top-down, bottom-up, and insertion) are applicable.

Worth noting here is that the application of BVHs is not only to broad-phase CD, but also to narrow-phase primitive CD where BVHs are built over complex parts of the objects in an attempt to restrict expensive primitive tests even further.

Besides bounding volumes and bounding volume hierarchies, there is another approach to broad-phase CD, spatial partitioning. As the name implies, spatial partitioning is an approach in which the world space is divided into volumes that contain parts of the model, or models as a whole.

The uniform grid is the simplest spatial partitioning technique. It divides a given space into equal sized volumes that are associated with the objects they contain. Broad-phase CD then looks for objects that occupy grid volumes that are close enough, thereby eliminating from consideration far away objects.

A more complex, but efficient grid implementation is the hierarchical grid in which cell sizes are varying as opposed to uniform grids where cell sizes are fixed. This allows hierarchical grids to better accommodate environments where different models have considerably different sizes and a one-size-fits-all approach is not applicable.

More commonly used than both uniform grids and hierarchical grids are trees. Here we discuss two common spatial partitioning trees; the octree, and the k-d tree.

An octree is a tree structure that represents an axes-aligned 3D partitioning of space where each of a non-leaf-node's eight children (hence the name octree) is one of eight volumes of the partitioned 3D space.

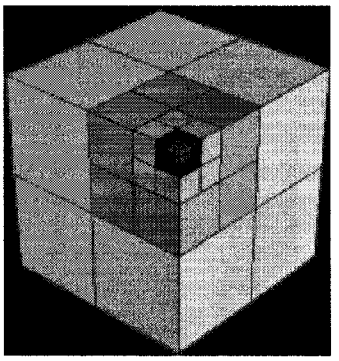

Figure 22: An octree conceptually divides a 3D volume into 8 equal-sized non-overlapping sub-volumes. This could be repeated recursively as the figure shows. [46]

A k-dimensional (or k-d) tree is a generalization of the octree. It is a tree structure that divides the space along k different dimensions (an octree is a 3-dimensional tree). Note that the division dimensions do not necessarily correspond to the dimensionality of the model space; a k-d tree uses splitting planes that are perpendicular to one of the coordinate system axes k number of times.

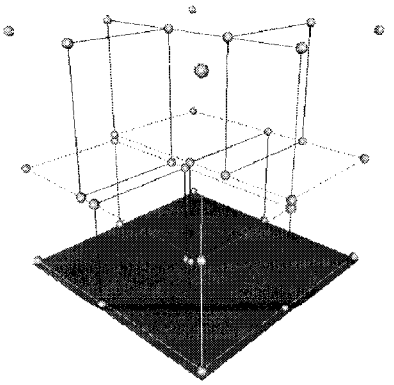

Figure 23: A 3-dimensional k-d tree. The first split (red) cuts the root cell (white) into two subcells, each of which is then split (green) into two subcells. Finally, each of those four is split (blue) into two subcells. Since there is no more splitting, the final eight are called leaf cells. The yellow spheres represent the tree vertices. [47]

CD tests that use spatial portioning in their broad-phase do so by traversing an octree or a k-d tree until they reach the volume of interest. On the other hand, and in the case of grids, a hashing function typically maps a 3D location to a volume the neighbors of which are also searched in finding our area of interest's bounding volume.

Other techniques: For the sake of comprehensiveness, we mention some other CD techniques used in computer graphics. They are however outside the scope of this work since we do not use them or any of their principles in our approaches.

Table 4 shows a summary of some of the most common CD algorithms.

Table 4: A summary of common CD algorithms [50].

Model Representation		Approach to Collision Detection	Simulation Environment			Type of Queries
			Processing	Motions	Object Type	
Convex Polytopes		Linear Programming	Pair	Static	Deformable	Boolean
		Minkoswky Sums and Convex Optimization	Pair	Static	Rigid	Disjoint Separation Distance
		Tracking Closet using Geometric Locality and Motion Coherence	Pair	Dynamic	Rigid	Disjoint Separation Distance
		Discrete Orientation Polytopes	n-body	Static	Deformable	Discrete Intersection
Polygonal	Convex	Calculating Minimal Separation	Pair	Static	Deformable	Disjoint Separation Distance
	Structured	Hierarchical Data Structures	Pair	Static	Deformable	Discrete Intersection
	General	Tighter-Fitting Bounding Volumes	Pair	Dynamic	Deformable	Discrete Intersection
Non-Polygonal	Implicit Surfaces	Analytical	n-body	Static	Deformable	Discrete Intersection

Linear Programming (LP) is a powerful technique used to find the closest points on and the distance between convex polytopes to detect whether they intersect or not. This is done by optimizing a linear objective function describing the polytopes.

Minkowsky Sums are also used in calculating minimum distances between objects by recasting the problem as a configuration problem in the application domain of this technique. Tracking Closest Features is an approach proposed in [48] and optimized in [49]. This approach uses the presumption that there exists some coherency in the motion of objects, so that if we know the closest point at a given time t, then we can assume that the closest point at time $t + 1$ is near where it was at time t. This extends to edges and faces of models, not just points.

2.2.1.3. Collision Detection in Haptic Scenes

Generally speaking, the inadaptability of graphics CD in hapto-visual applications forces haptic collision detection algorithms that borrow from the graphics domain to run at rates slower than desired (typically around 100 Hz) [51]. The alternative is running haptic CD (alongside other aspects of haptic rendering) on a dedicated machine [52].

Some graphic CD algorithms that can be adapted for use in haptics include those described in [53][54], [55][56][57], [58][59], [60], [61], and [62]. All of these algorithms use a variation of bounding volumes to build a hierarchy that is traversed in order to spatially narrow the area of interest to a small volume. The bounding volumes used, in the order referenced, are: spheres, axes-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), S-bounds, spherical shells, and k-discrete orientation polytopes (k-DOP's).

Other algorithms directly partition the world space using grids built with octrees [63], K-D trees or BSPs [64], voxel maps [65][66], or in some other adaptive or uniform fashion. Special hashing functions are also used in spatial partitioning by mapping a 3-dimensional coordinate to a 1-dimensional structure holding pointers to volumes in the world space [8].

In our review of the state of the art in haptic-specific collision detection, we looked at two open-source haptic API's, namely CHAI3D 1.51 [67] and H3D 1.5 [68], as well as the proprietary SensAble Open Haptics Toolkit [69]. All three APIs use a variation of the bounding-volume hierarchy (BVH) approach to perform a broad-phase narrowing of the model down to the area closest to the haptic probe, and then, in narrow-phase processing, determine collision by using an accurate (but expensive) method such as segment-triangle intersection detection.

2.2.2. Collision Response

Collision response algorithms compute the forces that ideally the user should feel based on the state of the haptic scene (2.2.4 discusses the forces actually felt). Naturally, the force vectors computed depend on how many degrees of freedom (DOF) the haptic device has. 1-DOF devices, such as pinchers or grippers, display trivial 1-dimensional force vectors. 2-DOF devices, such as a haptic mouse, display forces in 2-dimensions. Most stylus based devices (such as the SensAble Omni shown in Figure 7) are 3-DOF devices displaying force in the Cartesian dimensions while some add rotational forces (torque) in one, two, or three dimensions for a maximum of 6 degrees of freedom corresponding to the 6 possible transformations in 3D space.

The paradigm used to display a force vector on a haptic device is that of the spring; a spring with a coefficient k exerts a force vector $\mathbf{F}$ on an object attached to its end in the opposite direction of the object's pull and with magnitude linearly proportional to the elongation distance $|\mathbf{X}|$ that the spring experiences. This is called Hooke's law, famously stated in Latin as "Ut tensio, sic vis" which means "As the extension, so the force". In equation form, Hooke's Law is $\mathbf{F} = -k\mathbf{X}$.

For translational 3-DOF devices, there are two main methods to compute ideal forces (and hence create spring effects) by the collision response module of the haptic rendering algorithm. The first is Vector Field Mapping [33], a concept borrowed from physics, which assigns a force vector to each point in a given space. This force vector would affect any object that would come to be positioned at that point in space. If the set of force vectors is available, then computing the force affecting an end-effector at any point in a given space becomes a matter of looking up the force vector for that point. [33] claims that this is a method that "works well", however it is susceptible to temporal aliasing because it does not take into account previous end-effector positions. This is not to mention how the force vector map is built in the first place, and also its limitations when used in dynamic environments which dictate that the force map be updated to reflect the movement and, perhaps, deformation of the environment's models.

The second method used in computing force response is that first introduced in [70], then furthered in [71] and [9]. The notion of a god object in [70] and a proxy object in [71] places a spring between where the end-effector actually is and where it should be. Haptic probes could very well penetrate haptic models that are meant to be impenetrable, so although the actual haptic end-effector is inside the haptic model, it should be on its surface at the point closest to where its actual position is (with some constraints discussed later). This closest-point is dubbed the god object or the proxy-object and is simply the point where the haptic end-effector should be. It is self-evident that the deeper the end-effector is inside a conceptually impenetrable haptic model, the bigger the force that should be exerted to return the haptic end-effector to the model's surface, and this is where the spring comes in (see Figure 24).

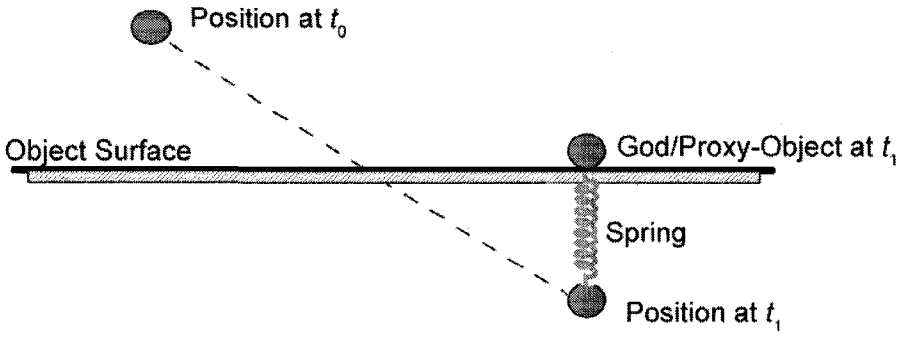

Figure 24: A god object or a proxy is placed at the closest point on the surface of the penetrated virtual model. A spring is then created pulling the actual end-effector from its actual position to the position of its god object.

2.2.2.1. The God Object/Proxy

Naively finding the nearest surface point and designating it as the god object is enough to render forces in many cases, but it fails in certain scenarios shown in Figure 25. Although the haptic probe has penetrated the box's right surface in Figure 25a (orange circle is the probe, and the blue circle is the god object), the nearest surface point to the probe's location as it crosses the Voronoi boundary abruptly becomes the point on top of the figure. If the god object were to be placed there, then the user would feel a pop-out (also referred to in the literature as a pop-through) force that is unrealistic. The same pop-out effect is felt in the thin object in Figure 25b; once the object's midline is crossed by the haptic probe, the god object would be placed on the left-hand surface, opposite of where the probe entered, and hence the probe will pop-out unrealistically.

To counter these effects, [70] and [71] impose constraints on the location of the god object/proxy.

The god object approach in [70] finds a set of model surfaces that have been crossed by the haptic probe since it was last sampled in the previous haptic frame. These surfaces are extrapolated into one to three planes (one for a surface, two for an edge, and three for a point), the equations of which are fed to a system that uses Lagrange multipliers to minimize

the distance between the haptic probe and the constraint surfaces effectively finding the god object.

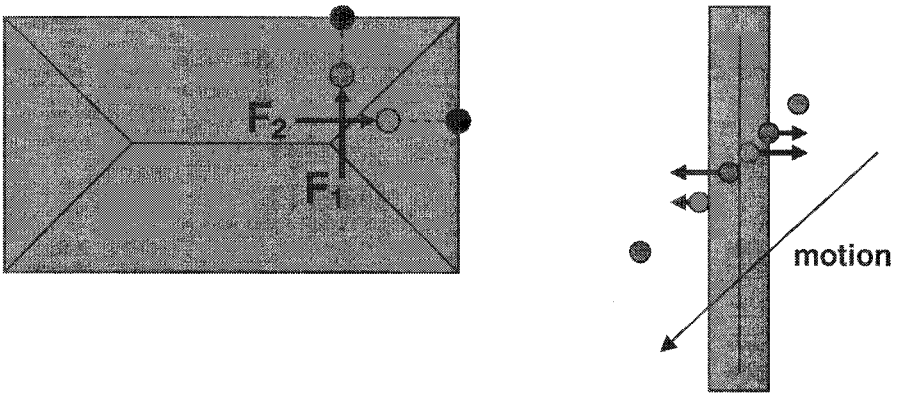

Figure 25: The pop-out effect (a) when crossing Voronoi boundaries, and (b) in thin objects. [72]

[71] shares the constraint-based approach first introduced in [70] but with two main modifications. First, [71] models the proxy (what the paper refers to the god object as) as a mass-less sphere; this is to avoid instances where the point haptic-probe may slip through a crack in a polygonal mesh during collision detection. Second, [71] adapts robotic path navigation to haptic models in order to determine the constraint surfaces, and then the location of the god object. When a robot is planning its path to a given destination, it will proceed following the shortest path it can find to that point (i.e. a straight line to its destination). If it is obstructed along the way, then it will pause at the obstacle and then proceed attempting to minimize its distance to the goal point. The final position the robot settles at is in essence the closest distance it can find to its destination along the obstacle surface it has encountered. Similarly, the approach in [71] advances the proxy along a straight line from its location at time t to its goal location at time $t+1$. If the proxy is obstructed by a model surface, it moves along that surface locally minimizing its distance to

its goal location until it reaches the minimum distance possible, the location of which becomes the proxy.

2.2.2.2. Force Shading

[9] improves on a refinement first introduced in [71], namely force shading, which borrows from Gouraud shading in graphics. Figure 26a shows a polygonal figure with flat shading. Note that each individual surface is colored the same in accordance with the lighting of the environment, but that neighboring surfaces have noticeably different colors. Graphics shading breaks down color discrepancies between neighboring surfaces basically by assigning weighted colors to the same surface, each pixel getting a weighted value dependent on its location on the surface. The result of a shaded surface is shown in Figure 26b.

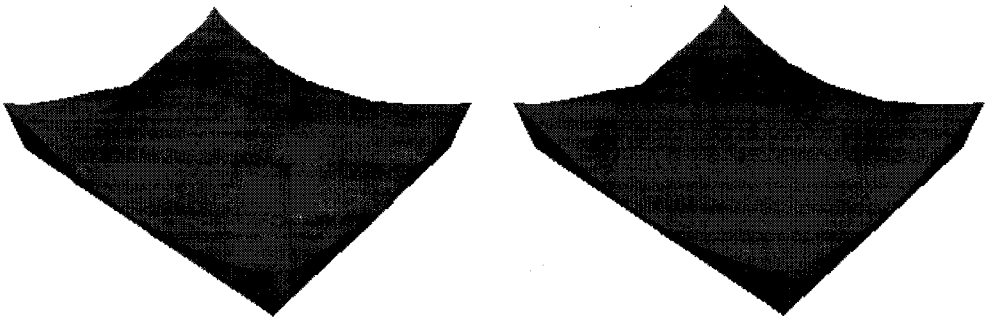

Figure 26: Gouraud shading in graphics is a basis for force shading in haptics. (a) The left polygonal mesh is flat shaded. (b) The right polygonal mesh is Gouraud shaded.

[9] brings this to haptics as is shown in Figure 27 (which demonstrates the concept in 2D). The normal force vector at any point inside the polygon (the intersection of the three segments starting at the polygon's vertices) is equal to the weighted average of the normals at the vertices of the polygon as per the following formula:

$$N' = \frac{\sum_{i=1}^{3} A_i \cdot N_i}{\sum_{i=1}^{3} A_i}$$

Where N' is the normal at a given point inside the polygon, A_i is the area of the triangle i, and N_i is the normal at vertex i.

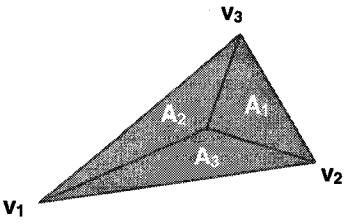

Figure 27: Force shading in haptics assigns a force vector to each point on a polygon equal to the weighted average of normals at the polygon's vertices.

Force shading does have the effect of rounding edges and peaks in models, however [9] asserts that the use can still feel the existence of these geometries thanks to the discontinuities in the force magnitudes.

2.2.2.3. Surface Properties

Finally, [71] proposes that controlling the location of the proxy is enough to create surface property effects such as friction (the proxy is moved at a slower rate on the surface than the probe moves inside the object), deformation (allowing the proxy to also penetrate the object), and texture (adjusting the proxy's location vis-à-vis a height field).

Since [70] and [71], there have been many works to try to improve the fidelity of the haptic force response. One of the more significant works is that in [73] which optimizes the proxy object approach for use with models that have a high number of polygons in their triangular meshes. Besides force shading, [9] also proposes to find the god object/proxy faster and more effectively by breaking down the world into a database of primitives with information about their neighbors, and also furthers [71]'s research in creating more haptic effects by manipulating the location of the proxy.

We now turn our attention specifically to haptic deformation.

Haptic deformation can be applied in two ways: The first is by updating the haptic model to reflect the deformation; and the second is by simply manipulating the god object's placement while keeping the model itself intact. The latter approach was first proposed by [71].

Updating the model to actually reflect the deformation's effects is the straightforward way of implementing haptic or graphic deformation. However, this method is the more expensive of the two since computations must take place to determine which points in the point cloud are affected, and then determine their new locations in response to the deformation. Extra computations are also necessary to update the spatial structure storing the model in memory and also to infer new points should we want to keep the model's resolution from degrading in case of stretching. Partitioning computations can be relatively cheap in case of non-adaptive grids such as uniform partitions, but can require expensive repartitioning in case of adaptive structures such as octrees.

How a model deforms, the mathematical and physical models for deformation, and the many approaches to practically implement geometry updates at runtime, are all research fields in and of themselves and are outside the scope of haptic rendering (see [74] for more information). Typically, the deformation algorithm, in the context of hapto-visual applications, is placed at one of two locations: either, as a black box, between the collision detection block and the force response block in the haptic rendering pipeline; or in a separate but parallel lower-rate pipeline. The first approach receives collision data (specifically, if and where collision has occurred) from the CD block, and based on the connectivity information of the model primitives in the area of interest and the preset deformation model, the model geometry is updated. The next step is to communicate to the force response algorithm the contact points but adjusted to reflect the deformation's effects. The upside of using this

approach is its guarantee that the force response algorithm is receiving the most up-to-date information about collision and so will render the correct forces at all times. The downside, however, is that since the deformation algorithm is placed inside the haptic rendering pipeline, it has to be fast enough not to degrade the haptic loop's run-rate. Often, to maintain its speed, the deformation algorithm will have to sacrifice accuracy or functionality.

The other approach is to have deformation run in parallel to haptic rendering and run at slower rates. Often in this approach, the deformation algorithm will have its own CD component and will communicate with the haptic rendering pipeline via the model geometry; it deforms the same geometry that the haptic rendering pipeline has access to. The upside to this approach is that deformation can be accurate and "take its time", so to say, since it runs independently of the high-rate haptic loop. The downside, however, is that synchronization issues might occur should the deformation algorithm attempt to update the model geometry just as the haptic rendering algorithm is accessing it.

The second approach to implementing haptic deformation, first introduced in [71], proposes manipulating the placement of the proxy object, without the need to update the model itself, in an effort to convince the mind that deformation has occurred. Instead of placing the god object on the surface of the model, it is placed further inside the model to create the perception of haptic deformation. This approach carries with it very small overhead, namely the calculation of the god object's position along the force vector that would be rendered in case of a rigid body, and hence is very attractive in haptic applications. There are some downsides, however. The first is that despite the fact that we are saving time and effort by only applying deformation to the current god object, the graphics must still reflect the deformation along the whole affected region in a hapto-visual environment. Another

downside is the need to have a stiffness property associated with every point or a collection of points in the point cloud to help the force response algorithm determine where to place the god object in accordance with the given point's stiffness (i.e. deformability). A final downside we mention here, explicitly stated in [71], is the possibility of an abrupt change in the force vector rendered if moving from a deformable surface to a non-deformable one. In such a scenario, the force vector rendered might be very large in order to compensate for the penetration allowed by the deformable surface once the HIP moves onto a non-deformable one. This sudden change in the force vector can lead to instability and degradation in the haptic experience.

2.2.3. Collision Detection and Force Response Algorithms for Use with Point Clouds

Our literature review produced only three works that looked at models primarily as point clouds for the purpose of collision detection and force response.

The US patent in [75] uses uniform spatial partitioning to arrive at a voxel containing a portion of the object's surface which is then decomposed into points. The distances between the point representing the haptic probe and all points in the aforementioned voxel are computed, and should any of them fall below a given threshold, a collision is detected. This method is rather incomplete since it does not account for points in neighboring voxels, and it also places no demand on the density of the model point cloud, so there is significant room for false results. [75] also does not address how a model is decomposed into its constituent vertices and how to check to see if the probe is inside or outside the model (necessary since [75] implements point-based and not segment-based CD). As for force response, [75] uses a penalty-based approach and so it will render incorrect forces in the cases mentioned in section 2.2.2.1.

[51] differs from [75] in two areas: how the algorithms find vertices neighboring the haptic end-effector's position, and how narrow-phase CD is performed after the neighboring vertices are found. As opposed to the approach in [75] explained above, [51] extracts, in a preprocessing phase, neighborhood relationships between the vertices using the Voronoi theory and stores these relationships in what the work refers to as a V-GRAPH (where V presumably references Voronoi diagramming).

Figure 28 shows a Voronoi diagram which is made up of a collection of cells represented in the figure by different colored regions. The main property of these cells is that each point that lies within their borders is closest to its cell's center (represented by the black dot in the figure) than to any other cell center in the diagram. [51] uses this special partitioning to quickly find the points in the point cloud that are nearest to the haptic probe's location.

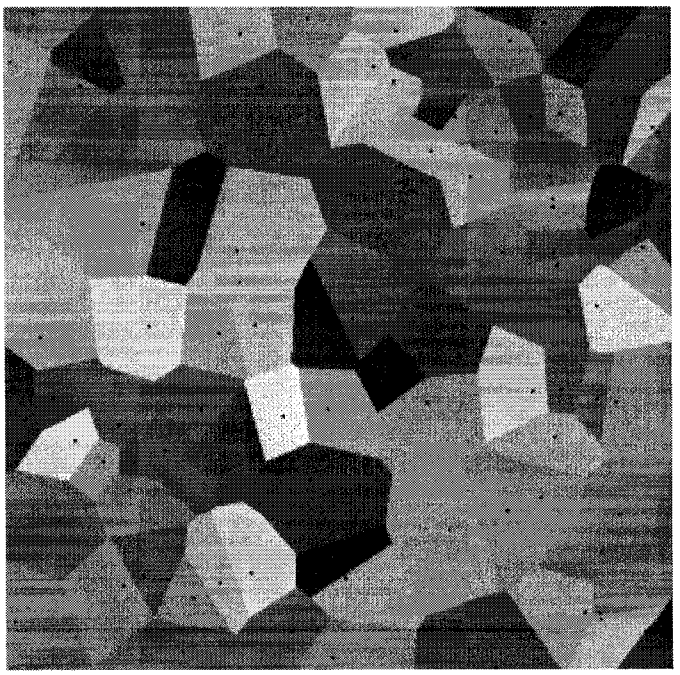

Figure 28: A Voronoi diagram.

At runtime, the CD algorithm in [51] uses the V-GRAPH to find vertices in the neighborhood of the haptic probe finally arriving at the one closest to the probe. Narrow-

phase CD involves a standard segment-triangle intersection test (although [51] looks at the model primarily as a point cloud, it does require its triangular mesh representation be present in order to build the Voronoi diagram, to perform narrow-phase CD, and to render a force response should collision occur).

The approach in [51] is particularly well-suited for highly detailed, complex virtual environments since it deals with the models as point-clouds and demands that these point clouds be dense. [51], however, does not explicitly state just how dense an input point cloud should be for its CD algorithm to work with haptics and, as mentioned above, it also uses the model's triangular mesh so it is not purely a point-cloud-based approach.

Finally, the third work we came across was [52], which describes a workflow to find the nearest surface point to a given haptic probe. The described workflow sets off in a preprocessing phase to build a bounding volume hierarchy of swept sphere volumes that bound the point cloud. This hierarchy is the basis for [52]'s broad-phase collision detection which localizes the area of interest to that surrounding the haptic probe. At the local level, and in narrow-phase collision detection, points neighboring the haptic probe are found by checking if they are bound by a small sphere centered at the probe. Then, using moving-least squares, an implicit surface is built at run-time from the neighborhood of points. The closest point on this surface to the haptic probe is found and should the distance between the two points fall below a certain threshold or should the probe penetrate the surface, collision is detected. The penetration distance (i.e. the distance between the probe and the nearest surface point) is then used to determine the penalty force response felt via the haptic device.

This approach has several drawbacks, the most significant of which is that it does not take into account constraint surfaces when rendering force and so is susceptible to produce

incorrect results in the scenarios mentioned in section 2.2.2.1. The approach in [52] is also susceptible to the surface reconstruction errors seen if naively reconstructing a point set in a local volume especially if the said volume is in a high curvature section of the model.

2.2.4. Control Algorithms

Control algorithms are a haptic rendering method's way of accommodating the hardware limitations of the haptic device while minimizing the effects of these accommodations on the realism of the user's experience. Simply put, control algorithms aim to minimize the difference between the force that should ideally be displayed, and that which actually is being felt by the user.

Haptic devices are limited by issues such as their weight, the operational boundaries of their mechanical components, their discrete time operation, and the sensitivity of their sensors. A haptic device's weight is a force felt by the user even when no force should be rendered and so is a hindrance to the realism of the haptic experience (this is only the case in impedance control devices as will be described later in this section). The operational boundaries of haptic devices are also a hindrance in that they can only recreate a finite force which may be less than what is ideal. Even worse, if a user pushes a haptic device hard enough against a virtual wall, he or she might very well break it. The discrete-time nature of the virtual haptic experience is also a concern because humans sample the world continuously, not discretely, through their sense of touch. This continuity can be lost in the digital world despite the high execution rate of the haptic loop (1 kHz). Finally, quantization errors are inevitable because of the finite resolution of the device's positional sensors.

A haptic device is both an input and output device where energy flows to and from the human user. The previously discussed shortcomings in haptic rendering inherently imply an

"energy leak" in the haptic feedback loop [33]; the virtual interaction is simply different than the real life interaction. The energy leak is cause for instability in the feedback loop and this instability can manifest itself as the haptic device behaving unexpectedly (e.g. shaking, kicking back, etc). There are two methods in the literature that address the haptic loop's instability.

The first is virtual coupling [76] that, as the name implies, couples the haptic device's end-effector to a restricted virtual counterpart through stiffness and damping connection. Stiffness and damping are introduced virtually by the control algorithm to counter any instability agents. Of course they may also be expressed in the force response algorithms by the user as we have discussed in section 2.1.2.2. [33] notes that although this approach ensures stability, it doesn't allow for higher stiffness levels in the virtual model.

The second approach to counter the haptic loop's instability is more basic. [77] suggests that a hapto-visual application have four independent threads running at different rates: the first is the graphic rendering loop running at or around the standard 30 Hz rate; the second is the simulation engine thread running at a rate suitable to the application (the higher the dynamicity of the environment, the higher this thread's run rate); the third is a lower-rate, broad-phase collision detection algorithm that narrows down the area examined for collision detection to that which is in the vicinity of the haptic tool; finally, the fourth thread is the narrow-phase collision detection and force response algorithms thread, running as fast as possible, updating the haptic scene as often as possible. The main idea behind this approach is to create a virtual continuity to the actual discrete nature of the haptic feedback loop by way of updating the haptic scene as often as possible. Some architectures take this approach

one step further by assigning dedicated machines to run the haptic servo loops (e.g. FCS HapticMASTER [12]).

By definition, control algorithms are closely related to the kind of hardware being used, so we next review the two force control paradigms used by haptic devices today: impedance control and admittance control.

In impedance control, the device reacts to the change of its end-effector's position by recreating force, so the model is "displacement in, force out". Most stylus-based haptic devices (e.g. Phantom Desktop [78], Phantom Omni [79]) follow this paradigm. If the device is moving in free space without coming into contact with a virtual haptic object, the feedback motors are completely disengaged and the only force the user experiences is the haptic device's stylus' weight (in case of a stylus-based device) and whatever friction is felt from moving the stylus against the mechanical parts of the device. This however changes when collision is detected in which case the haptic device's motors engage appropriately and the user feels force feedback. This paradigm is shown in Figure 29.

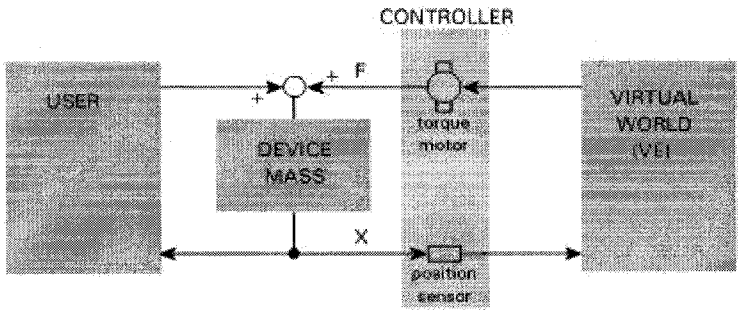

Figure 29: Impedance control paradigm [80].

In admittance control, the model is reversed. The device measures the force it experiences from the user and reacts by engaging its motors to move with certain velocity, acceleration, and of course transition; hence the paradigm is "force in, displacement out". The FCS

HapticMASTER [12] uses this paradigm. Free space is an issue for this force model since the slightest application of force would result in acceleration. On the other hand, much bigger force can be recreated since when collision is detected with a stiff surface, for instance, whatever the change of force relayed to the device is, no change in position occurs. This paradigm is shown in Figure 30.

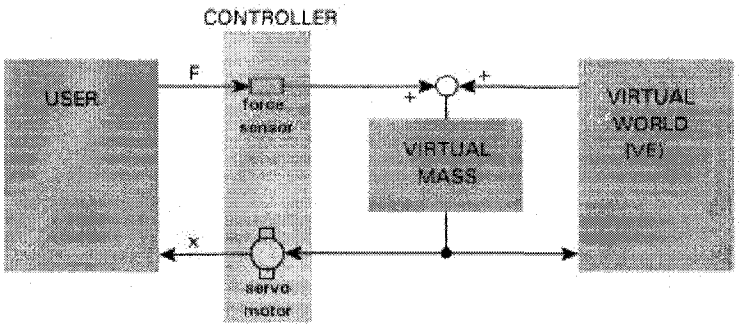

Figure 30: Admittance control paradigm [81].

Different haptic devices are better suited for different applications. The criteria to be considered for which device (and hence which force control paradigm) to use is presented in Table 5, but the rule of thumb for most applications is that for large force recreation, admittance control devices should be used, while for small-workspace, intricate applications, impedance control devices should be used.

Table 5: Impedance control vs. admittance control applications [82]

	Impedance	Admittance
Low mass	x	
Stable on physical surface	x	
Low costs	x	
Low friction	x	
Stable on virtual surface		x
Can simulate added mass		x
Crisp master-slave control		x
Robust device		x

2.3. Graphic Rendering of Point Clouds

Graphic rendering is a broad, deep, and well-researched field that is outside the scope of this work, however, as we have seen in this chapter, and specifically when discussing collision detection techniques, haptics often borrows from graphics but this is not always the case.

Haptic rendering is different than graphic rendering in that it is more local; you can only *feel* specific parts of a model (one part if using a single 3-DOF haptic device) but you typically *see* much more of the same model on, say, a computer screen. So, when graphically rendering a point cloud, surface reconstruction is a necessity for non-trivial point clouds; the gaps between the surface points must be filled in order to visualize a continuous surface. [83] surveys different point-based graphics rendering techniques showing that virtually all are, in one way or another, surface reconstruction algorithms; a class of algorithms we have already covered in section 2.1.4. The one relevant group of surface reconstruction approaches featured prominently in [83] but not mentioned in section 2.1.4 is the splats-based class of algorithms.

Splats basically are elliptical discs that are centered at surface points in the point cloud and variably sized and oriented to bridge the gaps between the said points thereby approximating the said point cloud's surface. Perhaps the most widely-cited splat-based rendering approach in the literature is QSplat which was first presented in [84].

[84] proposes the use of a multi-resolution bounding-sphere hierarchy to represent the point cloud model. This hierarchy is traversed over the course of graphic rendering in a breadth-first manner, progressively displaying the point cloud in finer detail as deeper levels in the hierarchy are processed. Once a leaf node in the hierarchy is reached or once the benefit of recursing further down a branch is deemed too low, a splat is drawn on the screen in the appropriate location.

In a preprocessing step, QSplat uses a special technique to clean up the raw point cloud that resulted from scanning a real-life model. The cleaned up point cloud is then fed to a marching cubes algorithm in order to create a triangular mesh which in turn is used to build the bounding spheres hierarchy mentioned above. This is all done in preparation for graphics rendering.

Mainly because QSplat creates a triangular mesh from the point cloud before rendering it to the screen, it is of limited use to us in our endeavor to find a way around recreating polygonal meshes before haptically rendering a point cloud. There are some lessons we can learn, however, from QSplat in storing and retrieving dense point clouds. Besides using the abovementioned level-of-detail-oriented bounding sphere hierarchy to store the model's vertices, QSplat uses quantization to efficiently encode positional, splat-radius, normal, and color information. It also uses a specific format to save point cloud information on disk and in memory in an effort to optimize loading times and minimize space requirements. How these lessons are applied to point clouds in the context of haptics is left for future work as reiterated in Chapter 5.

Chapter 3. Haptically Rendering Highly-Detailed Point Clouds

Motivated by the emergence of point clouds as a model representation of choice in 3D scanning and virtualization applications, and motivated by the lack of solid haptic rendering techniques for use directly with point clouds, we set on a course to develop a haptic rendering algorithm that meets all the goals we have set forth in section 1.5.

3.1. Collision Detection

When dealing with point clouds, the only direct information we have of the surface is that the points in the cloud are points on the modeled object's surface; there is no explicit description of the surface itself. But it is the surface of the model that is the focus of any collision query, so we must find a way to "fill the voids" between the points in the point cloud to create some semblance of a surface for the purposes of collision detection. In this section, we propose a way to approximate the point cloud's surface cheaply and effectively, and then move to describe a collision detection algorithm based on our surface approximation.

3.1.1. Defining a Point Cloud's Surface

If it were possible to store all vertices (surface points) that made up a model's surface, then collision detection between a haptic probe and the model would be a matter of searching for the probe's coordinates in whatever structure held the model's vertices (we assume for the moment continuous motion on part of the haptic probe). Of course it is not possible to store the infinite number of vertices that make up the model's surface (not explicitly anyway), but even if we could, there are limitations on the positional resolutions of haptic devices, as well as the precision of floating number representations in computers. So, practically speaking, if

we were able to produce a large and dense (but finite) data set of model vertices that were separated by distances less than or equal to a given haptic device's positional resolution, we could perform collision detection by searching for the probe's coordinates in the model vertices' data set. Such a collision detection query would be 100% accurate by definition, and depending on how the vertices are stored, it could also be fast.

Enter point clouds resulting from 3D scanning of real-life objects, which are in essence a large and dense, but finite, data set of model surface points that are separated by very small distances (see Table 2). Granted that such distances could be larger than haptic hardware's positional resolutions[1] but highly detailed models are a prime candidate for the accurate approach to collision detection explained above. Provisions must be made, however, to account for the gaps between the vertices that are separated by a distance greater than the resolution of a haptic device.

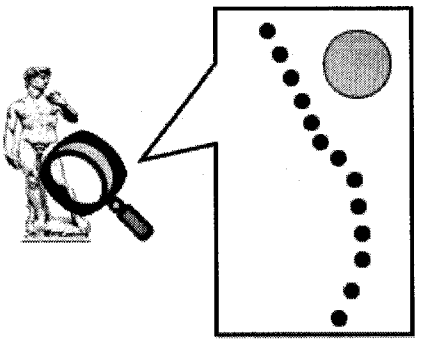

Figure 31: If the haptic device (grey circle) has a positional resolution worse than that surface points' (black dots) separation, then the surface represented by these points will feel continuous.

Figure 31 shows our approach's motivation graphically. The Michelangelo statue was scanned in the course of the work in [5] producing a large and dense data set that is made up

[1] Positional resolution of the: Phantom Omni is 0.055mm [50]; the Phantom Desktop is 0.023mm [51]; and the FCS HapticMASTER is 0.004 mm [52].

of the model surface vertices. If the separation between the vertices is less than the haptic device's positional resolution, as is the case in the figure, the device perceives the surface as continuous. However, we have not come across any models that had a resolution better than a haptic device's typical positional resolution, and so the gaps in the point cloud persist.

We propose to fill these gaps with very small, but necessarily touching if not overlapping, axes-aligned bounding boxes. Collision detection then becomes a matter of answering the question: does the haptic probe fall inside any of these AABBs? Point-in-AABB and segment-AABB intersection tests (which we will be providing motivation for using later in this section) are cheap and fast, as discussed in section 2.2.1, and that is a main reason why we choose AABBs in our implementation over other bounding volumes. Another reason is that a cube, which is a special case of an AABB and also the AABB of choice for the implementation of our CD algorithm as will be discussed later, shares the title with the sphere for the smallest bounding volume in terms of memory footprint; it only requires 4 variables to be represented: a length, an X-coordinate, a Y-coordinate, and a Z-coordinate.

3.1.1.1. Choosing the Right Size for the AABBs

As you might have already realized from the description of our approach, the accuracy of our CD algorithm is highly dependent on the density of the point cloud that is the model; the more vertices there are in a unit volume, the smaller the AABBs need to be to stay touching or overlapping, and the more accurate the CD result is. Uniformity in vertex density is also desirable because it makes implementing our algorithm simpler by allowing us to use AABBs of the same size as will be explained later. However, and speaking strictly from a performance point of view, uniformity is not as important as having a worst-case limit on the

density of the point cloud since point density, varying locally or uniform globally, is what determines the size of our AABBs and therefore the accuracy of our CD algorithm.

False positives reported by our CD algorithm can result if the AABBs that are centered at the model vertices are too large. False negatives can result if the AABBs are too small or non-overlapping. Both cases are shown in Figure 32 for point-in-AABB intersection tests. The figure shows the haptic probe (indicated by a grey dot) colliding with the object's surface at position 1 but since it is outside any of the AABBs, no collision is detected (false negative) while in position 2 it is inside an AABB but not colliding with the surface although a collision is falsely detected (false positive).

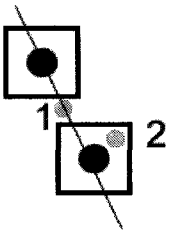

Figure 32: A false negative result in position 1, and a false positive one in position 2.

The right size for the AABBs is highly dependent on the separation between neighboring model vertices and so setting an AABB size for each vertex helps ensure that no gaps are present in our approximation of the model surface. Per-vertex AABB sizing also addresses non-uniformly distributed point clouds since it only concerns itself with three neighboring points at a time, which is the number of points it takes to build an AABB. The downside however, is that each point must know its neighbor, and that is information not explicitly available in a point cloud, and so we must deduce in a preprocessing phase before running the collision detection algorithm. Another downside is that each point in the point cloud must store an extra 3 floating point numbers representing the lengths of its AABB along the

three axes, which means an extra 12 to 24 bytes of memory per point in the point cloud. Figure 33a shows a 2D surface and how a per-vertex AABB sizing approximates it with no gaps.

An alternative to the above approach is to intelligently guess a rule-of-thumb size that we assign to all the AABBs in the model. This means that we do not require the deduction of neighborhood information pre-processing, nor are we adding any mentionable memory overhead. The downside to this approach is that, for an arbitrary point cloud, we cannot guarantee that our surface approximation will be gap-free as Figure 33b demonstrates.

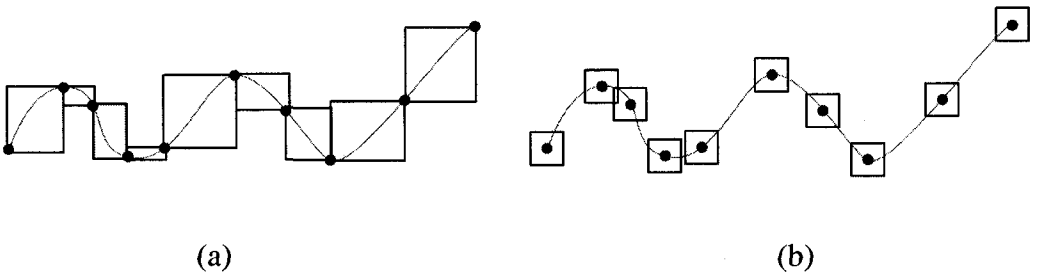

Figure 33: (a) Setting an AABB size per model surface point creates a gap-free surface, but requires knowing each vertex's neighbors. (b) Setting a rule-of-thumb AABB size is simpler to implement but can leave some gaps.

For the purposes of our work in CD, specifically that we are performing CD between a haptic probe and a highly-detailed point cloud with a resolution of 2.5 mm at worst, we hypothesize that rule-of-thumb AABB sizing will, by definition of the input point cloud, produce a gap free surface if the AABB was a cube with length equal to the worst local resolution in the point cloud. But even if this restriction was not put on the input point cloud, the gap seen in Figure 33b is unlikely to occur in a point cloud that results from a 3D scan unless it actually was a gap, or it was a surface that the 3D scanner missed. We say this because the nature of 3D scanning is too deliberate and highly detailed (even on surfaces that

are not uniformly sampled) for there to be this relatively large distance between two presumably neighboring points.

As we will discuss in section 3.2 when presenting our force response algorithm, we will need surface information anyway, so the preprocessing overhead of deducing neighborhood information will be incurred regardless. Despite this, we proceed in our implementation with the one-size-fits-all AABB which is guaranteed, by definition of our context, to produce gap-free surface approximations. This rule-of-thumb AABB sizing approach is easier to implement and has cheaper memory requirements (only the length of the AABB cube which is stored globally once per model).

From this point on in this work, when we refer to a vertex AABB or a point AABB, we mean the AABB centered at a given surface point in the point cloud. For the sake of clarity, we note that the AABBs are never built, nor are they stored anywhere. They are simply implied when performing collision detection.

3.1.1.2. Segment-AABB Collision Detection

Up to this point, we have implied that the problem of detecting collision between a haptic probe and a point cloud is a point-in-AABB intersection test, but this approach is theoretically susceptible to temporal aliasing (see Figure 19).

[50] states that a stylus-based haptic device typically is moved at speeds ranging from 1 mm/sec to 150 mm/sec. Also, [50] states that during contact with a model (haptic exploration), the stylus's speed drops at the high end to about 10 – 50 mm/sec. This means that at a sampling rate of 1 kHz, the discrete locations the haptic device is sampled at are 0.001 mm – 0.15 mm apart for non-contact movement, and 0.01 mm – 0.05 mm apart for haptic exploration. Keeping in mind that the vertex AABBs which we use for collision

detection typically range in size from 0.1 – 2.5 mm depending on the resolution of the point cloud, it becomes apparent that for the higher resolution models, temporal aliasing becomes a practical issue for the faster stylus speeds.

For this reason and also for more accurate force response calculations (discussed in section 3.2), we extend the point-AABB intersection test to a segment-AABB intersection test and define the problem of detecting collision between a haptic probe and a point cloud as a segment-AABB intersection test where the aforementioned segment is the line starting at the haptic probe's position in the previous haptic frame and ending at its position in the current haptic frame.

A segment-AABB intersection test is a simple and cheap operation that is based on the standard separating-axes test presented in section 2.2.1. Our implementation is copied from the highly efficient approach outlined in [8].

3.1.1.3. Vertex AABBs as Surface Approximations

What we essentially do in using AABBs to approximate a point cloud's surface is replace the infinitesimally thin surface of the point cloud with a collection of small cubes. One might argue that this collection of cubes, even if very small, effectively results in the smooth point cloud surface being replaced with a jagged one, which will also feel jagged, for the purpose of collision detection. Figure 34 demonstrates this point. Furthermore, while this approximation may be novel and cheap in terms of performing CD tests, it is inaccurate in those very terms; by sizing the AABBs to guarantee that there are no gaps in the model's surface we eliminate false negatives, but all we have done to address false positives is to demand a small AABB which reduces the possibility of a false positive occurring, but does not eliminate it.

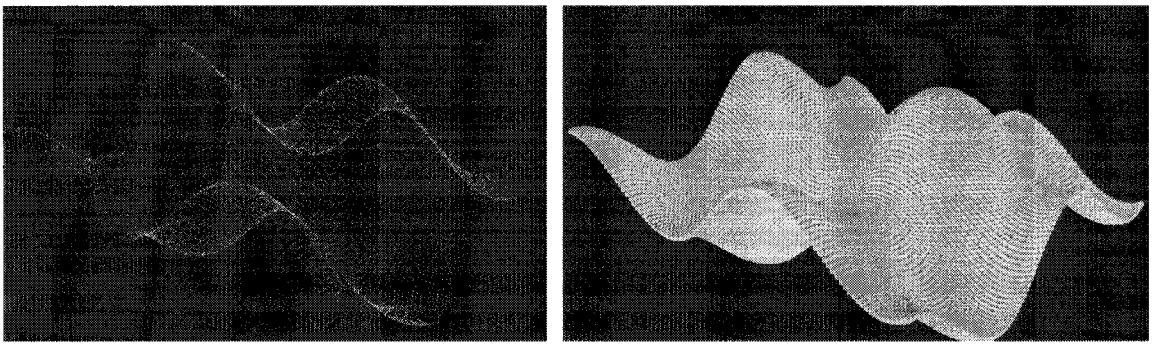

Figure 34: A point cloud is viewed by our collision detection algorithm as a cloud of small, slightly overlapping AABBs.

Our approach successfully renders point clouds haptically without that "jagged" feeling and without explicitly addressing false positives. This is because of how our force response algorithm operates (which will be discussed later in section 3.2). For now, suffice it to say that even if our CD algorithm detects collision, this does not necessarily mean that a response force will be rendered or that the rendered response force will reflect the jaggedness of the neighboring AABBs.

3.1.2. Spatial Partitioning

No matter how fast our segment-AABB test is – which incidentally is the narrow phase of our CD algorithm – the real bottleneck in any CD algorithm is finding the primitives to feed the intersection test functions. In our case, these primitives are: (1) the AABBs against which we test the (2) haptic path segment. The AABBs are centered at the model surface points (i.e. 3D coordinates), and the path segment is defined by the position of the haptic probe in the previous haptic frame and its position in the current one (i.e. 2 sets of 3D coordinates). So whichever spatial partitioning technique we use must be geared towards quickly retrieving point data.

Also, since one of our stated goals in section 1.5 is to support deformable models, our partitioning method must continue to provide speedy access to the model space despite the model's geometry changing.

Note that spatial partitioning constitutes the broad phase of our collision detection approach.

3.1.2.1. Modified Octrees

As explained in section 2.1, octrees are an efficient and widely used partitioning method for 3D objects that afford their user logarithmic access times to internal nodes ($O(log(n)$ where n is the number of primitives in the tree). On the surface, logarithmic access time may seem desirable for our purposes where n is typically in the hundreds-of-thousands or millions, but faster constant access time is even more desirable given the strict 1 ms ceiling we must stay underneath when preparing each haptic frame.

```
//A typical octree node data structure
Struct OctreeNode {
    float center[3];        //coordinates of the center point
    float halfWidth;        //the half-width of the node volume
    Node* ptrChildren[8];   //pointers to the node's 8 children
    Object* ptrObjList;     //point to a linked list of objects
                            //(dependent on implementation)
};
```

Figure 35: A typical octree node.

For the sake of further discussing the suitability of octrees, we note that their logarithmic access time comes at the expense of extra memory needed to store the structure of the octree in addition to the primitives it holds. An octree node, the pseudocode of which is shown in Figure 35, stores three floating point numbers for the coordinates of the cell's center, a fourth floating point number for the length of the half-width of the cell, eight pointers to the node's children, and, depending on implementation, a pointer to a linked list of objects stored in this node. To use a complete octree (which is a best-case-scenario octree in terms

of depth, and so access speed), to store a point cloud with n = 2,097,152 vertices, one will need to instantiate $\sum_{i=0}^{\log_8 n} 8^i = \sum_{i=0}^{7} 8^i = 2{,}392{,}064$ octree nodes, which is considerable overhead.

In an attempt to reduce this overhead, we modified the octree so that each node held up to 10 surface points. This will cost us extra time in the narrow phase of our CD, but it can considerably save memory; now our example octree containing n = 2,097,152 vertices only needs 247,172 node instances in the best case scenario.

Worse case scenarios are realized when an octree's memory requirements become prohibitive and access times slower. In case of models that are heavily clustered with vertices in regions distant from each other, and very sparse in the remaining regions, the whole octree's memory footprint can become quite large and access times in the clustered regions can suffer due to the deeper tree branching in those regions. 3D point clouds are such models; they are in essence made up of vertices that are highly cohesive in 2D (imagine crushing a coke can bringing all the can's surface point close to each other), but much more dispersed in 3D.

But the extra memory requirement is not the major disadvantage of using octrees. Besides non-constant access times, the cost of updating the tree during object deformation is the other main disadvantage; when the point cloud deforms, the octree's structure has to change to reflect the new partitioning. This may require only portions of the tree to be rebuilt, but if the deformation affects enough of the model space, then the whole tree might have to be rebuilt. This is simply too expensive to do at runtime.

Finally, we mention the issue of accessing a given point's neighborhood, which is relevant when discussing force response later in section 3.2. A point in an octree has immediate neighbors in its node and in the nodes adjacent to it. How these nodes are determined

depends on what the cutoff distance is for a point to be considered a neighbor or not, but we keep in mind that two octree nodes could be adjacent to each other in the model space but only share a distant grandparent in the octree structure. Taking advantage of the high density of vertices in the point cloud, we make the observation that examining octants which contain points a fixed distance from a given vertex is enough to retrieve all vertices in the vicinity of our point of interest. We propose an eight-point-test that we use to find neighboring octants and, thereby, neighboring points. The eight points mentioned can be thought of as the eight vertices of a cube centered at our point of interest. The half-diagonal of this cube is the distance cutoff for what is considered a neighbor and what is not. We query the octree to find the octants containing the eight vertices and then look in those octants to find neighboring points (i.e. points that fall under a certain distance threshold).

3.1.2.2. 3D Quadtrees

Going back to our analogy in the previous section about crushing a coke can to bring its surface points together in 2D as opposed to having them dispersed in 3D, we experimented with collapsing our 3D models into 2.5D models (X, Y, and a height value representing the Z coordinate) and storing them in a quadtree, a 3D quadtree if you will. The said quadtree would have most of the surface points clustered together. This is advantageous because it means that the tree depth is limited and so absolute access times are better (as is the case with octrees, and as mentioned in section 2.1.1, a quadtree has logarithmic complexity). This also means that less quadtree nodes need to be instantiated, and so memory requirements are smaller, which is to be expected also because each node stores pointers to four children instead of eight.

Naturally, there are disadvantages to using this 3D quadtree. First, given the high density that we expect our point clouds to have, collapsing the models into 2D space means significantly more surface points per quadtree cell, which means a larger number of pair-wise narrow-phase CD tests. If we reduce the size of the individual quadtree cells in an effort to limit the number of pair-wise tests, then we increase the number of quadtree cells necessary to cover the model, and so the memory requirements of the quadtree.

This tradeoff between memory footprint and structure-dependent speed (i.e. access time to specific nodes/cells and number of pair-wise tests once we get to those nodes) is unavoidable no matter the partitioning strategy we use, but it has been our practical experience that for the same dense 3D point cloud which we care most to access individual points in, octrees will have the bigger memory footprint, but will provide the faster access time to a given point in the cloud.

As is the case with octrees, we note that our 3D quadtrees also require costly updating (if not rebuilding) in case of model deformation. However, because the quadtree is only in 2D and the model is in 3D, deformation along the 3rd axis is inconsequential to the structure of the quadtree (but of course not the affected points contained in the quadtree) and so, generally speaking, these 2D structures are more tolerant to deformation than their 3D cousins.

Finally, and in regards to addressing neighboring quadrants (and so neighboring vertices), the eight-point-test, presented when talking about octrees, becomes the four-point-test, when talking about quadtrees. The four points are the vertices of the square centered at the given point the neighbors of which we are interested in.

3.1.2.3. Uniform Spatial Partitioning

Speaking in general terms, all non-uniform partitioning techniques are costly to update at runtime, and updating the partitions at runtime is what we have to do in order to support general model deformation. So we are forced to consider uniform partitioning, but very much to our delight.

Uniform spatial partitioning (USP) techniques are simple, fast, and effective. They involve laying a grid in the model space with equally sized cells, and so each cell will host its share of vertices; some will host no points at all, while other will host many. The fact that the grid has equal sized cells means that each cell is easily addressable, which means that it is quickly accessible. To find the index of a uniform cell along a certain axis given a coordinate in that axis, we simply divide the coordinate by the cell length (assuming the grid is anchored at the point of origin). So, as opposed to traversing trees in logarithmic time like we did with quadtrees and octrees, reaching a specific cell in a uniform spatial partition is done in constant time, $O(1)$, by performing the three division operations necessary to determine the cells' indices along the X, Y, and Z axes. This is, as all constant time algorithms are, done at a cost irrespective of the number of points in the model, the density of the model's point cloud, or the size of the model space. Furthermore, determining neighboring cells is trivial and is only a matter of incrementing or decrementing the cell's indices as desired.

The downside of using uniform grids is that although we can access each USP cell in constant time, we cannot predict how many points will be in that cell and so actually accessing a specific point can theoretically take linear time. We do not, however, expect this to be the case in reality (nor does this manifest itself in our experimental results) for the simple reason that the points that are being put in the USP grid are surface points. Surface

points, when examined locally, are dense in two dimensions only (representing the surface of the model). A model's surface, to the scanner, is infinitesimally thin and so the distribution of the surface points is "spread out". This means that we can expect a good distribution of points in the USP grid without having to set the grid cells' size to an unreasonably small value.

Another downside of using uniform spatial partitioning is the memory footprint such a data structure demands. A straightforward USP implementation would instantiate every USP cell in the grid, even the ones that contain no surface points, and so there is considerable waste in memory resources. A smarter and more complex implementation that uses dynamic memory and pointers can help reduce this overhead.

Finally, we argue that a USP approach is much better suited to accommodate deforming objects since deformation of the contained model has no bearing on the structure of the partitioning grid. There is, however, one case which requires a change in the structure of the USP grid in response to deformation, and that is if the object deforms beyond the boundaries of the USP. Imagine the tightest AABB possible around a given model. Typically, it is that AABB that is partitioned uniformly to create the uniform grid. Now, if the contained model deforms in such a way that parts of it escape the tightest fitting AABB, then these parts have now left the USP grid as well. This scenario could be addressed in a two ways without putting restrictions on the application: At the cost of extra memory, we could increase the dimensions of the USP pre-runtime, or; at the cost of extra computation and with some clever array and pointer manipulation, we could dynamically grow the USP at runtime to accommodate the spilling outside of the grid's original boundaries.

It is for the reasons above that we proceed in our approach using USP for the purpose of broad-phase CD.

3.1.3. Algorithm

Armed with constant time spatial addressing via uniform spatial partitioning and a speedy segment-AABB pair-wise intersection test, we present our CD algorithm in Figure 36.

```
1  //Define the segment from the position of the haptic probe in the
2  //previous haptic frame to its position in this haptic frame.
3  Segment oldHipThisHip;
4  oldHipThisHip.Set(oldHipPos, thisHipPos);
5
6  //Get the list of USP cells crossed by the given segment.
7  ListOfCells listOfCells;
8  listOfCells = GetCellsCrossedBy(oldHipThisHip);
9
10 if (listOfCells != NULL) { //If the segment crosses at least one USP
11
12      //Get the list of vertex AABBs (inside the previously compiled
13      //list of USP cells) that are crossed by the given segment.
14      ListOfVertices listOfVertices;
15      listOfVertices = GetVertexAABBsCrossedBy(oldHipThisHip);
16
17      if (listOfVertices != NULL) { //If the segment crosses at least
18                                    //one vertex AABB
19          collisionDetected = true; //Collision detected
20      }
21      else {                        //If no vertex AABBs are crossed
22          collisionDetected = false;//No collision is detected
23      }
24 }
25 else { //If the segment does not cross any USPs
26      collisionDetected = false; //No collision is detected
27 }
28
```

Figure 36: Our collision detection algorithm.

The broad phase of our CD algorithm constitutes of checking to see if the segment (line 4) from the old haptic probe position (also known as the haptic interaction point or HIP) to the current haptic probe position crosses any of the USP cells. Not only do we care for a Boolean answer to this question, but we also want to know which cells were crossed so we compile a list of the aforementioned cells (line 8).

If at least one cell was crossed (i.e. we detected collision in the broad phase of our algorithm), we proceed to narrow phase CD. Otherwise, no collision is possible (line 27).

The narrow phase of our CD algorithm constitutes of checking to see if inside any of the USP cells we listed in line 8, exists at least one vertex AABB that is also crossed by the segment `oldHipThisHip`. Again, this time for the purposes of our force response algorithm presented in section 3.2, we compile a list (line 15) of the intersected USP components, which this time around are model vertices whose AABBs have been crossed.

The implementation details of our algorithm will perhaps not serve to advance the reader's understanding of our approach, but we do mention some notes:

- If the HIP has not moved since the last haptic frame (i.e. displacement in all three dimensions is less than some small value ε) then the segment collapses to a point and both the broad phase and narrow phase CD tests become point-in-AABB tests.
- To find the different USPs crossed by the given segment, we divide up the segment into smaller ones so that each sub-segment's starting point is in a distinct USP cell (if the segment is long enough). We then determine the USPs crossed by calculating the indices of the USP cells containing each sub-segment's starting point. This guarantees that we get the minimum list of USP cells crossed.
- As a rule of thumb, the USP cell in our implementation is a cube with a length equal to five times the point cloud's worst resolution. Through trial and error, we found that this balance between the number of USP cells in the grid and the resulting number of points per cell produces the best results.

The experimental results of our algorithm are presented in section 3.6.

3.1.4. Limitations

We have already mentioned the question of false positives in section 3.1.1 and promised to address it in section 3.2. There is, however, a fringe case that theoretically is a limitation of our algorithm, but in practice it is a non-issue.

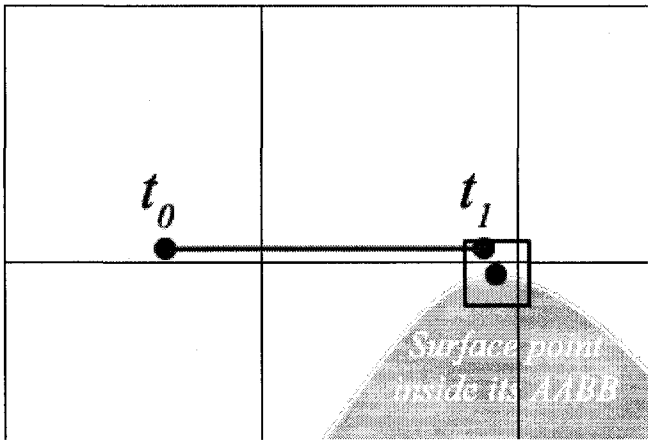

Figure 37: A missed collision (false positive).

We turn the reader's attention to Figure 37 where you can see that the segment crosses an AABB in a cell determined to be of no interest to our CD algorithm (the only cells we are interested in are the ones that contain the segment). While this theoretically is a missed collision, it has no bearing on the user's experience. The reason we say this is because of the minute distance that we are missing here even relative to the size of the AABB (which itself is no more than 2.5 mm at its largest). Even when we were trying to recreate this error in practice, we were hard pressed to position the haptic probe just right to feel the effect on force response.

For the sake of completeness, we suggest two methods to counteract this limitation despite its inconsequentiality. The first method increases the number of the USP cells examined in narrow phase CD by searching the adjacent cells to the ones listed in line 15 of Figure 36 for

vertex AABBs crossed by the given segment. This, of course, is done at the cost of increasing the number of pair-wise tests carried out.

The second approach we suggest is to substitute the segment-AABB test with a swept sphere-AABB test, where the haptic probe point is substituted with a sphere, and so the segment representing the haptic probe's path is substituted with a swept sphere volume. This ensures that all points close to the haptic probe's path are examined, however at the cost of the slower, more complex swept sphere-AABB test.

Both solutions fail the return-on-investment test.

3.1.5. Discussion

In this section, we discuss our CD algorithm in light of the design guidelines we set forth in section 2.2.1.1.

- Internal Object Representation and Environmental Simulation Parameters: Our algorithm is designed to work with highly-detailed point-based models of real-life objects that are free to move around (translate and rotate) as well as deform in response to either haptic or some other environmental stimulus. Our CD algorithm places no restriction on the number of models in the haptic scene; its scope of operation is strictly and blindly local to the haptic path segment and is unaffected by any other considerations.
- Memory Requirements: Haptic rendering generally favors speed over memory economy. This is because of the unforgiving 1 kHz required update rate which leaves only 1 ms for each haptic frame to be processed from beginning to end. Our algorithm places no specific demand on broad phase collision detection (which requires the extra data structure and therefore the extra memory) other than high enough speed not to contribute adversely to the haptic loop slowing to a rate worse than 1 kHz. Out of the different

spatial partitioning techniques we experimented with, we chose to use uniform grids which although theoretically have a big memory footprint, in practice this footprint can be reduced using clever implementation. Uniform grids are the fastest storage structure in terms of access speeds.

- Collision Detection Algorithm Specialization, Types of Queries, and Demands on Accuracy: As mentioned, our algorithm is geared for best operation with high-density point clouds, but also, under the title of specialization, it is also specifically designed to feed the force response algorithm which comes next in the haptic rendering pipeline. The information passed on down the aforementioned pipeline is the collection of vertices whose AABBs have been crossed by the haptic probe segment. This specialization not only speaks to the needs of the force response algorithm, but also allows us some leeway which, in our case, is enough to discount inaccuracies in our CD algorithm especially when it comes to false positives, as will be discussed in section 3.6.
- Speed: The speed requirements of haptic loops are unforgiving; the loop must finish processing in under 1 ms. [8] states that collision detection is the "most computationally demanding" aspect of any rendering cycle, be it haptic or graphic. Experimental results presented in section 3.6 show that our CD algorithm, as part of the complete haptic rendering pipeline presented in this thesis, are exceptionally fast well besting the 1 ms mark.
- Pragmatic Considerations: The development time of our CD algorithm was lengthy and the process was quite involved, however the reusability and utility justifies the efforts. We have come up with a provably fast and sufficiently accurate collision detection

algorithm for use with dense point clouds in the context of haptically rendering the said point clouds.

3.2. Force response

Next in the haptic rendering pipeline after collision detection is the response to the said collision, should it have occurred, which comes in the form of a force vector displayed on the haptic device.

Recall from section 2.2.2 that the predominant force response paradigm in haptics is the spring paradigm which models the response force as a spring anchored at a computed point on the model's surface being elongated a distance equal to that from the HIP's position to the aforementioned spring's anchor point. The deeper inside a model the haptic probe is, the more elongated the conceptual spring is and, so, the greater the force magnitude resisting the penetration. The key in this approach is where to place the spring anchor on the model surface. As discussed in our literature review, this question is answered by [70] and [71] which are the standard approaches used nowadays in force response computations. Our force (or collision) response algorithm is an adaptation to point clouds of the constraint-based god object/proxy approaches in [70] and [71].

3.2.1. Defining a Point Cloud's Surface

Determining the location of the god object (GO), assuming a frictionless surface, is where the "constraint" in the constraint-based approach comes in; we must know what surface the haptic probe entered, and then examine that surface and its immediate neighbors, if necessary, to see which constrain the god object or, in other words, which require that the god object be on. After the constraint surfaces are determined, the location of the god object is computed to be the point closest to the current HIP on the appropriate constraint

surface(s). Figure 38 shows a simple example of force rendered in response to a HIP penetrating a constraint surface. At time t_0 the HIP is outside the model space and so no force response is necessary. At some time between t_0 and t_1, the HIP penetrates the model surface and a collision is detected (i.e. the segment crosses a vertex AABB). The constraint surface in this example is the only surface in the figure and the god object is placed at the location on that surface closest to where the HIP is at time t_1. The force rendered is the vector from the HIP to the god object.

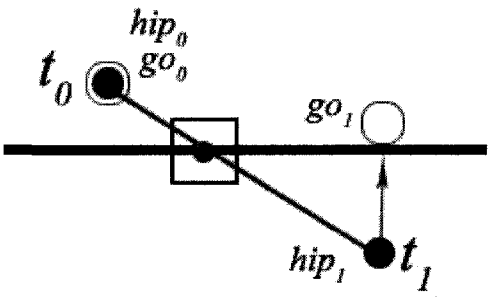

Figure 38: Force response.

It is evident from the example above that force response computations require surface information from the point cloud to determine the constraint surface.

AABBs worked with collision detection in approximating the cloud's surface but will not do in this case. This is because we do not only need to know which point is the collision point (i.e. where the haptic probe first entered the surface), but also which surfaces are the constraint surfaces (i.e. the surface(s) that sit under the first collision point and the later computed god object).

For the purposes of force response, we define the surface of a point cloud as the finite set of points S in the point cloud; a definition that may seem trivial but its significance lies in that it explicitly states that we will not interpolate surface points, bur rather only consider those that

already exist in the cloud. Inferring surface points is unnecessary because of the assumed high density of the point cloud. Furthermore, interpolating points in already high density models could be problematic because it can introduce a surface where there should be none, a common problem with point cloud surface reconstruction algorithms [1].

Going back to the example in Figure 38, if we are not interpolating surface points then the god object has to be a surface point in the point cloud, so we need to find a way to "search" in the point cloud for the final god object position given only the first collision point and the current HIP location. In other words, determining the god object's location becomes a constrained search problem. The least computationally expensive way to find the god object in the point cloud is to deduce neighborhood information and then follow a constrained path from the first collision point to the final god object location. The key words are "neighborhood information" and "constrained path", both will be discussed in sections 3.2.2 and 3.2.3 respectively.

3.2.2. Building Neighborhood Information

As we states in the previous section, our force response algorithm dictates that we know the neighbors of any given point on the model's surface so we need to deduce neighborhood information from the point cloud. This can be done any number of ways.

3.2.2.1. *k*-Nearest-Neighbor (kNN) Search

A simple approach is to examine each point in the point cloud and find the nearest k number of points to it. This approach is known as the *k-Nearest-Neighbor* (kNN) search. kNN search can be made to meet criteria like ensuring that the resulting neighborhood map is completely connected (by compiling the neighborhood information recursively rather than iteratively) or

that a point knows its nearest neighbor in some general direction (by partitioning the local space around the point and searching for the nearest neighbor in each partition).

kNN typically uses spatial partitioning to limit its search for the nearest neighbors to a small volume as opposed to naively searching all the points in the point cloud, which is a $O(n^2)$ endeavor. Worth noting here is that kNN search has the same vertex access time as any access operation performed on the spatial partitioning structure and, although it is susceptible to claiming nearby points as neighbors even if they are on a different surface, the risk of this error occurring decreases in proportion to the increase in the density of the point cloud. Adding a threshold related to the point cloud's density in the area being examined can also limit the occurrence of such an error; if a point lies too far from the current point being examined, it is excluded from the neighbor list even though it might be one of the nearest k points.

Note that kNN search dictates that the data structure holding the vertex contain space to store pointers to the k closest neighbors.

3.2.2.2. Fixed Distance Neighbor (FDN) Search

As opposed to finding a set number of nearest neighbors, *Fixed Distance Neighbor (FDN) search* finds all points within a fixed distance from the point of interest. This could be thought of as listing all the points that fall inside a sphere with radius r equal to the said fixed distance and centered at the point we are finding the neighbors for.

As opposed to kNN search, FDN search is local by definition. It also requires enough space in memory to store however many number of points fall within the fixed distance sphere. FDN search is more susceptible to mistakenly identify a neighbor point that actually is on a

different surface than the point of interest especially in highly dense models, so for this reason, it is less suitable to use in building our neighborhood information.

3.2.2.3. Voronoi Partitioning

Recall in section 2.2.1 discussing Voronoi diagrams (an example of which is shown in 2D in Figure 28 on page 68). Voronoi partitioning, a technique widely used in nearest-distance queries, has been used in the context of point cloud rendering in [51] producing neighborhood information tables but at the cost of significant preprocessing.

Suffice it to note here that the Voronoi method is a good candidate for integration into our work but we have not implemented it since it has already been researched in the context of haptic rendering in the work cited.

3.2.2.4. 3D Scanner Path

We present an approach to deducing neighborhood information in point clouds based on our knowledge of (or perhaps the ability to determine) the path a laser scanner or contact head takes when scanning an object; if we know the path the scanner takes as it moves about the model, then we can deduce which scanned points are neighbors.

To illustrate this point, consider the simple example of a laser scanning a flat surface at a horizontal resolution h samples/axis and proceeding linearly (i.e. scans a line, returns, scans another line, returns, etc). If each point scanned is indexed serially, then the point to the top of a given point x has index $x - h$, the point to its right has index $x + 1$, the point to its left has index $x - 1$, and the point beneath it has index $x + h$. Boundary limits of course apply. With some imagination on the reader's part, it should be easy to apply this logic to any kind of systematic path a 3D scanner takes when moving about an object.

This method is by far the cheapest in terms of building neighborhood information since that information is already available. Not only that, but it also is, by definition, the most accurate since the model is described by the motion of the scanner (in 2 of the 3 coordinates) and the scanning depth (in the 3rd coordinate).

The disadvantages of using this method, however, are that it is inapplicable to models scanned by hand and models processed post-scanning by software that strips away vertex indices or timestamps.

3.2.2.5. Modified kNN

In examining our different neighbor-search options, we decided to employ, if possible, our proposed approach of using a priori knowledge of the scanner path, or if this knowledge is unavailable, employ a modified kNN search algorithm.

Our version of kNN search finds the 8 nearest neighbors (if possible) of a given point by iteratively and uniformly partitioning the space around that point until one of two conditions are met: either the sum total of all points in the partition containing the point of interest and its immediate spatial neighbors is at most nine (eight neighbors plus the point of interest); or the partition's dimensions are larger than some preset value.

In regards to our iterative and uniform partitioning, we note that it is independent of the USP structure used to store the point cloud (but uses that structure to access points), is cube-based, and is centered at the point we are interested in finding the neighbors of (i.e. our point of interest). What this flavor of partitioning allows us to do is to approximate distances between points in the point cloud and our point of interest based on the inclusion of the former in different partitions. For example and when dealing with dense point clouds, we can say with high confidence that points that fall in the same partition as that centered

around our point of interest are closer to the said point than other points that fall in neighboring partitions. We cannot be certain of that because we are comparing distances by inclusion in cubes (not spheres) centered at a point, but for dense point clouds, the results have been shown experimentally to be suitable for our purposes (see section 3.6). In essence, this approach to distance approximation allows us to compare distances between neighboring points without explicitly calculating these distances, which is a higher cost practice.

Our partitioning terminates, as mentioned earlier, if one of two conditions are realized. The first occurs when the partitioning results in the cell that contains the point of interest, (either individually or in combination with the cells immediately adjacent to it) containing at most nine points. What this would mean is that the eight nearest points to our point of interest have been found. The second terminating condition is realized when the partitioning procedure has to grow the cell to a size larger than a preset value in order to find the eight nearest neighbors. This would occur only if the eight nearest neighbors are too far from the point of interest, in which case we only report as neighbors the points (if any) already found in the smaller partitioning.

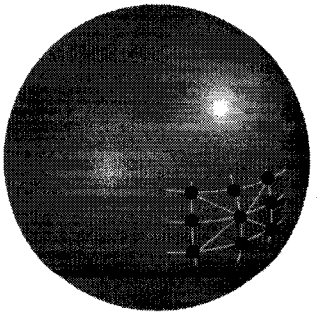

Figure 39: Ideally, we would want each point to have access to its eight closest neighbors on the 3D surface of the object in the general directions north, south, east, west, north-east, north-west, south-east, and south-west.

For a somewhat uniformly distributed dense point cloud (which typically results from scanner software preprocessing), we expect the eight nearest neighbors of a given surface point to appear as they do in Figure 39.

3.2.3. Defining the God Object

As previously defined, the god object is a point on the surface of the haptic model that represents where the haptic probe should be if we were able to stop it from penetrating the conceptually impenetrable haptic model. Also, as discussed in section 3.2.1, our force response algorithm puts a limitation on the god object requiring that it be an already existing surface point since our algorithm does not interpolate surface points. Also, as discussed, we do not have to interpolate surface points because of the high density of the point cloud. For the reader who foresees force discontinuities even in high density point clouds, suffice it to say for now that our force shading technique (presented in section 3.4) ensures smooth force throughout the model space.

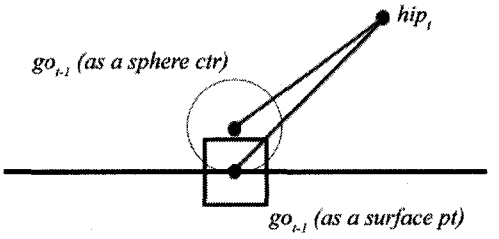

Figure 40: The god object as a surface point vs. a sphere. As a sphere, it overcomes the false positive CD result.

To the above description of the god object, we add one more detail; the god object can be, in some scenarios as will be discussed in section 3.2.4, a sphere that shares a surface point with the point cloud. The radius of the god object sphere is set to the half-diagonal of the vertex AABB plus a small margin of error (ε). This is done to address the false positive collision detection results when such results may adversely affect the final decision whether the

current haptic frame is a collision frame or not. Figure 40 shows how a god object sphere overcomes false positive CD.

3.2.4. Algorithm

Given the above definition of a point cloud's surface and the god object, we turn the reader's attention to Figure 41 which shows how our force response algorithm works with a variety of cases. We will explain our algorithm by example before outlining it in pseudocode form.

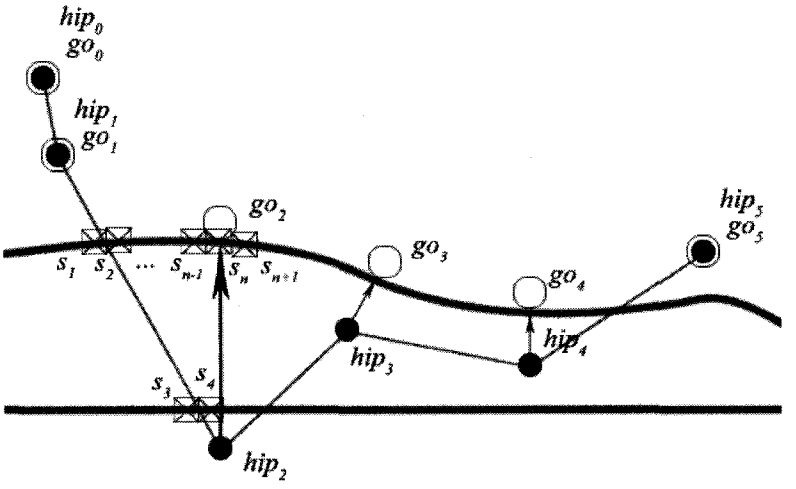

Figure 41: Different cases and how they are handled by our algorithm. Note that there is no explicit model surface in a point cloud (the surface is actually a collection of discrete points), but we draw the surface here for clarity's sake.

At time t_0, the application starts with hip_0 in free space collocated with go_0. At time t_1, the HIP has moved to hip_1 but the segment $hip_0 \rightarrow hip_1$ has not collided with the surface, so we set go_1 to hip_1. The segment $hip_1 \rightarrow hip_2$ however does collide with two of the model's surfaces at points s_1, s_2, s_3, and s_4 (remember that we define collision with the surface as the intersection of the haptic path over the past haptic frame with one or more AABBs centered around surface points – the crossed squares in the figure). In this case, the constraint surface should be the upper one so we find which one of the collision points (s_1, s_2, s_3, and s_4) is

closest to hip_1. In the diagram, s_1 is deemed closest, so the surface on which s_1 resides becomes the constraint surface. The next step is to find iteratively the surface point neighboring s_1 that is closest to hip_2. Noting that $dist(s_1, hip_2) > dist(s_2, hip_2) > dist(s_{n-1}, hip_2) > dist(s_n, hip_2)$ and that $dist(s_n, hip_2) < dist(s_{n+1}, hip_2)$, s_n, the local minimum, is named the god object at time t_2 (go_2). The force response is the vector from $hip_2 \rightarrow go_2$. Note that the god object is actually a sphere with a radius that clears it from its surface point's AABB. This is to avoid false positive collision detection when the god object leaves the surface sphere as will be explained at time t_5. At time t_3, the segment $hip_2 \rightarrow hip_3$ intersects a model surface, but there obviously still is collision, so in the case where the previous haptic frame witnesses a collision, we check the segment $hip_t \rightarrow go_{t-1}$. At time t_3, the latter also collides, so collision is still true, and the god object becomes the local minimum in the neighborhood of go_{t-1} as calculated at time t_2. At time t_4, we note that the previous haptic frame witnessed collision, and with that in mind we check the segment $hip_3 \rightarrow hip_4$ to find that there is no collision, so we deem that this also is a collision frame and find go_4 in the neighborhood of go_3. At time t_4, the HIP exits the model space; our algorithm determines that by checking the segment $hip_4 \rightarrow hip_5$ to see that it has collided with the model surface and then checks the segment $hip_5 \rightarrow sphere\text{-}go_4$ and finds that there is no collision. Note that if we checked $hip_5 \rightarrow go_4$ we would have detected collision because go_4 falls inside an AABB, while $sphere\text{-}go_4$ is outside the AABB. The coincidence of these conditions leads our algorithm to declare the haptic frame at t_5 a collision free frame, so go_5 is set to hip_5.

In recap, we present our full algorithm at a high-level in Figure 42. Note that in some instances we check the god object sphere, while in others we check the god object surface point.

```
IF (collision detected in previous frame)
      IF (hip_t→hip_t-1 collides with a surface)
            IF (hip_t→sphere-go_t-1 collides with a surface)
                  Collision detected in this frame
                  Starting at go_t-1 find local minimum distance s→hip_t
                  go_t = s
                  Render force vector hip_t→go_t
            ELSE
                  No collision in this frame
                  go_t = hip_t
            END IF
      ELSE
            Collision detected in this frame
            Starting at go_t-1 find local minimum distance s→hip_t
            go_t = s
            Render force vector hip_t→go_t
      END IF
ELSE
      IF (hip_t→hip_t-1 collides with a surface)
            Collision detected in this frame
            s = nearest collision point to hip_t-1
            Starting at s find local minimum distance s'→hip_t
            go_t = s'
            Render force vector hip_t→go_t
      ELSE
            No collision detected in this frame
            go_t = hip_t
      END IF
END IF
```

Figure 42: High-level pseudocode describing our force response algorithm.

3.2.5. Limitations

One limitation we are aware of in our algorithm results from how we restrict the god object's location to only the locations of the vertices in the point cloud. Whether it is examined as a point or a sphere (sharing a point with the point cloud), the god object's location cannot be arbitrary. This restriction, unaddressed, can cause force discontinuities as demonstrated in Figure 43. Note how hip_0, hip_1, and hip_2 all experience force vectors ending at the left surface point in the figure, which is the nearest surface point to them on the constraint surface. Once the middle threshold line (dashed in the figure) is crossed, the nearest surface point becomes the one on the right and so considerable force discontinuity is felt when moving from hip_2 to hip_3.

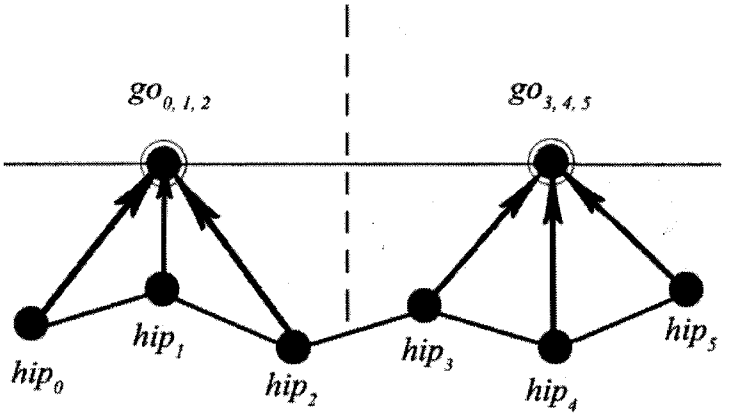

Figure 43: Force discontinuities resulting from restricting the god object's location to only the vertices in the point cloud.

Once a threshold like the one in the figure is crossed, the force vector's direction could potentially change abruptly, resulting in force discontinuity. The potential for this effect being felt is closely related to the distance between neighboring vertices (i.e. the cloud's resolution). In our implementation, we have found that the worst possible point cloud resolution that does not produce this effect is 0.2 mm; point clouds with worse resolutions exhibit this effect.

Since the problem is that the god object cannot be an arbitrary value and given that we do not interpolate surface points, only two courses of action are left. The first is to interpolate the force vector, a practice referred to as force shading (presented in section 2.2.2.2 and applied to point clouds in section 3.4). And the other is force filtering which is universally applicable regardless of any environment parameters since it places restrictions on the current frame's force vector in relation to what the force vector in the previous frame looked like.

Another limitation that we are aware of is much less likely to occur and, as was the case with the special scenario discussed in section 3.1.4, we were hard press to manually recreate it

even when we tried. In special cases when the HIP is hovering above the actual point cloud surface a distance less than the AABB half-length (so distances less than 1.25 mm), a false positive collision result could trigger our algorithm to render force incorrectly. This scenario is shown in Figure 44. Sometime between times t_0 and t_1, the segment representing the HIP path collided with the AABB bounding the surface point shown. This false positive collision result triggers the force response algorithm to render a force vector from the location of the HIP at time t_1 to the nearest surface point neighboring the collision one. This of course is incorrect.

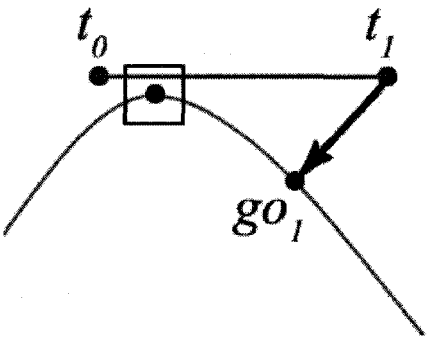

Figure 44: Incorrect force rendering.

Finally, we mention a limitation that also is unlikely, but not impossible, to occur in concave regions of low resolution point clouds. Recall that the resolution of the point cloud is what determines the size of the AABB so the worse the resolution, the bigger the AABB. Now consider Figure 45. The point hip_1 escapes the surface of the point cloud, but our algorithm does not realize this since the segment $hip_1 \rightarrow go\text{-}sphere_0$ is in collision with a vertex AABB that is close to go_0, and so continues to render force. To escape this scenario, the user would have to move the HIP along the surface to the right away from the larger AABBs on the top surface and then pull the HIP outside the model.

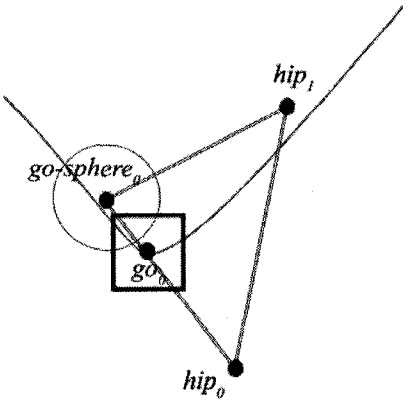

Figure 45: Incorrect force rendering.

3.2.6. Discussion

Our force response algorithm is an adaptation to point clouds of the standard constraint-based approaches to rendering force in haptic scenes and, as such, addresses the special cases that [70] and [71] were introduced to handle; cases like thin objects, concave models, and multi-surface constraining. As a matter of fact, a major advantage of using point clouds over polygonal meshes is that, for the purpose of determining the constraint surfaces in concave models, there is no need to split the model mesh; a path on the surface of the point cloud is travelled looking to minimize a Euclidean distance. Also, this approach is much simpler than, and as equally effective as, other minimization techniques such as Lagrange Multipliers (as used in [70]) or robot path planning (as used in [71]).

Furthermore, and as will be discussed in the next sections, our algorithm affords the developer the ability to render surface properties such as friction, deformability, and texture just by manipulating the position of the god object as proposed by [71].

In terms of memory requirements, our algorithm does not introduce any memory overhead beyond what is required by the CD algorithm. Our approach to FR does not cache any force

responses simply because, even in worst case runtimes as will be discussed in section 3.6, it runs fast enough not to justify the extra memory cost or trouble.

Our force response algorithm is also inherently accurate to the extent allowed by the resolution of the point cloud since the god object cannot be arbitrary. And in cases when the resolution is too low for this restriction not to manifest itself in the form of force discontinuities, force shading (as will be discussed in section 3.4) does allow us to create interpolated force vectors even though the god objects remain latched to the point cloud.

As went the argument when discussing our CD algorithm, the same argument goes for our FR approach supporting multiple dynamic objects. To calculate force response, our algorithm is blind to the global environment only seeing the constraint surface (via neighborhood information) and the locations of the HIPs and the god object(s).

3.3. Support for Dynamic Models

A hapto-visual virtual object can undergo rotation, translation, and deformation in response to some stimulus in its environment. In this section, we discuss how our algorithm supports these different scenarios.

3.3.1. Euclidean Transformations

Euclidean transformations (rotation, translation, and reflection) do not change the size or shape of the model, only its location in 3D space therefore the model remains unchanged in its local space. Simple keeping a 4x4 homogeneous coordinate matrix that describes the model's transformations (although reflection is not really applicable in hapto-visual virtual environments) allows us to move the HIP from the world's coordinate system to the model's local coordinate system at which point our CD and FR algorithms apply as is. There are two penalties for using this approach: The first is the time it takes to update the transformation

matrix when the model undergoes rotation or translation; and the second is the time it takes to perform matrix multiplication when converting the HIP into local model coordinate space and then the force vector to the world coordinate space. Both penalties are a series of basic arithmetic operations and are therefore very soft.

3.3.2. Deformation

Deformation is different from Euclidean transformations in that the model actually changes shape during the application's run-time. In this section, we differentiate between two types of deformation: The first is deformation occurring independently of haptic interaction; and the second is deformation occurring as a result of haptic interaction. We then present how our algorithm supports the latter type of deformation.

3.3.2.1. Types of Deformation

Neither our collision detection nor our force response algorithms cache model information, but they do take as an input the collision state from the previous haptic frame. This makes our algorithms unsuitable for applications where the models undergo deformation independent of haptic interaction. Consider a scenario where in a given haptic frame, the HIP is in free space, but in the next haptic frame, and without the HIP moving at all, it ends up inside the model due to the model's surface deformation. Our algorithm will detect no collision with the haptic surface and so it will continue to designate the HIP, incorrectly, as a point in free space.

On the other hand, in the case of deformation that occurs due to haptic interaction, our algorithms apply. This is because the only piece of information carried over from the previous haptic frame (the previous collision state) is accessible only to our algorithm which is, in this case, the only cause for deformation. And since all other model information is

retrieved as needed each haptic frame, there is no danger of missing the deformation's effects on the model.

3.3.2.2. Algorithm

As discussed in section 2.2.2.3, there are two ways to model deformation haptically: actually changing the model's geometry; or manipulating the location of the god object to simulate deformation.

Our algorithms work equally well with both approaches. For when the model geometry actually changes, our algorithm applies as is since it is not responsible for its input point cloud but rather the deformation algorithm is. Special care should be taken though to make sure that the contact points detected in the CD block do not deform before the FR block renders its force. This simply is a problem of synchronizing the haptic rendering loop with whichever loop the deformation algorithm resides in to ensure that they both are seeing the same model at all times.

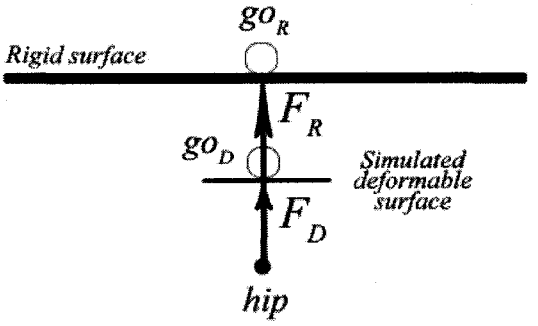

Figure 46: Simulating haptic deformation by way of changing the location of the god object from go_r to go_d.

The second approach, where the model's geometry stays the same but the god object's position is manipulated, requires changes to our FR algorithm. Figure 46 shows that at time t, the god object is found as if the model had a rigid surface (go_R). The force vector ($\mathbf{F}_R$) is

also computed as if the model had a rigid surface. Deformation is then simulated by scaling down $\mathbf{F}_R$ to reflect the virtual deformed surface; this scaling is done by a factor that is dependent on the stiffness at point go_R. Finally, we have $\mathbf{F}_D$; a force vector from the HIP location at time t to the simulated god object go_D.

3.3.2.3. Limitations

Being an adaptation of the method in [71], our simulated deformation algorithm is susceptible to the same instability that can be caused by simply manipulating the location of the god object as opposed to following the deformed geometry of the model. When the god object moves from a deformable surface to a rigid one, the force magnitude can change abruptly depending on how deep inside the deformable surface the HIP is. If the god object suddenly jumps to a new location (the surface of the rigid part of the model) from a distant one (the simulated deformed surface of the model) then an unrealistic and high-energy-gain force is inappropriately rendered.

3.4. Force Shading

As explained in section 2.2.2.2, and in the context of polygon-based haptic rendering, force shading is a practice that aims at smoothing the discontinuities felt during a haptic interaction should the force vector change abruptly from the previous haptic frame to the current one as a result of a change in the geometry of the model being felt.

For highly detailed point clouds with a resolution less than 0.2 mm, our rendering algorithm does not need to do explicit force shading; this is because the force vectors rendered are dependent on the individual points in the point cloud, and not the surface of the model, so there is a much smaller likelihood of a sudden change in force vectors, which is the reason

why force shading is implemented in the first place. For models with a resolution worse than 0.2 mm, force shading becomes necessary as discussed in section 3.2.5.

Recall from our presentation of the workings of our force response algorithm that the god objects we designate when drawing force vectors must be actual points on the point cloud; we do not interpolate surface points. So for point clouds where the resolution is worse than 0.2 mm, the force vectors rendered will change abruptly when the HIP crosses the imaginary threshold where the nearest surface point to it (the god object) changes as we saw in Figure 43. This will incorrectly feel to the user as a surface texture resembling ridges separated by a distance equal to the distance between the god object in the previous frame and the one in the current frame. On the other hand, when the surface points are less than 0.2 mm apart, this re-designation of the god object and the resulting force vector change is unperceivable by the application's user (as our experimental results show in section 3.6).

3.4.1. Algorithm

We propose to counter this imperfection with a force shading algorithm specific to point clouds that borrows from shading in polygonal-based haptics and graphics. Based on the weighted average approach proposed by Gouraud shading (presented in section 2.2.2.2) for graphics and adapted to polygonal haptics in [9], our force rendering algorithm first finds the second and third nearest surface points to the HIP position that are immediate neighbors of the god object (note that each vertex knows its neighbors). At this point, the shading algorithm in [9] would proceed to compute the normal (and by extension the force vector) of the projection of the HIP onto the said triangle by computing a weighted average of the normals at each of the triangle's vertices. In our approach, normal information is never

computed explicitly so we cannot use this method as is. The modification that we implement is described as follows:

$$F_{hip} = \frac{\sum_{i=1}^{3} A_i \cdot F_i}{\sum_{i=1}^{3} A_i}$$

Where $\mathbf{F}_{hip}$ is the force rendered, $\mathbf{F}_1$, $\mathbf{F}_2$, and $\mathbf{F}_3$ are the force vectors from the hip to the three vertices *go*, *go-nbr*$_1$, and *go-nbr*$_2$ respectively, and A_i is the area of the triangle opposite *go*, *go-nbr*$_1$, and *go-nbr*$_2$ respectively. Figure 47 demonstrates our approach.

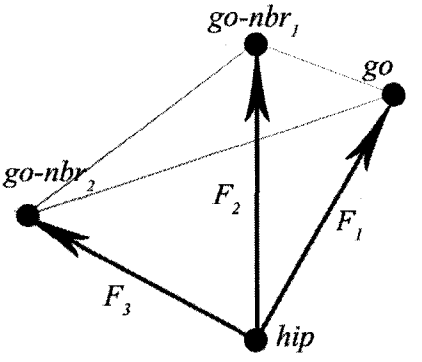

Figure 47: Force shading in point clouds.

We have also experimented with a second approach that computes another weighted average but the weight is not a vertex's opposite triangle's area as was the case above, but rather the magnitude of the force vector from the hip to a given vertex as described by:

$$F_{hip} = \frac{\sum_{i=1}^{3} (L - |F_i|).F_i}{L}, \text{ where } L = \sum_{i=1}^{3} |F_i|$$

This approach will only work for rigid surfaces because the deeper the HIP penetrates the object surface, the less difference in length there is between the three force vectors whose magnitudes we are using as a weight, and so the more central the god object becomes to the

triangle joining the three vertices regardless of the location of the HIP, which defeats the whole purpose of force shading.

We have been able to counteract this effect with a bit of a clever trick. If we place the haptic surface near the bottom of the haptic device's workspace and ensure that the surface is stiff then we do not run the risk of penetrating the model too much and the length-weighted force shading performs well.

3.4.2. Limitations

The force shading algorithm in [9] that we have adapted for use with point clouds runs the risk of "rounding off" edges and corners when actually the force discontinuity necessary to feel an edge or a corner is desirable. This point has been first brought up by [71] to which [9] responds that this could be averted if there is enough discontinuity in the vector magnitude to covey the haptics of an edge or corner. In our experience, and also mentioning the role that graphics often play in "selling" haptic sensations to the user, [9]'s argument is valid.

As for our length-weighted adaptation, we already mentioned that its shading performance degrades the deeper the HIP is inside the haptic model, which is a major limitation when rendering deformable surfaces.

3.5. Haptic Surface Properties

Besides deformation, discussed in section 3.3.2, which technically is a surface property, we discuss in this section texture and friction.

3.5.1. Texture

In section 2.2.2.3, we noted that the current practice in both haptic and graphic polygonal rendering is to separate texture from geometry. This is done because texture information is too geometrically complex to be processed by the rendering pipelines while still maintaining

the performance goals set forth for haptic and graphic rendering. Not only that, but following this approach is unnecessary because using height fields, for instance, to represent texture is good enough for the final rendered model to appear (graphically) or feel (haptically) textured.

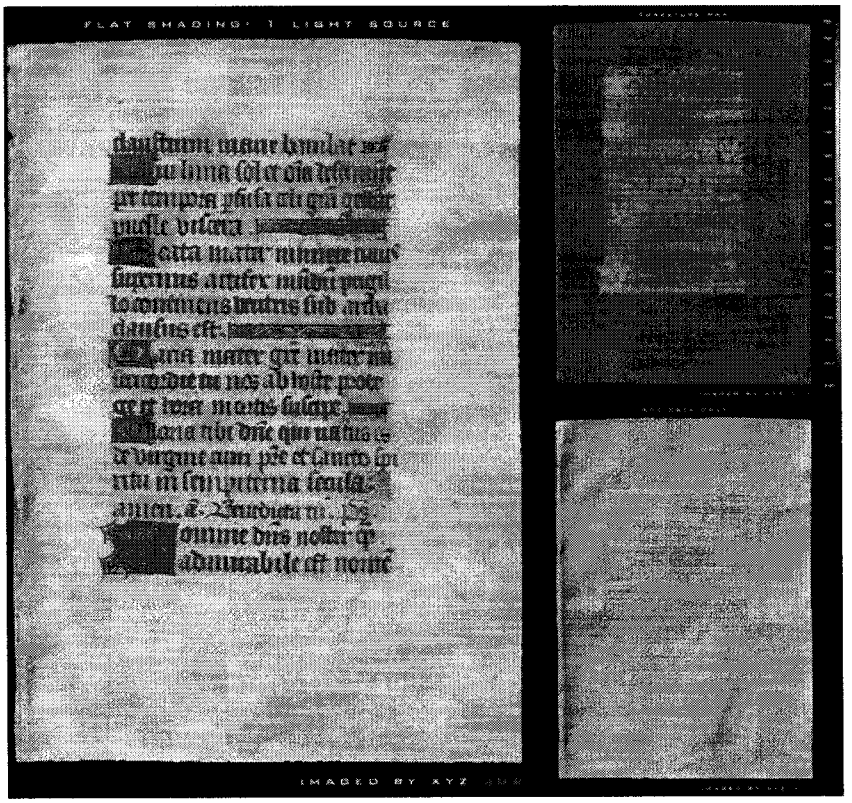

Figure 48: A vellum manuscript scanned at a resolution of 0.1 mm. The top right image is the curvature map of the manuscript while the bottom right image is the manuscript's height data. (Images from the XYZ-RGB website).

Point cloud models are different altogether; their texture is inherent in the cloud and there is simply no underlying geometry. Not only that, but in essence height fields, which are used to represent texture in polygonal models, are a form of point clouds; an X and Y coordinate with a height value (i.e. 2.5D space) is an instance of an X, Y, and Z coordinate (i.e. 3D space). This provides motivation to our work in Chapter 4. Furthermore, highly-detailed 3D model scanning is by design the main method with which real-life textures are digitized;

consider for example the Mona Lisa's height map (shown in Figure 5) or the vellum manuscript's height map (shown in Figure 48).

For all the above reasons, we say that our approach, which is by design a highly-detailed model rendering technique, is capable of rendering haptic texture.

This is not to say that we cannot introduce simulated texture. As was the case with simulated deformation, manipulating the location of the god object can also render different textures (first proposed by [71]). The simplest example we present is rendering lower resolution models (worse than 0.2 mm resolutions) without force shading using our approach. As was discussed in section 3.2.5, this creates force discontinuities that are inconsistent with smooth surfaces but these same force discontinuities can represent ridges on a non-smooth surface. Another haptic effect that we experimented with, this time by perturbing the force vectors rendered to some degree, is the feeling of a rough surface. Of course the force perturbation has to be within the limits of stable haptic rendering, but just as we were able to create the feeling of ridges by simply changing the location of the god object, we were able to create a rough surface by perturbing the force vectors right before they were sent to the haptic device for displaying.

In recap, if the texture that we want to render is inherent in the point cloud, then our algorithm conveys it as is. Otherwise, if the texture is simulated by means of perturbing the force vectors or manipulating the god object, then our algorithm also support this approach using the same mechanism we employed while simulating deformation.

3.5.2. Friction

As you may recall from our discussion of friction in section 2.2.2.3, it is basically a force vector that opposes the surface movement of the haptic probe once the latter is in contact

with the model. Section 2.2.2.3 also mentions [71] and [85] and how they propose a shortcut to render friction simply by modifying the location of the god object on the surface of the model rendered subject to some restrictions.

We apply friction to the point cloud surface using the virtual adhesion point suggested for polygonal haptic rendering by [9]. After our force response algorithm computes the god object location (which is where the god object should be if the haptic surface was frictionless), we move that location back along the surface of the model towards where the god object was in the previous haptic frame effectively rendering friction. The distance we roll back the god object by is dependent of course on the haptic surface friction coefficient which is either defined per surface point in case the model has different friction coefficients for different areas, or for the whole model, if the whole model shares the same friction coefficient. In case of the former, the friction coefficient that we consider is the one god object's in the previous haptic frame. Figure 49 demonstrates how we render haptic friction.

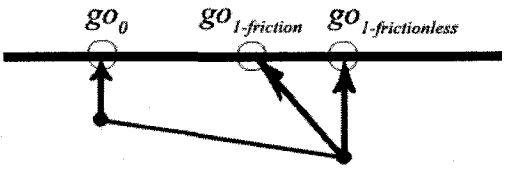

Figure 49: Rendering haptic friction in a point cloud.

3.6. Experimental Results

In this section we will present the experimental results we have collected as we tested our algorithms. Universal to all the listed experiments are the development tools and the testing setup. We used Microsoft Visual C++ 7.1 to code our algorithms, and ran the tests on an Intel Dual Xeon machine (2.8 GHz) with 2 GB of RAM. For the haptic device, we used the

SensAble Omni, and for the haptic API we used SensAble's Open Haptics Toolkit. Visualization was done using OpenSceneGraph and PlyTools. Our code was timed using Microsoft's high resolution QueryPerformanceCounter timer.

3.6.1. Neighborhood Building Using our Modified kNN Approach

As discussed in section 2.1.4, the preprocessing times that go into reconstructing a polygonal mesh from an unorganized point cloud are significant. Since we do not explicitly reconstruct model surfaces and given that our algorithms only rely on neighborhood information as mentioned in section 3.2.2, we expect the preprocessing times for our algorithms to be shorter.

In case of deducing neighborhood information from the path of the 3D scanner, we do not expect any preprocessing time overhead that is worth mentioning; the information is available to us either implicitly (via a function that describes that path of the scanner and so the relation between the different points) or explicitly (via a look-up table of sorts).

In using our modified kNN approach outlined in 3.2.2.5, the preprocessing times we report are shown in Table 6. When compared with the results presented in Table 3 on page 33, our preprocessing times compare very favorably, again because we only deduce neighborhood information and do so without explicit distance calculations.

Table 6: Preprocessing times required by our modified kNN algorithm

Method	Model	Reported time (mins)	Test machine
Modified kNN	Stanford dragon (~435K points)	~1.5 mins	2-CPU 1.6GHz Turion64 processor system with 2GB RAM
	Sinusoidal surface (1M+ points)	~3.36 mins	2-CPU 1.6GHz Turion64 processor system with 2GB RAM

Table 6 makes mention of a sinusoidal surface. This surface, which will be used as a test model in section 3.6.2 and is shown in Figure 50Figure 1, has been generated automatically to simulate a priori knowledge of the linear path of an assumed scanner (i.e. neighborhood information can be determined from the indices of the points in the point cloud). The sinusoidal surface was fed to our modified kNN algorithm to test the latter's ability to correctly deduce neighborhood information. In comparing the neighborhood information resulting from our kNN algorithm against that deduced from the simulated scanner path information, we found the two to be an exact match.

3.6.2. Haptic Rendering Experiments

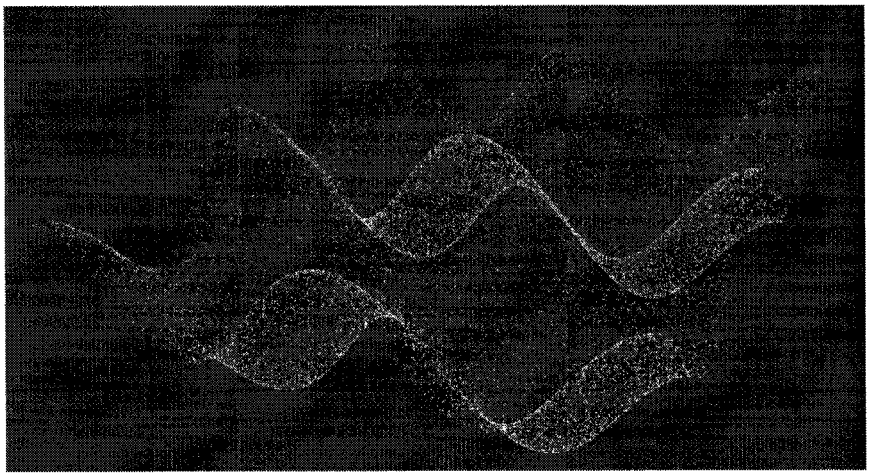

Figure 50: The point cloud for Test #1 (1,002,001 vertices, only 40,000 shown).

The first experiment consisted of loading the model in Figure 50 (without any surface properties) into a hapto visual environment. The goals of this experiment were to test the rendering algorithms basic functionalities and gauge its performance. Subjectively, we can say without qualification that the model was rendered as expected; in over a minute of inquisitive exploration, there were no missed collisions (i.e. no holes in the model), the force was continuous and smooth, and the haptic loop was stable. Our objective test results, shown

in Table 7 alongside the experiment's parameters, show the speed of our algorithm (slowest rendering time was only 0.2674 ms, well under the golden 1 ms mark).

Table 7: Information on test #1

Model Information	
Model description	Sinusoidal surface z = 10 * (cos(0.04*PI*x) + cos(0.04*PI*y))
Number of points	1,002,001 surface points
Resolution	0.1414 mm
Real-life dimensions	100 mm x 100 mm x 20 mm
Algorithm Parameters	
AABB size	0.15 mm
Neighborhood info	Deduced from provided simulated scanner path
Surface properties	None
Results (collected only when algorithm renders force)	
Collision test frames	54,717 haptic frames
Fastest rendering time	0.0221 ms
Slowest rendering time	0.2674 ms
Average rendering time	0.0374 ms

Our second test added surface properties to our first test model, namely deformation and friction. We split the model into four surface areas: the first was the frictionless surface in the first experiment; the second was a surface with a friction coefficient of 0.7; the third was a deformable surface that allowed the user to push down on the model and had a deformation coefficient of 0.5; and the fourth was a deformable region with friction and deformation coefficients for both properties equal to 0.5 and 0.7 respectively. We note here that deformation was achieved by manipulating the god object as discussed in section 3.3.2, and not by changing the model's geometry. As for the test results, we can say subjectively and without qualification that the model was rendered as expected; both the friction and deformable surfaces felt as desired. One interesting result is that, because we were not changing the structure of the model but only the location of the god object when rendering

deformation, we experienced a high energy gain when moving from the deformable surface to the non-deformable (rigid) ones. This is to be expected because the god object, while still in the deformable region of the model, was at a depth below the surface that was significantly lower than where it was placed when crossing over to the non-deformable regions. This sudden change in the magnitude of the force rendered was noticeable, manifesting itself in a "kickback" and degraded the haptic experience. This effect has been discussed in section 3.3.2.3. Table 8 shows the experiments' run-times which are not significantly slower than those shown in Table 7 despite the added deformation and friction computations.

Table 8: Information on test #2

Model Information	
Model description	Sinusoidal surface z = 10 * (cos(0.04*PI*x) + cos(0.04*PI*y))
Number of points	1,002,001 surface points
Resolution or density	0.1414 mm
Real-life dimensions	100 mm x 100 mm x 20 mm
Algorithm Parameters	
AABB size	0.15 mm
Neighborhood info	Deduced from provided simulated scanner path
Surface properties	One fourth of the surface area was rigid and frictionless, one fourth had friction (coefficient = 0.7), one fourth was deformable (coefficient = 0.5), and the last fourth was both with friction and deformable (coefficients = 0.7 and 0.5 respectively).
Results (collected only when algorithm renders force)	
Collision test frames	49,914 haptic frames
Fastest rendering time	0.0184 ms
Slowest rendering time	0.2680 ms
Average rendering time	0.0358 ms

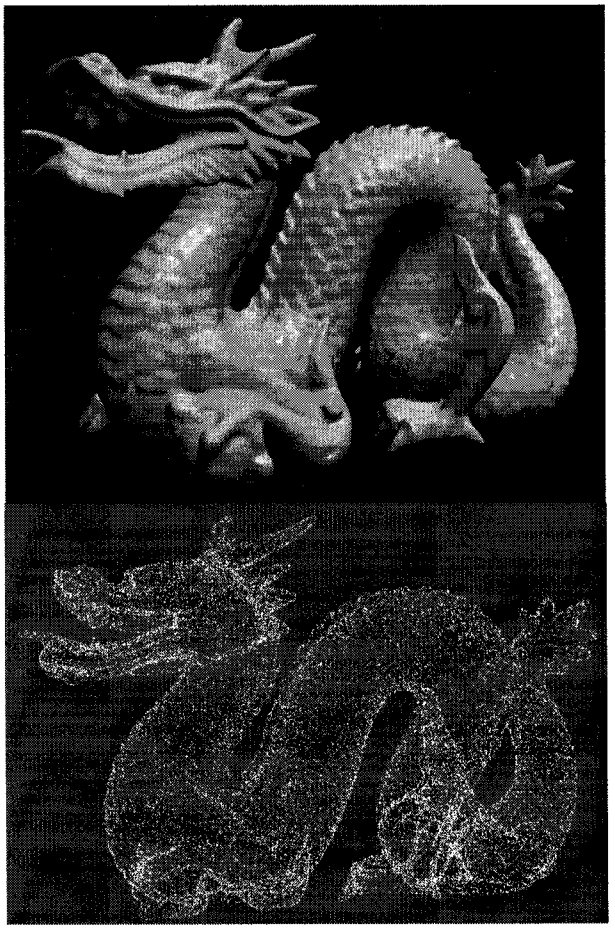

Figure 51: The Stanford Dragon. Top: Polygonal model. Bottom: Point cloud (437,645 vertices, only 40,000 shown).

Our third experiment was loading the model in Figure 51. The Stanford Dragon [26] is an approximately 200 mm x 150 mm x 100 mm model consisting of 437,645 points. Its resolution is not readily available and so we had to, after loading it from file and building the neighborhood information (using our *k*-Nearest-Neighbor method discussed in section X), set the AABB size to the worst resolution encountered which was 3.0 mm (the model was scaled from its original size before the test was carried out). Since the Stanford Dragon's resolution is significantly worse than the 0.2 mm threshold under which no force shading is necessary, we rendered the model using both our force shading methods: the triangle-area weighted method, and the length-weighted method (both presented in section 3.4). We also

ran the environment with no force shading to get a feel for the force discontinuities at a resolution of 3.0 mm.

The triangle-area-weighted method worked best; there were no perceivable force discontinuities, and edges throughout the model (e.g. the dragon's teeth) could easily be felt. There were some snags however. Most perceivable, around the bottom areas of the dragon's mane, there were regions where the haptic probe felt stuck even though visual inspection showed no obstacle to the probe's movement. Upon deeper inspection, we found that the reason for this inconsistency lied in an incorrect collision test result as discussed in section 3.2.5 (third issue listed). This is a direct result of the low resolution of the point cloud which results in larger AABBs. The extra computation time necessary for force shading, while not significant by any means, is apparent in the rendering times shown in when compared with the times in Table 7 and Table 8

Table 9: Information on test #3

Model Information	
Model description	The Stanford Dragon
Number of points	437,645
Resolution or density	Non-uniform. ≤3.0 mm.
Real-life dimensions	210 mm x 150 mm x 94 mm
Algorithm Parameters	
AABB size	3.0 mm
Neighborhood info	8-Nearest-Neighbors (kNN) as described in section 3.2.2.5
Surface properties	Triangle-weighted force shading.
Results (collected only when algorithm renders force)	
Collision test frames	116,013 haptic frames
Fastest rendering time	0.0265 ms
Slowest rendering time	0.3678 ms
Average rendering time	0.0634 ms

Length-weighted force shading did not work very well at all. Ridges were perceivable (although with less severity than with force shading) even when we stayed close to the model surface.

Finally, when running the algorithm with no shading at all, we felt texture artifacts throughout the model which actually simulate ridges that are 3 mm apart.

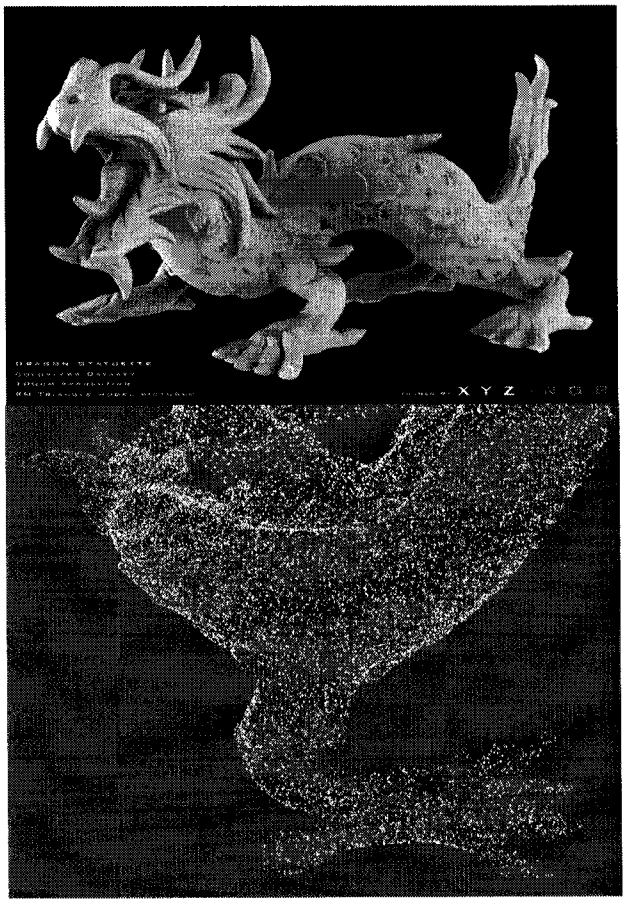

Figure 52: The XYZ-RGB Model. Top: The real-life model. Bottom: The point cloud around the dragon's left hind foot viewed from the bottom up. (3,609,600 vertices, only 40,000 shown).

The fourth test environment we ran was of a model from the XYZ-RGB archives, hosted online by Stanford University, also of a dragon (Figure 52). This time around, the resolution of the model was given (0.1 mm) so we had no problem determining the dimensions of the AABBs. This intricate and highly-detailed model was the largest we loaded containing

3,609,600 points. As you can see from Figure 52, the model is rich with texture and interesting areas that would challenge our algorithm. It rendered like a charm; truly with the really high resolution (0.1 mm), we were able to haptically explore the model and appreciate its many surface textures. Also, worth mentioning, is that we did not render the model with any force shading; it was not necessary for such a high resolution. Also, in regards to the "sticking" we felt in certain surface areas of the Stanford Dragon due to false positive CD, we were able to recreate this error in the XYZ-RGB Dragon, but we had to try very hard; even fairly involved exploration would not have encountered this problem.

Test results are shown in Table 10.

Table 10: Information on test #4

Model Information	
Model description	XYZ-RGB Dragon
Number of points	3,609,600
Resolution or density	0.1 mm
Real-life dimensions	~160 mm x ~85 mm x ~100 mm (scaled to fit inside Omni workspace)
Algorithm Parameters	
AABB size	0.1 mm
Neighborhood info	8-Nearest-Neighbors (kNN) as described in section 3.2.2.5
Surface properties	Inherent texture.
Results (collected only when algorithm renders force)	
Collision test frames	258,229 haptic frames
Fastest rendering time	0.0058 ms
Slowest rendering time	0.3650 ms
Average rendering time	0.0459 ms

The only knock on this test is that, when loaded, it consumes a good portion of memory. We hesitate to list a number here because our implementation and our coding techniques were by no means optimized to save memory, but the executable (which includes the model's

representation) occupied hundreds of megabytes of RAM. This overhead can very well be minimized using programming techniques that aim to economize memory usage, but as mentioned in Chapter 1, memory economy is not a stated concern of this work.

3.6.3. Comparison with Previous Work

As mentioned in section 2.2.3, we have only come across three works in our literature review that view models primarily as point clouds en route to haptically rendering them.

Table 11: Comparison between our approach and others found in the literature

	US patent [75]	V-GRAPH [51]	SSV BVH and run-time MLS [52]	Our approach
Preprocessing	Uniform spatial partitioning followed by decomposing polygonal mesh into point cloud	Build V-GRAPH using Voronoi diagramming and mesh information	Swept-sphere volumes put in bounding volume hierarchy	Uniform spatial partitioning followed by neighborhood information deduction
Preprocessing time	N/A	~5 mins for 1M+ points model	N/A	Negligible in case of provided scanner path information, ~1.5 mins for 435K point model, and ~3.36 mins for 1M+ point model
Broad-phase CD	Determine partition where HIP resides	Greedy visit along V-GRAPH leads to nearest surface point	Point-in-SSV-BVH	Check if segment crosses USP cells containing surface points
Narrow-phase CD	If distance between HIP and any surface point is below a given threshold, collision occurs	Segment-triangle intersection test against all triangles sharing vertex found in broad-phase CD	Build an MLS surface from all surface points within a sphere centered at HIP then perform query between HIP and MLS surface	Check if segment crosses AABBs centered at surface points retrieved in broad-phase CD

Force response	Penalty-based	(Presumably) Constraint-based	Penalty-based	Constraint-based
Reported metrics	N/A	CD < 0.07 ms for 1M+ point model	Haptic frame rendering in 0.2 ms for 24K model and 3.5ms for 543K model	See section 3.6.2

As Table 11 shows, all haptic rendering techniques (ours included) share a procedural division starting out with preprocessing which is done offline and the results of which are saved for later use when it is time to load the model in question. After preprocessing, collision detection in its broad and narrow phases follows, finally feeding the force response algorithm.

In this discussion, we will not address the US patent in [75] because it is very poorly described and so adds no value to our comparison. Discussion of [75] can be found in section 2.2.3.

In comparing the different preprocessing approaches, we have shown experimentally in section 3.6.1 that our approach has considerably faster preprocessing times than common surface reconstruction techniques. This is mainly due to the following features: first, we only deduce neighborhood information (as opposed to constructing full-fledged surfaces); second, we do so using a modified kNN search algorithm that is local and cheap yet accurate for dense point clouds; and third, we use uniform spatial partitioning which affords us constant time access to the USP cells containing our model's surface points. [51] also deduces neighborhood information (and nothing else) and so its preprocessing run-times are relatively fast (~5 mins for 1M+ point model) but this is where the similarities in

preprocessing stop. To build the V-GRAPH, [51] requires the availability of the model's triangular mesh and so it is not a pure point-cloud-based technique.

On the other hand, [52] is. The preprocessing in [52] is more elaborate than ours in terms of the data structure holding the model. Swept sphere volumes are built hierarchically around the model's surface points while in our approach, the surface points are simply stored in a uniform spatial partition. [52], however, does not deduce neighborhood information in the preprocessing phase and so we expect it to experience relatively short preprocessing times but no such times are reported in the literature. One point that should be made here is that having the model's surface points stored in a hierarchical structure by definition means that they are accessible in $O(\log n)$ time which may not be fast enough for haptic rendering in cases of large values for n. Although the authors of [52] make no such admission, but we suspect that the slower haptic rendering rates reported in their work are in no small part related to non-constant access times to small spatial partitions of the haptic model.

Collision detection in its two phases is approached differently by the different algorithms being compared. [51] does not use a bounding box approach at all in its broad-phase. Rather, it finds the nearest surface point to the HIP by traversing the V-GRAPH looking for a local minimum surface point. This surface point is then communicated to the narrow phase of the collision detection algorithm which, in turn, checks all surface triangles that share this point against the segment that represents the HIP's movement over the course of the current haptic frame. Should this triangle-segment intersection test return a collision result, then the current haptic frame is deemed to be a collision frame. This CD test is fast (reportedly running in under 0.07 ms for 1M+ point model) but assumes that the model's triangular mesh is available for examination.

The CD test in [52] is wholly different and more typical in that it uses bounding volumes in its broad phase. The algorithm searches for the HIP in the bounding volume hierarchy. If the HIP is found in a leaf node (i.e. it is close enough to a surface point or a group of surface points), then the broad phase invokes the narrow phase. During the course of the latter, a sphere is built around the HIP and the surface points that fall within the said sphere are used to build, at run-time, a surface using marching least squares. Once the surface is built, collision is tested for between the HIP and the MLS surface.

Our CD is different than the one in [51] in that it does not assume the availability of a triangular mesh, and different than the one in [52] in that it does not perform any surface reconstruction. Furthermore, because in both its phases it is a segment-AABB test, it is conceptually a simpler and faster test than the one in [51] and the one in [52]. It is, however and for the reasons outlined in section 3.1.4, less accurate than at least the approach in [51]. In section 3.1.4, we showed how our test is prone to produce false positives especially for less dense point clouds, but as also we have argued, these false positives are addressed in our force response algorithm. As for comparing the accuracy of our work with that in [52] no data is available in the latter to base a quantitative analysis on, but we do note that both approaches perform better as the point cloud grows denser; MLS produces better surfaces given more surface points and our CD algorithm uses smaller AABBs (and so is less prone to false positives) as the resolution of the point cloud improves.

In regards to force response, the availability of neighborhood (and so, constraint surface) information in both our work and [51] affords both the ability to implement the standard constraint-based approach to force rendering. This, however, is not the case for the approach

in [52] which is simply a penalty-based approach meaning that it is susceptible to incorrect force rendering in the special cases outlined in section 2.2.2.1.

From the above discussion, we see how our approach essentially is the only approach in the literature (as far as we know) that deals with the haptic rendering of point clouds and does so with a low-cost preprocessing phase, a fast collision detection phase, and a constraint-based force response phase. The inaccuracies in our CD test are largely negated by the implementation of our force response algorithm as demonstrated by our experimental results, our haptic rendering rates easily beat the 1 ms mark, and the quality of our haptic rendering is subjectively reported to be in line with other haptic rendering techniques. All these claims are substantiated in section 3.6.2.

3.6.4. Notes on our Experimental Results

Rendering times, resolutions, and even memory usage (even though the latter is not a concern in this work) can be reported objectively in milliseconds, millimeters, and megabytes conveying numerical information that speaks quantifiably about our results. The same, however, cannot be said when reporting the quality of haptic interaction; we are left with using subjective, non-quantifiable terms such as "as expected", "without incident", or "like a charm". This problem is not unique to our work, but rather is an issue that maligns haptic literature in general; how can you convey to the reader objectively that a model's surface indeed feels like it should or like you would expect it to feel given its shape, texture, and other surface properties?

One possible approach would be to perform a usability study where a group of subjects compares a given model rendered in some new way with the same model rendered in a pre-existing and established way. This is inapplicable in our case simply because there are no

pre-existing ways to haptically render point clouds, and haptically rendering their polygonal mesh representation will easily overwhelm any existing algorithm with hundreds of thousands if not millions of primitives.

Another approach would be to perhaps log the force vectors displayed on the haptic device and later perform analysis on them. To do this in the context of haptic rendering on a single-processor machine is self-defeating since the act of logging the force vectors will necessarily slow down the haptic loop's run rate; both are executed on the same processor. Should the logging be done on a multi-processor machine, then the loop rate is unaffected but analyzing force vectors can only expose unstable interactions and does nothing to convey whether the correct force vector has indeed been rendered.

Another issue is the repeatability of the input probe path; it is virtually impossible to carry out the same experiment more than once having a human repeat the previous path he or she has taken the haptic probe through in previous experiments. To counter this issue, we can potentially simulate the path a probe takes by feeding the rendering algorithm preset (X, Y, Z) values representing a probe's path, but if we do so, then we can no longer take into consideration the time communicating position and force information to the haptic device takes during the course of a haptic frame.

All this being said, and also leaning on the fact that we have not come across any work in the literature that quantifies its haptic interaction experience in ways other than the ones mentioned above, we are forced to settle for using subjective descriptors and point researchers in the field to this issue as a possible project for future work.

Chapter 4. Point-Based Haptic Texture

4.1. Height Fields, a Special Case of Point Clouds

Height fields are a common way to describe haptic texture in polygonal models. Not only that, but height fields have many more applications primarily in creating 3D models from 2D data. For example, height fields (called Digital Elevation Models) help create 3D models from satellite images, and, in geologic exploration, they help create 3D models from ground-penetrating radar images.

Height fields, which are in essence 2D uniform grids that carry height information in each cell, are a special kind of model representation; one that is pseudo-flat by definition, and when representing texture, one the height values of which changes slowly as the height field is navigated.

But a collection of 2D coordinates each with a height value is nothing more than a special case of a 3D point cloud, the specialization being that the cloud does not overlap itself when projected on a 2D surface. It is this perspective of a height field that we take advantage of in presenting an approach to haptically rendering height fields.

4.2. Collision Detection

In section 3.1, we justified using a segment-AABB intersection test with two main reasons: the first was to eliminate the chance of temporal aliasing, and the second was to provide necessary information to our constraint-based force response algorithm. If both of these reasons did not exist, then we could very easily use the faster point-in-AABB intersection test. In this section, we will present a point-in-AABB CD test for use with height fields, and then explain why we can abandon segment-AABB tests in favor of our approach.

4.2.1. Algorithm

Figure 53 shows a height field the height value of which is a top-anchor for an AABB. This AABB is of height equal to each field's height and of length and width equal to each field's length and width. What essentially the AABBs do is that they cover the inside of the height field model and approximate the model's surface. How good of an approximation job the boxes do solely depends on the density of the height field. However, we do expect high densities simply because of the applications that use height fields. Texture is, by definition, a high resolution model, and for haptically rendering geographic or geologic data, it obviously has to be scaled down considerably which we reason would create a high density model.

Terminology thus far used in the context of height fields is analogous to that which we have been using in the context of point clouds throughout this work. The only explicit differentiation we make here is that we define a height field's resolution along its 2D surface as opposed to how we defined the resolution of a point cloud in 3D as the furthest distance between any two immediately neighboring points. A height field resolution, for our purposes, is the furthest distance between any neighboring two points in the 2D space of the field.

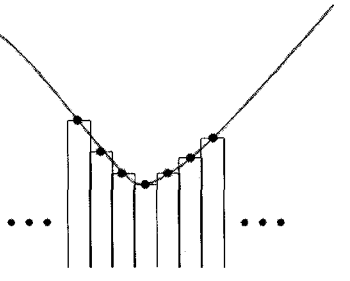

Figure 53: Detecting collision with a height field.

Furthermore, and because height fields are built on top of 2D regular grids, basing the length and width of our AABBs on the length and width of the height fields' cells guarantees that we have completely covered the inside of our model space.

At this point, as you might have guessed, narrow phase collision detection would consist of checking if the HIP is inside any of the AABBs mapping the figure. Broad phase CD is done simply by finding which grid the HIP is over using the same technique we used for uniform partitioning in our general point cloud algorithm; a simple approach because the 2D plane is uniformly partitioned.

Thanks to the uniform grid inherent in the height field, CD is constant in complexity and, in terms of absolute time, it takes near zero time to compute since it basically is two division operations (to compute the grid cell subscripts), a retrieval operation to get the height in the given cell, and a comparison to see if the HIP is in the AABB (i.e. has a z-value less than the height field) or not.

4.2.2. The Lack of Need for a Segment-AABB Intersection Test

There are two main reasons why a segment-AABB test was used in the first place: to avoid temporal aliasing, and to provide path data to the force response algorithm.

In the context of height fields and in regards to temporal aliasing, we say that it can only happen in one scenario, and that scenario is when the HIP is sampled outside the model in one frame, is moved fast enough to penetrate and pop-out the other side of a peak, and then sampled again in the next frame. Penetrating into the model cannot be missed because we know the model's inside (thanks to covering the inside with our AABBs), and penetrating into the model and exiting from an opposite surface is also impossible not only because we

have the model inside covered but also because, by definition of a height field (being a 2.5D structure), there is no "opposite surface".

Even the aforementioned scenario is unlikely to occur because of the high rate of the haptic loop when running our fast CD and FR algorithms. This will be discussed at more length in section 4.5.

The second reason why a segment-AABB test is used in the first place is that the FR algorithm needs to know the surface from which the probe entered the model in order to determine the location of the constrained god object (i.e. determine which surfaces are constraint surfaces). This is done to account for special cases such as thin or concave objects where simply looking for the nearest surface point can cause pop-out effects. In case of a height field, there is only one continuous surface; all the heights describe a single model and each point could be reached from any other point by virtue of traversing the grid in 2D, so what remains is determining where the HIP first made contact with the height field. We do this by storing the identifier of the grid cell where the first collision was detected, and then passing this information to the force response algorithm. We discuss why this works when presenting our FR algorithm next.

4.3. Force Response

Should a collision be detected in the height field, as described above, the force response algorithm is called with two parameters: the location of the first collision, and the current location of the HIP.

4.3.1. The Effects of Point-in-AABB CD on Force Response

Recall in the previous section our argument that since there is only one continuous surface then that surface is by default the constraint on our god object. Recall also how we

determined the first collision location; it was the first grid cell that the HIP entered as far as our CD algorithm could tell. As discussed in section 4.2, our CD algorithm is theoretically susceptible to temporal aliasing so of course if temporal aliasing occurs in the scenario mentioned in 4.2.2, there will be no force response, but if it occurs any other way, our method of only storing the location of the first collision will still work correctly for FR. This is because in the frame that the HIP's position is sampled after first entry (t_1 in Figure 54), the HIP would already be inside the model, and even though we missed the grid cell where it actually entered ($AABB_0$), we captured that grid cell's neighbor ($AABB_1$). Since our FR algorithm minimizes the distance between the haptic probe and the surface in the neighborhood of the first entry point, then our FR algorithm will still work despite this temporal aliasing; we're still in that neighborhood.

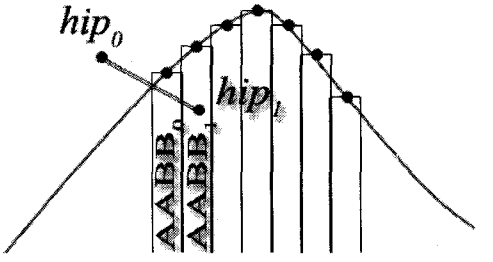

Figure 54: Temporal aliasing is tolerated by our force response algorithm.

4.3.2. Algorithm

As was the case with our general-purpose point cloud FR algorithm (section 3.2), our height field FR algorithm will search in the neighborhood of the entry point on the constraint surface for the surface point nearest the current HIP location and that point will be set as the god object. Neighborhood information is built in the height field; with the exception of border cases, each point's 8 surface neighbors necessarily reside in the 8 grid cells surrounding it.

4.4. Shared Arguments

By virtue of the arguments we presented for our general-purpose point cloud rendering approach being applicable to height fields' haptic rendering, we state explicitly that the latter supports the following modifications to the same extent as the former: Euclidean transformations as discussed in section 3.3.1, deformation as discussed in section 3.3.2, force shading as discussed in section 3.4, texturing as discussed in section 3.5.1, and friction as discussed in section 3.5.2.

4.5. Experimental Results

The first test model we chose was the sinusoidal surface in Figure 50 page 120**Error! Reference source not found.** which we also used as a test model for our general purpose approach. Being a $z = f(x,y)$ implicit function, the resulting point cloud qualified as a height field. Table 12 shows our test results.

Table 12: Information on Test #1.

Model Information	
Model description	Sinusoidal surface z = 10 * (cos(0.04*PI*x) + cos(0.04*PI*y))
Number of points	1,002,001 surface points
Resolution	0.1414 mm
Real-life dimensions	100 mm x 100 mm x 20 mm
Surface properties	None
Results (collected only when algorithm renders force)	
Collision test frames	65,820 haptic frames
Fastest rendering time	< 0.001 ms
Slowest rendering time	0.0632 ms
Average rendering time	0.0158 ms

The first result that jumps at us is the considerably faster rendering time which averaged approximate 0.0158 ms; specializing our general purpose algorithm to the purposes of rendering height fields has decreased average rendering time from 0.2674 ms to 0.0632 ms.

As for the subjective reporting on the quality of the haptic rendering, the model was rendered as expected with no artifacts or instability for the duration of an inquisitive exploration. Incidentally, there were no regions on the surface of this model where we were able to feel the effects of a missed CD due to temporal aliasing. Worth noting is that we also tested deformation, friction, and force shading with our first test model with no incidence.

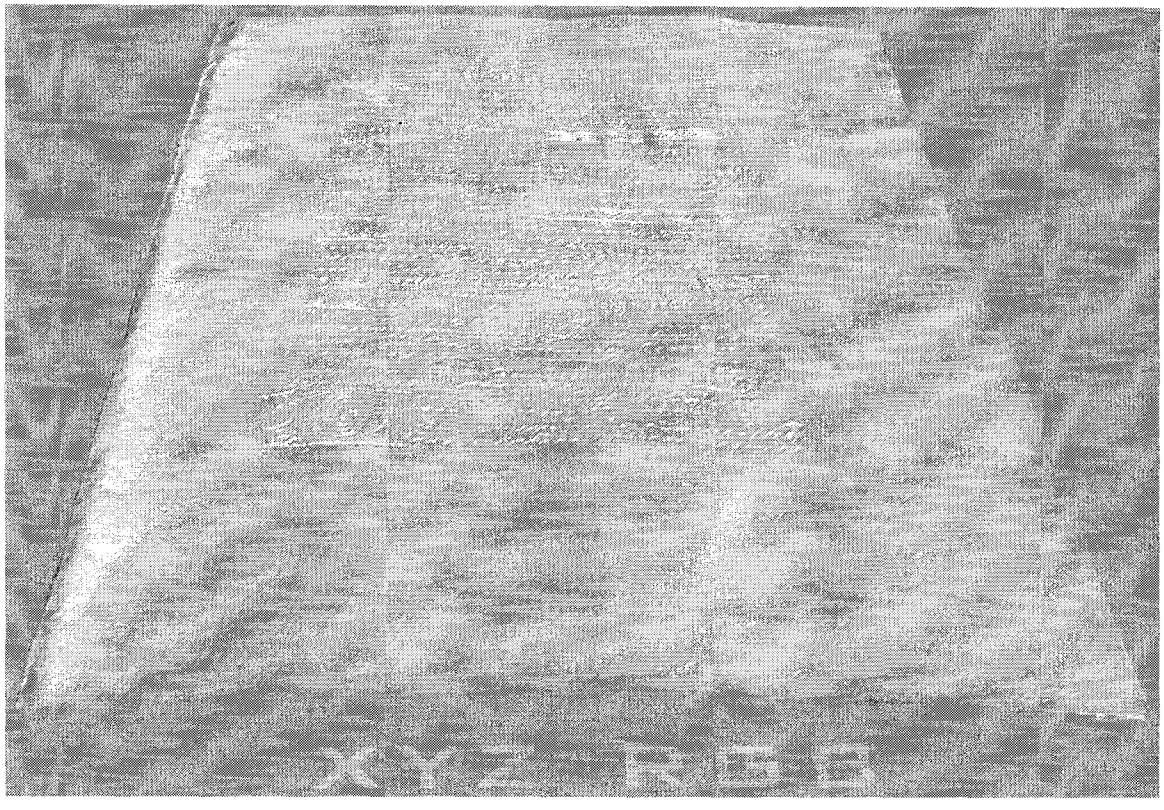

Figure 55: The XYZ-RGB Vellum Manuscript Model rendered in PlyTools.

The second test model we chose was the vellum manuscript; an XYZ-RGB model shown in Figure 48. The test results are shown in Table 13. It is models such as this one that our algorithm works very well with; the model is very flat (it's a page), is highly detailed (scanned at a resolution of 0.1 mm), and it has rich surface texture (shown in Figure 55 rendered graphically using PlyTools). The model was loaded into our test environment and was rendered without incident.

Table 13: Information on Test #2.

Model Information	
Model description	Vellum Manuscript (XYZ-RGB Model)
Number of points	2,152,840 surface points
Resolution	0.1 mm
Real-life dimensions	220 mm x 300 mm
Surface properties	Inherent texture.
Results (collected only when algorithm renders force)	
Collision test frames	32,597 haptic frames
Fastest rendering time	< 0.001 ms
Slowest rendering time	0.0632 ms
Average rendering time	0.0185 ms

Chapter 5. Conclusions and Future Directions

This work was a presentation of a comprehensive approach to haptically render high-resolution point clouds without first creating corresponding polygonal meshes. We addressed the conveyance of both kinesthetic and tactile information; our algorithms' support for static and dynamic models; two collision detection algorithms (one for general point clouds and one for height fields); and several adaptations of standard force response algorithms that brought constraint-based force response, force shading, deformation, friction, and texturing to the domain of point clouds.

Our work was motivated throughout by the close proximity that points in a point cloud have to each other, especially if they result from 3D scans of real-life objects. Thanks primarily to this proximity, we were able to redefine a point cloud's surface in such a way that allowed for fast collision detection and accurate force response both meeting or exceeding standards set by previous works in haptic rendering.

We have also met the goals we had set for our algorithms when presenting this work's problem statement: We demonstrated that both kinesthetic and tactile information is relayed reliably and fully (to the extent of the detail provided in the point cloud) to the user; we stayed away from explicit surface reconstruction techniques already used in the literature; we developed algorithms that were fast enough to run on a single processor well under the 1 ms ceiling for each haptic frame; and we showed how our algorithms' complexities are highly insensitive to the number of points input in the form of a point cloud describing a scanned real-life model.

This is not to say that there is no more work left to do in this largely untapped area.

Our approaches are ill-suited for use with low-resolution point clouds primarily because of our reliance on dense data to deduce neighborhood information and perform collision detection. There is room for research in both these fields. Furthermore, there are lessons to be learned from the graphics rendering of point clouds as alluded to in section 2.3, especially from milestone works such as QSplat [84]. QSplat is a level-of-detail driven algorithm that, among other things, progressively loads finer and finer models of a point cloud using some interesting techniques, many of which could be adapted to haptics continuing in the tradition of stealing from the graphics domain to advance haptics research. More generally, level-of-detail loading of point clouds into haptic environments is, we believe, a fertile area for more research. It is true that our algorithms presented in this work are largely insensitive, complexity wise, to the number of points in a point cloud, but memory resources are very much sensitive to million-primitive models being loaded and rendered whether graphically, haptically, or both. Further investigation of memory-conscious techniques in the domain of point cloud haptic rendering is needed.

Our research into tactile rendering in this work has also made clear, first hand, the ineptitude of current hardware in rendering high quality texture information. We are not qualified to give an opinion on what direction hardware specialists should take to build better tactile displays, but on the software side, we are confident that either point clouds or mathematical models of textured surfaces will be the representation technique of choice for modeling texture; they both are the only representations qualified to convey minute geometry information efficiently.

Finally, we reiterate the need to devise tests that can accurately quantify the haptic experience, in effect substituting subjective descriptors with metrics that could be compared

objectively. Instead of saying that a model "felt as expected", an objective score can perhaps be given relaying just how "well" a model was being haptically displayed.

To whoever proceeds with our suggestions for future research, we wish you the best of luck.

References

[1] R. Fabio, "From Point Cloud to Surface: The Modeling and Visualization Problem", International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 2003; XXXIV-5/W10:215-230

[2] R. Mencl. "Reconstruction of Surfaces from Unorganized 3D Points Clouds". 2001 PhD Thesis, Dortmund University, Germany

[3] H. Edelsbrunner. "Geometry and Topology for Mesh Generation", Cambridge Monographs on Applied and Computational Mathematics (Vol. 6, 2001). Cambridge University Press, UK

[4] 3D Examination of the Mona Lisa. National Research Council Canada. http://iit-iti.nrc-cnrc.gc.ca/projects-projets/monalisa-lajoconde_e.html. Last accessed on May, 9th 2007

[5] M. Levoy et al. "The digital Michelangelo project: 3D scanning of large statues", Proceedings of the 27th annual conference on Computer graphics and interactive techniques, p.131-144, July 2000

[6] G. Turk, M. Levoy, "Zippered polygon meshes from range images", In Proceedings of the 21st Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '94. ACM, New York, NY, 311-318

[7] B. Curless, M. Levoy, "A volumetric method for building complex models from range images". In Proceedings of the 23rd Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '96. ACM, New York, NY, 303-312

[8] C. Ericson. Real-Time Collision Detection. Morgan Kaufmann, 2004

[9] C. Ho, C. Basdogan, M. A. Srinivasan, "Efficient Point-Based Rendering Techniques for Haptic Display of Virtual Objects". Presence: Teleoper. Virtual Environ. 8, 5 (Oct. 1999), 477-491

[10] Reachin API Programming Guide. Reaching Technologies AB. 2004

[11] F. R. El-Far, M. Eid, M. Orozco, A. El Saddik, "Haptic Applications Meta-Language," IEEE DS-RT '06, pp. 261-264

[12] FCS HapticMASTER User Guide. Moog FCS 2002

[13] M. D. Minsky. "Computational Haptics: the Sandpaper System for Synthesizing Texture for a Force-Feedback Display". Doctoral Thesis. UMI Order Number: AAI0576366., Massachusetts Institute of Technology.

[14] J. Siira, D.K. Pai, "Haptic texturing-a stochastic approach," Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on , vol.1, no., pp.557-562 vol.1, 22-28 Apr 1996

[15] D.F. Green, J.K. Salisbury "Texture Sensing and Simulation Using the PHANToM: Towards Remote Sensing of Soil Porperties," Second PHANToM User's Group Workshop, Oct. 19-22, 1997

[16] Model Shop Color 3D Scanner. http://www.cyberware.com/products/scanners/ms.html. Last accessed Jan. 8th, 2008

[17] Faro International USA. http://www.faro.com/content.aspx?ct=US&content=pro&item=1. Last accessed Jan. 8th, 2008

[18] Metris - Model Maker D. http://www.metris.com/handheld_scanners/modelmaker_d/. Last accessed Jan. 8th, 2008

[19] Micrometric Vision CLS 60 data sheet. http://www.micrometric-vision.com/Brochure%20CLS60.pdf. Last accessed on Jan. 8th, 2008

[20] Micrometric Vision CLS 60 Plus data sheet. http://www.micrometric-vision.com/Brochure%20CLS60%20Plus.pdf. Last accessed on Jan. 8th. 2008

[21] Inition - Immersion Microscribe MX - Manual 3D Digitizers. http://www.inition.com/inition/product.php?URL_=product_digiscan_immersion_microscribemx&SubCatID_=32. Last accessed on Jan. 8th, 2008

[22] High Resolution 3D Color Laser Scanner. http://iit-iti.nrc-cnrc.gc.ca/vit-tiv/high-res-scanner_e.html. Last accessed on Jan. 8th, 2008

[23] Polhemus FastScan 3D. http://www.fastscan3d.com/. Last accessed on Jan. 8th, 2008

[24] Cyberware Large Statue Laser Scanner. http://www.cyberware.com/products/scanners/lss.html. Last accessed on Jan. 8th, 2008

[25] Point cloud model. Gallery of Volume Graphics. http://www2.imm.dtu.dk/~jab/gallery/volumegraphics.htm. Last accessed on Jun. 1, 2007

[26] The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/. Last accessed on Jan. 8th, 2008

[27] A. Mhatre and P. Kumar, "Projective clustering and its application to surface reconstruction: extended abstract". In Proceedings of the Twenty-Second Annual Symposium on Computational Geometry (Sedona, Arizona, USA, June 05 - 07, 2006). SCG '06. ACM, New York, NY, 477-478

[28] T. K. Dey and J. Sun, "Normal and Feature Approximations from Noisy Point Clouds". In Proc. of FSTTCS 2006, December 13-15, 2006, Kolkata, India

[29] N. Amenta, M. Bern, and M. Kamvysselis, "A new Voronoi-based surface reconstruction algorithm". In Proceedings of the 25th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '98. ACM, New York, NY

[30] T. K. Dey, J. Giesen, and J. Hudson, "Delaunay based shape reconstruction from large data". In Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics, San Diego, California

[31] N. Amenta, S. Choi, and R. K. Kolluri. "The power crust". In Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications SMA '06 Ann Arbor, Michigan, United States

[32] N. R. El-Far. Telephone conversation with Cyberware Technical Support. May 2008.

[33] K. Salisbury, F. Conti, F. Barbagli, "Haptic Rendering: Introductory Concepts," IEEE Computer Graphics and Applications, vol. 24, no. 2, pp. 24-32, Mar/Apr, 2004

[34] M. Eid et al. Haptic Rendering. Technical report, University of Ottawa. 2007

[35] X. Wu, "A linear-time simple bounding volume algorithm", In Graphics Gems III, D. Kirk, Ed. Academic Press Graphics Gems Series. Academic Press Professional, San Diego, CA, 301-306. 1992

[36] E. Welzl, "Smallest Enclosing Disks (Balls and Ellipsoids)," New Results and New Trends in Computer Science, pp. 359-370, 1991

[37] N. Capens, "Smallest Enclosing Spheres." flipcode Code Of The Day Forum, June 29, 2001. http://www.flipcode.com/cgi-bin/msg.cgi?showThread=COTD-SmallestEnclosingSpheres&forum=cotd&id=-1 and ftp://ftp.flipcode.com/cotd/MiniBall.zip

[38] J. O'Rourke, "Finding Minimal Enclosing Boxes", International Journal of Computer and Information Sciences, vol. 14, no. 3, pp. 183-199, June 1985

[39] G. Barequet, C. Bernard, L. Guibas, et al. "BOXTREE: A Hierarchical Representation for Surfaces in 3D." Computer Graphics Forum, vol. 15, no. 3, pp. 387-396, 1996

[40] S. Gottschalk. "Collision Queries Using Oriented Bounding Boxes," Ph.D. thesis, Department of Computer Science, University of North Carolina, Chapel Hill, 2000

[41] D. Eberly, 3D Game Engine Design. Morgan Kaufmann Publishers, 2001.

[42] B. Mirtich, "Fast and Accurate Computation of Polyhedral Mass Properties." Journal of Graphics Tools. vol. 1, no. 2, pp. 31-50, 1996

[43] P. Konečný et al. "Lower Bound of Distance in 3D." Proceedings of WSCG 1997, vol. 3, pp. 640-649, 1997. See also: Technical report FIMU-RS-97-01, Faculty of Informatics, Masaryk University, Brno, Czech republic

[44] J. Klosowski, M. Held, J. Mitchell. H. Sowizral, K. Zikan. "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs." IEEE Transactions on Visualization and Computer Graphics, vol. 4, no. 1, pp. 21-36, 1998

[45] k-DOP. Wikpedia. http://en.wikipedia.org/wiki/K-DOP. Last accessed on May 30, 2007.

[46] iEhovah Terrain Library. http://www.home.zonnet.nl/petervenis/data/octree.gif. Last accessed on Jun. 2, 2007

[47] kd-tree. Wikipedia. http://en.wikipedia.org/wiki/Kd-tree. Last accessed on Jun. 2, 2007

[48] M. C. Lin, J. F. Canny, J.F., "A fast algorithm for incremental distance calculation," Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on , vol., no., pp.1008-1014 vol.2, 9-11 Apr 1991

[49] J. D. Cohen, M. C. Lin, D. Manocha, M. Ponamgi, "I-COLLIDE: an interactive and exact collision detection system for large-scale environments". In Proceedings of the 1995 Symposium on interactive 3D Graphics (Monterey, California, United States, April 09 - 12, 1995). SI3D '95. ACM, New York, NY

[50] M. Eid et al. Implementation and Evaluation of Several Haptic Collision Detection Algorithms. Technical report, University of Ottawa. 2007

[51] M. de Pascale and D. Prattichizzo, "A framework for bounded-time collision detection in haptic interactions", In Proceedings of the ACM Symposium on

Virtual Reality Software and Technology (Limassol, Cyprus, November 01 - 03, 2006). VRST '06

[52] J. K. Lee, Y. J. Kim, "Haptic Rendering of Point Set Surfaces", EuroHaptics Conference, 2007 and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems. World Haptics 2007. Second Joint , vol., no., pp.513-518, 22-24 March 2007

[53] P. Hubbard, "Interactive collision detection", Virtual Reality, 1993. Proceedings., IEEE 1993 Symposium on Research Frontiers in, pages 24--31, 1993

[54] S. Quinlan, "Efficient distance computation between non-convex objects", Robotics and Automation, 1994. Proc. IEEE International Conference on, pages 3324--3329, 1994

[55] N. Beckmann et al, "The R*-tree: an efficient and robust access method for points and rectangles", Proceedings of the 1990 ACM SIGMOD international conference on Management of data, p.322-331, May 23-26, 1990, Atlantic City, New Jersey, United States

[56] M. Held, J. Klosowski, and J. Mitchell, "Evaluation of collision detection methods for virtual reality fly-throughs", Proc. 7th Canad. Conf. Comput. Geom, pp. 205-210, 1995

[57] G. van den Bergen. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees." Journal of Graphics Tools, vol. 2, no. 4, pp. 1-14, 1997

[58] S. Gottschalk , M. C. Lin , D. Manocha, "OBBTree: a hierarchical structure for rapid interference detection", Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, p.171-180, August 1996

[59] S. Redon, A. Kheddar, and S. Coquillart. "Collision Detection and Augmented Reality: Fast Continuous Collision Detection between Rigid Bodies". Computer Graphics Forum, 21(3):279, 2002

[60] S. Cameron, "Approximation hierarchies and S-bounds", Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications, p.129-137, June 05-07, 1991, Austin, Texas, United States

[61] S. Krishnan, A. Pattekar , M. Lin , D. Manocha, "Spherical shell: a higher order bounding volume for fast proximity queries, Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics : the algorithmic perspective: the algorithmic perspective", p.177-190, August 1998, Houston, Texas, United States

[62] J. Klosowski, M. Held, J. Mitchell. H. Sowizral, K. Zikan. "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs." IEEE Transactions on Visualization and Computer Graphics, vol. 4, no. 1, pp. 21-36, 1998

[63] H. Samet and R.E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics". Part I., IEEE Computer Graphics and Applications, v.8 n.3, p.48-68, May 1988

[64] B. Naylor, J. Amanatides, and W. Thibault, "Merging BSP trees yields polyhedral set operations", Proceedings of the 17th annual conference on Computer graphics and interactive techniques, p.115-124, September 1990, Dallas, TX, USA

[65] H. Konig and T. Strothotte. "Fast Collision Detection for Haptic Displays Using Polygonal Models". Proceedings of the Conference on Simulation and Visualization, 300, 2002

[66] D. Borro, A. Garcia-Alonso, and L. Matey. "Approximation of Optimal Voxel Size for Collision Detection in Maintainability Simulations within Massive Virtual Environments". Computer Graphics Forum, 23(1):13--23, 2004

[67] CHAI 3D: The Open Source Haptics Project. http://www.chai3d.org/. Last accessed on Mar 10, 2007

[68] H3D: Open Source Haptics. http://www.h3d.org/. Last accessed on Mar. 10, 2007

[69] SensAble 3D Touch SDK OpenHaptics Toolkit version 1.0 Programmers' Guide. 2004. http://www.sensable.com/61614600231945743464549037/Link.htm. Last accessed on Mar. 10, 2007

[70] C. Zilles, J. K. Salisbury, "A Constraint-Based God object Method for Haptic Display", Proc. IEE/RSJ Intl Conf. Intell. Robots and Systems, Human Robot Interaction, and Cooperative Robots, vol. 3, IEEE CS Press, 1995, pp 146-151

[71] D.C. Ruspini, K. Kolarc, and O. Khatib, "The Haptic Display of Complex Graphical Environments", Proc. ACM Siggraph, ACM Press, 1997, pp. 345-352

[72] CS239 Course Notes (Spring 2005). "Introduction to Haptic Rendering by Ming C. Lin" (Powerpoint presentation 21-Feb-2005). University of Carolina, Chapel Hill. http://www.cs.unc.edu/Courses/comp239-s05/index_files/05.02.21HapticRendering.ppt. Last accessed on Jan. 8th, 2008

[73] S.P. Walker, J. K. Salisbury. "Large haptic topographic maps: marsview and the proxy graph algorithm". In Proceedings of the 2003 Symposium on interactive 3D Graphics (Monterey, California, April 27 - 30, 2003). I3D '03. ACM Press, New York, NY, 83-92. 2003. DOI= http://doi.acm.org/10.1145/641480.641499

[74] U. Meier, O. López, C. Monserrat, M. Juan, M. Alcañiz, "Real-time deformable models for surgery simulation: a survey". Computer Methods and Programs in Biomedicine, Volume 77, Issue 3, Pages 183-197

[75] S.E. Lithicum et al. System and method for providing interactive haptic collision detection. US Patent 671421

[76] J. Colgate, M. Stanley, and J. Brown. "Issues in the haptic display of tool use", In Proceedings of the international Conference on intelligent Robots and Systems-Volume 3 - Volume 3 (August 05 - 09, 1995). IROS. IEEE Computer Society, Washington, DC, 3140. 1995

[77] Y. Adachi, T. Kumano, K. Ogino, "Intermediate representation for stiff virtual objects". In Proceedings of the Virtual Reality Annual international Symposium (Vrais'95) (March 11 - 15, 1995). VRAIS. IEEE Computer Society, Washington, DC, 203. 1995

[78] Phantom Desktop. Technical Specifications. http://www.sensable.com/haptic-phantom-desktop.htm. Last accessed on Apr. 15, 2007

[79] Phantom Omni. Technical Specifications. http://www.sensable.com/haptic-phantom-omni.htm. Last accessed on Apr. 15, 2007

[80] Impedance control. FCS Robotics, Robotic Technology. http://www.fcs-cs.com/robotics/technology. Last accessed on Dec. 1, 2004

[81] Admittance control. FCS Robotics, Robotic Technology. http://www.fcs-cs.com/robotics/technology. Last accessed on Dec. 1, 2004

[82] Comparison Table. FCS Robotics, Robotic Technology. http://www.fcs-cs.com/robotics/technology. Last accessed on Dec. 1, 2004

[83] L. Kobbelt and M. Botsch. "A survey of point-based techniques in computer graphics", Computers & Graphics 28(6): 801-814 (2004)

[84] S. Rusinkiewicz, M. Levoy, "QSplat: a multiresolution point rendering system for large meshes". In Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques International Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley Publishing Co., New York, NY, 343-352

[85] V. Hayward, B. Armstrong, "A New Compuatitional Model of Friction Applied to Haptic Rendering". In the Sixth international Symposium on Experimental Robotics VI (March 26 - 28, 1999). P. Corke and J. Trevelyan, Eds. Lecture Notes in Control and Information Sciences, vol. 250. Springer-Verlag, London, 403-412

[86] A. Gregory, M. C. Lin , S. Gottschalk , R. Taylor, "A Framework for Fast and Accurate Collision Detection for Haptic Interaction", Proceedings of the IEEE Virtual Reality, p.38, March 13-17, 1999

[87] N. R. El-Far, X. Shen, N. D. Georganas, "Applying Unison, a Generic Framework for Hapto-Visual Application Development, to an E-Commerce Application", Proc. IEEE Workshop on Haptic Audio Visual Environments and their Applications, Ottawa, ON, Canada, October 2004

[88] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, "Surface reconstruction from unorganized points", In Proceedings of the 19th Annual Conference on Computer Graphics and interactive Techniques J. J. Thomas, Ed. SIGGRAPH '92. ACM, New York, NY, 71-78

[89] D. Badouel, "An efficient ray-polygon intersection", Graphics gems, Academic Press Professional, Inc., San Diego, CA, 1990

[90] D. Bielser, M.H. Gross, "Interactive Simulation of Surgical Cuts," pg, p. 116, Eighth Pacific Conference on Computer Graphics and Applications (PG"00), 2000

[91] FCS HapticMASTER. Technical Specifications. http://www.est-kl.com/hardware/haptic/fcs/hapticmaster.html. Last accessed on Apr. 15, 2007

[92] G. Zachmann. "The BoxTree: Exact and Fast Collision Detection of Arbitrary Polyhedra." First Workshop on Simulation and Interaction in Virtual Environments (SIVE 95), University of Iowa, July 1995

[93] H. B. Morgenbesser, M.A. Srinivasan, "Force Shading for Haptic Perception", Proceedings of the 1996 ASME International Mechanical Engineering Congress and Exposition, Dynamic Systems and Control Division, Vol. 58, Atlanta, GA, 1996, pp. 407-412

[94] H. Samet, "Applications of Spatial Data Structures", Computer Graphics, Image Processing, and GIS. Addison-Wesley, 1993

[95] H. Samet, "The Design and Analysis of Spatial Data Structures", Addison-Wesley, 1990.

[96] HapticMASTER API Programming Guide. FCS Control Systems. 2002

[97] J. Arvo , David Kirk, "A survey of ray tracing acceleration techniques, An introduction to ray tracing", Academic Press Ltd., London, UK, 1989

[98] J. Cohen et al, "I-COLLIDE: an interactive and exact collision detection system for large-scale environments", In Proceedings of the 1995 Symposium on interactive

3D Graphics (Monterey, California, United States, April 09 - 12, 1995). SI3D '95. ACM Press, New York, NY

[99] M. Lin and S. Gottschalk, "Collision detection between geometric models: A survey", Proc. of IMA Conference on Mathematics of Surfaces, 1:602–608, 1998

[100] N. K. Govindaraju , M. C. Lin , D. Manocha, "Fast and reliable collision culling using graphics hardware", Proceedings of the ACM symposium on Virtual reality software and technology, November 10-12, 2004, Hong Kong

[101] P. Jimenez, F. Thomas, and C. Torras. "3D collision detection: a survey", Computers and Graphics, 25(2):269–285, 2001

[102] P. Liu, X. Shen, N.D. Georganas, G. Roth, "Multi-resolution Modeling and Locally Refined Collision Detection for Haptic Interaction", Proc. 3DIM 2005: The Fifth International Conference on 3-D Digital Imaging and Modeling, Ottawa, ON, Canada, June 2005

[103] S. Hadap et al. "Collision detection and proximity queries", In ACM SIGGRAPH 2004 Course Notes (Los Angeles, CA, August 08 - 12, 2004). SIGGRAPH '04. ACM Press, New York, NY

[104] SensAble – Phantom Desktop. http://www.sensable.com/haptic-phantom-desktop.htm. Last accessed on Mar. 2, 2007

[105] SWIFT++: Speedy Walking via Improved Feature Testing for Non-Convex Objects. http://www.cs.unc.edu/~geom/SWIFT++/. Last accessed on Mar 1, 2007

[106] T. Möller, B. Trumbore, "Fast, minimum storage ray-triangle intersection", Journal of Graphics Tools, v.2 n.1, p.21-28, 1997

[107] T.C. Hudson et al, "V-COLLIDE: accelerated collision detection for VRML", In Proceedings of the Second Symposium on Virtual Reality Modeling Language VRML '97 Monterey, California, United States, February 24 - 26, 1997

[108] Y. Kim, M. Lin, and D. Manocha, "DEEP: Dual-space Expansion for Estimating Penetration depth between convex polytopes". In IEEE Conference on Robotics and Automation ICRA '02