Geometric operators for motion planning

Jesse Himmelstein

▶ To cite this version:

Jesse Himmelstein. Geometric operators for motion planning. Computer Science [cs]. INSA de Toulouse, 2008. English. NNT : . tel-00348010

HAL Id: tel-00348010
https://theses.hal.science/tel-00348010

Submitted on 17 Dec 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

Préparée au Laboratoire d'Analyse de d'Architecture des Systèmes du CNRS

En vue de l'obtention du Doctorat de
l'Institut National des Sciences Appliquées de Toulouse

Ecole Doctorale Systèmes
Spécialité : Systèmes Informatiques

Par **Jesse Cooper HIMMELSTEIN**

Geometric Operators for Motion Planning

Soutenue le 19 septembre 2008 devant le jury :

Président	Thierry Siméon
Directeur de thèse	Jean-Paul LAUMOND
Rapporteurs	Emmanuel MAZER Dinesh MANOCHA
Examinateurs	Etienne FERRE

Abstract

Keywords: geometric reasoning, assembly sequence planning, motion planning, collision detection, swept volume calculation

Motion planning is building a considerable momentum within industrial settings. Whether for programming factory robots or calculating mechanical assembly sequences, motion planning through probabilistic algorithms has proved to be particularly efficient for solving complex problems that are difficult for human operators.

This doctoral thesis, a collaborative work between the research laboratory LAAS-CNRS and the startup company Kineo CAM, is aimed confronting motion planning problems encountered in the virtual factory. We have identified three domains that are of interest to industrial partners and we contribute to each: collision detection, swept volumes, and motion planning in collision.

Collision detection is a critical operator for analyzing digital models within their environment. Motion planning algorithms rely so heavily on collision detection that it has become a performance bottleneck. This explains why such a large variety of collision detection algorithms exist, each specialized for a particular type of geometry, such as polyhedra or voxels. Such a diverse solution space is a barrier for integrating multiple geometry types into the same architecture.

We propose a framework for performing proximity queries between heterogeneous geometries. While factoring out the algorithmic core common to spatial-division and bounding-volume schemes, the framework allows specialized collision tests between a pair of geometric primitives. New geometry types can thus be added easily and without hurting performance. We validate our approach on a humanoid robot that navigates an unknown environment using vision.

Swept volumes are a useful tool for visualizing the extent of a movement, such as the vibrations of an engine or the reaching of a digital human actor. The state-of-the-art approach exploits graphics hardware to quickly approximate swept volumes with a high accuracy, but only applies to a single watertight object. To adapt this algorithm to handle computer-aided design input, we modify its behavior to treat polygon soup models and discontinuous paths. We demonstrate its effectiveness on disassembly movements of mechanical pieces with a large number of triangles.

It can be challenging to manipulate the volume described by a polygon soup. Starting with the swept volume algorithm, we introduce operators to change the size of discrete objects. At a basic level, we calculate the Minkowski sum of the object and a sphere in order to inflate the object, and the Minkowski difference to deflate it. We test these operators on both static and moving objects.

Finally, we take on the problem of motion planning in collision. Although it may appear as a contradiction in terms, the ability to authorize a limited penetration during the planning process can be a powerful tool for certain difficult motion planning problems. For example, when calculating disassembly sequences, we can allow obstacles such as screws to move during the planning. In addition, by allowing collision we are able to solve forced passage problems. This is a difficult problem encountered in virtual mockups, where certain parts are slightly deformable or where we may be asked to find the "least-worst path" when no non-colliding path exists.

In this doctoral work we develop several contributions that apply to industrial robotics and automation. By focusing on the strict functional and usability requirements of the domain, we hope that our algorithms are directly applicable as well as scientifically valuable. We try to expose the advantages as well as the disadvantages of our approach throughout the thesis.

Thesis prepared at LAAS-CNRS: 7 avenue du Colonel Roche, 31077 Toulouse Cedex 4, FRANCE

Résumé

Mots-clés: raisonnement géométrique, planification de taches d'assemblage, planification de mouvement, détection de collision, calcul du volume balayé

La planification du mouvement connait une utilisation croissante dans le contexte industriel. Qu'elle soit destinée à la programmation des robots dans l'usine ou au calcul de l'assemblage d'une pièce mécanique, la planification au travers des algorithmes probabilistes est particulièrement efficace pour résoudre des problèmes complexes et difficiles pour l'opérateur humain.

Cette thèse CIFRE, effectuée en collaboration entre le laboratoire de recherche LAAS-CNRS et la jeune entreprise Kineo CAM, s'attache à résoudre la problématique de planification de mouvement dans l'usine numérique. Nous avons identifié trois domaines auxquels s'intéressent les partenaires industriels et nous apportons des contributions dans chacun d'eux: la détection de collision, le volume balayé et le mouvement en collision.

La détection de collision est un opérateur critique pour analyser des maquettes numériques. Les algorithmes de planification de mouvement font si souvent appel à cet opérateur qu'il représente un point critique pour les performances. C'est pourquoi, il existe une grande variété d'algorithmes spécialisés pour chaque type de géométries possibles. Cette diversité de solutions induit une difficulté pour l'intégration de plusieurs types de géométries dans la même architecture.

Nous proposons une structure algorithmique rassemblant des types géométriques hétérogènes pour effectuer les tests de proximité entre eux. Cette architecture distingue un noyau algorithmique commun entre des approches de division de l'espace, et des tests spécialisés pour un couple de primitives géométriques donné. Nous offrons ainsi la possibilité de facilement ajouter des types de données nouveaux sans pénaliser la performance. Notre approche est validée sur un cas de robot humanoïde qui navigue dans un environnement inconnu grâce à la vision.

Concernant le volume balayé, il est utilisé pour visualiser l'étendue d'un mouvement, qu'il soit la vibration d'un moteur ou le geste d'un mannequin virtuel. L'approche la plus innovante de la littérature repose sur la puissance du matériel graphique pour calculer une approximation du volume balayé très rapidement. Elle est toutefois limitée en entrée à un seul objet, qui lui-même doit décrire un volume fermé. Afin d'adapter cet algorithme au contexte de la conception numérique, nous modifions son comportement pour traiter des « soupes de polygones » ainsi que des trajectoires discontinues. Nous montrons son efficacité sur les mouvements de désassemblage pour des pièces avec un grand nombre de polygones.

Il est difficile de manipuler le volume décrit par une soupe de polygones. A partir du calcul du volume balayé, nous introduisons des opérateurs qui changent la taille de l'objet discret. Ces

operateurs calculent la somme de Minkowski entre l'objet et une sphère afin d'agrandir l'objet, et la différence de Minkowski pour le rétrécir. Nous obtenons les résultats sur les objets statiques ainsi que dynamiques.

Enfin, nous abordons le problème de la planification de mouvement en collision. Cette antilogie exprime la capacité d'autoriser une collision bornée pendant la recherche de trajectoire. Ceci permet de résoudre certains problèmes d'assemblage très difficiles. Par exemple, lors du calcul des séquences de désassemblage, il peut être utile de permettre à des « pièces obstacles » telles que les vis de se déplacer pendant la planification. De plus, en autorisant la collision, nous sommes capables de résoudre des problèmes de passage en force. Cette problématique se pose souvent dans la maquette numérique où certaines pièces sont « souples » ou si le problème consiste à identifier la trajectoire « la moins pire » quand aucun chemin sans collision n'existe.

Nous apportons dans ce travail plusieurs contributions qui s'appliquent à la conception numérique pour la robotique industrielle. Nous essayons de marier une approche scientifique avec des critères de fonctionnalités strictes pour mieux s'adapter aux utilisateurs de la conception numérique. Nous cherchons à exposer les avantages et les inconvénients de nos approches tout au long du manuscrit.

Thèse préparée au LAAS-CNRS : 7 avenue du Colonel Roche, 31077 Toulouse Cedex 4, FRANCE

Acknowledgments

I knew that writing a PhD would be challenging, and I knew that I would learn a lot in the process. But I hadn't suspected how downright fascinating it could be. As I dug deeper into the theory and technology, seemingly dry subjects revealed themselves to be full of wonder. In addition to the intellectual eye-opening, this PhD has been, at least up until the last month or so, a lot of fun. This is in no small measure thanks to a small number of people who have guided, supported, and pushed me along.

It is hard to know where to begin, but I must single out my incredible wife, Bénédicte Meillon, for helping me through this. Having done her own thesis may have made it easier for her to understand, but there is no explaining her endless patience, reassurance, and proofreading other than love.

When it comes to the work itself, Jean-Paul Laumond has been a fantastic advisor to me. Not only am I honored that he agreed to direct my thesis, but I appreciate how he has always put aside the time out of his overcharged schedule to deal with the most minute of my problems. Without his vision and perspective on both robotics and the world of research, I would not have known where to begin, let alone what direction to follow.

In the same breath, I must thank Etienne Ferré for his complementary perspective on industry and business. He is in a unique position to appreciate where the needs of our customers meet our technological capabilities, and his advice has invariably proved to be measured and just. I am also grateful for the liberty he has allowed me to pursue my ideas as well as his, for even when I have missed the mark completely, I believe that I am a stronger researcher for it.

On a day-to-day basis, Kineo CAM has been a wonderful place to work and learn. I would like to thank Laurent Maniscalco for providing me the opportunity to develop this thesis. As I was finishing my Master's degree, I found myself at a crossroads, unsure whether to explore new theories as an academic researcher or create useful products as a commercial developer. It has been rewarding to play both roles at Kineo CAM.

Also at Kineo, I'm grateful to Emilie for her assistance on administrative tasks and anything else that has come up in the past few years. The development team there—Guillaume, Ambroise, Nicholas and Antoine—have helped me track down the most annoying of bugs and think hard about software design. Although it is humbling to be around programmers smarter than myself, I am proud of the work we have done together and learned a lot from them.

Even if I have not gone to the LAAS on a daily basis, I have been fortunate to work with two other PhD students there, Alireza and Sébastien, and have enjoyed it immensely. I would also like to thank the director of the LAAS, Raja Chatila, for graciously allowing me to participate in the laboratory.

On a personal note, I can never thank my parents, Jay and Ellen, enough for their (seemingly endless) faith in me, which has been comforting and given me quite a high mark to

shoot for. Though far away, I feel their love always. Along with my sister Julia, I am truly lucky to know such people, to say nothing of having them as a family.

Here in France, I also consider myself lucky to have some wonderful *beaux-parents*. Since the first time I was invited to their dinner table, André and Danièle have taken me in as their own and introduced me to the wonderful French life of wine, duck, and conversation. Hanging out at Club Meillon on hot summer days with Brigitte has given me the relaxation I needed to finish this PhD.

Lastly, thanks to my little Zoë, whose beautiful smile and flailing arms give me a feeling I cannot describe.

Contents

1 Introduction 1

1.1 Brief History of Industrial Robotics 2
1.2 Role of Motion Planning 3
1.3 Challenges in Industrial Motion Planning 4
1.4 Presentation of this Work 5
1.4.1 Contributions 5
1.4.2 Organization 6
1.4.3 Publications 6

2 Motion Planning 9

2.1 Configuration Space 9
2.2 Probabilistic Roadmap Planner 12
2.3 Rapidly-Exploring Random Tree 13
2.3.1 Iterative Path Planner 15
2.3.2 Manhattan-Like RRT 16
2.4 Remarks for this Work 17

3 Wrapped Volumes 19

3.1 Swept Volumes and Wrapping 19
3.2 Related Work 21
3.2.1 Implicit Surfaces and Distance Fields 22
3.2.2 GPU-Based Directed Distances 23
3.3 Wrapping Polygon Soups 25
3.3.1 Trajectory Sampling 25
3.3.2 Distance Field Creation 26
3.3.3 Surface Extraction 27
3.3.4 Triangulation 29
3.3.5 Error Bounds 30
3.4 Wrapping with Offsets 31
3.4.1 Why Offset? 32

3.4.2 Inflation 33
3.4.3 Deflation 36
3.5 Experimental Results 38
3.5.1 Swept Volume 38
3.5.2 Offset 42
3.6 Conclusion 44

4 Generalized Collision Detection 47

4.1 Motivation 48
4.2 Related Work 48
4.3 Collision Detection Framework 49
4.4 Test Tree Descent 52
4.4.1 Dispatch and Detection 52
4.4.2 Tree Traversal 53
4.5 Test Tree Structure 53
4.5.1 Generic Traversal 53
4.5.2 Memory Optimization 54
4.6 Application Integration 55
4.6.1 Scene Graph 55
4.6.2 Collision Entities 55
4.6.3 Test Tree Construction 56
4.6.4 Aggregate Test Trees 57
4.7 Dynamic Voxel Map for Robotic Vision System 58
4.7.1 3D Reconstruction 58
4.7.2 Dynamic Voxel Map 59
4.7.3 Experimental Design 60
4.7.4 Results 61
4.8 Future Work 62

5 Motion Planning in Collision 65

5.1 Related Work 65
5.2 Implicitly Controlling Penetration Distance 66
5.3 Teleportation-Based Planner 68
5.3.1 Uniform Shoot 69
5.3.2 Distance Metric 70
5.3.3 Teleportation 71

5.3.4 Manhattan-Like RRT 72
5.4 Superposition Collision Operator 74
5.4.1 Comparison with the Teleportation-Based Planner 76
5.5 Distance Measurement 77
5.5.1 Separation Distance 78
5.5.2 Penetration Distance 78
5.6 Dynamic Play 80
5.6.1 Draft Paths 80
5.6.2 Play Optimization 80
5.7 Experimental Results 81
5.8 Conclusion 85

6 Future Directions **87**

6.1 Results and Future Work 87
6.2 Working with Industrial Robotics 89

Table of Figures

Figure 1.1: Historical landmarks in industrial robotics. 2

Figure 1.2: Polygon soup models. 5

Figure 1.3: Contributions. 6

Figure 2.1: Shakey the robot. 10

Figure 2.2: Configuration space for translations. 11

Figure 2.3: Motion planning in configuration space. 11

Figure 2.4: Probabilistic Roadmap Planner (PRM). 13

Figure 2.5: Rapidly-exploring Random Tree (RRT). 15

Figure 2.6: Iterative Path Planner (IPP). 16

Figure 2.7: Manhattan-Like RRT (ML-RTT). 17

Figure 3.1: Swept Volume (SV). 19

Figure 3.2: Wrapping. 20

Figure 3.3: Voxels used for wrapping. 22

Figure 3.4: Implicit distance calculation using Schroeder's inverse. 22

Figure 3.5: Offsets for implicit surfaces. 23

Figure 3.6: Directed distance fields. 24

Figure 3.7: Problems with regular path sampling. 26

Figure 3.8: Directed distance field gathering. 27

Figure 3.9: Grid division. 27

Figure 3.10: Surface extraction. 28

Figure 3.11: Walking along the grid. 30

Figure 3.12: Graph building. When the box on the left is placed in the grid, 30

Figure 3.13: Distance field sampling error. 31

Figure 3.14: Naive manipulation of directed distance fields. 34

Figure 3.15: Positive offset using spheres. 35

Figure 3.16: Positive offset error. 35

Figure 3.17: Positive offset integrated into surface extraction. 36

Figure 3.18: Negative offset using spheres. 37

Figure 3.19: Negative offset surface extraction. 38

Figure 3.20: Exhaust test case. 40
Figure 3.21: Seat test case. 40
Figure 3.22: Human test case. 41
Figure 3.23: Flat surface example. 41
Figure 3.24: Offset dragons. 42
Figure 3.25: Offset motors. 43
Figure 3.26: Compressor test case. 44
Figure 3.27: Offset compressors. 44

Figure 4.1: Testing polyhedrons. 50
Figure 4.2: Framework architecture. 51
Figure 4.3: Proximity test dispatch mechanism. 52
Figure 4.4: Element relations and traversal methods. 53
Figure 4.5: Compressed storage scheme for OBBs. 54
Figure 4.6: Scene trees. 56
Figure 4.7: Aggregate test tree. 57
Figure 4.8: HRP-2, a humanoid robot. 58
Figure 4.9: Occupancy grid. 59
Figure 4.10: Voxel hierarchy. 60
Figure 4.11: Environment discovery in three steps. 61

Figure 5.1: Penetration Distance. 67
Figure 5.2: Movement equivalent to upper bound of PD. 67
Figure 5.3: Example disassembly problem with active obstacles. 68
Figure 5.4: Active obstacle movement. 69
Figure 5.5: Taking active obstacles into account. 70
Figure 5.6: Ignoring active obstacle distances. 71
Figure 5.7: Discontinuous interpolations give better performance. 72
Figure 5.8: Classic ML-RRT. 73
Figure 5.9: ML-RRT with discontinuous interpolation. 74
Figure 5.10: Collision test with superposition operator. 75
Figure 5.11: Handling multiple obstacles. 75
Figure 5.12: Incompleteness of the superposition operator. 77
Figure 5.13: Generalized separation distance. 78
Figure 5.14: Dynamic play. 81
Figure 5.15: Windshield wiper test case. 82

Figure 5.16: Exhaust test case. 82

Figure 5.17: Starter test case. 83

Figure 5.18: Shocks test case. 83

Table of Algorithms

Algorithm 3.1: Fast Marching Method adapted to recognize surface points. 28

Algorithm 4.1: Simple polyhedron-polyhedron collision procedure. 50

Algorithm 4.2: Generic tree descent. 63

Algorithm 5.1: ML-RRT for active obstacles. 73

Algorithm 5.2: Modified ML-RRT for discontinuous interpolations. 74

Algorithm 5.3: Boolean soft collision detection routine. 76

Algorithm 5.4: Routine for finding the maximum PD estimate for a path. 79

List of Tables

Table 3.1: Swept Volume Test Models. 39

Table 3.2: Swept Volume Experimental Results. 39

Table 3.3: Offset Test Models. 42

Table 3.4: Offset Experimental Results. 43

Table 4.1: Time and Memory Performance. 61

Table 5.1: Active Obstacle Test Models. 84

Table 5.2: Active Obstacle Experimental Results. 84

"Intelligence is whatever machines haven't done yet."

Larry Tesler

1 Introduction

To many people, the word "robot" conjures up images of space exploration, relentless killing machines, or human-shaped automatons with synthesized voices. From its very beginnings, however, the field of robotics has been driven as much by the desire to efficiently manufacture products as to create artificial beings. Such robots may never need to explore unknown environments, ponder human emotion, or endure mutilation without flinching. They may not even resemble any natural creature. Instead, they are simply asked to manipulate cumbersome objects, over and over again, with exacting precision and speed. What industrial robotics lacks in drama, it makes up in utility.

This PhD research was performed under a French grant entitled "Conventions Industrielles de Formation par la Recherche"* (CIFRE). It is a joint initiative between a company, a student, and a laboratory, in which the student shares work time between the company and the lab, exploring a subject that is of interest to both. Through subsidizing a three-year contract, the government hopes to strengthen links between industry and academics along with stimulating applied research.

The company for which I have been working for is called Kineo Computer-Aided Motion (Kineo CAM), a startup specializing in path planning for industrial robotics. The researchers who founded it wanted to bridge the technological gap between what was considered "state-of-the-art" in industry and in motion planning research. In keeping with its legacy, Kineo CAM actively strives to deliver sophisticated algorithms as commercial products. In this manuscript, I will attempt to present my doctoral work in this same context— as applied research meant to answer practical and pressing needs of industrial production.

* "Industrial Agreement for Training through Research"

1.1 Brief History of Industrial Robotics

Whereas a general definition of "robot" is difficult to pin down[†], an industrial robot can be reasonably considered as an "automatically controlled, reprogrammable, multipurpose manipulator programmable in three or more axes" [ISO 1994]. The key concepts here, those that separate robots from most other machines, are *manipulation* and *programmability*.

In this spirit, the first well-known industrial robot may be the automated loom conceived in the early 19th century. Joseph Jacquard designed the machine to read and follow instructions encoded on punch cards, enabling the same machine to produce an infinity of different weaves without constant human guidance.

Jumping forward to the 1940s, John Parson pioneered the field of Numerically Controlled (NC) machining, in which an automatic device handles tools such as lathes or drills in order to reliably manufacture large quantities of parts. The input to the robot (early versions used punch cards once again) is a series of positions between which to maneuver the tool. One of the most interested clients in the technology was the US Air Force, who needed to mass-produce freeform surfaces.

A foundation of modern industrial robotics, the robotic arm wasn't invented until the 1950s. The first version was called the Unimate, and was put into use at a General Motors plant, where it extracted hot die castings and performed spot welding. The Unimate was succeeded by the Stanford Arm, which replaced its predecessor's hydraulics with electric motors, and added both optical and contact sensors. In both of these manipulators, the joints were connected by chains to motors installed in the base. In the 1980s, Takeo Kanade pioneered direct drive mechanics, in which the motors are put directly in the joints themselves, reducing friction and achieving higher accuracy and speed.

Industrial robotics has continued to evolve along these lines, encompassing such applications as painting, ironing, packaging, and testing. In "pick and place", a robot selects an object from a pick-up location and transports it to a destination, possibly changing its orientation along the way. An assembly task involves fitting parts together, and disassembly refers to extracting parts out.

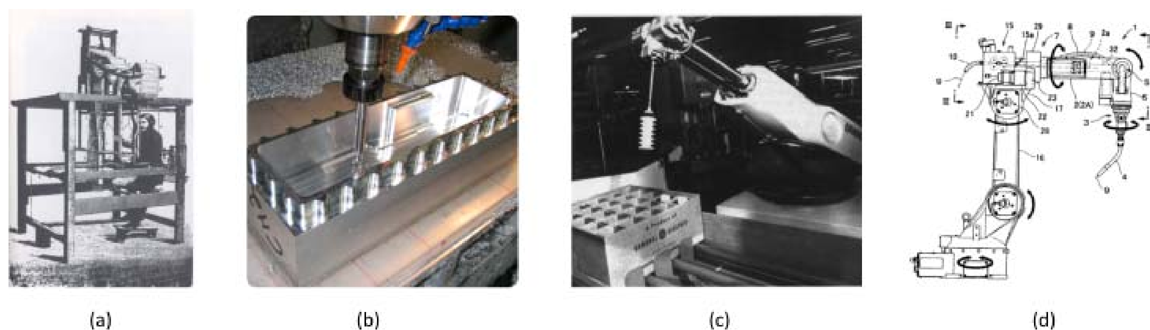

Figure 1.1: Historical landmarks in industrial robotics. From left to right, Jacquard's loom (a), an example of NC machining (b), the UNIMATE in a General Motors plant (c), and a patent drawing for a modern industrial robotic arm (d).

† Joseph Engelberger, founder of Unimation, Inc., the world's first industrial robot manufacturer, once said "I can't define a robot, but I know one when I see one."

1.2 Role of Motion Planning

Motion planning fits into a larger concept called computer planning [LaValle 2006]. Roughly put, computer planning is concerned with fulfilling a high-level request through a series of lower-level steps. In motion planning, this translates to finding a path for an object between two given positions while respecting a certain set of constraints. The most common constraint is avoiding collisions with other objects, but other possibilities include respecting robotic joint limits and moving along curved paths, as wheeled vehicles often do.

Initially, it was thought that all of computer planning could be fit under the same tent, based purely on a limited set of logical principles. Encouraged by successes in proving logical theorems, solving geometry and algebra puzzles, and interpreting instructions written in English, Artificial Intelligence (AI) researches in the 1960s predicated that within a generation computers and the robots that they controlled would have the intelligence and physical capacity of humans.

These rosy visions began to dissolve in the 1970s, as it became clear that AI wasn't going to fulfill its promises so easily. After several large setbacks in both financing and respect, the problem became known as Moravec's Paradox:

> *It is comparatively easy to make computers exhibit adult level performance on intelligence tests or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility…*
>
> *Encoded in the large, highly evolved sensory and motor portions of the human brain is a billion years of experience about the nature of the world and how to survive in it. The deliberate process we call reasoning is, I believe, the thinnest veneer of human thought, effective only because it is supported by this much older and much powerful, though usually unconscious, sensorimotor knowledge. We are all prodigious olympians in perceptual and motor areas, so good that we make the difficult look easy. Abstract thought, though, is a new trick, perhaps less than 100 thousand years old. We have not yet mastered it. It is not all that intrinsically difficult; it just seems so when we do it.* [Moravec 1988]

If the field of AI has regained some of the ground it had lost in those days, it has definitely changed its approach as well. Researchers now consider each problem in its own right, and rely on the unyielding advancement of computational power where pure mathematical analysis has proved insufficient. Computers are now capable of beating chess masters, recognizing human speech, and finding a needle in the haystack of the World Wide Web. On the robotics front, great progress has been made in autonomous cars and planes, and humanoid robots can navigate cluttered environments on their own two legs.

Likewise, motion planning has split off into its own field. Rather than reasoning "logically" about a posed problem, recent success derives from probabilistic approaches, in which the computer basically tries random plans until it finds one that works. This is made possible by the sheer number of motions that a modern computer can test in a single second[‡].

[‡] As of this writing, typical collision detection speed for a multiple axis robot in a complex environment is around 1 ms per test.

Motion planners are so successful that they can serve as building blocks in larger computer planning systems. When an industrial robot wants to find convenient places to put objects down, maneuver through changing environments, or find sequences in which to disassemble a complex part, it can turn to a motion planner to solve part of the problem, and use the results to plan larger actions.

Where motion planners cannot succeed on their own, they can work in cooperation with humans. In these systems, a human operator can suggest ideas to the computer, often through force-feedback devices, and the planner plays off those ideas as well as its own. Motion planning can also be applied to human movement. By taking ergonomics into account, motion planners can determine efficient ways for factory workers to carry out their tasks without causing long-term pain or disability.

1.3 Challenges in Industrial Motion Planning

Advancements in robotics technology benefit industry as much as any other section of the field, and fortunately there is significant cross-fertilization of ideas between researchers working in different application areas. Nevertheless, the demands, constraints, and assumptions of industrial robotics developers are not systematically shared by their counterparts elsewhere. For example, an industrial arm that manipulates identical parts all day long probably does not need vision, whereas a space-exploring robot would. But while the space-exploring robot could take its time deciding how to act, the arm cannot afford to be patient. The tradeoffs for design criteria such as performance, safety, energy consumption and weight are naturally shaped by the application in question. In turn, the problems posed to industrial motion planners embody the choices made.

Nowadays, most parts to be manufactured are designed directly on the computer. High-resolution models are therefore available. While guaranteeing accuracy, the use of these models quickly leads to an explosion in the amount of geometry data to be treated. A second side-effect of these models comes from their most common format— “polygon soups”. Polygon soups are simply a set of unorganized surfaces, commonly triangles. In contrast, most geometry data in other robotics domains describe volumes. The distinction may appear trivial, but volumes provide extra information that can be, and often is, exploited by computational geometry algorithms. Using polygon soups limits the choice and design of algorithms for motion planning (**Figure 1.2**).

As a general rule, an important design criterion for industrial motion planning is practicality. Practical can mean fast, because impossibly slow programs are unusable, but performance does not need to be real-time. Indeed, most motion planning is done off-line, meaning that the actual programming of the robot will be done after the motion has been found and analyzed. More often, practicality refers to ease of use. Algorithms with many critical parameters, in which setting these parameters incorrectly either finds a poor plan or fails to find one altogether, are by definition difficult to use. Given that users of industrial motion planning are rarely roboticists themselves, designers of these planners should not expect the users to be expert enough to set such parameters, unless they are fundamental properties of the robot or its environment.

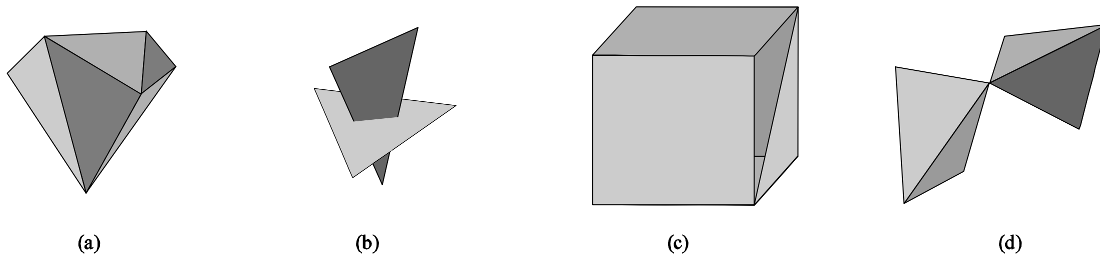

Figure 1.2: Polygon soup models. For performance reasons, three-dimensional objects are often treated as *polygon soups*, basically an organized collection of triangles. These models are called well-formed if they follow certain rules that make their interpretation straightforward (a). Since almost "anything goes" in polygon soups, deficiencies can arise that inhibit computational geometry algorithms. For example, triangles must share vertices and edges, and cannot cross each other anywhere else (b). Also, shapes must be volumes, and can't be missing any triangles (c). Finally, two shapes cannot meet each other at a single point or edge (d).

In short, industrial robotics has its own requirements and constraints that set it apart from other disciplines. The challenges of working in this field played an important role in determining how our work developed throughout this PhD.

1.4 Presentation of this Work

In this doctoral thesis, we develop geometric operators for motion planning within an industrial context. The operators in question are collision detection, volume wrapping, forced passages, and shrinking. For each of these operators, there is a fundamental interest for industrial applications. In some cases, equivalent operators already exist, but do not satisfy industrial constraints.

1.4.1 Contributions

Our first contribution is an operator for finding a volume that completely surrounds one or more geometric objects as tightly as possible. Imagine rolling plastic wrap around a spiky object such as a pineapple. The result preserves all the detail of the exterior, while ignoring the complex inner workings. Since it can be equally applied to motions, this operator can generate swept volumes. We then develop extensions to the wrapping operator that modify the size of the final produced volumes, either inflating or deflating them much like a balloon.

Probabilistic motion planning relies heavily on a fast, accurate, and robust collision detection operator. As the number of uses for motion planning multiplies, it is applied to a variety of different geometry types. Our second contribution is developing a framework upon which these geometry types can be seamlessly integrated. The result is an improved development process, reusing the fundamental logic behind collision detection while allowing for the many optimizations that may be asked of a collision detector.

Our third contribution is a process and an operator for motion planning in collision. Usually, collisions between the robot and its environment are to be strictly avoided during motion planning. But in several cases, such collisions are unavoidable. The solution may involve forcing an object through a narrow passage, or simulating slightly deformable objects. Even in the absence of such cases, it serves as a useful tool for analyzing why collision-free path planning failed. We propose two methods and demonstrate their complementary properties.

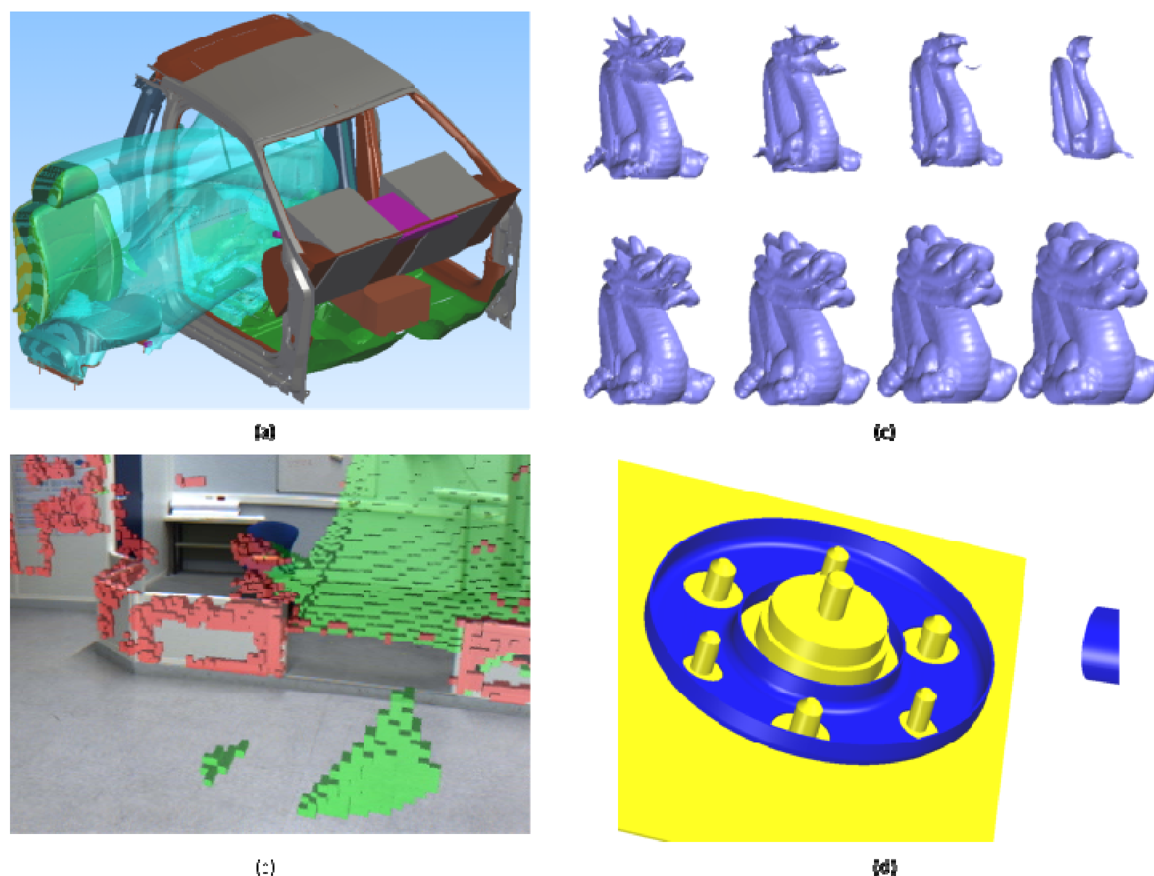

Figure 1.3: Contributions. In this doctoral work, we develop a wrapping operator that works on polygon soup models (a), and introduce inflation and deflation effects that change the size of the wrapping (b). Next, we develop a collision detection framework that handles heterogeneous geometry types such as polyhedrons and voxels (c). Finally, we develop methods to perform motion planning in collision to handle difficult forced passage problems (d).

1.4.2 Organization

Since motion planning represents an important thread running throughout our work, we found it important to introduce it in a technical context. Chapter 2 discusses its origins and briefly touches on the state-of-the-art in the field. Particular attention is paid to those algorithms that are directly used or built upon in our work.

Chapters 3 through 5 present the four major contributions of this doctoral thesis: wrapping, extensions for size manipulation, a generalized collision detection operator, and path planning in collision. Necessary background concepts are presented at the beginning for each chapter.

Finally, we conclude this thesis in Chapter 6 by discussing possibilities for future development of these operators as well as others in the context of industrial motion planning.

1.4.3 Publications

The following publications are associated with this work:

J. C. HIMMELSTEIN, E. FERRÉ, and J.-P. LAUMOND, "Swept Volume approximation of polygon soups," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2007, pp. 4854-4860.

J.C HIMMELSTEIN, A. NAKHAEI, G. GINIOUX, F. LAMIRAUX, E. FERRÉ, AND J.-P. LAUMOND, "Efficient Architecture for Collision Detection between Heterogeneous Data Structures," to appear at the *10th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, Hanoi, Vietnam, 17-20 December, 2008.

J.C. HIMMELSTEIN, E. FERRÉ, AND J.P. LAUMOND, "Swept Volume approximation of polygon soups," accepted by *IEEE Transactions on Automation Science and Engineering (T-ASE)*.

J.C. HIMMELSTEIN, E. FERRÉ, AND J.P. LAUMOND, "'Teleportation'-Based Motion Planner for Design Error Analysis," submitted to *2009 IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 12 - 17, 2009.

"Randomization is, evidently, a euphemism whose real meaning is: deliberately throwing away relevant information when it becomes too complicated for us to handle."

Edwin Thompson Jayne

2 Motion Planning

As explained in the introduction, motion planning plays a critical role in robotics. It can be used by itself (e.g. to help a robot navigate its environment), but also as a primitive for more complex planners (e.g. finding a disassembly sequence for a mechanical system). As the entirety of this doctoral work relates to motion planning in one way or another, we have devoted this chapter to presenting its development, paying special attention to material that is used directly by our work. For a more in-depth discussion and survey of the field, we refer readers to Lavalle's book [LaValle 2006].

The goal of motion planning is to find a movement that brings a robot from one configuration to another, while respecting certain constraints. Here we use the term *robot* loosely— it can consist of one or more objects, moving together or separately. A *configuration* is a complete description of the position of the robot. Since one of the most common constraints is that the robot must avoid colliding with its environment, we will consider that the environment is full of *obstacles* that the robot is not allowed to touch. Motion planning is also referred to as *path planning*, because the goal is to find a path in space for the robot to follow.

2.1 Configuration Space

Shakey, developed from 1966 through 1972, was one of the first robots to successfully navigate a room and accomplish simple tasks such as moving objects [Nilsson 1984]. It relied on laser range finders, bump sensors, and a TV camera to detect its environment (**Figure 2.1**). A general planner handled motion along with task sequences through logical reasoning about its environment. This was made possible through simplistic assumptions about environment geometry; all objects were represented as rectangles or circles on a 2D grid. In spite of being a very impressive accomplishment, researchers soon found that this approach to motion planning did not scale.

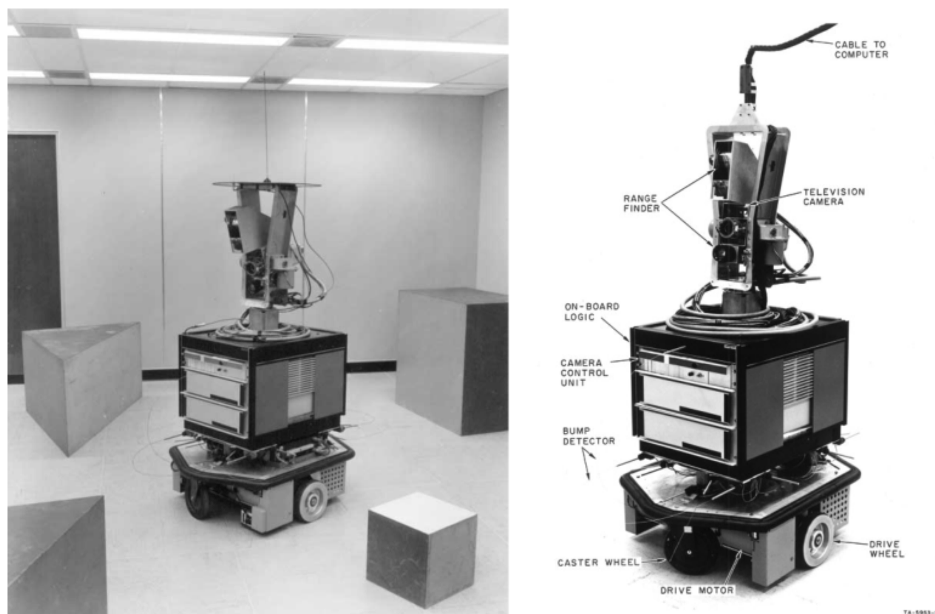

Figure 2.1: Shakey the robot. Shakey, developed by the Stanford Research Institute, was a milestone in robotics and computer planning research. It was capable of modeling a simple environment made out of blocks (left). Given a high-level task, it could reason spatially and logically about how to manipulate the environment to achieve its goal. It used a number of different sensors and motors to carry out the task (right).

Indeed, calculating collision-free paths with even slightly more complex geometry proved to be a difficult issue. Mathematically, this puzzle is referred to as the piano movers' problem, introduced by Schwartz and Sharir [Schwartz, et al. 1986]. A major insight, that of using *configuration space*, was advanced by Lozano-Pérez, originally for robot grasping tasks [Lozano-Pérez 1976]. It turned out to have a more general formulation, one that applies equally well to all types of robot geometry, including articulated manipulator arms and multiple robots [Lozano-Pérez 1983].

The idea is to shrink the robot to a point, while growing the obstacles by the same amount. Now we can plan for the movement of a point rather than a body. Since the relative sizes of the objects have stayed the same, the two problems are equivalent. This concept is easy to visualize in two dimensions, but becomes more difficult once rotations are considered. To generalize it, we define the *configuration space* C as the set of all possible configurations that the robot can attain. Each point in C defines a configuration, which is composed of a set of values— one for each degree-of-freedom (DOF) of the robot. The configuration space therefore has as many dimensions as the robot has DOFs. For example, a freely-rotating object in 3D has six translation DOFs, and three rotation DOFs. Depending on the robot under consideration, certain DOFs may be bounded, causing C to be bounded in those dimensions as well.

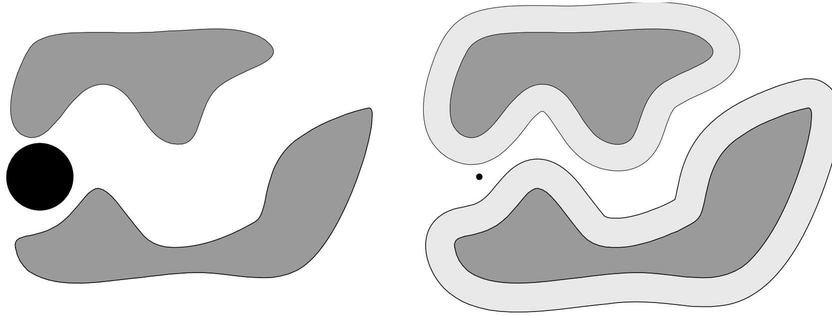

Figure 2.2: Configuration space for translations. Here the "robot" is the circle, and the other objects are obstacles (left). If only translations are taken into account, then we can picture configuration space by enlarging the obstacles by the size (or radius, in this case) of the robot, while simultaneously shrinking the robot to a point (right). The two situations are equivalent, but the second is both simpler to solve and generalizable to a large class of motion planning problems.

Placing the robot in some of these configurations will lead to a collision with the environment, while for others it will not. To distinguish between the two cases, we can split C into two regions: C_{obs} contains colliding configurations, and C_{free} the rest. Formally, given a robot $\mathcal{A}$ so that $\mathcal{A}(q)$ places it in a configuration q, as well as a set of obstacles $\mathcal{O}$, then

$$C_{obs} = \{q \in C : \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\} \quad (2.1)$$

$$C_{free} = C \setminus C_{obs} \quad (2.2)$$

Now we can reformulate the motion planning problem as finding a continuous curve or path in C_{free} that connects the starting configuration q_{init} to the goal configuration q_{end}. Such a path can be thought of as a function $\tau : [0,1] \to q$ which translates a number on the unit interval to a configuration for the robot. We can call a path *valid* if it stays within C_{free}

$$valid(\tau) \Longrightarrow \nexists t(t \in [0,1] \wedge \tau(t) \in C_{obs}) \quad (2.3)$$

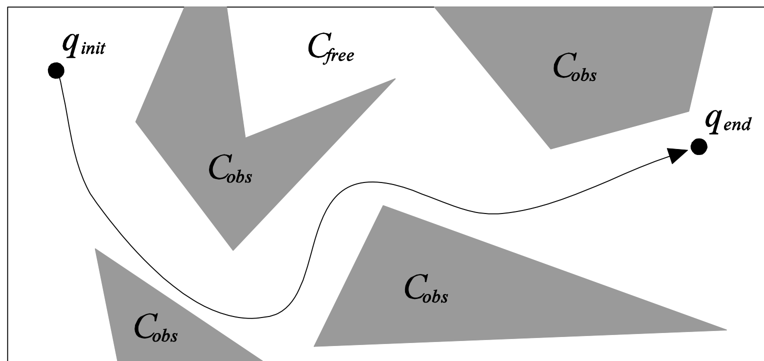

Figure 2.3: Motion planning in configuration space. Obstacles are part of C_{obs}, meaning that placing the robot in a configuration within that region would lead to collision. The remaining space represents C_{free}, the collision-free region. The goal of the path planning is to connect the initial configuration q_{init} to the ending configuration q_{end} with a continuous curve, called a path.

Unfortunately, since each additional DOF represents a whole dimension in C, the fastest complete algorithm that applies to general polyhedral objects has time-complexity exponential in the number of DOFs [Canny 1988]. In practice, solving the motion planning problem for

arbitrary environments and robots is prohibitively time-consuming [Svestka and Overmars 1998].

To escape what appears to be an analytical dead-end, algorithms have been developed that take a different tack. Using heuristics, they search through C, looking for paths that link q_{init} to q_{end}. Such searches are not formally *complete*, meaning that they are not guaranteed to solve a given path planning problem or even determine if a solution exists. Nevertheless, certain algorithms can be proven to be *probabilistically complete*, signifying that they are guaranteed to find a solution in finite time if one exists. In other terms, the probability of finding a solution converges to 1 as time approaches infinity. Such approaches include genetic algorithms, simulated annealing, and dynamic-graph search methods. The two most successful algorithms of this type, however, are the Probabilistic Roadmap Planner (PRM)* and the Rapidly-exploring Random Tree (RRT). Although our work draws almost exclusively upon the latter, understanding the PRM is nevertheless useful for understanding how such algorithms function. For brevity, we present only a simplified version of each, meant to help the reader understand the general techniques without burdening him with excessive detail.

2.2 Probabilistic Roadmap Planner

The principal data structure in the PRM is a *roadmap*, a graph in which the nodes represent configurations in C_{free} and the edges are simple paths (also called direct paths) between them, resting in C_{free}. Algorithmically, the PRM is split into two phases. In the *roadmap construction phase*, a random configuration q_{rand} is generated and tested for collision with the environment. If no collision is found, then the configuration is added to the roadmap. In order to link the new node with others in the roadmap, the planner picks a node q_{near} that is close to q_{rand}, and uses a *local planner* to connect the two if possible†. The simplest local planner is one that connects the two using a straight line segment, which in configuration space corresponds to a linear interpolation between the two configurations‡. In any case, the planner must verify that the direct path is collision-free before adding it to the roadmap. The planner may consider multiple close nodes before generating another random configuration and repeating the process. Since the goal of the construction phase is to generate a good sampling of C, this process generally continues until a certain number of nodes have been generated.

The actual path planning problem is solved during the *query phase*. Given q_{init} and q_{end}, the planner attempts to link them to roadmap nodes q'_{init} and q'_{end}, which themselves are connected through edges in the roadmap. The output path is simply the sequence of edges connecting $\{q_{init}, q'_{init}, \ldots, q'_{end}, q_{end}\}$ (**Figure 2.4**). Given that it may be possible to connect q_{init}

---

* The PRM is referred to elsewhere as the Probabilistic Path Planner (PPP).

† The notion of "close" is defined by a distance metric between two configurations. The simplest such metric is Euclidian distance in configuration space (i.e. $d = |q_1 - q_0|$). Depending on the robot, this may not correspond to a realistic measurement of movement from configuration to another. Additionally, rotations may require extra attention, as the distance that a point on the robot moves is likely larger than a simple difference in radians would indicate.

‡ Some robots are non-holonomic, meaning that they cannot move freely from one configuration to any other, and thus admit only certain paths (e.g. car-like vehicles). Such robots require a special local planner.

to multiple roadmap nodes, and that roadmap nodes might be connected through more than one edge sequence, multiple paths may be possible, in which case the planner chooses the shortest. If the PRM is unable to find a solution, it may be that the problem is impossible, but it may also be that it has not added enough random configurations to the roadmap. With only probabilistic completeness as a guide, it is practically impossible to tell the difference without outside analysis.

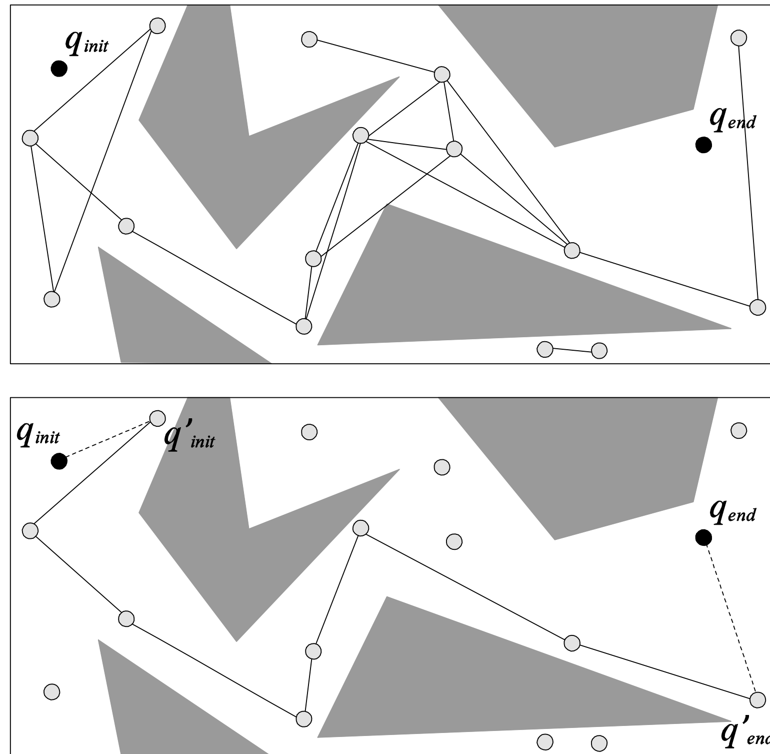

Figure 2.4: Probabilistic Roadmap Planner (PRM). The PRM is split into two phases. In the roadmap construction phase (top), random configurations in C_{free} are added to the roadmap and linked to existing nodes. In the query phase (bottom), q_{init} and q_{end} are linked to nearby roadmap nodes q'_{init} and q'_{end} and a graph search is performed to link the two using existing roadmap edges.

Since the PRM requires a pre-computation step, it is more suited to solving a sequence of problems in the same environment. This is called the *multiple-query* model. The RRT takes a different approach, generating the roadmap anew for each path planning problem. This *single-query* model has proven to be particularly efficient, in part by restricting its focus on only the regions of C that interest the user.

2.3 Rapidly-Exploring Random Tree

The RRT was originally developed by Lavalle in 1998 for robots with non-holonomic constraints [LaValle 1998]. He cites the difficulty designing a local planner for connecting the PRM roadmap nodes in an efficient manner as a reason for experimenting with a *diffusion* technique. In diffusion, the roadmap is rooted at q_{init} and the local planner is called upon to generate new nodes that branch of the existing roadmap. The roadmap is therefore is therefore connected by definition, and all direct paths are acceptable. Eventually, the roadmap should grow to approach nearly all configurations in C_{free}, including q_{end}, at which point the solution path is simply the

roadmap edges connecting q_{init} to q_{end}. Typically, diffusion does contain a pre-computation step as with the PRM.

The RRT is not the first example of a diffusion approach in probabilistic path planning. The Ariadne's Clew algorithm separated out a global exploration phase that grows the roadmap without bias towards the goal from a local planning method that attempts to reach the goal from a roadmap node [Mazer, et al. 1998]. The Expansive-space planner takes another tack, generating new configurations in the local neighborhoods of existing roadmap nodes. The planner both picks nodes and adds new configurations with a probability inversely proportional to the number of neighbors [Hsu 2000].

But although it was created for non-holonomic robots, the RRT has demonstrated a high level of efficiency and robustness for many types of path planning problems. The secret to its effectiveness derives from the way that it efficiently explores the configuration space. The algorithm can be broken down into the following steps:

1) *Shoot* – Choose q_{rand} through randomly sampling in C.
2) *Pick* – Find the closet node q_{near} in the roadmap to q_{rand}, using some distance metric.
3) *Extend* – Starting at q_{near}, move a distance x in the direction of q_{rand} to find q'_{rand}. If the direct path from q_{near} to q'_{rand} is collision-free, add it and q'_{rand} to the roadmap.

Starting with only q_{init} in the roadmap, the RRT repeats this process until it reaches q_{end} (**Figure 2.5**). Not only has the RRT been shown to be probabilistically complete, but it converges very quickly to a solution. An important difference from previous techniques such as random walks is the way in which q_{near} is chosen to best suit q_{rand}, and not the other way around[§]. This subtle change in tactic gives rise to a tendency to explore unknown areas of C, while scattering nodes in a way that approaches the sampling distribution.

Subsequent work on this algorithm lead to RRT-Connect, which introduced several enhancements [Kuffner and LaValle 2000]. First, instead of waiting for the roadmap to approach q_{end} by itself, a planner attempts to link q_{end} to the tree after each iteration. In addition, the extend step does not stop moving towards q_{rand} until an obstacle is hit. Finally, two trees are grown in parallel: one rooted at q_{init}, and the other at q_{end}. This last enhancement works especially well if both q_{init} and q_{end} are in tight spaces.

The rest of this chapter is devoted to discussing two variants of the RRT that are used in our work: the Iterative Path Planner, and the Manhattan-Like Path Planner. The first is targeted at industrial problems, where it produces especially good results. The second is intended for molecular path planning, and allows for obstacle movement.

[§] A more intuitive order calls for for the new node q_{rand} to be chosen in a small neighborhood of the existing node q_{near}. Even in the presence of heuristics favoring little-chosen nodes for q_{near}, this leads to an over-sampling of the "known" portions of C_{free}. The RRT elegantly side-steps this problem by choosing the nodes in the opposite order.

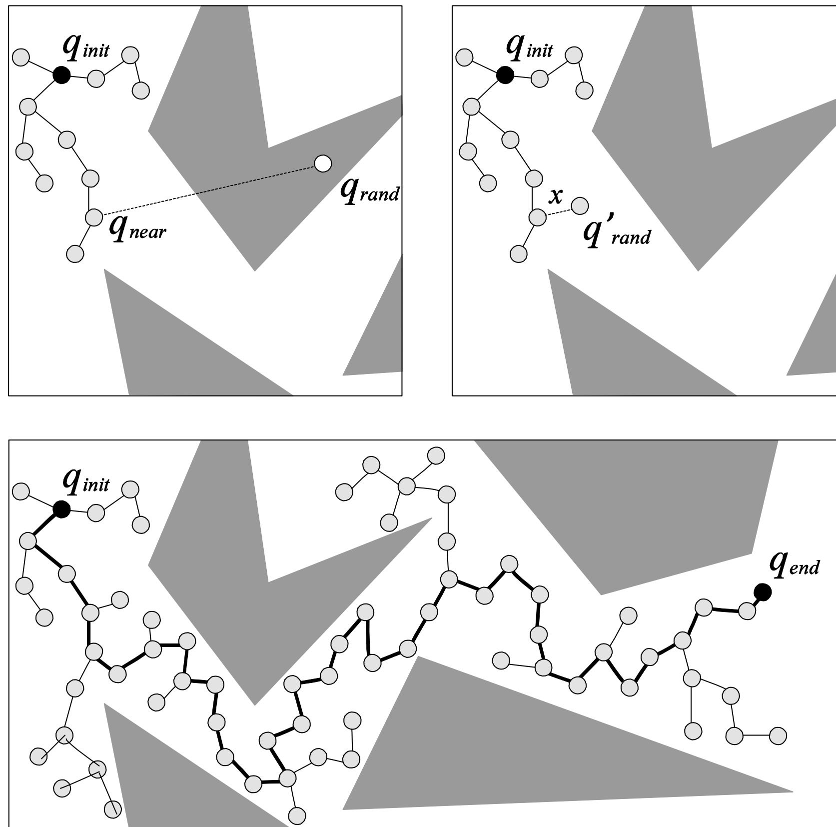

Figure 2.5: Rapidly-exploring Random Tree (RRT). The roadmap is rooted at q_{init}, and is grown through three-step iterations. In the *Shoot* step, a configuration q_{rand} is chosen randomly in the configuration space. Proceeding to the *Pick* step, the closest node in the roadmap to q_{rand} is selected, and is designated q_{near} (top left). Finally, a new configuration q'_{rand} is generated by moving along the vector from q_{near} to q_{rand} in the *Extend* step. If q'_{rand} is in C_{free}, it is added to the roadmap (top right). With this scheme, the tree expands to explore all of C_{free}, until reaching q_{end}. The final path is the sequence of links between the roadmap node linking q_{init} to q_{end} (in bold on bottom).

2.3.1 Iterative Path Planner

Product Lifecycle Management (PLM) tracks design information about a mechanical component, from cradle to grave. Designers are interested in studying the feasibility of assembling mechanical systems as well as disassembling them for maintenance. Path planning can be used to validate and suggest assembling motions for both human and robot operators.

The Iterative Path Planner (IPP) was created to handle such PLM cases [Ferré and Laumond 2004] without requiring parameter tuning. Based on the RRT, it introduces an additional variable, called *dynamic penetration*. The dynamic penetration value refers to potential overlaps in C_{obs} that may occur when adding an edge in the roadmap. By starting with a high dynamic penetration, and lowering it over a number of iterations, IPP can find *draft paths* that are colliding but nonetheless provide some guidance to where later refined paths might try to pass (**Figure 2.6**).

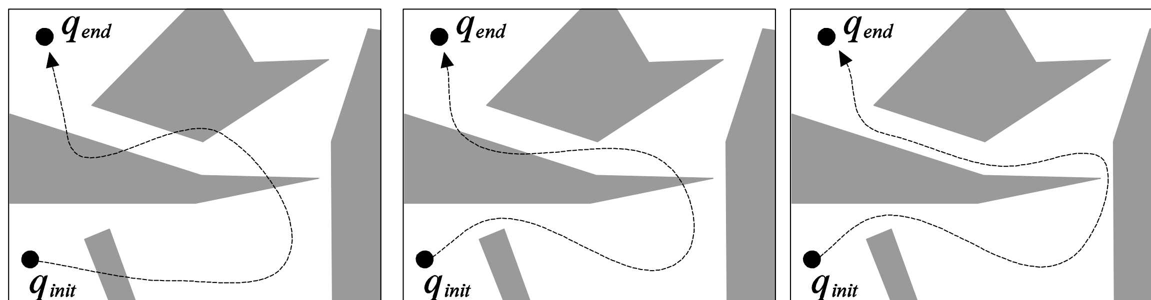

Figure 2.6: Iterative Path Planner (IPP). With an additional variable called *dynamic penetration*, IPP breaks down difficult problems into several steps. Starting with a high dynamic penetration, it finds a path that, while in collision, serves as a guide for further refinement (left). In subsequent steps, the dynamic penetration is lowered, forcing the planner to discover paths that collide less (center). Finally, when the dynamic penetration has been lowered to the user-defined level, the output path will be collision-free (right).

Dynamic penetration derives from the method used to validate direct paths in the roadmap. Instead of uniformly sampling the direct path and checking each configuration for collision, IPP assures that no point on the robot sweeps out a movement longer than a certain distance. Given a dynamic penetration value of ε, IPP finds the first configuration along the direct path where a point on the robot has moved up to 2ε. If this point is in C_{free}, then IPP advances along the direct path another 2ε, and so on until either a collision is found or the path is validated.

Although collisions may still occur between tested configurations, it can now be bounded. In the worst case, the robot entered in collision up to a distance ε and then returned to C_{free}. IPP adjusts this dynamic penetration value through an iterative process. In the first iteration, a very high dynamic penetration is chosen, one that allows the planner to quickly find a path to q_{end}. Once this is done, the penetration is drastically reduced, and the existing roadmap is reevaluated. Any segments that do not pass the stricter collision test are rejected, and the others become part of the new roadmap. This process is repeated until the penetration reaches a user-defined value.

Another modification to the classic RRT is the handling of the *Shoot* stage. PLM assembly cases often involve *free-flyers*, i.e. objects that are allowed to move freely in space. In contrast with rooted manipulator arms, the DOFs that correspond with free-flyer translation are unbounded, meaning that they can take any value in $\mathbb{R}$. This implies in turn that C is infinitely large. Practically speaking, it is difficult to choose a random configuration uniformly in an infinite space.

To work around this problem, IPP shoots around a local neighborhood of existing roadmap nodes. The neighborhood, consisting of one or more nodes, forms a finite subspace $C_{rdm} \subset C$. IPP chooses a configuration with a Gaussian distribution around C_{rdm} in order to allow configurations that are outside of, but not infinitely far from, the given neighborhood.

2.3.2 Manhattan-Like RRT

Certain molecular systems can be studied using path planning techniques. One important problem in this domain is determining if and how ligands (typically small substances that bind to biomolecules), can access target sites deep within large proteins. In particular, this

understanding can guide the development new of drugs that bond to protein receptor sites [Latombe 1999]. As the protein and ligand can be modeled as highly-articulated mechanical objects, ligand access can be posed as a motion planning problem.

In order to model molecules as mechanical systems, they are analyzed for the structure of their molecular bonds. Groups that are rigidly bonded to each other can be considered as rigid bodies. These rigids are linked by articulations that correspond to bond torsions. Collision detection is performed by considering only the van der Waals force, so that atoms are represented as spheres.

The Manhattan-Like RRT (ML-RRT) is destined for studying these kinds of problems [Cortés, et al. 2007]. The protein backbone is represented as a large rigid body, from which the ligand must escape. In doing so, it might push on the side-chains, which open up like barn doors before it. The number of DOFs in this scenario is very large, considerably slowing down even an optimized RRT.

In order to conquer the complexity, the ML-RRT divides the DOFs into two groups: *active DOFs* which include the ligand, and *passive DOFs* which include the side-chains. Each RRT iteration is split into two steps. First, only the active DOFs are planned for. If the extend step fails, however, the offending direct path is analyzed to see which objects were found in collision with the ligand. If the side-chains are the only objects in collision, then new configurations for them are chosen by sampling in a local neighborhood around their current positions (**Figure 2.7**).

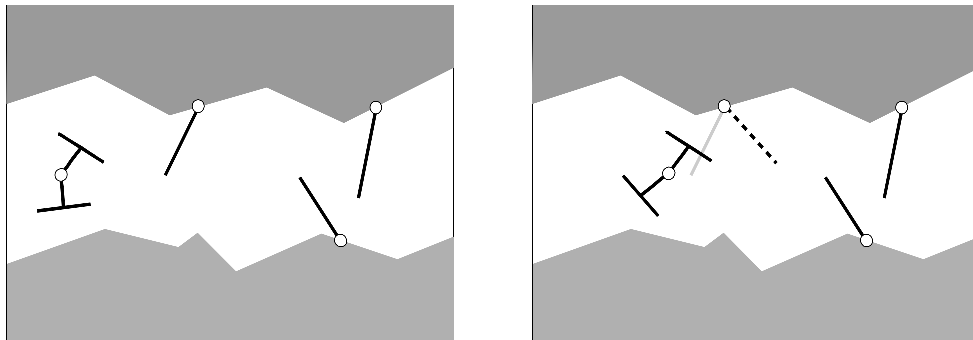

Figure 2.7: Manhattan-Like RRT (ML-RTT). A ligand can be modeled as an articulated object (here the joints are represented as circles). The protein within which it finds itself is composed of rigid objects (the grey areas) as well articulated side-chains. The ML-RTT considers the ligand DOFs as active, meaning that it plans for them first (the object shaped like an H on the left). If a collision occurs with one or more side-chains, however, these side-chain positions are perturbed. If the new position avoids collision, then it is added to the roadmap (the dashed line on the right).

2.4 Remarks for this Work

Having presented motion planning in general, it is important to understand where this doctoral work is situated within the field. Generally, we consider only static environments, in which there are no moving obstacles, other than those that the planner might move for its own purposes. Furthermore, we do not consider sensing, which typically introduces in unknowns and stochastic methods. By limiting our focus in this way, we are able to bring greater attention to the industrial problems that motivate this work.

This chapter is meant as an introduction to the field of motion planning, which provides the context for this doctoral work. In this next chapter, we discuss a tool for analyzing the output of the motion planning algorithms.

"Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away."

Antoine de Saint-Exupery

3 Wrapped Volumes

Motion planning is not an end unto itself. The larger goal may indeed be to program a robot to perform the movement, but a great number of systems wish to analyze the found path. Part of this analysis may require visualizing the motion. A natural way to view paths is to "play" them, in the form of an animation, in which the robot carries out the prescribed movement. However useful this form may be for analyzing the *dynamics* of the movement, we would yet sometimes like to visualize the *extent* of the movement.

3.1 Swept Volumes and Wrapping

As a matter of fact, the Swept Volume (SV) lets us do exactly that. It describes the totality of space touched by an object (or robot) as it is pulled along its trajectory (**Figure 3.1**). In a sense, the SV "flattens" the robot motion, removing the time dimension. That is, if a path is considered a function of time $\tau : [0,1] \rightarrow q$, so that at each instant t, it generates a configuration that can be applied to the robot $\mathcal{A}$, then the SV is the set of all points $a \in \mathbb{R}^3$ touched by the robot along the entire time interval.

$$SV(\mathcal{A},\tau) = \left\{ a \in \mathcal{A}\big(\tau(t)\big) : t \in [0,1] \right\} \tag{3.1}$$

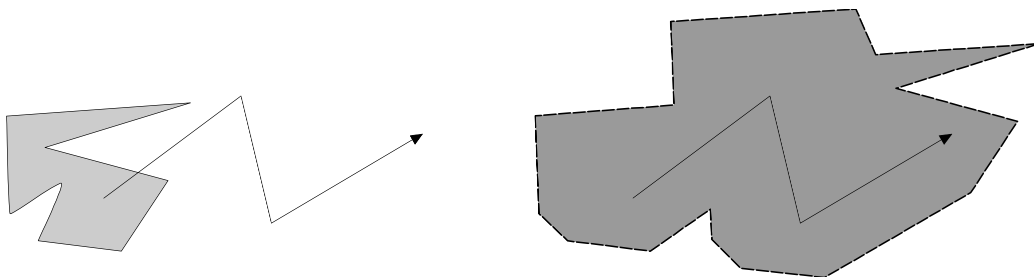

Figure 3.1: Swept Volume (SV). The SV of a path is the shape formed by adding together all the points in space touched by the object as it follows the movement. Here, the object on the left translates along the path shown by the arrow, resulting in the SV on the right. Often it suffices to consider the boundary of the SV, shown as a dashed line on the right.

Since their introduction in the 1960's, SVs have proved useful in many different areas, including numerically controlled machining verification [Abdel-Malek, et al. 2000], robot workspace analysis [Abrams, et al. 1999], geometric modeling [Conkey and Joy 2000], collision detection [Foisy and Hayward 1994], mechanical assembly [Law, et al. 1998], and ergonomic studies [Abdel-Malek, et al. 2004]. Ergonomics is concerned with designing comfortable workspaces that do not cause long-term discomfort for their human users. Through analysis of kinematic modeling of the human limbs (otherwise known as "reach envelopes") engineers can test the feasibility of an operational task and create one or more corresponding reaching paths [Abdel-Malek, et al. 2004].

Not only is the SV an excellent way to visually capture robot motion, but they have practical purposes beyond visualization. Once a path has been generated, engineers often desire to guarantee that it will stay collision-free despite the ongoing design, easing disassembly/maintenance tasks. The SV can represent the volume of one or more paths, and further placement of parts can be efficiently checked for collisions against it.

As its name implies, the SV describes a volume. In many cases, however, it suffices to consider the boundary of this volume. For the purposes of collision detection, for example, any obstacle entering the SV would have to cross its boundary. Most visualization tools would simply draw the boundary as well. If the boundary entraps parts of the object on the inside, then this detail can be discarded. We call this process *wrapping*.

The wrapping process can be thought of as submerging an object in colored paint. When the object is removed from the paint, all the colored surfaces are the boundary of the wrapped region (**Figure 3.2**). To put this idea in formal terms, we first consider that the 3D Euclidian space can be divided into a number of connected spaces by the object geometry. Assuming that the object being wrapped is finite (having a finite diameter), then only one of these spaces has infinite volume. We call this open set $\mathcal{S}_{outside}$, and its complement $\mathcal{S}_{inside}$ is a closed set containing one or more connected spaces. The wrapping operator approximates $\mathcal{S}_{inside}$, or, more precisely, its border.

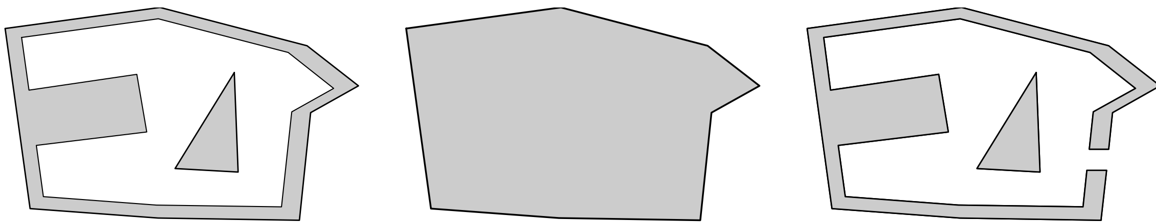

Figure 3.2: Wrapping. The object on the left is closed to the outside, meaning that even if the object was completely submerged in liquid, the inside would stay dry. The wrapping operator effectively removes this detail, rendering the volume in the middle. The object on the right, however, has a hole in it so that the liquid would touch the interior detail. Wrapping it would therefore have no effect.

Wrapping can be performed on a robot undergoing transformation. Instead of considering a continuous path, however, it takes as input a set of configurations $\mathcal{Q}$, and generates directly the boundary of the interior region $\mathcal{S}_{inside}$.

$$Wrap(\mathcal{A},\mathcal{Q}) = \partial\{\, a \in \mathcal{A}(q) : q \in \mathcal{Q}, a \notin \mathcal{S}_{outside} \,\} \quad (3.2)$$

Wrapping has uses independent of the SV. For example, we can consider the reach envelope of a robot as the wrapping of all its obtainable configurations. Or it could be drawn on as a Constructive Solid Geometry operator which finds the union of an object in several configurations. If, as in most cases, we are content with ignoring detail that is closed off within an SV, then we can reformulate the SV in terms of the wrapping operator.

$$SV(\mathcal{A},\tau) = Wrap(\mathcal{A},\{\ \tau(t) : t \in [0,1]\ \}) \tag{3.3}$$

In this chapter, we will deal with SV calculation through this formulation. Though we focus on SV calculation, most of the development applies to the general wrapping operator as well.

PLM designs are based around CAD data, and these models are infamously malformed. By malformed, we mean that they contain degeneracies such as cracks, intersections, or wrongly oriented polygons [Murali and Funkhouser 1997]. Many geometric algorithms require closed 2-manifold volumes (also called "watertight" models) to give meaningful results. Although malformed models can theoretically be transformed into proper volumes, in practice this is both a difficult and time-consuming pre-processing step. In addition, certain models contain flat surfaces surrounding no volume whatsoever. Such models may be dealt with more straightforwardly as polygon soups– unordered sets of triangles with no enforced connectivity constraints.

For this reason, we chose to adapt a state-of-the-art SV approximation algorithm to handle pure polygon soups. Our algorithm stays true to the original volume, and is even capable of generating both volumes and flat surfaces where appropriate.

3.2 Related Work

SV calculation dates back to the 1960s, originally in a 2D context. The problem of calculating the volume is often simplified to finding the boundary of the volume. Even so, the mathematics can be very complex, including self-intersections of the SV. Due to the sheer volume of the work on the subject, we refer the interested reader to the survey by Abdel-Malek *et al*. [Abdel-Malek, et al. 2002].

Modern analytical approaches include envelope theory [Martin and Stephenson 1990], singularity theory (a.k.a. Manifold Stratification or Jacobian rank deficiency method) [Abdel-Malek and Othman 1999, Abdel-Malek, et al. 2000, Abdel-Malek and Yeh 1997], and Sweep Differential Equations [Blackmore and Leu 1990, Martin and Stephenson 1990]. However, the type of data that we are treating does not lend itself easily to mathematical analysis.

If approximations can suffice, then voxels provide a simpler and more practical approach. In the voxelization process, the workspace is divided up into small cubes of a fixed size, called *voxels* [Kaufman, et al. 1993]. A voxel is *active* if the space within it contains object geometry, and *inactive* if it does not. Although we have not found a method for generating SVs from voxels described in the literature, it appears straightforward enough to sample a path, and activate all voxels touched by a robot in those sampled configurations. The final SV is simply composed of the set of all active voxels. An advantage of this approach lies in that voxels can be easily activated by polygon soup models. The disadvantage is that the accuracy is limited by the voxel size. The resulting wrapping will definitely contain the robot, but will generally overestimate the wrapped size (**Figure 2.1**).

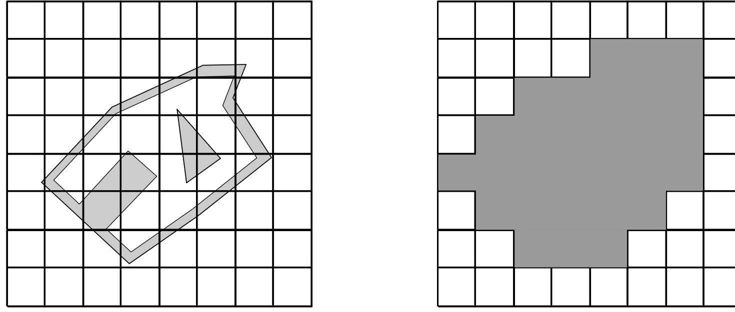

Figure 3.3: Voxels used for wrapping. A voxel is a cube in the workspace that is either active or inactive. By marking all voxels that intersect the object as active, it is easy to calculate the wrapping, or SV. The precision is limited to the size of the voxel.

3.2.1 Implicit Surfaces and Distance Fields

Schroeder *et al* [Schroeder, et al. 1994] introduced another type of method- manipulating numerical approximations of implicit surfaces. They begin with a model from which the SV will be generated. They impose a grid on the model space and assign each grid point $\boldsymbol{p}_m$ a value $f(\boldsymbol{p}_m)$ equal to its distance from the model surface. The workspace (a volume bounding the entire sweep) is imposed a grid as well, with initial distance values $f(\boldsymbol{p}_w)$ of infinity. As the object is swept along its trajectory, the inverse transform of each workspace point is calculated, to find the nearest neighbor points in model space. A new distance value $f'(\boldsymbol{p}_w)$ is evaluated as the trilinear interpolation of the model space distance values. The workspace distance value is then taken as the minimum of the old and new values (**Figure 3.4**). Such grids of distance values are called *distance fields*.

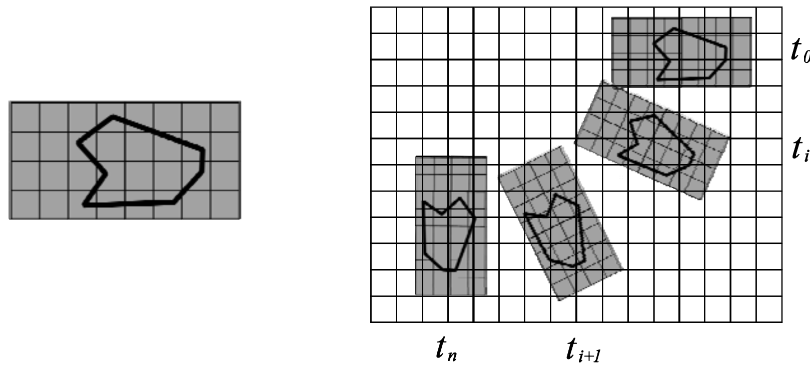

Figure 3.4: Implicit distance calculation using Schroeder's inverse. The geometry of the robot on the left is specified in its own reference frame called model space. A grid is imposed upon it, and a value is assigned to each grid point, equal to the distance from the object surface. On the right, the workspace is created so that it is large enough to contain the final SV. It is assigned a grid as well, with all values equal to infinity. As the object is swept through the workspace, a transformation is applied at each step in order to transform from model space coordinates to workspace coordinates. Once the nearest model space points have been found, trilinear interpolation is used to derive a new distance value. If this distance value is less than the current one for the workspace point, it is substituted for the old one.

It is important to note that the distance values for grid points lying inside the volume are given negative values. Thus, the boundary of the SV can be approximated as an isosurface where the distance value equals zero. To this end, the Marching Cubes algorithm interpolates $f(\boldsymbol{p})$ between two grid points [Lorensen and Cline 1987]. Where $f(\boldsymbol{p})$ equals zero, the algorithm will output a vertex, and connect it to neighboring vertices to create a watertight manifold.

A consequence of this approach is that a volume must cover at least one grid point in order for it to be detected. This is due to the fact that interpolating $f(\boldsymbol{p})$ between two grid points that are considered “outside” the swept volume (i.e. $f(\boldsymbol{p}) > 0$) will never produce a value equal to zero. Therefore, this algorithm can neither detect nor generate flat surfaces.

Nevertheless, it appears that the work of Schroeder *et al* [Schroeder, et al. 1994] has been used in commercial products to generate swept volumes of polygon soup models [Law, et al. 1998]. Although not explicitly discussed in the literature, we can hypothesize that this feat is accomplished by tessellating at a positive distance value (i.e. $f(\boldsymbol{p}) = d,\ d > 0$). This is equivalent to adding an offset d around all surfaces, essentially “growing” the volumes. In this case, surfaces become volumes. If d is high enough, any point on the polygon soup will grow into a volume sufficiently large to cover at least one grid point, making these points detectable, at the cost of deforming the resulting volume. For our work, we insisted on the capability of generating surfaces without offsetting.

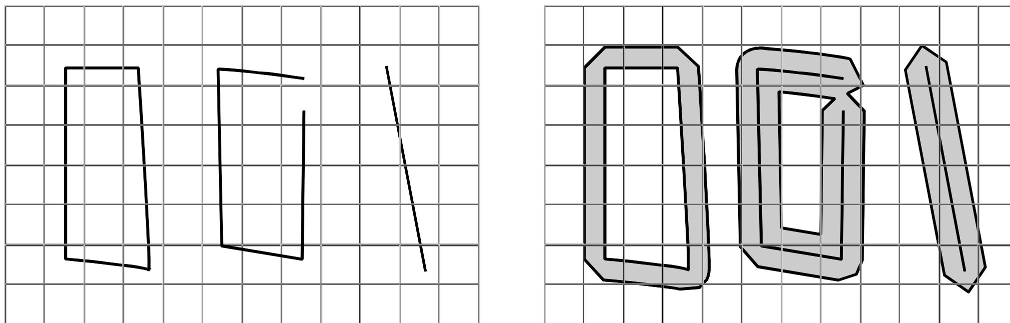

Figure 3.5: Offsets for implicit surfaces. On the left, we show 3 different types of geometry: a closed volume, a volume with a hole, and a flat surface. Without an offset, only the closed geometry would be correctly wrapped. On the right, the effect of adding a sufficient offset is shown. The line segment would be dilated into an object large enough to be wrapped by the algorithm, and the hole would be closed. However, all objects would be affected by the offset, including what normally could be wrapped without it.

The distance fields used in this algorithm can be generated through regular sampling of a bounding volume [Bloomenthal 1988]. Such sampling lends itself naturally to the powerful parallel processing capabilities of a graphics card, now commonly referred to as a Graphics Processing Unit (GPU) [Kenneth E. Hoff, et al. 1999].

3.2.2 GPU-Based Directed Distances

Kim *et al.* [Kim, et al. 2003] combine and extend the implicit surface approach to quickly find the SV boundary using the GPU. The essential concept behind their approach is to transform the implicit function evaluation into a ray-casting problem. Large number of parallel rays can be evaluated by the depth buffer of the GPU, intended to ensure that objects closer to the viewer obscure those behind them.

The algorithm takes a triangulated mesh and a trajectory composed of rigid motions as input. The edges and faces of the mesh are treated as ruled and developable surfaces, and triangulated along the trajectory within a certain error threshold. The new object includes the SV boundary, but contains surfaces on the interior of SV as well.

To remove the interior surfaces, the object is split into slices, and a 2D grid is imposed onto each slice (**Figure 3.6**). Using the GPU, distance fields are found along the edges between neighboring grid points. The distance fields are directed (along the 3 major axes), rather than the scalar values as in Schroeder *et al*[2]*.* They are also unsigned, as there is not yet a notion of interior and exterior.

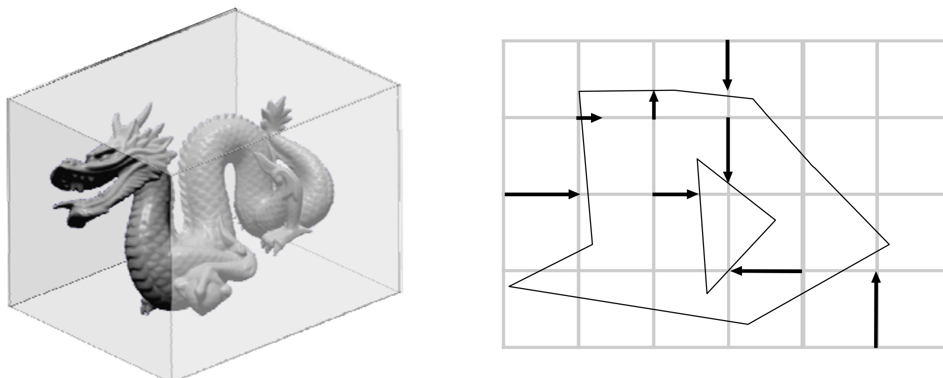

Figure 3.6: Directed distance fields. In order to efficiently calculate distance fields using the GPU, directed distance fields replace omni-directional ones. The workspace is represented by a bounding box (left) surrounding the robot within it (here, the dragon model). A regular grid then is imposed upon the workspace. This grid divides the workspace in slices, and depth measurements are performed for each slice. The depth measurements are attached to each edge of each grid cell, in all 6 directions. A portion of edges are illustrated as arrows (right), stopping where they hit the surface or when they reach the next grid cell.

The grid points are then classified as outside or inside the SV using a propagating front level set method. To produce the volume, the surface of the SV is extracted using the Extended Marching Cubes (EMC) algorithm [Kobbelt, et al. 2001], which exploits the directed distance fields and triangle normals to provide a more faithful triangulation than traditional Marching Cubes. The final step is a topological check. If the surface is not evaluated as closed and watertight, the spatial grid is refined and the algorithm executed again.

Their algorithm represents an advance from that of Schroeder *et al.* both in performance (thanks to the GPU distance-fields) and quality (through EMC and the absence of interpolation). However, the range of acceptable input is limited; only a single watertight 2-manifold is allowed. This restriction is imposed by the tessellation method and the final topological check. To handle difficult cases in PLM, critical modifications need to be made. Such modifications constitute the main contributions of our work on wrapped volumes.

3.3 Wrapping Polygon Soups

We have devised a fast SV approximation algorithm to accept arbitrary polygon soups as input, and generate watertight volumes or flat surfaces as output, depending on the result. Given a robot $\mathcal{A}$, and a trajectory τ, our algorithm generates $\mathcal{W}$, a triangulation of the boundary of $Wrap(\mathcal{A}, \tau)$. To do so, it follows these steps:

1) *Trajectory sampling* – Convert τ to a set of configurations Q.
2) *Workspace creation* – Create an bounding box that contains all Q.
3) *Distance-field gathering* – From each side of the box, slice the box along the grid. For each slice, obtain the distance field values from the GPU depth-buffer.
4) *Surface extraction* – Using an advancing front method, determine which grid edges cross the surface of the wrapped volume. Discard the others.
5) *Triangulation* – Triangulate the surface points associated with the remaining grid edges to produce $\mathcal{W}$.

The remainder of this section is devoted to explaining these steps in more detail. Only Step 2 is omitted since it is fairly straightforward.

3.3.1 Trajectory Sampling

Rather than creating a swept mesh through ruled and developable surfaces as with Kim *et al* [Kim, et al. 2003], we decided to sample the surface at a certain number of intervals. A motivating factor behind this choice is our desire to focus on the more generalized wrapping in addition to SV calculation. Decomposing the trajectory into a set of configurations allows for a convenient way to pose the SV problem in terms of wrapping.

A *sampling function* γ converts from a trajectory τ to a set of configurations reached by that trajectory, $\gamma : \tau \rightarrow \{ \tau(t) : t \in [0,1] \}$. The simplest such function naïvely samples along along regular intervals of t, producing $n + 1$ samples including the endpoints.

$$\gamma_{regular}(\tau, n) = \left\{ \tau\left(\frac{i}{n}\right) \right\}_{i=0}^{n} \tag{3.4}$$

However, arbitrary motions can lead to situations where certain points sweep large paths while others barely move. To limit error in this case, it would be necessary to impose a large number of intervals, many of which would be wasted on more constant movements. Additionally, determining the error bound implied by a given number of samples is not trivial (**Figure 3.7**). Another approach, used by continuous collision detection methods, would be to consider the movements *between* each pair of samples rather than the samples themselves [Redon, et al. 2004]. However, this approach boils down to calculating a kind of swept volume for each interval. Given the malformed geometry that we expect as input, calculating each of these shorter swept volumes would be just as demanding, if not more, as calculating one volume for the whole.

A response to this problem lies in [Ferré and Laumond 2004], where the authors define a *bounded distance* operator on robot paths. Given the robot $\mathcal{A}$, we can examine the distance of the longest path swept out by a point between two instants along the trajectory.

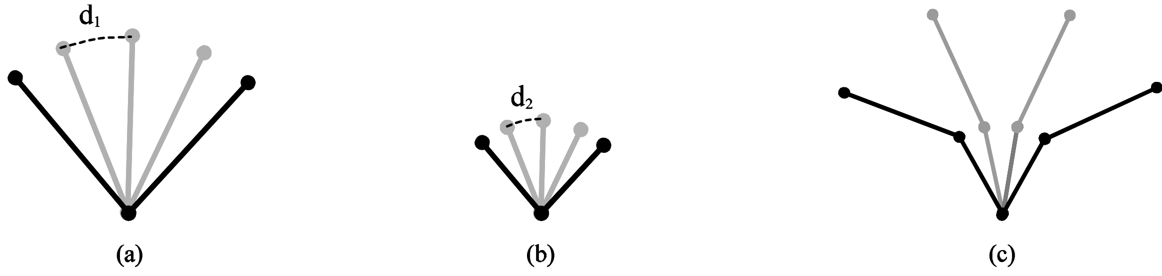

Figure 3.7: Problems with regular path sampling. In (a) and (b), a simple straight arm is rotated about one end. Sampling this trajectory uniformly would result in the very different error bounds d_1 in (a) and d_2 in (b). This problem is only further aggravated when kinematic chains, such as (c), are introduced. The bounded distance operator aims at controlling the maximum error.

$$maxSweep(\tau, t_0, t_1) = \max_{a \in \mathcal{A}} \left(\int_{t_0}^{t_1} \left\| \frac{d\big(\tau(t)\big)}{dt} \right\| dt \right) \tag{3.5}$$

We note that two configurations $\tau(t_0)$ and $\tau(t_1)$ are bounded by distance Δ if no point on the robot sweeps out a path longer than Δ between them.

$$bounded(\tau, t_0, t_1, \Delta) \Rightarrow maxSweep(\tau, t_0, t_1) \leq \Delta \tag{3.6}$$

In practice, it suffices to examine the trajectories swept by vertices lying on a bounding volume (such as a convex hull) of each robot body [Schwarzer, et al. 2004].

To generate a sequence of configurations along a trajectory using this notion, we assume a function $\delta(\tau, t_0, \Delta)$ such that $maxSweep\big(\tau, t_0, \delta(\tau, t_0, \Delta)\big) = \Delta$. Then we can create our trajectory sampling function on δ.

$$\gamma_{bounded}(\tau, \Delta) = \left\{ \tau\big(\delta(\tau, 0, i\Delta)\big) \right\}, \quad i \geq 0, \delta(\tau, 0, i\Delta) \leq 1 \tag{3.7}$$

3.3.2 Distance Field Creation

By using directed distance fields instead of omni-directional distance fields, Kim *et al.* are able to use basic GPU functionality to calculate them [Kim, et al. 2003]. In a nutshell, a transformation matrix is put into place so that the "camera" points down an axis of the workspace. In addition, orthogonal projection is used so that perspective does not come into play. The front plane of the view frustrum is set right before the camera position, and the back of plane is set to the slice depth. The remaining planes are similarly aligned to fit the workspace box nicely (**Figure 3.8**).

One minor innovation that we bring to this step is the way in which we slice up the workspace box. Instead of using a fixed number of grid cells in each direction, as in [Kim, et al. 2003], we let the number of grid cells fluctuate so that the size of each grid cell is identical along each axe. In other words, our grid cells are cubes instead of cuboids. By doing so, we drastically reduce the number of total grid cells that need to be taken into account (**Figure 3.9**). This impacts the performance of the rendering as well as the surface extraction and triangulation steps.

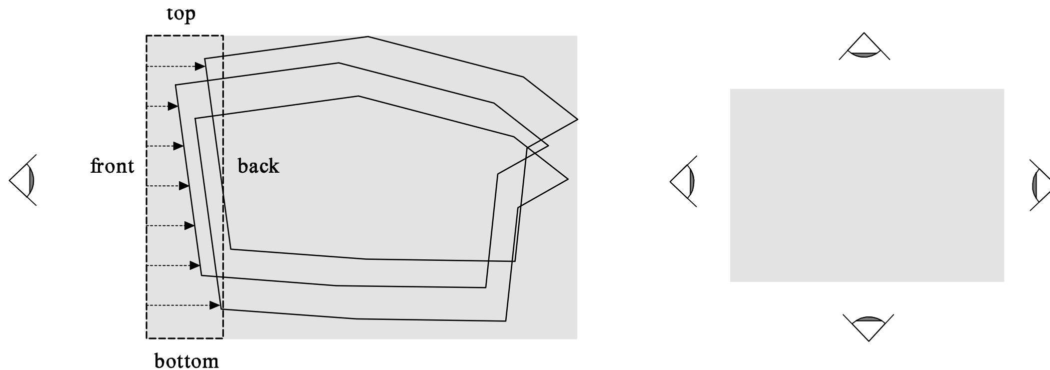

Figure 3.8: Directed distance field gathering. To obtain distance fields, the virtual "camera" is positioned so that it sees only one slice at a time, in orthogonal projection (one such view frustum is outlined in dashed lines). Then the robot geometry is rendered on the GPU. The depth buffer stores the distance to the geometry that was closest to the camera, and so by reading back the buffer we obtain the entire distance field. This process is repeated for each slice of the view, and then for each of the 6 sides of the workspace box.

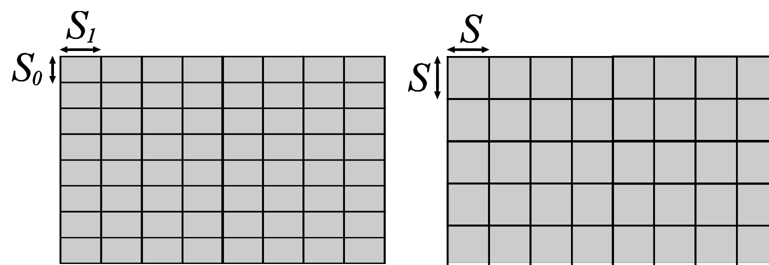

Figure 3.9: Grid division. In the original algorithm, the workspace was divided into an equal number of grid cells for each axis. Since the size of the grid cell plays a role in the final error bound, the size must be reduced along the longest axis. For environments that are not cubic, this logic leads to distortion effects, in which the shorter axes are "squished" and contain grid cells that do not contribute to lowering the error bound. On the left, S_1 is the desired grid cell length, and $S_0 < S_1$ is a side-effect. By fixing the grid cell length in all directions, and letting the number of grid cells vary per axis, we can drastically reduce the number of total grid cells, while keeping the same error bound (right). Here the grid on the left has $8 \cdot 8 = 64$ cells, whereas the one on the right only has $5 \cdot 8 = 40$.

3.3.3 Surface Extraction

In implicit modeling, a field function $f(p)$ defines a value for each point p in space. Surfaces are therefore equivalent to contours sharing a field value. In our case, $f(p)$ returns the distance to one of the polygons drawn by the GPU. However, $f(p)$ will be zero for surfaces lying within the SV as well as along the boundary.

When dealing with volumes, it suffices to negate $f(p)$ for all p within the SV. As long as at least one grid point falls within the volume, it creates a difference that can be detected by the discrete surface extraction procedure. However, when dealing with surfaces that do not contain any volume this method fails to find the contour.

To properly detect these surfaces, we have modified the fast marching level-set method presented by [Kim, et al. 2003], which itself was based upon [Sethian 1996]. Whereas they use it to simply classify grid *points* as inside or outside, we employ it to tag grid *edges* that cross the surface. These edges can later be used by our specialized triangulation algorithm (**Figure 3.10**).

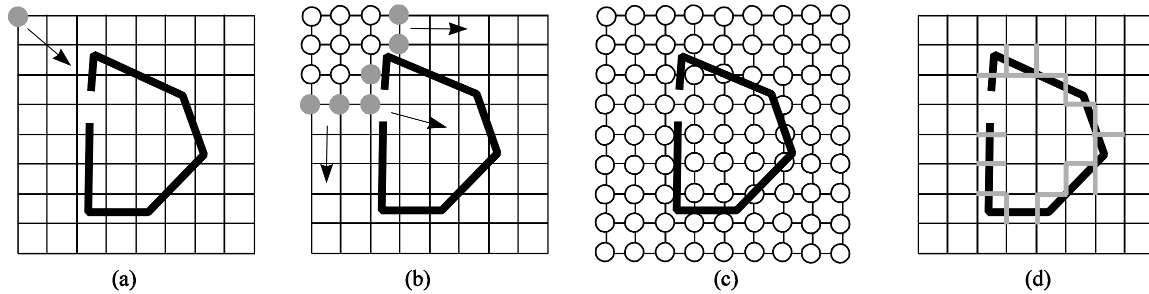

Figure 3.10: Surface extraction. The advancing front can enters holes in non-watertight objects, and thus fail to produce a usable iso-surface. The front, represented by grey circles, starts at the upper left corner in (a), moving down and to the right. In (b), the front has advanced up to the hole in the volume. By the time the front is exhausted, in (c), it has completely filled the space. Since no grid points lie inside the object, it is impossible to recover the correct surface. Although the hole is exaggerated here for the purposes of example, the same phenomenon is manifested by the smallest of imperfections in common PLM models. Our modified surface extractor gets around this problem by marking grid edges that cross the surface, such as the highlighted edges in (d). This modification requires a custom triangulation method in the place of the Marching Cubes algorithm.

The method is based on an advancing front that starts from the extremities of the grid and slowly moves inwards until it hits the wrapping, at which point it stops. In order to calculate the front advancement in a sequential manner, a queue is used that contains all grid points currently participating in the front. So as to avoid adding the same grid point to the queue multiple times, all grid points are tagged with a *state* of 3 possible values. The state is initially *Far*, meaning that the grid point has not been reached by the advancing front. A point in the front is in the *Trial* state, and once a point has been analyzed it is *Known*.

In addition, grid edges are associated with a color, initially white. Once a grid edge is reached by the front, it is colored black if it crosses the wrapping surface (**Algorithm 3.1**). Once the queue is exhausted, these black edges are precisely those that cross the wrapping boundary.

Note that this surface detection algorithm refuses to enter closed volumes, but also recognizes non-volumetric surfaces, and is therefore appropriate for any kind of geometrical model, polygon soup or otherwise.

```
1.   while front ≠ ∅:
2.     p ← pop(front)
3.     state(p) ← Known
4.     for each neighbor q of p:
5.       if crosses_surface(pq→) then
6.         color(pq→) ← Black
7.       else if state(q) = Unknown then
8.         state(q) ← Trial
9.         push(front, q)
10.      end if
11.    end for each
12.  end while
```

Algorithm 3.1: Fast Marching Method adapted to recognize surface points. The algorithm works upon a queue of grid points called `front`. At each iteration, it takes a new grid point from the queue, and examines the edges with its neighbors. If the wrapped surface crosses an edge then this edge is colored black (all edges start in white). If the edge does not cross the surface, then the neighbor grid point is placed in the queue. The algorithm terminates when the queue is empty, by which time all edges on the wrapping surface have been colored black. Only these edges need to be taken into account for the following triangulation step.

3.3.4 Triangulation

From the distance fields gathered earlier, we know where each grid edge crosses the wrapping surface. We are interested in the 3D coordinates of these crossings, which we will call *detected surface points*, since they form the wrapping surface. Once the detected surface points are identified (i.e. those edges colored black by **Algorithm 3.1**), the next step is to triangulate them.

A large number of published tessellation algorithms could be used for this task. In particular, algorithms that tessellate point clouds would be appropriate [Boissonnat and Yvinec 1998, Sack and Urrutia 2000]. However, rather than dealing with the detected surface points as an unorganized set of points, we can exploit the known structure of the data returned by the level-set method to achieve linear-time triangulation.

Intuitively, we link each detected surface point with its neighbors, and then triangulate the result. In order to find a neighbor for a detected surface point, we start at the grid edge in question and proceed to walk along the grid in a closed curve until we cross the surface again. A small number of these walks will yield enough neighbors to triangulate with.

In order to keep track of this information, we transpose the detected surface points onto an undirected graph. Each node of the graph represents a detected surface point, and the edges between nodes provide adjacency information. Intuitively, each point should be linked with its neighbors in the final triangulation. In terms of the graph, this means that each point participates in an elementary cycle with its neighbors.

In order to link a node of the graph with its neighbors in the point cloud, we follow the underlying grid structure. Starting from the detected surface point of the node, we engage on four pre-determined walks along the edges of the grid, identical except for their starting directions (**Figure 3.11**). We continue each walk until we cross the surface again. The node corresponding to this detected surface point is our neighbor, and an edge is added to the graph between these two nodes. Since the final edge brings us back to the grid point from which the surface was initially detected (i.e. the walk traces a closed curve), we are guaranteed to hit the surface at some point during the walk.

From here it is fairly easy to tessellate the graph. By following those cycles that do not contain any others, we construct a simple loop that can be easily triangulated. For example, in **Figure 3.12**, nodes $\{\overrightarrow{ef}, \overrightarrow{ab}, \overrightarrow{cd}, \overrightarrow{jd}, \overrightarrow{ib}, \overrightarrow{hf}\}$ form such a cycle. The algorithm terminates when there are no cycles left. By choosing a circular direction (clockwise or counter-clockwise) when enumerating the cycles, all triangles can be oriented to face out of the volume.

The graph representation defines a valid topological relationship between points in the cloud. It is important to note that the graph always represents a closed 2-manifold volume, rather than a polygon soup. In other words, this algorithm tessellates all objects (including those detected to have strictly planar sections) as closed polyhedra (although duplicate polygons could be filtered to preserve single-sided planar geometry). Certain degeneracies may still arise, such as non-manifold edges or isolated line segments. These degeneracies can be detected and removed before the triangulation process is launched.

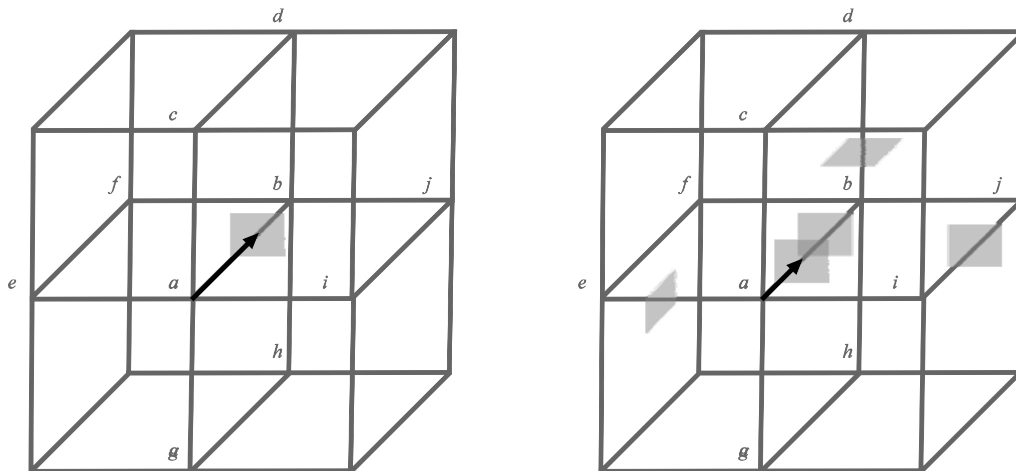

Figure 3.11: Walking along the grid. In order to triangulate the detected surface points, we first need to determine which pairs of surface points are "neighbors". To do so, we start at a single detected surface point and take 4 pre-determined walks along the grid. Along each walk, we check if certain grid edges cross the surface, and stop if that is the case. Each walk touches 4 edges maximum and is guaranteed to reach a neighboring surface point. In the example on the left, the surface was detected along the edge $\overrightarrow{ab}$. The first walk leaves upwards from a along edge $\overrightarrow{ac}$, then forwards along $\overrightarrow{cd}$, downwards along $\overrightarrow{db}$, and finally back again along $\overrightarrow{ba}$. Similarly the walk moving left from a involves the sequence of edges $\{\overrightarrow{ae}, \overrightarrow{ef}, \overrightarrow{fb}, \overrightarrow{ba}\}$, the one moving down $\{\overrightarrow{ag}, \overrightarrow{gh}, \overrightarrow{hb}, \overrightarrow{ba}\}$, and the one moving right $\{\overrightarrow{ai}, \overrightarrow{ij}, \overrightarrow{jb}, \overrightarrow{ba}\}$. On the right, we include neighboring detected surface points. The upwards walk would therefore touch and stop on $\overrightarrow{db}$, the leftwards walk would touch $\overrightarrow{ae}$, the rightwards one $\overrightarrow{ij}$, and finally the downwards walk would loop around to touch the opposing surface point on $\overrightarrow{ba}$.

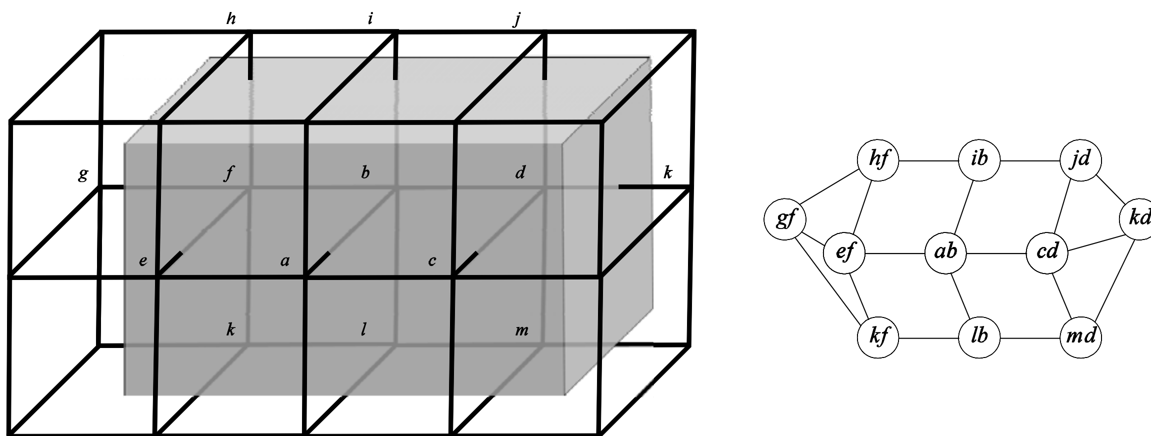

Figure 3.12: Graph building. When the box on the left is placed in the grid, 11 surface points are detected at the intersections. They are transposed onto the graph on the right, by connecting each detected surface point (described by the corresponding grid edge) to its neighbors. Note that the letters on the left designate the grid points, whereas the nodes on the right reference grid edges by a pair of grid points. For simplicity, the grid in this example is smaller than the box. In practice, however, the grid always contains the object to wrap, and thus its graph would completely surround the volume and not betray boundary edges as in this example.

3.3.5 Error Bounds

Both the distance field generation and trajectory sampling steps are potential sources of sampling error in our approximation algorithm. Each one is controlled by a single parameter, and their combination defines the error bound as well as the time and memory complexity of the

algorithm (there is an additional error related to hardware, since the GPU depth buffer is used to calculate the distance fields, and so precision is limited by the width of the buffer).

Instead of dividing the workspace into an equal number of grid points on each axis, which leads to cuboid workspace cells, our algorithm takes the slice depth S as an input parameter, and divides the workspace so that the cells are cubes of this size. This has the advantage of reducing the number of grid points under consideration, without reducing the error bound.

Consider sampling an object that does not move. Between the grid lines, no measurements are taken. Therefore the error is limited to the possible distance the surface could move within the grid cell, which is equal to its diagonal length $\sqrt{3}S$ (**Figure 3.13**).

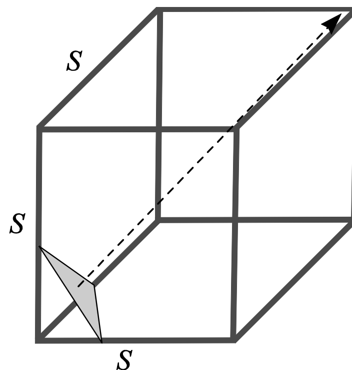

Figure 3.13: Distance field sampling error. Since the SV boundary is detected where it crosses the edges of a grid cell, fluctuations of the boundary within a cell go unnoticed. The inaccuracy of the resulting approximation (the grey triangle) is therefore limited by the size of the grid cell, S. The worst case error is the diagonal length of the grid cell, $\sqrt{3}S$.

Now consider the movement of the object along its path. As described in Section 3.3.1, we can guarantee that no point on the object traces a path longer than Δ between two sampled configurations. Assuming that the point is detected at both configurations, the farthest it could move from the detected surface is $\Delta/2$.

It is now possible to take both sources of error into account. In the worst case, the surface will be sampled at a distance S from its real location. In addition, between the two sampled configurations, the surface will have moved $\Delta/2$. The resulting effect is the sum of the two: $\varepsilon \leq \sqrt{3}S + \Delta/2$.

The error bound ε is a useful measurement, for it is independent of the size of the object or the length of its path. With this simple relation, the user has the possibility to adjust the speed and memory use of the algorithm while staying within a global error bound.

3.4 Wrapping with Offsets

In previous sections, we have described how a polygon soup can be turned into a wrapped volume. The key steps are sampling the geometry along a regular lattice, removing those points "inaccessible" from the exterior, and then connect the remaining sampled points in order to reconstruct the surface of the wrapped volume. The lattice serves as the central data structure underlying the operation, providing a stable scaffolding to which both the input and output

geometry is linked. It is important to note that once the interior points are removed, the lattice is guaranteed to describe a closed volume, regardless of whether the input was one itself. The lattice offers a chance to operate on a well-behaved volume before it is once again reduced into a collection of triangles.

In this section, we propose to manipulate the volume present on the lattice during a wrapping operation in order to modify the size of the resulting object. Since the final object surface is displaced or offset from the original, we call this operation *wrapping with an offset*. A positive offset corresponds to an inflation (or growing) of the volume, whereas a negative offset leads to deflation (or shrinking).

3.4.1 Why Offset?

In Section 3.2.1, we discussed how our approach to generating wrapped volumes is distinguished by its ability to handle polygon soup input without offsets. The algorithm wraps as closely as possible to the input in order to produce the highest fidelity approximation that it is capable of. Nevertheless, there are times where an offset is desirable.

Geometric tolerances are used in manufacturing processes to designate bounds of acceptable production error [Jayaraman and Srinivasan 1989]. Given that all final products differ slightly from their design, tolerancing specifies the allowed variation in forms and dimensions of geometric features, as well as the distances between them. Offsetting is one way to visualize an object that includes tolerances by inflating the object to include all possible variations.

In robotics, users of motion planning techniques often wish for their planned path to avoid obstacles as much as possible in order to minimize the risk of collision. Certain algorithms expressly seek the path that stays in the middle of open spaces, but sampling-based planners mostly content themselves with finding collision-free paths. In order to force such planners to avoid close brushes, tolerances can be added to the robot. For each configuration, the collision detector measures the distance to the nearest obstacle along with the tolerance, and declares a collision if the former is less than the latter. If swept volumes are used to reserve space for future path planning, then a swept volume of a path can be inflated by the tolerance so that it will be implicitly included in the volume.

There are reasons to deflate volumes as well. Where a path planner is unable to find a solution, it can try to reduce the size of the robot or the obstacles. The resulting path could give indications regarding the obstacles that are causing the problem. Such a technique can also be used to solve *forced passage* problems, in which the robot is allowed to collide with obstacles up to a certain point. Another approach for forced passage problems is discussed in Chapter 5.

The simplest method for modifying the size of a geometric object is to apply a uniform scaling. In uniform scaling, relative distances between points are preserved. At first, this would seem to be a useful attribute, but in fact it makes it unfit for path planning applications. For example, one could expect that shrinking obstacles would enlarge a narrow passage between them. And yet reducing obstacle sizes through uniform scaling has the opposite effect.

Instead of scaling, we would like our offset operator to perform “fattening” or “thinning” on objects. Inflating obstacles should not create collisions that did not previously exist, nor should deflating them avoid collisions that did. One mathematical tool that allows this capability

is the Minkowski sum, which provides a way to add one object to another. The Minkowski sum is defined as

$$A \oplus B = \{\, a + b \,:\, a \in A, b \in B \,\} \tag{3.8}$$

and the Minkowski difference $A \ominus B$ is simply equivalent to $A \oplus (-B)$.

Minkowski sums involving spheres are especially useful. Adding an object to a sphere of radius r effectively enlarges the object by r. Similarly, subtracting the sphere shrinks it by the same amount. This technique corresponds nicely to the positive and negative offset operations [Rossignac and Requicha 1986]. Unfortunately, calculating Minkowski sums of arbitrary 3D polyhedrons is a difficult problem in and of itself [Varadhan and Manocha 2006].

Using volumetric structures instead of surfaces opens up other possibilities. For example, researchers have defined a thinning operator for tetrahedral models. Surface vertices are pushed inward, but only to the extent that the result is contained within the original volume. This approach has been used for path planning, but requires well-formed models that can be tetrahedralized [Hsu, et al. 2006].

Rather than trying to deform a triangle-soup model, a voxelization, or a volumetric mesh, we would like to work directly on the intermediate grid structure created by our wrapping operator. We have devised algorithms that both inflate and deflate the wrapped structure.

3.4.2 Inflation

In their original work on SV approximation, Schroeder *et al.* treat the distance to the SV surface as an implicit function, evaluated at each grid point. They then extract the implicit surface formed where the distance is zero [Schroeder, et al. 1994]. This formulation allows for elegant thinning and fattening of the volume, simply by adding a scalar offset value to the distance. That is, if the distance from a grid point p to the object surface is given by $d(p)$, then accounting for an offset δ is as simple as extracting the surface where $d(p) - \delta = 0$. This works equally well with positive and negative values of δ, although no skeleton is preserved for negative offsets.

It is tempting to perform the same trick for the directed distance fields that we gather with the GPU, but a quick check shows that this intuition is misleading. By "pulling" the points towards the six sides of the workspace, the directed distance fields no longer agree on the position of the object to be wrapped. One possible way around this problem is to consider multiple instantiations of the object, each transposed from the original position, and wrapping the entire set, but it is not clear how to control the error that would arise, nor how performance would be impacted (**Figure 3.14**).

To address this problem, we need a method that acts on all the directed distance fields equally and simultaneously. Inspired by the use of Minkowski sums to perform dilation, we propose replacing the object with a union of spheres, each having a radius equal to the desired offset. Given a simple definition of a sphere,

$$sphere(c, r) = \{\, a \in \mathbb{R}^3 \,:\, \|a - c\| \leq r \,\} \tag{3.9}$$

we can define the inflation operator as,

$$inflation(\mathcal{A}_W, \delta) = \bigcup_{a \,\in\, \mathcal{A}_W} sphere(a, \delta) \tag{3.10}$$

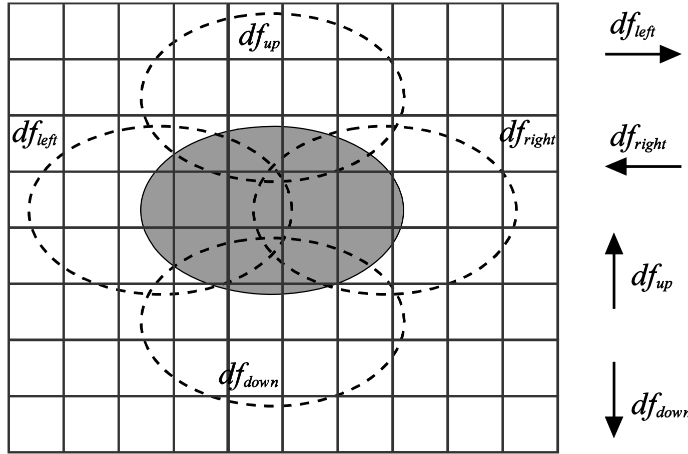

Figure 3.14: Naive manipulation of directed distance fields. If each directed distance field were to be simply offset by a scalar value as is done for undirected distance fields, they would no longer agree on the position of the object to be wrapped. Here, the ellipse in grey has been offset twice along each axis, corresponding to the 4 directed distance fields df_{left}, df_{right}, df_{up}, and df_{down}. The positive offset leads df_{left} to consider the object as to the left of its original one, but the other fields each have a different conception, condemning the surface extraction and tessellation methods to failure.

Since we use the lattice of directed distance fields to store our points, only the intersections between the spheres and the lattice are of importance. As with the original distance fields, some of these points will lie on the surface and others will not. We can use the advancing front method to separate the two. For this reason, there is no need to consider spheres based around points on the interior of the wrapped object, since they will not contribute to the offset surface (**Figure 3.15**).

The volume produced by our inflation operator is guaranteed to contain the original volume. Since each point in the wrapping is surrounded by a sphere of radius δ, each vertex p_w on the original detected surface will be encircled by multiple vertices p_δ where $\|p_\delta - p_w\| = \delta$. Therefore, p_w will never be reached by the surface extraction method. In general, if $\zeta(\mathcal{A}_W)$ is the subset of $\mathcal{A}_W$ that is extracted as surface points along the grid, then

$$\forall p_w \forall p_\delta \big(p_w \in \zeta(\mathcal{A}) \wedge p_\delta \in \zeta\big(inflation(A, \delta)\big) \longrightarrow \|p_w - p_\delta\| \geq \delta \big) \quad (3.11)$$

Between extracted surface points, the surface is linearly interpolated and then triangulated. Therefore the distance between the inflated surface and the wrapped surface may be less than δ. The worst case occurs if a vertex is found at the exact center of a lattice edge (of length S) and if $\delta < S/_2$, leading to an error of up to δ. An obvious workaround to this problem is to decrease S with small values of δ (**Figure 3.16**).

In practice, it would be inefficient to explicitly mark every intersection between the grid and the offset balls, since many of the newly marked surface points will not lie on the surface and therefore will not contribute to the final volume. Instead, the offset operation can be integrated into a surface extraction procedure. Just as for the normal surface extraction scheme (Section 3.3.3), this is an advancing front method. In place of checking if the depth fields contain the surface along the edge between two grid points, we check for intersections with balls from neighboring edges, and mark the minimum.

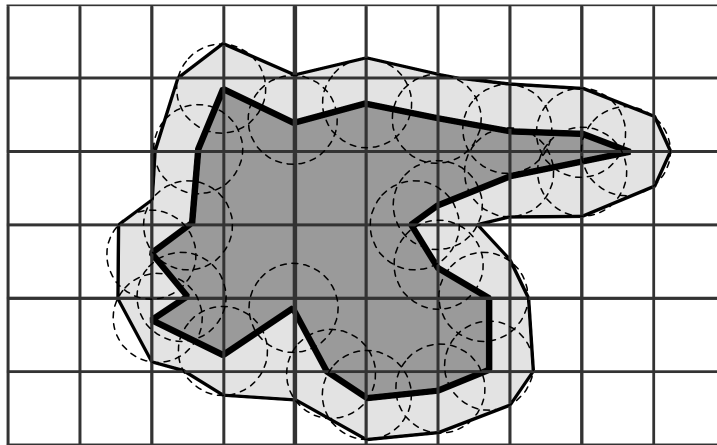

Figure 3.15: Positive offset using spheres. The original wrapped object (filled in dark grey and bordered with a thick line), is offset by considering a sphere around each detected surface point on the lattice (shown as a dotted circle). Each intersection between the spheres and the lattice is marked in the lattice as if geometry had been detected there. By running the surface extraction scheme a second time, the offset surface will be obtained (filled in light grey and bordered with a thinner line). The offset volume is guaranteed to contain the original. Certain detail may be masked by the rounding out operation, as can be seen in the lower-left part of the object, but reducing the grid cell size can refine the shape.

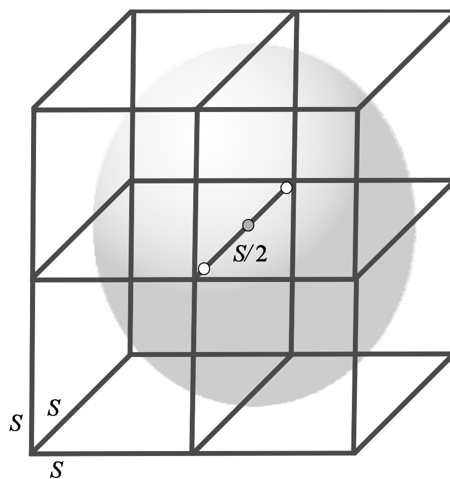

Figure 3.16: Positive offset error. In the worst case, a vertex will be located at the center of an edge (the grey circle). If the offset δ is less than half the length S of the edge, then only two vertices will be created on that same edge (the white circles), since the sphere surrounding the original vertex does not touch other edges. The resulting triangulation could carry an error of up to $S/_2$ in this case.

Given a function describing a line segment

$$segment(p_0, p_1) = \{\, p_0 + u(p_1 - p_0) : \; 0 \leq u \leq 1 \,\} \tag{3.12}$$

and the minimum distance between two subsets

$$distance(P, Q) = \min_{p \,\in\, P, q \,\in\, Q} \|p - q\| \tag{3.13}$$

the distance to the surface is simply the distance to the intersection of the current edge with a sphere.

$$distToSurface(p_0, p_1) = \min_{a \,\in\, \mathcal{A}} \|p_0 - distance(segment(p_0, p_1), sphere(a, \delta))\| \tag{3.14}$$

Line-sphere intersection tests may be trivial to perform, but they remain many times slower than the single depth field lookup used by the classic surface extraction procedure. For an extreme case, imagine testing each edge against all the spheres associated with any surface point on the grid. This would obviously be unnecessary, since some spheres would be too far, and others too close, to possibly intersect with the given edge.

Since the edge length and sphere radius (i.e. offset) remains constant for all line-sphere intersection tests, we pre-calculate the set of neighboring edges that run the chance of intersecting with a given edge. This set describes the relative positions of neighboring edges that can be applied to any edge under consideration. Additional filtering can be performed for each edge, based on the direction of the edge and the positions of the neighboring spheres. Finally, only the remaining spheres are given the complete intersection test (**Figure 3.17**).

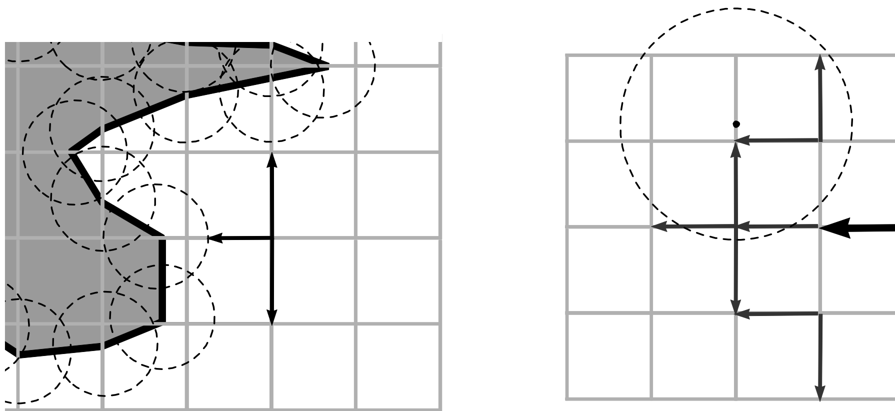

Figure 3.17: Positive offset integrated into surface extraction. Instead of checking for intersections with the object surface, the offset surface extraction method checks for intersection with spheres placed around previously detected surface points (left). As before, some edges contain no such intersections (the up and down arrows here) whereas some others do (the left arrow here). Since the sphere radii and the grid cell size are known *a priori*, a subset of edges can be calculated before the offset process begins (right). Here, the thick left arrow indicates the edge under consideration. Regardless of its location on the grid, only certain edges (thinner arrows) need be checked for sphere intersections. The sphere shown here can be culled, as it lies on an edge that is too far from the one under consideration.

3.4.3 Deflation

The principle behind the thinning operator follows the same lines of the inflation method. The idea is to place spheres around the wrapped surface points and then *remove* those spheres from the wrapping (this is similar to the Minkowski subtraction of a sphere from the wrapped object). For the time being, we will not concern ourselves with saving the skeleton of the object, meaning that the surface can disappear if the offset is large enough. For certain applications this is unacceptable, and so in the next section we address topology preservation (**Figure 3.18**).

For this thinning method to work, only spheres around the surface points should be taken into account. Their union is removed from the original volume. Formally, if $\partial\mathcal{A}_W$ indicates the boundary of $\mathcal{A}_W$, then

$$deflation(\mathcal{A}_W, \delta) = \mathcal{A}_W \setminus \bigcup_{a \in \partial\mathcal{A}_W} sphere(a, \delta) \tag{3.15}$$

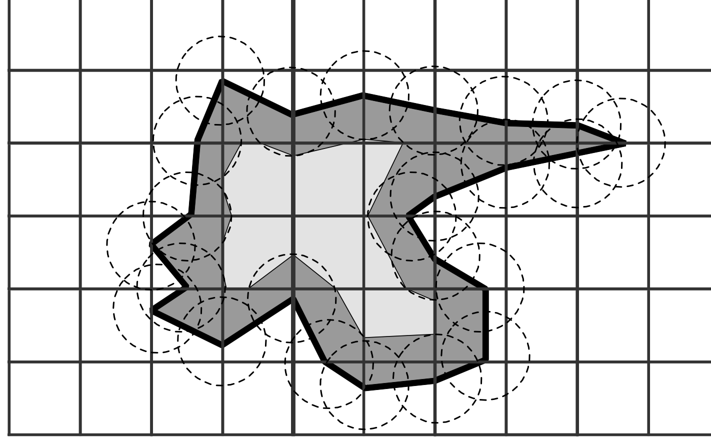

Figure 3.18: Negative offset using spheres. The technique is the contrary of the positive offset method, in that the spheres around detected surface points are removed from instead of added to the original wrapping. Notice that if the skeleton is not preserved, then portions of the object can disappear, such as the upper-right part of this volume.

Although the theory for this fattening method is not much more sophisticated than for the thinning one, the implementation is significantly more complex. For positive offsets, the surface extraction algorithm stops advancing where it detects an intersection with any single sphere. For negative offsets, on the other hand, the surface extraction method must detect where it leaves the union of spheres. This means that the advancing front method must keep track of what spheres the front is lying within.

Our technique works by examining the intersections between the edge and each sphere under consideration. Since portions of the intersection that lie outside the original volume $\mathcal{A}$ are necessarily outside the shrunken one, we ignore them.

$$edgeIntersection(\mathcal{A}_W, p_0, p_1) = \mathcal{A}_W \cap segment(p_0, p_1) \cap \bigcup_{a \in \partial\mathcal{A}_W} sphere(a, \delta) \tag{3.16}$$

This intersection may contain several disconnected intervals, each of which we would like to examine separately.

In order to split the intersection into disconnected intervals, let a subset of a region be called *full* if it does not contain any point outside of that region. Furthermore, a *maximal full* subset is both full and is not contained within any other full subset of the region. The set of maximum full subsets represents the disconnected intervals.

Given this set of intervals, we would like to decide which one is closest to the starting point of an edge. Generally, the closest interval to a subset P is the one having the smallest minimum distance from P.

$$closest(P, Q) = q : \left(q \in Q, distance(P, q) = \min_{r \in Q} distance(P, r)\right) \tag{3.17}$$

Let the set of maximal intervals of $edgeIntersection(\mathcal{A}, p_0, p_1)$ be called $\sigma(\mathcal{A}_W, p_0, p_1)$. Then the farthest point of the closest member of $\sigma(\mathcal{A}, p_0, p_1)$ to p_0 is at the frontier of the union of spheres.

$$\sigma_c(\mathcal{A}, p_0, p_1) = closest\big(\{p_0\}, \sigma(\mathcal{A}_W, p_0, p_1)\big) \tag{3.18}$$

The distance to the offset surface is therefore equal to the distance to the farthest point in σ_C (**Figure 3.19**).

$$distToSurface(p_0, p_1) = \max_{p \in \sigma_C(\mathcal{A}, p_0, p_1)} \|p_0 - p\| \tag{3.19}$$

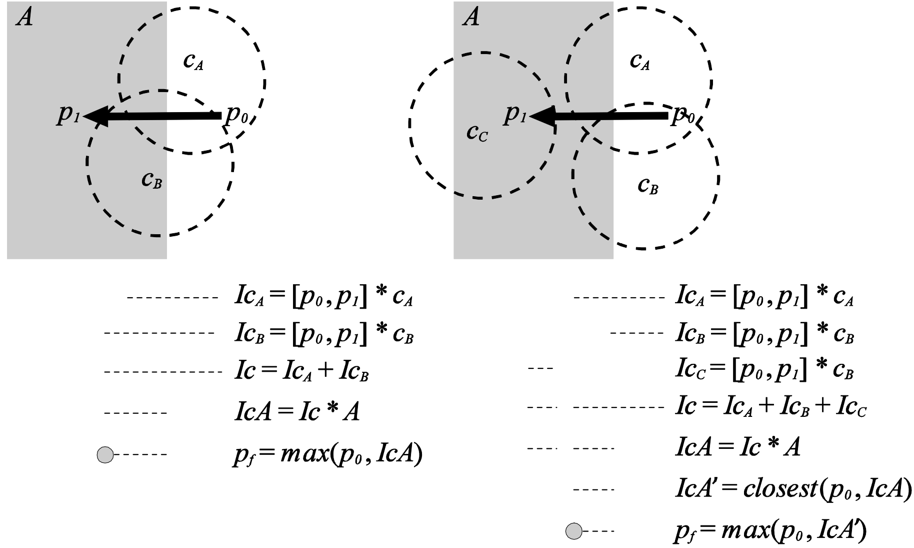

Figure 3.19: Negative offset surface extraction. To demonstrate how the negative offset operator is implemented within a surface extraction method, we give two examples here. The first (left) contains two spheres, c_A and c_B, the original volume A, and a line segment between points p_0 and p_1. Intersections are performed between each sphere and the line segment, yielding Ic_A and Ic_B (shown as dashed lines). Their union Ic is then intersected with the volume A to obtain the single interval IcA. Finally, the farthest point along IcA from p_0 is shown as p_f. The second example (right) is slightly more complex. Since c_C is disconnected from the two other spheres, IcA contains two separated intervals as well. Only the closest one, IcA' is taken for the final distance operation. This guarantees that the advancing front does not "jump" from one sphere to another.

3.5 Experimental Results

We have implemented the wrapping algorithm in C++, with graphic routines in OpenGL. It is designed as a module for Kineo Path Planner™, and benefits from its stable implementation of the bounded move operator. Since it relies only very basic graphic card functionality, any 2nd generation GPU supporting z-buffer and frame-buffer readback suffices.

All tests have been conducted on an Intel Core 2 Duo running at 2.66 GHz with 4GB RAM (the work is only done on one core, however). The GPU is an nVidia Quadro FX 4600 with 768 MB dedicated memory. The operating system is 64-bit Windows Vista.

3.5.1 Swept Volume

We have used 3 models for testing the swept volume: a car exhaust (**Figure 3.20**), a car seat (**Figure 3.21**), and a virtual human actor (**Figure 3.22**). All have come from actual PLM cases encountered by Kineo CAM™. The path being swept is the result of a path planning process.

Table 3.1 lists the number of triangles for each model, as well as the size of the Oriented Bounding Box (OBB) surrounding the swept path, which serves as the workspace for our algorithm. The performance results are shown in **Table 3.2**. Each model was calculated at two different error levels.

The visual quality of the SV mesh is quite high, and suitable for PLM. However, the triangle count of the SV is also important, and so in our application we pass the generated SVs directly through a simplification procedure, such as the vertex decimation algorithm presented in [Schroeder, et al. 1992].

Despite their appearance, none of the test models are watertight 2-manifold meshes. To further demonstrate the applicability of our algorithm to non- volumetric geometry, we sweep a simple flat surface in **Figure 3.23**.

Model	# Triangles	OBB Size (mm)
Seat	30765	2027 x 1578 x 1193
Exhaust	32641	1940 x 597 x 666
Human	55632	1667 x 1102 x 805

Table 3.1: Swept Volume Test Models.

Model	Parameters		Statistics					Performance		
	S (mm)	Δ (mm)	ε (mm)	# Samples	# Drawn Triangles	Grid Size	# Grid Points	Time (s)	Memory (MB)	# Produced Triangles
Exhaust	15	10	31	582	15.1G	130x47x25	275k	10.5	20	99.8k
Exhaust	7	6	15.1	970	48.1G	228x100x96	2.2M	52.8	146.3	181.9k
Seat	15	10	31	195	7.3G	136x106x80	1.2M	18.2	56.4	91.7k
Seat	10	6	20.3	325	18.3G	203x158x120	3.8M	52.7	201.5	451.7k
Human	15	10	31	693	24.1G	139x92x67	857k	26.4	28.5	55k
Human	10	6	20.3	1155	48.6G	167x111x81	1.5M	65.9	85.1	124.9k

Table 3.2: Swept Volume Experimental Results. Swept volumes for each of the three models are generated with two different pairs of parameters: the slice depth S and the bounding distance Δ. The error ε is directly determined by these two parameters. The chosen bounding distance Δ leads to a certain number of samples along the path. To gather distance fields, the object is drawn at each of these samples for each slice of the workspace, leading to a large number of drawn triangles. The grid size relates the number of grid point along each of the workspace axes, a result of the chosen slice depth S and the size of the workspace. The number of grid points is simply the product of the grid size. In terms of performance, the calculation time is listed along with the peak memory consumption and the number of triangles in the final model.

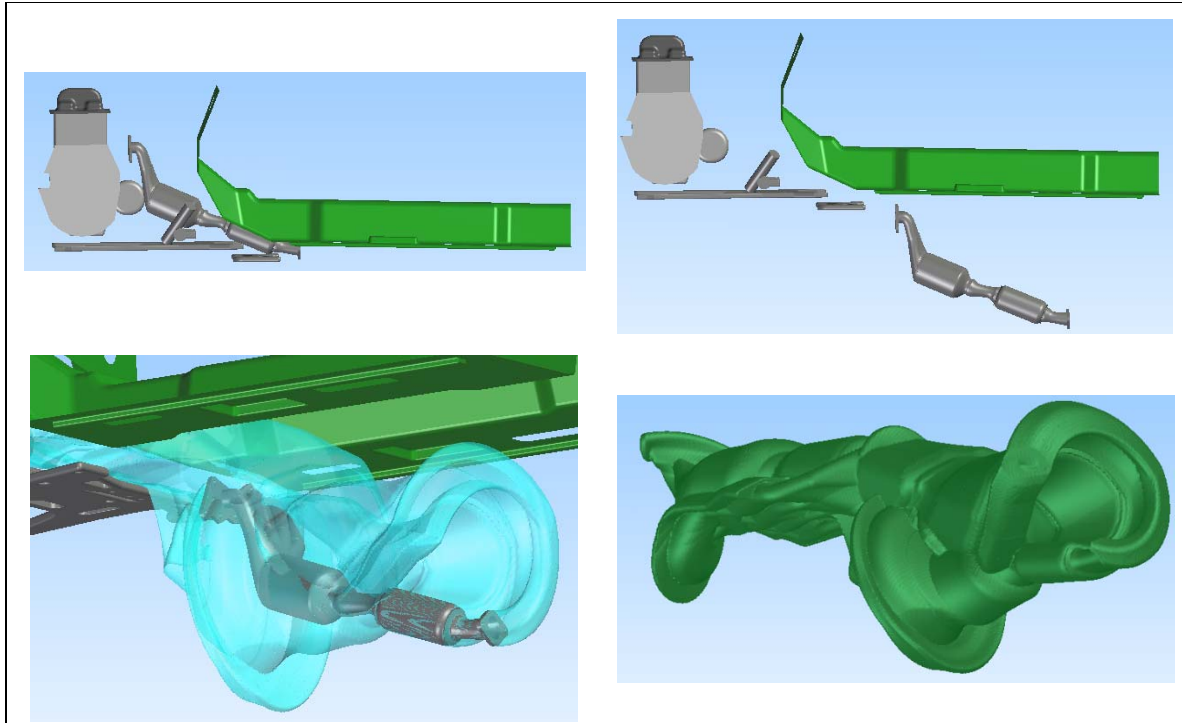

Figure 3.20: Exhaust test case. The *Exhaust* scenario is shown in the top and results in the SV shown as a blue transparent mesh in the lower left, or as a solid green object in the lower right.

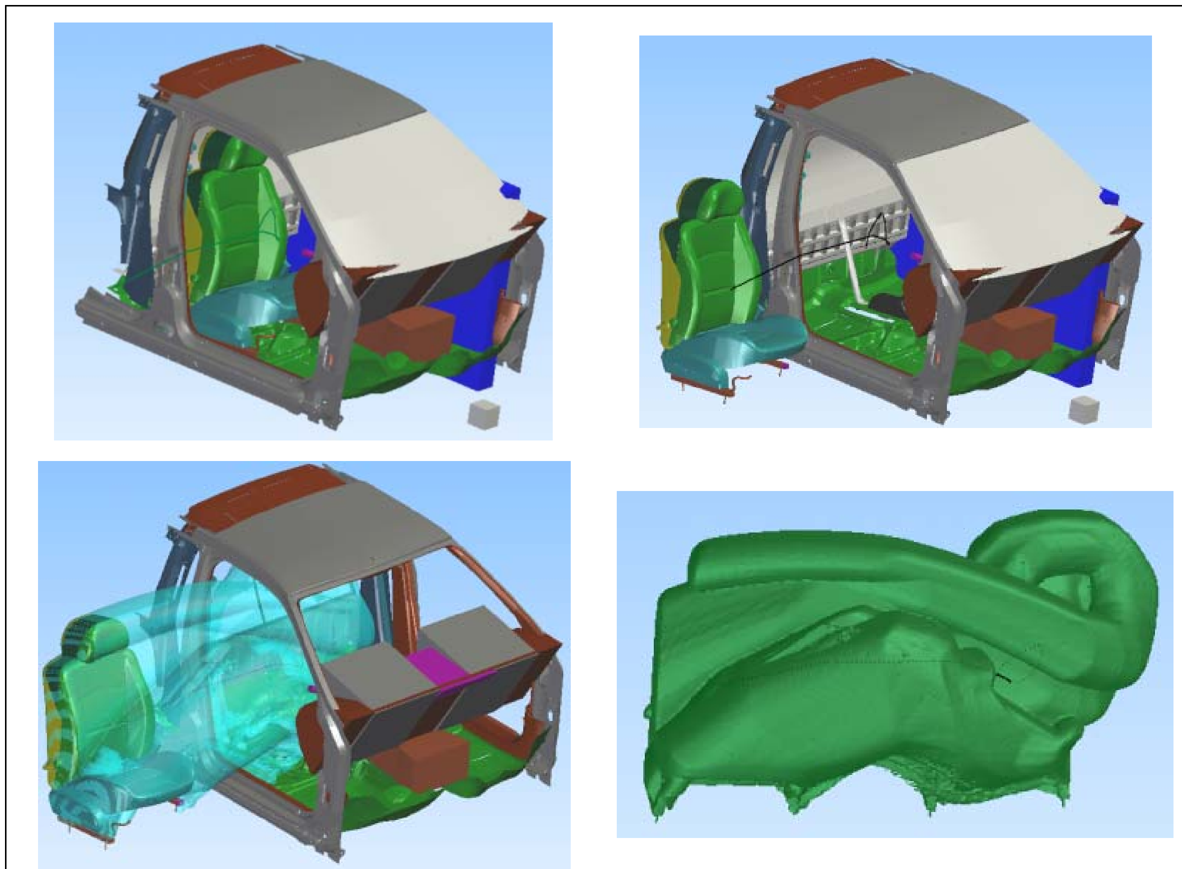

Figure 3.21: Seat test case. The *Seat* scenario is shown in the top and results in the SV shown as a blue transparent mesh in the lower left, or as a solid green object in the lower right.

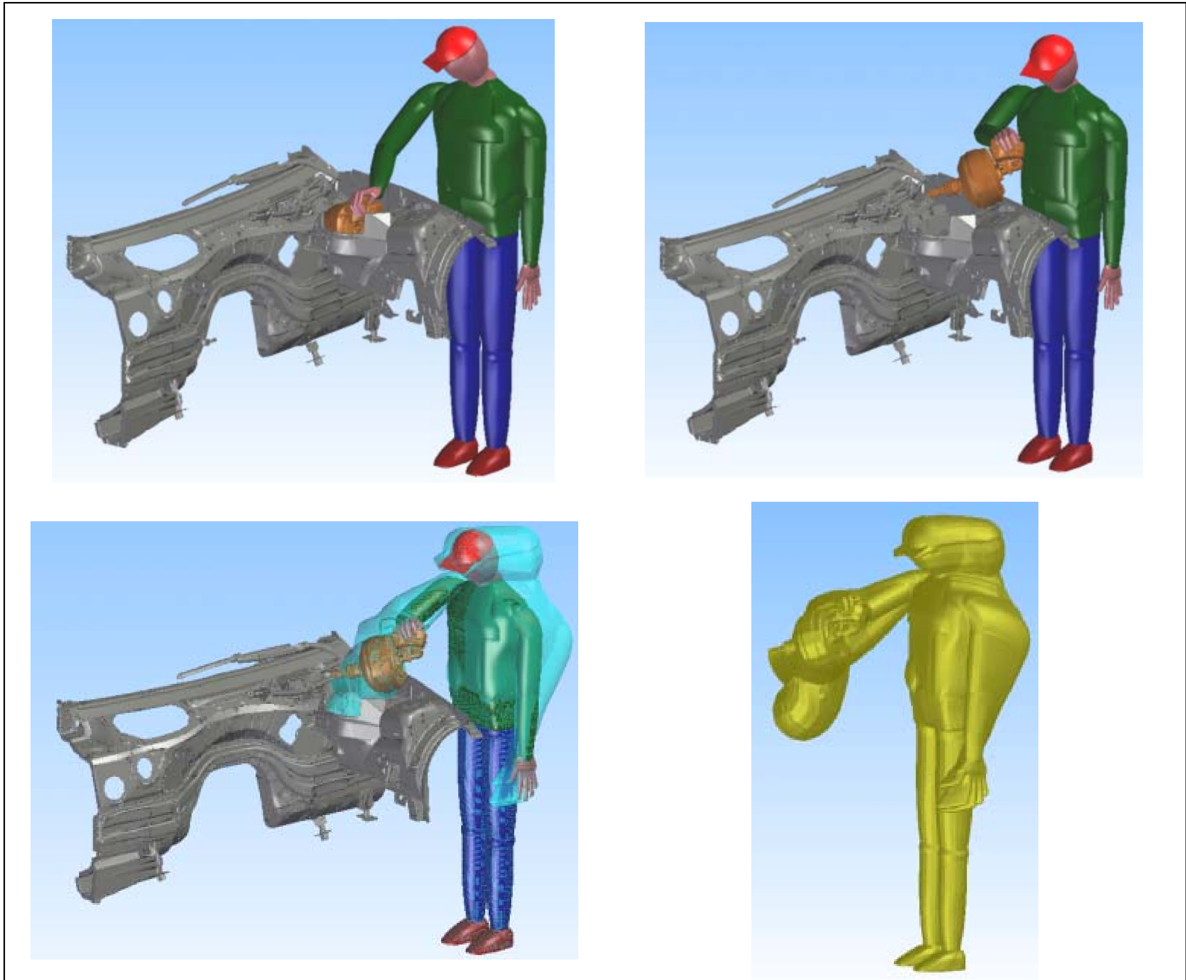

Figure 3.22: Human test case. The *Human* scenario is shown in the top and results in the SV shown as a blue transparent mesh in the lower left, or as a solid gold object in the lower right.

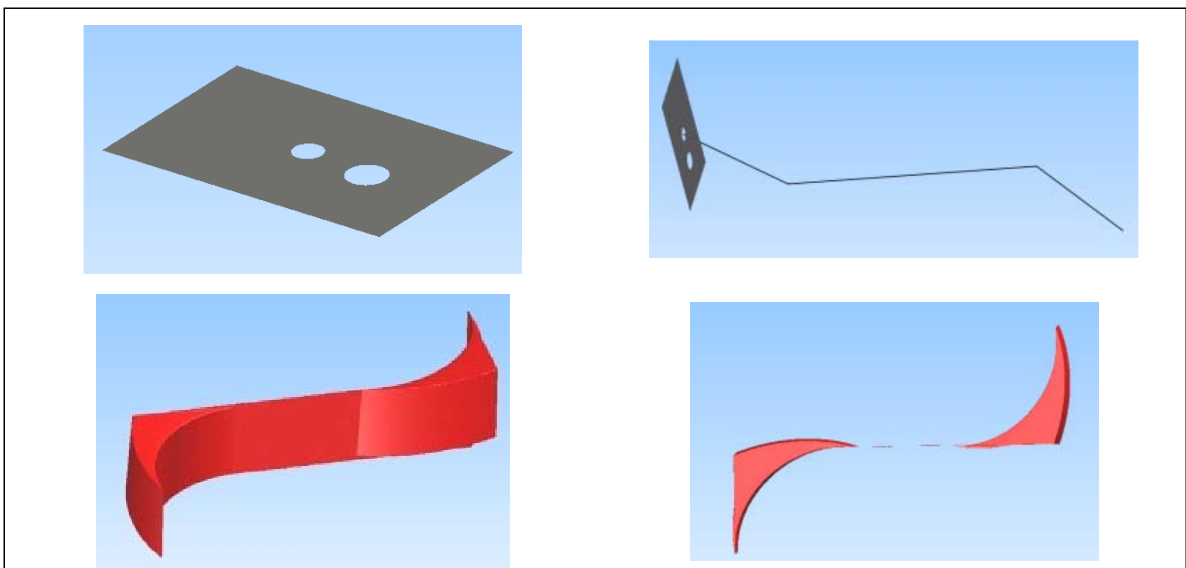

Figure 3.23: Flat surface example. The surface with two holes (first) is swept along an S-shaped curve. (second). The resulting SV (viewed from an angle in third, side in fourth) displays the flat surface in the middle as well as the volumes on the ends.

3.5.2 Offset

As the offsetting procedure can intervene in any wrapping context, we could simply test it on the swept volume examples just presented. But since it is a bit harder to observe the effects on large swept surfaces, we have chosen instead to present two simpler wrapping models followed by a single swept volume.

The first example is the well-known *Dragon* model (**Figure 3.24**), and the second is a mechanical piece from an automotive *Motor* (**Figure 3.25**). Finally, we introduce a mechanical disassembly problem involving a *Compressor* (**Figure 3.26**) for which we calculate the swept volume and offsets around it (**Figure 3.27**). More information on the models is given in **Table 3.3**, and experimental results in **Table 3.4**.

Model	# Triangles	OBB Size (mm)
Dragon	871k	209 x 165 x 91
Engine	437k	394 x 284 x 110
Compressor	120k	691 x 286 x 247

Table 3.3: Offset Test Models.

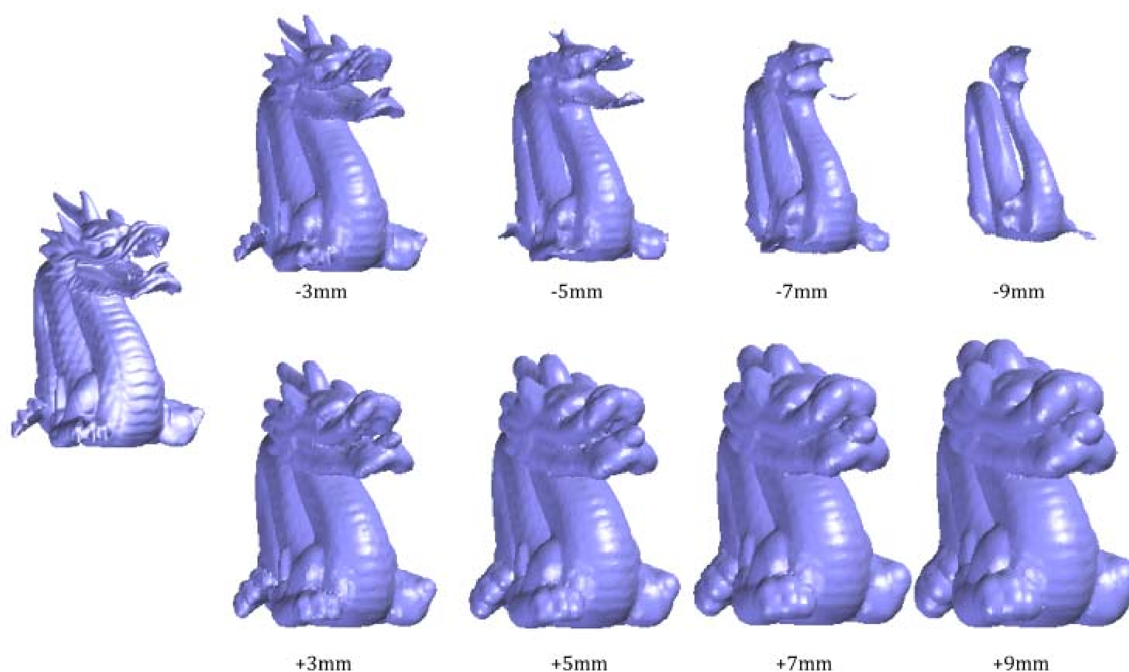

Figure 3.24: Offset dragons. The original model (left center) was deflated by four different amounts (top row) and inflated by the same amounts (bottom row). In this example, the grid cell size S is 2mm.

Model	Parameters			Statistics			Performance			
	S (mm)	Δ (mm)	δ (mm)	# Drawn Triangles	# Grid Points	# Spheres	Offset Time (s)	Wrapping Time (s)	Memory (MB)	# Produced Triangles
Dragon	2	N/A	3	871k	401k	26.5k	69.51	22.7	134.6	62k
Dragon	2	N/A	5	871k	401k	26.5k	131.27	22.79	135.7	67k
Dragon	2	N/A	-3	871k	401k	26.5k	175.16	19.15	152	40k
Dragon	2	N/A	-5	871k	401k	26.5k	448.38	18.78	238.8	29k
Engine	2	N/A	3	438k	1.55G	72.7k	267.03	23.57	325.8	154k
Engine	2	N/A	5	438k	1.55G	72.7k	503.23	23.56	336.7	162k
Engine	2	N/A	-3	438k	1.55G	72.7k	723.02	22.54	692.3	118k
Engine	2	N/A	-5	438k	1.55G	72.7k	1797.06	21.73	807	87k
Compressor	5	1	3	100M	48.8M	22.1k	9.47	73.96	86.1	44k
Compressor	5	1	6	100M	48.8M	22.1k	39.79	73.89	97.2	45k
Compressor	5	1	-3	100M	48.8M	22.1k	27.17	74.15	152.3	61k
Compressor	5	1	6	100M	48.8M	22.1k	106.08	73.74	170.3	42k

Table 3.4: Offset Experimental Results. Offset wrappings for each of the models are generated with different combinations of three parameters: the slice depth S, the bounding distance Δ (only applicable to the swept volume *Seat* case), and the offset distance δ. The number of drawn triangles for non-swept wrappings is fairly low, but the number of grid points roughly determines the number of detected surface points, and therefore the number of spheres that must be taken into account to perform the offset. Finally, we give some performance results regarding time, memory consumption, and the triangle count of the final model. The offset time is separated out from the rest of the wrapping algorithm.

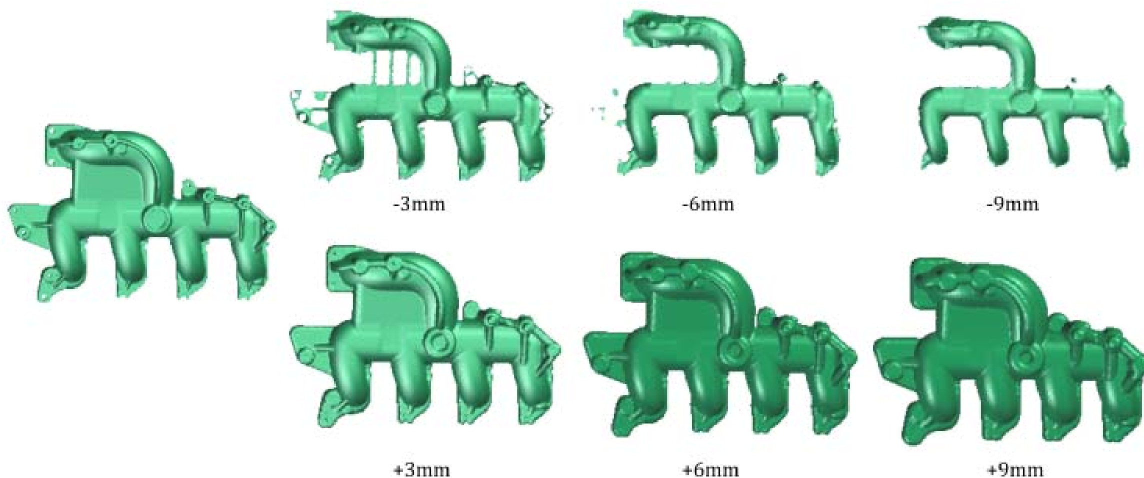

Figure 3.25: Offset motors. The original model (left center) was deflated three times (top row) and then inflated by the same amounts (bottom row). In this example, the grid cell size S is 2mm.

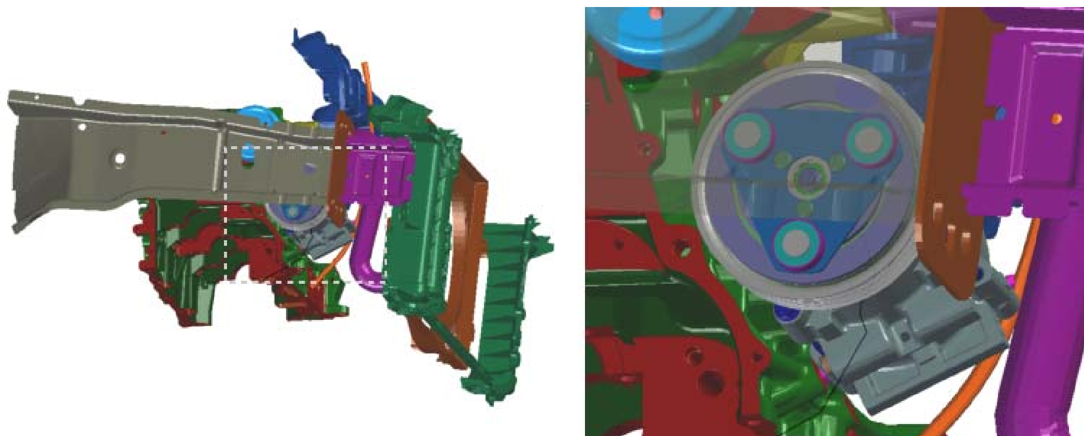

Figure 3.26: Compressor test case. In this example, the compressor must be removed from the assembly along the path (left). On the right, a close-up on the compressor, with the plate in front of it rendered with transparency.

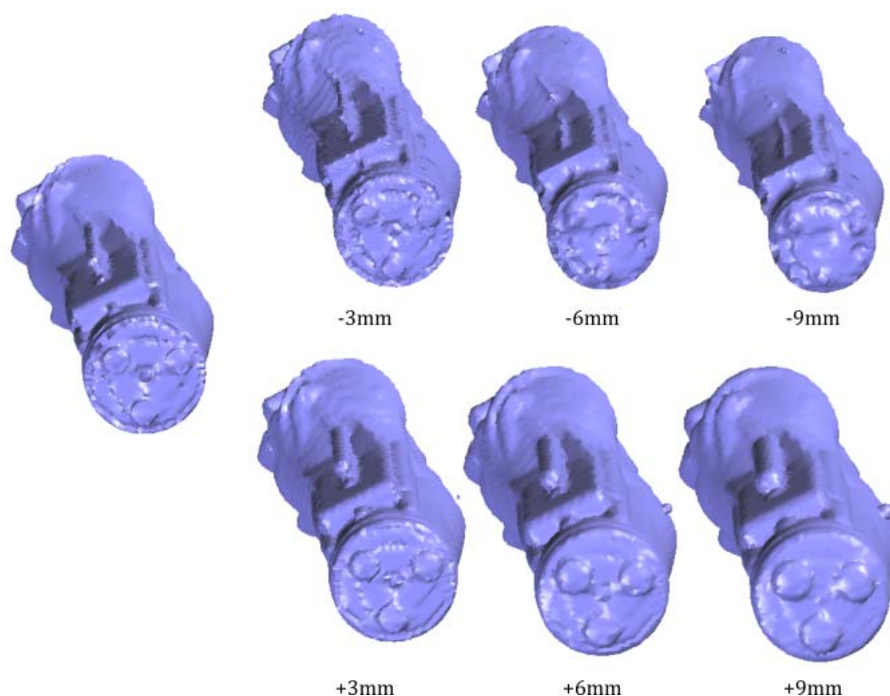

Figure 3.27: Offset compressors. The swept volume of the compressor (left center) was deflated three times (top row) and then inflated by the same amounts (bottom row). In this example, the grid cell size S is 5mm and the bounding distance Δ is 1mm.

3.6 Conclusion

Our wrapping approximation algorithm successfully deals with real challenges posed by PLM, including disassembly and ergonomic studies. Its fast execution allows for a rapid analysis of the given paths and for subsequent collision detection and path-planning requirements. By relaxing the requirements of watertight 2-manifold geometry, no pre-processing is needed to handle arbitrary CAD models without offsetting. If offsetting *is* desired, we can incorporate both inflation and deflation operators into the wrapping process.

There are several areas for future work. We have shown how the intermediate graph data structure, representing a volumetric mesh, has potential for manipulating the object before it takes on polygon soup form. Other algorithms that require closed watertight geometry could be run at this point. In particular, mesh simplification is a common second step to reduce the triangle count of the wrapped output, and it could be easier to run it on the graph then on the triangulated result. In addition, we would like to consider introducing sub-sampled points when feature geometry is detected, as is done by algorithms such as EMC.

Finally, our use of the GPU is quite limited (drawing polygons and reading from the z-buffer). It would be interesting to explore how applying modern GPGPU (General-Purpose GPU) techniques could both accelerate the performance and add new capabilities. For example, massive number of line-sphere intersection tests could be performed in parallel on the GPU.

One common use of swept volumes is to reserve space in mechanical designs for maintenance or reachability tasks. When someone changes the design, they can use a collision detector to verify that they have not encroached on the reserved space. In the next chapter, we discuss a general framework for collision detection that can handle not only the input and output to the wrapping operator, but a large variety of different geometry types.

"If it ain't broke, don't fix it!"

Bert Lance

4 Generalized Collision Detection

If probabilistic methods have enjoyed such success with complex path planning problems, it is in large part thanks to the collision detection procedures on which they stand. Since configuration space is so difficult to analyze in practice, probabilistic methods rely for a large part on trial-and-error. This is said not to deny that planning methods employ effective strategies in searching through the configuration space. But such strategy can only point the way to interesting areas to explore, and depend on the brute force of collision detection to clear the path.

Collision detection is a (if not *the*) limiting factor when it comes to path planning performance. It is therefore a sensitive subject, for which the implementation may prove to be just as important as the theory behind it. Once a collision detection library has been exhaustively poked, prodded, and tweaked to squeeze every last ounce of performance out of it, it is saner to leave it undisturbed.

Which explains why, when faced with a new collision detection matchup—voxels vs. polygons—we hesitated to simply create a new collision detector. And yet, in the name of science, we chose instead to open up the black box and extract the innards. The reason was simply enough— we realized that most of logic behind the routines was indifferent to the kind of geometry being tested. Why should we have to rewrite the entire engine for voxels when the difference appeared trivial?

Furthermore, collision detection is not only a limiting factor for path planning *performance*, but also for path planning *generality*. The methods that we reviewed in Section 2 deal strictly with configuration space. They are blissfully unaware of what the robot is composed of, or the mechanics of testing the validity of a configuration. As long as its collision detector can supply a response to its queries, the path planner is ready to set out exploring. If new geometry types can be seamlessly integrated into a single collision detector, then path planning for them is essentially free.

It is important to understand that the problem dealt with in this chapter is more concerned with software design then computational geometry. Nevertheless, we feel that its relevance for path planning warrants its presence here.

4.1 Motivation

Collision detection plays a critical role in many domains such as robotics, Product Lifecycle Management (PLM), computer graphics, and virtual environments. It touches on many active areas of research, including mobile robotics, robotic surgery, and humanoid robots. Performance is only second to robustness in the list of requirements for most collision detectors.

Such performance optimization may be achieved through specialization of the collision detection routines for the specific problem at hand. As a result, the number and variety of geometric data representations taken in account is usually quite restricted. Through intimate knowledge of the limitations, assumptions, and implications of both a given geometry type and its implementation, the collision detection code can avoid extraneous tests and optimize the remaining ones. If necessary, extra data structures may be built upon the geometry data to accelerate the process even more.

As a side-effect of allowing for the desired optimization, this process tightly binds the collision detection algorithm to the data representation. The tight coupling represents a barrier to algorithm reuse, a roadblock that becomes most apparent once a new geometry type must be added to the mix. In this situation, developers of a dedicated collision detector are faced with two imperfect choices: create a second collision detection algorithm specialized for the new collision tests (which must be independently tested and maintained), or convert the new data type to the old one before handing it off to the collision detector.

The second approach may seem appealing, but could in fact sacrifice many of the advantages of the “natural” model representation. There is such a variety of attributes for different geometry types that it is difficult to convert from one to another without losing some functionality. For example, we could convert from a voxel map into a polygon soup and its bounding volume hierarchy, which is quite capable of representing voxels as 3D boxes. But since this bounding volume hierarchy must be rebuilt each time the model changes, frequent updates of the voxel map would impose a significant performance penalty.

We decided upon a third approach, to create a software architecture that supports any number of geometry types as well as all the optimizations necessary to achieve high performance. Applications for this architecture include PLM, bioinformatics, as well as robotics. In this work, we present an example in the last domain. Section 4.7 presents an example of how dynamic voxel maps can be tested against static polygon soup models for a humanoid robot exploration scenario.

4.2 Related Work

The collision detection problem has been extensively studied in many different contexts. The tight performance constraints placed on a collision checker (both in terms of time and space complexity) have lead to specialized collision detection structures and algorithms for many

different geometry representations. For a presentation on the state of the art, we refer the reader to [Lin and Manocha 2004].

To handle the kind of Product Lifecycle Management (PLM) path planning problems discussed in [Laumond 2006], high performance Oriented Bounding Box (OBB) trees that deal with unstructured polygon soup models are ideal [Gottschalk 1998, Gottschalk, et al. 1996, Lin, et al. 1996]. Other bounding volumes that would work well include AABB trees [Bergen 1997, Larsson and Akenine-Möller 2005] and k-DOPs [Klosowski, et al. 1998]. Such structures can play a role even in a mobile robotics context. Since high-fidelity polygon soup models are often available for the mobile robot as well as certain obstacles in the robot's environment, they are a natural choice for the "static" portion of the robot's collision detection solution.

Our autonomous humanoid robot uses stereo vision and occupancy grids to construct a unified 3D environment [Braillon, et al. 2008, Elfes 1989]. For dynamic environments, voxel maps provide very fast collision detection [Gibson 1995]. By organizing them hierarchically, voxel maps can be dynamically updated in a memory and time efficient manner while the robot explores its environment [McNeely, et al. 1999].

Despite the impressive amount of collision detection research, we have been unable to find published literature on generalized frameworks for collision detection. Such a framework constitutes our contribution. In specific, we define a generic algorithm that descends a pair of bounding-volume trees in tandem while dispatching all concrete proximity tests to specialized handlers. Our architecture is designed to allow for customized data structures in order to achieve the same level of performance as a dedicated collision detector. Finally, we demonstrate how the framework can be integrated into a larger application around the common "scene graph" structure.

4.3 Collision Detection Framework

In order to illustrate how collision detection typically functions, we begin with the simple example of testing two polyhedrons against each other. Consider two polygon soup models *A* and *B*, composed of unordered sets of triangles. We would like to know if they collide or not.

As described in [LaValle 2006, Lin, et al. 1996, Quinlan 1994], to avoid testing each triangle from *A* against each from *B*, bounding volumes can be placed around each set of triangles. In this example, OBBs are used. Only if the OBBs of *A* and *B* overlap do we need to test each triangle against the others. **Algorithm 4.1** describes such a function, and **Figure 4.1** lays out the tests needed if *A* has 3 triangles and *B* only 2.

An example execution is the following: *A*'s OBB is tested against *B*'s, and they are found to overlap. *A*'s OBB is then checked against *B*'s first triangle. No overlap is detected, so the second triangle is tried. This time, they are found to overlap, so *A*'s triangles are checked one-by-one against *B*'s second triangle. At *A*'s third triangle an overlap is detected, and the algorithm terminates.

It is easy to see that in this reduced example we need three kinds of tests: OBB-OBB, OBB-triangle, and triangle-triangle. The `overlaps()` function in **Algorithm 4.1** would need to distinguish between them in order to carry out the correct calculation.

```
1.   function test(a, b) : Boolean
2.     Boolean collides ← false
3.     if overlaps(a, b)
4.       if isOBB(b)
5.         foreach c in children(b)
6.           collides ← collides or test(a, c)
7.         end foreach
8.       else if isOBB(a)
9.         foreach c in children(a)
10.          collides ← collides or test(c, b)
11.        end foreach
12.      else
13.        // both a and b are triangles
14.        collides ← true
15.      end if
16.    end if
17.    return collides
18.  end function
```

Algorithm 4.1: Simple polyhedron-polyhedron collision procedure. The types of *a* and *b* could be OBBs or triangles.

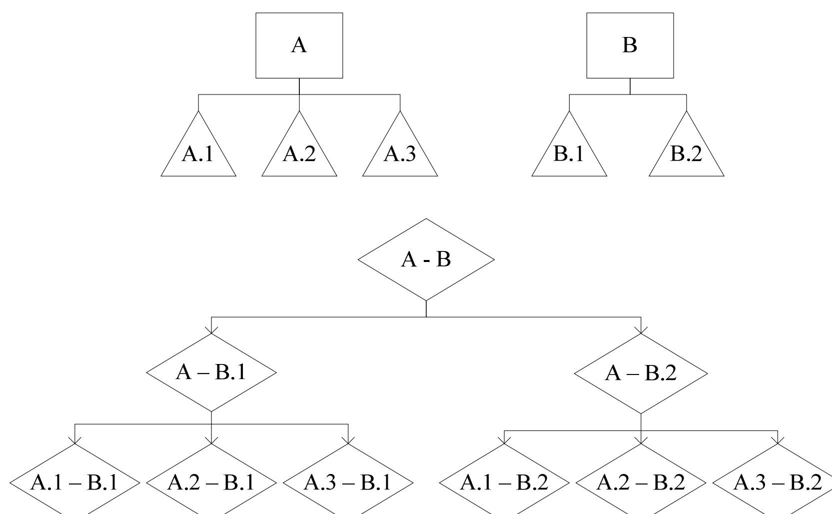

Figure 4.1: Testing polyhedrons. Take two sets of triangles, *A* and *B*, the first containing 3 triangles and the second only 2. Given a bounding volume around each set, it is possible to reduce the number of tests that must be executed in order to test for collision. First, the bounding volumes are tested against each other. If they overlap, then one of them (e.g. *A*) is tested against each triangle of the other. Only if an overlap is detected again are the triangles from *A* tested against the triangle of *B*. The execution can be represented by the bottom flow diagram.

Although the example just given applies only to polygon soups, many different geometry types employ hierarchical structures to carry out collision detection. In general, all bounding-volume and spatial partitioning techniques use a divide and conquer strategy [Lin and Manocha 2004]. Both bottom-up and top-down approaches result in the same type of hierarchical structure. In our framework, this hierarchical structure is termed a *test tree*, and is composed of *elements*, which can be leaves (triangles, in the previous example) or branches (OBBs). In order to support a wide range of geometries, a branch can have any number of children. Test trees are discussed further in Section 4.5.

Given the generic tree structure, it is possible to generalize the procedure presented in **Algorithm 4.1**. First, two elements are tested against each other. If no overlap is detected, then the function returns `false`. If the elements do overlap and they are both leaves (e.g. collision between two triangles), then the procedure simply returns `true`. Otherwise, it is necessary to explore further in the tree to determine if a collision exists. One of the two elements is chosen for expansion, and the procedure is called recursively for each of its children, or until a collision is found. We call this algorithm *test tree descent* and it forms the core of our framework. Section 4.4 presents it in greater detail.

In the previous example, we have only checked if *A* and *B* collide. Other common tasks include listing all of the collisions between *A* and *B* (i.e. a list of pairs of overlapping triangles), and finding the distance between *A* and *B* if they do not collide. Each of these tasks is an example of a *proximity query*, and modifies not only the test tree descent but also the kind of information returned by proximity tests between the elements. In the previous example, only a simple overlap test was required between elements, which might be faster than calculating the distance between them.

As a whole, the architecture of the framework can be decomposed into three parts (**Figure 4.2**). The core is the test tree descent algorithm, which is immutable and applies equally to test trees of any geometry. It uses the test tree element interface to traverse the test trees.

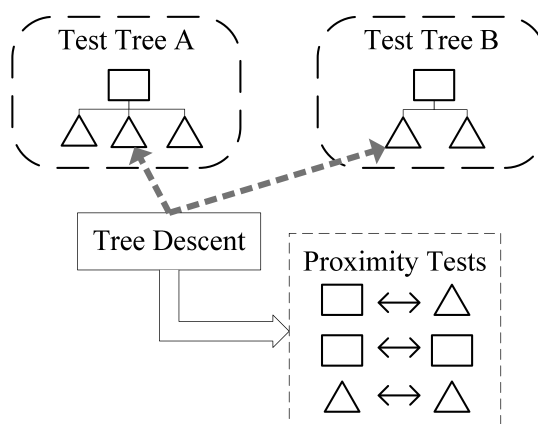

Figure 4.2: Framework architecture. The tree descent algorithm always references two elements at a time, one from each test tree. To test for collision and distances between elements, it calls on one of a number of proximity tests, organized into a bank. Each proximity test is specialized to analyze the interaction between two types of geometric objects. In the diagram, the dashed borders around the test trees and the proximity tests are used to indicate that the user can extend those portions of the architecture.

For all tests between test tree elements, the tree descent algorithm refers to a bank of proximity tests. A proximity test generally takes two elements as input and outputs the distance between them. The result of the test depends on the proximity query posed. The proximity tests are organized by the type of geometrical elements that they take as input (e.g. OBB-triangle, or triangle-triangle).

4.4 Test Tree Descent

Through a defined test tree element interface and a bank of user-defined proximity tests, the test tree descent algorithm can be cleanly separated from the data upon which it functions. It consists only of generic logic, and requires no direct modification from the user.

4.4.1 Dispatch and Detection

Given two test tree elements, we must be able to analyze their interaction for collision and distance results. Since the test tree elements do not necessarily have knowledge of each other, this is an example of a multiple dispatch problem. Approaches to resolve this problem vary by programming language. For C++, [Pescio 1998] discusses it and offers several new solutions.

In addition to the classic definition, however, we have the requirement that the proximity tests (program logic) should be separated from test tree elements (data structures) in order to define multiple tests between elements. If multiple proximity tests exist for the same pair of elements, then the user should be able to decide, at runtime, which ones are used. Such dynamism facilitates the implementation of custom collision detection logic, but prevents us from implementing the multiple dispatch using methods based on C++ templates, which would hardcode the dispatch logic during compilation.

Our solution is based on a simple function table, indexed by element type (**Figure 4.3**), that dispatches proximity tests at runtime. The user can register and unregister proximity test objects with the dispatcher at runtime. For any pair of test tree elements, a single lookup retrieves the address of the object whose virtual function handles the given pair. Since C++ provides only weak support for reflection, a virtual method was added to the test tree element interface that receives a unique identifier, assigned by the dispatcher upon registration.

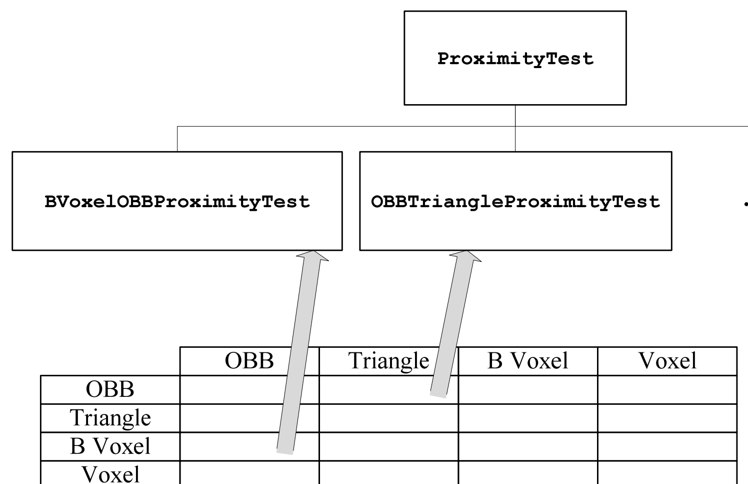

Figure 4.3: Proximity test dispatch mechanism. Each proximity test is a C++ class that inherits from a base class `ProximityTest` who defines a virtual method for performing the test. In the class diagram above, two example proximity test classes are defined: `OBBTriangleProximityTest`, and `BVoxelOBBProximityTest`. The dispatcher (bottom) has a table of pointers to the proximity test classes, organized by the type of test tree elements that the test can handle. To compare two elements, the dispatcher looks up the two types in the table and calls the corresponding virtual function on the selected proximity test. In this example, the `OBBTriangleProximityTest` handles an OBB as the left element and a polyhedron triangle as the right. The `BVoxelOBBProximityTest` does the same for bounding voxels and OBBs.

4.4.2 Tree Traversal

Tree descent follows a simple recursive pattern, addressing two elements at a time, one from each test tree. First, the proper detector executes a proximity test on a pair of elements. Based on the result of the test, and the proximity query chosen, the tree descent algorithm may choose to stop, back up, or proceed further down the test trees.

The generality of our framework derives from the fact that the tree descent algorithm is wholly ignorant of the type of data it is dealing with. All special knowledge of the geometry concerned is handled by the proximity tests, which can themselves be dynamically substituted for each other at run time. For a more complete description, **Algorithm 4.2** lists pseudo-code for the procedure.

4.5 Test Tree Structure

Test tree elements are two-faced. They must implement a common interface to allow the descent algorithm to traverse the tree in a generic fashion. They also must provide specialized information to the proximity tests that handle them. Along with the proximity tests, they allow the developer to include as little or as much optimization as needed for the application.

4.5.1 Generic Traversal

Examination of the pseudo-code in **Algorithm 4.2** reveals that only one element from each test tree is referenced by the tree descent algorithm at one time. Additionally, the test trees are not traversed in a random fashion. Instead, the algorithm starts at the root nodes and then either references the first child of an element or its next sibling. We can exploit this restricted access pattern to allow for time and memory optimizations.

We define a simple element interface that only allows three methods for tree traversal to access other elements: `firstChild()`, `nextSibling()`, and `parent()`. Their meanings are illustrated in **Figure 4.4**. Each method returns a reference to another element, which is used from then on. The methods `hasChildren()` and `hasNextSibling()` simply provide information about the existence of related elements.

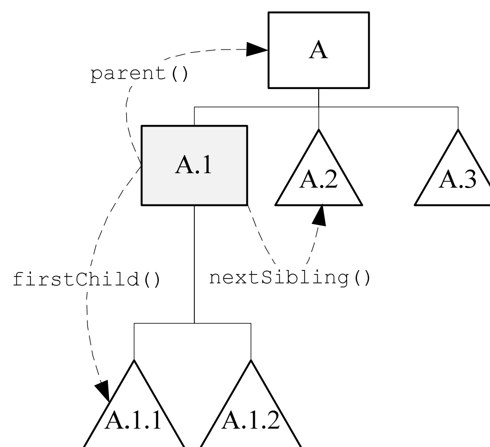

Figure 4.4: Element relations and traversal methods. In the restricted tree traversal pattern used, the highlighted element, *A.1*, is linked to other elements in the tree only through three relations: `parent()`, `firstChild()`, and `nextSibling()`.

To illustrate these methods, let us take the example of an OBB tree. In order to open the custom data structure up to the tree descent procedure, we define a test tree that contains two types of elements: OBBs and triangles. An OBB element always responds true to `hasChildren()`, and may return references to triangles or to other OBBs when `firstChild()` is called. A triangle element, on the other hand, always replies false to `hasChildren()`. Both elements may or may not have siblings, and respond accordingly.

4.5.2 Memory Optimization

Since only one element from a test tree is used at any one time, the entire tree of element objects need never be constructed in its entirety. Therefore, it is possible to design tree structures that are constructed on the fly, or that change dynamically.

Another advantage of the thin tree element interface is its proxy support. With this approach, the data associated with a test tree is not stored as individual element objects. Instead, the geometry data is stored separately, often in some optimized fashion. The element objects themselves simply point to sections of this data store. Not only does this allow the designer to optimize the data storage as needed, but the element objects themselves can be reused.

To illustrate, consider once again the OBB tree example. Upon construction, the triangles are sorted by a space-partitioning algorithm to build a binary tree of OBBs. To optimize memory, the OBBs are stored in a large array, with compressed descriptions of their positions and indexes of their children. Triangles are stored in a similar fashion. With this setup, the OBB and triangle elements can simply store the indices relative to their respective arrays. When a traversal method is called, the element can look up the data at that index, unpacking it if necessary, and then answer the request (**Figure 4.5**).

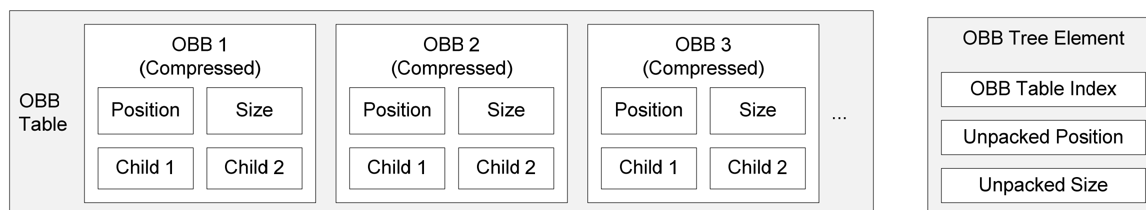

Figure 4.5: Compressed storage scheme for OBBs. In order to save space, a designer may wish to specify a custom storage scheme for their data. Here, all OBBs are stored in a packed format in a single table (left). Each OBB in the table has two indices or pointers to child OBBs within the same table. The collision detection architecture has no knowledge of this custom format. Instead, a tree element is defined which is capable of providing the uncompressed version to the proximity tests designed for OBBs (right). The tree element simply contains an index to the element in the table that it points to, thus avoiding duplicating the geometry data.

In order to save memory, the test tree used in our example constructs only one OBB element and one triangle element. Instead of always returning a reference to another element object in response to a traversal method call, these “singleton” elements only return references to themselves or to each other. Consider what happens when `firstChild()` is called on an OBB element. If the child is also an OBB, then the element only needs to update its own internal reference and simply return a reference to itself. If the child is a triangle, then it retrieves the triangle element of the test tree and updates the internal data of the triangle before returning a

reference to it. The other two tree traversal methods, `nextSibling()` and `parent()`, are implemented in a similar fashion.

In this way, only two element objects (one for OBBs and one for triangles) are needed to act as the entire test tree, and they can be allocated before the tree descent begins, saving both time and memory.

4.6 Application Integration

Collision detectors are rarely used in isolation. Instead, they are integrated into larger applications for computer graphics, robotics, PLM, etc. A common denominator of these applications is a hierarchical organization of objects, called a *scene graph*. In this section we discuss the scene graph, its relation to test trees, and the use of aggregate test trees.

4.6.1 Scene Graph

A scene graph is a hierarchal organization of objects related to 3D rendering. Leaves are generally geometric objects. Branch nodes in the tree are called *assemblies*, and by manipulating them, the user implicitly manipulates the child elements as well. As far as geometry elements are concerned, this means that each is assigned a position relative to its parent. To find the absolute position of an object, it is necessary to compose the relative positions of its lineage, up to the root of the hierarchy. Such an organization lends itself easily to kinematic chains.

A scene graph is both logical and dynamic. These two properties make it well adapted to user manipulation but also less suitable for collision detection. Since the scene graph is a logical rather than spatial organization, adjacent elements in the tree are not necessarily next to each other. Being able to add and remove elements within a scene graph makes it difficult to keep structures specific for collision detection (such as an OBB tree) up to date.

Take the example of a single polyhedron *A* that we would like to test repeatedly against a static scene composed of two other polyhedrons: *B* and *C*. It would be faster to group *B* and *C* together before collision detection begins, rather than testing *A* against *B* and then *A* against *C* at each step. By grouping, we mean building an OBB tree upon the union of both polyhedrons (by considering the triangles of both during OBB construction). To this end, we introduce the notion of a *collision entity*.

4.6.2 Collision Entities

When the user wishes to run collision detection in our framework, he creates test trees that integrate one or more geometric objects in the scene graph. The test tree is thus a separate and parallel construct to the scene graph, containing a reference to an element (geometry or assembly) in the scene graph, which then becomes a *collision entity* (**Figure 4.6**). In order to enforce consistency between the test trees and scene graphs, it is illegal to modify geometries on or beneath a collision entity, and our framework blocks this action when detected.

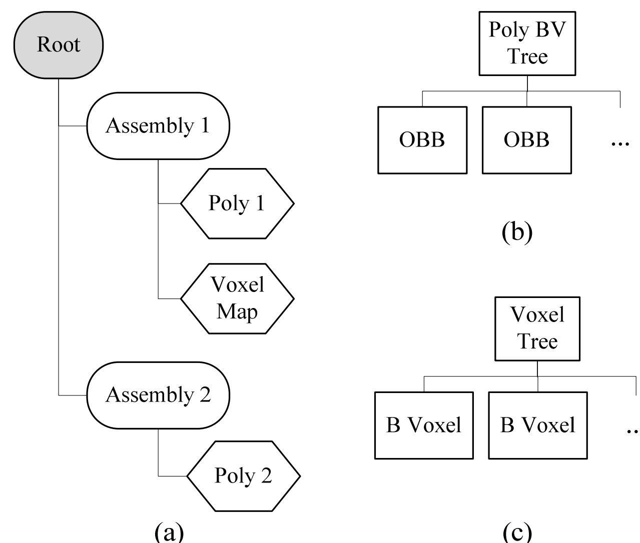

Figure 4.6: Scene trees. The scene graph (a) contains three geometric objects shown as hexagons. They are grouped in logical assemblies, shown as rounded rectangles, which allow the user to manipulate them as a whole. This structure is not necessarily optimal for collision detection, and so our framework builds a test tree based on the geometric types of the object. In this case, Poly 1 and Poly 2 could be put into one or more "Poly BV Trees" (b) and the voxel map in a "Voxel Tree" (c). In this simple example, both test trees have the scene root as their collision entity, although this is not a necessary condition.

It is during test tree construction that any expensive pre-computation operations are performed (such as recursive partitioning). The user may remove and rebuild collision entities as he sees fit, in order to balance performance (putting multiple geometries in a single test tree to boost query execution) and flexibility (geometries integrated in a test tree are immutable).

4.6.3 Test Tree Construction

As described above, collision tests are run on test trees instead of directly upon geometric elements in the scene graph. It was necessary to devise a simple and generic method to build a test tree from a collision entity (an object in the scene graph that serves as the base for the test tree). The central idea is that a test tree built upon a single geometry object contains only that object, and a test tree built upon an assembly contains all the geometry objects beneath that assembly.

The construction procedure begins with an empty test tree. Beginning at the given collision entity, it traverses the scene graph in a depth– or breadth–first manner. At each geometrical object, it asks the test tree to integrate it if possible. The procedure sends two additional messages before and after the traversal to provide the test tree opportunities to initialize and finalize its internal data structures.

For example, consider a test tree that assembles polyhedrons in an OBB tree. It would decline to accept objects other than polyhedrons, and contain a set of triangles which is initially empty. When the construction procedure asks the test tree to integrate a polyhedron, the test tree simply adds all the given triangles to the set. Upon receiving the finalization message, the test tree constructs an OBB tree around the set of triangles.

In certain scene graphs, an assembly may be the parent of objects of heterogeneous types. To allow such an assembly to be made into a collision entity, we use a *mixed test tree*. A

mixed test tree contains multiple test trees, each of which accepts complementary types of geometry objects. When an object is put up for integration, the mixed test tree proposes it to each enclosed test tree in turn, and integrates it into the first that accepts it. Finally, empty trees are trimmed once the construction is complete. During the collision detection process, the mixed test tree element will reference the root elements of its enclosed test trees.

4.6.4 Aggregate Test Trees

A geometry object in the scene graph may only be integrated into a single test tree, and only two test trees may be tested against each other at once. We would nonetheless like to group objects together into sets, in order to test one set against another. An *aggregate test tree* allows us to accomplish just that.

An aggregate test tree is created upon collision entity construction (and so should not be confused with the mixed test tree discussed above). Instead, they are created at test time and are meant to be disposed of once the test is complete, so as to avoid inconsistencies resulting from construction or removal of the referenced collision entities.

There are two aggregate test trees built into our framework. The first emulates the traversal of the scene graph starting from a given element. It contains a reference to the scene graph element, initially set to its “root” element, which is under examination. When its traversal methods are called, it inspects the referenced element and traverses the scene graph in a similar manner, until it encounters a collision entity. If `firstChild()` is called upon it at that point, it returns the root element of the test tree anchored at the referenced object (**Figure 4.7**).

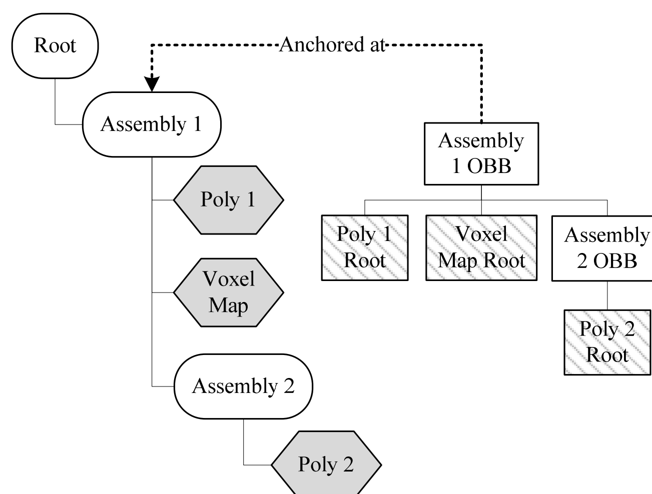

Figure 4.7: Aggregate test tree. In the scene graph on the left, there are 3 collision entities, shaded in gray. To assemble them all temporarily, they are collected in an aggregate tree on the right. Those elements in the scene test tree that are collision entities are represented by the root of their test tree.

The second aggregate represents a list, with a reference initially pointing to the head of the list. The `nextSibling()` method advances the reference. Just as for the previous aggregate, the `firstChild()` method returns the root element of the referenced test tree. Such a structure is especially useful for grouping collision entities that are separated from each other hierarchically but not spatially.

Between these two aggregate test trees, objects can be tested together, sacrificing some performance for the flexibility derived from not combining them in the same tree. At the same time, the tree descent algorithm need not have any knowledge of these structures, since they transparently act as any other test tree.

4.7 Dynamic Voxel Map for Robotic Vision System

A critical issue in autonomous robotics is to provide perception-based geometric models for automated motion planning and control. In such a context the system may consider various types of geometric models (occupancy grids, closed polyedra, polygons soups, voxels, etc.) according to the choice of sensors. Nevertheless, an accurate CAD model of the robot itself is often available to the system designers. In such a case, an astute collision detector could address heterogeneous data structures in order to minimize sensing error while optimizing performance.

We conducted an experiment involving HRP-2 (**Figure 4.8**), a humanoid robot [Inamura, et al. 2006, Yokoi, et al. 2008]. The goal is to allow the robot to autonomously explore unknown environments using stereo vision and probabilistic path planning techniques. By testing a hybrid collision detection scheme against a homogenous polyhedral one, we can demonstrate the effectiveness of our approach.

Figure 4.8: HRP-2, a humanoid robot. HRP-2 is 154 cm tall, and weights 58 kg with batteries. It has over 30 DOFs and is equipped for both stereo vision and force sensing.

4.7.1 3D Reconstruction

To represent the environment, we use a 3D-occupancy grid which is frequently refreshed with information from HPR-2's cameras [Braillon, et al. 2008, Elfes 1989]. Each grid cell is assigned a triplet representing the probabilities that it contains an obstacle, free space, or is indeterminate. Originally, all cells are considered indeterminate. Based on these values, the grid cells are classified into one of three discrete categories: OBSTACLE, UNKNOWN or FREE (**Figure 4.9**). The UNKNOWN category may refer to a cell that the robot has not yet observed (e.g. it is behind an obstacle) or is unsure about (e.g. not enough 3D points have been gathered in that volume).

As the robot moves, it gradually discovers more of the environment around it. Additionally, the environment itself may change. In both cases, the robot takes the new

information into account in the 3D model. Apart from its own position and that of its goal, all information about the environment is derived from its stereo vision. To carry out the task, it proceeds as follows:

First, it examines the environment through taking hundreds of images. Next, it uses the vision data to classify the 3D grid cells, creating two occupancy grids: one for OBSTACLE cells, and the other for UNKNOWN (free space is defined as the absence of grid cells). The robot then searches for a path delivering it to the goal, without considering the UNKNOWN grid. Assuming that a path is found, it is examined to determine if it enters into the UNKNOWN area. If the path does not touch UNKNOWN, the problem is solved and the robot can reach the goal. Otherwise, it walks up to the border of the UNKNOWN area and stops to take more photos. This information is used to update the occupancy grids, and the process continues iteratively.

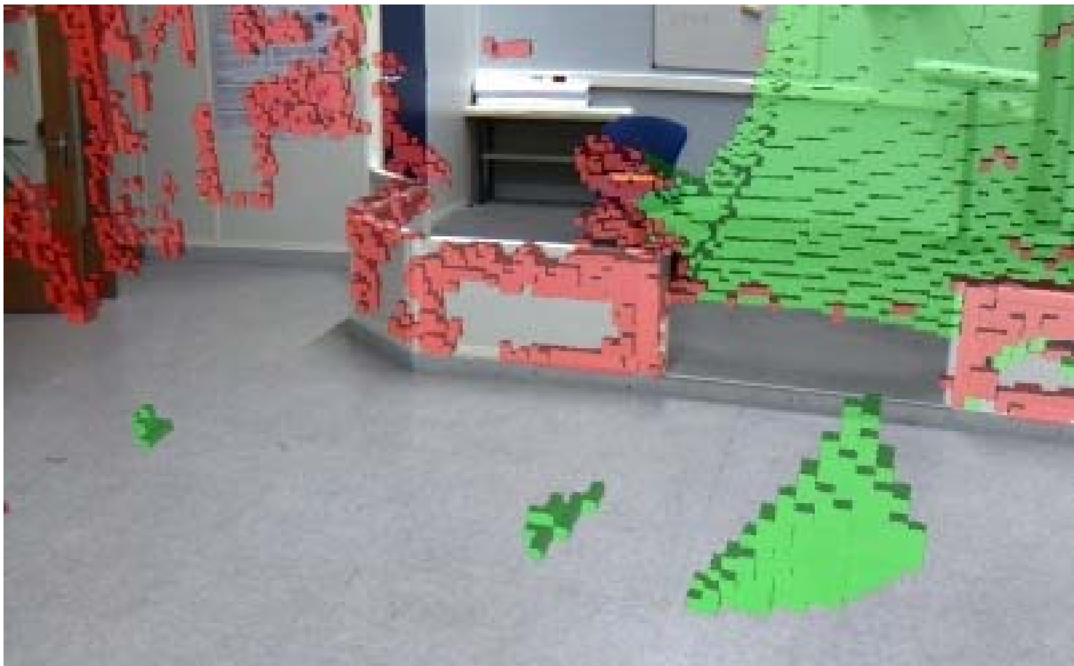

Figure 4.9: Occupancy grid. Data from several stereo images are combined to calculate the probabilities that a grid cell contains an obstacle. Here, the calculated grid cells are superimposed upon an acquired image. The red cells represent obstacles, and the green cells are unknown. The absence of cells indicates that the space is considered free.

4.7.2 Dynamic Voxel Map

A natural representation of the 3D occupancy grid is a voxel map. Such a space-partitioning method can benefit from a hierarchical organization, with larger voxels bounding a predetermined set of smaller ones [McNeely, et al. 1999].

One advantage of voxel maps is the efficient manner in which they can be updated. Removing or adding a voxel at the lowest level sends a message to the parent voxel informing of a change. In this way, bounding voxels can be created and removed on the fly in order to assure consistency (**Figure 4.10**).

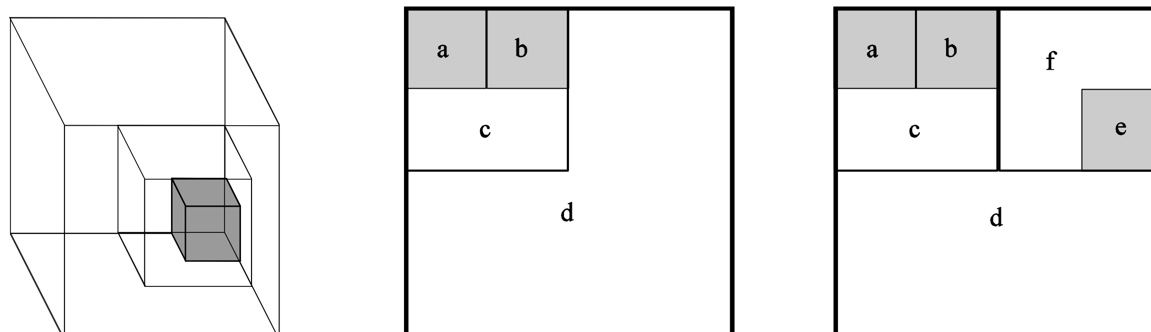

Figure 4.10: Voxel hierarchy. The left is a 3D view of 3 different levels of voxel sizes, one inside the other. In the center, we switch to 2D to show two primitive voxels (*a* and *b*) were detected by the vision system. Each level of the hierarchy (here there are three) contains a larger voxel bounding those below it— *c* contains both primitives on the second level and *d* contains *c* on the third. On the right, a third primitive voxel *e* is detected. Automatically, the bounding voxel *f* is created around it on the second level. No modification on the third level of the hierarchy is needed in this case.

Just as polyhedron objects in the scene graph may be transformed into an OBB tree for collision detection, so can voxel map objects be transformed into a voxel map test tree. However, unlike the OBB tree construction process, the voxel map test tree requires no pre-computation. Since it merely references the included voxel maps, it does not need to be rebuilt each time a change occurs.

Indeed, removing or adding a voxel to the map immediately affects the structure of the corresponding test tree as it is gradually exposed during tree descent. Such ability exploits the flexibility of this architecture in supporting both static and dynamic structures.

4.7.3 Experimental Design

Since an accurate and precise polygon soup model of HPR-2 is available we would like to use it for collision detection. The occupancy grid, however, could have multiple representations. By tessellating the boxes formed by the grid cells, the grid can be converted into a polygon soup model for use by a dedicated polygon soup collision detector. Using a dynamic voxel map, however, requires a hybrid voxel-polygon collision detector. In our experiment, we tested the performance of the two collision detectors.

The robot was placed in a typical exploration scenario, in which it maneuvers around an obstacle in order to reach its goal position (the environment measures 6x6x1m). Since the obstacle partially blocks the robot's view of the scene, it takes three iterations of the exploration process in order for the robot to complete its task (**Figure 4.11**).

In order to get consistent results for the collision detection performance, we pre-converted the image data into occupancy grid cells (20cm^3) for each of the three iterations. We measured the time and memory taken for each collision detector to process and initialize the occupancy grid data. Finally, we measured the time taken per collision test during the path planning process.

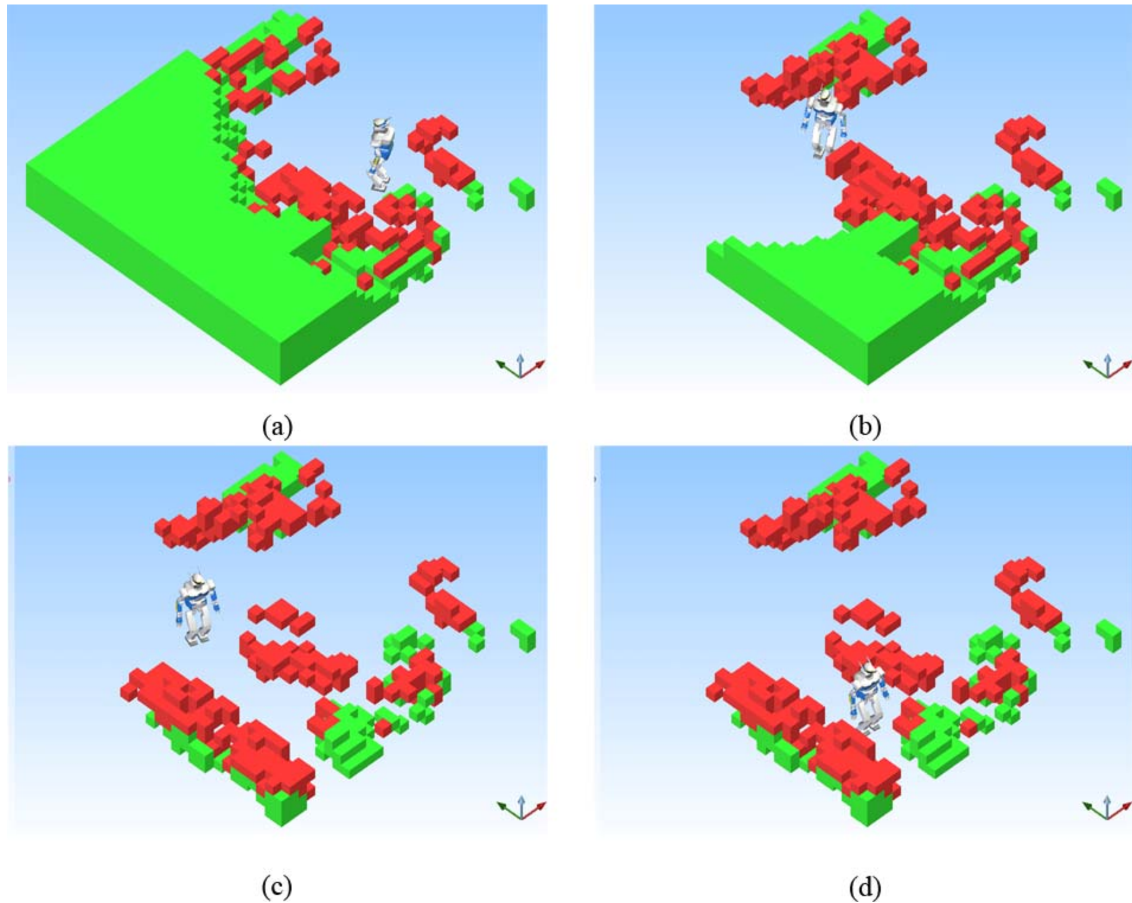

Figure 4.11: **Environment discovery in three steps.** When HPR-2 is placed at its initial position, it has no prior information of the environment. It builds a tentative representation of the environment from the acquired image data (a), and moves towards the boundary of the UNKNOWN area (green), while avoiding obstacles (red). Once there, it refines its estimation of the environment (b), and proceeds with the second iteration of its exploration algorithm. After this movement, it can perceive a clear path reaching the goal (c). By moving there (d), it completes the task.

4.7.4 Results

As explained in the last section, the experiment is divided into three iterations, each having a different number of grid cells for each category. **Table 4.1** presents the time needed to create the collision detection structures as well as the mean time per proximity test during the path planning process and the associated memory consumption.

Iteration	# Grid Cells		Type	Setup Time (ms)	Time per Collision Test (ms)	Memory (kB)
	Obstacle	Unknown				
1	147	2332	Voxel	20	0.924	74
			Poly	65 761	1.083	111
2	202	1012	Voxel	10	0.779	73
			Poly	17 626	1.292	92
3	284	306	Voxel	7	0.547	73
			Poly	5 304	1.712	83

Table 4.1: Time and Memory Performance.

Although our results show modest differences in memory consumption and collision test time, they might not convincingly argue for the use of a hybrid collision detector. The setup time, however, tells a more dramatic story. The voxel map is able to incorporate the changes in the occupancy grid hundreds, if not thousands, of times faster than the OBB tree. Simply put, an OBB tree derives its speed from pre-calculation step, while voxel maps are almost purely dynamic in nature.

4.8 Future Work

Our framework allows for high-performance collision detection between heterogeneous geometry types. By generalizing the collision detection problem and identifying its components, we have extracted a generic algorithm that descends a pair of hierarchical structures, using the results of dynamically-dispatched proximity tests to guide the search.

To satisfy the performance constraints placed on collision detection, elements in the hierarchical structures are only required to implement a thin interface that allows the implementation of a number of time and memory optimization schemes.

Finally, we validate our work on an autonomous humanoid robot exploring an unknown environment using stereo vision and an occupancy grid. Using our framework, we created a hybrid collision detector that tests voxels and polygons in order to dramatically decrease refresh times.

In the future, it would be interesting to explore the compatibility of this approach with geometric data structures that do not use bounding-volume or space-partitioning methods. Another question raised by this research is the feasibility of generalizing the proximity queries and the results that they return.

The major motivation for the development of a collision detection framework is to extend the capabilities of motion planning algorithms. In the next chapter, we leave collision detection behind in order to deal with a pure path-planning problem— permitting and controlling collisions during path planning.

```
1.  function startTest(treeA : TestTree, treeB : TestTree) : void
2.    test(root(treeA), root(treeB))
3.  end function
4.
5.  function test(a : Element, b : Element) : void
6.    detect(a, b)
7.
8.    Boolean childrenExist ← hasChildren(a) or hasChildren(b)
9.    if childrenExist and shouldContinue() and shouldDescend()
10.     if shouldDescendLeft(element(a), element(b))
11.       c ← firstChild(a)
12.       test(c, b)
13.       while shouldContinue() and hasNextSibling(c)
14.         c ← nextSibling(c)
15.         test(c, b)
16.       end while
17.       parent(c)
18.     else
19.       c ← firstChild(b)
20.       test(a, c)
21.       while shouldContinue() and hasNextSibling(c)
22.         c ← nextSibling(c)
23.         test(a, c)
24.       end while
25.       parent(c)
26.     end if
27.   end if
28. end function
29.
30. function shouldContinue() : Boolean
31.   return result() is not COLLISION or query().continueOnCollision()
32. end function
33.
34. function shouldDescend() : Boolean
35.   return result() is OVERLAP or query().continueOnDistance()
36. end function
37.
38. function shouldDescendLeft(a : Element, b : Element) : Boolean
39.   if not hasChildren(a)
40.     return false
41.   else if not hasChildren(b)
42.     return true
43.   else
44.     return heuristic(a) > heuristic(b)
45.   end if
46. end function
```

Algorithm 4.2: Generic tree descent. The function `test()` is recursive, and is originally called the root elements of the two test trees. The `detect()` function dispatches the proximity test to the proper handler. Finally, `query()` returns the current proximity query and `result()` the last proximity test result.

"Nature does not consist entirely, or even largely, of problems designed by a Grand Examiner to come out neatly in finite terms, and whatever subject we tackle the first need is to overcome timidity about approximating."

Harold B. Jeffreys

5 Motion Planning in Collision

By convention, if not by definition, motion planning is concerned with finding collision-free paths. When penetrations are allowed, as by IPP (presented in Chapter 2.3.1), they are likely temporary side-products of the planning process, meant to be dealt away with before the final result is presented to the user[*]. There are at least two cases, however, in which it is useful to expressly seek out *colliding* paths:

1) *Forced passage* – Take the example of a seat that must be mounted though the metal skeleton of a car. The path planner may be unable to find a collision-free path, although a human operator could quickly recognize that the cushioned seat could be "pushed" through the opening.
2) *Identifying design errors* – If no solution exists for a path planning case, it remains unclear how to analyze the problem. Should the part itself be moved or redesigned, or is there another part of the assembly that should be examined? In the labyrinthine models that PLM designers often work with, the answer may be all but obvious.

Informally, we would like a motion planner that finds collision-free paths if they exist, and if not, finds paths that collide "the least possible." This problem is neither well-posed nor well-addressed in the literature. In order to give it a more concrete form, we need a method for measuring the "amount of collision" for a path.

5.1 Related Work

As far as we know, design error analysis has not been studied to a great degree by the scientific community. The closest related fields are probably assembly sequencing and geometric tolerancing, which are briefly introduced here.

[*] It is not always possible to guarantee complete lack of collisions. In many cases, it suffices to limit penetration to a very small amount.

Assembly sequencing addresses the problem of constructing a product out of its parts. Typically, a sequence specifies a series of assembly tasks for the parts, and possibly indicates which tasks can be carried out in parallel. This problem was most actively studied in the 1980s and early 1990s [Cao and Sanderson 1992, Delchambre 1990, Latombe and Wilson 1995, Milner, et al. 1994, Seow and Devanathan 1994, Wilson and Latombe 1995], although recent work also exists which uses probabilistic path planning in order to find sequences [Sundaram, et al. 2001].

Geometric tolerancing attempts to account for inevitable production errors by assigning acceptable margins of error to features of mechanical parts in the design stage. Tolerances can be assigned for feature placement, size, as well as shape [Inui, et al. 1995, Jayaraman and Srinivasan 1989, Joskowicz, et al. 1997, Srinivasan and Jayaraman 1989, Yap and Chang 1997].

However, the problem we are trying to address deals more with correcting design errors than production ones. We draw on path planning strategies where obstacles are allowed to move within the scene [Dacre-Wright, et al. 1992, Wilfong 1988]. In particular, the ML-RTT described in Chapter 2.3.2 is aimed at molecular disassembly problems [Cortés, et al. 2007]. In the molecular model, parts of the protein, called side-chains, can be pivoted around an axis by the moving ligand. The ML-RTT decouples the movement of the ligand and the side-chains. At each iteration, the planner first attempts to move the ligand. If the ligand motion is blocked by one or more side chains, then these side-chain positions are resampled and the planner tries to reconnect the new position to the current one. Paths resulting from the ML-RTT appear to show the side-chain “pushing open” the side-chains blocking its path like barn doors.

5.2 Implicitly Controlling Penetration Distance

One way to analyze collisions is to quantify the penetration between overlapping objects. The most common measure is called Penetration Distance (PD) and relates the minimum translation distance needed to move one object out of collision with another [Cameron and Culley 1986]. In other words, it is the length of the smallest vector (in translation space) required to shift an object out of collision (**Figure 5.1**). Recently, PD has been generalized to include rotations [Zhang, et al. 2007], but we stick with the classic definition in this work.

To express PD mathematically, we will introduce an operator that translates an object A by a vector $\boldsymbol{t}$.

$$A + \boldsymbol{t} = \{a + \boldsymbol{t} : a \in A\} \tag{5.1}$$

Then the PD between two objects A and B can be defined as

$$PD(A, B) = \min_{\boldsymbol{t}}\{\|t\| : (A + \boldsymbol{t}) \cup B = \emptyset\} \tag{5.2}$$

PD has several nice properties that make it appropriate for a generalized motion planner: it applies to any type of geometry, requires no parameterization, and is commutative (i.e. $PD(A, B) = PD(B, A)$). A reasonable approach for motion planning could be to simply measure the PD at each collision, and allow configurations that have a PD inferior to a certain value.

However, even with the fastest methods available to us today [Fisher and Lin 2001, Kim, et al. 2002, Zhang, et al. 2007], explicitly calculating PD would overly burden the collision detection routines, which already take over 80% of computation time in typical examples. From

our practical experience with PLM cases, collision detection times must be on the order of 1 ms per test, and PD calculations are currently far from able to achieve this level of performance.

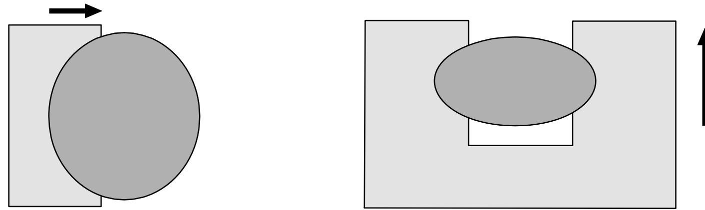

Figure 5.1: Penetration Distance. The PD is equal to the translation distance needed to separate two objects. For the case on the left, the arrow indicates a vector that the ellipse could follow to get out of collision. Since PD is a symmetric measurement, it would be equivalent to move the box to the left by the same amount. The case on the right demonstrates how PD does not always correspond to an intuitive notion of measuring collision. Rather than trying to shrink the sides of the ellipse or turn it, the shortest vector out of collision is to move it vertically out of the other object.

We propose a different approach. Instead of trying to measure the PD directly, why not let one of the two objects move out of collision? If such a movement is possible, and the distance of the movement can be calculated, then we can deduce an upper bound on the PD of the collision.

$$[(A+\boldsymbol{t}) \cup B = \emptyset \ \wedge \ \|\boldsymbol{t}\| \leq p] \Longrightarrow [PD(A,B) \leq p] \tag{5.3}$$

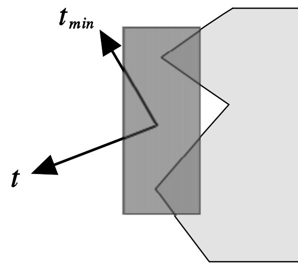

Figure 5.2: Movement equivalent to upper bound of PD. Instead of calculating the PD exactly, we only know that the movement $\boldsymbol{t}$ brings it out of collision. This implies that that $\boldsymbol{PD} \leq \|\boldsymbol{t}\|$, even if the minimum vector $\boldsymbol{t_{min}}$, remains unknown.

With this in mind, it is easy to see that given a colliding position and a distance p, if we are able to find a non-colliding position so that the distance between the two is less than p, the penetration distance must be less than or equal to p. Although the exact penetration distance is still unknown, it is at least bounded (**Figure 5.2**).

The central idea of our approach is to fix *p* from the outset, and search for non-colliding positions that respect this distance in order to control the penetration. We call this “allowed penetration” the *play*, in reference to loose mechanical parts.

To use our algorithm, the user specifies certain parts as *active obstacles* and assigns each a play value. During the path planning process, these parts are allowed to move within the boundaries specified by their play. Such movement can open up passages that may have been inadvertently closed during the design process, or allow an object to force its way through a narrow opening. By analyzing the extent and direction of movement needed to find a solution, the user can gain valuable knowledge about the design issues as well as possible ways to address them.

Our algorithmic problem statement is the following: given a starting and ending configuration for a robot, and a list of active as well as passive obstacles, to find a path that brings the robot from one configuration to the other without collision with any obstacles, but letting the passive obstacles move up to their given limits (**Figure 5.3**).

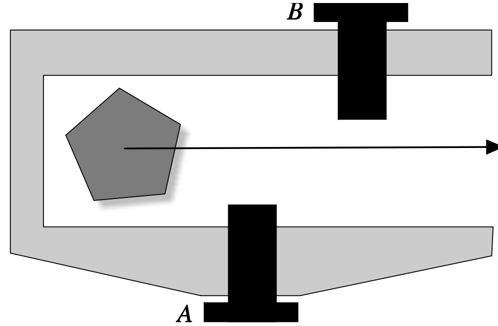

Figure 5.3: Example disassembly problem with active obstacles. Extracting the pentagon from the assembly involves passing by two screws, *A* and *B*. There is no solution to this problem if the obstacles are not allowed to move. In order to apply our algorithm, we assign starting and ending positions to the pentagon, and specify that the two screws are active obstacles. The rest of the scene is considered as passive obstacles. Through our algorithm, we can determine that it is necessary to move screw *B* in order to extract the pentagon, but not screw *A*. Additionally, we can estimate the necessary movement of screw *B*.

In this chapter, we put forward two algorithms capable of solving this problem. The first, called the Teleportation-Based Planner, integrates the obstacles directly in the configuration space of the robot. The second, called the *superposition operator*, works on the collision detection layer, and can be integrated into any path planner.

5.3 Teleportation-Based Planner†

Our first approach is based around encoding the positions of the obstacles directly into C alongside the robot. Since we are only interested in translational PD, the position of each active obstacle is dictated by three translation DOFs that, taken together, represent an offset from the initial position of the obstacle.

The magnitude of this offset is limited to be less than or equal to its play. That is, if the three DOF values for a configuration q are labeled q_x, q_y, and q_z:

$$\forall q \left(valid(q) \Rightarrow \sqrt{q_x^2 + q_y^2 + q_z^2} \leq p \right) \quad (5.4)$$

In other words, the movement of the center of an active obstacle is constrained to translation within a sphere of radius p around its initial position (**Figure 5.4**).

Once DOFs are added for each active obstacle, a RRT will be able to move the obstacles just by assigning values to these DOFs. The only modification necessary in order for this

† An original contribution of this work in the context of probabilistic path planning is the use of a discontinuous interpolation between roadmap nodes, which explains the reference to "teleportation" in the title.

approach to work is to add the distance constraint specified in (5.4). This can be done in the *Shoot* stage of the RRT (see Section 2.3 for details).

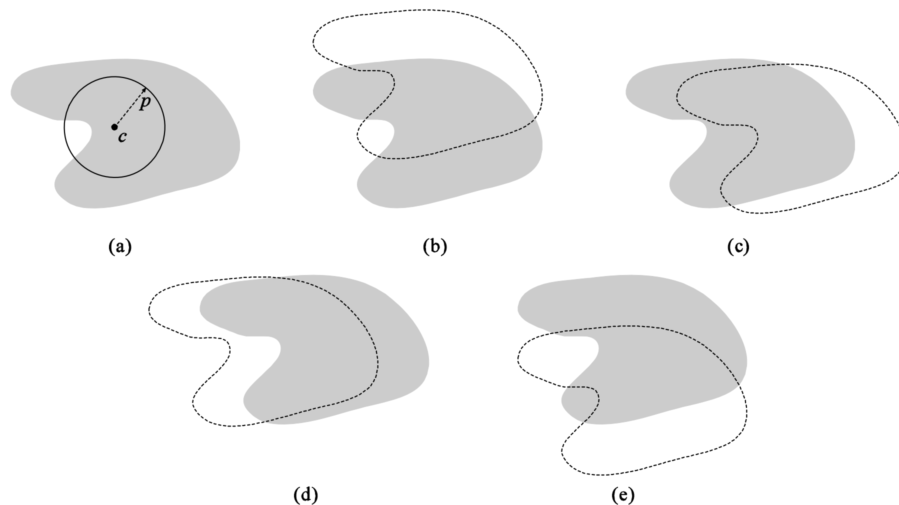

Figure 5.4: Active obstacle movement. The movement of the center of an active obstacle (the grey object) is limited to a sphere of size $\boldsymbol{p}$ around its initial position, $\boldsymbol{c}$. The object and the sphere are shown in (a), and (b) through (e) outline several possible positions that the obstacle might move to.

As shown in [Kuffner and LaValle 2000], if a solution to a path planning problem exists, the probability that the RRT will find it in a finite amount of time equals 1. Furthermore, the distribution of RRT vertices converges to the sampling distribution as the algorithm progresses. Since the non-colliding active obstacle positions—and therefore the minimum penetration vectors—are chosen through the same process, this planner is probabilistically complete.

Theoretically, the changes just discussed are enough for an RRT to solve path planning problems with active obstacles. Its performance, however, leaves much to be desired. By analyzing its behavior, we have developed several enhancements to overcome difficulties associated with our approach.

Our method essentially involves adding all active obstacles to the robot. With three DOFs per obstacle, this can quickly multiply the dimensionality of the configuration space, invoking the "curse of dimensionality" that haunts high-dimension problems. Thus, adding these DOFs simultaneously allows for a solution to be found, and makes that solution take longer to find.

In general, we have found that the best way to combat the curse of dimensionality brought on by the active obstacle DOFs is to treat them in a different and more efficient manner, better adapted to their particularities.

5.3.1 Uniform Shoot

The first step of the RRT iteration, *Shoot*, involves choosing a configuration randomly within the configuration space. Since our implementation is based upon IPP, an enlarged local neighborhood around the existing roadmap is used to approximate the known configuration space for non-limited DOFs (see Chapter 2.3.1).

This approach is valid for limited DOFs as well, as long as the enlarged local neighborhood is allowed to grow quickly enough. In such a case, the neighborhood has soon expanded to

include both limits, assuming that no constraints prevented it from finding valid configurations near those points.

For active obstacle DOFs, however, there is no advantage in restricting the configuration choice to a local neighborhood around previous nodes. Additionally, since they cannot collide with the rest of the scene, there is no reason to prevent them from exploring all possible positions. In fact, a typical situation involves pushing an active obstacle DOF to its extreme value, a position more easily attainable through uniform sampling then local.

5.3.2 Distance Metric

The second technique that we propose is to modify the distance metric used by the *Pick* step of the RRT. The distance metric plays an important role, since the choice of which node will be extended towards the shot configuration can seriously impact performance. Any interpolation between a picked node and a shot configuration can lead to a collision, and by picking nodes that are "closer" to the shot configurations, this possibility can be minimized.

Although many distance metrics can be used, a common choice for robotic systems is a scaled DOF difference. With this metric, each DOF d is associated with a weight w. The distance between two configurations q and q' is thus:

$$\sqrt{\sum_{d}\left(w_d(q'_d - q_d)\right)^2} \tag{5.5}$$

One advantage of this technique is that DOFs can be given higher or lower weight in accordance with their "importance," whether geometrical, mechanical, or otherwise defined by the application.

By default, active obstacle DOFs have the same weight as other translation DOFs in the robot. But when the number of active obstacle DOFs becomes larger than that of the robot, this leads to the planner taking more account of the obstacles than of the object we are trying to plan for. In such a case, the planner will pick nodes for which the robot is arbitrarily far away from the shot configuration, only because the obstacles are closer (**Figure 5.5**).

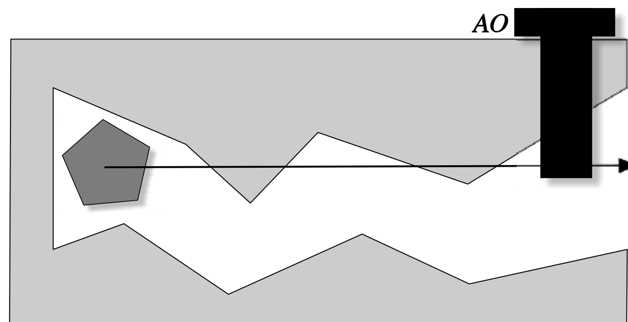

Figure 5.5: Taking active obstacles into account. In this example, we are trying to extract the pentagon. It must follow a torturous narrow passage in order to escape, in which the active obstacle (the black screw) plays no role. When picking nodes to extend from, it would be inefficient to choose based on the position of the active obstacle in this situation.

The previous scenario suggests that active obstacle DOF weights should be reduced as much as possible. Nevertheless, setting them to zero, while an improvement over equal

weighting, is actually counterproductive (**Figure 5.6**). Since active obstacle positions are shot randomly, it is worthwhile to save those that lead to roadmap growth. In typical examples, the obstacles are "pushed" out of the way by the robot. Once out of the way, they can be left there to allow further maneuvering.

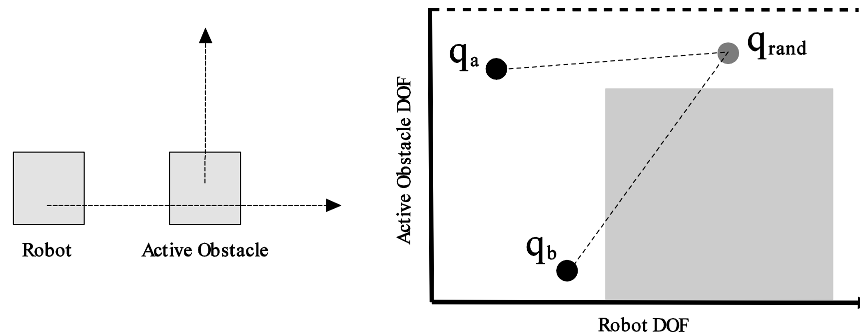

Figure 5.6: Ignoring active obstacle distances. In this simple example, the active obstacle must move up for the path target to escape. In configuration space (right) this corresponds to a narrow passage in C_{free}. Given a shot configuration q_{rand}, the planner may have to choose between two configurations from which to extend: q_a and q_b. If only the path target DOF is taken into account, the latter is closer, although the former is a better choice for expansion. In general, active obstacle DOFs should be taken into account in order to exploit such passages.

Therefore a balance must be struck. In practice, we have found that very low but positive weights work best, i.e. several orders of magnitude less than the normal robot DOF weights.

5.3.3 Teleportation

The third enhancement that we can bring to the RRT algorithm relates to the interpolation between successive configurations, also called the direct paths. As the output of an RRT is only a sequence of configurations, the interpolation provides all information about object positions between configurations.

The interpolation used for robot movement depends on the context, but it is safe to assume almost all are continuous, in that there are no sudden "jumps" from one position to another. By default, the active obstacle movement inherits this behavior, smoothly going from one configuration to the next. For example, a simple linear interpolation from q to q' with the variable $t \in [0,1]$ can be expressed as follows:

$$h_{linear}(q, q', t) = q + t(q' - q) \tag{5.6}$$

Since we are not attempting to simulate physical movement of the active obstacles, continuous interpolation leads to overly conservative behavior. As previously described, only a single active obstacle position is necessary to validate the PD constraint. Therefore, interpolating the active obstacle positions, although giving valid results, provides no more information than simply choosing a single position (e.g. the position being interpolated to). As **Figure 5.7** illustrates, interpolating the active obstacle DOFs linearly often creates more possibilities for collision rather than reducing them.

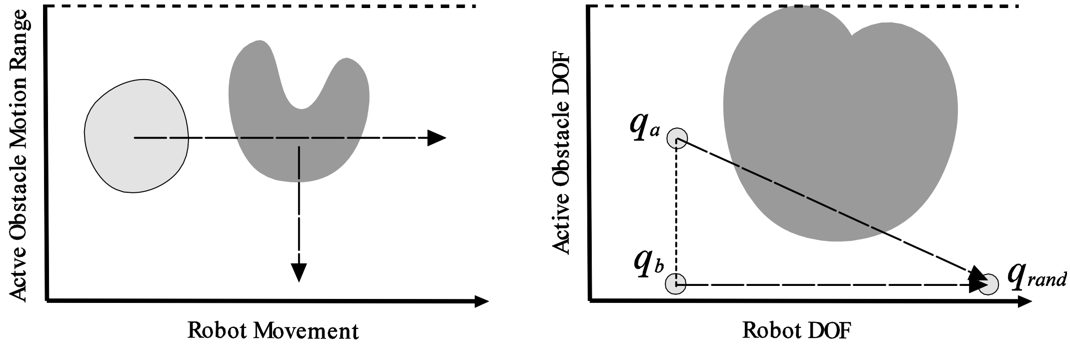

Figure 5.7: Discontinuous interpolations give better performance. Translating the light-grey object to the right along a single axis leads to a collision with the darker-grey obstacle, even if the obstacle is itself linearly interpolated towards the bottom (left). The same situation can be illustrated in configuration space (r) in which a linear interpolation from q_a to q_{rand} leads to collision with the obstacle. By adapting a hybrid interpolation in which the active obstacle instantaneously jumps to q_b while the robot is interpolated smoothly, this collision can be avoided.

Taking this analysis into account, we've adapted a hybrid interpolation, in which the robot DOFs are interpolated normally (e.g. with h_{linear}) but the active obstacle DOFs are interpolated with the following function:

$$h_{jump}(q, q', t) = \begin{cases} q, & t = 0 \\ q', & t > 1 \end{cases} \quad (5.7)$$

Through its discontinuity, the h_{jump} interpolation method leads to much faster exploration, as well as introducing certain maneuvers that could not arise in a continuous method, such as the active obstacle "jumping over" the robot. Such behavior is counterintuitive if obstacle movement is considered as physical motion, but is allowed by our PD constraint.

5.3.4 Manhattan-Like RRT

The ML-RRT uses a Manhattan-like method to connect a node to a shot configuration in the *Extend* step of an RRT iteration [Cortés, et al. 2007]. The step is broken up into two parts: first the robot motion is dealt with, followed by the obstacle motion. With a few adjustments, we've adopted the ML-RRT to the active obstacle context.

In the first part, the motion of the robot is tested progressively along the interpolation between the node and the shot configuration. If no collision was detected, then a path segment from the node and the shot is added to the roadmap. If a collision was found, then the added segment only reaches until the last non-colliding configuration tested.

If no collision occurred in the first part, then the second part is skipped. Otherwise, if the collision occurred only with active obstacles, then the planner attempts to add a path segment connecting the previously added node to a new configuration in which only those active obstacles are in movement (**Figure 5.8**).

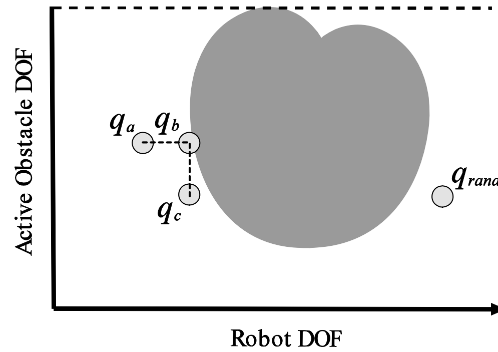

Figure 5.8: Classic ML-RRT. The extension of q_a to q_{rand} is broken up into two components. First, the robot DOF is interpolated until a collision is detected, and a node posed at that point, q_b. Next, the movement of all colliding active obstacles is interpolated, leading to the segment from q_b to q_c.

The algorithm can be succinctly described using a function m that combines two configurations into one, based on the membership of a DOF. If a DOF belongs to an object provided in the set *L*, then its value is that of the second configuration. Otherwise, it takes on the value from the first configuration.

$$m(q, q', L) = (d_0, \dots, d_n), \quad d_i = \begin{cases} q'_i, & d \in L \\ q_i, & d \notin L \end{cases} \tag{5.8}$$

Let the function *expand(from, to)* progressively test an interpolation for collision, and return the last non-colliding configuration found as well and the set of colliding obstacles (L_C). Let the set SO contain all the active obstacles, and PT the robot. Now the Manhattan extend step can be written as follows:

```
1.  procedure extend(node, shot)
2.    cm ← m(node, shot, PT)
3.    (cpt, LC) ← expand(node, cm)
4.    if cpt ≠ node then
5.      addEdge(node, cpt)
6.      node ← cpt
7.    end if
8.    if LC ≠ ∅ and LC ⊆ SO then
9.      cm ← m(node, shot, LC)
10.     (cso, LC) ← expand(node, cm)
11.     if cso ≠ node then
12.       addEdge(node, cso)
13.     end if
14.   end if
```

Algorithm 5.1: ML-RRT for active obstacles.

By decoupling the movement of the robot from the active obstacles, the active obstacle positions are not modified as long as they do not collide with the robot. When they do, they tend to move only to allow the robot to pass, giving the impression that the robot is physically pushing them. Once an active obstacle is moved out of the way, it tends to stay in that position, which is advantageous to the planner.

Since the ML-RRT extension step breaks up the interpolation into two parts, it reduces the effect of a discontinuous interpolation. As the active obstacle DOFs are not touched by the first segment, the "jump" has no effect on it. The second segment will have the discontinuity, but will not use it to further explore the space.

We can improve efficiency by having the two work in concert. It is necessary to modify the second edge that the ML-RRT attempts to add so that the robot DOFs vary along with those of the colliding active obstacles. The only adjustment to the previous algorithm is on line 9.

```
9.      cm ← m(node, shot, LC ∪ PT)
```

Algorithm 5.2: Modified ML-RRT for discontinuous interpolations.

The combination of the ML-RRT and the discontinuous interpolation then retains the interests of both, as only the active obstacles in collision are moved, but the space is explored rapidly without enforcing a "physically realistic" maneuver (**Figure 5.9**).

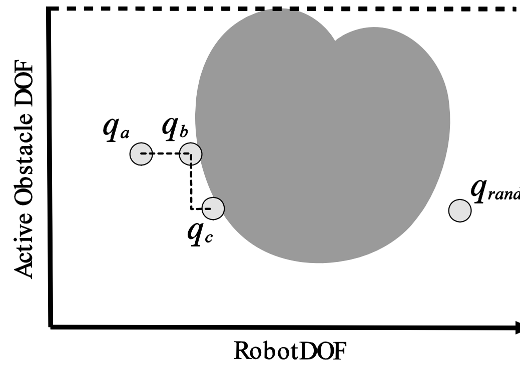

Figure 5.9: ML-RRT with discontinuous interpolation. By including the robot in the objects interpolated for the second segment that the ML-RRT attempts to add, the configuration q_c stays much tighter to the obstacle. In this example, with only one active obstacle, the second segment tested by the planner is from q_b to q_{rand}, leading to the node q_c where it collides with the obstacle.

5.4 Superposition Collision Operator

Instead of integrating the obstacles into the configuration space of the robot, an alternative approach is to handle the active obstacles strictly on the level of collision detection. In this view, it is up to the collision detector to validate positions where the robot collides with active obstacles. It can do so by searching for a non-colliding position for the obstacle that is less than the allowed PD away from its origin.

The intuition behind the soft collision operator is simply to check a certain number of "likely" non-colliding positions during the collision test. If any of the tested positions do not collide, then the configuration can be considered valid. For succinctness, we will call these tested positions *superpositions*, a pun on quantum superposition, in which particles can exist in multiple states simultaneously (**Figure 5.10**).

Let the expression f_{rigid} be true if that an object A translated by $\boldsymbol{t}$ is free of collision with B.

$$f_{solid}(A, B, \boldsymbol{t}) \equiv (A + \boldsymbol{t}) \cap B = \emptyset \quad (5.9)$$

The expression f_{pd} determines if two objects can be separated by a translation of less than the distance p.

$$f_{pd}(A, B, p) \equiv \exists \boldsymbol{t} \left(\|\boldsymbol{t}\| \leq p \wedge f_{rigid}(A, B, \boldsymbol{t}) \right) \quad (5.10)$$

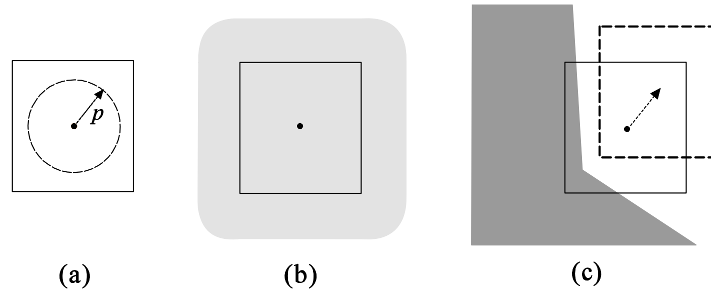

Figure 5.10: Collision test with superposition operator. In (a), the square has an allowed penetration of p. Its center can therefore "move" within a circle of radius p. If we consider all the possible positions that the square can move to, we obtain the grey cloud shown in (b). In (c), the square collides with the dark-grey obstacle. By translating the square to the upper right, we can find the dashed position, which does not collide with the obstacle. For the purposes of the superposition operator, this position is therefore non-colliding.

The goal of the superposition operator is to approximate f_{pd}. It does so by testing a set of translation vectors that are representative of the space. Given that traditional collision tests (and proximity tests in general) are deterministic, we decided to forgo a random sampling of vectors. In its place, we choose positions on the surface of a scaled sphere of radius p. The translations are applied in the current rotational frame of the robot, and thus its rotations are implicitly taken into account.

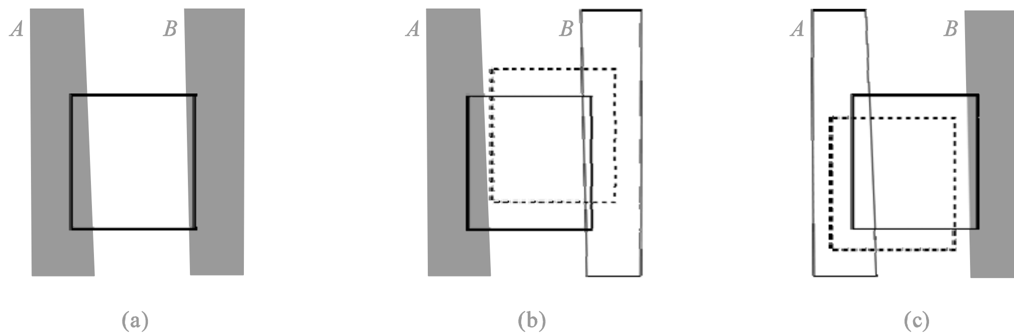

Figure 5.11: Handling multiple obstacles. In (a), the square is moving through a narrow tunnel created by two active obstacles, A and B. The superposition operator considers collisions with each obstacle separately. In (b), only the obstacle A is taken into account, and so the dashed position is found that avoids collision. In (c), the same calculation is performed for B. The superposition operator therefore concludes that the square is not in collision with the obstacles, even though there is no *single* position that avoids contact with all obstacles. This explains the choice of name *superposition* operator.

When multiple obstacles are involved, only a single non-colliding active obstacle position is required for each, and not necessarily the same one (**Figure 5.11**). The following expression tests A against a set B' composed of pairs *(obstacle, penetration)*.

$$f'_{pd}(A, B') \equiv \forall B \forall p \left((B, p) \in B' \Rightarrow f_{pd}(A, B, p) \right) \tag{5.11}$$

Algorithmically, the test *isFree* can be described as follows, given the robot $\mathcal{A}$, a set of rigid obstacles R, and the pairs of active obstacles B' (**Algorithm 5.3**).

```
1.  function isFree(A, R, B'): Boolean
2.    for each r in R
3.      if f_solid(A, r, 0) return False
4.    end for each
5.    for each (obs, pd) in B'
6.      if not f_so(A, obs, pd) return False
7.    end for each
8.    return True
9.  end function
10.
11. function f_so(A, obs, pd): Boolean
12.   if f_solid (A, obs, 0) return True
13.   for each t in SampleVectors
14.     if f_solid (A, obs, t * pd) then
15.       return True
16.     end if
17.   end for each
18.   return False
19. end function
```

Algorithm 5.3: Boolean soft collision detection routine.

The test relies on the constant set $SampleVectors$, which must be initialized with translational vectors (of unit length, if only the surface of the sphere is desired). The size of the set is itself a performance factor. Larger (denser) sets increase the probability of a given test passing, but also increase the time per test before concluding failure.

5.4.1 Comparison with the Teleportation-Based Planner

The Teleportation-based planner and the superposition collision operator can be used to solve the same kind of forced-passage problems and design-error studies. However, they have different theoretical and practical characteristics. In this section, we will contrast and compare them.

The teleportation-based planner is a fundamentally probabilistic approach to finding active obstacle positions. The positions are generated by the same RRT path planning procedure (*Shoot*, *Pick*, *Extent*, etc.) that handles the robot. Conversely, the superposition operator is deterministic, which means that given the same situation, it will always return the same result. Unfortunately, this repeatability comes at the cost of its algorithmic completeness.

Simply put, since the superposition operator only tests a small sample from the infinite number of possible offsets that fulfill the PD constraint, it cannot guarantee the correct result. Since it is deterministic, it will always test the same samples, in contrast to the “random” sampling performed by the Teleportation planner. One such example which can occur is when obstacles overlap in space, and the robot must pass through an active obstacle at a lesser penetration (**Figure 5.12**).

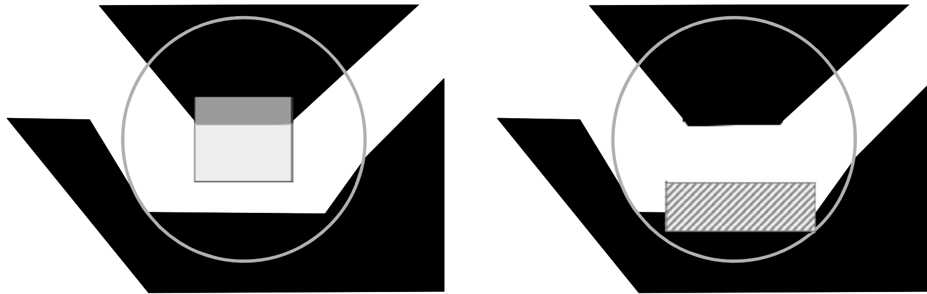

Figure 5.12: Incompleteness of the superposition operator. On the left, the transparent box collides with an active obstacle (both black regions are part of the active obstacle). The active obstacle has a play equal to the radius of the circle. Although a solution exists, whether or not the superposition operator finds it depends on which offset vectors are considered. This situation does not lead to probabalistic incompleteness, however, as the path planner could always test the configuration where the box is within the corridor, active obstacle or not. In the case on the right, an additional *rigid* obstacle prevents such a movement, rendering the operator incapable of finding a valid solution.

That said, the superposition operator works very well in practice, where the user generally allows only “small” penetrations. This concept is characterized by both the robot and the active obstacles having widths that are much larger than the play, even at their thinnest portions[‡]. In such cases, the superposition operator will find solutions where the robot only grazes the surface of an obstacle, without moving to the other side of it, entering inside it, nor moving into a hole large enough for it.

For small penetrations, the superposition operator acts as a heuristic, testing offsets that are most likely to escape collision. Longer offsets are more likely to do so. By testing offsets of length *play*, not only is the superposition operator faster than the teleportation planner in most cases, but it scales much better to a large number of active obstacles.

An additional advantage of the superposition operator over the Teleportation planner is its ability to consider multiple positions for each active obstacle *along the same direct path*. To validate a direct path before adding it to the roadmap, the collision checker usually tests multiple configurations along the path. Since each collision detection involving an active obstacle will invoke the superposition operator, this process effectively allows for different active obstacle positions to be considered at each configuration. Such a process is beyond the capabilities of the Teleportation planner, where the active obstacle offsets are encoded directly in the configurations.

5.5 Distance Measurement

Up until now, we have been considering only Boolean collision tests for the active obstacles. There are cases, however, in which a numerical measure of distance provides valuable information. In this section, we present approximations of two distance measurements for active obstacles: separation distance and penetration distance.

[‡] It is challenging to discuss the size of 3D objects in a general way. Informally, we can imagine that the objects represent volumes that can be roughly separated into portions with different “widths.” The smallest such width can be considered the “minimum width” of the object.

5.5.1 Separation Distance

Some path planners, such as IPP, progressively test a path segment for collision by estimating the distance between the robot and the obstacles around it [Ferré and Laumond 2004, Saha and Isto 2006]. The distance metric needed for this operation is a lower bound on separation distance. But when objects collide, even if one is an active obstacle, this distance becomes null, and path planners must resort to an arbitrary epsilon value, reducing performance in the process.

To work around this problem, we can define a separation distance for active obstacles. Following intuition, if two objects avoid collision because one is an active obstacle, then the separation distance between them is equal to the amount that the play of the active obstacle can be reduced without introducing a collision (**Figure 5.13**).

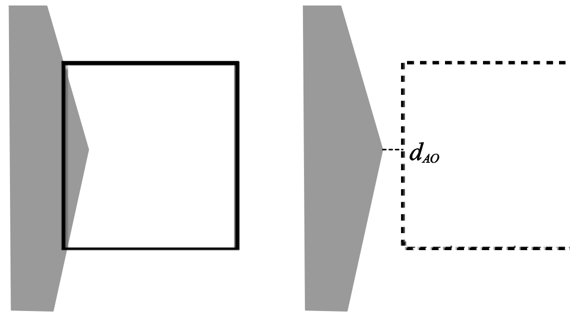

Figure 5.13: Generalized separation distance. The square collides with the active obstacle (left), but an offset position (dashed square) has been found that does not collide (right). The distance d_{AO} separates the active obstacle position from the obstacle, and so the play can be reduced by at least this value before a new collision would be created.

We define the function d_{solid} that measures the distance separating two rigid objects [Cameron and Culley 1986].

$$d_{solid}(A, B) = \min_{\boldsymbol{t} \in \mathbb{R}^3} \{\|\boldsymbol{t}\| : (A + \boldsymbol{t}) \cap B \neq \emptyset\} \tag{5.12}$$

The teleportation planner will only test a single offset for each active obstacle, but the superposition operator will test the entire set $SampleVectors$. In either case, we would like to know the maximum separation distance between the tested positions and the rigid object. Given that the set T contains either the single offset or the contents of $SampleVectors$, we can define the separation distance for active obstacles.

$$d_{AO}(A, B, p) = \max_{\boldsymbol{t} \in T} d_{solid}(A + \boldsymbol{t}, B) \tag{5.13}$$

Therefore, p can be reduced by d_{AO} without causing a collision. For the purposes of path planning, we can define a function that returns the correct separation distance depending on whether the objects collide or not.

$$d_{sep}(A, B, p) = \begin{cases} d_{solid}(A, B), & A \cap B = \emptyset \\ d_{AO}(A, B, p), & otherwise \end{cases} \tag{5.14}$$

5.5.2 Penetration Distance

Another case for a distance metric arises when the path planning is complete. Although we are sure that the output path respects the penetration constraints, we do not have a grip on the

actual penetration values. For evaluation purposes, it is useful to reduce the upper PD bound as much as possible while neither recalculating a path nor explicitly calculating the PD.

In this context, an upper PD bound can be arrived at indirectly, by reducing the allowed PD by the maximum distance found between the tested soft object positions and the rigid object. As the allowed PD could be reduced by d_{AO} without causing a collision, we can simply adjust the allowed PD by this amount.

$$pd(A, B, p) \ \leq \ p - d_{AO}(A, B, p) \tag{5.15}$$

This measurement only applies to a given position and active obstacle. When multiple active obstacles are present, the PD of a rigid object becomes the maximum PD calculated.

$$pd'(A, B') \leq \max_{(B,p) \in B'} pd(A, B, p) \tag{5.16}$$

Finding the maximum penetration distance along a path can be most easily done by testing a series of sampled configurations. However, such a method does not guarantee that a larger, undetected, penetration is not hiding between the samples. Intuitively, more samples are needed when pd approaches p (d_{AO} tends toward zero), and less when pd is large (d_{AO} tends toward p).

Inspired by the dynamic collision checker developed in IPP, we can use the value returned by the pd function for one configuration to decide where along the path to choose the next. We assume that the function $b(\lambda, c, \Delta)$ provides a subsequent configuration to q along a path λ so that no point on the robot λ_t) has moved more than a distance Δ.

$$[b(\lambda, \mathrm{q}, \Delta) = q'] \Rightarrow \left[\max_{a \in \lambda_t} \| q' \times a - \mathrm{q} \times a \| \leq \Delta\right] \tag{5.17}$$

Here the cross operator $\times$ transforms a point by a configuration. We can now establish a function that returns a subsequent configuration respecting the bounds provided by b. For completeness, we include a minimum advancement variable ε so that the collision test advances even when the PD approaches zero.

$$q_{next}(\lambda, \mathrm{q}, \varepsilon) = b(\lambda, q, \max\{pd'(\lambda_t, B', c^0), \varepsilon\}) \tag{5.18}$$

Finally, we can establish a simple iterative algorithm that finds the maximum PD estimate along a path.

```
1.  function maxPD(λ, B', ε): Number
2.     q ← start(λ)
3.     dmax ← 0
4.     repeat
5.       d ← pd'(λt, B', q)
6.       dmax ← max(dmax, d)
7.       if q = end(λ) return dmax
8.       q ← b(λ, q, max(d, ε))
9.     end repeat
10. end function
```

Algorithm 5.4: Routine for finding the maximum PD estimate for a path.

5.6 Dynamic Play

The previous sections describe how to solve a path planning problem with static play values. Letting the play vary, however, opens up two new possibilities: searching for the smallest acceptable play, and using high play values to suggest draft paths that can be refined when the play is lowered.

Both these ideas are based on the assumption that higher play values make a path planning problem easier to solve. Although this is not strictly true in every case (see Section 5.4.1), it has shown to be correct in the large majority of cases. A problem with higher play admits a larger number variety in active obstacle positions, opening up new solutions. It can be compared to the strategy called free space dilation, which is known for accelerating the path planning process for rigid environments [Hsu, et al. 1998].

5.6.1 Draft Paths

Just as IPP uses "draft paths" in order to discover in-collision paths that are refined in successive iterations, we would like to allow large amounts of penetration with active obstacles and then iteratively refine the solutions by gradually reducing the play until a solution is found at the desired value [Ferré and Laumond 2004]. The succession of play values used forms a sequence that we call dp.

The initial play value should be very high and yet still provide some information useful for subsequent draft paths (i.e. an allowed PD of infinity would quickly find a solution, but be useless as a basis for the following iterations). The ideal value is difficult to find *à priori*, and so we chose to simply raise the user-specified play several orders of magnitude. This method creates sufficiently large values and is model independent, at the expense of over-estimating the "ideal" value.

Once the initial problem has been solved, we would like to evaluate the play required by the solution path, which is likely to be much lower than the initial value. This task can be easily accomplished with the *maxPD* algorithm described in Section 5.5.2, and fixes an upper bound p_{high} on *dp*. The lower bound p_{low} is the play assigned by the user.

Logically, the problem becomes more difficult to solve as p approaches p_{low}. It makes sense then to gradually approach p_{low} by smaller and smaller steps. One approach is to decide on the number of steps n in advance, and scale the intermediate values appropriately.

$$dp(n,s) = \left\{p_0, \dots, p_{n-2}, p_{low} \,:\, p_i = p_{low} + \frac{p_{high} - p_{low}}{s^i}\right\} \tag{5.19}$$

The scaling factor s in this equation determines the rate of convergence towards the lower bound (**Figure 5.14**).

5.6.2 Play Optimization

Given a scenario with a soft obstacle, what is the minimum allowed PD for which we are able to find a solution? Although we cannot solve this problem for an exact solution, we can use an iterative process to find increasingly better estimates (i.e. tighter bounds).

In contrast with the draft path case, the "target" play that we are approaching is also the unknown that we are solving for, and the so the sequence of dynamic play values cannot be

predetermined. Complicating the matter further, we only have probabilistic guarantees of finding a solution if one exists, and no practical way to establish the absence of one. Simple methods such as timeouts are problematic due to the high variance in running times [Isto, et al. 20003].

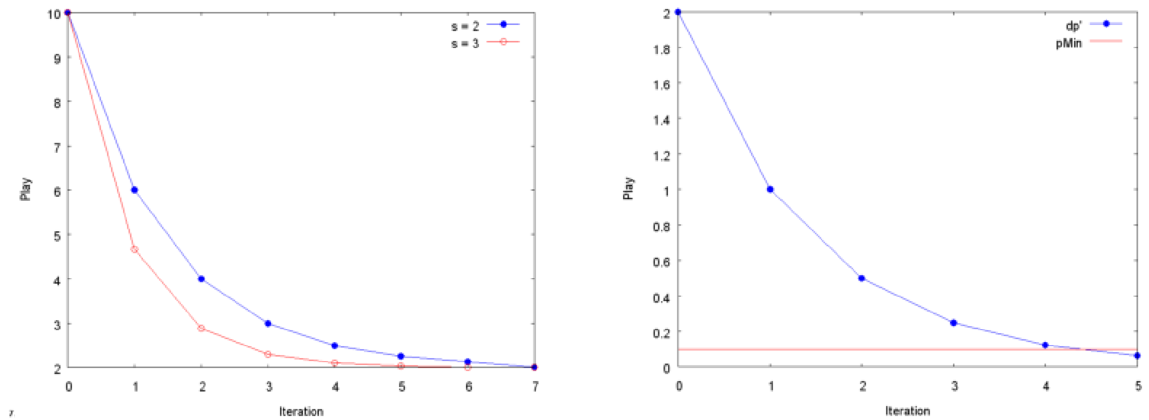

Figure 5.14: Dynamic play. On the left, the play ranges from 10mm to 2mm over 8 steps. The graph compares the effects of two different values of the scaling factor s. The larger the scale, the faster the play decays. On the right, we look for the minimum value for which we can still find a solution. Starting at a play value of 2mm, and with a scale factor s of 2, we quickly approach the unknown minimum play p_{min} of 0.1mm. The first 5 iterations can complete, but the 6th will never succeed, since it is solving for a value below the minimum. The conclusion of our algorithm is that p_{min} lies somewhere between those two play values.

For this reason, although it would be ideal to perform some kind of binary search for the minimum play value, we stick with a strictly decreasing sequence of values, and let the user determine when to stop the optimization process. To reduce the chance that we overshoot the minimum play p_{min}, we can adopt a geometric progression with a low scaling factor (**Figure 5.14**).

$$dp'(s) = \left\{\frac{p_{low}}{s^i} : i \geq 0\right\} \tag{5.20}$$

This sequence functions well if the unknown p_{min} is non-zero. But in the exceptional case where no penetration is needed, the infinite sequence dp' will never reach zero. To work around this problem, we can call the *maxPD* function on the intermediate paths in order to detect if the play can be reduced to zero.

5.7 Experimental Results

We have tested our approach on four real industrial assembly cases. The first involves a car windshield wiper, for which there is no solution for extracting a wiper from its holder. Through specifying a suspect part as an active obstacle and running our algorithm, a disassembly path is found (**Figure 5.15**). The second case considers an automobile exhaust system. The design flaw in this case is the placement of a bracket that closes the already narrow passage (**Figure 5.16**). In both these cases, once the offending part has been identified, we can move on to characterizing the design error and suggest ways to correct it.

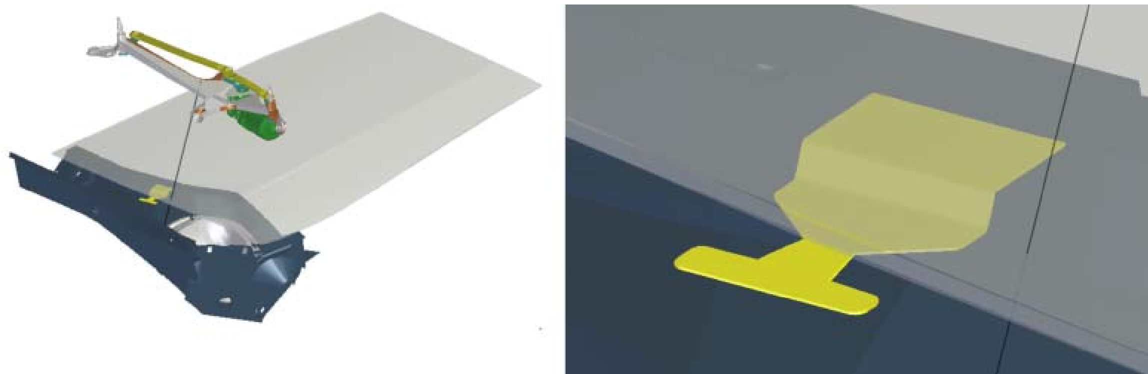

Figure 5.15: Windshield wiper test case. The goal is to assemble the wiper into its holder (left). The close-up (right) reveals the yellow plate that prevents this task from being carried out. This design error can be fixed with the help of our algorithm, once the yellow part is made into an active obstacle.

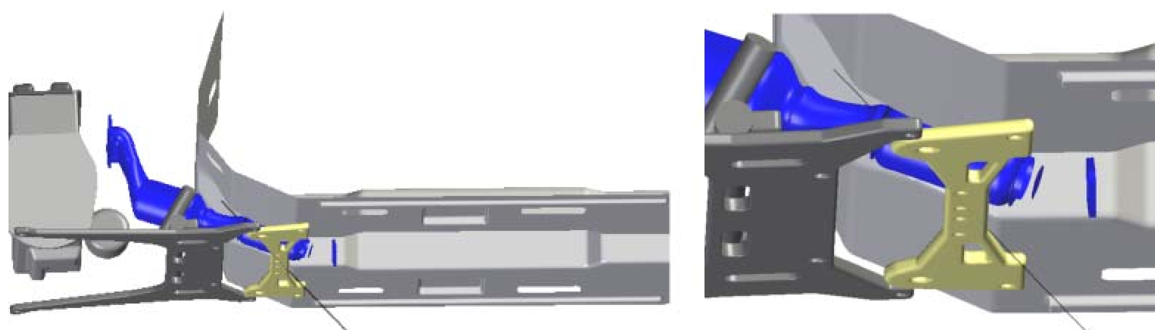

Figure 5.16: Exhaust test case. Extracting the blue exhaust system (left) is prevented by the yellow part, shown in the close-up (right). By declaring the yellow part an active obstacle, we can evaluate the necessary movement to correct the design error.

To do so, we start with a high play for the active obstacle, solve for a path, and then reduce the play and solve again. This results in a series of increasingly difficult path planning problems. For example, we were unable to find a solution for the wiper at less than 4.2mm. Therefore, we can presume that this is close to the minimum play necessary to let the wiper pass.

The third case and forth cases uses a different methodology. Rather than identifying a specific part and trying to find its minimum play, we are simply searching for the path that is in least collision. The third case is a disassembly task in which a starter must be removed from a complex car motor but cannot fit between the obstacles (**Figure 5.17**). To solve this forced passage problem, we designate all the parts surrounding the target object as active obstacles, and let the planner find a solution path.

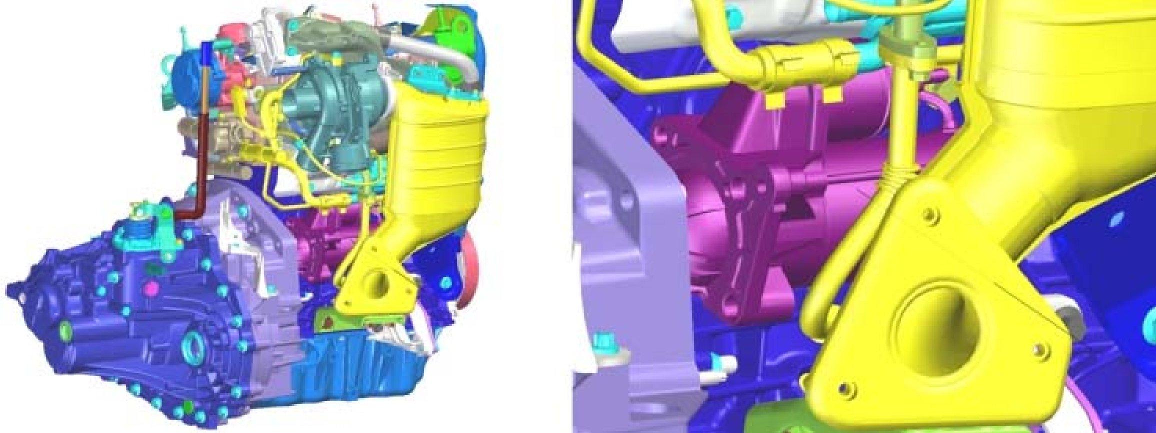

Figure 5.17: Starter test case. The goal is to extract the automobile starter in purple from the engine block (left). Since there is no non-colliding solution, we use our algorithm to find a "forced passage" or "least-worse" path. To this end, we designate several candidate parts as active obstacles, in yellow.

The fourth and final case is part of a shock absorbing system. This is another example of a forced passage, but a more subtle one, with a required penetration of less than a millimeter. The difficulty involves a mirrored setup of screws that must be evaded while simultaneously avoiding a large block (**Figure 5.18**). Because the screws are positioned at an awkward angle, the object must be forced out.

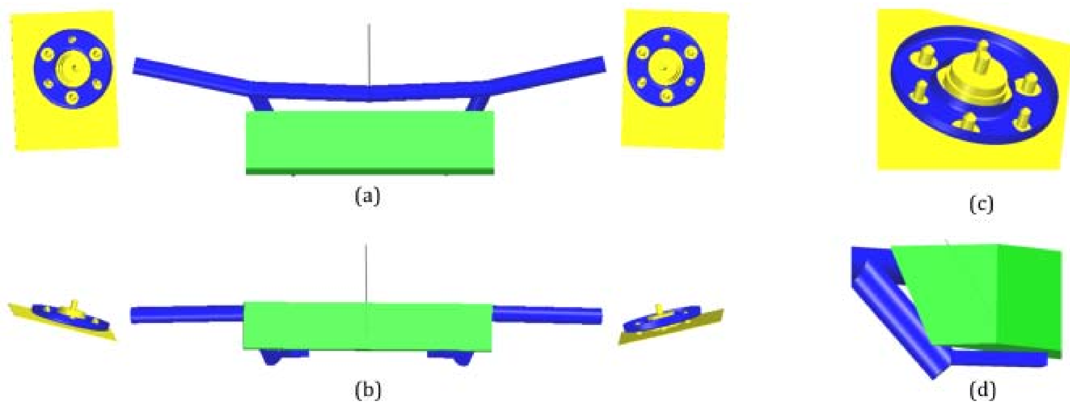

Figure 5.18: Shocks test case. The part in blue, although it appears disconnected, must move as one to both avoid the green block and the yellow screw setup in order to escape. Shown from the top (a), the bottom (b), and with close-ups on both the screws (c) and the locking mechanism (d), this is a difficult problem with no known collision-free solution. By designating both sections in yellow as separate active obstacles, we can solve for a path quickly.

To obtain performance results, we ran our algorithm 10 times at different play values. Information on the models is provided in **Table 5.1** and experimental results in **Table 5.2**.

Model	# Triangles	Approximate Minimum Play (mm)
Wiper	26486	4.2
Exhaust	32641	10
Starter	1936667	14
Shock	53132	0.9

Table 5.1: Active Obstacle Test Models. For each model, we give the geometrical complexity and the minimum play for which we have found a solution.

Model	Play (mm)	Algorithm	Performance (s)				
			Mean	Std. Dev.	1st Quartile	Median	3rd Quartile
Wiper	10	Teleportation	55.57	27.03	39.73	47.75	60.24
Wiper	10	Superposition	18.8	31.15	7.25	9.5	11.7
Wiper	7	Teleportation	544.38	328.59	262.4	600.6	645.13
Wiper	7	Superposition	74.1	192.02	9.6	11.54	15.75
Exhaust	15	Teleportation	975.17	735.84	700.66	935.26	1090.66
Exhaust	15	Superposition	1198.8	1612.67	194.5	799.5	1351.75
Exhaust	10	Teleportation	1973.71	2272.31	354.49	634.95	2852.99
Exhaust	10	Superposition	912.9	901.03	355.25	522	1234.75
Starter	20	Teleportation	193.4	98.62	149	187.5	212
Starter	20	Superposition	1120.7	1035.08	481	736	1082.25
Starter	15	Teleportation	931.4	1080.84	278.25	477.5	951
Starter	15	Superposition	2293.4	2442.31	550	856	3920
Shocks	2	Teleportation	98	58.484	71.75	82.5	96
Shocks	2	Superposition	39	24.71	16	35.5	54.74
Shocks	1	Teleportation	1711.3	898.95	1008.75	1271	489.5
Shocks	1	Superposition	916.1	1001.59	142.5	544	2497

Table 5.2: Active Obstacle Experimental Results. Each of the four models is tested at two different play values and for each of the two algorithms. Since probabilistic path planning tends to produce long-tailed performance distributions with large variances, we provide the quartiles as well. The results show that, in general, the Superposition operator gives significantly better results than the Teleportation-based Planner. A notable exception to this trend is found in the Starter model. Also we can observe that the lower the play, more difficult the problem becomes. An interesting contradicting example is the Exhaust case, for which a play of 10mm is found faster than one at 15mm.

5.8 Conclusion

We have attempted to tackle the poorly-posed problem of finding paths that are in collision. We introduce a formulation of the problem in which a set of obstacles (called *active obstacles*) are allowed a certain amount of penetration (called *play*) into the robot. The novelty of our approach to solving these cases is the implicit bounding of penetration distance through the path planning process rather than after–the-fact measurement. The advantage of pre– rather than post–PD measurement lies in performance and generality *vis-à-vis* the geometry in question, making it an ideal candidate for inclusion in industrial analysis software.

Our approach can assist mechanical designers in identifying and treating flaws in complex CAD assemblies. We present two algorithms to solve the problem. The first, the Teleportation-based planner, integrates the active obstacles directly into the configuration space, and lets an RRT solve for their positions along with that of the robot. The second, the superposition collision operator, builds only upon the collision-detection routines.

Comparing the two approaches, we observe the collision-detection approach is much faster in practice, but is not probabilistically complete as is the Teleportation planner. It would be advantageous to combine the two methods in order to benefit from the advantages of both. One simple way to do so would be to integrate the active obstacles into the robot as for the Teleportation planner, but include the superposition collision detection as well. If the superposition operator tested the planner-given position as well as the predefined offsets, the combined approach would be rendered probabilistically complete.

A useful capability currently missing from our approach is the ability to restrict the directions of movement of the active obstacles. In the disassembly case described by **Figure 5.3**, for example, one could limit the screws to only move within a given axis. This would be fairly easy to accomplish with either the Teleportation-based planner or the superposition operator.

Another interesting topic for future work concerns how best to handle multiple active obstacles that are very close to one another. Such a case multiplies the difficulty of the problem by forcing the planner to deal with several moving obstacles at once instead of in series. It would be interesting to have the planner intelligently group the active obstacles based on their relative positions in order to manipulate several at once.

Finally, in order to converge closer to the minimum allowed play, it would be ideal to allow the play to rise as well as fall. This raises difficult questions involving the detection of path-planning failure.

"The more technology becomes complicated inside, the more it has to be simple outside."

Derrick de Kerckhove

6 Future Directions

This doctoral work has not been focused on a single problem, but rather on several different aspects of robotic motion planning. As a cooperative work between a scientific laboratory and a technology start-up, we have focused on concrete issues of importance to the industrial community. We have nonetheless endeavored to place these projects in their larger scientific context.

6.1 Results and Future Work

In Chapter 3, we introduce a wrapping operator that encloses a collection of still or moving objects. It is designed to be both fast and general enough to handle the large polygon soup models and discontinuous paths that are often found in computer-aided design. By manipulating the intermediate grid data structure, we have been able to add an additional offset operator to inflate or deflate the volumes. We can foresee three major improvements:

First, In terms of efficiency, this offset operator is based on the calculation of many sphere-line intersections. Since these operations could be carried out in parallel, the intersection test appears to be ideal for porting over to the GPU. In terms of applicability, the deflation operator is inappropriate for non-volumetric geometry, as it removes all flat surfaces. Second, when used for swept volumes, an additional bounding distance parameter is needed to define the sampling density. For the time being, it is unclear which combination of these two parameters for a given error bound would produce the fastest execution.

Third, if an additional operator could be defined that uses the grid structure to calculate an object skeleton, then this skeleton could be preserved during the deflation process in order to produce a combination of flat surfaces and volumes where appropriate. This could also open the possibility of using the offset operator for path planning applications, since the skeleton could preserve topological equivalence.

We next turned our attention to motion planning. Through the abstract notion of configuration space, sampling-based motion planners can treat both the kinematic chain of the robot and the geometrical form of the objects (robot and obstacles) as black boxes. Provided that a collision detector can reply to proximity queries for the space in question, the same motion planning algorithm can apply to a large number of situations. But in order to obtain excellent performance, collision detection algorithms are highly specialized and “close to the data,” making it difficult to substitute one type of geometry for another in practice.

In Chapter 4, we have attempted to resolve this contradiction by identifying common threads between collision detectors. Reasoning through a common hierarchical structure, we put in place an algorithm that descends two such hierarchies in parallel. Proximity tests for each pair of elements is efficiently dispatched to specialized code, and the results are returned in a general form for use by the path planner. With such a project, it is necessary to strike a balance between generality and performance. Such a decision should be up to the developer of each geometry type. We have attempted to allow a range of possibilities by defining a unifying iterator interface that can be adapted to a variety of data compression techniques.

One part of the architecture that is not currently extendable is the proximity query itself. In future work, it would be useful to factor out this as well, allowing easy redefinition of new proximity queries such as n-nearest points, penetration distance, etc. Another interesting area for exploration is the inclusion of new techniques using the GPU, if common algorithms and structures could be identified, as well as maximizing performance on multi-core processors.

Finally, Chapter 5 addresses the problem of motion planning in the presence of collision. We have defined an allowed penetration distance with certain obstacles and have augmented the classic IPP algorithm to handle them. We present two approaches to this problem. The first adds DOFs for the active obstacles to the configuration space and then modifies certain aspects of the planner in order to explore that space more appropriately. The second deals with the active obstacles solely on the collision test level and in a deterministic fashion. For the majority of examples that we have tested, the first approach is slower than the second, but has an important property of probabilistic completeness which the second does not. A direction for future work might be to fuse the two algorithms together, drawing on the efficiency of the first approach and the random explorations of the second.

Additionally, this approach depends on the explicit designation of active obstacles. In some situations, it may be appropriate to divide a large obstacle into multiple active obstacles (e.g. a tunnel can be divided into its wall components). The issue may be complicated further when multiple active obstacles are concerned. In the forced passage problem, for example, it is unclear how best to distribute a given allowed penetration among the active obstacles, nor how to reduce and balance the different play values when dynamic play is used. The importance of user intervention and semantics is a fundamental property of our approach, and could be seen a drawback in some circumstances. In order to mitigate the issue, it could be interesting to develop a strong interface that helps the user decide what obstacles should be active, and how to determine the play.

6.2 Working with Industrial Robotics

From working with clients of Kineo CAM, we have seen how motion planning is both an enormous help and cause of great confusion for them. Specifically, it is very difficult for users to understand how best to parameterize an algorithm or notice potential failure points. It is for this reason that IPP, which automatically tunes the RRT extend-step parameter, has made such inroads with the industrial community. We believe that there is even more untapped potential in simplifying the algorithms so that users can accomplish more while the system handles any unknowns in the system without their intervention. Doing so also eases the reuse of motion planners for larger problems such as robot placement and sequence planning.

On another note, motion planning stands to benefit greatly from increasing parallelization in computer processing. Roadmap-based algorithms such as the PRM and the RRT could easily be performed in parallel. Interesting work could be done in designing mechanisms to synchronize updates to the roadmap, which is the only necessary shared structure. Improved processing power also opens the door up to simulating more complex deformable objects, such as flexible tubes and bendable plastics, both common elements of industrial designs.

An industrial robot might not have the flair of its cousins in space-exploration or human interaction, but it has become a mainstay of the manufacturing and maintenance industries. As almost all of product design and prototyping is now done on the computer, we believe that industrial robotics will look even more to motion planning techniques in the future to help it accomplish tasks as quickly, and as simply, as possible.

Bibliography

K. ABDEL-MALEK, D. BLACKMORE, and K. JOY, "Swept Volumes: Foundations, Perspectives, and Applications," *International Journal of Shape Modeling*, 2002.

K. ABDEL-MALEK and S. OTHMAN, "Multiple Sweeping Using the Denavit-Hartenber Representation Method," *Computer-Aided Design and Applications*, vol. 31, pp. 567-583, 1999.

K. ABDEL-MALEK, W. SEAMAN, and H.-J. YEH, "Nc Verification of up to 5 Axis Machining Processes Using Manifold Stratification," *ASME Journal of Manufacturing Science and Engineering*, vol. 122, pp. 1-11, 2000.

K. ABDEL-MALEK, J. YANG, R. BRAND, and E. TANBOUR, "Towards Understanding the Workspace of Human Limbs," *Ergonomics*, vol. 47, pp. 1386-1406, 2004.

K. ABDEL-MALEK and H.-J. YEH, "On the Determination of Starting Points for Parametric Surface Intersections," *Computer Aided Design*, vol. 29, pp. 21-35, 1997.

S. ABRAMS, P. K. ALLEN, and K. TARABANIS, "Computing Camera Viewpoints in an Active Robot Work Cell," *International Journal of Robotics Research*, vol. 18, pp. 267-285, 1999.

G. V. D. BERGEN, "Efficient Collision Detection of Complex Deformable Models Using AABB Trees," *Journal of Graphics Tools*, vol. 2, pp. 1-13, 1997.

D. BLACKMORE and M. C. LEU, "A Differential Equation Approach to Swept Volumes," in *Proceedings of Rensselaer's 2nd International Conference on Computer Integrated Manufacturing* (1990), Troy, New York, USA, May 1990, pp. 143-149.

J. BLOOMENTHAL, "Polygonization of Implicit Surfaces," *Computer Aided Geometric Design*, vol. 5, pp. 34 -355, 1988.

J.-D. BOISSONNAT and M. YVINEC, *Algorithmic Geometry*: Cambridge University Press, 1998.

C. BRAILLON, C. PRADALIER, K. USHER, J. CROWLEY, and C. LAUGIER, "Occupancy Grids from Stereo and Optical flow Data," in *Experimental Robotics*, vol. 39, *Springer Tracts in Advanced Robotics*. Berlin: Springer, 2008, pp. 367-376.

S. A. CAMERON and B. CULLEY, "Determining the Minimum Translational Distance between Two Convex Polyhedra," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)* (1986), March, pp. 591-596.

J. CANNY, *The Complexity of Robot Motion Planning*, PhD thesis. MIT, Cambridge, MA, USA, 1988.

T. CAO and A. C. SANDERSON, "Task Decomposition and Analysis of Assembly Sequence Plans Using Petri Nets," in *Proceedings of Computer Integrated Manufacturing* (1992), May 20-22, pp. 138-147.

J. CONKEY and K. I. JOY, "Using Isosurface Methods for Visualizing the Envelope of a Swept Trivariate Solid," in *Proceedings of Pacific Graphics* (2000), Hong Kong, October 3-5, 2000, pp. 272--280.

J. CORTÉS, L. JAILLET, and T. SIMÉON, "Molecular Disassembly with RRT-Like Algorithms," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)* (2007), Rome, Italy, pp. 3301-3306.

B. DACRE-WRIGHT, J.-P. LAUMOND, and R. ALAMI, "Motion Planning for a Robot and a Movable Object Amidst Polygonal Obstacles," in *Proceedings of IEEE International Conference on Robotics and Automation* (1992), Nice, France, May 10-15, pp. 2474-2480.

A. DELCHAMBRE, "A Pragmatic Approach to Computer-Aided Assembly Planning," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)* (1990), pp. 1600-1605.

A. ELFES, "Using Occupancy Grids for Mobile Robot Perception and Navigation," in *Computer*, vol. 22, 1989, pp. 46-57.

E. FERRÉ and J.-P. LAUMOND, "An Iterative Diffusion Algorithm for Part Disassembly," in *Proceedings of International Conference on Robotics and Automation* (2004), New Orleans (USA), pp. 3149-3154.

S. FISHER and M. C. LIN, "Deformed Distance Fields for Simulation of Non-Penetrating Flexible Bodies," in *Proceedings of Eurographic Workshop on Computer Animation and Simulation* (2001), Manchester, UK, pp. 99-111.

A. FOISY and V. HAYWARD, "A Safe Swept Volume Method for Robust Collision Detection," in *Proceedings of Robotics Research, Sixth International Symposium* (1994).

S. F. GIBSON, "Beyond Volume Rendering: Visualization, Haptic Exploration, and Physical Modeling of Voxel-Based Objects," in *Visualization in Scientific Computing '95*, R. Scanteni, J. v. Wijk, and P. Zanarini, Eds.: Springer-Verlag Wien, 1995, pp. 9-24.

S. GOTTSCHALK, *Collision Queries Using Oriented Bounding Boxes*thesis. University of North Carolina, 1998.

S. GOTTSCHALK, M. C. LIN, and D. MANOCHA, "Obbtree: A Hierarchical Structure for Rapid Interference Detection," in *Proceedings of Computer Graphics and Interactive Techniques* (1996), pp. 171-180.

D. HSU, *Randomized Single-Query Motion Planning in Expansive Spaces*, PhD thesis. Stanford University, Stanford, CA, USA, 2000.

D. HSU, L. E. KAVRAKI, J.-C. LABOMBE, R. MOTWANI, and S. SORKIN, "On Finding Narrow Passages with Probabilistic Roadmap Planners," in *Proceedings of Robotics: The algorithmic perspective* (1998), Houston, Texas, USA, pp. 141-153.

D. HSU, G. SANCHEZ-ANTE, H.-L. CHENG, and J.-C. LATOMBE, "Multi-Level Free-Space Dilation for Sampling Narrow Passages in PRM Planning," in *Proceedings of IEEE International Conference on Robotics and Automation* (2006), Orlando, Florida, USA, pp. 1255- 1260.

T. INAMURA, K. OKADA, M. INABA, and H. INOUE, "Hrp-2w: A Humanoid Platform for Research on Support Behavior in Daily Life Environments," in *Proceedings of International Conference on Intelligent Autonomous Systems* (2006), pp. 732-739.

M. INUI, M. MIURA, and F. KIMURA, "Analysis of Position Uncertainties of Iarts in an Assembly Using Configuration Space in Octree Representation," in *Proceedings of ACM Symposium on Solid and Physical Modeling* (1995), Salt Lake City, Utah, USA, pp. 73-82.

ISO, "Manipulating Industrial Robots - Vocabulary," International Organization for Standardization, Report 8373:1994, 1994.

P. ISTO, M. MÄNTYLÄ, and J. TUOMINEN, "On Addressing the Run-Cost Variance in Randomized Motion Planners," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)* (20003), pp. 2934-2939.

R. JAYARAMAN and V. SRINIVASAN, "Geometric Tolerancing: I. Virtual Boundary Requirements," *IBM Journal of Research and Development*, vol. 33, pp. 90-104, 1989.

L. JOSKOWICZ, E. SACKS, and V. SRINIVASAN, "Kinematic Tolerance Analysis," *Computer-Aided Design*, vol. 29, pp. 147-157, 1997.

A. KAUFMAN, D. COHEN, and R. YAGEL, "Volume Graphics," in *IEEE Computer*, vol. 26, 1993, pp. 51-64.

I. KENNETH E. HOFF, T. CULVER, J. KEYSER, M. LIN, and D. MANOCHA, "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware," in *Proceedings of SIGGRAPH* (1999), pp. 277-286.

Y. J. KIM, M. C. LIN, and D. MANOCHA, "Fast Penetration Depth Estimation Using Rasterization Hardware and Hierarchical Refinement," in *Proceedings of Workshop on Algorithmic Foundations of Robotics (WAFR)* (2002).

Y. J. KIM, G. VARADHAN, M. C. LIN, and D. MANOCHA, "Fast Swept Volume Approximation of Complex Polyhedral Models," *ACM Symposium on Solid and Physical Modeling*, pp. 11-22, 2003.

J. T. KLOSOWSKI, M. HELD, J. S. B. MITCHELL, H. SOWIZRAL, and K. ZIKAN, "Efficient Collision Detection Using Bounding Volume Hierarchies of K-Dops," in *Proceedings of Visualization and Computer Graphics* (1998), pp. 21-36.

L. P. KOBBELT, M. BOTSCH, U. SCHWANECKE, and H.-P. SEIDEL, "Feature Sensitive Surface Extraction from Volume Data," in *Proceedings of SIGGRAPH* (2001), pp. 57-66.

J. J. KUFFNER and S. M. LAVALLE, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)* (2000), San Francisco, CA, USA, pp. 995-1001.

T. LARSSON and T. AKENINE-MÖLLER, "A Dynamic Bounding Volume Hierarchy for Generalized Collision Detection," in *Proceedings of Workshop On Virtual Reality Interaction and Physical Simulation* (2005).

J.-C. LATOMBE, "Motion Planning: A Journey of Robots, Molecules, Digital Actors, and Other Artifacts," *International Journal of Robotics Research*, vol. 18, pp. 1119-1128, 1999.

J.-C. LATOMBE and R. H. WILSON, "Assembly Sequencing with Toleranced Parts," in *Proceedings of ACM Symposium on Solid and Physical Modeling* (1995), pp. 83-94.

J.-P. LAUMOND, "Motion Planning for Plm: State of the Art and Perspectives," *International Journal of Product Lifecycle Management*, vol. 1, pp. 129-142, 2006.

S. M. LAVALLE, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," Computer Science Dept., Iowa State University TR 98-11, 1998.

S. M. LAVALLE, *Planning Algorithms*: Cambridge University Press, 2006.

C. C. LAW, LISA S AVILA, and W. J. SCHROEDER, "Application of Path Planning and Visualization for Industrial Design and Maintainability Analysis," in *Proceedings of Reliability and Maintainability Symposium* (1998).

M. LIN, D. MANOCHA, J. COHEN, and S. GOTTSCHALK, "Collision Detection: Algorithms and Applications," in *Algorithms for Robotics Motion and Manipulation: 1996 Workshop on*

the Algorithmic Foundations of Robotics, J.-P. Laumond and M. Overmars, Eds.: A K Peters, Ltd., 1996, pp. 129-142.

M. C. LIN and D. MANOCHA, "Collision and Proximity Queries," in *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, Eds. Boca Raton, FL, USA: CRC Press, 2004, pp. 787-808.

W. E. LORENSEN and H. E. CLINE, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," in *Proceedings of SIGGRAPH* (1987), pp. 163-169.

T. LOZANO-PÉREZ, *The Design of a Mechanical Assembly System*, Master's thesis. MIT, Cambridge, MA, USA, 1976.

T. LOZANO-PÉREZ, "Spatial Planning: A Configuration Space Approach," *IEEE Transactions on Computers*, vol. 32, pp. 108-120, 1983.

R. R. MARTIN and P. C. STEPHENSON, "Sweeping of Three-Dimensional Objects," *Computer Aided Design*, vol. 22, pp. 223-234, 1990.

E. MAZER, J. M. AHUACTZIN, and P. BESSIÈRE, "The Ariadne's Clew Algorithm," *Journal of Artificial Intelligence Research (JAIR)*, vol. 9, pp. 295-316, 1998.

W. A. MCNEELY, K. D. PUTERBAUGH, and J. J. TROY, "Six Degree-of-Freedom Haptic Rendering Using Voxel Sampling," in *Proceedings of Computer Graphics and Interactive Techniques* (1999), pp. 401-408.

J. M. MILNER, S. C. GRAVES, and D. E. WHITNEY, "Using Simulated Annealing to Select Least-Cost Assembly Sequences," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)* (1994), pp. 2058-2063.

H. MORAVEC, *Mind Children: The Future of Robot and Human Intelligence*. Cambridge, MA, USA: Harvard University Press, 1988.

T. M. MURALI and T. A. FUNKHOUSER, "Consistent Solid and Boundary Representations from Arbitrary Polygonal Data," in *Proceedings of SIGGRAPH Symposium on Interactive 3D Graphics* (1997).

N. J. NILSSON, "Shakey the Robot," SRI International 323, 1984.

C. PESCIO, "Multiple Dispatch: A New Approach Using Templates and RTTI," *C++ Report*, 1998.

S. QUINLAN, "Efficient Distance Computation between Non-Convex Objects," in *Proceedings of International Conference on Robotics and Automation* (1994).

S. REDON, Y. J. KIM, M. C. LIN, and D. MANOCHA, "Fast Continuous Collision Detection for Articulated Models," in *Proceedings of ACM Symposium on Solid Modeling and Applications* (2004), Genoa, Italy, pp. 145-156.

J. R. ROSSIGNAC and A. A. G. REQUICHA, "Offsetting Operations in Solid Modelling," *Computer Aided Geometric Design*, vol. 3, pp. 129-148, 1986.

J. R. SACK and J. URRUTIA, *Handbook of Computational Geometry*. North Holland: Elsevier, 2000.

M. SAHA and P. ISTO, "Motion Planning for Robotic Manipulation of Deformable Linear Objects," in *Proceedings of IEEE International Conference on Robotics and Automation* (2006), pp. 2478-2484.

W. J. SCHROEDER, W. E. LORENSEN, and S. LINTHICUM, "Implicit Modeling of Swept Surfaces and Volumes," in *Proceedings of IEEE Visualization* (1994), pp. 40-45.

W. J. SCHROEDER, J. A. ZARGE, and W. E. LORENSEN, "Decimation of Triangle Meshes," in *Proceedings of International Conference on Computer Graphics and Interactive Techniques* (1992), pp. 65-70.

J. T. SCHWARTZ, M. SHARIR, and J. E. HOPCROFT, *Planning, Geometry, and Complexity of Robot Motion*, vol. 4. Norwood, New Jersey, USA: Ablex Publishing Corporation, 1986.

F. SCHWARZER, M. SAHA, and J.-C. LATOMBE, "Exact Collision Checking of Robot Paths," in *Algorithmic Foundations of Robotics*, J. D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, Eds.: Springer 2004, pp. 25-41.

K. T. SEOW and R. DEVANATHAN, "A Temporal Framework for Assembly Sequence Representation and Analysis," *IEEE Transactions on Robotics and Automation*, vol. 10, pp. 220-229, 1994.

J. A. SETHIAN, "A Fast Marching Level Set Method for Monotonically Advancing Fronts," *Proceedings of the National Academy of Sciences (USA)*, vol. 93, pp. 1591-1595, 1996.

V. SRINIVASAN and R. JAYARAMAN, "Geometric Tolerancing: Ii. Conditional Tolerances," *IBM Journal of Research and Development*, vol. 33, pp. 105-124, 1989.

S. SUNDARAM, I. REMMLER, and N. M. AMATO, "Disassembly Sequencing Using a Motion Planning Approach," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)* (2001), pp. 1475-1480.

P. SVESTKA and M. H. OVERMARS, "Probabilistic Path Planning," in *Robot Motion Planning and Control*. Secaucus, NJ, USA: Springer-Verlag, 1998.

G. VARADHAN and D. MANOCHA, "Accurate Minkowski Sum Approximation of Polyhedral Models," *Graphical Models*, vol. 68, pp. 343-355, 2006.

G. WILFONG, "Motion Planning in the Presence of Movable Obstacles," in *Proceedings of Symposium on Computational Geometry* (1988), Urbana-Champaign, Illinois, United States, pp. 279-288.

R. H. WILSON and J.-C. LATOMBE, "Geometric Reasoning About Mechanical Assembly," *Artificial Intelligence*, vol. 71, pp. 371-396, 1995.

C. K. YAP and E.-C. CHANG, "Issues in the Metrology of Geometric Tolerancing," in *Algorithms for Robot Motion Planning and Manipulation*, J.-P. L. a. M. Overmars, Ed. Wellesley, Massachusetts, USA: A.K. Peters, 1997, pp. 393-400.

K. YOKOI, N. E. SIAN, T. SAKAGUCHI, O. STASSE, Y. KAWAI, and K.-I. MARUYAMA, "Humanoid Robot HRP-2 with Human Supervision," in *Experimental Robotics*, vol. 39, *Springer Tracts in Advanced Robotics*. Berlin: Springer, 2008, pp. 513-522.

L. ZHANG, Y. J. KIM, and D. MANOCHA, "A Fast and Practical Algorithm for Generalized Penetration Depth Computation," in *Proceedings of Robotics: Science and Systems Conference (RSS)* (2007).