

SANDIA REPORT

SAND97-0366 • UC-705

Unlimited Release

Printed February 1997

A Configuration Space Toolkit for Automated Spatial Reasoning: Technical Results and LDRD Project Final Report

Patrick G. Xavier, Robert A. LaFarge

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000

Approved for public release; distribution is unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A04
Microfiche copy: A01

A Configuration Space Toolkit for Automated Spatial Reasoning: Technical Results and LDRD Project Final Report

Patrick G. Xavier

Robert A. LaFarge

Intelligent Systems and Robotics Center
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-1008

Abstract

A robot's configuration space (c-space) is the space of its kinematic degrees of freedom, e.g., the joint-space of an arm. Sets in c-space can be defined that characterize a variety of spatial relationships, such as contact between the robot and its environment. C-space techniques have been fundamental to research progress in areas such as motion planning and physically-based reasoning. However, practical progress has been slowed by the difficulty of implementing the c-space abstraction inside each application. For this reason, we proposed a Configuration Space Toolkit of high-performance algorithms and data structures meeting these needs. Our intent was to develop this robotics software to provide enabling technology to emerging applications that apply the c-space abstraction, such as advanced motion planning, teleoperation supervision, mechanism functional analysis, and design tools.

This final report presents the research results and technical achievements of this LDRD project. Key results and achievements included (1) a hybrid Common LISP/C prototype that implements the basic C-Space abstraction, (2) a new, generic, algorithm for constructing hierarchical geometric representations, and (3) a C++ implementation of an algorithm for fast distance computation, interference detection, and c-space point-classification. Since the project conclusion, motion planning researchers in Sandia's Intelligent Systems and Robotics Center have been using the CSTk libcstk.so C++ library. The code continues to be used, supported, and improved by projects in the ISRC.

1 Introduction

1.1 Overview

A robot's configuration space (C-space) is the space of its kinematic degrees of freedom, e.g., the joint-space of an arm. Sets in C-space can be defined to characterize a variety of spatial relationships, such as contact between the robot and its environment. C-space techniques have been fundamental to recent basic progress in robotics areas such as motion planning and physically-based reasoning. However, practical progress has been slowed by the difficulty of implementing the C-space abstraction inside each application. For this reason, we proposed a Configuration Space Toolkit (CSTk) of high-performance algorithms and data structures meeting these needs. The project intent was to develop this robotics software to provide enabling technology to key applications, such as motion planning, teleoperation supervision, mechanism functional analysis, and design tools.

The C-Space Toolkit project differed from previous efforts in several ways. First, our goal was a suite of C-space tools to cover a broad range of end-uses. Second, we emphasized efficient implementations. Third, although our prototype Common LISP and current C++ implementations model only faceted real-world objects, the Toolkit is designed for extensibility to other geometric models via any solid modeler that supports certain primitive operations. Most previous applications using C-space implementations have focused on demonstrating the validity of a method based on a C-space approach and are limited to one type of geometric model. Our goal was to make C-space techniques easier to use, computationally efficient, and largely independent of any particular geometric modeler.

Our approach was first to develop an implementation that supported configuration spaces of arbitrary dimension for planar objects. As we acquired an understanding of how computational complexity issues manifested themselves in practice, we evolved an architecture for the toolkit. Specifically, we realized that the efficient implementation of high-level abstract and set operations on C-space objects depends on low-level operations on the simplest of these objects. Furthermore, these low-level operations in combination with functions that define the simple C-space were adequate for several targeted applications. We called this set of operations the *C-Space Toolkit Kernel*. We describe techniques for three key tasks in this report: point-classification in C-space and distance computation (§2.3), the construction of hierarchical representations of geometric objects (§3), and the computation of surface normals (§4).

As completion of the Kernel neared, we faced the choice of either fleshing out the CSTk above the kernel level in Common LISP or implementing a "production" version of the CSTk in a more efficient and widely used language. We chose the latter and, at conclusion of project, delivered a usable C++ implementation of the C-Space Toolkit to our motion planning group for integration with the SANDROS motion planner. After conclusion of project, CSTk code has continued to be actively used and further developed. This will be reported in forthcoming publications.

1.2 Configuration Space Basics

We now give some basic definitions.

A *configuration space* of an object is a space of parameter values that completely specifies the position of all points of that object relative to some frame. When the space is dimensionally minimal, it is often referred to as *the* configuration space of that object. For example, the configuration space of a robot manipulator is its joint space, i.e., the set of its possible joint values. Assuming no constraints, the configuration space of rigid body in the plane is $\mathbb{R}^2 \times S^1$, and the configuration space of a rigid 3D body is $\mathbb{R}^3 \times SO(3)$.

Constraints on an object in the world (workspace) can be mapped to entities in configuration space. In particular, one can map obstacles in a robot’s workspace to *configuration space obstacles*, or *C-obstacles*. Suppose that object \mathcal{R} has configuration space \mathcal{C} , and that for any configuration $\mathbf{x} \in \mathcal{C}$, $\mathcal{R}(\mathbf{x})$ is the region of the workspace \mathcal{W} occupied by \mathcal{R} when it is in configuration \mathbf{x} . Let \mathcal{O} be the regions of \mathcal{W} occupied by obstacles. Then the C-obstacles of \mathcal{R} in \mathcal{W} are given by

$$\{\mathbf{x} \in \mathcal{C} : \mathcal{R}(\mathbf{x}) \cap \mathcal{O} \neq \emptyset\}.$$

Relative to arbitrary regions and surfaces in the workspace and arbitrary subsets of \mathcal{R} , we can define simple *C-space objects*, or *C-objects*, which are sets in C-space. Other sets can be arbitrarily defined in C-space, with or without reference to objects in the workspace. When a technique makes use of C-objects, we say it uses the *C-space abstraction*. The C-Space Toolkit attempts to implement the significant parts of the C-space abstraction with respect to C-objects that depend on entities in or mathematically defined in the workspace.

1.3 Applications and Intended Applications

There have been literally hundreds of robotics research papers that apply the C-space abstraction. Here we attempt to summarize the work that we see as most influential in the application of C-space in practice now and in the near future. Supporting current and expected practical research and applications of C-space was a major goal of the C-Space Toolkit project.

In robotics, the C-space abstraction was originally applied in motion planning. [LP83] introduced the use of C-space to reduce the path planning problem for a robot to a path-planning problem for a point. This led to results in the theoretical analysis of motion planning problems [CR87] and development of complete shortest-path planning algorithms [SS84, Can88]. C-space has also been of practical significance to motion planning, since it was used in early implemented, full-DOF algorithms (e.g., [Don87]) and practical algorithms (e.g., [BL89]). To this day, many of the most successful path planners [KL94, CQM95, CH92] operate in C-space, and their development continues.

As physicists had used the C-space abstraction in classical mechanics, it was natural for robotics researchers to cast many other problems into the C-space abstraction. Significant examples include the analysis and planning of motions when there is sensor, model, or control uncertainty, when contact affects motion, or when the task is to manipulate an object. [LPMT84] introduced a new paradigm of fine-motion planning in robotics combining the use of sensor, model, and control uncertainty with a dynamics model and C-space sets. [Erd86] introduced techniques for computing *backprojections* — regions of C-space from which reaching a goal configuration is guaranteed to be achieved and detected, given sensing, control/dynamics, geometry and uncertainty models.

[Bro86] applied C-space techniques to analyzing and planning the motions of physical objects, with grasping as the particular application. [Don89] extensively used C-space in extending the work of [LPMT84, Erd86] to cover a theory of error detection and control. C-space techniques were also instrumental in the development of algorithms for sensorless manipulation, such as those of [EM88].

More recently, researchers developed techniques that incorporated a non-linear dynamics model [Bro91] and brought probabilistic techniques [GM90, BC94] into the definitions and computation of C-space sets. These developments have been important in bringing of C-space techniques for manipulation and grasping into the practical realm.

1.4 Project Accomplishments

The C-Space Toolkit Project achieved three major accomplishments:

1. We completed the development and implementation of a hybrid Common LISP/C prototype of the CSTk for rigid bodies modeled as unions of convex polygons and polyhedra, with further support for planar objects. In this phase of the project we focused on the toolkit architecture, major algorithms, and data structures. (See §2.) The most important computation finds the distance and closest features of two bodies at a given configuration. The results of this query are used in doing point classification¹ and surface normal computation in C-space. The algorithm and experimental results using our C++ implementation (see below) are presented in §2.3.
2. We developed a new, generic, algorithm for constructing hierarchical geometric representations. (See §3.) Our algorithm uses a heuristic clustering technique to produce spatially balanced, coherent hierarchies. Its worst-case running time is $O(N^2 \log N)$, but for non-pathological cases it is $O(N \log N)$, where N is the number of input primitives. We have completed a C++ implementation for input collections of 3D convex polygons and 3D convex polyhedra and conducted simple experiments with scenes of up to 12,000 polygons. A 12,000 polygon scene takes from less than a minute to a few minutes to process, depending on the connectivity of the polygons. We present examples using spheres and convex hulls as hierarchy primitives. The algorithm has been used successfully with scenes of up to 98K polygons.
3. We implemented the most computationally intensive sections of the CSTk in C++. (See §2.3.4 and §2.3.5.) This subset of the CSTk was integrated with the SANDROS motion planner shortly after conclusion of our LDRD funding. The use of the C-Space Toolkit code immediately brought SANDROS planning time down into the range of minutes for real world gross-motion examples.

After the conclusion of the project, the C++ library has provided the foundation for further research. Further development on the code has supported work in motion planning and assembly

¹Determining its set membership, e.g., with respect to C-space obstacles.

planning. These developments will be documented in forthcoming publications. (See [Xav97, WX97]; the mapping to SAND reports has not been determined at this time.) In addition, the results of the C-Space Toolkit project will be used in the Sandia ISRC's development of a dynamical simulator for mechanisms and processes with intermittent contacts.

1.5 Report Roadmap

The rest of this report begins, in §2, by providing a high-level view of the C-Space Toolkit and presenting an algorithm for the key operation in answering C-space queries. We provide motivation for the project, describe how the intended run-time Toolkit operations can be supported by a kernel of operations, and show that the key to efficiently performing these operations is a fast algorithm for extended point classification. We provide an algorithm for hierarchical distance computation that fills this role, and present experimental results showing the advantages of (a) using convex hulls over axis-aligned bounding boxes in the hierarchy and (b) caching previous minimal-distance witness feature pairs².

The next section, §3, presents a generic algorithm that automatically constructs hierarchical geometric representations, such as those taken as input by the hierarchical distance algorithm. The algorithm can be applied to any of a number of geometric primitives, so long as a convex "wrapper" primitive is available. We present experimental results for convex-faceted arbitrary polyhedra, i.e., geometric object represented by collections of convex polygons. Our algorithm differs from previous algorithms by providing control over the spatial coherence of the hierarchy. This is achieved by using a clustering technique. The algorithm is $O(N \log^2 N)$ in the worst case, but we argue that $O(N \log N)$ can be expected when the input is not pathological.

The final section in the body of this report, §4 develops the mathematics the C-Space Toolkit uses to get from the generalized point-classification query to C-space normals. We note that this capability was only implemented in the Common LISP prototype. In Appendix A, we present an example illustrate how to use the C-Space Toolkit, using the Common LISP prototype.

§2.3.2 and §3.1.4 contain brief reviews of previous and related work, at the time of the project end.

2 Toolkit Architecture and A Central Algorithm

2.1 Introduction

We present two closely related aspects of the C-Space Toolkit (CSTk): its kernel architecture and the computational problem we saw as the performance bottleneck and attempted to address.

First, in §2.2, we describe the development of the CSTk architecture. We argue that present and future applications of C-space, combined with computation and data complexity issues, point to an architecture centered around a kernel of basic operations. We then list operations supported

²For two given objects, pairs of features whose distance is the minimal distance between the objects.

in the kernel and describe its role in supporting high-level set operations as a part of the C-space abstraction.

In §2.3, we argue that the bottleneck is the point-classification problem for one of the lowest-level C-objects: the C-obstacle corresponding to interference between a robot and a workspace obstacle or between two moving objects. This is closely related to distance computation between two rigid bodies, and we present an algorithm for distance computation and interference detection. This algorithm was implemented in C++, and we present experimental results.

2.2 Architecture

2.2.1 On Supporting Applications and Research

We now consider the various ways in which applications and research efforts could make use of a C-space toolkit, as seen in 1993:

- Path planning requires collision detection in which conservative approximations are often acceptable.
- Physical simulation requires higher accuracy and the determination of which objects are in or nearly in contact (in the world). Collision avoidance has similar needs.
- Fine motion planning utilizes C-obstacle surface normals and boolean C-space set operations. In addition, it could benefit from the generalization of CSG (constructive solid geometry) operations to C-space and from swept-hyperplane tools for constructing sets.
- Functional analysis might require computing the connectivity and dimensionality of free space regions.

Not only is the list of desired capabilities long, but there are a great many cases. The algorithms and Toolkit code appropriate for a given robot will depend on the dimension and topology of its underlying C-space, the dimension of the workspace, and the relative orientation of joints. The effect of the last factor initially might not be obvious, but we observe that collision detection for the first three links of a PUMA arm can exploit the fact that the second and third joints have parallel axes [BN90a].

The range of geometric primitives also increases the possible scope. While many motion planners and collision detectors have assumed that robots and all objects in the world are composed of convex polyhedra, this assumption needs eventually to be removed. The Toolkit design must include provisions for the incorporating special algorithms for shapes described by NURBS, parameterized surfaces, etc. Finally, users have a range of performance demands. For example, collision-avoidance for one robot in an static environment requires only moderate speed and accuracy, compared to the needs of physically accurate dynamic simulation for process evaluation.

With such an enormous range of desired capabilities, we realized that there was little hope of developing a library that supported all of them. Carefully restricting the scope of the project was necessary. We therefore set two objectives for the CSTk.

- The CSTk would support a subset of capabilities well enough to bring an applied research project from the research stage to the development stage or beyond.
- The CSTk would be extensible to support more geometric primitives and more capabilities as new techniques were developed.

2.2.2 C-Space as an Abstract Data Type

To understand our choice of what to develop and how it fits into a long-term framework for implementing the C-space abstraction, it is necessary to review what a *representation* is.

Basically, *a software system can be considered to support the representation of an object if it can answer a characteristic set of queries about that object.* For example, we might say that a system contains a representation of a C-space obstacle if the user can call functions that answer questions such as whether a configuration is within that obstacle, what the normal to a regular surface point is, and what (a basis for) the tangent space at a regular surface point is. *A software system is said to support an operation if it can compute a representation of the result of applying that operation to the objects for which it has representations.* For example, if a system supports the union operation over C-space obstacles, then if it has been instructed to form the union of two C-space obstacles, the system will be able to (a) answer all required queries about the resulting object and (b) use the resulting object as a union operand in the same sense.

The key point is that the data structure in memory and on disk does not need to closely correspond to the mathematical object — it does not need to be *explicit*. For example, there is no reason to maintain stored structures for each vertex, hyper-edge, etc. of a C-space obstacle. The above definition of a *representation* might seem overly broad at first — it implies that a geometric description of a robot and workspace objects and a record of C-space operations would constitute a C-space representation, provided that this information was adequate to allow the system to answer the appropriate set of queries. At first, this might not fit the intuitive notion of a representation, but the apparent discrepancy is a matter of response time, not information content. A simple example of how computation can “make up” for a minimal data structure comes from the database domain, where a set of base relations and logs of the queries and transactions allows the current state of a database to be recovered when the main data structures are damaged. Although the above notions of “representation” and “operation support” are simple, they are important to the architecture of the C-Space Toolkit.

2.2.3 A Kernel-based Toolkit

Gross-motion planning and collision avoidance/detection applications require determining the distance between objects that must not come in contact with each other. In the C-space view, these objects *induce* C-space obstacles. Thus, the distance computation is a type of point-classification for C-space objects defined by the geometry of objects in real space. If we further extend point-classification to include determining the (real) object features that are *witnesses* to the distance

measurement, then we also have enough information to compute normal and tangent directions on the surfaces of induced C-space objects.

At the same time, when building an exact local topological representation of C-space, such *extended point-classification* information allows one to substantially reduce the number of C-surfaces that need to be intersected, and normal and tangent information can be computed by numerical techniques. Furthermore, in higher dimensions, any explicit construction must be local and incremental to avoid the combinatorial explosion. *Thus, we see extended point classification as the key to implementing explicit representations of C-objects.*

Because (a) immediately foreseen applications and the practical use of exact representations both appear to call for the extended point-classification query described above, and (b) implementation of extended point classification can be compartmentalized, we decided to build the C-Space Toolkit around a Kernel devoted to extended point-classification. The rest of the Kernel includes simple operations for defining (induced) C-space objects, manipulating them as sets, and associating arbitrary data with them. Additional Toolkit components, such as explicit C-space set-construction for fine-motion planning, can be built as pseudo-applications on top of the Kernel. By extending the geometric models of the world and the robot handled by the kernel, one would automatically extend what is handled by the pseudo-applications. In turn, applications that only require the view of C-space provided by the Toolkit would also be automatically extended.

2.2.4 C-Space Toolkit Definitions and Operations

We now give some definitions needed in our description of the Kernel. Types of objects not supported by the Kernel are noted.

First, *induced C-objects* (induced C-space sets) are defined by the robot geometry (including kinematics) in combination with real objects or mathematical entities in the workspace. Specifically, the objects that determine the underlying configuration space will be known as *r-objects*, and all other objects in the workspace will be known as *s-objects*. Induced C-objects are simply sets in the configuration space that correspond to contact or interference between r-objects and s-objects or among r-objects. The most common example of an induced C-object is a C-space obstacle corresponding to contact between a robot body and a workspace obstacle. An r-object can be used as a nominal s-object in order to define a C-object — to define one corresponding to interference between two robots, for example.

Second, *pure C-space sets* are defined mathematically in C-space without reference to entities in the workspace. The set of configurations forbidden by joint limits is an example of such a set. The CSTk Kernel allows the definition of joint limits but does not support their abstraction to C-object status. This would be an easy extension.

Third, *ordinary compound C-space sets* are the results of Boolean set operations on pure, induced, or ordinary compound C-space sets. An example would be the set that corresponds to a robot being in contact with both obstacle A and obstacle B. The CSTk Kernel supports the union of C-objects, but not operational closure under other Booleans. An extension similar to the implementation of the support for unions can easily be done for point classification, but not for distance.

Finally, *special C-space sets* depend on non-Boolean operations on C-objects. These operations include swept volume or expansion according to a differential rule, and the class of sets includes *special compound C-space sets* — sets arising from Boolean set operations on members of this class. C-space sets that are not special — i.e., those from the previous three classes — are simply referred to as *ordinary*. Although the Kernel does not support special C-space sets, we expect that the extended point classification query with respect to ordinary C-space sets would be useful in their implementation.

We now describe basic operations and queries in the Kernel. The most basic operations are needed to define induced C-objects.

1. *Input r-objects. E.g., input the robot's geometry and kinematics and specify the underlying C-space.*
2. *Add (delete) s-objects to (from) the robot's environment.*
3. *Define C-objects that arise from a collection of r-objects and a collection of s-objects*

There are four core queries on top of which other queries are be implemented. For each configuration \mathbf{x} and C-object Y , these are

4. *Determine whether $\mathbf{x} \in Y$. This reduces to instances of interference detection, possibly combined with walking a Boolean tree.*
5. *If $\mathbf{x} \in \partial Y$ (or $\mathbf{x} \tilde{\in} \partial Y$), then determine which pairs of r-object and s-object features correspond to this contact.*
6. *At configuration \mathbf{x} , determine what is the minimum distance between the r-objects and s-objects that induce a given C-object Y , and which (r-object, s-object) feature pair(s) is a witness pair.*
7. *At configuration \mathbf{x} , find all (r-object, s-object) feature pair(s) that are at most a given positive distance ϵ_q apart and that (partially) induce a given C-object Y .*

The two last queries are supported for induced C-objects and their unions. Although this distance computation is really a function on $C\text{-space} \times C\text{-objects}$ rather than a measure of a quantity that naturally exists in C-space, it is extremely useful in motion planning, collision avoidance, and simulation. Furthermore, C-space “distance” is often much more expensive to compute than workspace distance in many applications and is often not intuitive. Although the result of the workspace distance query will not generally map to the distance to the C-object, and although the C-object induced by \mathcal{A} and \mathcal{B} might be null when $d(\mathcal{A}(\mathbf{x}), \mathcal{B})$ is always finite, a related workspace distance query can in practice be used to select a set of candidate C-object surfaces that might contain the closest point as long as the workspace distance is sufficiently small.

Furthermore, when the Jacobian and forward kinematic map are available, a pair of distance witnesses can be associated with a distance gradient direction in C-space. When the two features

are very close together, this vector approximates the C-object's surface normal. Thus, another query we include in the Kernel is

8. *Compute the distance gradient vectors for all (r-object, s-object) feature pair(s) that are at most a given positive distance ϵ_q apart.*

The mathematical development for this computation is given in §4.

For extensibility to curved surface geometries, surface patches could be associated with sets of polygons and polyhedra that spatially approximate them. Exact distance computation and interference computation could then be done by using the queries to the polygon/polyhedra model to narrow down the set of pairs of relevant surface patches and then performing computations on these pairs. If the distance function between surface patches can be differentiated, then extension of the Kernel to the surface patch type can be completed.

2.2.5 An Implementation Choice: Speed Versus Scope

We completely implemented the C-Space Toolkit Kernel for rigid objects modeled as unions of convex polygons and convex polyhedra in a Common LISP prototype. After implementing a couple of simple pseudo-applications and demonstration programs that illustrated programming in the C-space abstraction with the Kernel, we faced a choice. Either we could implement more of the C-space abstraction in Common LISP, or we could implement the most crucial code from the Kernel in C++ for internal release and use in further research. We chose the latter for two reasons.

First, we saw that there was much potential for immediate use of a fast C-Space Toolkit Kernel in an easily linkable C++ library, while demand would be small for a more complete but slower implementation of the C-space abstraction in Common LISP. Among the waiting internal customers were the researchers who developed the SANDROS motion-planning algorithm [CH92], which relies on the use of the C-space abstraction and showed great promise. The development of their efficient path-planning algorithm into a practical tool was hindered by the lack of adequately fast code for exact distance computation. In addition, without such code, it was nearly impossible to perform experiments with realistically geometrically complex examples, and this was an obstacle to further theoretical insight and algorithm development.

Second, the planned functionality beyond the Toolkit Kernel had been dictated by the needs of research in fine-motion planning and the analysis of manipulation tasks, and these needs for the C-space abstraction had changed. At the beginning of the project, explicit representations of C-space sets were strongly desired. However, as the project progressed, a consensus developed that this was not computationally feasible for C-spaces of more than three or four dimensions. Furthermore, with the development of a framework for the probabilistic analysis of manipulation tasks [BC93, BC94, BC95], it appeared that the C-Space Toolkit could better support research in manipulation by contributing to efficient high-fidelity dynamical simulation. The key contributions we foresaw were in distance computation, collision detection, and contact determination — problems that mapped to the Toolkit Kernel.

2.3 Point-Classification in C-Space and Distance Computation

In this section we present methods for computing the C-space extended point classification query described in §2.2.3. The main algorithm we present in §2.3.3 not only computes this query but several variations of the distance query. We present optimizations used in our implementation at the time of the project conclusion and experimental results in §2.3.4 and §2.3.5. The experimental results show the effectiveness of using convex-hull-based geometric hierarchies, especially in distance computation.

2.3.1 Background: Collision, Interference, and Distance

As pointed out in our definition of induced C-objects in §2.2.4, point classification for induced C-objects reduces to interference detection between the r-object and the s-object. We now review the informal definitions of interference detection, distance computation, and collision detection for a single moving body. *Interference detection* (or *clash detection*) for a rigid body \mathcal{R} at a position and orientation \mathbf{x} and obstacles \mathcal{O} means to determine whether $\mathcal{R}(\mathbf{x}) \cap \mathcal{O}$ is non-empty, with $\mathcal{R}(\mathbf{x})$ denoting the image of \mathcal{R} in the world. *Distance computation* means to determine the minimum distance between all points in $\mathcal{R}(\mathbf{x})$ and those in \mathcal{O} . *Collision detection* for a rigid body \mathcal{R} moving over a path segment among obstacles \mathcal{O} means to determine whether at any point along the path the body contacts or intersects any obstacles. In other words, if we let \mathcal{C} denote the space of positions and orientations and let $\mathbf{p} : [0, 1] \rightarrow \mathcal{C}$ denote the path segment, then collision detection asks whether there is any $t \in [0, 1]$ such that $\mathcal{R}(\mathbf{p}(t)) \cap \mathcal{O}$ is non-empty.

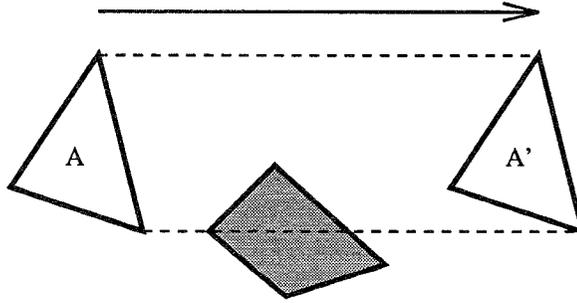


Figure 1: Between query times, the triangle moves linearly from A to A' . Because there is no interference with the obstacle at the query times, simple interference checking fails to detect the collision with the obstacle. Basic distance queries at the query points would indicate that it might be possible that there is a collision, and further computations could be done to decide.

Obviously, interference detection is equivalent to detecting whether or not the minimum distance is zero (or less, if negative distances can be computed.) Since point classification relative to an induced C-object reduces to interference detection, point-classification relative to the union of induced C-objects does also. It follows that point-classification relative to an ordinary compound C-object reduces to point classification relative to pure C-objects and interference detection. We note that distance computation for ordinary compound C-objects is much more difficult.

Thresholded interference checking means determining whether or not the minimum distance is equal to or below a threshold distance. *Witness-collecting thresholded interference checking* means collecting all pairs of features that are closer together than the threshold distance; this is query 7 described above. With the threshold set to zero it is sufficient for answering the extended point-classification query relative to induced C-objects. Extended point-classification relative to an ordinary compound C-object reduces to this query and extended point-classification relative to pure C-space objects.

Simple use of interference detection can obviously miss collisions. (See Figure 1.) Thresholded interference checking can prevent this, but at the risk of detecting false collisions. With distance computations alone, robust collision detection can be done with binary searches in time. However, it is probably more efficient to use witness-collecting thresholded interference detection to collect a small set of candidate collision-pairs before the search step gets too small, and then use the binary search technique on the individual pairs. Alternatively, we can apply the methods of [Can86, Cam90] to them. (After the conclusion of the C-Space Toolkit project, we developed an efficient technique that can be used in place of these when the collision interval is not required. See [Xav97].) Since collisions in C-space correspond exactly to collisions in the world, what we have described suffices for collision detection in C-space.

2.3.2 Previous and Related Work

We review some of the extensive body of work on interference detection and distance computation between two arbitrary objects that move rigidly. An overview covering a more complete set of previous and related work (from three research communities) can be obtained by consulting [GLM96, Hub96, HKM95]. A related and important problem concerns collision detection among members of a collection of objects that move arbitrarily but continuously with respect to each other. For example, see [Bar90, LMC94, MC95, CLMP95].

For a number of years, hierarchical geometric representations have been used to avoid *all pairs comparisons* in interference detection, distance computation, and collision detection. The binary-space partitioning tree (BSP-tree) [FKN80, Nay92] and variants (e.g., [Van91]) have been successfully used in exact interference checking and contact determination, but they do not readily yield distance information. The octree [JT80, Mea82] is another space-partitioning data-structure that has been used in interference detection (for example, [ANFN88]), but it must be used with other geometric objects to obtain computations matching the accuracy of faceted boundary representations or BSP-trees.

Other hierarchical structures use various primitives to bound the object or its surface at each level of the hierarchy, although sometimes holes are treated explicitly, as in [Fav89, CLMP95]. Successful implementations for exact interference detection and distance computation have been based on several geometric primitives. The convex-hull based implementations of [CLMP95, Cam96b, Cam96a, SHMA96], the sphere-based techniques of [Qui94], and bounding-prism methods of [GLM96] are among those known to be fast at the current time. We note that most of these had not been reported halfway into the CSTk project.

Much work has also been done on optimizing the minimum-distance computation between primitives. The convex-hull distance algorithm of [GJK88], referred to as “Gilbert’s Algorithm” is used by many of the systems mentioned above, as well as by ours. [CL91, Cam96a] have presented fast incremental convex-hull distance schemes, which exhibit constant-time complexity when the object configurations change only by small amounts between queries. There is potential to adapt these techniques into our system.

[Can86] describes a technique for exact collision detection for convex polyhedra moving along path segments linear in translation and the quaternion representation of $SO(3)$. Finally, [Cam90] uses space-time techniques and develops algorithms and hierarchical structures for exact collision detection for non-convex objects.

2.3.3 Basic Hierarchical Distance Calculation

We represent the boundary (closed surface) of a body with a bounding-volume hierarchy, which we generate with the algorithm described in §3. The hierarchy is a binary tree in which the subtree rooted at a node, or *body*, represents the union of the primitive geometric objects at its leaves. Each node of our hierarchical geometric representation contains a *conservative approximation*, or *wrapper*, of the object represented by its subtree. In the current C-Space Toolkit Kernel implementation, our trees contain a convex hull (convex polyhedron) at each interior node, and convex polygon or polyhedron at each leaf. We chose to use convex polyhedra as the bounding volume primitive because it is relatively easy to convert from them to various other bounding volumes. (This choice also eventually allowed an extension to swept r-objects after the conclusion of the project.)

We use one basic algorithm to perform distance and witness-collecting thresholded interference detection, with the latter extended to also compute the minimum distance. To do this, we use a recursive divide-and-conquer, branch-and-bound algorithm. We now describe a simplified version that finds the minimum distance and collects all pairs of primitive geometric features (convex polygons and convex polyhedra) closer than the threshold but does not identify the minimum-distance witness pair when there are multiple witnesses.

The algorithm starts at the top nodes of both bodies with the minimum known distance *dist* (the bound) set at infinity. If both nodes are leaves, then the distance between the primitives at these leaves is computed. If it is smaller than the *hitDist* threshold, then the pair of primitives is added to the list of witnesses. If it is smaller than the current minimum distance *dist*, *dist* is set to this value. *dist* is returned to the caller.

If either of the two current nodes is not a leaf, then the distance between the convex hulls of the nodes is compared with *dist* and *hitDist*. If the hull distance is greater than both, *dist* is returned to the caller. Otherwise, a pair of recursive calls (the branches) is made with the one of the current nodes and each of the children of the other as arguments. If both current nodes are not leaves, then the algorithm uses the children of the one with the greater principal axis length. The greater of the current and threshold distances is passed as the threshold distance arguments.

Pseudo-code for the algorithm, with the recursion implemented via a stack, is shown in Figure 2. In the current implementation, both *hullDist(..)* and *primDist(..)* compute the distance between two

Multi-Use Algorithm

```
real cstkDist(body *R, body *S), real distThresh, real hitThresh, pairSet, &witnesses)
{
  real dist ← distThresh;
  pairStack stack;
  body *b1,*b2;
  stack.push(R,S);
  while (!stack.isEmpty()) {
    stack.pop(&b1,&b2);
    if (isLeaf(b1) ∧ isLeaf(b2)){
      if (primDist(b1,b2) ≤ hitThresh)
        witnesses.add(b1,b2);
      dist ← min(dist,primDist(b1,b2));
    }
    else if (hullDist(b1,b2) > max(hitThresh,dist)) // *
      continue;
    else if (isLeaf(b1) ∨ (!isLeaf(b2) ∧ len(b1) < len(b2))) {
      stack.push(b1, b2→child1);
      stack.push(b1, b2→child2);
    } else {
      stack.push(b1→child1, b2);
      stack.push(b1→child2, b2);
    }
  }
  return dist;
}
```

Figure 2: This pseudo-code gives the simple basic algorithm, using a stack to implement the recursion.

convex polygons or polyhedra are efficiently done by our implementation of Gilbert's Algorithm [GJK88].

2.3.4 Implementation

The algorithm sketched above was implemented both in the our prototype Common LISP and current C++ code. We use the Quickhull code [BDH] from the University of Minnesota for computing convex hulls. We began our implementation on a Sun SPARCStation 10/51; we are currently developing the code on an SGI Indigo2 R4400/250.

During the C-Space Toolkit project, we implemented two main optimizations, which we briefly describe below.

The first optimization allows the Toolkit to re-use memory and computational results for queries involving r-objects (movable bodies). For each r-object \mathcal{R} in a query at configuration \mathbf{x} (including r-objects that are used as nominal s-objects in defining C-objects), the Multi-Use Algorithm requires some subset of $\mathcal{R}(\mathbf{x})$ that is often substantially smaller than the whole. In our optimization, each node of $\mathcal{R}(\mathbf{x})$ is computed lazily as needed and is associated with a node of \mathcal{R} . On subsequent queries, memory occupied by nodes of $\mathcal{R}(\mathbf{x})$ is re-used, and the use of configuration timestamps allows the system to avoid re-computation when possible.

The second optimization improves the performance for distance queries. In this optimization, for each pair of bodies $(\mathcal{R}, \mathcal{S})$ that defines a simple induced C-object, we cache the pair of nodes $(\mathcal{R}_w, \mathcal{S}_w)$ that are witnesses to the last distance computed for the body pair. The next time the body pair $(\mathcal{R}, \mathcal{S})$ is needed to answer a query, the distance between \mathcal{R}_w and \mathcal{S}_w at the new configuration is used as the distance threshold *distThresh* by the Multi-Use Algorithm unless a larger distance is given as the argument. We call this optimization *witness caching*.

A minor optimization we used was to limit the maximum number of vertices on convex-hulls to an *a priori* bound, and to use approximately minimal-volume (not axis-aligned) bounding boxes instead of convex-hulls that would violate this limit. Additionally, our implementation of Gilbert’s Algorithm takes an optional *maxdist* argument and can cut short computation if the actual distance exceeds it.

2.3.5 Experiments

We conducted simple experiments comparing the use of convex-hull distance against the use of (axis-aligned-) bounding-box distance in deciding whether to skip the recursions in the Multi-Use Algorithm. (See the line marked with the * comment in Figure 2.) For both versions of the algorithm, we experimented with and without witness caching.

We tested the algorithms on simple point-classification and distance queries in three different geometric scenarios. In the first scenario (see Figure 3), coded “FL”, a 626-polygon lamp translates collision-free through an environment with five obstacles containing a total of 1526 polygons. In the second (Figure 4), coded “FP”, a 1920-polygon teapot translates through an environment of 5 obstacles with a total of 3020 polygons, with contact occurring at the beginning, in the middle, and at the end of the motion. In the third scenario (Figure 5), coded “GPM”, a gear (3920 polygons) drops through a hole in a flat plate (4054 polygons) and onto the spindle of a motor (2252 polygons). The motor is roughly unit scale, and the translational path is one unit long. The clearance between the gear and the motor spindle is 0.00125 units.

The results are summarized in Table 1. The point-classification experiments are coded “I” (for interference detection), and the distance experiments are coded “D”). There are a total of four different distance-check/optimization combinations: bounding-boxes (“B”), bounding-boxes with witness-caching (“BC”), convex-hulls without witness-caching (“H”), and convex-hulls with witness-caching (“HC”). In all runs, 200 queries were done during the motion. Because the point of the experiments was to get a rough comparison between methods, the times given in the table are simply from typical runs when no disk activity could be heard. The experiments were done on an

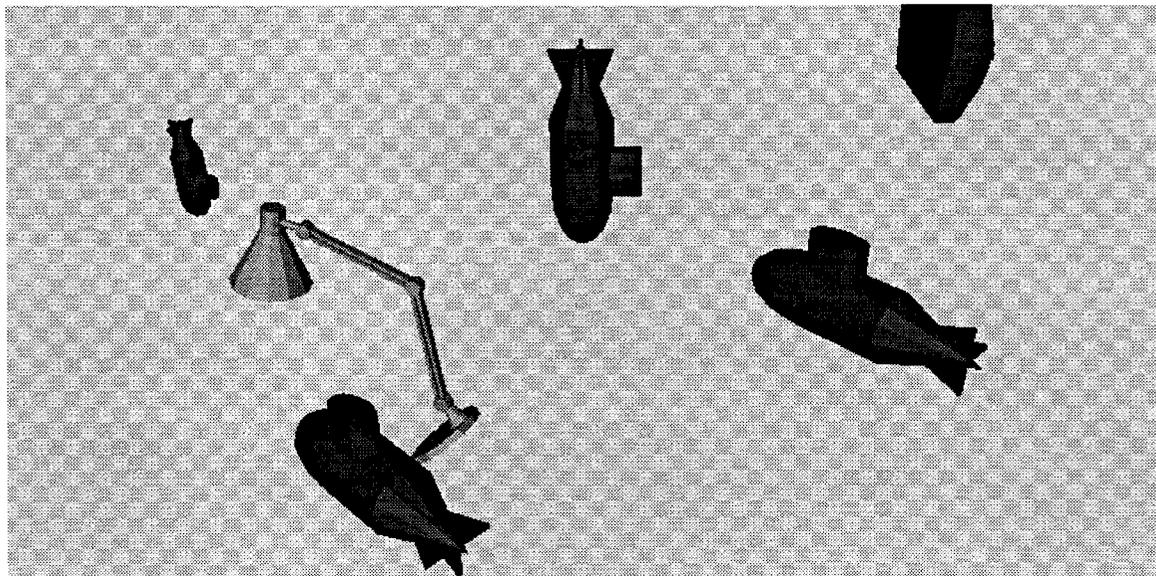
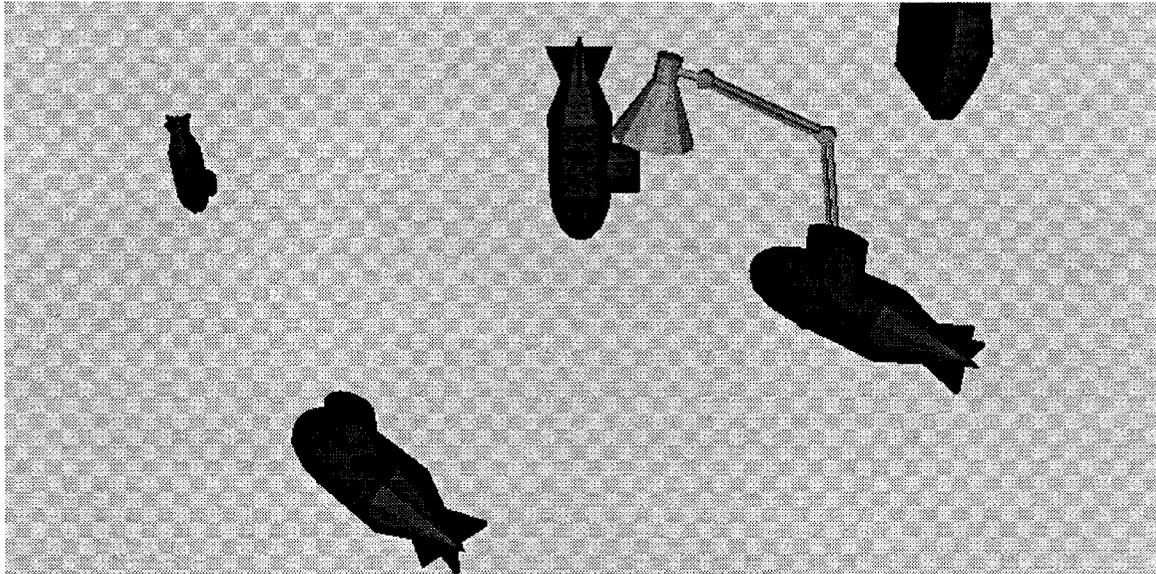


Figure 3: FL, an example typical of gross-motion problems or free-flight. Moving lamp is shown at beginning and end of motion.

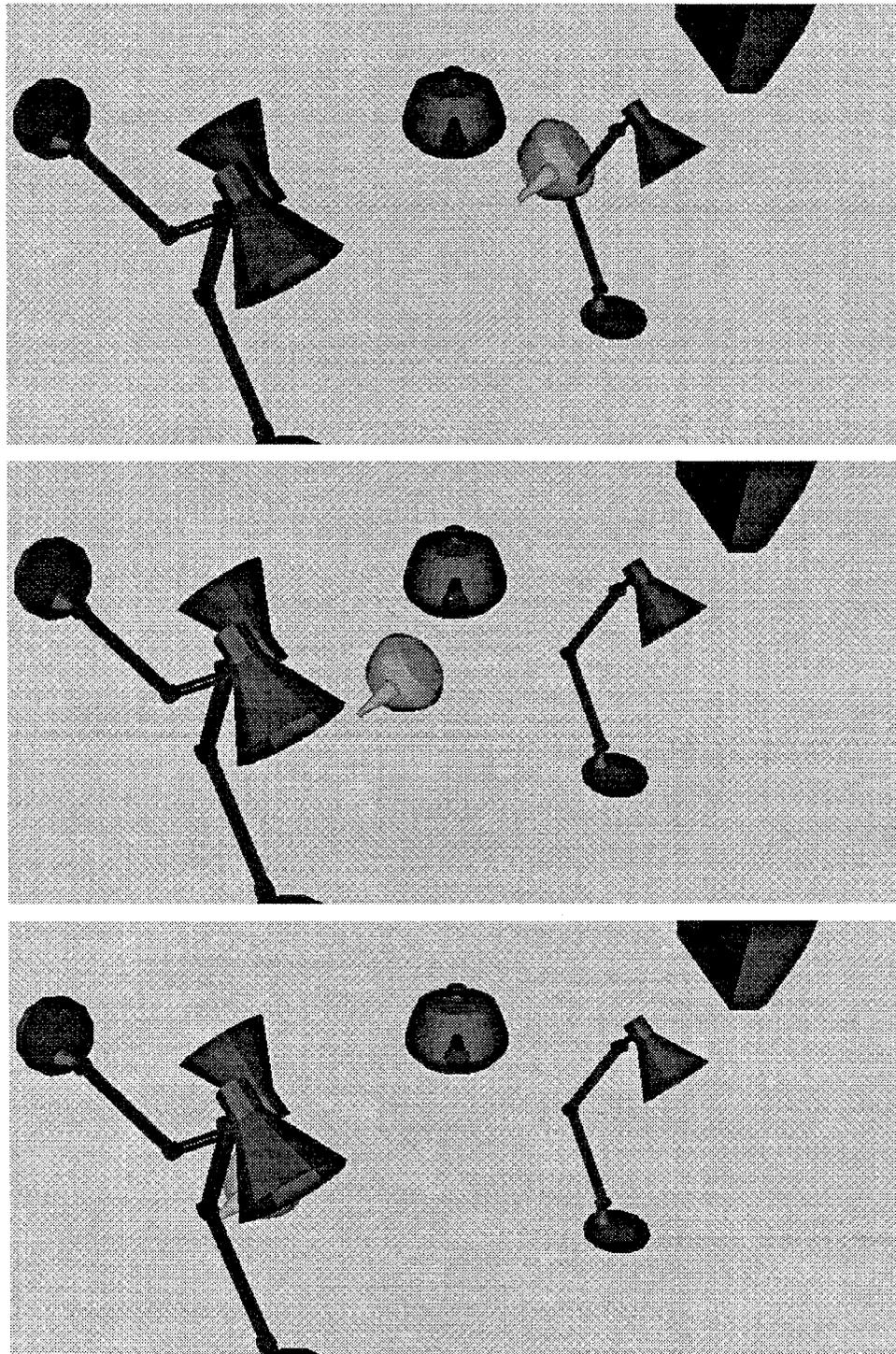


Figure 4: FP, an example in which both contact and large distances occur. Moving teapot is shown at the beginning, middle, and end of its path. It is mostly obscured in the last frame. Note that the teapot and lamp models are used in varying scales.

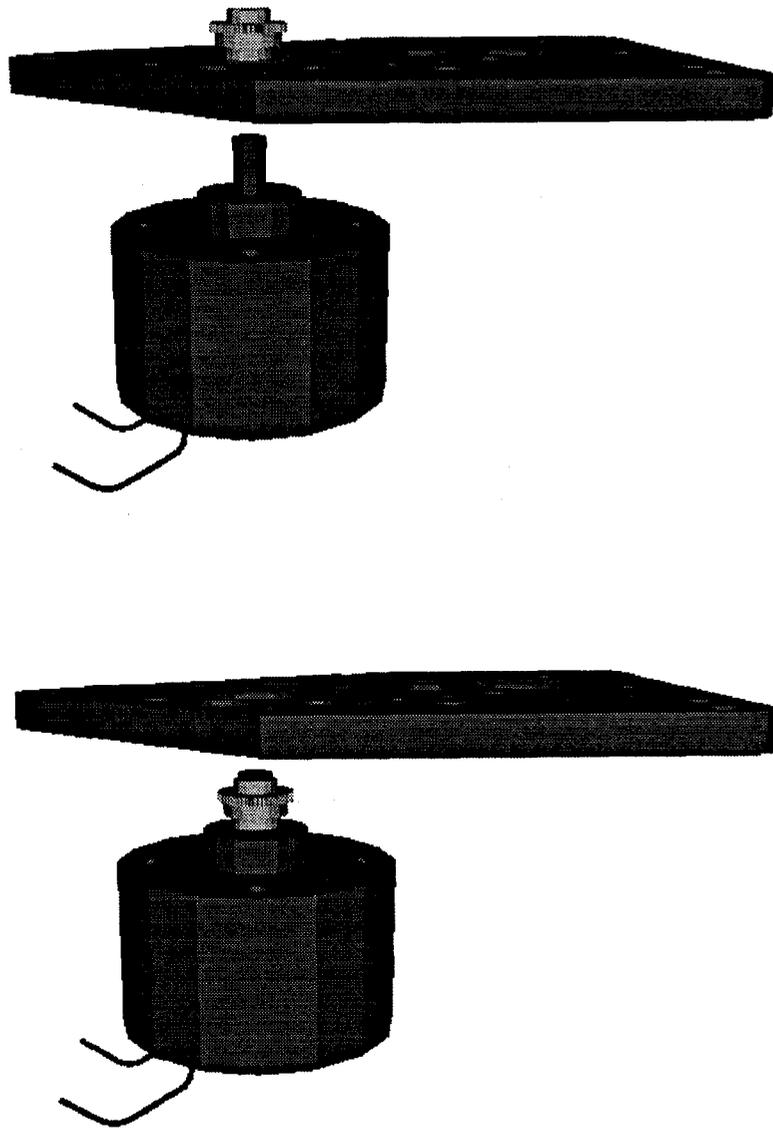


Figure 5: GPM, an example with tight tolerances and semi-pathological difficulty due to concavities and symmetries. Moving gear is shown at the middle and end of the motion.

		Algorithm	B	BC	H	HC
scenario	experiment	<i>hitThresh</i>	time (secs.)			
FL	I	0.1	.22	.22	.37	.40
FL	D		72.6	16.5	11.0	1.1
FP	I	0.01	2.4	2.3	2.2	1.7
FP	D		229.8	17.8	88.7	5.8
GPM	I	0.0001	8.5	8.5	6.0	6.2
GPM	I	0.00128	10.4	10.2	7.6	7.6
GPM	D		316.2	154.9	87.8	29.4

Table 1: Distance computation (D) and point-classification (I, for interference detection). 200 queries were done in each motion sequence. Scenario FL: 626 polygons moving, 1526 stationary; scenario FP: 1920 polygons moving, 3020 stationary; scenario GPM: 3.9K polygons moving, 6.3K polygons stationary.

SGI Indigo2 R4400/250.

We see four main conclusions. First, checking the distance between axis-aligned bounding-boxes of nodes to decide whether to prune recursions (tree-descents) is competitive with the use of convex-hull distance for interference detection (point classification). Second, in distance computation the convex-hull check greatly outperforms the axis-aligned bounding-box check. In our tests, the improvement ranged from a factor of 2.6 (FP/D/H vs. FP/D/B) to 15 (FL/D/HC vs. FP/D/BC). Third, the witness cache does not have much effect on interference detection. Finally, the witness-cache improves distance computation performance for both the bounding-box and convex-hull versions. Speed-up ranged from a factor of 2 (GPM/D/B/C vs. GPM/D/B) to a factor of 15.3 (FP/D/HC vs. FP/D/H).

We note that the GPM examples are computationally intensive for two reasons. Not only are clearances small in the hole and on the spindle, but in both cases, the geometric symmetry means that there are many pairs of polygons approximately the minimal distance (0.00125) apart. This is a semi-pathological but not uncommon occurrence. While the gear moves onto the spindle in the last 20-25 percent of the motion, there are typically 500 pairs of polygons that are about the same, minimal distance apart, and distance calculation slows to less than 10Hz in this segment of the path.

2.4 Conclusions and Further Work

We have described the reasoning that guided the architecture we chose for the C-Space Toolkit. The implementation of the C-Space Toolkit Kernel not only has immediate use in motion planning and simulation, but lays the foundation for further research and development using the C-space abstraction. The key operation in the kernel is extended point-classification, which maps to interference detection between models in the world. We sketched a hierarchical algorithm for this problem that also performs distance computation. We presented experimental results showing that convex-hull-based hierarchies are superior to axis-aligned, bounding-box hierarchies in distance computation

and at least equal in interference detection when bodies are close.

We note that after the conclusion of the C-Space Toolkit project, we made further advances in distance computation and interference detection by building on our implemented techniques. Key results have sped up these computations by roughly a factor of two and extended them to cover linear-translationally swept bodies exactly and rotationally swept bodies approximately. Possible future work involves implementing an incrementally constant-time version of Gilbert's algorithm for convex-hull distance and extending our witness-caching scheme to associate different witnesses pairs with different regions of C-space. A version of this latter technique was tried in our Common LISP prototype with some success.

3 A Generic Algorithm for Constructing Hierarchical Representations of Geometric Objects

3.1 Introduction

3.1.1 Overview

For a number of years, the robotics community has utilized hierarchical geometric representations to avoid *all pairs comparisons* in distance computations and collision detection, with motion planning being the main application. (For example, see [FT87, Qui94].) The automatic generation of hierarchical geometric representations, however, is still a research area, especially if one wants to control properties such as spatial balance, structural balance, coherence, and tightness.

We present a practical generic algorithm for generating hierarchical representations.³ Our algorithm is generic in that it can be applied to collections of any convex primitive, e.g., a scene of composed of convex polygons or polyhedra. The core of the algorithm is a greedy bottom-up clustering technique that constructs a tree of sets. Tuning parameters allow control of the balance of desirable properties. A hierarchy of convex solids is then constructed by walking the tree. Our algorithm is practical in that it runs in time roughly $O(N \log N)$. Our preliminary C++ implementation takes less than four minutes to construct hierarchical groupings from inputs of over 10,000 polygons when there is no polygon adjacency information, and less than a minute when this information can be recovered via vertex coordinates.⁴ Building a hierarchy of solids then takes seconds to minutes, depending on the solids and the desired tightness of the geometrical approximation.

3.1.2 Context

Geometric query algorithms (e.g., distance) frequently take tree-structured representations as input. A typical representation is a tree whose nodes each contain a *geometric primitive*, such as a sphere, convex polygon, or convex polyhedron. The subtree rooted at a node represents the union of the

³An earlier version of this section has appeared as [Xav96] but did not include the results presented in and related to §3.2.5.

⁴These timings were done on a SGI Indigo2 R4400/250.

primitives at its leaves. We require that each node of a hierarchical geometric representation contain a conservative approximation, or *wrapper*, of the object represented by its subtree.

A typical branch-and-bound algorithm to answer a query about the geometric object represented by such a tree does a partial traversal, making a local query about the wrapper or primitive at each node it visits. Thus, a optimal tree would minimize

$$\sum_{\alpha \in \{\text{nodes}\}} c_l(\alpha)p(\alpha), \quad (1)$$

where $c_l(\alpha)$ is the cost of answering the local query to α and $p(\alpha)$ is the probability that a local query at α will be made in answering a global query. The catch is that even the relative probabilities are generally not known.

There are, however, several applicable heuristic properties. First, a *structurally balanced* tree is suited for the divide-and-conquer nature of distance computation. Second, it is intuitive that the probability that a node is “relevant” to a query would grow with its volume. This leads to the principle that a tree should be *spatially balanced*. Third, we desire *spatially coherent* trees, in which represented geometric objects that are spatially close are also structurally close. Not only are spatially coherent trees intuitively good according to the expression above, but because they agree with human intuition, algorithms that make use of them are more likely to perform as expected.

3.1.3 Our Results

We present a generic algorithm for the following problem. Given a collection of N geometric primitives for which convex wrappers can be efficiently computed, construct a hierarchy of convex geometric primitives such that (a) its leaves cover the input primitives, and (b) each node contains a wrapper for the objects at the leaves of the subtree rooted at it. We assume that the diameter of the largest primitive in the input is at most N orders of magnitude greater than that of the smallest.

Our approach has two parts. First, in the *Grouping Phase*, we construct a binary tree of sets, with edges corresponding to the subset relationship. This tree is called the *Subset Tree*. Second, in the *Geometrical Construction Phase*, we construct a structurally identical *Wrapper Tree*, in which each node contains a convex wrapper that spatially covers the union of the set elements at the corresponding node in the Subset Tree. Each Wrapper Tree leaf contains the geometric primitive from the corresponding Subset Tree leaf.

We first describe a generic algorithm for the Grouping Phase. This algorithm represents the spatial extent of a set of geometric primitives with a convex *p-set wrapper*, which can be any solid geometric primitive, and uses a heuristic clustering technique to pair up nodes to be siblings. Our description treats creating and merging of p-set wrappers as bottom-level operations. Parameters to a cost function can be used to control the relative emphases on structural balance, spatial balance, and spatial coherence. We present an informal analysis for the case in which the p-set wrappers are axis-aligned bounding-boxes. The analysis yields a worst case running time is $O(N^2 \log N)$, with an $O(N \log N)$ expected case under certain assumptions. We also show that using convex hulls for p-set wrappers yields an $O(N^2 \log^2 N)$ worst case, with $O(N^2 \log N)$ expected.

Recall that any convex primitive can be used for the wrapper type in constructing the Wrapper Tree. We describe three straightforward techniques for the Construction Phase: an $O(N^2 \log N)$ method using convex hulls as primitives, an $O(N)$ bounding sphere technique, and an $O(N \log N)$ bounding sphere technique that produces much tighter hierarchies.

We have implemented the grouping algorithm and both the sphere and convex-hull Wrapper Tree constructions in C++. A 12,000-polygon scene takes less than four minutes to process from scratch if the input primitives are not too packed. If the ratio of the number of polygons to the number of connected components of polygons, with the connectivity relationship defined by adjacency, is large enough, then this can be done in less than a minute. This time can be further cut by an order of magnitude if instead of starting from scratch (1) component objects are pre-grouped and processed separately first, and (2) repeated component groups are processed separately and then copy-moved. Constructing the Wrapper Tree from the Subset Tree takes seconds to minutes, depending on the wrapper-type used. We provide illustrative examples.

3.1.4 Previous and Related Work

There is much previous work in hierarchical representations of geometrical objects. A good starting point reference on the use of hierarchical representations in collision avoidance is Faverjon [FT87, Fav89]. Work on the construction of the hierarchical representations falls into three categories: progressive refinement hierarchies, space subdivision, and constructive solid geometry (CSG). While CSG [Req80] is not related to our work, the other two are.

Notable early work in progressive refinement hierarchies includes that of Clark [Cla76], who describes no constructions but introduces a recursive-descent algorithmic paradigm and other important concepts. Rubin and Whitted [RW80] suggest a sort of clustering for automating the hierarchy construction from primitives. Weghorst, Hooper, and Greenberg [WHG84] describe an optimality measure for these hierarchies for ray-tracing, but require construction by the user. Goldsmith and Salmon [GS87] present an implemented method for automatically generating hierarchies of convex bounding volumes. Their algorithm is similar to a “top-down” version of ours, but it has less control over the structural and spatial balance and depends on randomization for protection against “badly ordered data”. One motivation for the presented work was to ensure desirable tree properties.

To speed collision detection, Quinlan [Qui94] constructs hierarchies of spheres using a divide-and-conquer method off-line. Earlier work by DelPobil, *et al* [dPSL92] constructs a representation with a given number of spheres, but restricts the input. Hubbard [Hub94] directly optimizes the tightness of the approximate representation at each level and includes bounds on the distance between each sphere and the object. His use of a medial-axis computation slows the method considerably, but it appears useful for off-line processing.

Related spatial-subdivision algorithms fall into two categories: those based on binary space partitioning (BSP) trees [FKN80] and those based on octrees. BSP trees and variants such as Vanecek’s B-Rep Index have been used effectively in clash detection and modeling [NAT90, Nay92, Van91]. However, these structures have not yet been shown to be efficient for distance computation or the creation of general progressive-refinement hierarchies. Octrees for representing 3D geometric ob-

jects have been in the literature for at least a decade and a half; see [Man88] for a review. Extended octrees (see [BN90b], for example) are more economical and permit exact representations. Octrees have also been used to create initial structures in generating other hierarchies [Hub93]. Non-uniform hierarchical space subdivisions based on balancing the number of primitives on each side of a cutting plane have also been used to create hierarchies, as reported by Zachmann and Felger [ZF95]. Although this type of partitioning is very fast, our previous experiments with a similar algorithm showed that the hierarchical representations were more prone to grouping artifacts than what we present here.

3.2 Building Hierarchies Via A Clustering Technique

3.2.1 Basic Idea: A Hierarchical Clustering Technique

We now describe the Grouping Phase of our algorithm. Recall that the general idea is to use a heuristic clustering technique to build a Subset Tree whose leaves contain the input primitives. Because of the need to handle inputs of size 10^3 to size 10^5 or even 10^6 , we must rule out any algorithm with an expected runtime of $O(N^2)$ or more. While we can't minimize cost (1) because of unknown distributions, we still want to have some control over spatial balance, structural balance, and coherence. Intuitively, this points to some sort of bottom-up construction.

Specifically, we had the following basic idea:

- Maintain a collection of nodes without parents.
- Until this collection contains just one node (the root of the Subset Tree), do the following: consider the convex hulls of the contents of pairs of nodes, and make a parent for the pair whose convex-hull diameter is minimal.

We call making two nodes the children of a new node *merging* them. This also entails conceptually combining the sets that correspond to the node and creating a new p-set wrapper for the resulting set.

One problem with this approach is it appears to be at least $O(N^2 \log N)$ because of the number of node pairs that need to be considered. In addition, it lacks control over coherence; it ignores the heuristic that it is qualitatively better to merge two nodes if they are spatially contiguous or close than if they are slightly smaller but not contiguous.

Our solution to both problems is based on the observation that the diameter of new parent grows monotonically. Let us assume that the primitives are not long and skinny (or flat) and not closely packed like uncooked spaghetti. Then, suppose we set a diameter limit, and only consider pairs of nodes with p-set wrappers that have smaller diameters and centers that are closer than this diameter; when the queue is empty, we simply double the diameter limit. If the limit is set appropriately, this greatly cuts the number of nodes any given node could merge with. In fact, if we bound the aspect ratios of the primitives, then each node can merge with at most a constant number of others. This cuts the length of the queue of pairs of parent-less nodes to $O(N)$. Furthermore, by spatially hashing the parent-less nodes, we make constant the time it takes to determine which nodes another node can merge with.

3.2.2 The Clustering Algorithm

There are other factors we might want to consider in determining the priority of merging a pair of nodes. We define the *Balance* of a node to be the greater ratio of diameters of its children. We define the *Fill* of a node to be the ratio of its diameter to the sum of the diameters of its children. These measures can be used in assigning the *Cost* of merging a pair of nodes. The cost is associated with the p-set wrapper of the would-be new node. We have been experimenting with a *Cost* of the form:

$$Cost = diameter^A \times (B \times Fill + C \times Balance), \quad (2)$$

where A , B , and C , are non-negative constants. Roughly, increasing A improves spatial balance of the hierarchy. B and C are used to tune the emphases on spatial coherence and structural balance.

To speed the algorithm, instead of using convex hulls for the geometric extent of nodes and L_2 distance to measure gaps, we can use axis-aligned bounding boxes and L_∞ distance. We will use the term *body* to refer to a node, its p-set wrapper, and the set contained by that node. We now sketch the algorithm.

Basic Clustering Algorithm

1. Let $GLimit$ be the maximum gap allowed between two bodies to be merged. This must be positive if the data may contain more than one connected component. Let $DLimit$ be the maximum diameter of bodies eligible for merging.
2. If there is only one parent-less body, return it; it is the root of the Subset Tree. Otherwise, let the set of active bodies $ABods$ be the set of parent-less bodies that obey $DLimit$. Let the rest of the parent-less bodies form the set of dormant bodies $DBods$.
3. Let $CSize = DLimit + GLimit$. Subdivide the bounding box of the input into voxels of edge-length $CSize$ to create a body-center occupancy array. Then scan in the centers of the active bodies to determine possible eligible pairs, and use the gap limit $GLimit$ to filter these to obtain the queue of eligible pairs $MPairs$ sorted by increasing $Cost$.
4. Until $MPairs$ is empty do the following:
 - (a) Pop off the first pair in $MPairs$, and remove other pairs that share an element with this pair. Create a merged body from this pair.
 - (b) If this body meets the $DLimit$ criterion, then add it to the set $ABods$ and determine which other members of $ABods$ it is eligible to merge with, using and updating the array from step 3. Enqueue the new eligible pairs in $MPairs$.
 - (c) Otherwise, add the new body to $DBods$.
5. Double $DLimit$ and $GLimit$. Go to 2.

3.2.3 Subset Tree Analysis

We now present an informal analysis of the algorithm.

Since the hierarchy is a binary tree with N leaves, exactly $N - 1$ interior nodes are created, and at most N bodies (nodes) are active at any time. Let us assume that during the construction no body is eligible to merge with more than k other bodies for some constant k that depends on how “pathological” the input is. We justify the assumption by considering the case in which input is uniformly sized and uniformly distributed and in which (2) is simplified to be *diameter*. The key observation is that because *DLimit* doubles in step 4 and the ordering of *MPairs* favors the creation of roughly cuboid bodies, doubling the *DLimit* and *GLimit* in step four does not, on average, increase the number of bodies any one body is eligible to merge with.

The assumption justified, it follows that the maximum length of the queue *MPairs* is kN , and at most kN queue operations are performed. These operations take total time $O(kN(\log k + \log N))$ time. Furthermore, consulting the array of lists in steps 3 and 4(b) also takes a constant amount of time dependent on k for each body. We note that the array can be implemented virtually with a spatial hash table, using array coordinates as keys and cell contents as data; thus, the storage for the array only really need be size $O(N)$. The other operations are constructing the geometric extents of the bodies, merging them, and determining the diameters and gaps. Assuming that we use axis-aligned bounding boxes, each takes constant time. Since all other individual operations are constant time, the total running time is then $O(kN(\log k + \log N))$, or $O(N \log N)$ for a given quality of input. In the worst case, with “pathological” inputs such as a box of spaghetti, the constant k is replaced by N . Thus the worst case running time is $O(N^2 \log N)$. Finally, ordering of *MPairs* also balances the tree to be of $O(\log N)$ depth, with the constant depending on the *Cost* function and the quality of the input; the assumption about the ratio between greatest and least diameters among the input primitives also comes into play.

We note that if convex hulls are used to model geometric extent, then each merge takes time $O(n \log n)$ for a pair with $O(n)$ vertices. However, if we assume the input primitives each have a constant-bounded number of vertices, then the overall running time would be $O(N \log^2 N)$ because of the balance of the tree.

3.2.4 Building a Geometric Hierarchy

To construct the Wrapper Tree for the output of our algorithm, we do a depth-first traversal of the Subset Tree, assigning a wrapper to each node once we’re done with its descendents. There are two ways we can assign a wrapper to a parent node. The first option is to geometrically merge the wrappers at its children. The second is to create a wrapper that geometrically covers the primitives at the leaves of its subtrees — i.e., the members of the set at the corresponding node in the Subset Tree. If we use convex hulls for the wrapper type, then these options are geometrically identical, costing $O(N \log^2 N)$ overall. However, if we use bounding spheres, bounding boxes, bounding ellipsoids, or any other sort of convex object for wrappers, then the second option results in geometric hierarchies that cover the collection of input primitive much more tightly.

The apparent advantage of the first option is that for bounding spheres, bounding boxes, and

bounding ellipsoids, the cost of merging a pair is constant, so that the overall time for this phase would be $O(N)$. However, for axis-aligned bounding boxes, the constructions are identical. In addition, we note that computing an approximately minimum bounding sphere or rectangular prism for n points takes time $O(n)$ [Wu92]. Furthermore, a bounding sphere or bounding prism of the vertices of a collection of convex polygons and convex polyhedra bounds the collection itself. Recalling that we assume a constant-bounded number of vertices on each input primitive and that the Subset Tree will be of depth $O(\log N)$, we see that the cost of computing a Wrapper Tree of bounding-spheres or rectangular prisms directly from the corresponding nodes in the Subset Tree is $O(N \log N)$, which is perfectly acceptable since the constant is small and this complexity matches the cost of constructing the Subset Tree with axis-aligned bounding boxes as p-set wrappers.

3.2.5 A Clustering Algorithm Using Connectivity

We now describe a variation of the Clustering Algorithm that greatly improves speed when the ratio of number of polygons to the number of connected components of polygons, with the connectivity relationship defined by adjacency, is significantly large.

The basic idea is again simple. Instead of comparing bounding-box distance to the gap limit in determining whether two bodies are eligible for merging, we check whether they are related by a pair of edge-adjacent polygons. That is, we check whether there is some polygon A in one body and some polygon B in the other such that A has an edge with vertices identical to those of some edge in B . The box-diameter test is still used as a filter in determining eligibility. We observe that the idea generalizes easily from polygons and edge-adjacency to any set of primitive geometric objects for which we can define a geometric adjacency relationship.

We now sketch the algorithm, which we say builds *chunks*, each of which is a maximal connected body. To make the presentation simpler, we will restrict ourselves to the polygon and edge-adjacency case and assume a half-edge data structure — each edge is directed and belongs to exactly one polygon. We will say that a body *owns* all of the edges of the polygons that belong to it. Also, we say that two edges *have vertex v in common* if they each have a vertex with the same coordinates as v . The algorithm starts with each polygon defining a body, like the algorithm in §3.2.2.

Adjacent-Element Clustering Algorithm

1. Construct the set of *edge families* defined by the criteria that edge e_i is in the same edge family as e_j if and only if e_i and e_j have all their vertices in common.
2. For each edge family, create the set of all pairs of different edges in the family, which is the set of *sibling pairs* in that family.
3. Let $DLimit$ be the maximum diameter of bodies eligible for merging.
4. If there is only one parent-less body, return it; it is the single chunk. If there are no edge families with sibling pairs, return the current set of parent-less bodies. Otherwise, let the set of active bodies $ABods$ be the set of parent-less bodies that obey $DLimit$. Let the rest of the parent-less bodies form the set of dormant bodies $DBods$.

5. Let the set of eligible pairs of bodies be the set of bodies such that body-pair $(B_i, B_j) \in MPairs$ if and only if $B_i \neq B_j$ and B_i owns an edge that is a sibling of an edge owned by B_j . Then sort this set by increasing *Cost* to obtain the queue of eligible pairs *MPairs*.
6. Until *MPairs* is empty do the following:
 - (a) Pop off the first pair in *MPairs*, and remove other pairs that share an element with this pair. Create a merged body from this pair.
 - (b) Remove each sibling pair whose members are owned by the new body from its edge family
 - (c) If the new (merged) body meets the *DLimit* criterion, then add it to the set *ABods* and determine which other members of *ABods* it is eligible to merge with, using the sibling-pair criterion in step 4. Enqueue the new eligible pairs in *MPairs*.
 - (d) Otherwise, add the new body to *DBods*.
7. Double *DLimit*. Go to 4.

The complexity analysis is basically the same as for the Basic Clustering Algorithm in §3.2.2. The only difference that must be accounted for is the first two steps. Let N_e be the number of edges and N_g be the number of polygons. Assuming a constant bound on vertex degree, there will be a total of $O(N_e)$ sibling pairs, and the first two steps of the algorithm take time $O(N_e)$ if a hash table is used to detect common vertices. Assuming that N_e is linearly related to the number of polygons, the final complexities of $O(N^2 \log N)$ worst-case and $O(N \log N)$ when there are no pathologies. However, since there is no gap *GLimit* in this algorithm and the only eligible bodies actually are edge adjacent, the constant complexity factor is smaller.

The output of the Adjacent Element Clustering Algorithm can be used as the input for the input to the Basic Clustering Algorithm. The resulting algorithm we refer to as the Simple Hybrid Clustering Algorithm. The obvious caveat is that the chunks might be themselves pathological in shape. For example, in an engine model, one can expect to find a significant number of tubes, routed cables, and pipes. A reasonable approach would be to use the Adjacent Element Clustering Algorithm to pre-process the raw polygon data to obtain a significantly smaller number of bodies for input to the Basic Clustering Algorithm.

3.3 Implementation and Examples

3.3.1 Implementation

We have completed a preliminary implementation of our algorithms in C++. By “preliminary” we mean that we have not yet attempted to optimize our code, e.g., by using *ad hoc* malloc-frugal memory management. For computing bounding spheres we use axis-aligned bounding-box centers. We use the Quickhull [BDH] code from the University of Minnesota for computing convex hulls. We began our implementation on a Sun SPARCStation 10/51; we are currently developing the code on an SGI Indigo2 R4400/250.

3.3.2 Examples

In this section we present some illustrative examples of up to 12,144 polygons. In all cases we used axis-aligned bounding boxes for p-set wrappers when building the Subset Tree.

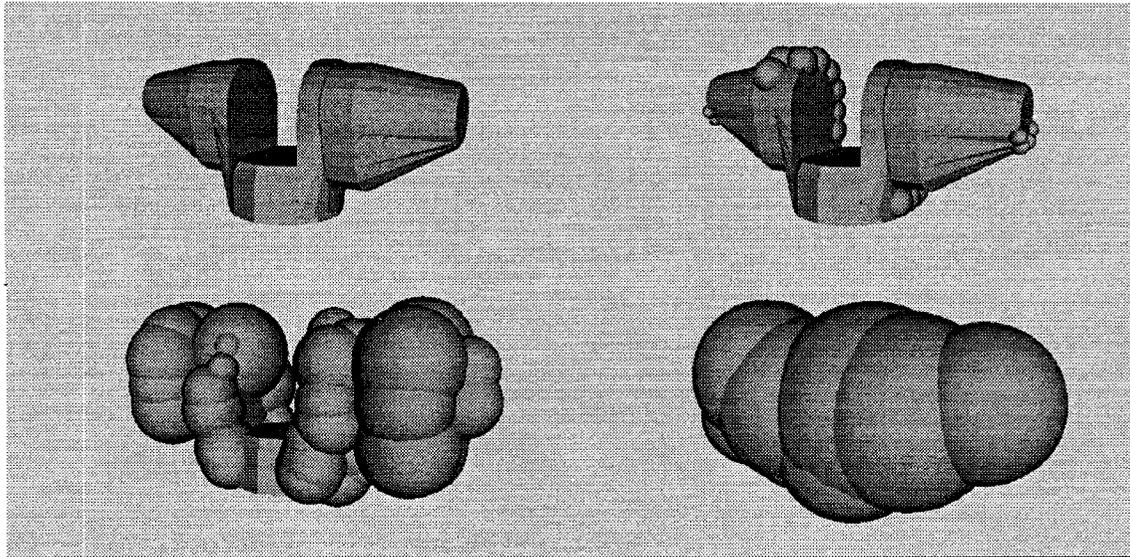


Figure 6: Four levels of the sphere hierarchy constructed from the 276-polygon model of a robot base part.

Figure 6 shows four levels of a tree generated with a 276-polygon robot base model as input, which yields a 12- or 13-level tree. (There is some randomization when ties occur.) The first frame shows the input. The second, third, and fourth frames show different levels of the bounding-sphere Wrapper Tree, progressing to the root. Leaves that are at or below the traversal level are represented as polygons, while interior nodes are represented as spheres. The three frames show representations of 251, 104, and, 8 nodes, respectively. Note that our clustering method does not guarantee a bound on structural balance; however, the implementation has mostly generated trees no more than $2 \log_2 N$ deep.

Figure 7 demonstrates the difference between the Subset Tree and the Wrapper Tree. The first frame shows a 626-polygon model of a lamp. For this input, the implementation produced hierarchies 16 or 17 levels deep. The next three frames again show different tree levels, with 414, 167, and 4 nodes respectively. The last two frames show 167- and 4-node levels of the corresponding convex-hull Wrapper Tree constructed from the same Subset Tree. The quality of the convex-hull Wrapper Tree suggests that our algorithm would produce very good hierarchies of bounding ellipses or rectangular prisms.

Our next example (Figure 8) illustrates a convex-hull Wrapper Tree and shows again that using axis-aligned bounding boxes for p-set wrappers produces a good Subset Tree. The first frame shows a 3130-polygon example generated by replicating and randomly moving the lamp model four times.

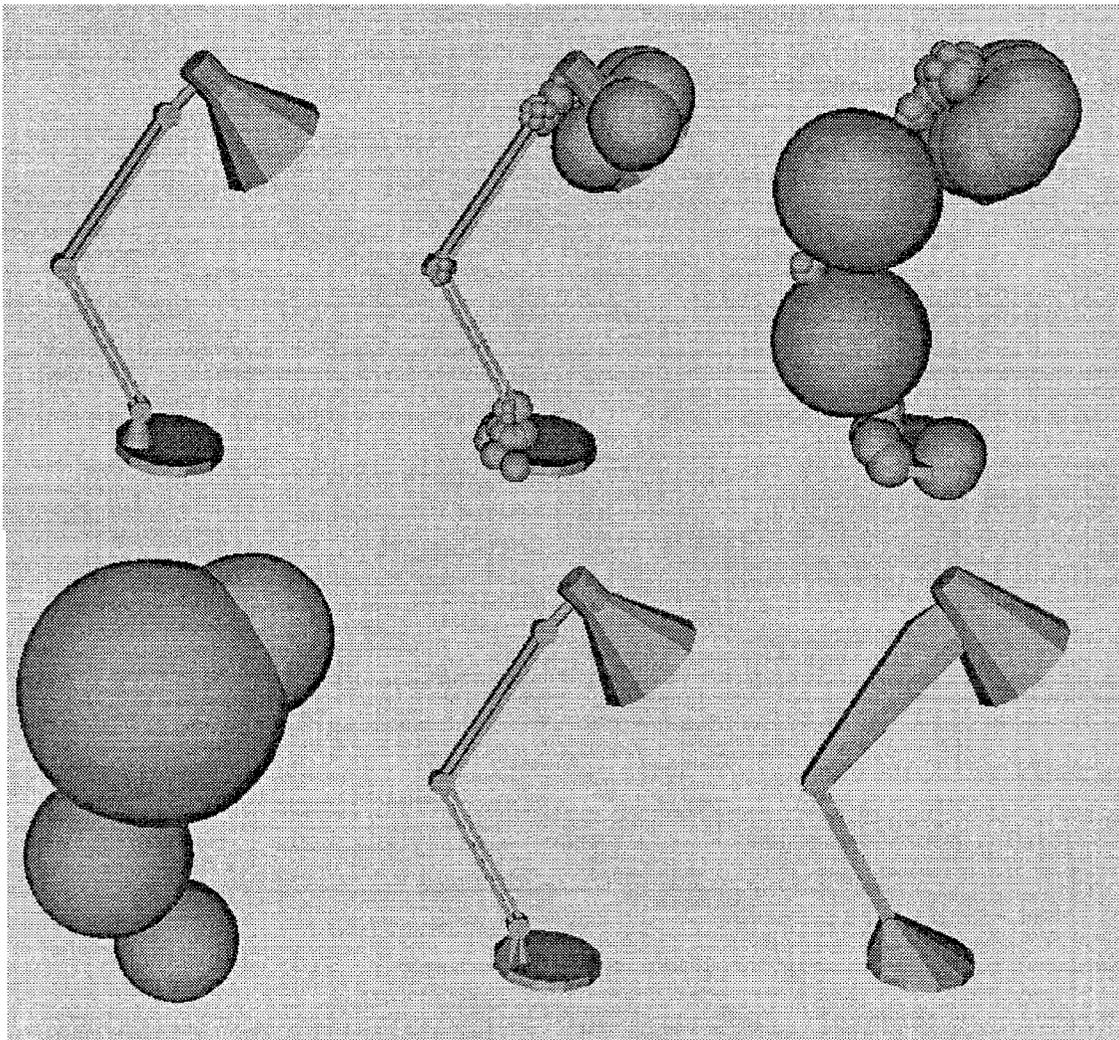


Figure 7: Four levels of the sphere hierarchy constructed from the 626-polygon model of a lamp; two levels from the convex-hull Wrapper tree.

The second frame shows the 8-node level of the 21-level tree produced by our implementation of the Basic Clustering Algorithm (§3.2.2) in 35 seconds. Thus, three levels from the root, eight convex hulls cover the five lamps. Using the Adjacent Element Clustering Algorithm (§3.2.5) to pre-process the input results in a total running time of 12 seconds.

Our final example (Figure 9) was generated by replicating and randomly moving the robot base model 43 times. The input to the algorithms was the resulting 12144-polygon list; this is shown in the first frame. Execution time was about 210 seconds using the Basic Clustering Algorithm and about 50 seconds using the Simple Hybrid Clustering Algorithm, which applies the Adjacent Element and Basic Clustering Algorithms to the input in sequence. (See §3.2.5.) Increasing the average spacing so that fewer objects intersect decreases the running time. The second and third

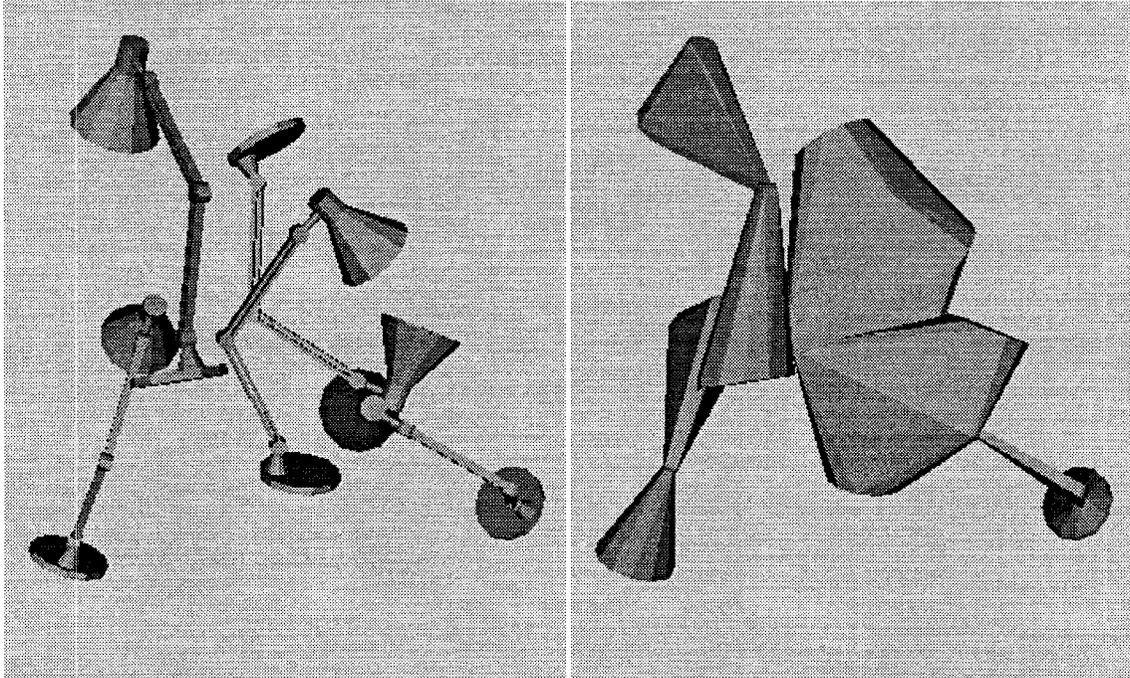


Figure 8: Multiple lamps, 3130 polygons; convex-hull Wrapper Tree level of 8 nodes.

frames show two Wrapper Tree levels of 1604 and 126 nodes, respectively.

Finally, our implementation of the Basic Hybrid Clustering Algorithm took less than 25 minutes to process a 98,000 polygon input⁵ that the Basic Clustering Algorithm could not process alone because of the 128MB memory limit on the test workstation.

3.4 Future Work

There are many directions for future work — in applications, in improving our algorithm, and in combining hierarchical techniques with other methods for distance computation and collision detection.

Towards improving our algorithm, the first question that arises is how to better handle large and flat or long and thin input primitives, such as the polygons that model the surface of the lamp arm. A simple approach would be to pre-process the input to obtain polygons or polyhedra with limited aspect (primary-axis:secondary-axis) ratios, but this is only a partial answer. In addition, we would like to study the structure of the trees produced, in order to improve our control over structural and spatial balance. It would be also useful for the algorithm to compute bounds tightness of the approximation.

Another issue that arises is how to adapt the algorithm or use its output to produce a “near-optimal” collection of wrappers that cover the input, given a limit on the size of this output. We see

⁵The “Stalagmite” example from U. North Carolina, Chapel Hill [GLM96].

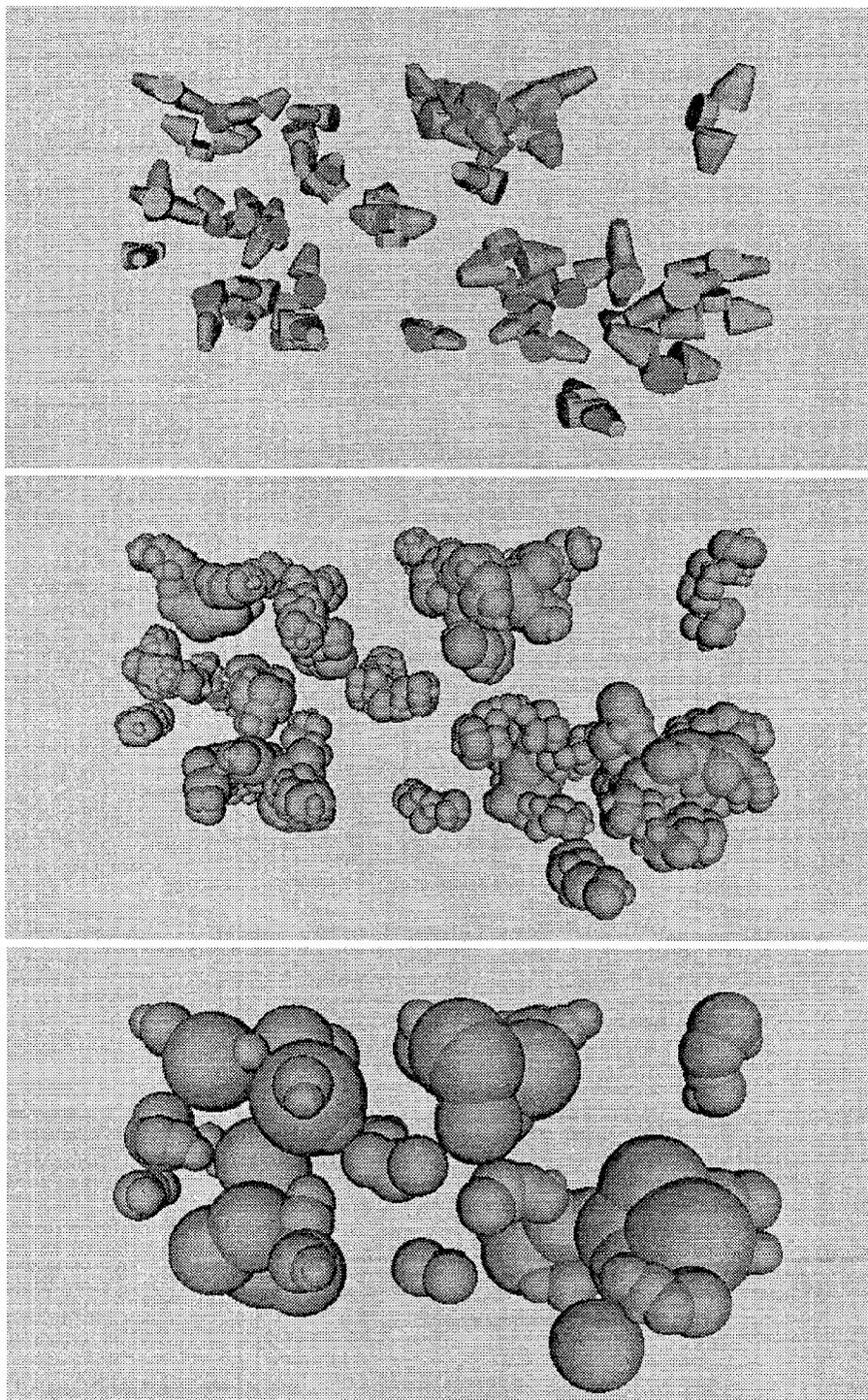


Figure 9: Three levels from the hierarchy constructed from a 12,144-polygon input.

this as a step to combining hierarchical representations with Baraff's [Bar90] and Lin's [LMC94] technique of updating bounding-box relationships. Their "Sweep and Prune" technique is faster than hierarchical techniques for contact-set detection when more than a certain fraction of primitives are close, and it is also advantageous when there are multiple moving bodies or the bodies are not rigid. However, there is the disadvantage that without some hierarchical organization, all the primitives must be updated. (This is somewhat unavoidable when the objects are flexible.) Furthermore, since Canny's and Lin's method [CL91] for computing the distance between collections of polygons that form the boundaries of two convex objects is a clear winner over tree-walking, a complete framework would include an algorithm for lopping off maximum convex boundary sections.

Finally, we would like to pursue a more efficient algorithm. The current algorithm is a suitable tool for on-line use with input size up to 10^3 , and it appears quite fast for off-line use with input sizes into the 10^4 and even 10^5 range. Although models and scenes are often broken up into individual object models each well within this range, in certain applications we encounter raw model data sets that are orders of magnitude larger. We hope to pursue a top-down algorithm based on Goldsmith and Salmon's [GS87] that promises a smaller time-constant than that of our current algorithm. Key extensions are needed to improve control over coherence and spatial and structural balance.

3.5 Conclusions

We have presented a general, bottom-up clustering technique for automatically building hierarchical representations of objects. The first phase of the technique employs a heuristic clustering technique to construct a binary Subset Tree in time $O(N^2 \log N)$ worst-case and $O(N \log N)$ expected. The second phase then uses this hierarchy to construct a Wrapper Tree of geometric objects using any of variety of primitives.

Our preliminary C++ implementation builds a sphere hierarchy from a 12,000-polygon input in less than five minutes. Most of this is spent in the grouping algorithm. Future work includes incorporating the technique into general distance computation and collision-detection schemes and investigating whether Goldsmith and Salmon's faster "top-down" grouping technique can be used while maintaining adequate control over characteristics of the hierarchies.

4 Forward Kinematics, Jacobian, and Contact Normals

This section describes the techniques used by the C-Space Toolkit to compute functions dependent on the kinematics of robotic manipulators. These functions include:

1. the forward kinematics, which relate the static Cartesian position and orientation of the end-effector (or tool) reference frame to the inertial reference frame through the joint parameters;
2. the Jacobian, which specifies the mapping of the joint velocities to the velocities in Cartesian space;

3. and the C-space object (C-object) contact normals, which define the C-space direction corresponding to the fastest rate of change of the distance between the r-object and s-object corresponding to the C-object contact point.

Computation of the first two functions has been discussed in the literature. (See, for example, [Cra89]) However, a thorough understanding of these calculations is necessary to understand computation of the third, which is the main point of this section (see §4.4), and which we did not find adequately fleshed out in the literature. Finally, although our discussion focuses on typical manipulator kinematics, our contact normals implementation generalizes to arbitrary kinematics, as long as local forward kinematic maps and local Jacobian functions are available.

Presentation of these techniques requires a discussion of the reference frames and the frame to frame mappings (§4.1) used by the C-Space Toolkit. Careful selection of the reference frame initial conditions can result in more efficient algorithms for the computation of the forward kinematics (§4.2) and the Jacobian (§4.3).

4.1 Reference Frames and Transformations

Because the location and orientation of objects in the workspace is so important to the use of the C-Space Toolkit, a thorough understanding of how the reference frames and transformations are modeled is essential. For example, because a simple robotic manipulator is a series of connected bodies, the computation of the location and orientation of the end effector reference frame requires modeling intermediate reference frames. Their default orientation is dependent on the type of connection. For all reference frames, the system used is right-handed. Thus, $\mathbf{i} \times \mathbf{j} = \mathbf{k}$ where “ \times ” is the vector cross product and $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are unit vectors in the X, Y, and Z directions, respectively.

A robotic manipulator is modeled as a connected series of rigid bodies. Each body is called a *link*, and the location of the connection is called a *joint*. Each link in the robot has an attached reference frame whose initial orientation is dependent on the joint type. The two standard types of joints supported in the C-Space Toolkit are revolute and prismatic (see Figure 11).

As illustrated in Figure 10, the location and orientation of a body is really the location and orientation of an attached reference frame relative to an inertial coordinate system, (X_I, Y_I, Z_I) . The end-effector reference frame is denoted by (X_n, Y_n, Z_n) , and the $Link_1$ reference frame is denoted by (X_1, Y_1, Z_1) . Each reference frame has its own rules for its default configuration, depending on the geometry of the joint.

The inertial reference frame, (X_I, Y_I, Z_I) , is a “fixed in space” coordinate system that usually has its origin at the center of the first joint with the Y axis pointing up and the X and Z axis usually aligned along some relevant features, such as the workcell boundaries. The reference frame for the i^{th} link has its origin at the i^{th} joint. For a revolute joint, the Z axis is along the axis of rotation and the X axis is along the link’s principal axis. For a prismatic joint, the Z axis is along the link’s principal axis, and the X and Y axes are matched to some body feature. Again, Figure 11 illustrates the default configuration for the two types of joints.

To describe the orientation and location of a body requires that the C-Space Toolkit create a mapping between the inertial reference frame and the body. The map has two parts: a rotation

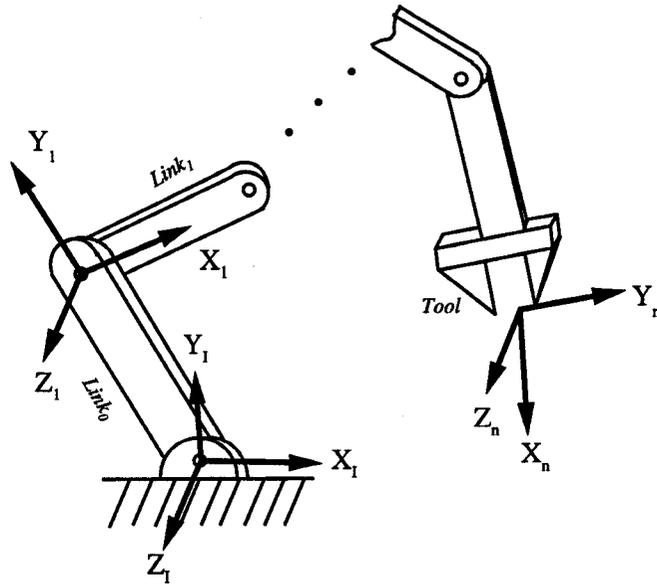


Figure 10: Inertial and End-Effector Reference Frames.

matrix which describes the angular orientation of the reference frame and a 3x1 addend which describes the translation of the origin. This mapping will be equivalent to a homogeneous transform between the two frames.

The rotational matrix is defined in terms of a set of Euler angles (ψ, θ, ϕ) in a yaw-pitch-roll sequence. Figure 12 illustrates the sequence of Euler angle rotations from the inertial reference frame to a body-fixed frame. The first rotation is a yaw of angle ψ about the Y_I axis creating the intermediate (X_1, Y_1, Z_1) reference frame. Next is a pitch of angle θ about the Z_1 axis to create the (X_2, Y_2, Z_2) reference frame. Finally, the (X_B, Y_B, Z_B) reference frame is created by a roll angle ϕ about the X_B axis. Mathematically, this sequence of rotation is modeled as:

$$\begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix} = \begin{pmatrix} \cos \psi & 0 & -\sin \psi \\ 0 & 1 & 0 \\ \sin \psi & 0 & \cos \psi \end{pmatrix} \begin{pmatrix} X_I \\ Y_I \\ Z_I \end{pmatrix} \quad (3)$$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix} \quad (4)$$

$$\begin{pmatrix} X_B \\ Y_B \\ Z_B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & \sin \phi \\ 0 & -\sin \phi & \cos \phi \end{pmatrix} \begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix}. \quad (5)$$

Since most of the pertinent force and moment variables must be computed in inertial coordinates, these three equations are combined to form the body-to-inertial-frame transformation matrix. This allows a body-fixed vector to be transformed to inertial coordinates (rotation only):

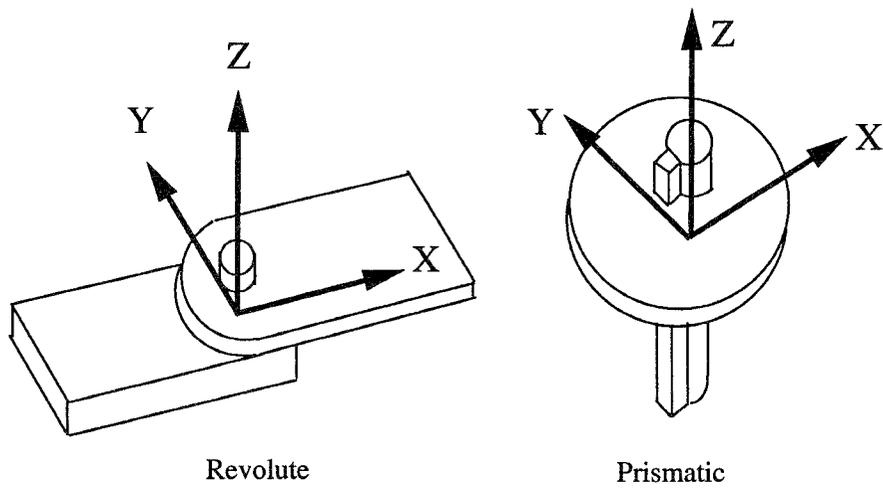


Figure 11: Joint Description.

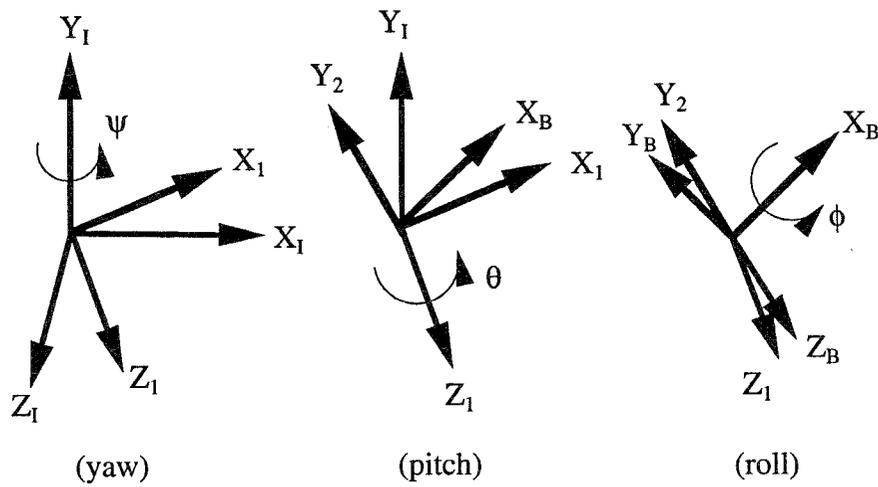


Figure 12: Euler Angle Rotation

$$\begin{pmatrix} X_I \\ Y_I \\ Z_I \end{pmatrix} = \begin{pmatrix} \cos \theta \cos \psi & -\cos \phi \sin \theta \cos \psi + \sin \phi \sin \psi & \sin \phi \sin \theta \cos \psi + \cos \phi \sin \psi \\ \sin \theta & \cos \phi \cos \theta & -\sin \phi \cos \theta \\ -\cos \theta \sin \psi & \cos \phi \sin \theta \sin \psi + \sin \phi \cos \psi & -\sin \phi \sin \theta \sin \psi + \cos \phi \cos \psi \end{pmatrix} \begin{pmatrix} X_B \\ Y_B \\ Z_B \end{pmatrix}. \quad (6)$$

Note that the columns of the matrix are the unit vectors of reference frame B transformed to reference frame I. Equation (6) can be used to find the Euler angles if the transformation is known.

For example, it is useful for finding the inertial Euler angles of an end-effector, since there can be many intermediate reference frames and transformations that complicate the process. (Computing the transformation is discussed later.) If $T_{i,j}$ represents the transformation element in row i and column j , the Euler angles are given by

$$\begin{aligned}\psi &= -\arctan T_{3,1}/T_{1,1} \\ \theta &= \arcsin T_{2,1} \\ \phi &= -\arctan T_{2,3}/T_{2,2}.\end{aligned}\tag{7}$$

Note that there is a singularity when $\cos \theta = 0$.

It is not much more complicated for mappings that allow translation. If the body-to-inertial rotational transformation matrix is denoted as T_B^I , the vectors in the above equation are denoted as \mathbf{x}_I and \mathbf{x}_B respectively, and the vector from the origin of inertial reference frame to the origin of the body reference frame is denoted as \mathbf{p}_I , then a body fixed vector is translated into inertial coordinates by:

$$\mathbf{x}_I = T_B^I \mathbf{x}_B + \mathbf{p}_I.\tag{8}$$

This set of operations lets us express the (local) mapping as a homogeneous transform. The homogeneous transform, \mathcal{T}_B^I , is a 4x4 matrix that allows the right hand side of (8) to simplify. The construct, which is explained further in [Cra89], goes like this.

1. All 3-vectors are cast into 4-vectors by appending a 1 as the last element.
2. The translation of the origin of the body reference frame is added as the last column of the 4 x 4 transformation matrix.
3. A row consisting of (0, 0, 0, 1) is added as the last row of the transformation matrix.

If $\mathbf{0}$ is assumed to be the 3D zero vector, these rules recast (8) into the following form:

$$\begin{bmatrix} \mathbf{x}_I \\ 1 \end{bmatrix} = \begin{bmatrix} T_B^I & \mathbf{p}_I \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_B \\ 1 \end{bmatrix}.\tag{9}$$

The inverse transform can be computed quite easily by solving (8) for \mathbf{x}_B .

$$\begin{bmatrix} \mathbf{x}_B \\ 1 \end{bmatrix} = \begin{bmatrix} T_I^B & -T_I^B \mathbf{p}_I \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_I \\ 1 \end{bmatrix}\tag{10}$$

where the inverse of the rotational transform is equal to its transpose.

$$T_B^I = [T_I^B]^{-1}.\tag{11}$$

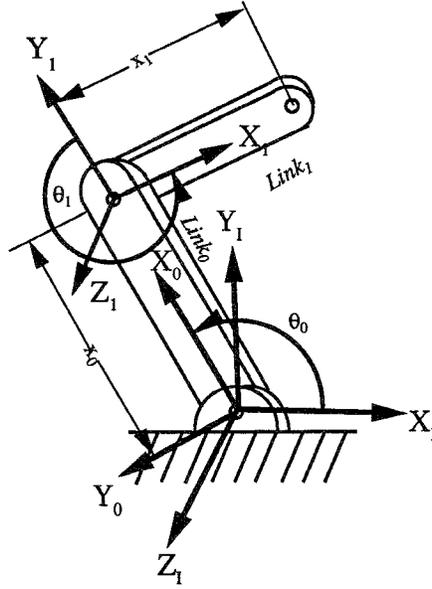


Figure 13: Forward Kinematics

4.2 Forward Kinematics

The purpose of a homogeneous transform construction is to cast the matrix multiply and vector add of an affine transform (8) into a transform that requires just a matrix multiply. The advantage of casting this problem into such a construct is that it allows transforms to be “compounded”, much as (3)-(5) were compounded to produce (6). Therefore, finding the location of the origin of the end-effector reference frame in Figure 10 requires defining intermediate homogeneous transforms \mathcal{T}_i^{i-1} between reference frames i and $(i - 1)$ such that

$$X_{i-1} = [\mathcal{T}_i^{i-1}] X_i. \quad (12)$$

If T_i^{i-1} is the i to $(i - 1)$ rotational transformation matrix and \mathbf{p}_{i-1} is the translation of the origin of the reference frame i with respect to reference frame $i - 1$ on the $i - 1$ link, then

$$\mathcal{T}_i^{i-1} = \left[\begin{array}{c|c} T_i^{i-1} & \mathbf{p}_{i-1} \\ \hline \mathbf{0}^T & 1 \end{array} \right]. \quad (13)$$

The location of the origin of the end-effector reference frame is computed in inertial homogeneous coordinates (\mathcal{X}_I) by compounding the intermediate transforms to form:

$$\mathcal{X}_I = \mathcal{T}_0^I \mathcal{T}_1^0 \cdots \mathcal{T}_{n-1}^{n-2} \left[\begin{array}{c} \mathbf{p}_{i-1} \\ 1 \end{array} \right]. \quad (14)$$

An example using the first two reference frames in Figure 10 will help clarify the concepts. As illustrated in Figure 13, the joint for $Link_0$ is a revolute joint located at the origin of inertial

reference frame. The joint angle, θ_0 , corresponds to an Euler pitch angle of θ_0 with zero yaw and roll angles. The homogeneous transformation matrix, \mathcal{T}_0^I , is defined by:

$$\mathcal{T}_0^I = \begin{bmatrix} \cos \theta_0 & -\sin \theta_0 & 0 & 0 \\ \sin \theta_0 & \cos \theta_0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (15)$$

The joint for *Link*₁ is another revolute joint located at a distance of x_0 from reference frame 0. Again, the joint angle, θ_1 , corresponds to an Euler pitch angle of θ_1 with zero yaw and roll angles. The homogeneous transformation matrix, \mathcal{T}_1^0 is defined by:

$$\mathcal{T}_1^0 = \begin{bmatrix} \cos \theta_1 & -\sin \theta_1 & 0 & x_0 \\ \sin \theta_1 & \cos \theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (16)$$

If the joint for *Link*₂ is located at a distance of x_1 from reference frame 1, then this location can be computed in inertial coordinates by

$$\mathcal{X}_I = \mathcal{T}_1^I \begin{bmatrix} x_1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (17)$$

where

$$\mathcal{T}_1^I = \mathcal{T}_0^I \mathcal{T}_1^0. \quad (18)$$

Because of the construct of the homogeneous transform, the rotational transformation matrices can be compounded in the same manner.

$$T_n^I = T_0^I T_1^0 \dots T_n^{n-1} \quad (19)$$

The Jacobian can be constructed in a similarly recursive manner.

4.3 Jacobian

A fundamental computation in robotics is the relationship between the end-effector velocity and the joint-space (C-space) velocity. This relationship is a local linear map whose matrix is the Jacobian.

For a 3-D manipulator, the configuration of the end-effector is a 6-vector combining the position and angular orientation of the attached reference frame with respect to the inertial frame. For Figure 10, the configuration \mathbf{r} of the end-effector reference frame (X_n, Y_n, Z_n) is given by:

$$\mathbf{r} = \begin{bmatrix} x \\ y \\ z \\ \psi \\ \theta \\ \phi \end{bmatrix}. \quad (20)$$

The nonlinear relationship between \mathbf{r} and Θ , the n -vector of the joint parameters, is usually given by:

$$\mathbf{r} = \mathbf{f}(\Theta). \quad (21)$$

Differentiating (21) with respect to time yields

$$\dot{\mathbf{r}} = \mathbf{J}(\Theta)\dot{\Theta} \quad (22)$$

with the $6 \times n$ Jacobian matrix, $\mathbf{J}(\Theta)$, being defined by:

$$\mathbf{J}(\Theta) = \frac{\delta \mathbf{f}}{\delta \Theta}. \quad (23)$$

Extensive research has showed that it would computationally inefficient to evaluate the closed form solution of the Jacobian, but that it is computationally efficient to rewrite the Jacobian in the following form:

$$\dot{\mathbf{r}}_\omega = \mathbf{J}_\omega(\Theta)\dot{\Theta} \quad (24)$$

where $\dot{\mathbf{r}}_\omega$ is defined by

$$\dot{\mathbf{r}}_\omega = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ p \\ q \\ r \end{bmatrix} \quad (25)$$

and $\omega = (p, q, r)$ represents the angular velocity vector of the end-effector coordinates, i.e., about the X_n , Y_n , and Z_n axes respectively. The relationship between ω and Euler angle velocities can be obtained from Figure 12 and equations (3)–(5). If \mathbf{i}_i , \mathbf{j}_i , \mathbf{k}_i are unit vectors in the X_i , Y_i , and Z_i directions, respectively, then the Euler angle velocity vectors ($\dot{\psi}$, $\dot{\theta}$, $\dot{\phi}$) are defined by:

$$\dot{\psi} = \dot{\psi} \mathbf{j}_I \quad (26)$$

$$\dot{\theta} = \dot{\theta} \mathbf{k}_2 \quad (27)$$

$$\dot{\phi} = \dot{\phi} \mathbf{i}_n. \quad (28)$$

Transferring the yaw and pitch angular velocities to the end-effector reference frame leads to:

$$\dot{\psi} = \dot{\psi}(\sin \theta \mathbf{i}_n + \cos \phi \cos \theta \mathbf{j}_n - \sin \phi \cos \theta \mathbf{k}_n) \quad (29)$$

$$\dot{\theta} = \dot{\theta}(\sin \phi \mathbf{j}_n + \cos \phi \mathbf{k}_n). \quad (30)$$

Collecting all the terms in the X_i , Y_i , and Z_i directions, respectively yields:

$$\boldsymbol{\omega} = \begin{bmatrix} \sin \theta & 0 & 1 \\ \cos \phi \cos \theta & \sin \phi & 0 \\ -\sin \phi \cos \theta & \cos \phi & 0 \end{bmatrix} \begin{bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix}. \quad (31)$$

The casting of $\mathbf{J}(\Theta)$ to $\mathbf{J}_\omega(\Theta)$ eliminates singularity problems similar to those in (7). Since it is recommended that (24) be used as a differential relationship rather than (22), $\mathbf{J}_\omega(\Theta)$ is called the *basic Jacobian matrix* [Nak91].

The basic Jacobian matrix, \mathbf{J}_ω , can be recursively computed as follows:

$$\mathbf{J}_\omega = \begin{bmatrix} \mathbf{J}_{\omega 1} & \mathbf{J}_{\omega 0} & \cdots & \mathbf{J}_{\omega n-1} \end{bmatrix}. \quad (32)$$

where $\mathbf{J}_{\omega i}$ is the i^{th} column vector of \mathbf{J}_ω is computed by:

$$\mathbf{J}_{\omega i} = \begin{cases} \begin{pmatrix} \mathbf{k}_i \times \mathbf{p}_i \\ \mathbf{k}_i \end{pmatrix} & \text{(revolute joint)} \\ \begin{pmatrix} \mathbf{k}_i \\ \mathbf{0} \end{pmatrix} & \text{(prismatic joint)} \end{cases} \quad (33)$$

where \mathbf{k}_i is a unit vector in the z direction of the i^{th} link frame and \mathbf{p}_i is a vector from the origin of the i link frame to the end effector reference frame (illustrated in Figure 14). All these vectors are written in the inertial reference frame.

Again, an example might help clarify the concepts. Figure 10 has been modified by inserting the end-effector at the end of link 1 (see Figure 15). Expanding (18) yields the rotation transformation:

$$T_1^I = \begin{bmatrix} \cos(\theta_0 + \theta_1) & -\sin(\theta_0 + \theta_1) & 0 \\ \sin(\theta_0 + \theta_1) & \cos(\theta_0 + \theta_1) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (34)$$

If the link lengths again are x_0 and x_1 and if $\theta_{01} = \theta_0 + \theta_1$, then the vector \mathbf{p}_0 and \mathbf{p}_1 written in inertial coordinates are:

$$\mathbf{p}_0 = \begin{pmatrix} x_0 \cos \theta_0 + x_1 \cos \theta_{01} \\ x_0 \sin \theta_0 + x_1 \sin \theta_{01} \\ 0 \end{pmatrix} \quad (35)$$

$$\mathbf{p}_1 = \begin{pmatrix} x_1 \cos \theta_{01} \\ x_1 \sin \theta_{01} \\ 0 \end{pmatrix}, \quad (36)$$

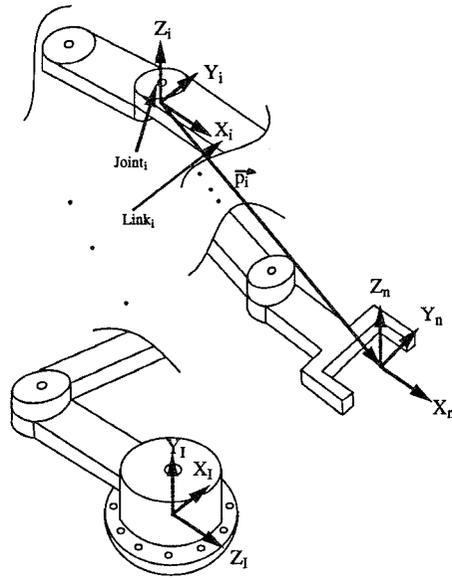


Figure 14: Construction of the Jacobian

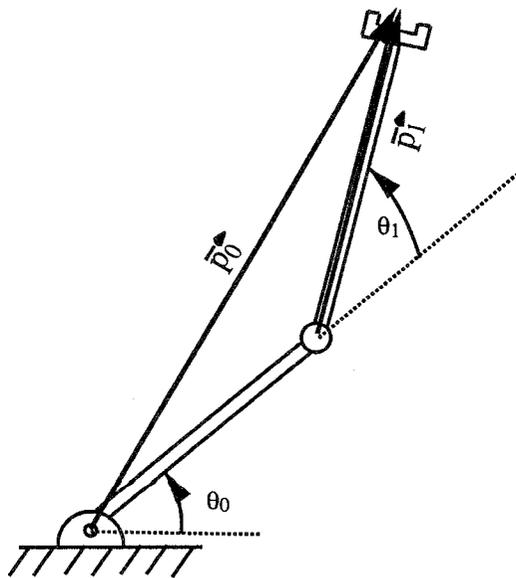


Figure 15: Jacobian Example

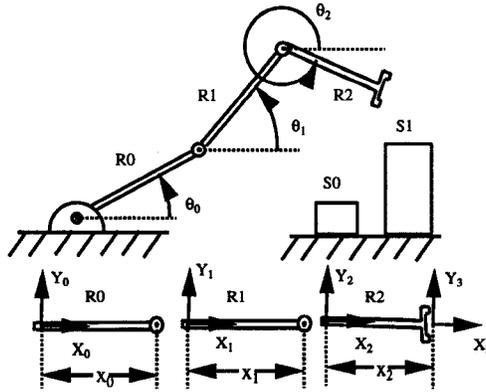


Figure 16: C-Space Toolkit Example

and the Jacobian is:

$$J_{\omega} = \begin{bmatrix} -(x_0 \sin \theta_0 + x_1 \sin \theta_{01}) & -x_1 \sin \theta_{01} \\ x_0 \cos \theta_0 + x_1 \cos \theta_{01} & x_1 \cos \theta_{01} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \end{bmatrix}. \quad (37)$$

4.4 Contact Normals

For a meaningful discussion of contact normals, a basic understanding of how to use the C-Space Toolkit to model a manipulator with obstacles in its workspace will be very useful. The discussion concerning the C-Space Toolkit will be general in nature, with an emphasis on classes and functions in the toolkit as implemented in our Common LISP prototype.

The C-Space Toolkit allows users to choose a 2-D or 3-D world space. After a decision is made, a set of objects that moves as a rigid body can be modeled as a union of convex polygons in 2D or convex polyhedra and convex polygons in 3D. For each such rigid body, a hierarchical representation, called a *movable augmented body*, is computed by `make-kd-hierarchy`, where k is either 2 or 3. Thus, an augmented body is created for each obstacle and for each rigid moving part (link) of the robot. The robot's moving parts are each called an *r-body* and each obstacle is called an *s-body*. For the example illustrated in Figure 16, the *r-bodies* are $R0$, $R1$, $R2$; and the *s-bodies* are $S0$, $S1$ respectively.

Now, each *r-body* needs the functions specifying its forward kinematic map and Jacobian to yield an *r-object*. The joint type and the location and orientation of the joint reference frame with respect to the previous joint reference frame are used by the function `build-ensemble` to construct the forward kinematic map and Jacobian. For the example in Figure 16, all the joints are revolute. For a 2-D world-space, the joint 0 reference frame (X_0, Y_0) is located at $(0, 0, 0)$ (given

as (x, y, θ)) with respect to the inertial reference frame. The joint 1 reference frame (X_1, Y_1) is located at $(x_0, 0, 0)$ with respect to the joint 0 reference frame. The joint 2 reference frame (X_2, Y_2) is located at $(x_1, 0, 0)$ with respect to the joint 1 reference frame. Finally, the end effector reference frame (X_3, Y_3) is located at $(x_2, 0, 0)$ with respect to the joint 2 reference frame.

Next, we use the function `make-c-space-planar-world` to define the underlying C-space. For the current example, the underlying C-space is $S^1 \times S^1 \times S^1$, where S^1 is represented by the interval $[0, 2\pi)$. The world space, the k -D box where the r-objects' reference frame origins can move, must also be defined. For the current example, the two of the R^1 intervals $[0, x_0 + x_1 + x_2]$ can be used to define the constraints in the x and y directions.

The next step is to define the simple induced C-objects using `make-induced-c-object`. Simple induced C-objects are (r-object, s-object) pairs that can collide. An s-object is either an augmented-body or an r-object. For the current example, the simple induced C-objects pairs are $(R0, S)$, $(R1, S)$, $(R2, S)$, $(R0, R2)$, where S is the union of $S0$ and $S1$. The function `make-c-space-object-union` creates a structure (i.e., tree) that is used to determine sub-queries. It is up to the users to define the control for the manipulator motion. When the motion produces contact, or nearly contact, the C-object surface normal is computed, and by "projecting away" the normal components of the intended motion, as "sliding" motion can be accomplished.

In contact with or near a C-object, the normal to the contact surface is given by:

$$\mathbf{n} = \nabla_{\Theta} D(\mathbf{r}) \quad (38)$$

where D is the distance function between the r-object and the s-object and ∇ is the gradient operator. If the chain rule is applied to (38), then the C-object surface normal is:

$$\mathbf{n} = \nabla_{\Theta} \mathbf{r}(\Theta) \nabla_{\mathbf{r}} D(\Theta) = \mathbf{J}_{\omega}^T \nabla_{\mathbf{r}} \mathbf{D}(\Theta). \quad (39)$$

Equation (39) allows the gradient to be computed with respect to the rigid body configuration and then transferred to the C-space coordinates. The gradient with respect to the rigid body configuration is computed using a case analysis dependent on the contact mode. The contact modes given as r-object/s-object feature pairs are summarized below.

2-D Contact Modes	3-D Contact Modes
point/edge	point/edge
edge/point	edge/point
point/point	point/point
	point/face
	face/point
	edge/edge

All other cases are simply multiple contacts. For the 2-D edge-edge contact mode, Figure 17 illustrates that the possible contact modes are (a) a pair of edge/point, (b) a point/edge and edge/point pair, or (c) a pair of point/edge contacts.

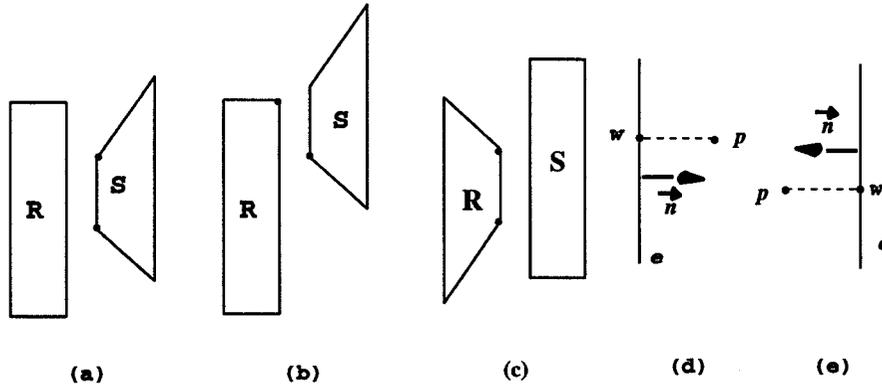


Figure 17: 2-D Edge/Edge Contact Modes

A couple of examples illustrate the kind of case analysis that was done. For the case of a 2-D edge/point R/S contact (Figure 17 (d)), the features necessary for the analysis are edge e , the edge normal vector \mathbf{n} , and points \mathbf{p} and \mathbf{w} .

1. With the current configuration for the r-body being $\mathbf{r} = (x, y, \theta)$, the distance function is defined by

$$D_{ep} = \mathbf{n}(\mathbf{r}) \cdot \mathbf{p} - \mathbf{n}(\mathbf{r}) \cdot \mathbf{w}(\mathbf{r}).$$

2. Since the current position of any body/feature is defined by a map from its initial position to its current position, the current edge normal (\mathbf{n}) is defined in terms of the current configuration and the initial normal (n_x, n_y) :

$$\mathbf{n} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} n_x \\ n_y \end{pmatrix}.$$

3. The witness point \mathbf{w} is similarly described in term of the current configuration and the initial witness point $\mathbf{w}_0 = (w_x, w_y)$:

$$\mathbf{w} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} w_x \\ w_y \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix}.$$

4. Since the point $\mathbf{p} = (p_x, p_y)$ does not move, the distance function can be expanded to

$$D_{ep} = p_x(n_x \cos \theta - n_y \sin \theta) + p_y(n_x \sin \theta + n_y \cos \theta) - \mathbf{n} \cdot \mathbf{w}_0 - x(n_x \cos \theta - n_y \sin \theta) - y(n_x \sin \theta + n_y \cos \theta).$$

5. The gradient of the distance function is given by:

$$\nabla_{\mathbf{r}}(D_{ep}) = \begin{pmatrix} n_y \sin \theta - n_x \cos \theta \\ n_x \sin \theta + n_y \cos \theta \\ \left(-p_x(n_x \sin \theta + n_y \cos \theta) + p_y(n_x \cos \theta - n_y \sin \theta) \right) \\ + (n_x x + n_y y) \sin \theta + n_y x - n_x y \cos \theta \end{pmatrix}.$$

The contact normal for the 2-D point/edge contact (Figure 17 (e)) is derived as follows.

1. The distance function is given by:

$$D_{pe} = \mathbf{n} \cdot \mathbf{p}(\mathbf{r}) - \mathbf{n} \cdot \mathbf{w}.$$

2. The current edge normal is again given by:

$$\mathbf{n} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} n_x \\ n_y \end{pmatrix}.$$

3. The point, described in terms of the current configuration and its original location $p_0 = (x_0, y_0)$ is:

$$\mathbf{p} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix}.$$

4. The distance function is expanded to:

$$D_{pe} = \mathbf{n} \cdot \begin{pmatrix} x_0 \cos \theta - y_0 \sin \theta + x \\ x_0 \sin \theta + y_0 \cos \theta + y \end{pmatrix} - \mathbf{n} \cdot \mathbf{w}.$$

5. The gradient of the distance function is given by:

$$\nabla_{\mathbf{r}}(D_{ep}) = \begin{pmatrix} n_x \\ n_y \\ -n_x(x_0 \sin \theta + y_0 \cos \theta) + n_y((x_0 \cos \theta) - y_0 \sin \theta) \end{pmatrix}.$$

All the other cases are derived in a similar manner, and have been implemented in the prototype implementation of the CSTk Kernel.

The appendix describes how to write code that uses these functions in a motion filter that converts a C-space motion that would cause a robotic manipulator to penetrate obstacles into a motion that slides along the obstacles.

5 Conclusion and Ongoing and Future Work

We have presented the motivation behind the C-Space Toolkit project, described its development, and presented its technical and research results. The project had several important achievements.

- We developed a Toolkit architecture based on a Kernel of operations that meets immediate and short-term needs for an implementation of the C-space abstraction but that is designed for extensibility in underlying geometry and in set-oriented operations.
- We developed and implemented (C++) an efficient algorithm supporting extended point classification in C-space, distance computation and interference detection.
- We developed and implemented (C++) a fast, generic algorithm for constructing hierarchical geometric representations that are spatially balanced, with optional control over structural balance and coherence.
- We implemented a Common LISP/C prototype of the full C-Space Toolkit Kernel.

In this report, we also presented the underlying mathematics that link our geometrical algorithms to the C-space abstraction. All our implementations support three-dimensional model geometry.

The results of the C-Space Toolkit project have provided the foundation for further research. Just after the project's FY1995 conclusion, Sandia's SANDROS motion planner [CH92] was ported to use the the C-Space Toolkit's C++ library, libcstk.so. Initial results brought SANDROS planning time for real world gross-motion example problems down from hours to minutes. Subsequent experimental research led to improvements both in SANDROS and libcstk.so that have made the integrated system performance practical for path-planning in everyday robotics [WX97]. Other research building on the code base developed under the C-Space Toolkit Project has led to results in collision detection [Xav97] that support assembly planning and the accurate simulation of rigid-body systems with intermittent contacts. These results will be reported in forthcoming SAND reports.

A Appendix: 3-D Slider Example

The best way to illustrate how to use the C-Space is through an example. In this case, a Puma-like robot encounters a tilted cylinder (Figure 18) and slides along the surface. Following the description of how to use the CSTk Common LISP prototype presented in the contact normals discussion in §4.4, a 3D world is space assumed, and the three Puma rigid bodies (Figure 19) are defined in `*puma260-kdrons*`. The cylinder is defined in `*s-drons3*`. Defining a hierarchical representation is accomplished in the following Lisp code:

```
(setq *test3d-r-body0*  
      (make-3d-hierarchy-better (list (first *puma260-kdrons*))))  
(setq *test3d-r-body1*  
      (make-3d-hierarchy-better (list (second *puma260-kdrons*))))
```

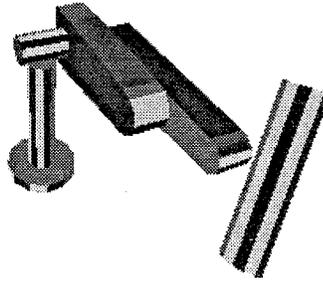


Figure 18: C-Space Toolkit Example

```
(setq *test3d-r-body2*
      (make-3d-hierarchy-better (list (third *puma260-kdrons*))))
```

Next, the ensemble of rigid bodies is built, using the `build-ensemble` function. When calling this function the user specifies a list of which r-body to use, where on the previous body the reference frame is to go (i.e. the parameter `:at`), the type of joint (i.e. the parameter `:how`), and the index of the joint parameter (i.e. the parameter `:using`).

```
(setq *ens3d*
      (build-ensemble
        (list
          (list *test3d-r-body0* 0
                :at #(0.0d0 0.0d0 0.0d0 0.0d0 0.0d0 0.0d0)
                :how 'revolute :using 0)
          (list *test3d-r-body1* 1
                :at #(0.0d0 2.875d0 0.0d0 0.0d0 0.0d0 (/ *pi* 2))
                :how 'revolute :using 1) ;; y and z rotated 90 degrees
          (list *test3d-r-body2* 2
                :at #(6.5d0 0.0d0 -2.125d0 0.0d0 0.0d0 0.0d0 0.0d0)
                :how 'revolute :using 2))))
```

The `build-ensemble` function builds the forward kinematic map and the Jacobian. The s-body and the r-objects are now created.

```
(setq *test3d-s-body* (make-3d-hierarchy-better *s-drons3*))
(setq *r-obj3d-a* (first *ens3d*))
(setq *r-obj3d-b* (second *ens3d*))
(setq *r-obj3d-c* (third *ens3d*))
```

The underlying C-space is defined with the function `make-c-space-3d-world`. Here, the call tells the function to use the default $S^1 \times S^1 \times S^1$ space for the joint parameters and a world space of ± 20 for each of the X , Y , and Z axes.

```
(setq *c-space-3d*
      (make-c-space-3d-world '(() () ())
                            '((-20 20) (-20 20) (-20 20))
                            *ens3d*))
```

The simple induced c-objects are created by pairing each of the r-objects (robot links) with the s-body (cylinder) and including the underlying C-space. The `:contact-epsilon` parameter is the stand-off distance where contact analysis is considered.

```
(setq *c-obj3d-a*
      (make-induced-c-object *r-obj3d-a* *test3d-s-body* *c-space-3d*
                            :contact-epsilon max-shell))
(setq *c-obj3d-b*
      (make-induced-c-object *r-obj3d-b* *test3d-s-body* *c-space-3d*
                            :contact-epsilon max-shell))
(setq *c-obj3d-c*
      (make-induced-c-object *r-obj3d-c* *test3d-s-body* *c-space-3d*
                            :contact-epsilon max-shell))
```

The function `make-c-object-union` is called to create the tree structure for sub-queries.

```
(setq *c-obj-union*
      (make-c-object-union (list *c-obj3d-a* *c-obj3d-b* *c-obj3d-c*)))
```

The user now has a C-space representation ready for an application. In this case, the application is to make the robot move in a prescribed manner unless it encounters an obstacle. If it does, the robot should move tangent to any surface until it clears the obstacle. The function `slide` takes as input a unioned C-object, a configuration of joint parameters, a step in configuration space, and a stand-off distance. The function first takes the step size. If nothing is encountered, it returns the new configuration. If an obstacle is encountered or if the robot is within the stand-off distance of the obstacle, the step is recomputed to slide the robot along the obstacle surface using the contact normal information. The new configuration is returned. If a new step cannot be computed, `NIL` is returned. The pseudo-code using `slide` looks like:

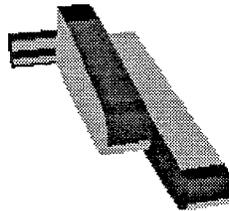


Figure 19: Puma r-bodies. Note the absence of the Puma base, which is not a moving body in our simplified model.

```
(defun run-slider (*c-obj-union* start-config step number-of-steps)
  (show-c-object *c-obj-union* start-config)
  (setq config start-config)
  (setq count 0)
  (another-loop-macro
   (setq new-config (slide *c-obj-union* config step ))
   (setq count count+1)
   :quit-loop-if (or (null new-config) (>= count number-of-steps))
   (setq config new-config)
   (show-c-object *c-obj-union* config)))
```

The function `slide` itself calls the CSTk function `cstk-pseudo-normal` to get the point-classification with respect to the `c-object`, the `c-object` normals, the primitive witness features, and the minimum distance between `r-objects` and `s-objects` that define the lowest-level `c-objects`. These values are used by functions `slide` calls to compute a safe, adjusted step. If the configuration is inside the `c-object`, `slide` returns `NIL`. Otherwise, it calls `slide-step` to get a good `c-space` step that does not move closer to the `c-object`. If the distance is closer than the `desired-dist`, this step is adjusted with the help of `compute-spring-step`. The step is then returned.

```
(defmethod slide ((c-obj union-c-object) config step
                 &key max-shell (desired-dist (/ max-shell 10))
                 normalize? (re-norm-fac 1))
  (declare (ignore re-norm-fac))
```

```

(multiple-value-bind (c-o-norms dist p-c witnesses)
  (cstk-pseudo-normal config c-obj :max-shell max-shell
    :normalize? t)
  (unless (eq p-c 'INSIDE)
    (let ((good-step
          (slide-step step c-o-norms witnesses dist
            :max-shell max-shell :normalize? normalize?
            :desired-dist desired-dist)))
      (when good-step
        (let ((adjustment
              (compute-spring-step c-o-norms witnesses
                desired-dist)))
          (if adjustment
            (vec-add good-step (vec-add adjustment config))
            (vec-add good-step config))))))))))

```

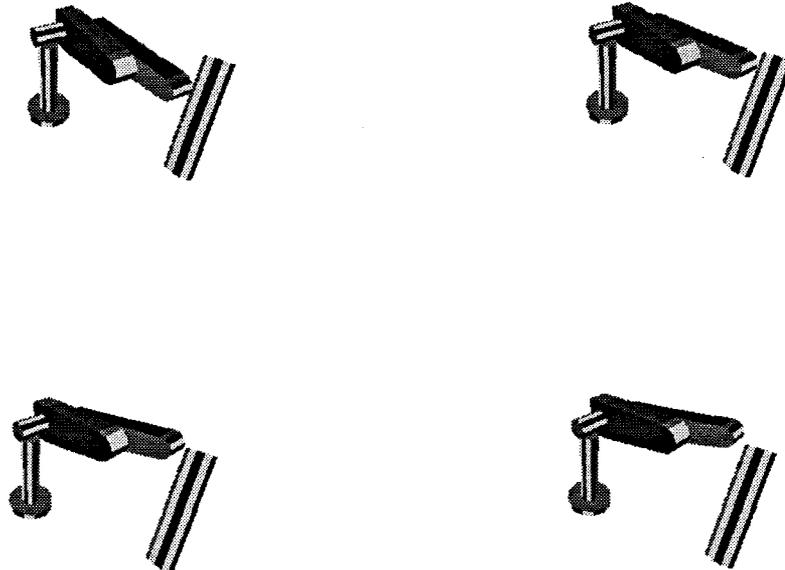


Figure 20: A Puma-like robot encounters a tilted cylinder (Figure 18) and slides along the surface. The figure shows the configurations 10, 20, 30, and 40 steps into the simulation.

Figure 20 illustrates the results of using the slider routine for an initial configuration of $(.17, 0, 0)$ and a step of $(.01, 0, 0)$. The figure shows the configuration after 10, 20, 30, and 40 steps. While a

few intermediate function calls have been omitted from this discussion, users should get a feel for how easy it is to use the toolkit relative to the difficulty of the task.

B Project-Related Information

Awards: None.

Publications and presentation:

- P. Xavier. "A Generic Algorithm for Constructing Hierarchical Representations of Geometric Objects", *Proc. 1996 IEEE Int'l Conf. on Robotics and Automation*, Minneapolis, MN, April 1996, pp. 3644-3651. Presented, April 1996.
- P. Xavier and R. Lafarge. "A C-Space Toolkit For Automated Spatial Reasoning: Technical Results and LDRD Project Final Report", Sandia Report SAND97-0366, Sandia National Laboratories, February 1997.

Patents: none.

Copyrights: At the time of this printing, the "C-Space Toolkit" software is going through the copyrighting process.

Employee recruitment: None external. Robert A. Lafarge, internal.

Student involvement: None.

Follow-on work: [CH92] was ported to use the the C-Space Toolkit's C++ library, libcstk.so. Initial results brought SANDROS planning time for real world gross-motion example problems down from hours to minutes. Subsequent experimental research led to improvements both in SANDROS and libcstk.so that have made the integrated system performance practical for path-planning in everyday robotics [WX97]. Motion-planning activity has been in support of joint research and development with Deneb Robotics and General Motors Truck and Bus Division. Other research building on the code base developed under the C-Space Toolkit Project has led to results in collision detection [Xav97] that support assembly planning and the accurate simulation of rigid-body systems with intermittent contacts.

Original report due date: 1/31/96.

References

- [ANFN88] S. Arimoto, H. Nohorio, S. Fukuda, and A. Noda. A feasible approach to automatic planning of collision-free robot motions. In *Int'l Symp. on Robotics Research*, 1988.
- [Bar90] D. Baraff. Curved surfaces and coherence for nonpenetrating rigid body simulation. *Computer Graphics (Proc. SIGGRAPH)*, 24(4):19–28, August 1990.
- [BC93] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A research agenda. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages (3)549–556, May 1993.
- [BC94] R. C. Brost and A. D. Christiansen. Probabilistic analysis of manipulation tasks: A computational framework. Sandia Report SAND92-2033, Sandia Nat'l Laboratories, January 1994.
- [BC95] R. C. Brost and A. D. Christiansen. Empirical verification of fine-motion planning theories. In *Preprints of the Fourth International Symposium of Experimental Robotics*, May 1995.
- [BDH] C. B. Barber, D .P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. To appear in *ACM Trans. on Mathematical Software*. Also appears as Tech. Rept. GCG 53, Geometry Center at U. Minnesota, Minnesota, MN, 1993.
- [BL89] J. Barraquand and J. C. Latombe. On non-holonomic mobile robots and optimal maneuvering. In *Proc. IEEE Int'l Symp. on Intelligent Control*, pages 340–347, Albany, NY, September 1989.
- [BN90a] M. Branicky and W. Newman. Rapid computation of configuration space obstacles. In *Proc. IEEE ICRA 1990*, pages 304–310, Cincinnati, Ohio, 1990.
- [BN90b] P. Brunet and I. Navazo. Solid representation and operation using extended octrees. *ACM Trans. on Graphics*, 9(2):170–197, April 1990.
- [Bro86] R. Brost. Automatic grasp planning in the presence of uncertainty. In *Proc. of the 1986 IEEE Int'l Conf. on Robotics and Automation*, San Francisco, CA, 1986.
- [Bro91] R. Brost. *Analysis and Planning of Planar Manipulation Tasks*. PhD thesis, Carnegie Mellon University, 1991.
- [Cam90] S. Cameron. Collision detection by 4D intersection testing. *Int'l Journal of Robotics Research*, 6(3):291–302, June 1990.
- [Cam96a] S. Cameron. A comparison of two fast algorithms for computing the distance between convex polyhedra. *submitted to IEEE Trans. on Robotics and Automation*, July 1996.

- [Cam96b] S. Cameron. Dealing with geometric complexity in motion planning. In *IEEE ICRA 1996 Workshop on Practical Motion Planning in Robotics*, Minneapolis, MN, 1996.
- [Can86] J. Canny. Collision detection for moving polyhedra. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(2):200–209, 1986.
- [Can88] J. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, Massachusetts, 1988. Book version of Canny's 1986 Ph.D. thesis.
- [CH92] P. C. Chen and Y. Hwang. Sandros: A motion planner with performance proportional to task difficulty. In *Proc. of the 1992 IEEE Int'l Conf. on Robotics and Automation*, pages 2346–2352, Nice, France, 1992.
- [CL91] J. Canny and M. C. Lin. A fast algorithm for incremental distance calculation. In *Proc. IEEE ICRA 1991*, pages 1008–1014, Sacramento, California, 1991.
- [Cla76] J. H. Clark. Hierarchical geometry models for visible surface algorithms. *Communications of the ACM*, 19(10):547–555, 1976.
- [CLMP95] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proc. ACM Interactive 3D Graphics Conf.*, pages 189–196, 1995.
- [CQM95] S. Cameron, C. Qin, and A. McLean. Toward efficient motion planning for manipulators with complex geometry. In *Proc. IEEE Int'l Symp. on Assembly and Task Planning*, pages 207–212, 1995.
- [CR87] J. Canny and J Reif. New lower bound techniques for robot motion planning. In *Proc. of the 28th Annual Symp. on the Foundations of Computer Science*, Los Angeles, California, 1987.
- [Cra89] J. J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, New York, NY, 2nd edition, 1989.
- [Don87] B. Donald. A search algorithm for motion planning with six degrees of freedom. *Artificial Intelligence*, 31(3):295–353, 1987.
- [Don89] B. Donald. *Error Detection and Recovery in Robotics*. LNCS 336. Springer-Verlag, Berlin, Germany, 1989. Book version of Donald's 1987 Ph.D. thesis.
- [dPSL92] A. del Pobil, M. Serna, and J. Llovet. A new representation for collision avoidance and detection. In *Proc. IEEE ICRA 1992*, pages 246–251, 1992.
- [EM88] M. Erdmann and M. T. Mason. An exploration of sensorless manipulation. *IEEE Journal of Robotics and Automation*, 4(4):369–379, 1988.

- [Erd86] M. Erdmann. Using backprojections for fine motion planning with uncertainty. *Int'l Journal of Robotics Research*, 5(1), 1986.
- [Fav89] B. Faverjon. Hierarchical object models for efficient anti-collision algorithms. In *Proc. IEEE ICRA 1989*, pages 333–340, Scottsdale, AZ, 1989.
- [FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Proc. of ACM SIGGRAPH*, pages 124–133, 1980.
- [FT87] B. Faverjon and P. Tournassoud. A local based approach for path planning of manipulators with a high number of degrees of freedom. In *Proc. IEEE ICRA 1987*, pages 1152–1159, Raleigh, North Carolina, 1987.
- [GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2), April 1988.
- [GLM96] S. Gottschalk, M.C. Lin, and D. Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *Proc. ACM SIGGRAPH '96*, 1996.
- [GM90] K. Y. Goldberg and M. T. Mason. Bayesian grasping. In *Proc. 1990 IEEE Int'l Conf. on Robotics and Automation*, pages 1264–1269, Cincinnati, OH, May 1990.
- [GS87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.
- [HKM95] M. Held, J. Klosowski, and J. S. B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In *Proc. 7th Canadian Conf. on Computational Geometry*, Quebec, Canada, 1995.
- [Hub93] P. Hubbard. Interactive collision detection. In *Proc. IEEE Symposium on Research Frontiers in Virtual Reality*, pages 24–31, October 1993.
- [Hub94] P. Hubbard. *Collision Detection for Interactive Graphics Applications*. PhD thesis, Brown University, Providence, RI, October 1994.
- [Hub96] P. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. on Graphics*, 15(3), July 1996.
- [JT80] C. L. Jackins and S. L. Tanimoto. Octrees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, (14):249–270, 1980.
- [KL94] L. E. Kavaraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proc. 1994 IEEE Int'l Conf. on Robotics and Automation*, pages 2138–2145, San Diego, CA, 1994.

- [LMC94] M. Lin, D. Manocha, and J. Canny. Fast contact determination in dynamic environments. In *Proc. IEEE ICRA 1994*, pages 602–607, San Diego, CA, May 1994.
- [LP83] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Trans. on Computers*, C-32(2):108–120, 1983. Also MIT A.I. Memo 605, December 1982.
- [LPMT84] T. Lozano-Pérez, M. Mason, and R. Taylor. Automatic synthesis of fine-motion strategies for robots. *Int'l Journal of Robotics Research*, 3(1):3–24, Spring 1984.
- [Man88] M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.
- [MC95] B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. In *Proc. Monterey Symp. on Real-Time Interactive Graphics*, April 1995.
- [Mea82] D Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, (19):129–147, 1982.
- [Nak91] Y. Nakamura. *Advanced Robotics: Redundancy and Optimization*. Addison-Wesley, New York, NY, 1991.
- [NAT90] B. Naylor, J. Amanatides, and W. Thibault. Merging bsp trees yields polyhedral set operations. *Computer Graphics (Proc. SIGGRAPH 1990)*, 24, August 1990.
- [Nay92] B. Naylor. Interactive solid geometry via partitioning trees. In *Proc. of Graphics Interface*, pages 11–18, May 1992.
- [Qui94] S. Quinlan. Efficient distance computation between non-convex objects. In *Proc. 1994 IEEE Int'l Conf on Robotics and Automation*, San Diego, CA, 1994.
- [Req80] A. A. G. Requicha. Representations of solid objects — theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.
- [RW80] S. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics (Proc. SIGGRAPH)*, pages 110–116, July 1980.
- [SHMA96] Y. Sato, M. Hirata, T. Maruyama, and Y. Arita. Efficient collision detection using fast distance calculation algorithms for convex and non-convex objects. In *Proc. 1995 IEEE Int'l Conf. on Robotics and Automation*, pages 772–778, Minneapolis, MN, April 1996.
- [SS84] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 144–153, 1984.
- [Van91] G. Vanecek. Brep index, a multi-dimensional space partitioning tree. In *Proc. 1st ACM-SIGGRAPH Symp. on Solid Modeling Foundations and CAD/CAM Applications*, pages 35–44, Austin, TX, June 1991.

- [WHG84] H. Weghorst, G. Hooper, and D. Greenberg. Improved computational methods for ray tracing. *ACM Trans. on Graphics*, 3(1):52–69, January 1984.
- [Wu92] Xiaolin Wu. A linear-time simple bounding volume algorithm. In D. Kirk, editor, *Graphics Gems III*, pages 301–306. Academic Press, San Diego, CA, 1992.
- [WX97] P. Watterberg and P. Xavier. Path planning for everyday robotics with SANDROS. In *Proc. 1997 IEEE Int'l Conf. on Robotics and Automation*, Albuquerque, NM, April 1997.
- [Xav96] P. Xavier. A generic algorithm for constructing hierarchical representations of geometric objects. In *Proc. 1996 IEEE Int'l Conf. on Robotics and Automation*, pages 3644–3651, Minneapolis, MN, April 1996.
- [Xav97] P. Xavier. Fast swept-volume distance for robust collision detection. In *Proc. 1997 IEEE Int'l Conf. on Robotics and Automation*, Albuquerque, NM, April 1997.
- [ZF95] G. Zachmann and W. Felger. The box tree: Enabling real time and exact collision detection of arbitrary polyhedra. In *Proc. 1st Workshop on Simulation and Interaction in Virtual Environments*, pages 104–113, Iowa City, IA, July 1995.

Distribution:

MS 0188	LDRD Office, 4523	(1)
MS 9018	Central Technical Files, 8940-2	(1)
MS 0899	Technical Library, 4414	(5)
MS 0619	Review & Approval Desk, 12690, for DOE/OSTI	(2)
MS 1008	Patrick Xavier, 9621	(25)
MS 1008	Robert Lafarge, 9621	(15)