FCL: A General Purpose Library for Collision and Proximity Queries

Jia Pan1  Sachin Chitta2  Dinesh Manocha1

Abstract—We present a new collision and proximity library that integrates several techniques for fast accurate collision checking and proximity detection. Our library is based on hierarchical representations and designed to perform multiple proximity queries on different model representations. The set of queries include discrete collision detection, continuous collision detection, separation distance computation and penetration depth estimation. The input models may correspond to triangulated rigid or deformable models and articulated models. Moreover, FCL can perform probabilistic collision checking between noisy point clouds that are captured using cameras or LIDAR sensors. The main benefit of FCL lies in the fact that it provides a unified interface that can be used by various applications. Furthermore, its flexible architecture makes it easier to implement new algorithms within this framework. The runtime performance of the library is comparable to state of the art collision and proximity algorithms. We demonstrate its performance on synthetic datasets as well as motion planning and grasping computations performed using a two-armed mobile manipulation robot.

I. INTRODUCTION

The problems of collision and proximity computation are widely studied in various fields including robotics, simulated environments, haptics, computer games and computational geometry. The set of queries include discrete collision checking, separation distance computation between two non-overlapping objects, first point of contact computation between continuous moving objects, and penetration depth computation between overlapping objects. Furthermore, the underlying geometric representations may correspond to rigid objects (e.g., computer games), articulated models (e.g., mobile manipulators), deformable models (e.g., surgical or cloth simulators) or point-cloud datasets (e.g., captured using camera or LIDAR sensors on a robot).

Many efficient algorithms have been proposed to perform collision and proximity queries on various types of models. At a broad level, they can be classified based on the underlying query or model representation [16], [6]. Some of the commonly used techniques for general polygonal models are based on bounding volume hierarchies, which can be used for collision and separation distance queries, and can be extended to deformable models. Moreover, many of these algorithms have been used to design widely used libraries such as I-COLLIDE, Bullet, ODE, RAPID, PQP, SOLID, OPCODE, V-Clip, Self-CCD, etc. However, these libraries have two main restrictions: 1) They are limited to specific queries (e.g., discrete collision checking or separation distance computation) on certain types of models (e.g., convex polytopes or rigid objects). 2) It is hard to modify or extend these libraries in terms of using a different algorithm or representation. For example, SOLID [3] is designed to perform collision checking using OBB trees; RAPID is designed for performing collision checking using OBB Trees [9], and PQP performs separation distance queries using rectangular swept sphere (RSS) trees [13]. It is hard to use a different bounding volume with each of these libraries or use a different hierarchy computation or traversal scheme.

Many applications need to perform different collision and proximity queries. Figure 1 shows an example task where the PR2 mobile manipulation robot is executing a pick and place task using a combination of grasping, motion planning and control algorithms. Continuous collision detection queries are useful for grasp planning executed by the robot to generate grasps for the objects. The robot uses sample-based motion planners to compute collision-free paths. It is well known that a high fraction of running time for sample-based planning is spent in collision/proximity queries, underlining the need for fast efficient proximity and collision queries.

The local planning algorithms that form the underlying basis of sampling-based planners usually carry out multiple discretized collision queries in each step of the planning process and can be accelerated by continuous collision checking algorithms. Many sampling schemes either use separation distance computation [15] or penetration-depth estimation (e.g., retraction planners) to compute samples in narrow passages. Proximity information can also be used to plan paths that are further away from obstacles, allowing the robot to execute the plans at higher speeds.

Main Results: We present a new collision checking library, labeled FCL (Fast Collision Library), which provides a unified interface to perform different proximity queries. Furthermore, it is able to handle a wide class of models, including rigid and deformable objects, articulated models

Fig. 1: The PR2 is a mobile manipulation system with integrated stereo and laser sensors. Left: A pick and place task with the PR2 was among the experimental tasks used to validate FCL. Right: A visualization of the environment that the PR2 is working with.
and noisy point clouds. We propose a new system architecture that is quite flexible in terms of performing different set of queries and can be easily extended in terms of adding new algorithms and representations. It includes proximity computations between convex polytopes, general polygonal models, articulated models, deformable models and noisy point clouds. In order to perform different queries, FCL models them as as a traversal process along a bounding volume hierarchy. Different queries use the same traversal framework, but differ in terms of intermediate data and traversal strategies. The overall performance of the FCL is comparable to state-of-the-art algorithms.

We validate our techniques on a real-world system through integration with a pick and place manipulation task performed on the PR2 robot (Figure 1). The PR2 robot has both stereo and laser range finders that provide point cloud data at a high rate. The robot must be able to calculate feasible motion (i.e., motion that is collision-free and satisfies some additional dynamics constraints) quickly through cluttered environments. We integrate our collision checking methods with the open source OMPL motion planning library [2] and demonstrate fast and accurate motion planning that allows the robot to complete its task. FCL is available as an independent library at https://kforge.ros.org/project/fcl/. A ROS interface to FCL is also provided so that users can easily call FCL collision/proximity queries for different robots.

The rest of the paper is organized in the following manner. We survey related work on collision and proximity queries in Section II. Section III gives an overview of the library and the detailed architecture is described in Section IV. We highlight a few applications and performance in Section V.

II. BACKGROUND AND RELATED WORK

In this section, we first give a brief overview of prior work on collision and proximity queries. Furthermore, we highlight the underlying algorithms that are used in FCL to perform different queries.

A. Collision and Proximity Computation Algorithms

The problems of collision detection and distance computations are well studied [6], [7], [16]. At a broad level, they can be classified based on algorithms for convex polytopes, bounding volume hierarchies, continuous collision detection, broad-phase collision detection and point-cloud collisions. Table I shows all the queries currently supported by FCL.

1) Convex Polytope based Collision: Many methods have been proposed to compute the Euclidean distance between two convex polytopes, such as the Gilbert–Johnson–Keerthi (GJK) algorithm [8] and the Lin-Canny algorithm [17]. The optimization technique based on Minkowski formulation in GJK algorithm can also be used to compute translational penetration depth. The convex polytope collision in FCL is based on GJK and EPA (Expanding Polytope Algorithm) [25].

2) Bounding Volume Hierarchy based Collision: Some of the most widely used algorithms for triangulated or polygonal models are based on bounding volume hierarchies. Typical examples of bounding volumes include axis-aligned boxes (AABB) [3], spheres, oriented boxes (OBB) [9], discrete oriented polytope (k-DOP) [12] and swept sphere volumes (SSV) [13], and they have been mainly used to perform discrete collision detection and separation distance queries. Furthermore, they can be extended to deformable models by updating the hierarchies during each step of the simulation [23]. Spatial decomposition techniques, such as kd-trees and octrees, have also been used for collision checking, though techniques based on BVHs are considered faster. FCL can support different bounding volume hierarchies to perform various queries on rigid and deformable models.

3) Continuous Collision Detection: In many applications (e.g., local planning in motion planning [15]), we need to check the collision for objects moving along a continuous path. One solution is to sample along the path and then perform collision checking at discrete time steps. Such discrete collision checking method may miss collisions between the sampled time steps. While adaptive sampling strategies and predictive methods can be used to alleviate this problem [15], they can be relatively slow. In order to provide rigorous guarantee, continuous collision detection (CCD) techniques have been proposed [22], which compute the first time of contact between two moving objects along a continuous path. CCD is typically performed by using bounding volume hierarchies. The BVH used for CCD computations provides a conservative bound for the swept volume of an object generated during the given time interval. The BVH is usually generated by refitting the static object BVH in a bottom-up manner and is based on the motion and trajectory of the object. These include algorithms for linearly interpolating motion between two static configurations [10] and arbitrary in-between rigid motions [22]. Another popular framework for CCD is conservative advancement (CA), which incrementally advances objects by a time step while avoiding collision. In order to determine the conservative time step, it needs to compute the minimal separation distance between the objects and uses it to estimate conservative motion bounds. Tang et al. [24] and Pan et al. [21] present algorithms to compute motion bounds for screw motion and spline motion. Conservative advancement algorithm can be applied to non-convex models based on BVHs. FCL uses these CCD and CA algorithms.

4) Broad-Phase Collision Detection: Many applications need to perform collisions between a large number of objects, including different links of an articulate models and self-collisions in deformable models. In order to avoid \( O(n^2) \) collision checking between the objects or primitives, broad-phase collision detection algorithms are used to cull away object pairs that are far-away from each other. Widely used N-body collision algorithms include sweep and prune (SaP) [5] and spatial subdivision [7]. FCL currently uses SaP which tends to work well when the moving objects have a high spatial coherence. In the ROS interface of FCL, the broad-phase collision algorithm is combined with the kinematic model implementation in ROS to perform robot self-collision as well as collision detection with the environment.
TABLE I: The design of FCL makes it possible to perform different proximity queries on various models. Current capabilities of FCL are shown with √ symbol. Symbol X are capabilities that are currently not supported and may be added in the future. And symbol x are capabilities that can be implemented by existing capabilities of FCL. The continuous collision detection between point clouds (√) is not completely implemented, and the current version only supports CCD queries between a point cloud and a triangulated (mesh) representation.

<table>
<thead>
<tr>
<th></th>
<th>Rigid Objects</th>
<th>Point Cloud</th>
<th>Deformable Objects</th>
<th>Articulated Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collision Detection</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Continuous Collision Detection</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Self Collision Detection</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Penetration Estimation</td>
<td>√</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Distance Computation</td>
<td>√</td>
<td>X</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Broad-phase Collision</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

5) Point Cloud Collision Detection: There has been relatively little work in terms of handling collisions between point clouds or between point clouds and unstructured meshes. With the recent advances in RGB-D cameras and LIDAR sensors, there is increased interest in performing various queries on noisy point-cloud datasets. FCL supports collision checking between triangle meshes/soups and point clouds as well as collision checking between point clouds. The former is useful for collision checking between robot parts and the environment, while the latter is used for collision checking between scanned objects (e.g. held by a robot gripper) and the environment. The point cloud collision checking algorithms in FCL also take into account noise in the point cloud data arising from various sensors as well as the inherent shape uncertainty in point cloud data arising from discretization [19].

6) Parallel Collision and Proximity Computation: The popularity of multi-core CPUs and multi-core GPUs makes it necessary to design parallel collision and proximity computation algorithms that can exploit the capabilities of multi-core processors [23]. Many GPU-based parallel collision and proximity computation algorithms have been proposed, especially for a single collision query [14], multiple collision queries [20] or N-body collision [18]. In practice, GPU-based algorithms can offer considerable speedup over CPU-based algorithms and are a good candidate for future extensions to FCL.

III. OVERVIEW

In this section, we first give a formal description of various queries supported in FCL and then give an overview of the FCL library.

A. Definitions

Given two objects A and B as well as their configurations \( q_A \) and \( q_B \), the discrete collision query (DCD query) returns a yes/no answer about whether the two objects are in collision or not, i.e., whether

\[
A(q_A) \cap B(q_B) \neq \emptyset
\]

is true. Optionally, collision query can also return the contact points where object intersection happens and the corresponding contact normals.

Given two objects A and B as well as their motions \( q_A(t) \) and \( q_B(t) \), where \( t \in [0,1] \), the continuous collision query (CCD query) returns a yes/no answer about whether the two objects are in collision within interval \( [0,1] \), i.e., whether

\[
\exists t \in [0,1], A(q_A(t)) \cap B(q_B(t)) \neq \emptyset.
\]

If a collision occurs, it also returns the first time of contacts:

\[
toc = \inf \{ t : A(q_A(t)) \cap B(q_B(t)) \neq \emptyset \}.
\]

Given two non-overlapping objects A and B as well as their configurations \( q_A \) and \( q_B \), the separation distance query (SD query) returns the distance between them:

\[
dis = \inf \{ \| x - y \| : x \in A(q_A), y \in B(q_B) \}.
\]

Optionally, distance query can also return the closest pair of points:

\[
\arg \min_{x \in A(q_A), y \in B(q_B)} \| x - y \|.
\]

Given two objects A and B as well as their configurations \( q_A \) and \( q_B \), the penetration depth query (PD query) returns the translational penetration depth between the objects when they are in collision:

\[
\text{pd} = \inf \{ \| d \| : (A(q_A) + d) \cap B(q_B) = \emptyset \}.
\]

The exact computation of penetration depth between non-convex or deformable models has a high complexity \( O(n^6) \) [11]. As a result, FCL provides capability to approximate penetration depth between two mesh-based models or mesh-polytope pair by computing the penetration between two colliding triangles or triangle-polytope pairs. FCL also provides penetration depth between two polytopes using EPA algorithm [25].

Given a set of objects \( \{ A_i \}_{i=1}^n \) with their configurations \( q_i \), broad-phase collision query returns a yes/no answer about whether any two of the objects are in collision or not : i.e., whether

\[
\exists i \neq j \in \{1,2,...,n\}, A_i(q_i) \cap A_j(q_j) \neq \emptyset.
\]

Optionally, it also returns all the pairs of in-collision objects.

B. FCL Overview

FCL is a fully templated, modern C++ library that can perform the collision and proximity queries highlighted above. From an application perspective, FCL is meant to provide unified and extendable interfaces for collision and proximity computation algorithms. Moreover, it is designed to support different data representations, including triangle meshes/soups and shape primitives (e.g., sphere and cylinder) in a consistent manner. To achieve the above goals, the FCL framework models all collision and proximity queries between two objects as a traversal process along a hierarchical structure. Different queries will share the same traversal framework but may use different intermediate data...
and traversal strategies. As shown in Figure 2, the traversal process is performed in three steps in FCL:

1) Object representation: The objects in a query are represented by a hierarchical structure suitable for the required query. For example, basic geometric shapes (e.g., cones, cylinders, spheres) are represented in one-level hierarchy with a corresponding polytope as the unique node. Arbitrary geometric objects are represented using a bounding volume hierarchy of triangle meshes/soups or point clouds. The hierarchical structure along with the object’s configuration information is stored in a structure called CollisionObject, which also contains the object’s shape information in the previous time frame if the object is deformable.

2) Traversal node initialization: The traversal node is the structure which stores the complete information required to perform a traversal process for a specific type of collision/proximity query, which may be different among different types of queries on different object representations. For example, for a continuous collision query, we need to set the object shape information for a previous time frame. The traversal node also decides the traversal strategy when traversing the hierarchy structure. For example, if only yes/no answer is required for collision query, the traversal can stop once the collision is found. Such an early stop strategy is, however, not valid for distance queries.

3) Hierarchy traversal: After the traversal node is prepared, we execute the traversal along the hierarchical structure to perform the required collision/proximity query.

Collision queries for articulated bodies or environments with multiple moving/deformable object need to be performed efficiently. In FCL, this is handled by the CollisionManager, which uses the two-object collision query and the articulated body information to perform N-body collision checking using SaP algorithms.

### A. Object Representation

The object representation component in FCL involves modeling the objects in a hierarchical data structure so that collision and proximity algorithms can be performed efficiently. FCL supports objects in the form of unstructured triangle meshes/soups and basic geometric shapes, such as sphere and cylinder, which are widely used in many robotics applications. FCL can also handle deformable models.

FCL supports seven types of basic geometric shapes, including sphere, box, cone, cylinder, capsule, convex mesh and plane, which are represented as a hierarchy with one node. All these shapes implement two interfaces: 1) overlap, which checks for overlap between the bounding volume of the geometric shape with a bounding volume corresponding to a node in the BVH of some other hierarchy. It is used to perform culling operations. 2) intersect, which checks for exact intersection between the geometric objects or the triangle/point primitives of the other object.

The unstructured mesh/soup is represented as a bounding volume hierarchy and the specific type of bounding volume is specified as a template parameter. In FCL, four BV types are currently supported: AABB, OBB, RSS and kDOP. FCL includes the routines to perform overlap and distance queries using BVs. Each BV is suitable for a different kind of application. For example, OBB is regarded as a tight fitting bounding volume to the underlying shape or primitives, but performing an overlap test using OBB is more expensive as compared to AABB or kDOPs [9]. RSS is regarded as the most efficient primitive to perform separation distance computation [13]. OBB and RSS are considered as more efficient in terms of performing CCD between rigid models, because the BVH structure is unchanged during the motion and therefore we only need to update the transformation matrix associated with each BV and avoid the time-consuming refitting procedure. kDOP and AABB are more suitable for CCD between deformable models, because the cost of refitting these BVHs is relatively lower as compared to OBB or RSS.

The BVH structure in FCL stores both the vertices and triangle information of the underlying object, though the triangle information is not used when representing point clouds. In order to perform CCD computations, the BVH structure also keeps track of the position of the vertices from the last time step. As a result, the BVH structure can handle triangle meshes/soups and point clouds in a consistent manner.

FCL provides functions to help construct the hierarchical representation for a given object, which maintain a state machine of the construction process in order to guarantee the output is a valid BVH structure. As shown in Figure 3, the state machine consists of three parts. The first is the standard way to construct a BVH: we start from an empty BVH and construct a valid BVH structure by adding vertices/triangles into it (empty → building → built in Figure 3). However, in many applications (e.g., the local planning in motion planning), there exists spatial coherence between adjacent
collision queries. In this case, we keep the structure of a BVH unchanged and only update the positions of the triangles associated with the leaf nodes and the intermediate BVs (i.e., refitting the BVH). Therefore, we provide a second method to replace the BVH: we start from a valid BVH, replace the object geometry representation by a new geometric representation and finally compute a valid BVH (built → replacing → built in Figure 3). In order to perform CCD queries, the BVH needs to provide a conservative bound for the swept volume of an object generated during a given time interval. Therefore, we update the BVH in order to consider the motion information and obtain a valid BVH for CCD (built → updating → updated in Figure 3). When the underlying topology of the object (i.e., the triangle information) changes, we empty the BVH state and build it from scratch (built → empty or updated → empty in Figure 3).

The BVH construction or refitting recursively split the underlying geometry primitives into two parts and fit a tight bounding box to it. There are different approaches to performing split and fit operations and it is useful to provide flexibility so that users can choose or implement the split and fit operations based on the underlying application. Therefore, our BVH structure composes the base classes to perform split and fit operations and we also provide default implementations of the splitter and fitter.

Note that object representation may influence the collision results. For example, assuming that A is a small box within a large cylinder B, a collision query will return collision if both A and B are represented as polytopes because polytopes model objects as solid shapes. However, if both A and B are represented as triangle meshes, a collision query will return collision free because a triangle mesh only models the object surfaces. Such differences in collision results caused by object representation occur rarely since configurations like the one where cylinder B contains A seldom happen. In general, most objects always start from collision-free states. However, these differences may cause problems in simulation.

B. Traversal Node Initialization

Given two bounding volume hierarchies, the collision or proximity computation between them is usually performed by traversing the bounding volume test tree (BVTT) constructed from the two BVHs [9]. Different collision or proximity computation algorithms tend to use different traversal schemes though the traversal framework is the same among different algorithms. In FCL, we separate the hardly changed traversal framework from the traversal data and traversal strategies used during traversal, which are different for various algorithms. The advantage of such design is that when implementing a new collision algorithm, developers only need to implement the new traversal data and traversal strategies instead of implementing the whole collision framework from scratch.

In FCL, we use the traversal node to provide all the necessary information in order to access BVH hierarchy structure and determine traversal orders. As shown in Figure 4, all traversal node types are derived from one base classTraversalNode. Among the routines provided by TraversalNode, getFirstOverSecond is used to determine traversal orders, i.e., which subtree of BVTT to traverse. All other routines provide the necessary information for traversing BVTT’s tree hierarchy. Notice that these interfaces also work for polytopes because they are recognized as special BV with a single BV node. For example, if the collision is between a mesh and a cylinder, then we only need to let isSecondNodeLeaf and firstOverSecond always return true.

Two subclasses CollisionTraversalNode and DistanceTraversalNode define the routines for collision and proximity computation, respectively. BVTesting checks the intersection between two bounding volumes and leafTesting checks the intersection between primitives (e.g., triangles or point clouds). canStop determines whether the collision is found or the distance is computed and we can stop the recursion.

CollisionTraversalNode has three subclasses to handle collision between BVHs, collision between BVH and basic shape and collision between basic shapes, respectively. The leaf classes are traversal node types for different collision algorithms, such as continuous collision, point cloud collision, etc. When bounding volume type is OBB and RSS, we provide more efficient implementations which exploit special properties of the two BV types.

Currently, FCL only provides proximity computation between triangle meshes/soups. We also provide conservative advancement traversal node which inherits from the BVHDistance node.

C. Traversal Recurse

The recursive traversal framework is identical among the different algorithms. We provide different recursive approaches which can cover all the algorithms described above. The first is the recursive algorithm for collision, as shown in Algo 1, whose input should be any traversal node derived from CollisionTraversalNode. The second is the recursive algorithm for proximity computation, as shown in Algo 2, which needs input traversal node to be derived from DistanceTraversalNode. We also provide self-collision recurse which is similar to collision recurse.

The traversal recursive framework in FCL also provides a technique called front list, which can accelerate the collision and proximity queries when there exists spatial coherence among queries. Intuitively, the front list is a set of internal and leaf nodes of the BVTT hierarchy where the traversal terminates while performing a collision query during a given time instance. The front list reflects how much of the tree is traversed for each instance of the collision query. For collision queries with spatial coherence, their front lists will be quite similar. Therefore, instead of starting each query from BVTT root node, we can only perform the first query by starting from BVTT root and then start the following queries from its previous query’s front list.
In this section we show the performance of FCL in simulation as well as on real robot applications.

A. Basic Queries

The timing of different queries for a moving piano in the room are shown in Table II. In the results, we can find distance query and CCD query are both slower than DCD query.

B. Collision Checking between Moving Objects

We can check the collision between moving objects by performing a single CCD and by performing multiple DCDs on multiple interpolated configurations. As shown in Table III, CCD-based moving object collision is faster than DCD-based, though a single DCD query is cheaper than a single CCD query.

C. Collision Checking between Deformable Objects

We also test FCL’s performance on two widely used deformable benchmarks (Figure 5) and the performance is shown in Table IV. We also compare performance of CCD-based and DCD based methods.

D. Collision Checking between Point Clouds

FCL supports an experimental algorithm of performing collision query between point clouds or point cloud and meshes [19]. We show its performance on the piano in a room benchmark in Table V, where it is 10 times slower than mesh-based DCD because it uses expensive operations based on support vector machine. We also tested it successfully on a large environment with a real PR2 robot, but in that case it was 50-100 times slower than mesh-based DCD.
Algorithm 1: collisionRecurse(node, b1, b2, front_list), node must derive from CollisionTraversalNode.

1 begin
2    l1 ← node.isFirstNodeLeaf(b1)
3    l2 ← node.isSecondNodeLeaf(b2)
4 if l1 and l2 then
5      updateFrontList(front_list, b1, b2)
6 if node.BVTesting(b1, b2) then
7      return
8 node.leafTesting(b1, b2)
9 return
10 if node.BVTesting(b1, b2) then
11      updateFrontList(front_list, b1, b2)
12 return
13 if node.firstOverSecond(b1, b2) then
14      c1 ← node.getFirstLeftChild(b1)
15      c2 ← node.getFirstRightChild(b1)
16      collisionRecurse(node, c1, b2, front_list)
17 if node.canStop() and !front_list then
18      collisionRecurse(node, c2, b2, front_list)
19 else
20      c1 ← node.getSecondLeftChild(b2)
21      c2 ← node.getSecondRightChild(b2)
22      collisionRecurse(node, b1, c1, front_list)
23 if node.canStop() and !front_list then
24      return
25 collisionRecurse(node, b1, c2, front_list)
26 end

Algorithm 2: distanceRecurse(node, b1, b2, front_list), node must derive from DistanceTraversalNode.

1 begin
2    l1 ← node.isFirstNodeLeaf(b1)
3    l2 ← node.isSecondNodeLeaf(b2)
4 if l1 and l2 then
5      updateFrontList(front_list, b1, b2)
6      node.leafTesting(b1, b2)
7 return
8 if node.firstOverSecond(b1, b2) then
9      a1 ← node.getFirstLeftChild(b1); a2 ← b2
10      c1 ← node.getFirstRightChild(b1); c2 ← b2
11 else
12      a1 ← b1; a2 ← node.getSecondLeftChild(b2)
13      c1 ← b1; c2 ← node.getSecondRightChild(b2)
14      d1 ← node.BVTesting(a1, a2)
15      d2 ← node.BVTesting(c1, c2)
16 if d2 < d1 then
17      if node.canStop(d2) then
18        updateFrontList(front_list, d2, c1, c2)
19      else distanceRecurse(node, c1, d2, front_list)
20      if node.canStop(d1) then
21        updateFrontList(front_list, d1, a1, a2)
22      else distanceRecurse(node, a1, d1, front_list)
23 if node.canStop(d1) then
24        updateFrontList(front_list, d1, a1, a2)
25      else distanceRecurse(node, a1, d1, front_list)
26      if node.canStop(d2) then
27        updateFrontList(front_list, d2, c1, c2)
28 else distanceRecurse(node, c1, d2, front_list)
29 end

E. Comparison Tests in ROS

FCL was integrated with the collision checking testing infrastructure in ROS. The infrastructure generates a series of random environments with a hundred objects in random locations. Each environment consists of objects of one type from among the following: box, cylinder, sphere and mesh primitives. The PR2 robot was used as the robot model in this test and the two arms of the robot were posed in a 1000 random configurations in each environment. FCL was compared against the existing collision checking capabilities in ROS which are based on ODE [1]. Table VI presents the aggregate collision checking times (in seconds) for these 1000 queries for FCL and ODE. Although the implementation of collision checking in ODE does not represent the state of the art, it is still among the most popular implementations used by a variety of motion planning packages.

F. Motion Planning for a Real Robot

FCL was further validated by application to a range of problems on the PR2 mobile manipulation robot. The first set of experiments involved motion planning in a simple environment. FCL was integrated as the collision checking library for the motion planners used in this experiment. Figure 6 shows a series of snapshots showing the motion of the robot as it moves one arm around a pole in the environment.

G. Grasping and Manipulation

FCL was further integrated as the collision checking library for the grasping and manipulation pipeline imple-
TABLE V: Point cloud collision query timing (in milliseconds): Each query is performed for 100 random configurations and the average timing of a single query is shown in the table.

<table>
<thead>
<tr>
<th></th>
<th>ODE</th>
<th>FCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boxes</td>
<td>0.780889</td>
<td>0.700815</td>
</tr>
<tr>
<td>Spheres</td>
<td>0.487176</td>
<td>0.570329</td>
</tr>
<tr>
<td>Cylinders</td>
<td>0.236988</td>
<td>0.264515</td>
</tr>
</tbody>
</table>

TABLE VI: Collision checking timing (in seconds) for PR2 robot model posed in a 1000 random configurations in environments with 100 obstacles of a single primitive type in random locations.

<table>
<thead>
<tr>
<th></th>
<th>ODE</th>
<th>FCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boxes</td>
<td>4.079125</td>
<td>4.340178</td>
</tr>
<tr>
<td>Spheres</td>
<td>2.340178</td>
<td>2.40228</td>
</tr>
</tbody>
</table>

the collision map) for other parts of the environment.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduce FCL, a new library for collision and proximity operations. It provides unified interfaces for various algorithms and is convenient to be extended for including state-of-the-arts methods in the future. It has been tested on both simulation and real PR2 environments.

There are many avenues for future work, such as providing more robust and more efficient point cloud collision, testing CCD in planning and grasping pipeline on real robot, and providing good maintenance for the library.

REFERENCES