

Efficient Architecture for Collision Detection between Heterogeneous Data Structures

Application for Vision-Guided Robots

Jesse Himmelstein
Guillaume Ginioux
Etienne Ferré

Kineo CAM
Toulouse, France
{jh, gg, ef}@kineocam.com

Alireza Nakhaei
Florent Lamiroux
Jean-Paul Laumond

LAAS-CNRS, University of Toulouse
Toulouse, France
{anakhaei, florent, jpl}@laas.fr

Abstract—Many collision detection methods exist, each specialized for certain data types under certain constraints. In order to enable rapid development of efficient collision detection procedures, we propose an extensible software architecture that allows for cross-queries between data types, while permitting the time and memory optimizations needed for high-performance. By decomposing collision detection into well-defined algorithmic and data components, we can use the same tree-descent algorithm to execute proximity queries, regardless the data type. We validate our implementation on a path planning problem in which a vision guided humanoid represented by an OBB tree explores a dynamic environment composed of voxel maps.

Keywords— collision detection, software design, robot navigation

I. INTRODUCTION

This paper deals with the integration of multiple kinds of geometric objects into a collision detector in the context of robot motion planning. We do not contribute new collision detection algorithms, but rather a generic framework for evaluating proximity queries between heterogeneous data structures.

A. Motivation

Collision detection plays a critical role in many domains such as robotics, Product Lifecycle Management (PLM), computer graphics, and virtual environment. Applications including mobile robotics, robotic surgery, and humanoid robots are active areas of research that rely heavily on collision detection.

A robot can use path planning along with vision sensors to navigate in unknown environments. A single path planning computation invokes collision and distance tests so frequently that such tests often represent a performance bottleneck. In order to insure its reactivity, a well-optimized collision detector must therefore be integrated into such a robot solution.

Such optimization may be achieved through specialization

of collision detection code to handle a single geometric data representation (or perhaps a small number of them). Through intimate knowledge of the limitations, assumptions, and implications of both a given geometry type and its implementation, the collision detection code can avoid extraneous tests and optimize the remaining ones. If necessary, extra data structures may be built upon the geometry data to accelerate the process even more.

This process tightly binds the collision detection algorithm to the data representation, and becomes a problem once a new geometry type is added to the mix. To consider not only the tests between two objects of the new type, but also between the new and old types, a user of a dedicated collision detector is faced with two imperfect choices: create a second collision detection algorithm specialized for the new collision tests (which must be independently tested and maintained), or convert the new data type to the old one before running the tests.

This second approach may seem appealing, but would in fact sacrifice many of the advantages of the “natural” model representations. In our case, for example, we could convert a voxel map into a triangle soup and then let the existing OBB tree collision routine handle it. But since a bounding volume hierarchy must be rebuilt each time the model changes, frequent updates to the voxel map would impose a significant performance penalty.

We decided upon a third approach, to create a software architecture that supports any number of geometry types as well as all the optimizations necessary to achieve high performance. Applications for this architecture include PLM which deals with polyhedrons and polygon soups, bioinformatics in which atoms can be modeled as spheres, as well as robotics. The last domain is the one that we will develop in this article by applying our framework to detection between voxels and polygon soups.

B. Related Work

The collision detection problem has been extensively studied in many different contexts. The tight performance constraints placed on a collision checker (both in terms of time and space complexity) have lead to specialized collision detection structures and algorithms for many different geometry representations. For a presentation on the state of the art, we refer the reader to [1].

To handle the kind of Product Lifecycle Management (PLM) path planning problems discussed in [2], high performance Oriented Bounding Box (OBB) trees that deal with unstructured polygon soup models are ideal [3-5]. Other bounding volumes that would work well include AABB trees [6, 7] and k-DOPs [8]. Since high-fidelity polygon soup models are available for the mobile robot and certain obstacles in the robot’s environment, it is also a natural choice for the “static” portion of the robot’s collision detection solution.

The autonomous humanoid robot uses stereo vision and occupancy grids to construct a unified 3D environment [16, 17]. For dynamic environments, voxel maps provide very fast collision detection [9]. By organizing them hierarchically, such a map can be dynamically updated in a memory and time efficient manner while the robot explores its environment [10].

C. Contributions

Our central contributions are the following:

- Defining an algorithm that descends a pair of trees in tandem in order to perform generic collision detection, while dispatching all concrete proximity tests to specialized handlers.
- Creating an extensible hierarchical structure suitable for space-partitioning and bounding-volume methods that allows for custom optimized data structures.

D. Outline

We begin in Section II by introducing a general collision detection procedure and describe how our framework implements that procedure. To discuss the framework itself in more detail, we describe the tree descent algorithm in Section III, and the geometrical tree structure in Section IV.

In Section V, we validate our work on a vision-guided humanoid robot that explores dynamic environments using a combination of voxel and polyhedral models. Finally, we conclude with opportunities for future work in Section VI.

II. COLLISION DETECTION ALGORITHM ANALYSIS

By analyzing a typical example of collision detection in practice, we can identify the essential procedures and data structures needed by a generic framework.

A. Collision Detection Example

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

As described in [5, 11, 12], to avoid testing each triangle from *A* against each from *B*, bounding volumes can be placed around each set of triangles. In this example, OBBs are used. Only if the OBBs of *A* and *B* overlap do we need to test their triangles against each other. Algorithm 1 describes such a function, and Figure 1 lays out the tests needed if *A* has 3 triangles and *B* only 2.

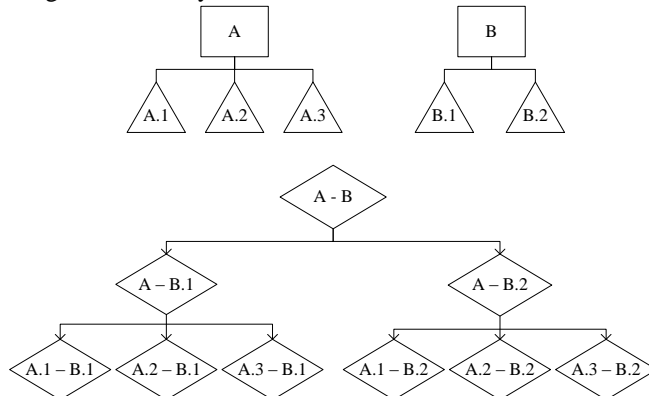


Figure 1 Testing polyhedrons. Take two polyhedrons, *A* and *B*, the first composed of 3 triangles and the second only 2. Given a bounding volume around each polyhedron, it is possible to reduce the number of tests that must be executed in order to test for collision. First, the bounding volumes are tested against each other. If they overlap, then one of them (e.g. *A*) is tested against each triangle of the other. Only if an overlap is detected again, are the triangles from *A* tested against the triangle of *B*. The execution can be represented by the bottom flow diagram.

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

```

function test(a, b) : Boolean
  Boolean collides = false
  if overlaps(a, b)
    if isOBB(b)
      foreach c in children(b)
        collides = collides or test(a, c)
      end foreach
    else if isOBB(a)
      foreach c in children(a)
        collides = collides or test(c, b)
      end foreach
    else
      // both a and b are triangles
      collides = true
    end if
  end if

  return collides
end function
  
```

Algorithm 1 Simple polyhedron-polyhedron collision procedure. The types of *a* and *b* could be OBBs or triangles.

It is easy to see that in this reduced example we need three kinds of tests: OBB-OBB, OBB-triangle, and triangle-triangle. The overlaps() function referenced in Algorithm 1 would

need to distinguish between them in order to carry out the correct calculation.

B. Tree Structure

Although the example just given applies only to polygon soups, many different geometry types employ hierarchical structures to carry out collision detection. In general, all bounding-volume and spatial partitioning techniques use a divide and conquer strategy [1]. Both bottom-up and top-down approaches result in the same type of hierarchical structure.

In our framework, this hierarchical structure is termed a *test tree* whose elements can be leaves (triangles, in the previous example) or branches (OBBs). In order to support a wide range of geometries, a branch can have any number of children. Test trees are discussed further in Section IV.

C. Comparison and Descent

Given the generic tree structure, it is possible to generalize the procedure presented in Algorithm 1. First, two elements are tested against each other. If no overlap is detected, then the function returns `false`. If the elements do overlap and they are both leaves (e.g. collision between two triangles), then the procedure simply returns `true`. Otherwise, it is necessary to explore further in the tree to determine if a collision exists. One of the two elements is chosen for expansion, and the procedure is called recursively for each of its children, or until a collision is found.

We call this algorithm *test tree descent* and it forms the core of our framework. Section III describes it in greater detail.

D. Proximity Queries

In the previous example, we have only checked if *A* and *B* collide. Other common tasks include listing all of the collisions between *A* and *B* (i.e. a list of pairs of overlapping triangles), and finding the distance between *A* and *B* if they do not collide.

Each of these tasks is an example of a *proximity query*, and modifies not only the test tree descent but also the kind of information returned by proximity tests between the elements. In the previous example, only a simple overlap test was required between elements, which might be faster than calculating the distance between them.

E. Framework Architecture

As a whole, the architecture of the framework can be decomposed into three parts (Figure 2). The core is the test tree descent algorithm, which is immutable and applies equally to test trees of any geometry. It uses the test tree element interface to traverse the test trees.

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

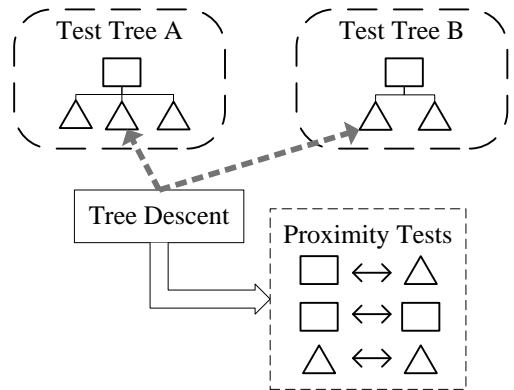


Figure 2 Framework architecture. The tree descent algorithm always references two elements at a time, one from each test tree. To test for collision and distances between elements, it calls on one of a number of proximity tests, organized into a bank. Each proximity test is specialized to analyze the interaction between two types of geometric objects. In the diagram, the dashed borders around the test trees and the proximity tests are used to indicate that the user can extend those portions of the architecture.

III. TEST TREE DESCENT

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

A. Dispatch and detection

Given two test tree elements, we must be able to analyze their interaction for collision and distance results. Since the test tree elements do not necessarily have knowledge of each other, this is an example of a multiple dispatch problem. Approaches to resolve this problem vary by programming language. For C++, [13] discusses it and proposes several new solutions.

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

Our solution is based on a simple function table, indexed by element type (Figure 3), that dispatches proximity tests at runtime. The user can register and unregister proximity test objects with the dispatcher at runtime. For any pair of test tree elements, a single lookup retrieves the address of the object whose virtual function handles the given pair. Since C++ provides only weak support for reflection, a virtual method was added to the test tree element interface that receives a unique identifier, assigned by the dispatcher upon registration.

B. Tree descent

Tree descent follows a simple recursive pattern, addressing

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

The generality of our framework derives the fact that the tree descent algorithm is wholly ignorant of the type of data it is dealing with. All special knowledge of the geometry concerned is handled by the proximity tests, which can themselves be dynamically substituted for each other at run time.

For a more complete description, Algorithm 2 lists pseudo-code for the procedure.

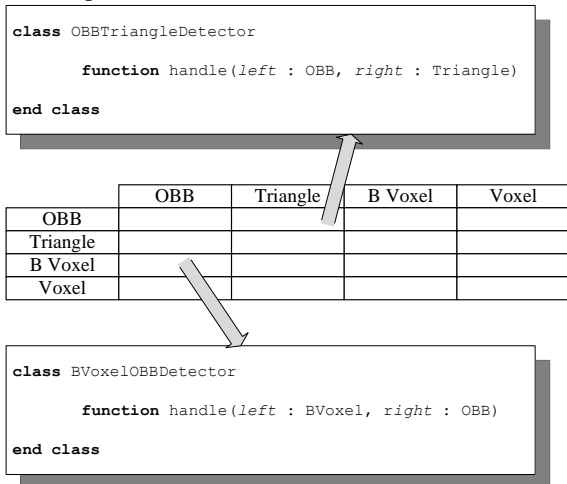


Figure 3 Proximity test dispatch mechanism. The dispatcher (center) has a table of proximity tests, organized by the type of test tree elements that a test can handle. To compare two elements, the dispatcher looks up the two types in the table and calls the corresponding virtual function on the selected proximity test. In this example, the `OBOTriangleDetector` (top) handles an OBB as the left element and a polyhedron triangle as the right. The `BVoxelOBOTDetector` (bottom) does the same for bounding voxels and OBBs.

IV. TEST TREE STRUCTURE

Test tree elements must implement a common interface to allow the descent algorithm to traverse the tree. They also must provide specialized information to the proximity tests that handle them.

A. Generic traversal

Examination of the pseudo-code in Algorithm 2 reveals that only one element from each test tree is being compared at once. Additionally, the test trees are not traversed in a random fashion. Instead, the algorithm starts at the root nodes and then either references a element’s first child or a element’s next sibling. We can exploit this restricted access pattern to allow for time and memory optimizations.

We define a simple element interface that only allows three methods for tree traversal to access other elements: `firstChild()`, `nextSibling()`, and `parent()`. Their meanings are illustrated in Figure 4. Each method returns a reference to another element, which is used from then on. The

methods `hasChildren()` and `hasNextSibling()` simply provide information about the existence of an element’s relations.

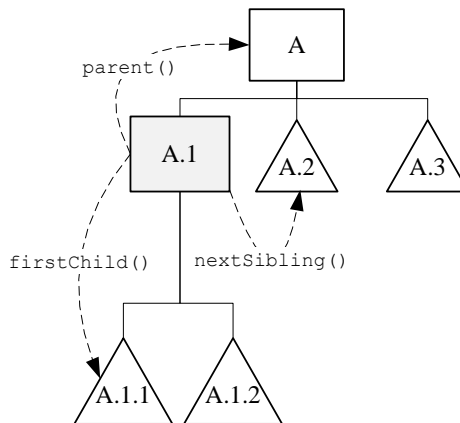


Figure 4 Element relations and traversal methods. In the restricted tree traversal pattern used, the highlighted element, `A.1`, is linked to other elements in the tree only through three relations: `parent()`, `firstChild()`, and `nextSibling()`.

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

B. Memory Optimization

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

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

To illustrate, consider once again the OBB tree example. Upon construction, the triangles are sorted by a space-partitioning algorithm to build a binary tree of OBBs. To optimize memory, the OBBs are stored in a large array, with compressed descriptions of their positions and indexes of their children. Triangles are stored in a similar fashion.

With this setup, the OBB and triangle elements can simply store the indices relative to their respective arrays. When a traversal method is called, the element can look up the data at that index, unpacking it if necessary, and then answer the request.

In order to save memory, the test tree used in our example constructs only one OBB element and one triangle element. Instead of always returning a reference to another element object in response to a traversal method call, these “singleton” elements only return references to themselves or to each other.

Consider what happens when `firstChild()` is called on an OBB element. If the child is also an OBB, then the element only needs to update its own internal reference and simply return a reference to itself. If the child is a triangle, then it retrieves the triangle element of the test tree and updates the triangle’s internal data before returning a reference to it. The other two tree traversal methods, `nextSibling()` and `parent()`, are implemented in a similar fashion.

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

V. DYNAMIC VOXEL MAP FOR ROBOTIC VISION SYSTEM

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

We conducted an experiment involving HPR-2 (Figure 5), a humanoid robot [14, 15]. The goal is to allow the robot to autonomously explore unknown environments using stereo vision and probabilistic path planning techniques. By testing a hybrid collision detection scheme against a homogenous polyhedral one, we can demonstrate the effectiveness of our approach.



Figure 5 The HPR-2 humanoid robot.

A. 3D Reconstruction

To represent the environment, we use a 3D-occupancy grid which is frequently refreshed with information from HPR-2’s cameras [16, 17]. Each grid cell is assigned a triplet representing the probabilities that it contains an obstacle, free space, or is indeterminate. Originally, all cells are considered indeterminate. Based on these values, the grid cells are classified into one of three discrete categories: OBSTACLE,

UNKNOWN or FREE (Figure 6). The UNKNOWN category may refer to a cell that the robot has not yet observed (e.g. it is behind an obstacle) or is unsure about (e.g. not enough 3D points have been gathered in that volume).

As the robot moves, it gradually discovers more of the environment around it. Additionally, the environment itself may change. In both cases, the robot takes the new information into account in the 3D model.

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

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

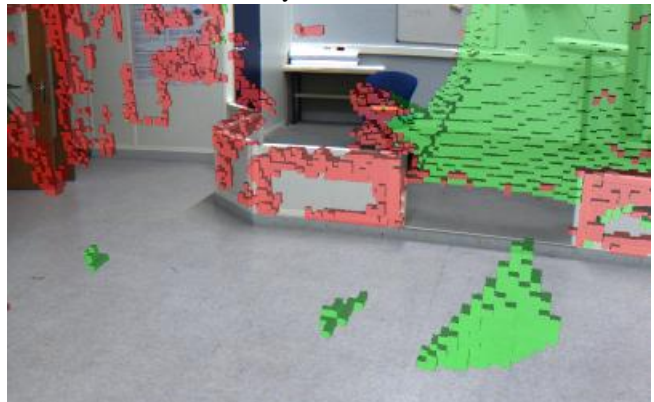


Figure 6 Occupancy grid. Data from several stereo images are combined to calculate the probabilities that a grid cell contains an obstacle. Here, the calculated grid cells are superimposed upon an acquired image. The red cells represent obstacles, and the green cells are unknown. The absence of cells indicates that the space is considered free.

B. Dynamic Voxel Map

A natural representation of the 3D occupancy grid is a voxel map. Such a space-partitioning method can benefit from a hierarchical organization, with larger voxels bounding a predetermined set of smaller ones [10].

One advantage of voxel maps is the efficient manner in which they can be updated. Removing or adding a voxel at the lowest level sends a message to the parent voxel informing of a change. In this way, bounding voxels can be created and removed on the fly in order to assure consistency (Figure 7).

Just as polyhedron objects in the scene tree may be transformed into an OBB tree for collision detection, so can voxel map objects be transformed into a voxel map test tree. However, unlike the OBB tree construction process, the voxel

map test tree requires no pre-computation. Since it merely references the included voxel maps, it does not need to be rebuilt each time a change occurs.

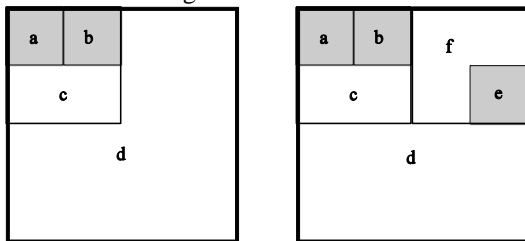


Figure 7 Voxel hierarchy. On the left, two primitive voxels (*a* and *b*) were detected by the vision system. Each level of the hierarchy (here there are three) contains a larger voxel bounding those below it- *c* contains both primitives on the second level and *d* contains *c* on the third. On the right, a third primitive voxel *e* is detected. Automatically, the bounding voxel *f* is created around it on the second level. No modification on the third level of the hierarchy is needed in this case.

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

C. Experimental Design

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

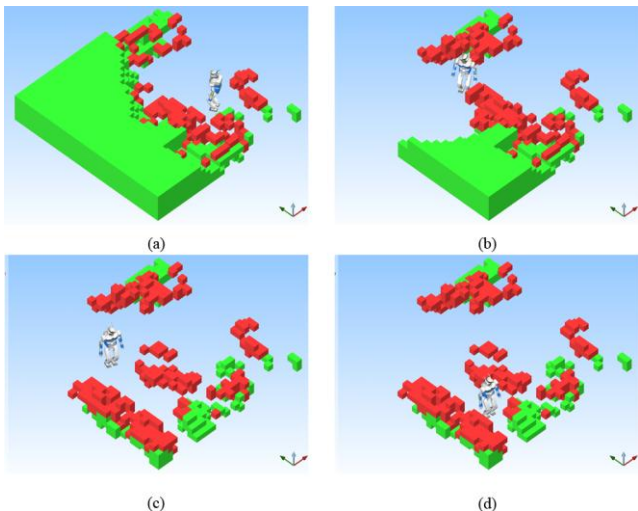


Figure 8 Environment discovery in three steps. When HPR-2 is placed at its initial position, it has no prior information of the environment. It builds a tentative representation of the environment from the acquired image data (a), and moves towards the boundary of the UNKNOWN area (green), while avoiding obstacles (red). Once there, it refines its estimation of the environment (b), and proceeds with the second iteration of its exploration algorithm. After this movement, it can perceive a clear path reaching the goal (c). By moving there (d), it completes the task.

The robot was placed in a typical exploration scenario, in

which it maneuvers around an obstacle in order to reach its goal position (the environment measures 6x6x1m). Since the obstacle partially blocks the robot’s view of the scene, it takes three iterations of the exploration process in order for the robot to complete its task.

In order to get consistent results for the collision detection performance, we pre-converted the image data into occupancy grid cells (20cm³) for each of the three iterations. We measured the time and memory taken for each collision detector to process and initialize the occupancy grid data. Finally, we measured the time taken per collision test during the path planning process.

D. Results

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

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

TABLE I. TIME PERFORMANCE

| Iteration | Grid Cells | | Type | Setup Time (ms) | Time per Collision Test (ms) |
|-----------|------------|---------|-------|-----------------|------------------------------|
| | Obstacle | Unknown | | | |
| 1 | 147 | 2332 | Voxel | 20 | 0.924 |
| | | | Poly | 65,761 | 1.083 |
| 2 | 202 | 1012 | Voxel | 10 | 0.779 |
| | | | Poly | 17,626 | 1.292 |
| 3 | 284 | 306 | Voxel | 7 | 0.547 |
| | | | Poly | 5,304 | 1.712 |

TABLE II. MEMORY CONSUMPTION

| Iteration | Grid Cells | | Type | Memory (kB) |
|-----------|------------|---------|---------|-------------|
| | Obstacle | Unknown | | |
| 1 | 147 | 2332 | Voxel | 74 |
| | | | Polygon | 111 |
| 2 | 202 | 1012 | Voxel | 73 |
| | | | Polygon | 92 |
| 3 | 284 | 306 | Voxel | 73 |
| | | | Polygon | 83 |

The setup time gap only widens as the number of voxels grows. Cutting the occupancy grid size in half (from 20cm to 10cm), for example, took the polyhedral collision detector 9.7 minutes to process, against 38.1 ms for the voxel map. Since this level of performance is unacceptable for our application, we didn't conduct further tests at this resolution.

VI. CONCLUSION AND FUTURE WORK

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

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

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

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

REFERENCES

- [1] M. C. Lin and D. Manocha, "Collision And Proximity Queries," in *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, Eds. Boca Raton, FL, USA: CRC Press, 2004, pp. 787-808.
- [2] J.-P. Laumond, "Motion planning for PLM: state of the art and perspectives," *International Journal of Product Lifecycle Management*, vol. 1, pp. 129-142, 2006.
- [3] S. Gottschalk, M. C. Lin, and D. Manocha, "OBbTree: a hierarchical structure for rapid interference detection," in *Proceedings of Computer Graphics and Interactive Techniques*, 1996, pp. 171-180.
- [4] S. Gottschalk, "Collision Queries using Oriented Bounding Boxes," in *Department of Computer Science: University of North Carolina*, 1998.
- [5] M. Lin, D. Manocha, J. Cohen, and S. Gottschalk, "Collision detection: Algorithms and applications," in *Algorithms for Robotics Motion and Manipulation: 1996 Workshop on the Algorithmic Foundations of Robotics*, J.-P. Laumond and M. Overmars, Eds.: A K Peters, Ltd., 1996, pp. 129-142.
- [6] G. v. d. Bergen, "Efficient Collision Detection of Complex Deformable Models using AABB Trees," *Journal of Graphics Tools*, vol. 2, pp. 1-13, 1997.
- [7] T. Larsson and T. Akenine-Möller, "A Dynamic Bounding Volume Hierarchy for Generalized Collision Detection," in *Proceedings of Workshop On Virtual Reality Interaction and Physical Simulation*, 2005.
- [8] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs," in *Proceedings of Visualization and Computer Graphics*, 1998, pp. 21-36.
- [9] S. F. Gibson, "Beyond Volume Rendering: Visualization, Haptic Exploration, and Physical Modeling of Voxel-based Objects," in *Visualization in Scientific Computing '95*, R. Scanteni, J. v. Wijk, and P. Zanarini, Eds.: Springer-Verlag Wien, 1995, pp. 9-24.
- [10] W. A. McNeely, K. D. Puterbaugh, and J. J. Troy, "Six degree-of-freedom haptic rendering using voxel sampling," in *Proceedings of Computer Graphics and Interactive Techniques*, 1999 pp. 401-408.
- [11] S. M. LaValle, *Planning Algorithms*: Cambridge University Press, 2006.
- [12] S. Quinlan, "Efficient Distance Computation between Non-Convex Objects," in *Proceedings of International Conference on Robotics and Automation*, 1994.
- [13] C. Pescio, "Multiple Dispatch: A new approach using templates and RTTI," *C++ Report*, 1998.
- [14] T. Inamura, K. Okada, M. Inaba, and H. Inoue, "HRP-2W: A Humanoid Platform for Research on Support Behavior in Daily life Environments," in *Proceedings of International Conference on Intelligent Autonomous Systems*, 2006, pp. 732-739.
- [15] K. Yokoi, N. E. Sian, T. Sakaguchi, O. Stasse, Y. Kawai, and K.-i. Maruyama, "Humanoid Robot HRP-2 with Human Supervision," in *Experimental Robotics*, vol. 39, *Springer Tracts in Advanced Robotics*. Berlin: Springer, 2008, pp. 513-522.
- [16] C. Braillon, C. Pradalier, K. Usher, J. Crowley, and C. Laugier, "Occupancy grids from stereo and optical flow data," in *Experimental Robotics*, vol. 39, *Springer Tracts in Advanced Robotics*. Berlin: Springer, 2008, pp. 367-376.
- [17] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," in *Computer*, vol. 22, 1989 pp. 46-57.

APPENDIX

```

function startTest(treeA : TestTree, treeB : TestTree) : void
  test(root(treeA), root(treeB))
end function

function test(a : ElementRef, b : ElementRef) : void
  detect(element(a), element(b))

  Boolean childrenExist = hasChildren(element(a)) or hasChildren(element(b))
  if childrenExist and shouldContinue() and shouldDescend()
    ElementRef c
    if shouldDescendLeft(element(a), element(b))
      c = a
    else
      c = b
    end

    element(c) = firstChild(element(c))

    test(a, b)

    while shouldContinue() and hasNextSibling(element(c))
      element(c) = nextSibling(element(c))
      test(a, b)
    end while

    element(c) = parent(element(c))
  end if
end function

function shouldContinue() : Boolean
  return result() is not COLLISION or query() is not EXHAUSTIVE_COLLISION
end function

function shouldDescend() : Boolean
  return result() is OVERLAP or (query() is EXACT_DISTANCE and
    result() is WITHIN_MIN_DISTANCE)
end function

function shouldDescendLeft(a : Element, b : Element) : Boolean
  if not hasChildren(a)
    return false
  else if not hasChildren(b)
    return true
  else
    return heuristic(a) > heuristic(b)
  end if
end function

```

Algorithm 2 Generic tree descent algorithm. The function `test()` is recursive, and is originally called the root elements of the two test trees. The `detect()` function dispatches the proximity test to the proper handler. `ElementRef` is a mutable reference to the element, so that multiple copies of the reference all point to the same element, retrieved with the `element()` function. Finally, `query()` returns the current proximity query and `result()` the last proximity test result.