4
Proximity Queries

One of the most important properties of paths is that we should not have any collision while following those paths. So far, we assumed that one can perform collision detection and use that information for computing collision-free paths. In this chapter, we discuss fundamental techniques for checking collision detection and other proximity queries that are related to motion planning.

In most planners, we use the following two proximity queries:

- Clear (q). Check whether a robot at a configuration q has collisions or not.

- Link (q, q'). Is the straight line between two configurations q and q' collision free?

To answer these proximity queries, we commonly use collision detection and distance computation between two objects. These two techniques share a lot of similar components, mainly because collision detection can be reduced to the case of running the distance computation and checking the distance to be zero.

Collision detection can be classified into discrete and continuous collision detection. Discrete collision detection checks collision between a robot and its environment only at discrete time steps. It tends to be fast and simple to be implemented. As a result, this discrete collision detection is much widely used in robotics and other related applications including games. On the other hand, continuous collision detection checks any collisions while the robot is continuous moved during a time period, instead of checking at discrete time steps.

Discrete collision detection is fast and thus widely used, while continuous collision detection guarantees not to miss collisions.

The discrete collision detection may miss collisions, especially when the robot moves so fast. On the other hand, continuous collision detection can detect collisions even at that case. Unfortunately, continuous collision detection requires much higher computational power over the discrete one, since it needs to consider continuous motions. Since continuous collision detection can provide robust and higher accuracy in terms of detecting collision, there have been research demands on accelerating its performance [1]. Since discrete collision detection is more widely used for now, we simply use collision detection to denote discrete collision detection unless mentioned otherwise.

[1] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-Eui Yoon. HPCCD: Hybrid parallel continuous collision detection. *Comput. Graph. Forum (Pacific Graphics)*, 28(7), 2009

Let us now talk about the Link (q, q') operator. A simple way of performing the link operator is to use collision detection at discrete steps on the straight line between two configurations (Fig. 4.1). In other words, we sample discrete configurations on the straight line and check whether we have collisions or not at those discrete configurations. If we have at least one collision, we report failure for that link operator.

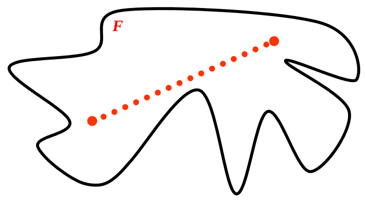

Figure 4.1: One can check collisions at discrete positions on the straight line given by a link operator. This approach has a potential to miss collision.

Instead of using the discrete collision detection mentioned above, we can also use continuous collision or distance computation. Here we discuss a way of using distance computation to prevent missing any collisions on the straight line given by the link operator.

Its main idea utilizes to identify nearest neighbor from the robot to the environment. Once we can compute the nearest neighbor, we can compute the minimum distance from the robot to the environment. As a result, minimum distance computation is also known as nearest neighbor search. The overall procedure for performing Link (q_0, q_1) based on this approach is shown as follow:

- **Check whether two configurations are free space.** Check whether q_0 is within the minimum distance from q_1 to the environment by performing the minimum distance computation. We do the similar operations for q_1. If so, we return true, since we can guarantee that we can reach one configuration from the other one without having any collisions.

- **Divide-and-conquer.** When the above case is not passed, we now consider the middle point, q', between q_0 and q_1, and check whether we have collision on the middle point. If so, we return false. Otherwise, we recursively test Link (q_0, q') and Link (q', q_1).

The algorithm of computing the minimum distance is discussed in Ch. 4.2.

4.1 *Collision Detection*

We start from a simple case for detecting collisions between a point and a triangular model, i.e., mesh. We also assume that the collision arises in the boundary of models, and this assumption can be easily lifted later. In a naive setting, we need to check a collision between

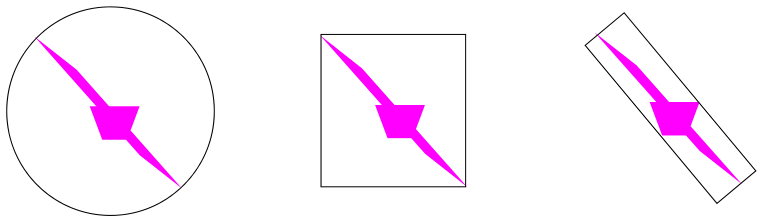

Figure 4.2: From the left, sphere, axis-aligned box, and oriented box are used as bounding volumes for the model.

the point and every triangle of the model, which can be very time consuming.

A common way of accelerating this process is to use a bounding volume (BV) and a hierarchy of bounding volumes, i.e., bounding volume hierarchy (BVH) of the model. A bounding volume is a simple geometric primitive that can enclose the underlying objects (e.g., triangles, points, etc.). Some of commonly used BVs are shown in Fig. 4.2.

Bounding volumes should be easy to be tested for check collisions, while tightly bounding the underlying objects.

One of the most simple BV type is sphere and we can very easily test whether the point has collision against the sphere, by simply measuring the distance between the point and the center of sphere. However, the sphere may not be tightly bounding the underlying objects, as shown in Fig. 4.2. Tightly bounding the model is important, since when there is a collision between the point and the BV, we have to check the collision between the point and the underlying object for computing the collision detection. In other words, as the BV is loosely enclosing the model, there is a higher probability that we have to check the exact level collision between the point and model, even after checking between the point and BV.

Another simple BV types are axis-aligned boxes or oriented boxes. As seen in the figure, as we use such boxes, we may be able to more tightly bound the object, but computing oriented boxes requires more computations. We cannot say which one is better than others in general, but depending on your models, one BV type can be better to others in terms of performance of checking collisions.

Typically, the model has many of geometric primitives, e.g., triangles or points, and thus using a few BVs for enclosing them may not provide enough culling power and performance. As a result, we use a bounding volume hierarchy. Fig. 4.3 shows input triangles on the left and also shows a computed BVH using axis-aligned bounding boxes. We build such BVHs in a hierarchical manner. Specifically, we build a box from all the geometry and partition triangles into the left and right nodes; there can be many choices for partitioning triangles, but one simple choice is to use a middle plane (or line) breaking

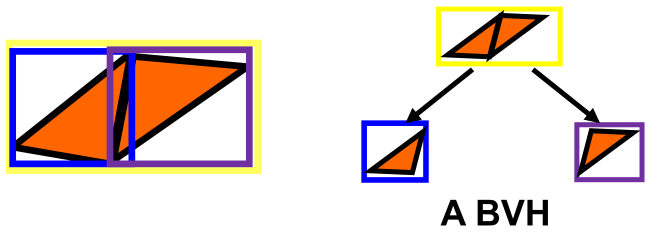

Figure 4.3: On the right, we show a BVH using axis-aligned bounding boxes that are computed hierarchically from the triangles shown on the left.

the longest edge of the box and partition triangles against the plane. We recursively perform this process until each node contains a few geometric primitives.

Bounding volume hierarchies are commonly used for various geometric computing including collision detection and many other applications.

Using the BVH for checking any collisions can be done hierarchically by traversing the BVH in the top-down manner. Starting from the root node of the BVH, we check whether there is collision between the node and the point under the test. If there is no collision, we simply stop the traversal. Otherwise, we refine the node into its two child nodes and recursively perform the collision tests on those two child nodes.

The concept of using BVs and BVHs is very widely used in many proximity computing including distance computation and ray tracing to name a few [2].

[2] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs. In *IEEE Symp. on Interactive Ray Tracing*, pages 39–46, 2006

4.2 *Minimum Distance Computation*

We used the minimum distance computation for performing the link operator that checks a collision-free path between two configurations (Ch. 4.1). The minimum distance computation is also known as nearest neighbor search, which is also very frequently used in many other applications including motion planning and image search [3].

[3] J. Heo, Z. Lin, and S. Yoon. Distance encoded product quantization for approximate k-nearest neighbor search in high-dimensional space. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(9), 2019

In this section, we briefly explain how to compute the minimum distance between a point to a triangular model using its BVH using spheres. At a high level, the minimum distance value, d_m, is initialized by the infinite value. We then traverse the BVH in a depth-first manner and prune nodes that cannot minimize the current minimum distance value d_m for accelerating the process.

Suppose that we refine the green node as shown in Fig. 4.4-a). The distance between the point and the sphere of the green node, 40, is smaller than d_m, which was set ∞. Note that the distance with the node is not reflected into d_m, since that distance is not measured by one of actual primitives of the model. We then refine its child node, which is a green leaf node shown in b), whose distance to the point is smaller than the d_m. We also recursively refine its node, and thus we fetch its triangle, as shown in c). We then measure the distance between the triangle and the point, which is 42, smaller than the

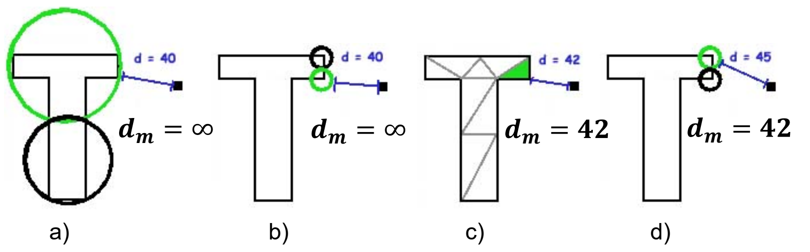

Figure 4.4: This shows the process of traversing the BVH for computing the minimum distance.

current d_m. As a result, we update d_m by 42, which is the distance measured by one of actual primitives. In d), we traverse another node, but whose distance to the point, 45, is higher than d_m, and thus we cull to traverse the node, since there is no chances to find a closer primitive than the current minimum distance 42. We recursively perform this process, until we traverse all the nodes.

Let us discuss the time complexity of building the BVH and computing the minimum distance. For building the BVH, let us assume that the computed tree is balanced. In this case, we have $O(\log n)$ tree height, where n is the number of primitives of the model. At each level, we have to go through all the primitives and partition them into left or right nodes. As a result, the time complexity of building the tree is $O(n \log n)$.

For using the tree at test time, we may have to traverse all the nodes in the tree at the worse case, resulting in $O(n)$ time complexity, assuming that no nodes are pruned. However, in most cases, we simply need to traverse up to some leaf nodes and prune most of other nodes, which can result in $O(\log n)$ time complexity. Overall, the time complexity at the test time can vary between $O(\log n)$ and $O(n)$, depending on the geometric configurations between obstacles and the robot.

Point clouds. So far, we discussed various techniques assuming that the environment is represented by polygonal shapes. In practice, the environment is commonly represented by a set of points, i.e., point clouds, since various sensors (e.g., RGB-Depth sensor or LIDAR sensors) provide sensor data as a set of points associated with depths or RGB colors. Most motion planning techniques do not change irrespective of input types of the obstacles, but there are better techniques for point clouds related to collision detection such as grid maps. This issue is discussed in Ch. **??** YOON: Fix

4.2.1 Common Questions

Can we use BVHs for point clouds that are directly acquired from various sensors without converting them into a mesh? BVHs are a general data structure that can be used for many different representations including meshes and point clouds. As a result, BVHs can be used for identifying collisions among point clouds. Nonetheless, we need to have another concept of identifying collisions from point clouds, since it is rather unintuitive to compute collisions from point clouds, unlike detecting collisions from meshes. Checking collisions on point clouds becomes a recent hot trend and you can find some papers about the topic.