# Chapter 3

# Collision Detection

Collision detection is an area of research in its own right, and has been extensively studied (for a survey see [68]). Despite the large body of existing work, further study of collision detection in the context of motion planning is still important for two reasons. The first is that path planning time in many domains is dominated by collision checks, meaning that optimization of this aspect provides an overall increase in efficiency. Randomized planners in particular tend to make a very large numbers of obstacle checks in solving a planning problem. This is an unavoidable fundamental property of current algorithms, because they discover the structure of the world through queries rather than some global analysis of the geometry. The second reason is that much of the collision detection research has focused on computer graphics and physical simulation, while the types of queries made for motion planning differ. In particular graphics and simulation is primarily concerned with "all-pairs" collision detection algorithms, as every object in the scene can undergo a transformation over time. In path planning, collision tests are geared toward incremental "what-if" tests of small changes in a state, and are structured as a tree or graph instead of a single evolving timeline. An exploration of collision detection is motivated by these differences, as they can potentially guide us to alternative solutions that work better for motion planning than an approach tailored for rendering or simulation.

The most general of the common approaches to collision detection support queries between complex polyhedra. For mobile robots however, it is often acceptable to make a more simplified model of the agent that consists of a small number of circles, spheres, or cylinders [75]. Many real applications desire the extra safety of an exclusion radius around the agent, for which bounding spheres and cylinders work as an approximation. One thing we can gain from this simplifying assumption is exact checks for safety along continuous trajectories; While polyhedral models are normally tested at only specific points, leading to the possi-

bility of “tunneling” between objects, circular and spherical geometries can be treated be checked as continuous sweeps over a trajectory. Thus we trade completeness (in the sense of the actual agent’s shape) in return for an exact queries on the simplified model.

In contrast to the simplicity of the agent model, for the agent’s environment we would like to support as many rich object geometries as possible. In particular we would like to be able to describe man-made objects, natural terrains, and minimally processed sensor data such as range maps. Thus the approach we take is to draw on the best existing representations for complex domains in the literature, and then take advantage of our simplified agent geometries to gain efficiency compared to more general approaches.

Typical models of the environment consist of collections of geometric objects of varying size and geometry [68]. This feature has led practical approaches to collision detection to break the task into two stages, so called *broad-phase* and *narrow-phase* collision detection:

**Broad-Phase**: Determines which geometric objects need to be checked for a certain query, ruling out distant objects based on bounding boxes or other partitionings of the space.

**Narrow-Phase**: For the objects that overlap in the broad-phase, a check is performed between the query and a single primitive object that represents an obstacle. Often objects correspond to rigid bodies.

We first focus on the various methods for broad-phase collision detection, attempting to group some of the myriad of available approaches into broader categories with common properties. Narrow-phase collision detection is mostly driven by the exact geometry of the query and the objects used to model the environment, so we only examine it within the context of our system. A broad look at narrow-phase methods are given in Lin’s summary [68]. In the second part of this chapter, the approach we have taken is examined in detail. This comprises two methods for broad-phase collision detection, including a novel approach called *extent-masks* which leverages the parallelism inherent in modern computers, and a scalable implementation of hierarchical bounding volumes. Next our methods for narrow-phase collision detection are described, including an approach for sphere-based narrow-phase collision checks that leverages a point-object distance primitive to derive all other necessary checks. Finally, we examine the performance of the developed collision detection system.

### 3.0.1 Contributions of this Chapter

- The *Extent Masks* algorithm is a novel contribution of this thesis research.

- The mathematical approach to iterative swept-sphere collision detection, although simple, appears to be novel. Quinlan's [75] related approach of "bubbles" does not use the explicit iterative formulation.
- The heuristic used for choosing splitting planes for building obstacle trees has not appeared in any previous work of which the author is aware.
- The summary of existing work contributes observations of similarity between several algorithms when applied to collision detection for motion planners. In particular, spatial partition trees and hierarchical bounding volumes become quite similar for motion planning applications when common optimizations are applied.

## 3.1 Existing Approaches to Broad-Phase Collision Detection

Many methods for broad-phase collision detection have been described in literature and employed in implementing collision detection libraries. Numerous variants and derived versions of the algorithms exist, making an exhaustive description impractical. Instead, we group the methods into several categories and cover their usual properties. Below we describe the following categories of broad-phase methods:

- Coordinate Sorting and Sweep-and-Prune
- Spatial Hashing
- Spatial Partitioning Trees
- Hierarchical Bounding Volumes
- Extent Masks

### 3.1.1 Coordinate Sorting and Sweep-and-Prune

One early method for broad-phase collision detection is *coordinate sorting*. In this method, a marker for the start and end $[x_k, x'_k]$ for each object $k$ is placed in several arrays, representing the axes of the workspace $(x, y, \ldots)$. The arrays are kept sorted by value along that axis, and determining if two objects may overlap along an axis involves searching for start or end markers of one object contained between the other object's markers. To carry out the test in higher dimensions, each axis is checked in turn, and collisions can be ruled out if

their is no overlap in any of the dimensions. If the objects overlap along all three axes, this indicates that their axis-aligned bounding boxes (AABBs) overlap, and this pair of objects must be tested in the narrow phase of collision detection. This method works well when the objects are spread along most axes, but fares quite poorly if many objects are stacked along any axis, which leads to an $O(n^2)$ worst case. Unfortunately it is quite common in structured environments to have many aligned objects, so coordinate sorting is generally not used for collision detection. One adaptation of coordinate sorting that has proved quite popular is the *sweep and prune* method used in I-COLLIDE [28]. This method combines initial coordinate sorting followed by tracking objects over time. Sorted coordinate axes are maintained over time while objects move, and with high inter-frame coherence, the number of swaps of end markers that need to be performed each time is relatively low. The crucial insight for collision detection is that a collision can begin or end only when two end markers of different objects switch positions. Any time such an event happens, the pair of objects are checked for overlap between their bounding boxes, and this state is recorded until the objects no longer overlap (which can also only occur during a marker swap along some axis). This method works well for evolving systems with small step sizes (such as physical simulation). However, for path planning we are more interested in "random-access" queries without any inter-frame coherence (the obstacles normally have inter-frame coherence, but the queries do not).

### 3.1.2 Spatial Hashing

Another method for broad-phase collision detection based on approximating axis-aligned bounding boxes is the *uniform spatial hash table* (USHT). A uniform spatial hash table breaks the workspace into a set of cells, with evenly spaced divisions along each axis. Each cell has a linked list of objects that overlap that cell. In order to check a region for collision with a set of objects, first each object is added to all the cells it overlaps (which is easy to calculate for AABBs with ranges along each axis), and for the query every cell is checked and narrow-phase collision performed between the query and each object on the list. Because objects often overlap several cells, a narrow-phase check could be performed many times with the same object during a single query. To mitigate this, we can maintain a unique integer query ID, with each object listing the last query which it has been checked against. If a query encounters an object it has already checked, it can skip performing the more expensive narrow-phase check. Spatial hashing works well for multiple similar sized objects and queries, and unlike coordinate sorting, it still does well in cases where there is a significant amount of overlap in one or more dimensions. However, it has a speed-vs-accuracy tradeoff based on the cell size, where small cells are more accurate at determining collisions but involve more work in adding the object to the additional cell lists. For the best performance, the cell size

must often be manually tuned. There is also potentially a steep memory tradeoff, although this can be handled by hashing the cell id using a traditional hash function to map it to a smaller number of storage cells. Hash collisions are handled by the already existing linked lists at each bucket/cell.

```
type SpatialHashTable = array[MaxX,MaxY] of list[ObjectRef];

function AddObject(table:SpatialHashTable, o:ObjectRef) : unit
1   let e^0 = ⌊ (o.bbox.cen − o.bbox.rad) / CellSize ⌋;
2   let e^1 = ⌈ (o.bbox.cen + o.bbox.rad) / CellSize ⌉;
3   foreach y:ℤ ∈ [e^0_x, e^1_x] do
4       foreach x:ℤ ∈ [e^0_y, e^1_y] do
5           table[x,y] ← table[x,y] ∪ {object.id};

function CheckBox(table:SpatialHashTable, q:Query) : Status
1   let e^0 = ⌊ (q.bbox.cen − q.bbox.rad) / CellSize ⌋;
2   let e^1 = ⌈ (q.bbox.cen + q.bbox.rad) / CellSize ⌉;
3   foreach y:ℤ ∈ [e^0_x, e^1_x] do
4       foreach x:ℤ ∈ [e^0_y, e^1_y] do
5           foreach o:ObjectRef ∈ table[x,y] do
6               if Overlap(o.bbox, q.bbox) ∧ NarrowPhase(o,q) ≠ Free then
7                   return Collision
8   return Free;
```

Table 3.1: Algorithm for constructing and using a basic spatial hash table

### 3.1.3 Spatial Partitioning Trees

Another broad category of broad-phase collision detection methods are those based on spatial partition trees. The simplest spatial partition is the KD-Tree, which recursively splits the environment into two halves along one of the domain axes. To build the tree, a splitting axis and location is determined, and the obstacles are put in one of two subtrees based on which side the obstacle falls on. Obstacles that straddle the splitting plane can either be placed in both subtrees, or it can be placed in one of the child trees which are then allowed to overlap slightly. The former approach is common in graphics for visibility queries, while collision detection often employs overlapping subtrees. If we relax the constraint of axis-aligned splitting planes to allow arbitrary orientations, the structure is a binary space partition tree or BSP-tree [39]. Both KD-trees and BSP-trees can be constructed in a number of ways,

although the most common approach is recursive subdivision. The construction starts with all the primitive objects in a single node, and splits the node into two children based on a plane chosen using some heuristic. The construction then proceeds recursively on the children until some minimal number of objects is reached, resulting in a leaf node. Pseudo-code for implementing a variant of a spatial tree with per-node bounding boxes (making it technically equivalent to a AABB-tree) is given in Table 3.2. It assumes a function *ChoosePlane* for choosing an appropriate splitting plane based on a set of objects. Many heuristics for choosing splitting planes have been put forward, although the most common approaches are to evenly split the longest axis of the current subvolume, or to split the objects into sets with an equal number of objects. An examples KD-tree and BSP-tree can be seen in Figure 3.1. Once such a hierarchical representation is created, it can be used for queries in a straightforward manner. A query need only be checked against the subtrees that it overlaps; any time that the query volume is one one side of a splitting plane, only that child needs to be checked. If the query straddles a splitting plane, the query volume is subdivided and each portion is checked recursively. If the query is itself represented as a spatial tree (such as a BSP), the checking can be quite efficient [71].

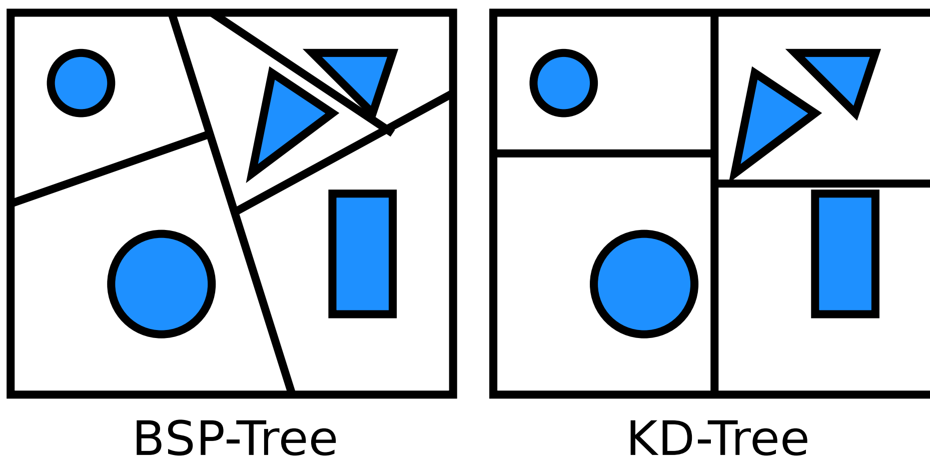


Figure 3.1: Examples of KD and BSP spatial partitioning trees

One method not mentioned above is the *interval tree*, which is popular in computational geometry [30, 31]. It can be thought of as a special case of a KD-tree with a special construction algorithm. The elements of an interval tree are pairs of extents for a particular axis $[x_k, x'_k]$, along with a maximum extent for the entire subtree rooted at this element. An interval tree is then a balanced binary tree built out of initial starting extent at each node. Then in a single linear pass the maximum subtree extents can be calculated. To go beyond a single dimension, the tree alternates the dimension represented at each depth level in the tree. An interval tree can thus be represented by an ordinary KD-tree, and we need only

to use the balanced binary tree for construction, which splitting planes based on the initial extents for the various objects. The KD-tree then needs to keep track of the maximal extent at each level of its tree. If a global AABB is constructed, then during any descent of the tree an AABB for subtree rooted at that node can be maintained.

### 3.1.4 Hierarchical Bounding Volumes

The final major category of broad-phase collision detectors are those that use hierarchical bounding volumes (HBVs) as their representation. The bounding volumes are typically either spheres, axis-aligned bounding boxes (AABBs) or oriented bounding boxed (OBBs). In these representations, a bounding volume is placed around all the objects at the root of a tree, and subtrees contain subsets of the objects with their own associated bounding volumes. Examples of an AABB-tree and an OBB-tree are shown in Figure 3.2. Since broad-phase collision detection is simply to find out which bounding volumes overlap and require further processing, HBVs and a natural hierarchical extension of the brute force method of checking every primitive. Pseudo-code for the spatial tree with per-node bounding boxes is listed in Table 3.3. In all variants, each group of primitives under a subtree has a "global" bounding volume around the entire group, so if the query does not overlap with that volume, we can rule out all that subtree and thus all the objects in that subset. If the query does overlap, then we recurse down that subtree checking further subtrees and eventually leaf objects if needed. For construction of HBVs many methods have been put forward, and in general they are heuristics designed to decrease the total volume of the bounding boxes as much as possible at each level in the tree, or in other words, to look for splits resulting in the tightest fitting volumes. This avoids the exponential time required for finding an optimal fit. Some of the most common heuristic methods for constructing HBVs are the same as for spatial partitioning trees – namely selecting a splitting plane based on some criteria and building the subtrees recursively. As was mentioned in the previous section, example pseudo-code for constructing this type of spatial/HBV tree variant is given in Table 3.2.

## 3.2 Approach to Collision Checking

### 3.2.1 Extent Masks

The *extent masks* approach is a novel method for broad-phase collision detection that offers exact bounding box queries and consistent performance suitable for a real-time planner. It

```
struct Plane
var p : Point
var n : Vector
end

struct SpatialTree
var left, right : SpatialTree
var bbox : AABBox
var objects : ObjectList
end

function Split(objects:ObjectList,plane:Plane) : (ObjectList,ObjectList)
1    var l, r : ObjectList
2    foreach o:Object ∈ objects do
3        if (o.cen - plane.p) · plane.n > 0
4            then r ← r ∪ {o}
5            else l ← l ∪ {o}
6    return (l,r)

function MakeLeaf(objects:ObjectList) : SpatialTree
1    var n : SpatialTree;
2    n.bbox ← EmptyBBox
3    foreach o:Object ∈ objects do
4        n.bbox ← Union(n.bbox,o.bbox)
5    n.objects ← objects
6    return n

function MakeInterior(objects:ObjectList) : SpatialTree
1    let plane = ChooseSplitPlane(objects)
2    let (l, r) = Split(objects,plane)
3    var n : SpatialTree;
4    n.left ← MakeSpatialTree(l)
5    n.right ← MakeSpatialTree(r)
6    n.bbox ← Union(n.left.bbox, n.right.bbox)
7    return n

function MakeSpatialTree(objects:ObjectList) : SpatialTree
1    if ListLength(objects) < LeafSize
2        then return MakeLeaf(objects)
3        else return MakeInterior(objects)
```

Table 3.2: Algorithm for building a spatial tree with per-node bounding boxes

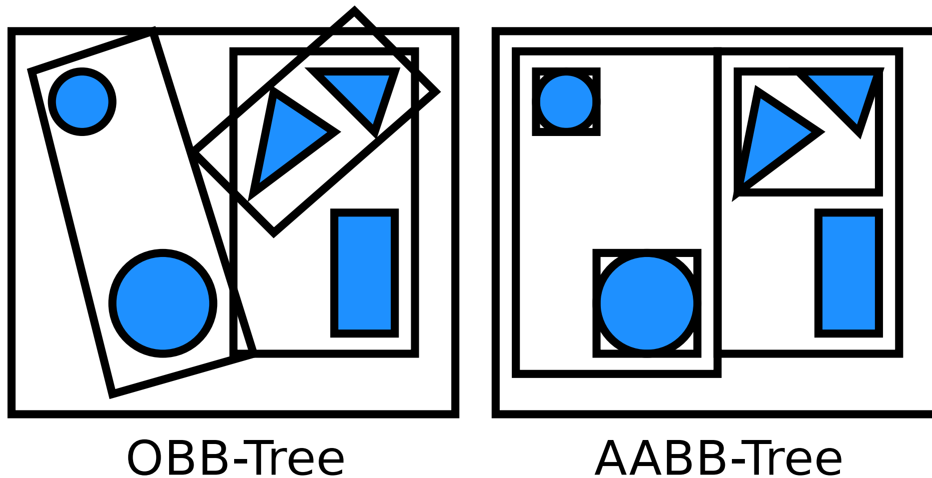


Figure 3.2: Examples of several space partitioning hierarchies

leverages the parallelism of bit operations on modern computers for speed, taking advantage of the common case of the relatively few dynamic obstacles present in many domains. First, we can define a mask function for the $x$ axis as $m_x(x, i) = \{0, 1\}$, and analogously for the other axes. It is 1 if the object overlaps on that axis at location $x$, and 0 otherwise. An example domain with three obstacles is shown in Figure 3.3, with a color-coded mask functions shown along the $x$ and $y$ axes. To perform a query, we want to find the set of all "ones" for $m$ at a given location $x$, and then integrate that function between two values $x_0$ and $x_1$ to get a union set of all obstacles overlapping a range, as shown in Figure 3.4. We then want to take the union of the sets from each axis to get the overall set of overlapping axis-aligned bounding boxes. Thus for the algorithm to work, we need thus primitives for taking the union along an axis, and intersection between axes.

To represent the mask function, we first use binary integers to represent the value of the function at any axis location $x$, with the $i$'th binary digit (bit) representing the value of $m_x(x, i)$. We refer to this binary integer representing a set as a *bitmask*. Next, noting that the function only changes at $x$ values where an obstacle begins or ends (its extents), we can compactly represent the function with an array of $x$ values and the associated binary integer to represent the set of overlapping obstacles. Finally, to allow for taking fast unions, we represent the function $m$ in a cumulative fashion along $x$, recording all obstacles that have started before a value of $x$, and all that have ended before a value of $x$. Table 3.4 gives the representation and implementation in pseudo-code for creating extent masks. First in the function *MaskSignalAdd*, objects are added to the array unsorted, with entries for the beginning and ending extents where only the bit for that obstacle is set. The function

```
function CheckBox(tree:SpatialTree, q:Query) : Status
1   if Overlap(tree.bbox,q.bbox)
2       then if tree.objects ≠ {}
3           then foreach o:Object ∈ tree.objects
4               do if Overlap(o.bbox, q.bbox) ∧ NarrowPhase(o,q) ≠ Free
5                   then return Collision
6           else if CheckBox(tree.left,q) = Free ∧ CheckBox(tree.right,q) = Free
7               then return Free
8               else return Collision
9       else return Free
```

Table 3.3: Algorithm for checking a spatial tree with per-node bounding boxes

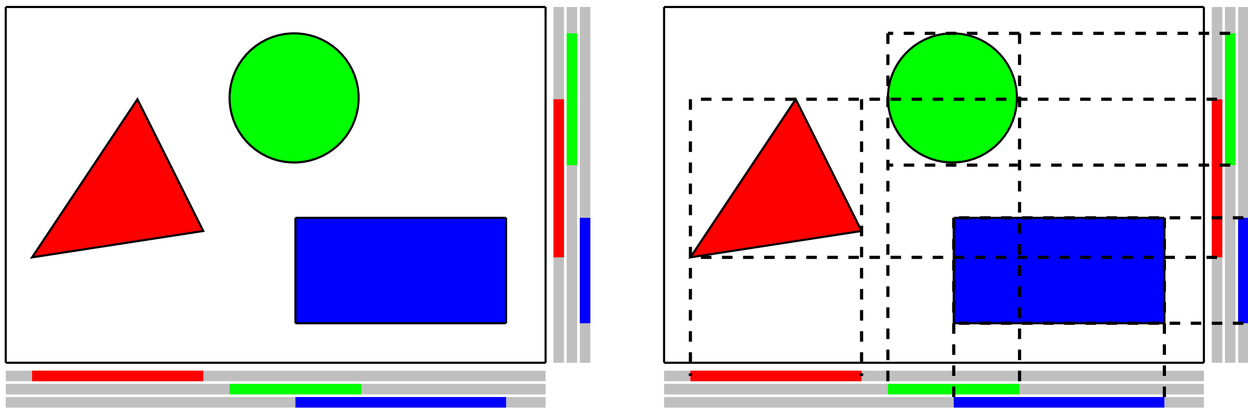

Figure 3.3: Example environment with the corresponding projected extent masks

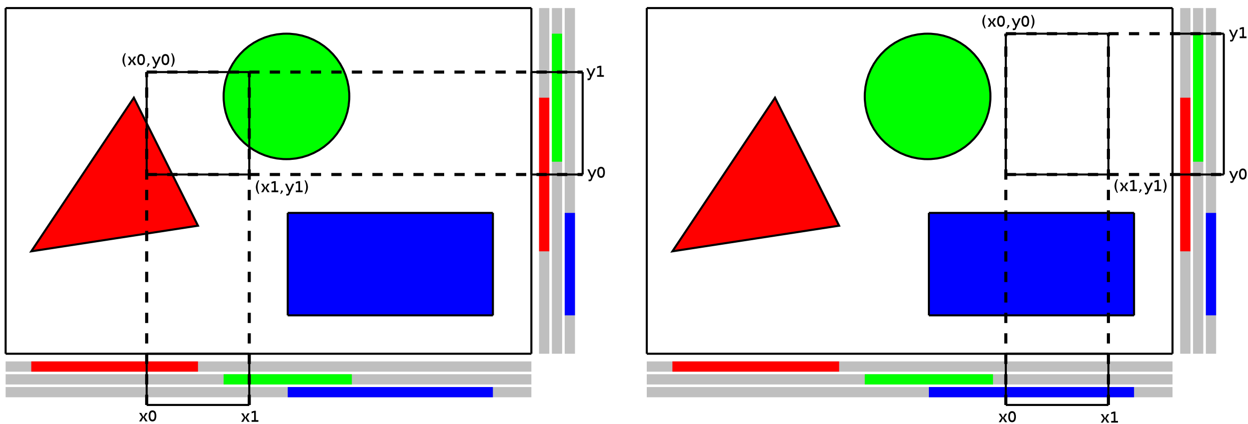


Figure 3.4: Two broad-phase bounding box checks using projected extent masks

*MaskSignalSetup* first sorts the array by axis value, and then takes the union of the start and end sets using a bitwise-or operation on the binary number representation. In Table 3.5, we can see how to query the extent mask efficiently. A standard binary search is used to find the correct entries for the start and end of a range, and bitwise operations are again used to extract all objects that have a starting extent by the end of the range, but which do not have an ending extent before the beginning of the range. This corresponds exactly to the set of obstacles overlapping the range. Extending the algorithm to multiple axes is trivial, as we need only query each axis and take the union using a bitwise-and to combine the results of *MaskSignalGet* from each axis.

The efficiency of the extent masks approach relies critically on the speed of the bitwise operations on the obstacle sets represented is integers. However, if the bitmask is represented by a 32-bit or 64-bit unsigned integer on the machine, all of these bitwise operations can be carried out in a single machine instruction. Thus as long as the number of obstacles is smaller than the number of bits the machine can use to represent an integer, those operations are constant time. Combined with the binary search, this leads to a complexity of $O(\log n)$ for one axis and $O(d \log n)$ for $d$ axes. Of course, to scale beyond the word size of the machine integers, we must employ arrays of machine integers leading to an ultimate complexity of $O(dn)$, but this linear component is quite small in practice, and many domains do not require more than 64 dynamic obstacles. As we show later in this chapter, complex static geometries can be collected into a single large obstacle at the broad-phase collision level, so the limitation is not particularly restrictive in practice.

### 3.2.2 Heuristically Balanced AABB tree

In cases where large numbers of obstacles were needed, and static lookup structures were not appropriate, another method based on HBVs was implemented. It uses the algorithm of Table 3.2, constructing a tree of axis-aligned bounding boxes using splitting planes to build the tree in an efficient top-down fashion. The resulting tree does not guarantee logarithmic access times, but performs well in practice for relatively uniform distribution of obstacles. The method used for the *ChooseSplitPlane* function is to try a set of candidate splitting planes, evaluated with the function *EvaluateSplitPlane* listed in Table 3.6. The candidate planes are located at the median location of all the current objects, and oriented along each possible axis.

The evaluation function for splitting planes is a heuristic guided on two main principles:

- Queries are uniform, thus subtree being descended is proportional to its area

```
struct MaskSignalEntry
var x : ℝ
var start,end : BitMask
end

struct MaskSignal
var a : array[MaxObjects*2] of MaskSignalEntry
var n : ℤ
end

function MaskSignalAdd(m:MaskSignal, x0,x1:ℝ, i:ℤ)
1     m.a[n+0].x ← x0
2     m.a[n+0].start ← BitMaskSingleBit(i)
3     m.a[n+0].end ← 0
4     m.a[n+1].x ← x1
5     m.a[n+1].start ← 0
6     m.a[n+1].end ← BitMaskSingleBit(i)
7     m.n ← m.n + 2

function MaskSignalSetup(m:MaskSignal)
1     Sort(m.a, m.n)
2     var s,e : BitMask
3     s ← 0
4     e ← 0
5     for i = 0 to n-1 do
6         s ← BitwiseOr(s, m.a[i].start)
7         e ← BitwiseOr(e, m.a[i].end)
8         m.a[i].start ← s
9         m.a[i].end ← e
```

Table 3.4: Algorithm for building a mask signal

```
function BinarySearch(m:MaskSignal, x:ℝ, l,r:ℤ) : ℤ
1    while r - l > 1
2        let c = ⌊ (l + r) / 2 ⌋
3        if x < m.a[c].x
4            then r ← m
5            else l ← m

function MaskSignalGet(m:MaskSignal, x0,x1 : ℝ) : BitMask
1    let l = BinarySearch(m,x0,0,m.n)
2    let r = BinarySearch(m,x1,0,m.n)
3    return BitwiseAnd(m.a[r].start, BitwiseNegate(m.a[r].end))
```

Table 3.5: Algorithm for querying a mask signal

```
function EvaluateSplitPlane(objects:ObjectList, p:Plane) : ℝ
1    var num : array[2] of ℤ
2    var bbox : array[2] of AABBox
3    var s : ℤ
4    num[0] ← 0
5    num[1] ← 0
6    bbox[0] ← EmptyBBox
7    bbox[1] ← EmptyBBox
8    foreach o:Object ∈ objects do
9        if (o.cen - plane.p) · plane.n > 0
10           then s ← 1
11           else s ← 0
12       num[s] ← num[s] + 1
13       bbox[s] ← Union(bbox[s],o.bbox)
14   return Area(bbox[0])*num[0] + Area(bbox[1])*num[1]
```

Table 3.6: Our method to evaluate a splitting plane for building a hierarchical bounding box representation for collision checking.

(2D) or volume (3D)

- The worst case complexity for accessing a subtree is linear

The evaluation metric is thus to consider a split by calculating the resulting bounding boxes and number of objects in each subtree, and estimating the worst-case expected time as the sum of the areas of the bounding boxes multiplied by the number of objects in each subtree. If the splitting plane is chosen with the smallest cost evaluation value, then the tree greedily minimizes the expected worst-case running time for accesses. This is of course subject to the limitations of choosing from a small number of axis-aligned splitting planes, and thus does not minimize the running time in any global way for all the possible partitionings. However it tends to guide the tree building procedure to construct a tree balanced enough to support fast accesses as is shown in Section 3.2.4.

### 3.2.3 Narrow-Phase Collision Checking

In comparison to broad-phase collision detection, narrow-phase collision detection in our system is quite a bit more specialized in its approach. Much of the structure is dictated by our circle/sphere queries and the geometry of the primitive objects rather than adherence to an approach from the literature on collision checking. The primitive queries supported by obstacles our system are the following:

1. Checking a point $q$ and radius $r$ for collision
2. Checking a swept circle or sphere from $q_0$ to $q_1$ and radius $r$
3. Calculating the distance from $q$ to the nearest point on an obstacle
4. Calculating the nearest point $p$ on the obstacle to a query point $q$
5. Optionally calculating two tangents to the obstacle from a point $q$ (this allows local avoidance heuristics in the planner)

In the originally developed 2D implementation, each of the above functions was implemented for maximum efficiency, although this required a larger amount of work than necessary for adding new obstacle types. In particular, moving to 3D complicated the geometry for swept queries significantly. This motivated a two-tiered approach where default implementations were provided for all but the nearest point queries, and a particular obstacle could override the default implementations with a more efficient specific version if needed. The virtual

method feature of C++ made this relatively easy to implement; Default versions of most functions were provided in the base class. In particular, the default implementations used the following properties:

- Calculating the distance to an object depends only on the distance from the query $q$ to the nearest point on the obstacle
- Checking a point $q$ can simply check the distance against $r$
- Checking a swept query can be implemented iteratively using only the distance query

While the methods for point queries are obvious, the method used for checking a swept circle or sphere requires more explanation. A visualization of the approach appears in Figure 3.5. For any given point along a swept trajectory, the current distance to obstacles, or clearance, determines how far along a trajectory is safe. The checker can then step forward by that distance, and recursively check the remaining swept area. The figure shows a dark blue trajectory to be checked, the light blue is the current clearance, and the red spheres show the steps that can be safely taken each iteration. Pseudo-code for the approach can be found in Figure 3.7 Normally, few iterations are required, although the algorithm can take many steps if it is very close to an obstacle. This is handled by failing after a certain number of iterations have been exceeded. Though this is yet another pessimistic approximation, long paths running very close to obstacles are not typically desirable for execution by mobile robots anyway.

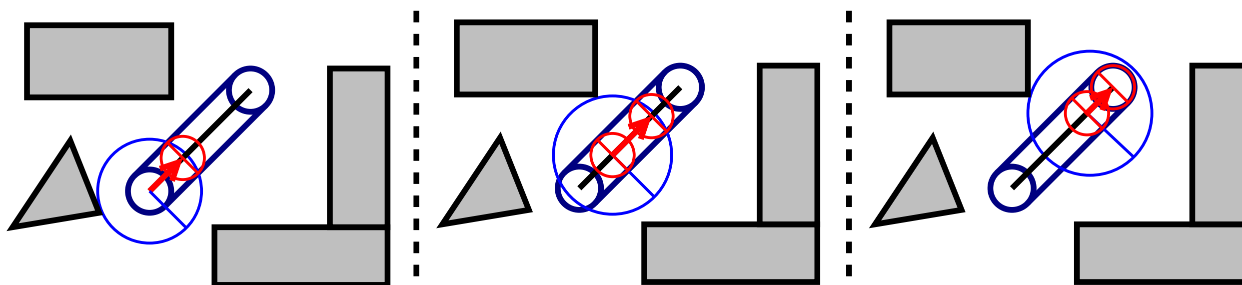

Figure 3.5: Example swept circle obstacle check using only distance queries.

While in our current implementation, only line-swept-spheres are supported, the distance query stepping method of checking a swept sphere can be applied to other trajectory functions as well. For some continuous trajectory function $x(t)$ where time $t \in [0, t_f]$, starting from an initial position $x(0)$ with a distance of at least $D(x(0)) = d$ from all obstacles, we need only find the first $t$ such that $\|x(t) - x(0)\|^2 = d^2$, or verify that no such $t$ exists for $t \in [0, t_f]$. For functions with bounded curvature, such as lines and circular arcs, these calculations are

```
function CheckSweptQuery(q0, q1 : Point, r : ℝ) : bool
1     let l =Dist(q0, q1);
2     let d = (q1 − q0)/l;
3     var s : ℝ;
4     s ← 0;
5     while s < l do
6         let q = q0 + ds
7         let f = CalcFreeDist(q);
8         if f < r
9             then return Collision
10            else s ← s + √(f² − r²)
11    return Free;
```

Table 3.7: Algorithm for checking a swept query based on distance

straightforward. In particular, for a linear trajectory defined by $x(t) = a + bt$, then the solution is $t_i = \sqrt{d^s - r^2}/\|b\|$. If $t_i > t_f$, then the trajectory is verified to be free, otherwise a recursive check must be made with a new trajectory starting at time $t_i$. Thanks to the square root, $t_i$ increases quite rapidly from zero even with very small clearances, which is the reason few steps are normally needed for a typical check, and results in the approach being quite efficient in practice.

The following primitive obstacles are implemented by our collision checker:

- Axis-aligned rectangles
- Circles/spheres
- Convex polygons
- Regular 2D grids
- 3D terrain sampled on a regular grid (elevation map)
- Large 3D polygon soups

The complexity of the individual obstacles ranges greatly, from simple geometric shapes such as circles and rectangles all the way up to 200x200 terrain meshes and polygon soups. While the geometric shapes are straightforward to develop a distance metric for, the terrain obstacle posed the challenge of efficient distance calculation. Since the terrain is a 2D grid projected into 3D as a low-curvature mesh (i.e. a relatively flat manifold), the approach taken was to break it up using a 2D K-D tree in grid space, and then to bound the individual nodes

with an axis-aligned bounding box in the 3D space. The first few levels of a terrain tree are shown in Figure 3.6. To query the distance from a point to the terrain, we follow the branches of the KD tree nearest first, calculating a distance to the actual terrain once a leaf node is reached. After that, the remainder of the traversals and be compared against this known minimum and pruned if the bounding box is further than the current minimum. This aggressive pruning and the relatively flat nature of actual terrain meshes yields near logarithmic access times for the queries. The nodes descended during a particular distance query are shown in Figure 3.7. A similar approach was taken for polygon soups, where a large static set of triangles are stored in a KD tree with per-node bounding boxes. Planes splitting normals were determined by the largest axis, and the plane splitting locations were set at the median of that axis. Due to the terrains being static, the lookup structures could be calculated once for each object when loading the domain, and reused in many iterations of the planner. Construction times for the lookup structure were minimal compared to the time required to load the terrain from the disk.

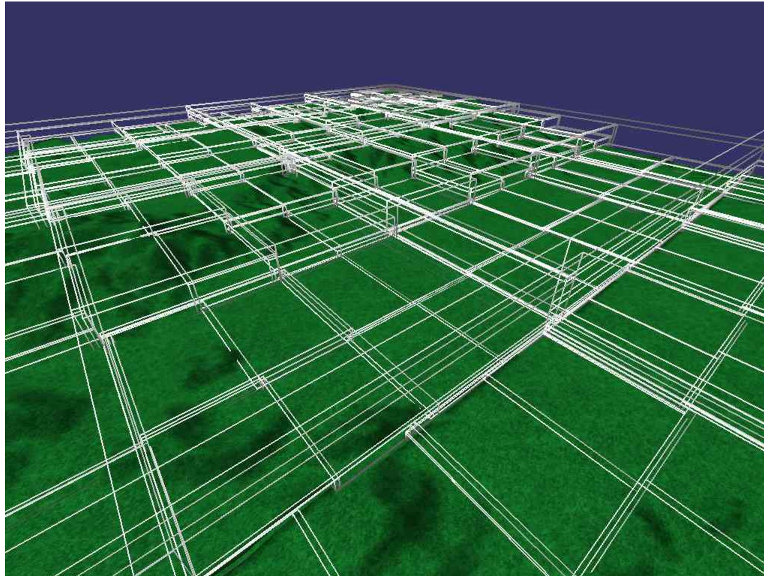

Figure 3.6: The first levels of bounding boxes on a terrain K-D tree.

### 3.2.4 Performance Measurement

Although we are ultimately interested in the performance of a motion planner as a whole, we can compare collision detection methods directly to get an idea of what sort of performance to expect. We will compare collision detection using three broad phase methods along

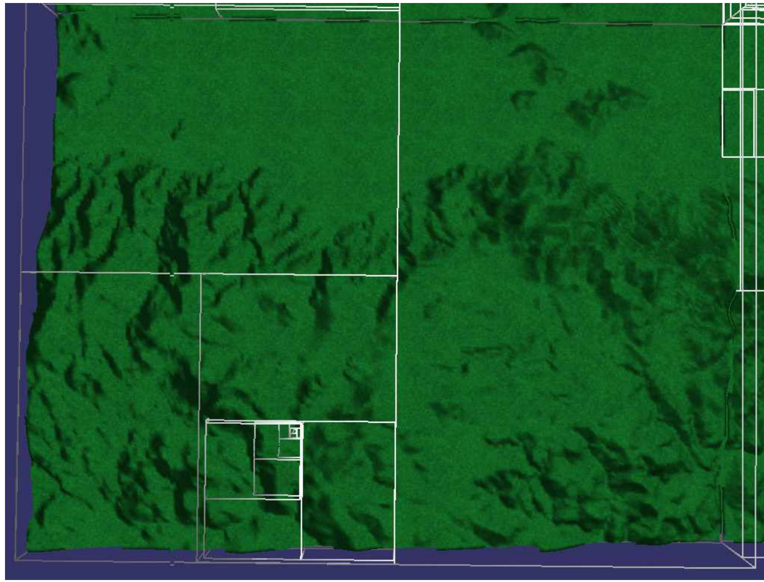

Figure 3.7: Nodes expanded on an example terrain distance query.

with our narrow phase approach, but we will limit our comparison to the two environments represented by Figure 3.8, called *circles* and *rectangles*. Each environment represents an enclosed 5.5m by 4m with 64 randomly placed obstacles. In addition, a variant of the circles domain was created with up to 256 circles with half the expected radius (thus covering approximately the same fraction of the workspace). A radius of 90mm was selected for each of the queries, thus representing an environment similar to the RoboCup small-size domain.

We first tested the mask extents implementation on the circles domain to establish the type of per-query run-time distribution. The probability density function is shown in Figure 3.9. Three variants are shown, one each for cases of a detected collision or a no-collision case, as well as the average case for the environment.

The following observations can be made of the collision query time probability density function are the following.

- The query time averages only 0.375 *microseconds*.
- The distribution is approximately Gaussian, and thus can be reasonably summarized by means and confidence intervals.
- The query time varies depending on whether the query location is free or not, however the difference is not so large that it is likely to have a significant effect on callers.

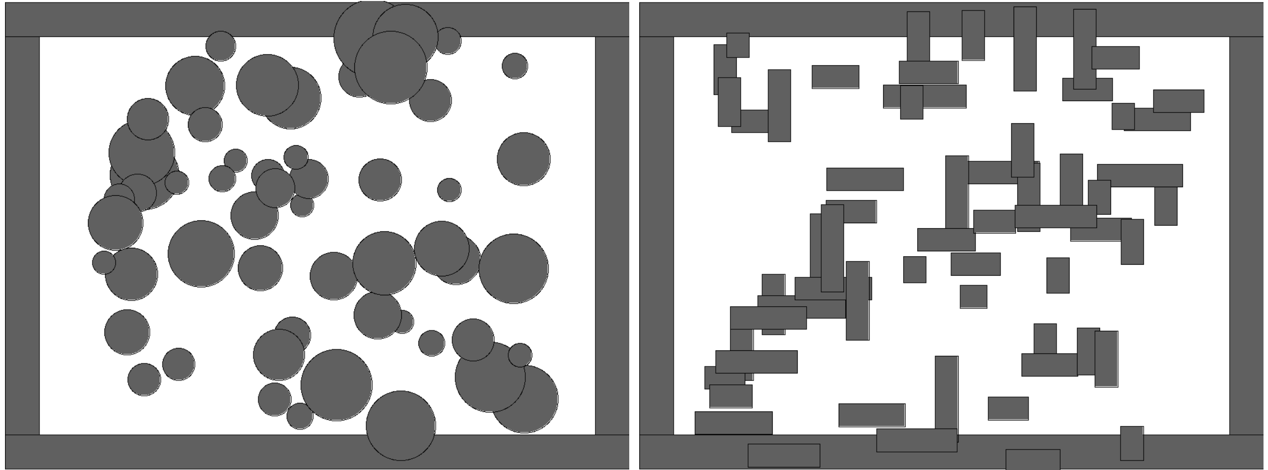

Figure 3.8: Two domains with 64 obstacles used as collision checking benchmarks.

After establishing that queries are indeed quite fast (over 2.5 million per second), the next question then becomes how well this speed scales with the number of obstacles in the domain. To examine this scalability we compared three algorithms; The extent masks approach, the AABB tree method, and a baseline linear array method that simply checks the obstacles one at a time in array order. The scaling tests are based on the 256-circle domain, but with only the first $n$ obstacles included for each datapoint as $n$ was varied from 0 to 256 obstacles. Figure 3.10 shows how extent masks scales. The major features are that the scaling is sublinear, that the confidence interval is reasonable narrow, and that it is limited to only 64 obstacles due to the machine representation of bitmasks. In Figure 3.11, we can see the scaling of the bounding volume tree approach. Note here that the variance is significantly more than for extent masks, however the average performance is superior. The AABB tree scales extremely well up to 256 obstacles, with a very sublinear trend on both the average case and the 95% (upper) confidence bound. The AABB tree also displays some non-monotonic variability based on the number of obstacles, which can be attributed to the different trees that are constructed from the different subsets of obstacles. Finally, the linear array baseline implementation is shown in Figure 3.12. The 95% confidence bound scales linearly, while the average case is improved due to short-circuiting on queries which hit an obstacle. The variance increases steadily with time, also due to the short circuiting.

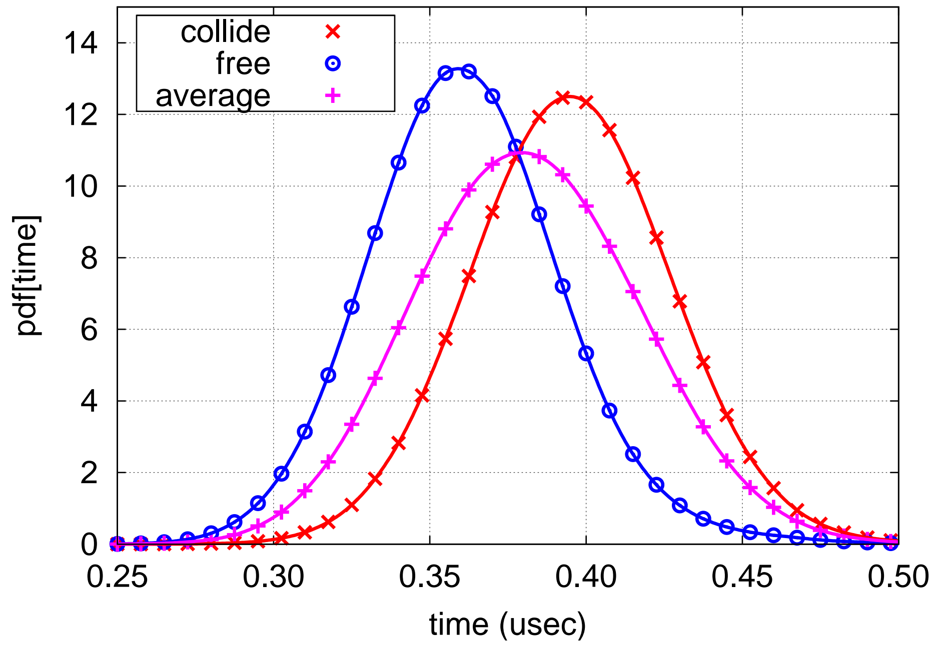


Figure 3.9: Collision query time distribution with 64 circular obstacles.

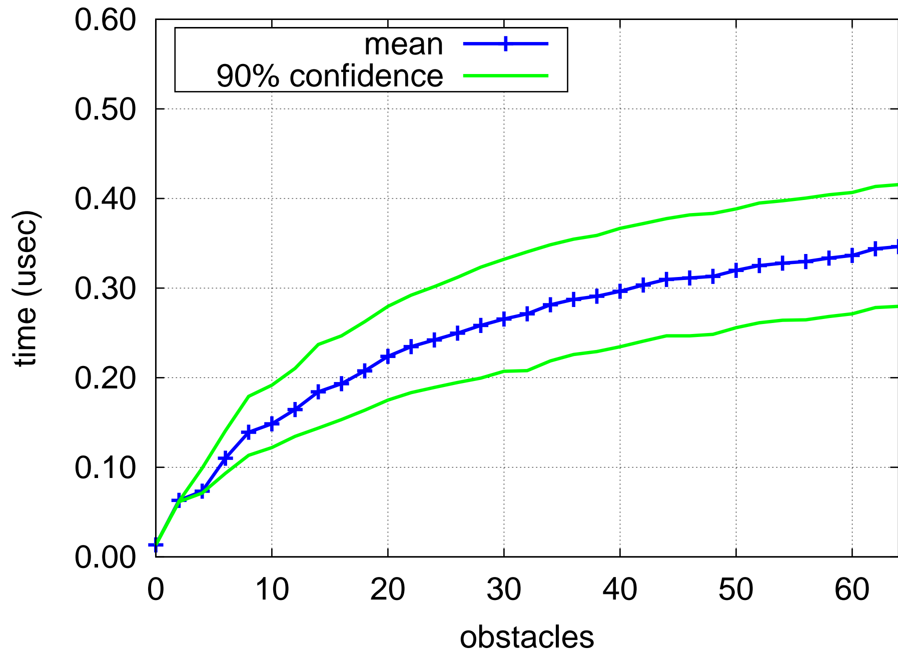


Figure 3.10: Scaling of collision queries with obstacles for extent masks

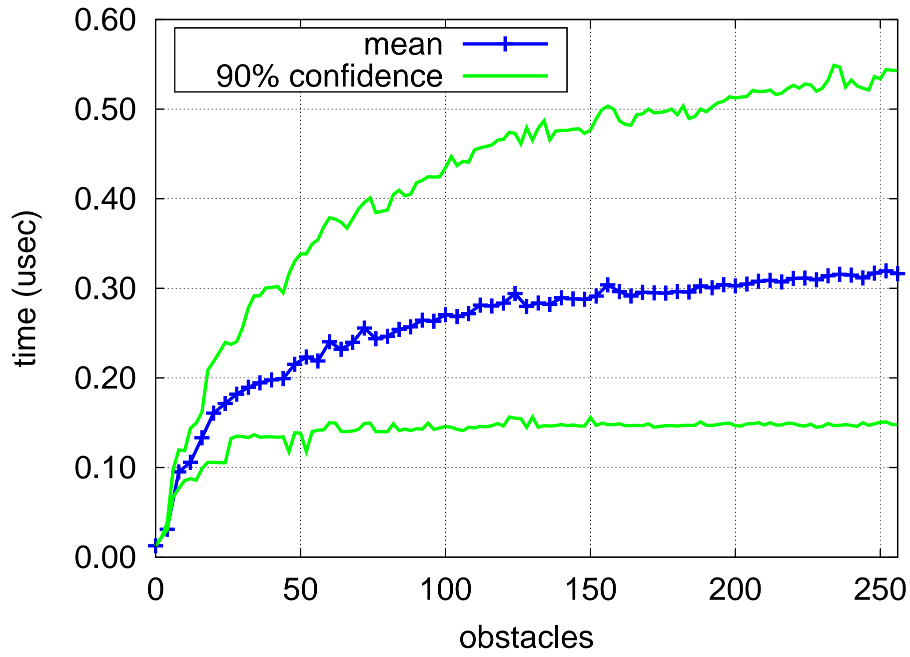


Figure 3.11: Scaling of collision queries with obstacles for AABB tree

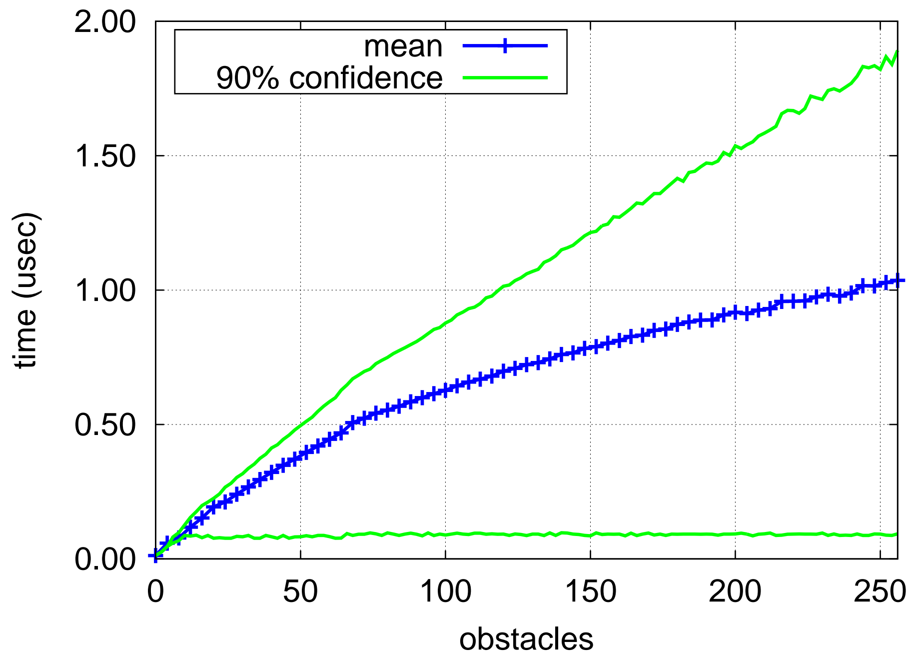


Figure 3.12: Scaling of collision queries with obstacles for a linear array

## 3.3 Summary

This chapter explored common methods for collision detection, and described the approaches taken for collision detection in this thesis. Broad-phase methods include the low-variance extent masks approach which takes advantage of the machine parallelism, and a heuristic for building axis-aligned bounding box hierarchies (AABB trees). The narrow phase uses a modular approach to allow easy implementation of new obstacle types using only a distance query, while numerous obstacle geometries are supported for narrow-phase collision detection. The collision detection method works very quickly for environments of moderate complexity, resulting in collision query times of less than half a microsecond.