Kinetic Compressed Quadtrees in the Black-Box Model with Applications to Collision Detection for Low-Density Scenes*

Mark de Berg, Marcel Roeloffzen, and Bettina Speckmann

Dept. of Computer Science, TU Eindhoven, The Netherlands
{mdberg,mroeloff,speckman}@win.tue.nl

Abstract. We present an efficient method for maintaining a compressed quadtree for a set of moving points in $\mathbb{R}^d$. Our method works in the *black-box KDS model*, where we receive the locations of the points at regular time steps and we know a bound $d_{\max}$ on the maximum displacement of any point within one time step. When the number of points within any ball of radius $d_{\max}$ is at most k at any time, then our update algorithm runs in $O(n \log k)$ time. We generalize this result to constant-complexity moving objects in $\mathbb{R}^d$. The compressed quadtree we maintain has size $O(n)$; under similar conditions as for the case of moving points it can be maintained in $O(n \log \lambda)$ time per time step, where λ is the density of the set of objects. The compressed quadtree can be used to perform broad-phase collision detection for moving objects; it will report in $O((\lambda + k)n)$ time a superset of all intersecting pairs of objects.

1 Introduction

(Kinetic) Collision Detection. Collision detection [17,19] is an important problem in computer graphics, robotics, and N-body simulations. One is given a set S of n objects, some or all of which are moving, and the task is to detect the collisions that occur. The most common way to perform collision detection is to use *time-slicing* and test for collisions at regular time steps; for graphics applications this is typically every frame. This approach can be wasteful, in particular if computations are performed from scratch every time: if the objects moved only a little, then much of the computation may be unnecessary. An alternative is to use the *kinetic-data-structure (KDS) framework* introduced by Basch *et al.* [4]. A KDS for collision detection maintains a collection of certificates (elementary geometric tests) such that there is no collision as long as the certificates remain true. The failure times of the certificates—these can be computed from the motion equations of the objects—are stored in an event queue. When the next event happens, it is checked whether there is a real collision and the set of certificates and the event queue are updated. (In addition, if there is a collision the motion equations of the objects involved are changed based on the collision response.)

* M. Roeloffzen and B. Speckmann were supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 600.065.120 and 639.022.707, respectively.

L. Epstein and P. Ferragina (Eds.): ESA 2012, LNCS 7501, pp. 383–394, 2012.

KDSs for collision detection have been proposed for 2D collision detection among polygonal objects [2,15], for 3D collision detection among spheres [14], and for 3D collision detection among fat convex objects [1].

The KDS framework is elegant and can lead to efficient algorithms, but it has its drawbacks. One is that it requires knowledge of the exact trajectories (motion equations) to compute when certificates fail. Such knowledge is not always available. Another disadvantage is that some KDSs are complicated and may not be efficient in practice—the collision-detection KDS for fat objects [1] is an example. We therefore study collision detection in the more practical *black-box model* [9,12], where we receive new locations for the objects of S at regular time steps $t = 1, 2, \ldots$This brings us back to the time-slicing approach. The goal is now to use *temporal coherence* to speed up the computations—that is, not to perform the computations from scratch at every time step—and to *prove* under which conditions this leads to an efficient solution.

Following a previous paper [9] in which we studied Delaunay triangulations and convex hulls in the black-box model we make the following[1] assumption. Let $A(t)$ denote the location of $A \in S$ at time t and let $S(t) = \{A(t) \mid A \in S\}$.

***Displacement Assumption**:*
- *There is a maximum displacement $d_{\max}$ such that for each object $A \in S$ and any time step t we have $\mathrm{dist}(A(t), A(t+1)) \leqslant d_{\max}$.*
- *Any ball of radius $d_{\max}$ intersects at most $k \ll n$ objects from $S(t)$, at any time step t.*

In the above, $\mathrm{dist}(A(t), A(t+1))$ denotes the distance between $A(t)$ and $A(t+1)$. When the objects are points this simply refers to the Euclidean distance between them. For non-point objects the distance is defined in Section 3.

The Displacement Assumption bounds how far an object can move in one time step, relative to the distances between the objects. In practise one would expect that the time intervals are such that $k = O(1)$. Note that the parameter k is not known to the algorithm, it is used only in the analysis.

Collision Detection and Spatial Decompositions. In practice collision detection is often performed in two phases: a *broad phase* that serves as a filter and reports a (hopefully small) set of potentially colliding pairs of objects—this set should include all pairs that actually collide—and a *narrow phase* that tests each of these pairs to determine if there is indeed a collision. Our paper is concerned with broad-phase collision detection; more information on the narrow phase can be found in a survey by Kockara *et al.* [16].

A natural way to perform broad-phase collision detection is to use a decomposition of the space into cells [10,18]. One then reports, for each cell in the decomposition, all pairs of objects intersecting the cell. This is also the approach that we take. For this to be efficient, one needs a decomposition with few cells such that each cell is intersected by few objects. In general this is impossible for

[1] The Displacement Assumption was phrased slightly differently in [9], but it is essentially the same.

convex cells: Chazelle [11] showed that there are sets of disjoint objects in $\mathbb{R}^d$ such that any convex decomposition in which each cell intersects $O(1)$ objects must consist of $\Omega(n^2)$ cells. However, if we take the *density* λ [6,8]—see Section 3—into account then a decomposition with a linear number of cells, each intersecting $O(\lambda)$ objects, exists and can be constructed in $O(n \log n)$ time [5]. (Practical applications typically have $\lambda \ll n$; in fact many papers assume that $\lambda = O(1)$.) Our main goal is thus to maintain a decomposition with these properties in $o(n \log n)$ time per time step under the Displacement Assumption.

Our Results. The known linear-size decomposition for low-density scenes [5] is a binary space partition (BSP), but that BSP seems difficult to maintain efficiently as the objects move. We therefore use a different decomposition, namely a compressed quadtree. In Section 2 we study the problem of maintaining a compressed quadtree for a set of moving points. We show that this can be done in $O(n \log k)$ time per time step for any (fixed) dimension d, where k is the parameter from the Displacement Assumption. (The dependency of our bounds on the dimension d is exponential, as is usually the case when quadtrees are used.) Note that $k \leqslant n$ and hence our result is never worse than recomputing the quadtree from scratch. Since hierarchical space decompositions such as (compressed) quadtrees have many uses, this result is of independent interest.

In Section 3 we turn our attention to compressed quadtrees for low-density scenes. It is known that a compressed quadtree on the bounding-box vertices of a set S of n objects in the plane has $O(n)$ cells that each intersect $O(\lambda)$ objects, where λ is the density of S [7]. We first prove that, similar to the planar case, a compressed quadtree on the bounding-box vertices of a set S of objects in $\mathbb{R}^d$ with density λ, has $O(n)$ cells each intersecting $O(\lambda)$ objects. We also show, using our result on compressed quadtrees for points, how to maintain a quadtree on a (well chosen) subset of $O(n/\lambda)$ bounding-box vertices such that each of the $O(n/\lambda)$ cells in the quadtree is intersected by $O(\lambda + k)$ objects. This quadtree can be maintained in $O(n \log \lambda)$ time per time step and can be used to report $O((\lambda + k)n)$ potentially overlapping pairs of objects. It is known that the density a set of disjoint fat objects is $O(1)$ [8]. Hence, our approach is particularly efficient for collision detection between fat objects.

2 Maintaining a Compressed Quadtree for Moving Points

A *quadtree subdivision* for a set P of n points in $\mathbb{R}^d$ is a recursive subdivision of an initial hypercube containing the points of P into 2^d equal-sized sub-hypercubes ("quadrants"). The subdivision process continues until a stopping criterion is met. We use a commonly employed criterion, namely that each final hypercube contains at most one point. A *quadtree* is a tree structure representing such a recursive subdivision. (In higher dimensions, especially in $\mathbb{R}^3$, the term *octree* is often used instead of quadtree. As our figures and examples will always be in the plane, we prefer to use the term quadtree.) A hypercube that can result from such a recursive partitioning of the initial hypercube is called a *canonical hypercube*.

In a quadtree subdivision there can be splits where only one of the resulting quadrants contains points. In the quadtree this corresponds to a node with only one non-empty child. There can even be many consecutive splits of this type, leading to a long path of nodes with only one non-empty child. As a result, a quadtree does not necessarily have linear size. A *compressed quadtree* replaces such paths by *compressed nodes*, which have two children: a child for the *hole* representing the smallest canonical hypercube containing all points, and a child for the *donut* representing the rest of the hypercube. Thus a donut is the set-theoretic difference of two hypercubes, one contained in the other. Note that the inner hypercube can share part of its boundary with the outer one; in the plane the donut then becomes a U-shape—see Fig. 1—or an L-shape. An internal node that is not compressed is called a *regular node*. In the following, we use region(v) to denote the region corresponding to a node v. For a node v whose region is a hypercube we use size(v) to denote the edge length of region(v); when v is a donut, size(v) refers to the edge length of its outer hypercube.

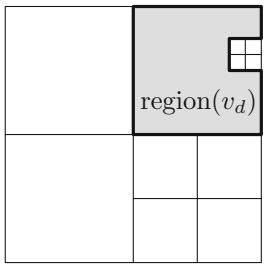

Fig. 1. A quadtree subdivion with a compressed node. The gray region denotes the region of the donut.

A compressed quadtree for a set of points has linear size and it can be constructed in $O(n \log n)$ time [3] in the appropriate model of computation [13, Chapter 2]. In this model we can compute the smallest canonical hypercube containing two points in $O(1)$ time. In this model and under the Displacement Assumption, our goal is to update the compressed quadtree in $O(n \log k)$ time.

Let $P(t)$ denote the set of points at time t, and let $\mathcal{T}(t)$ denote the compressed quadtree on $P(t)$. Our task is thus to compute $\mathcal{T}(t+1)$ from $\mathcal{T}(t)$. The root of $\mathcal{T}(t)$ represents a hypercube that contains all points of $P(t)$. We assume without loss of generality that both the size of the initial hypercube and $d_{\max}$ are powers of 2, so that there exists canonical hypercubes of size $d_{\max}$. As the points move, it can happen that the initial hypercube no longer contains all the points. When this happens, we enlarge the initial hypercube. The easy details are omitted here.

Since we assume that points move only a small distance—at most $d_{\max}$—we would like to maintain $\mathcal{T}$ by moving points to neighboring cells and then locally recomputing the quadtree. To this end we first prune the tree by removing all nodes whose region has size less than $d_{\max}$; moreover for compressed nodes u with size(u) $\geqslant d_{\max}$ but size(u_h) $< d_{\max}$ for the hole u_h of u, we replace u_h by a canonical hypercube of size $d_{\max}$ containing region(u_h). We use $\mathcal{T}_0$ to denote this reduced quadtree. Note that $\mathcal{T}_0$ is the compressed version of the quadtree for $P(t)$ that uses the following stopping criterion for the recursive subdivision process: a hypercube σ becomes a leaf when (i) σ contains at most one point from $P(t)$, or (ii) σ has edge length $d_{\max}$. Each leaf in $\mathcal{T}_0$ has a list of all points contained in its region. By the Displacement Assumption, this list contains $O(k)$ points. The reduced quadtree $\mathcal{T}_0$ can easily be computed from $\mathcal{T}(t)$ in $O(n)$ time.

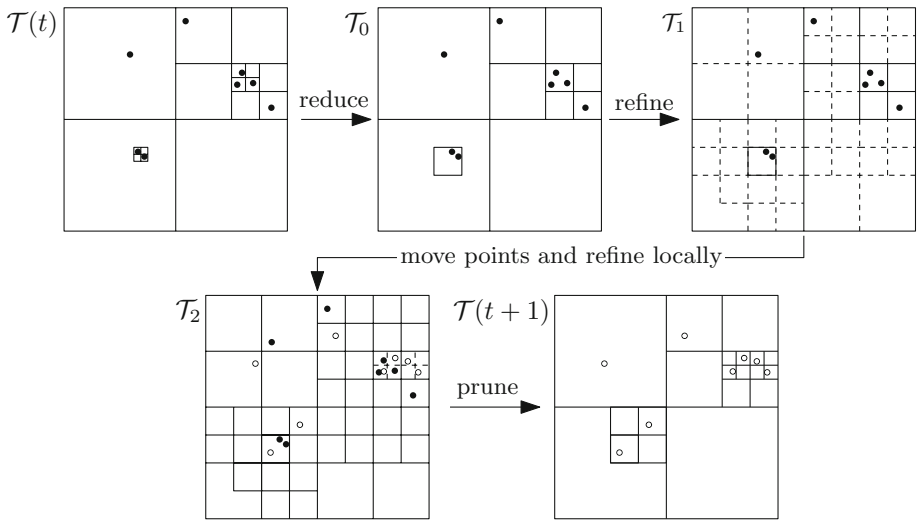

Fig. 2. The various intermediate quadtrees while updating $\mathcal{T}(t)$ to $\mathcal{T}(t+1)$. New edges in each step are illustrated by dashed edges, solid disks are points at time t and open disks are points at time $t+1$.

We do not lose any useful information in this step as within some region(v) with size(v) $= d_{\max}$ the points can move (nearly) arbitrarily and, hence, the subtree of u at time $t+1$ need not have any relation to the subtree of u at time t.

Now we would like to move the points from the leaf regions of $\mathcal{T}_0$ to neighboring leaf regions according to their new positions at time $t+1$, and then locally rebuild the compressed quadtree. The main problem in this approach is that a leaf region can border many other leaf regions and many points may move into it. Constructing the subtree replacing that leaf from scratch is then too expensive. We solve this by first building an intermediate tree $\mathcal{T}_1$, which is a suitable refinement of $\mathcal{T}_0$. We then move the points from $\mathcal{T}_0$ into $\mathcal{T}_1$—the definition of $\mathcal{T}_1$ guarantees that any node receives only $O(k)$ points—and we locally refine any nodes that contain more than one point, giving us another intermediary tree $\mathcal{T}_2$. Finally, we prune $\mathcal{T}_2$ to obtain $\mathcal{T}(t+1)$. An example of the different stages of updating a quadtree of eight points is shown in Fig. 2. The individual steps are described in more detail below.

Refining the Tree: Computing $\mathcal{T}_1$ from $\mathcal{T}_0$. The intermediary tree $\mathcal{T}_1$ has the property that each region in $\mathcal{T}_1$ has $O(1)$ neighbors in $\mathcal{T}_0$. To make this precise we define for a node v in $\mathcal{T}_1$ some related internal or leaf nodes in $\mathcal{T}_0$.

- original(v): The lowest node u in $\mathcal{T}_0$ such that region(v) $\subseteq$ region(u).
- $\mathcal{N}(v)$: The collection $\mathcal{N}(v)$ of *neighbors* of v consists of all nodes $w \in \mathcal{T}_0$ that are at least as large as v (this to avoid multiple neighbors on one "side" of region(v)) and are adjacent to region(v). So $\mathcal{N}(v)$ consists of all nodes $w \in \mathcal{T}_0$ such that:

(*i*) size(w) $\geqslant$ size(v),
(*ii*) the regions region(w) and region(v) touch (that is, their boundaries intersect but their interiors are disjoint), and
(*iii*) w does not have a proper descendant w' with properties (i) and (ii) (so that w is a lowest node with properties (i) and (ii)).

Note that for every node in $\mathcal{T}_1$ we have $|\mathcal{N}(v)| \leqslant 3^d - 1$, hence $|\mathcal{N}(v)| = O(1)$ for any fixed dimension. We next modify $\mathcal{N}(v)$ and original(v) slightly, as follows. Any node $w_d \in \mathcal{N}(v) \cup \{\text{original}(v)\}$ that is a donut with a hole w_h that touches the boundary of region(v) and with size(w_h) $<$ size(v) is replaced by its parent (which is a compressed node). The set $\mathcal{N}(v)$ and original(v) will be used in our algorithm to determine if region(v) needs to be split—this is the case when it has a neighbor that is not a leaf or when original(v) is not a leaf—and later we need $\mathcal{N}(v)$ to know from which regions points can move into region(v).

It is important to remember that original(v) and the nodes in $\mathcal{N}(v)$ are nodes in $\mathcal{T}_0$, while v is a node in $\mathcal{T}_1$. Thus the neighbors never get refined which is needed to ensure that $\mathcal{T}_1$ has linear size. We now express the conditions on $\mathcal{T}_1$:

(*i*) For every node u in $\mathcal{T}_0$ such that region(u) is a hypercube, there is a node v in $\mathcal{T}_1$ with region(v) = region(u). Thus, the subdivision induced by $\mathcal{T}_1$ is a refinement of the subdivision induced by $\mathcal{T}_0$.
(*ii*) For each leaf v in $\mathcal{T}_1$, every node $u \in \mathcal{N}(v)$ is a leaf of $\mathcal{T}_0$.

We construct $\mathcal{T}_1$ top-down. Whenever we create a new node v of $\mathcal{T}_1$, we also store pointers from v to original(v) and to the nodes in $\mathcal{N}(v)$. (This is not explicitly stated in Algorithm 1 below.) We obtain these pointers from the parent of v and the original and neighbors of that parent in $O(1)$ time. How we refine each node v in $\mathcal{T}_1$ depends on original(v) and the neighbors in $\mathcal{N}(v)$. We distinguish three cases: in Case 1 at least one of the neighbors or original(v) is a regular node, in Case 2 no neighbor or original(v) is a regular node and at least one is a compressed node, and in Case 3 all neighbors as well as original(v) are leaf nodes. How we refine v in Case 2 depends on the holes of the compressed nodes in $\mathcal{N}(v) \cup \{\text{original}(v)\}$. To determine how to refine v we use a set H_v which is defined as follows. Let R be a hypercube and f a facet of any dimension of R. Let $\text{mirror}_f(R)$ denote a hypercube R^* for which size(R^*) = size(R) and $R^* \cap R = f$. We say that $\text{mirror}_f(R)$ is a mirrored version of R along f. The set H_v^* then consists of the mirrored hypercubes of every hole w_h of a compressed node in $\mathcal{N}(v)$ as well as the holes themselves. The set H_v is then defined as

$$H_v = \{R \mid R \in H_v^* \text{ and } R \subset \text{region}(v)\}.$$

Note that H_v is never empty, because a compressed node in $\mathcal{N}(v)$ must have a hole smaller than region(v) along the boundary of region(v). The different cases are described in more detail in Algorithm 1 and illustrated in Fig. 3.

Lemma 1. *The tree $\mathcal{T}_1$ created by Algorithm 1 is a compressed quadtree with properties (i) and (ii) stated above.*

Algorithm 1. REFINE($\mathcal{T}_0$)

Create a root node v for $\mathcal{T}_1$ with original$(v) = \text{root}(\mathcal{T}_0)$ and $\mathcal{N}(v) = \emptyset$.
Initialize a queue Q and add v to Q.
while Q *is not empty* **do**
  $v \leftarrow \text{pop}(Q)$
  Case 1: original(v) *or a neighbor in* $\mathcal{N}(v)$ *is a regular node.*
    Make v into a regular node, create 2^d children for it and add these to Q.
  Case 2: *Case 1 does not apply, and* original(v) *is a compressed node or a neighbor* $w \in \mathcal{N}(v)$ *is a compressed node.*
    $\sigma \leftarrow$ the smallest canonical hypercube containing the regions of H_v.
    Case 2a: $\sigma = \text{region}(v)$.
      Make v into a regular node, create 2^d children for it and add these to Q.
    Case 2b: $\sigma \subset \text{region}(v)$.
      Make v into a compressed node by creating two children for it, one child v_h corresponding to the hole in region(v) and one child v_d corresponding to the donut region, with $\text{region}(v_\text{h}) = \sigma$ and $\text{region}(v_\text{d}) = \text{region}(v) \setminus \sigma$. Add v_h to Q.
  Case 3: *Cases 1 and 2 do not apply;* original(v) *and every neighbor* $w \in \mathcal{N}(v)$ *are leaves in* $\mathcal{T}_0$.
    The node v remains a leaf.

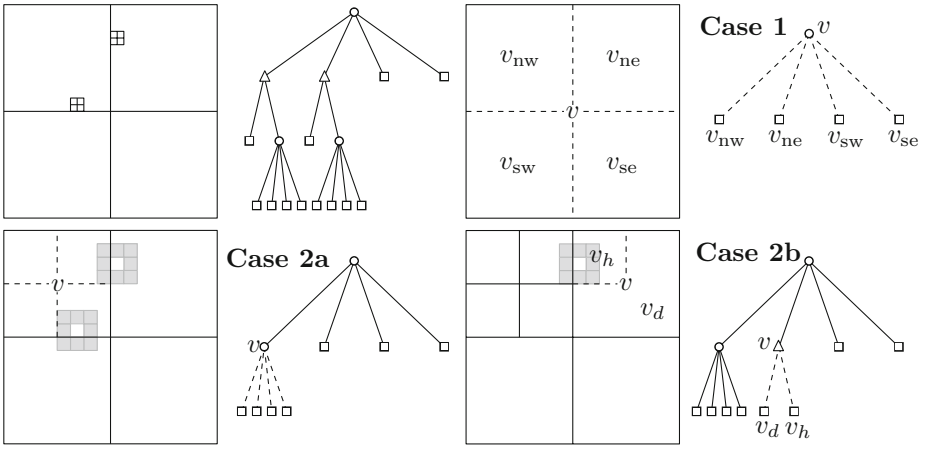

Fig. 3. Several steps from Algorithm 1. The tree on the top left is $\mathcal{T}_0$, the others illustrate consecutive steps in the construction of $\mathcal{T}_1$. The gray squares are the mirrored versions of the holes as used in Case 2.

Proof. First note that Algorithm 1 initially constructs a quadtree consisting of a single leaf. In the subsequent steps of the algorithm a leaf may be replaced by a compressed node or regular node, hence $\mathcal{T}_1$ is also a valid compressed quadtree.

For property (ii) it is sufficient to note that only in Case 3, where by definition the neighbors are leaves, the node v remains a leaf.

To prove property (i), suppose for a contradiction that there is a node u in $\mathcal{T}_0$ for which region(u) is a hypercube and there is no node v in $\mathcal{T}_1$ with region$(v) =$ region(u). Let v' be the lowest node in $\mathcal{T}_1$ such that region$(v') \supset$ region(u). Then original(v') is an ancestor of u. Hence, when v' is handled by Algorithm 1, either Case 1, 2a or 2b applies. Thus v' will be subdivided such that one of its children is a hypercube containing region(u), contradicting the fact that region(v') is the lowest node v' such that region$(v') \supset$ region(u). □

Note that Algorithm 1 must terminate, since every node in $\mathcal{T}_1$ is treated at most once and $\mathcal{T}_1$ will never contain nodes whose region is smaller than the smallest node in $\mathcal{T}_0$. We can actually prove the following stronger statement.

Lemma 2. *The tree $\mathcal{T}_1$ contains $O(n)$ nodes and can be constructed in $O(n)$ time, where n is the number of points in P.*

Moving the Points: Computing $\mathcal{T}_2$ from $\mathcal{T}_1$. For a node v we define $P_t(v)$ as the set of points in P contained in region(v) at time t. For each leaf v in $\mathcal{T}_1$ we find $P_{t+1}(v)$ by inspecting all nodes $w \in \{\text{original}(v)\} \cup \mathcal{N}(v)$—these nodes must be leaves of $\mathcal{T}_0$. Recall that $\mathcal{N}(v)$ contains a neighbor for every j-facet of region(v) $(0 \leqslant j < d)$, and that these neighbors have size at least $d_{\max}$. This implies that the regions region(w) for $w \in \{\text{original}(v)\} \cup \mathcal{N}(v)$, contain all the points that can possibly move into region(v). For each such node w we then check for all points in $P_t(w)$ if they are inside region(v) at time $t + 1$; if so we add them to $P_{t+1}(v)$. Using a charging scheme we can show that this takes $O(n)$ time in total. Afterwards some cells may contain more than one point and we compute the quadtree for these cells from scratch. Since every cell contains $O(k)$ points this can be done in $O(n \log k)$ time in total.

As a last step we prune the tree. We replace regular nodes in which only one child contains points by compressed nodes and merge nested compressed nodes. This is straightforward to do in linear time by traversing the tree in a bottom-up manner. The following theorem summarizes the result from this section.

Theorem 1. *Let P be a set of n moving points in $\mathbb{R}^d$. Under the Displacement Assumption, we can maintain a compressed quadtree for P in $O(n \log k)$ time per time step, where k is the parameter in the Displacement Assumption.*

Note that the $\log k$ factor comes purely from computing subtrees within a cell of size $d_{\max}$. If we are satisfied with maintaining $\mathcal{T}_0$, in which cells of size $d_{\max}$ are not subdivided, then updating takes only $O(n)$ time per time step.

3 Maintaining a Compressed Quadtree for Moving Objects

Let S be a set of n moving objects in $\mathbb{R}^d$. In this section we show how to maintain a compressed quadtree for S, based on our results from the previous section. We assume that each object $A \in S$ has constant complexity, so that we can

test in $O(1)$ time whether A intersects a given region (hypercube or donut) in a quadtree, and we can compute its axis-aligned bounding box $\mathrm{bb}(A)$ in $O(1)$ time. (For complex objects, we can either treat their constituent surface patches as the elementary objects, or we can work with a suitable simplification of the object.) Recall that the Displacement Assumption stipulates that $\mathrm{dist}(A(t), A(t+1)) \leqslant d_{\max}$ for any object $A \in S$ and any time t. For objects we interpret this as follows. If the objects only translate then the distance refers to the length of the translation vector. Otherwise, we require that the Hausdorff distance from $A(t+1)$ to $A(t)$ should not be more than $d_{\max}$—no point on $A(t+1)$ should be more than distance $d_{\max}$ from its nearest point in $A(t)$—and that no bounding-box vertex should move by more than $d_{\max}$. (Note that the bounding box is allowed to deform due to rotation or deformation of A.)

The idea of our approach is simple: We find a suitable subset $P^*(t)$—which is defined later—of the set $P(t)$ of $2^d n$ bounding-box vertices of the objects, such that every cell in the quadtree on $P^*(t)$ is intersected by a small number of objects. Our analysis will be done in terms of the so-called *density* of S, which is defined as follows. The density of a set S of objects is the smallest number λ such that any ball B intersects at most λ objects $A \in S$ with $\mathrm{diam}(A) \geqslant \mathrm{diam}(B)$ [6]. In the following, we assume that the density of $S(t)$ is at most λ, for any t. For $d = 2$ De Berg *et al.* [7] prove that each region in the subdivision induced by a compressed quadtree on the bounding-box vertices is intersected by $O(\lambda)$ objects. We extend this result to higher dimensions.

Lemma 3. *Let S be a set of n objects in $\mathbb{R}^d$, and let $\mathcal{T}$ be a compressed quadtree on the set of bounding-box vertices of S. Then* region(v) *intersects $O(\lambda)$ objects $A \in S$ for any leaf v of $\mathcal{T}$, where λ is the density of S.*

Proof (sketch). Assume v is a donut. (The case where v is a hypercube is easier.) It is known that any hypercube that does not contain any bounding-box vertices is intersected by $O(\lambda)$ objects [8]. Hence, it suffices to show that region(v) can always be covered by $O(1)$ hypercubes that lie inside the donut.

Let σ be the outer hypercube defining the donut region(v) and let σ_{in} be the inner hypercube (the hole). Our method to cover region(v) is as follows. A cell in a d-dimensional grid is surrounded by $3^d - 1$ other cells, that is, there are $3^d - 1$ other cells whose boundaries intersect the boundary of the given cell. We want to cover region$(v) = \sigma \setminus \sigma_{\mathrm{in}}$ using at most $3^d - 1$ hypercubes that surround σ_{in} in a somewhat similar manner. Consider two canonical hypercubes $\sigma_A := [a_1 : a'_1] \times \cdots \times [a_d : a'_d]$ and $\sigma_B := [b_1 : b'_1] \times \cdots \times [b_d : b'_d]$ of the same size, and let $(\alpha_1, \ldots, \alpha_d) \in \{-, 0, +\}^d$. We say that σ_A is an $(\alpha_1, \ldots, \alpha_d)$-*neighbor* of σ_B if the following holds:

- if $\alpha_i = -$ then $a'_i = b_i$
- if $\alpha_i = 0$ then $[a_i : a'_i] = [b_i : b'_i]$
- if $\alpha_i = +$ then $a_i = b'_i$

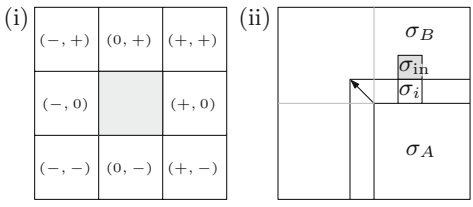

Fig. 4. (i) Labeling the neighbors of a cell. (ii) Growing σ_A until it contains σ_i.

In other words, α_i indicates the relative position of σ_A and σ_B in the i-th dimension. Fig. 4 illustrates this in the plane.

Now consider the donut $\sigma \setminus \sigma_{\text{in}}$ that we want to cover, and let j be such that $\text{diam}(\sigma) = 2^j \cdot \text{diam}(\sigma_{\text{in}})$. We will prove by induction on j that we can cover any donut region in a compressed quadtree by a collection C of hypercubes. The hypercubes in the constructed collection C will have a label from $\{-, 0, +\}^d$, where each label is used at most once—except $(0, \ldots, 0)$ which will not be used at all—and the following invariant is maintained:

> *Invariant:* If a hypercube $\sigma_i \in C$ has label $(\alpha_1, \ldots, \alpha_d)$, then there are canonical hypercubes σ_A and σ_B of the same size such that $\sigma_A \subset \sigma_i$ and $\sigma_{\text{in}} \subset \sigma_B$ and σ_A is an $(\alpha_1, \ldots, \alpha_d)$-neighbor of σ_B.

The invariant is maintained by stretching the hypercubes of octants that do not contain σ_{in} until they hit σ_{in}. These stretched hypercubes then replace the ones with the same labels in C. □

In the lemma above we consider a compressed quadtree on the set $P(t)$ of all bounding-box vertices. In our method we use a subset $P^*(t) \subset P(t)$ to define the compressed quadtree. Furthermore, to update the quadtree efficiently we cannot afford to refine the quadtree until each node contains only one point—even when the objects adhere to the Displacement Assumption, there can be $\Omega(n)$ bounding-box vertices in a hypercube of size $d_{\max}$. Instead we use the same stopping criterion as we did for $\mathcal{T}_0$ for a set of points: a hypercube σ is a leaf in the compressed quadtree when (i) σ contains at most one point from $P^*(t)$, or (ii) σ has edge length $d_{\max}$. Next we define $P^*(t)$.

We define the *Z-order* of $P(t)$ as the order in which the points are visited while doing an in-order treewalk on a non-compressed quadtree on $P(t)$ where for each interior node the children are visited in a specific order. (In two dimensions this order is: top left, top right, bottom left and bottom right.) For any set of n points such a Z-order can be computed in $O(n \log n)$ time [13] as long as a smallest canonical square of two points can be computed in $O(1)$ time. Ideally our subset $P^*(t)$ consists of every λ^{th} point in this Z-order, but for cells of size $d_{\max}$ (which can contain many points) we cannot afford to compute the exact Z-order. Instead we define an *approximate Z-order* as a Z-order where points within a cell of size $d_{\max}$ have an arbitrary ordering. The set $P^*(t)$ then consists of every λ^{th} point on such an approximate Z-order. This implies that $P^*(t)$ is not uniquely defined, but each choice of $P^*(t)$ leads to the same quadtree, since cells of size $d_{\max}$ are not refined further. In the rest of the section we use $\mathcal{T}^*(t)$ to denote the compressed quadtree on $P^*(t)$ with the above mentioned stopping criterion. Since $P^*(t)$ contains $O(n/\lambda)$ points, $\mathcal{T}^*(t)$ has $O(n/\lambda)$ nodes. Next we bound the number of objects that intersect a cell in $\mathcal{T}^*(t)$.

Lemma 4. *Every cell in the subdivision induced $\mathcal{T}^*(t)$ is intersected by $O(\lambda+k)$ objects from $S(t)$.*

Proof. Let σ be an arbitrary cell in the subdivision induced by $\mathcal{T}^*(t)$. If σ has edge length $d_{\max}$ then by the Displacement Assumption it intersects $O(k)$ objects. If σ is larger than $d_{\max}$ it can either be a hypercube or a donut. If σ is a

hypercube, then by definition all points of $P(t)$ contained in σ must be consecutive in the approximate Z-order. It follows that σ contains at most 2λ points of $P(t)$. If σ is a donut, then the points of $P(t)$ contained in σ can be divided into two sets of consecutive points from the approximate Z-order. Since the donut does not contain any point of $P^*(t)$ it contains at most 2λ points of $P(t)$. In either case, by Lemma 3, σ is intersected by $O(\lambda)$ objects for which there is no bounding-box vertex in σ and, hence, by $O(\lambda)$ objects in total. □

We maintain $\mathcal{T}^*(t)$ in the same way as we did for points, but we also maintain for each cell a list of objects that intersect that cell and we have to update $P^*(t)$ at each time step. This leads to two additional steps in our algorithm.

We compute $P^*(t+1)$ just after we have moved the points of $P(t+1)$, but before we do any local refining. We obtain $P^*(t+1)$ by traversing $\mathcal{T}_1$ (in Z-order) and for each cell larger than $d_{\max}$ computing the local Z-order of the points of $P(t+1)$ that are contained in the cell. Since each such cell contains $O(\lambda)$ points this takes $O(n \log \lambda)$ time in total. We then locally refine each cell that contains more than one point of $P^*(t+1)$ until it adheres to the stopping criterion.

We also have to construct for each leaf v a list $S_{t+1}(v)$ of objects that intersect region(v) at time $t+1$. We compute $S_{t+1}(v)$ in the same way as we computed $P_{t+1}(v)$, by inspecting the objects that intersect neighbors and then testing which of these objects intersect region(v). To do this in $O(1)$ time per object we need the assumption that objects have constant complexity. Since $\mathcal{T}_2$ still has $O(n/\lambda)$ nodes we can do this in $O(n)$ time.

The resulting tree adheres to the stopping criterion with respect to $P^*(t+1)$ and each leaf has a list of objects that intersect it. We then prune the tree to obtain $\mathcal{T}^*(t+1)$. We can conclude with the following theorem.

Theorem 2. *Let S be a set of n moving objects in $\mathbb{R}^d$. Under the Displacement Assumption, we can maintain a compressed quadtree for S in $O(n \log \lambda)$ time per time step, such that each leaf region intersects $O(\lambda + k)$ objects, where λ is an upper bound on the density of S at any time and k is the parameter in the Displacement Assumption. The compressed quadtree has $O(n/\lambda)$ nodes, and the sum of the number of objects over all the leafs is $O(n)$.*

Notice that the parameter k does not show up in the update time, unlike for points. The reason for this is that we stop the construction when the size of a region drops below $d_{\max}$. The parameter k does however show up when we look at the number of potentially colliding objects. For each leaf v in $\mathcal{T}^*(t)$ we report all pairs of objects in $S_t(v)$. Observe that $|S_t(v)| = O(\lambda + m_v)$ where m_v is the number of bounding-box vertices in region(v). Summing $|S_t(v)|^2$ over all nodes v in $\mathcal{T}^*(t)$ we get $\sum_{v \in \mathcal{T}_0} |S_t(v)|^2 = O(\lambda + k) \sum_{v \in \mathcal{T}_0} O(\lambda + m_v) = O((\lambda + k)n)$.

Corollary 1. *Let S be a set of n objects with density λ. Under the Displacement Assumption, we can maintain a compressed quadtree on S in $O(n)$ time per time step that can be used for broad-phase collision detection, reporting $O((\lambda + k)n)$ pairs of potentially colliding objects.*

References

1. Abam, M.A., de Berg, M., Poon, S.-H., Speckmann, B.: Kinetic collision detection for convex fat objects. Algorithmica 53(4), 457–473 (2009)
2. Agarwal, P.K., Basch, J., Guibas, L.J., Hershberger, J., Zhang, L.: Deformable free-space tilings for kinetic collision detection. Int. J. Robotics Research 21(3), 179–197 (2002)
3. Aluru, S., Sevilgen, F.E.: Dynamic compressed hyperoctrees with application to the N-body problem. In: Proc. 19th Conf. Found. Softw. Tech. Theoret. Comput. Sci., pp. 21–33 (1999)
4. Basch, J., Guibas, L.J., Hershberger, J.: Data structures for mobile data. In: Proc. 8th ACM-SIAM Symp. Discr. Alg., pp. 747–756 (1997)
5. de Berg, M.: Linear size binary space partitions for uncluttered scenes. Algorithmica 28, 353–366 (2000)
6. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer (2008)
7. de Berg, M., Haverkort, H., Thite, S., Toma, L.: Star-quadtrees and guard-quadtrees: I/O-efficient indexes for fat triangulations and low-density planar subdivisions. Comput. Geom. Theory Appl. 43, 493–513 (2010)
8. de Berg, M., Katz, M.J., van der Stappen, A.F., Vleugels, J.: Realistic input models for geometric algorithms. Algorithmica 34(1), 81–97 (2002)
9. de Berg, M., Roeloffzen, M., Speckmann, B.: Kinetic convex hulls and Delaunay triangulations in the black-box model. In: Proc. 27th ACM Symp. Comput. Geom., pp. 244–253 (2011)
10. Borro, D., Garcia-Alonso, A., Matey, L.: Approximation of optimal voxel size for collision detection in maintainability simulations within massive virtual environments. Comp. Graph. Forum 23(1), 13–23 (2004)
11. Chazelle, B.: Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. SIAM J. Comput. 13, 488–507 (1984)
12. Gao, J., Guibas, L.J., Nguyen, A.: Deformable spanners and applications. In: Proc. 20th ACM Symp. Comput. Geom., pp. 190–199 (2004)
13. Har-Peled, S.: Geometric Approximation Algorithms. American Mathematical Society (2011)
14. Kim, D.-J., Guibas, L.J., Shin, S.Y.: Fast collision detection among multiple moving spheres. IEEE Trans. Vis. Comp. Gr. 4, 230–242 (1998)
15. Kirkpatrick, D., Snoeyink, J., Speckmann, B.: Kinetic collision detection for simple polygons. Int. J. Comput. Geom. Appl. 12(1-2), 3–27 (2002)
16. Kockara, S., Halic, T., Iqbal, K., Bayrak, C., Rowe, R.: Collision detection: A survey. In: Proc. of Systems, Man and Cybernetics, pp. 4046–4051 (2007)
17. Lin, M., Gottschalk, S.: Collision detection between geometric models: A survey. In: Proc. of IMA Conf. Math. Surfaces, pp. 37–56 (1998)
18. Moore, M., Wilhelms, J.: Collision detection and response for computer animation. SIGGRAPH Comput. Graph. 22, 289–298 (1988)
19. Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., Cani, M., Faure, F., Thalmann, M.N., Strasser, W., Volino, P.: Collision detection for deformable objects. Comp. Graph. Forum 24, 119–140 (2005)