Kinetic collision detection for low-density scenes in the black-box model

Kinetic Collision Detection for Low-Density Scenes in the Black-Box Model

Mark de Berg* Marcel Roeloffzen* Bettina Speckmann*

Abstract

We present an efficient method for collision detection in the *black-box KDS model* for a set S of n objects in the plane. In this model we receive the object locations at regular time steps and we know a bound $d_{\max}$ on the maximum displacement of any object within one time step. Our method maintains, in $O((\lambda+k)n)$ time per time step, a compressed quadtree on the bounding-box vertices of the objects; here λ denotes the density of S and k denotes the maximum number of objects that can intersect any disk of radius $d_{\max}$. Collisions can then be detected by testing $O((\lambda+k)^2 n)$ pairs of objects for intersection.

1 Introduction

Collision detection [12, 13] 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 which occur. In practice collision detection is often performed in two phases: a *broad phase* that serves as a filter and reports a (small) set of potentially colliding pairs of objects, and a *narrow phase* that tests each of these pairs to determine if there is indeed a collision. Here we are concerned only with broad-phase collision detection; more information on the narrow phase can be found in a survey by Kockara *et al.* [11].

Related work. The most common way to perform collision detection is to 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. In addition, even with small time steps, collisions can be missed.

An alternative is to use the *kinetic-data-structure (KDS) framework* introduced by Basch *et al.* [3]. 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.) KDSs for collision detection have been proposed for 2D collision detection among polygonal objects [2, 10], for 3D collision detection among spheres [9], 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* [5, 7]: We receive, for each object $A_i \in S$, its location $A_i(t)$, at regular time steps $t = 1, 2, \ldots$ and we know an upper bound on the maximum displacement $d_{\max}$ of any object within one time step. Our main goal is to obtain provable bounds for broad-phase collision detection in the black-box model.

Results. We present an algorithm for maintaining a compressed quadtree on the set S of objects, and we prove that our algorithm runs in $O((\lambda+k)n)$ time per time step; here λ denotes the density [6] of S, and k denotes the maximum number of objects intersecting any disk of radius $d_{\max}$. The compressed quadtree can be used to report the at most $O((\lambda+k)^2 n)$ potentially colliding pairs of objects to the narrow phase. The basis of our algorithm is a technique to efficiently maintain a compressed quadtree for a set of moving points, which is of independent interest. We describe our results for the planar case, but they generalize to 3- or higher-dimensional space in a straightforward manner.

2 Preliminaries

Assumptions on distribution and displacement. Let S be the set of constant complexity objects in the plane—e.g. a set of triangles or disks—, let $A_j(t)$ be the object $A_j \in S$ at time t, and let

*Department of Computer Science, TU Eindhoven, the Netherlands, {mdberg,mroeloff,speckman}@win.tue.nl.
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.

$S(t) = \{A_1(t), \ldots, A_n(t)\}$. To use temporal coherence in our algorithm, we need bounds on the maximum distance the objects can move in relation to their inter-distances—otherwise the locations at time t have no relation to those at time $t+1$ and we can do nothing but compute $\mathcal{T}(t)$ from scratch. Following De Berg *et al.* [5] we make the following assumption.

Displacement Assumption: There is a maximum displacement $d_{\max}$ such that $\mathrm{dist}(A(t), A(t+1)) \leqslant d_{\max}$ for each object $A \in S$ and any time step t.

For simplicity we assume the objects only translate, and we define $\mathrm{dist}(A(t), A(t+1))$ as the length of the translation vector between $A(t)$ and $A(t+1)$. However, our algorithm also works in more general cases, as long as the boundaries of the objects do not move too much. As mentioned, we need to relate the maximum displacement to the inter-object distances:

Distribution Assumption: Any disk of radius $d_{\max}$ intersects at most k objects from $S(t)$, at any time step t.

Our algorithms do not know the value of k; it is used only in the analysis. We also use the concept of *density* [6]. A set S of objects has density λ if, for any disk D, the number of objects $A_j \in S$ intersecting D having $\mathrm{diam}(A_j) \geqslant \mathrm{diam}(D)$ is at most λ. We assume that the density of $S(t)$ is λ at every time step t.

The compressed quadtree. A *quadtree* for a set of points inside a square is a tree representing a subdivision of that square into four equal-sized subsquares (quadrants) that continues recursively until a stopping criterion is met. There can be splits where only one of the four resulting quadrants contains points. In the quadtree this corresponds to a path of nodes with only one non-empty child. A *compressed quadtree* replaces such paths by *compressed nodes*, which have two children: a child for the *hole* representing the smallest quadtree square containing all points, and a child for the *donut* representing the rest of the square. A compressed quadtree for a set of points has linear size. Our main data structure is a compressed quadtree $\mathcal{T}$ on the set P of bounding-box vertices of the objects in S, with the following stopping criterion.

Stopping Criterion: A square σ becomes a leaf when (i) σ contains at most one point from P, or (ii) σ has edge length at most $d_{\max}$.

We use region(v) to denote the region associated with a node v of $\mathcal{T}$ and assume that region(root($\mathcal{T}$)) is a fixed, large square that always contains all objects.

Observation 1 *Under our Stopping Criterion,* region(v) *intersects $O(\lambda + k)$ objects from S for any leaf v.*

The leaf regions of $\mathcal{T}$ intersect only few objects: regions with edge length larger than $d_{\max}$ contain at most one bounding-box vertex and, hence, intersect $O(\lambda)$ objects [4], and regions with edge length at most $d_{\max}$ intersect at most $O(k)$ objects by definition of k.

For each leaf v in $\mathcal{T}$ we maintain a list of objects intersecting region(v). Broad-phase collision detection is then performed by reporting for each leaf v all $O((\lambda + k)^2)$ pairs of objects intersecting region(v). Since the tree $\mathcal{T}$ contains $O(n)$ nodes we report at most $O((\lambda + k)^2 n)$ pairs.

A compressed quadtree for a set of n points can be constructed in $O(n \log n)$ time [8]. This holds in an appropriate model of computation, where we can find the smallest canonical square—a canonical square is any square that results from recursive subdivision of the given initial square—containing two given points in $O(1)$ time. Our goal is to show that, in this model of computation, we can efficiently maintain our compressed quadtree as the objects move. We use $\mathcal{T}(t)$ to denote the compressed quadtree on the bounding-box vertices of $S(t)$. In the remainder of this abstract we sketch how to create $\mathcal{T}(t+1)$ from $\mathcal{T}(t)$ in $O((\lambda+k)n)$ time, resulting in the following theorem.

Theorem 1 *Let S be a set of n moving objects in the plane that adheres to the Displacement and Distribution Assumption and with maximum density λ. We can maintain a compressed quadtree for S in $O((\lambda + k)n)$ time per time step, which allows us to perform broad-phase collision detection resulting in $O((\lambda + k)^2)$ pairs of potentially colliding objects.*

3 Maintaining the compressed quadtree

Our compressed quadtree is built on the points in P (the bounding-box vertices). The main problem in updating $\mathcal{T}$ 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 therefore too expensive. We solve this by first refining $\mathcal{T}(t)$ into an intermediary tree $\mathcal{T}_1(t)$. We then insert the (moved) points into $\mathcal{T}_1(t)$ to obtain $\mathcal{T}_2(t)$. We insert the (moved) objects into $\mathcal{T}_2(t)$ to obtain $\mathcal{T}_3(t)$ which we prune into $\mathcal{T}(t+1)$ (see Fig 1). Below we describe these steps in more detail. Note that $\mathcal{T}(t)$ remains unchanged, whereas $\mathcal{T}_1(t)$ is first constructed and then changed into $\mathcal{T}_2(t)$, $\mathcal{T}_3(t)$ and $\mathcal{T}(t+1)$.

Refine. The intermediary tree $\mathcal{T}_1(t)$ has the property

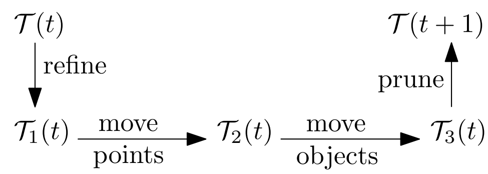

Figure 1: Constructing $\mathcal{T}(t+1)$ from $\mathcal{T}(t)$.

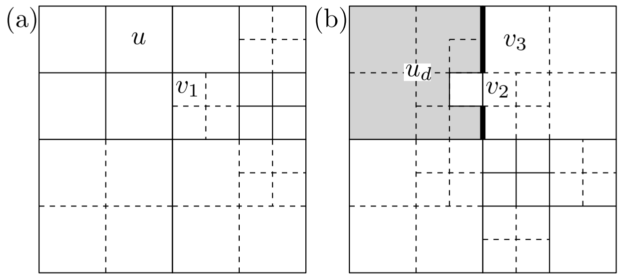

Figure 2: Subdivision defined by $\mathcal{T}(t)$ and its refinement (dashed lines), u_d and u are nodes of $\mathcal{T}(t)$, v_1, v_2, and v_3 are nodes of $\mathcal{T}_1(t)$, region(u_d) is a donut (gray), its right boundary consists of two segments (fat).

that each region in $\mathcal{T}_1(t)$ has $O(1)$ neighbor regions in $\mathcal{T}(t)$. To make this precise we define for a node v in $\mathcal{T}_1(t)$ some related internal or leaf nodes in $\mathcal{T}(t)$:

- original(v): The lowest node u in $\mathcal{T}(t)$ such that region(v) $\subseteq$ region(u) and region(u) is a square.
- $\mathrm{nbr_n}(v)$, $\mathrm{nbr_e}(v)$, $\mathrm{nbr_s}(v)$, $\mathrm{nbr_w}(v)$: We call these the horizontal and vertical neighbors of v in $\mathcal{T}(t)$. We define the west neighbor $\mathrm{nbr_w}(v)$ as the lowest node u in $\mathcal{T}(t)$ such that the left boundary of region(v) is included in the right boundary of region(u) (in Fig. 2(b) $\mathrm{nbr_w}(v_3) = u_d$). The other vertical and horizontal neighbors are defined similarly. (Not all neighbors need to exist.)
- $\mathrm{nbr_{nw}}(v)$, $\mathrm{nbr_{ne}}(v)$, $\mathrm{nbr_{se}}(v)$, $\mathrm{nbr_{sw}}(v)$: These are the diagonal neighbors of v in $\mathcal{T}(t)$. We define $\mathrm{nbr_{nw}}(v)$ as the lowest node u in $\mathcal{T}(t)$ such that the north-west corner of region(v) is the south-east corner—or a corner of a donut cell created by the corresponding hole—of region(u) (in Fig. 2(a) $\mathrm{nbr_{nw}}(v_1) = u$ and in Fig. 2(b) $\mathrm{nbr_{nw}}(v_2) = u_d$).
- $\mathrm{nbr_{h1}}(v), \mathrm{nbr_{h2}}(v)$: These neighbors exist only for donut cells with a hole along the boundary and are the diagonal neighbors of the corner points along the boundary that are created by the hole.

For ease of notation we define the following sets:

$$\mathcal{N}_{\mathrm{hv}}(v) = \{\mathrm{nbr_n}(v), \mathrm{nbr_e}(v), \mathrm{nbr_s}(v), \mathrm{nbr_w}(v)\}$$
$$\mathcal{N}_{\mathrm{d}}(v) = \{\mathrm{nbr_{nw}}(v), \mathrm{nbr_{ne}}(v), \mathrm{nbr_{se}}(v), \mathrm{nbr_{sw}}(v), \mathrm{nbr_{h1}}(v), \mathrm{nbr_{h2}}(v)\}$$

We can now express the conditions on $\mathcal{T}_1(t)$:

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

(*ii*) For each leaf v in $\mathcal{T}_1(t)$ every node $u \in \mathcal{N}_{\mathrm{hv}}(v)$ is a leaf of $\mathcal{T}(t)$.

We construct $\mathcal{T}_1(t)$ top-down. Whenever we create a new node v of $\mathcal{T}_1(t)$, it receives a pointer to original(v) and pointers to its horizontal and vertical neighbors in $\mathcal{T}(t)$ (the nodes in $\mathcal{N}_{\mathrm{hv}}(v)$). These are obtained from the parent of v and the original and neighbors of that parent. How we refine each node v in $\mathcal{T}_1(t)$ depends on original(v) and the neighbors in $\mathcal{N}_{\mathrm{hv}}(v)$ and is described in more detail in Algorithm 1.

We can prove that each of the cases occurs at most $O(n)$ times and, hence, $\mathcal{T}_1(t)$ also contains $O(n)$ nodes. Besides the pointers to the nodes in $\mathcal{N}_{\mathrm{hv}}(v)$, we also need pointers to the set $\mathcal{N}_{\mathrm{d}}(v)$ of diagonal neighbors of each node v in $\mathcal{T}_1(t)$. The details of this are not difficult and omitted due to space limitations.

Moving the bounding-box vertices. We first create for each leaf v in $\mathcal{T}_1(t)$ a list of all points in P contained in region(v) at time $t+1$. We traverse $\mathcal{T}_1(t)$ and for each leaf v we encounter we inspect original(v) and all its neighbors in $\mathcal{N}_{\mathrm{hv}}(v) \cup \mathcal{N}_{\mathrm{d}}(v)$—recall that these neighbors are leaves of $\mathcal{T}(t)$. The points contained in these at most eleven leaves—ten neighbors and one original—are the only ones that can be in region(v) at time $t+1$ as points in other cells have more than distance $d_{\max}$ to region(v).

For each of the points in these eleven nodes we check if they are inside region(v) and if so we add them to v. This takes constant time assuming each of these nodes contains only one point. Due to the definition of our quadtree there can be nodes containing more than one point, corresponding to squares that were not refined because their edge length is $d_{\max}$. Fortunately, these nodes can only be neighbor or original to at most nine nodes in $\mathcal{T}_1(t)$. Each point in these cells is inspected at most nine times and hence points from these cells only require $O(n)$ time in total.

After moving points into v there may be more than one point in v. To ensure that the tree still adheres to the Stopping Criterion we refine v. Since the points come from a constant number of nodes they occupy a constant number of cells of size $d_{\max}$. Building a compressed quadtree on these cells takes constant time. The result is the second intermediary tree $\mathcal{T}_2(t)$.

Moving the objects. The objects are moved in the same way as the points. We traverse $\mathcal{T}_2(t)$ and for each node v we test each object from $S(t)$ intersecting a neighbor or original of v for intersection with region(v). Since $\mathcal{T}(t)$ adheres to the Stopping Criterion at time t, it follows from Observation 1 that we test only $O(\lambda + k)$ objects for each node in $\mathcal{T}_2(t)$. This results in the intermediary tree $\mathcal{T}_3(t)$.

Pruning the tree. We finally prune $\mathcal{T}_3(t)$ to remove any unnecessary compressed nodes or splits. Splits that put all vertices into one child are replaced by compressed nodes and nested compressed nodes—where the hole of one compressed node is another compressed node—are reduced to a single compressed node. This results in $\mathcal{T}(t+1)$, the compressed quadtree for the objects at time $t+1$.

Algorithm 1: REFINE($\mathcal{T}(t)$)

Create root($\mathcal{T}_1(t)$). Set $\mathcal{N}_{\mathrm{hv}}(\mathrm{root}(\mathcal{T}_1(t))) = \emptyset$ and original(root($\mathcal{T}_1(t)$)) = root($\mathcal{T}(t)$);
Add root($\mathcal{T}_1(t)$) to empty queue Q;
while Q *is not empty* **do**
    $v \leftarrow \mathrm{pop}(Q)$;
    Case 1: original(v) **is split OR a neighbor in** $\mathcal{N}_{\mathrm{hv}}(v)$ **is split:** (see figure) v becomes a split node and we create four children $v_{\mathrm{nw}}, v_{\mathrm{ne}}, v_{\mathrm{se}}, v_{\mathrm{sw}}$ which we add to Q;
    Case 2: original(v) **is a compressed node OR a neighbor in** $\mathcal{N}_{\mathrm{hv}}(v)$ **has a hole on shared boundary:**
        mirror the top level square of each adjacent hole into region(v) and mirror the hole of original(v) along its top, left, bottom and right boundary;
        scs $\leftarrow$ the smallest canonical square of the mirrored squares in region(v);
        Case 2a: *scs* **is empty:** v remains a leaf;
        Case 2b: $scs = \mathrm{region}(v)$**:** v becomes a split node and we create four children $v_{\mathrm{nw}}, v_{\mathrm{ne}}, v_{\mathrm{se}}, v_{\mathrm{sw}}$ which we add to Q;
        Case 2c: $scs \subset \mathrm{region}(v)$**:** v becomes a compressed node and we create v_{d} and v_{h} such that $\mathrm{region}(v_{\mathrm{d}}) = \mathrm{region}(v) \backslash scs$ and $\mathrm{region}(v_{\mathrm{h}}) = scs$. Add v_{h} to Q;
    Case 3: otherwise: v remains a leaf;

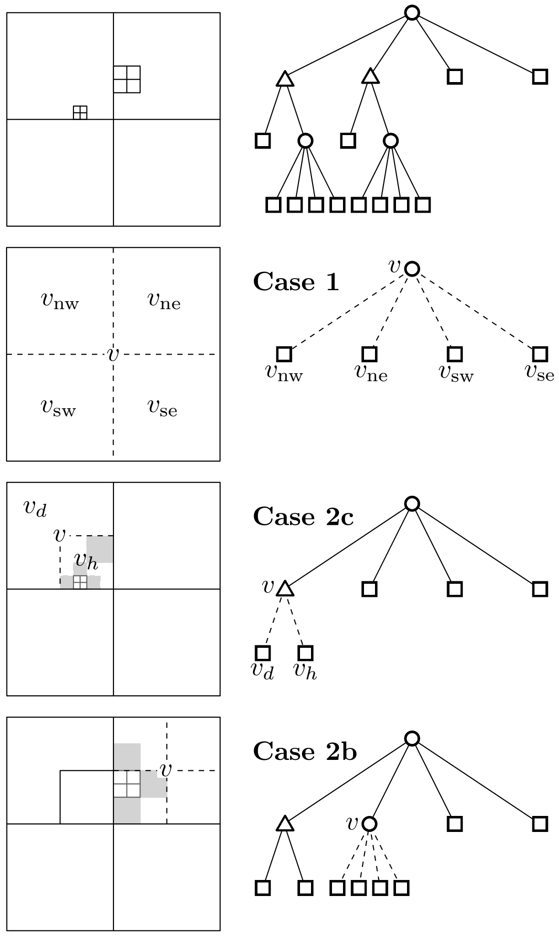

Figure 3: Algorithm 1 (left) to construct $\mathcal{T}_1(t)$ and an illustration (right) of several consecutive steps of the algorithm showing the Cases 1, 2b and 2c. The top tree is the input tree $\mathcal{T}(t)$.

References

[1] M.A. Abam, M. de Berg, S.-H. Poon, and B. Speckmann. Kinetic collision detection for convex fat objects. *Algorithmica*, 53(4):457–473, 2009.

[2] P.K. Agarwal, J. Basch, L.J. Guibas, J. Hershberger, and L. Zhang. Deformable free-space tilings for kinetic collision detection. *Int. J. Robotics Research*, 21(3):179–197, 2002.

[3] J. Basch, L.J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Symp. Discr. Alg.* , pages 747–756, 1997.

[4] M. de Berg, H. Haverkort, S. Thite, and L. Toma. 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.

[5] M. de Berg, M. Roeloffzen and B. Speckmann. Kinetic convex hulls and Delaunay triangulations in the black-box model. In *Proc. 27th ACM Symp. Comput. Geom.*, pages 244–253, 2011.

[6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications (3rd edition).* Springer, 2008.

[7] J. Gao, L.J. Guibas, A. Nguyen. Deformable spanners and applications. In *Proc. 20th ACM Symp. Comput. Geom.*, pages 190–199, 2004.

[8] Sariel Har-Peled. *Geometric Approximation Algorithms.* American Mathematical Society, 2011.

[9] D.-J. Kim, L.J. Guibas, and S.Y. Shin. Fast collision detection among multiple moving spheres. *IEEE Trans. Vis. Comp. Gr.*, 4:230–242 (1998).

[10] D. Kirkpatrick, J. Snoeyink, and B. Speckmann. Kinetic Collision Detection for Simple Polygons. *Int. J. Comput. Geom. Appl.*, 12(1-2):3–27 (2002).

[11] S. Kockara, T. Halic, K. Iqbal, C. Bayrak and R. Rowe. Collision detection: A survey. In *Proc. of SMC*, pages 4046–4051, 2007.

[12] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *Proc. of IMA Conf. Math. Surfaces*, pages 37–56, 1998.

[13] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M. Cani, F. Faure, Magnenat N. Thalmann, W. Strasser and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24:119–140, 2005.