SIGRAD 2016
M. Hayashi (Editors)

Output Sensitive Collision Detection for Unisize Boxes

Gabriele Capannini and Thomas Larsson

Mälardalen University, Sweden

Abstract

We show how a recent collision detection method, which is based on the familiar sweep and prune concept, can gain improved performance for the special class of simulations that only involves axis-aligned bounding boxes of the same size. The proposed modifications lead to a worst-case optimal output-sensitive algorithm in 2D. Furthermore, the experimental result shows that our method gives generous speedups in practice and that dynamic scenes with one million objects can be processed at interactive rates even on a laptop.

Categories and Subject Descriptors (according to ACM CCS): F.2.2 [Analysis Of Algorithms And Problem Complexity]: Nonnumerical Algorithms and Problems—Geometrical problems and computations; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures;

1. Introduction

Fast collision detection is a required operation in many types of physical simulation, video games, and computer graphics applications. It is also an important operation in virtual reality systems and robotics. Clearly, to determine all contacts in a virtual environment with n moving bodies or geometrical objects at interactive rates is a challenging computational problem.

In 1992, Baraff introduced a well-known collision detection algorithm which is now known as the Sweep and Prune (SaP) method [Bar92]. The original algorithm has later been improved to deal with larger inputs and more challenging scenarios, for instance, by using extended data structures and parallelization [TBW09, LHLK10]. A recent variant gains efficiency through a dual-axis sweeping approach to defeat bottlenecks arising from large chunks of overlapping intervals along the projection axes [CL16].

In what follows, we propose a new variant of this dual-axis approach which uses an improved data structure to speed-up the SaP computation when all the simulated objects can be enclosed in equally sized axis-aligned bounding boxes (AABBs). Thus, we study the problem: Given n dynamic axis-aligned boxes of the same size, report all pairwise intersections. In particular, we are able to give an algorithm that runs in $O(n \cdot \log n + |C|)$ time, where $|C|$ is the size of the output, given simulation scenarios that essentially are two-dimensional. Furthermore, it is straightforward to apply our method also in fully three-dimensional simulations with an expected high performance in practice, but without the stated theoretical guarantee.

2. Background

Collision detection is a well-studied topic in the computer graphics community. What algorithm that is preferable depends on several factors such as the geometric representation, nature of body motions, type of query, required accuracy, and overall scene complexity. Application-specific knowledge can often be exploited to accelerate the process. Therefore, numerous algorithms and data structures have been presented for varying circumstances, contexts, and applications.

Strategies based on sorting or bucketing are often used, for example SaP, uniform grids, and hashing techniques. Many other approaches involve the use of hierarchical data structures, such as bounding volume hierarchies, k-d-trees, quadtrees, and octrees [Sam05]. However, it is beyond the scope of this study to discuss and evaluate such techniques. For a broader view of the collision detection problem, interested readers may turn to existing surveys (see e.g. [Eri04, TKH*05] and chapter 2 in [Wel13]).

Our efforts have been focused on finding efficient variants of SaP. Besides the initial research efforts [Bar92,CLMP95], several attempts have been made to improve the efficiency of SaP. Simulations of large datasets require a mechanism to handle the complexity arising from the growth of the number of interval overlaps along the projection axes. Therefore,

several recent methods use a combination of SaP and spatial subdivision [TBW09, LHLK10]. Alternatively, the SaP method can also be enhanced by realizing efficient range queries during the sweeping of a secondary axis [CL16].

Clearly, variants of SaP have been successfully applied to different kinds of multi-body simulations with varying scene complexity, object representation, and type of motion [CS06, TBW09, CL16]. Thus, the SaP family of methods appears to be quite general and powerful. In the next section, we show how SaP can be further optimized in cases where the sizes of the simulated objects are bounded in such a way that they can be approximated by unisize boxes.

3. Our Algorithm

This section introduces the proposed approach to SaP for 2D scenarios and equally sized AABBs. In this type of computation, boundaries of each box project on each coordinate axis an interval $a = [a^+, a^-]$, such that $a^+ < a^-$ and $|a| = a^- - a^+$ denotes the length of a. At each frame of the simulation, objects move so that their AABB projections can overlap and, when the projections of two boxes overlap on all axes, the AABBs collide. Clearly, two intervals a and b overlap iff:

$$a^+ \leq b^- \wedge b^+ \leq a^- \tag{1}$$

As Figure 1 shows, any two overlapping intervals a and b satisfy Equation 1.

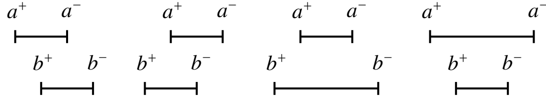


Figure 1: *There are four cases for which a and b overlap.*

The original SaP algorithm consists in sorting the box boundaries (a.k.a. *endpoints*) related to a coordinate axis, then sweeping the result to find out the colliding box pairs [Bar92]. An object a is added to a list (a.k.a. *activelist*) when its *low-endpoint* a^+ is picked and it is successively removed once its *high-endpoint* a^- is encountered. Furthermore, when the object is removed from the activelist, a set of full box-box tests is performed to discover possibly collisions between the removed object and the ones remaining in the activelist. During this phase, many false positives are encountered, namely the objects which do not intersect the one removed on the remaining axis. In a previous paper, we showed how our dual-axis approach can reduce the number of false positives substantially in 3D [CL16]. We propose, now, an approach which completely avoid false positives.

Claim 1 *By means of a stable sorting algorithm and under the unisize assumption, the endpoints projected on a coordinate axis can be ordered in such a way that if two intervals a and b overlap then:* $a^+ \prec b^+ \prec a^- \prec b^-$ *or* $b^+ \prec a^+ \prec b^- \prec a^-$.

In other words, Claim 1 states that the AABB intervals can be represented so that any of them does not fully include another one in the list of sorted endpoints. As a consequence, given an interval a, the corresponding set of overlapping intervals on the first axis is identified by the objects of which one endpoint is between a^+ and a^-. To ensure Claim 1, we arrange the endpoints in memory in a particular way: the $2n$ endpoints related to each axis are stored in separate arrays and, for each interval a of which object id is $i \in [0, n-1]$, a^+ and a^- are placed at position i and $i+n$ of each array, respectively (as shown in Figure 2).

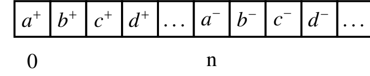


Figure 2: *Array representation of the intervals with low endpoints preceding the high endpoints.*

Moreover, the sorting algorithm used here is a variant of the stable Least Significant Digit Radix Sort (LSDRS) algorithm [Knu98], which returns the indexes of the sorted items instead of permuting the input. So, for each pair of intervals a and b where the endpoints differ, Claim 1 is straightforwardly proved by Equation 1 and the fact that all intervals have the same length, i.e., $|a| = |b|$. Indeed, under the unisize assumption, all the overlapping intervals are included in the first two cases shown in Figure 1. Otherwise, if some endpoints coincide for any couple of intervals, the stable sorting guarantees the order stated in Claim 1, i.e., the low-endpoints precede the high ones while endpoints of the same type are ordered according to their object id.

Algorithm 1 Bi-dimensional unisize SaP in 2D.

Input: $\Omega = \{0, \ldots, n-1\}$ ▷ object *ids*
Input: P_x ▷ first axis endpoints
Input: P_y ▷ second axis endpoints
Output: C ▷ colliding object pairs (init ∅)

```
 1: I_x ← Sort(P_x)                          ▷ first axis sorting
 2: for i ← 0 to 2n − 1 do
 3:     R[I_x[i]] ← i
 4: I_y ← Sort(P_y)                          ▷ second axis sorting
 5: for i ← 0 to 2n − 1 do                   ▷ second axis sweeping
 6:     p ← I_y[i]
 7:     if p < n then
 8:         S ← S ∪ {R[p], R[p+n]}
 9:         for each q ∈ S : R[p] < q < R[p+n] do
10:             C ← C ∪ (p, q)
11:     else
12:         S ← S \ {R[p−n], R[p]}
13: return C
```

In Algorithm 1, which describes the entire approach, we exploit Claim 1 by sorting one of the two axes and populating the array $R[]$ with the positions of the ordered endpoints

(loop on Line 2). Hence, for any interval a, the values $R[a^+]$ and $R[a^-]$ are the boundaries of a list of $R[]$ values (a.k.a. *ranks*) of which intervals overlap a. The endpoints on the second axis are sorted (Line 4) and then swept in the loop starting on Line 5. As in the original SaP, the active objects are stored in the set S, but here, they are represented by means of their rank. In particular, the endpoint ranks $R[p]$ and $R[p+n]$ related to the fist axis interval of an object p are added to S when the low-endpoint of p is picked from the second sorted axis, see Line 8. These values, corresponding to $R[p-n]$ and $R[p]$ on Line 12, are successively removed when the high-endpoint of p is picked. For any object a, the corresponding colliding objects are the active ones of which rank q is $R[a^+] < q < R[a^-]$ when a^+ is picked. Such a filtering operation is made by means of a *range query* performed on S using $R[a^+]$ and $R[a^-]$ as search boundaries (Line 9). As a consequence of Claim 1 and given that objects in S overlap the current object a on the second axis, it follows that the set $S \cap R[a^+ .. a^-]$, returned by the range query, immediately represents the objects colliding with a.

4. Complexity Analysis

In what follows, the time complexity is analyzed in the RAM model with word size k [CR72]. It means that primitive operations on k-bit operands are performed in constant time.

The complexity of Algorithm 1 depends on LSDRS, which runs in linear time, and the two loops starting on Lines 2 and 5, respectively. The first one costs $O(n)$ as it consists of $2n$ assignments. The second loop costs $O(n)$ multiplied by the cost of the operations performed on S: *ins*, *del* and *range query*. The implementation of S, consists in a perfect k-ary tree with n leaves, see [CL16]. This tree (a.k.a. SuccTree) has two main features: each node is represented by only one bit and the available operations (listed above) make use of bit-level parallelism to run in asymptotically optimal time. This yields a complexity of $O(\log_k n)$ for insertion and deletion, while performing a range query costs $O(\ell \cdot \log_k n)$, where ℓ denotes the number of values returned by the query.

Here, we enhance such a tree by linking the currently stored values in an ordered linked list. This does not degrade the complexity of *ins* and *del*, but the required memory grows by a factor k due to the linked list. Thus, we can perform the range query on Line 9 in $O(\ell)$ time by accessing directly the leaf node corresponding to the value $R[a^+]$ and iterating through the list up to $R[a^-]$. Thus, the overall complexity of Algorithm 1 is $O(n \cdot \log_k n + |C|)$ where $|C|$ equals the total number of values returned by the range queries performed, i.e., the number of collisions.

As a final remark, under the assumption of non-penetrating rigid bodies, we have that the number of box pairs in contact $|C| = O(n)$, which means that the complexity of Algorithm 1 becomes $O(n \cdot \log_k n)$.

5. Experimental Evaluation

We implemented our solution in C/C++ using the gcc 5.3.0 compiler, and the runs were executed single-threaded on a 2.80 GHz Intel i7-4810MQ CPU with 16 GB RAM running Ubuntu 14.04. In all runs, it was confirmed that the output array with overlapping box-pairs was correct by comparing the results of the used algorithms.

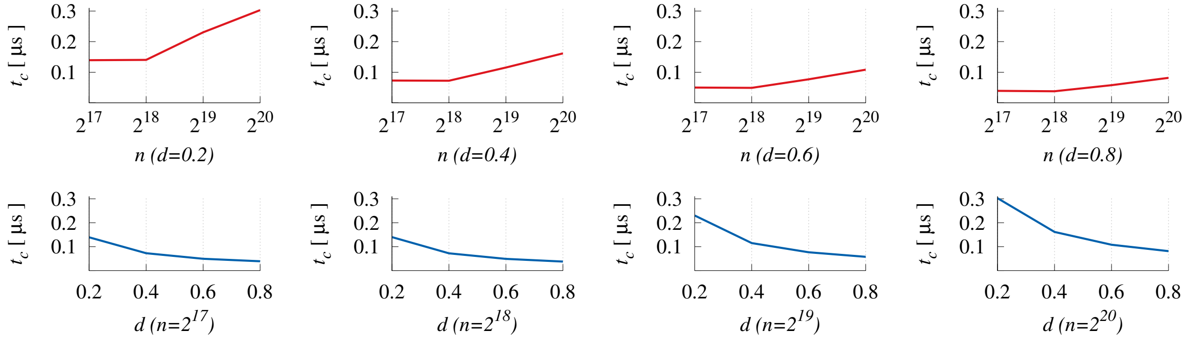
d	n	[Bar92]	[KMZ13]	Our
0.2	2^{17}	0.093	0.090	0.017
0.2	2^{18}	0.236	0.199	0.035
0.2	2^{19}	0.664	0.438	0.090
0.2	2^{20}	1.883	0.955	0.220
0.4	2^{17}	0.129	0.098	0.017
0.4	2^{18}	0.332	0.215	0.036
0.4	2^{19}	0.936	0.470	0.091
0.4	2^{20}	2.653	1.021	0.216
0.6	2^{17}	0.157	0.103	0.017
0.6	2^{18}	0.405	0.225	0.036
0.6	2^{19}	1.143	0.489	0.087
0.6	2^{20}	3.239	1.063	0.219
0.8	2^{17}	0.181	0.106	0.018
0.8	2^{18}	0.502	0.232	0.037
0.8	2^{19}	1.318	0.504	0.087
0.8	2^{20}	3.747	1.093	0.215

Table 1: *Elapsed collision detection time (in seconds).*

Our first experiment consisted of n equally-sized axis-aligned squares that moved randomly in a planar environment. The simulation was repeated for different spatial densities d, where d was the sum of the space occupied by the squares divided by the space of the environment. Initially, the squares were randomly placed in the environment under a uniform distribution. The squares bounced on the environment borders, but since no collision response was used, the squares were able to pass through each other. In general, this increases the number of overlaps that has to be reported in each frame of the simulation.

The average runtimes per frame are reported in Table 1. We compared our algorithm to the original SaP and to the algorithm for finding all intersecting pairs of iso-oriented boxes available in CGal [KMZ13]. As can be seen, our method was significantly faster in all cases. It outperformed the original SaP by more than an order of magnitude (for large datasets). Compared to the algorithm provided in the CGal library, we observed speedups around 5×. Furthermore, in contrast to the other used algorithms, our solution maintained almost the same performance regardless of the density.

To examine the relation between the empirical performance of our solution and its theoretical complexity, we now



d	0.2	0.2	0.2	0.2	0.4	0.4	0.4	0.4	0.6	0.6	0.6	0.6	0.8	0.8	0.8	0.8
n	2^{17}	2^{18}	2^{19}	2^{20}	2^{17}	2^{18}	2^{19}	2^{20}	2^{17}	2^{18}	2^{19}	2^{20}	2^{17}	2^{18}	2^{19}	2^{20}
$\|C\|$	53	105	210	420	105	210	420	840	158	315	631	1260	211	421	841	1681
t_p	7	14	48	127	7	15	48	135	7	15	48	136	8	15	48	137

Figure 3: *Time-per-collision t_c calculated in µs as the pairing phase runtime t_p over the number of collisions $|C|$ by varying the number of objects n and their density d. In the table at the bottom, t_p and $|C|$ are shown in ms and thousands, respectively.*

consider the behaviour of our algorithm in more detail. Section 4 shows that the loop starting on Line 5 of Algorithm 1 (referred as *pairing phase* in the rest of the section) dominates the complexity of the algorithm. Hence, in what follows, we focus our attention only on the performance of that phase. Since $|C| = O(n^2)$ and given that the complexity is $O(n \cdot \log_k n + |C|)$, the number of collisions $|C|$ dominates the entire formula in high-density simulations. In simpler scenarios, however, the object density is low and the number of collisions decreases and the time spent for updating S becomes more relevant than the time spent for enumerating the colliding pairs.

Graphs depicted in Figure 3 show the throughput of the pairing phase computed as the runtime over the number of collisions detected by varying d and n. Given a fixed density d, as n increases, the time-per-collision grows as well, because the number of *ins* and *del* operations performed on S increases. Clearly also $|C|$ increases, but, as shown in the table in Figure 3, it grows proportionally to n (due to the fact that objects can pass through each other and they are uniformly distributed in the world space) while the time spent for updating S grows as $O(n \cdot \log_k n)$. Given a fixed input size n, as d grows, the time-per-collision is expected to be almost constant as the number of insertion and deletion performed on S is the same for all densities. Instead, as the results show, it decreases due to the implementation of S. In fact, as the object density augments, the values stored in the SuccTree get more dense, which increases the efficiency of the *ins* and *del* operations as shown in [CL16]. Furthermore, the average time spent for adding and removing the objects to the SuccTree is better amortized due to the increased number of collision detected.

Finally, comparing the time-per-collision in all graphs, we observe that the worst time-per-collision results are related to large input sizes with low densities. This is due to the new implementation of S which, by means of the linked list, is able to drastically reduce the complexity of the range query (see Section 4), but, on the other hand, its implementation requires an array of $O(n)$ entries. When a range query is performed, such an array is accessed in an ordered manner by "jumping" to the next item from the lower boundary until the upper one is reached. Especially when the object density is low, two consecutive items get farther away from each other so that jumping to the next item implies a higher number of cache misses, which degrades the time-per-collision. To get a better analysis of this behavior, a more accurate complexity model is probably required. As a consequence, further investigations could be done by means of the cache-oblivious model [FLPR99].

To further demonstrate the performance of our algorithm, we ran one more experiment. The simulation lasted for 500 frames and 100 000 moving cubes were included. Since the motions of the objects were restricted to a plane, the simulation space was essentially two-dimensional. At the beginning, the objects were laid out uniformly in a square with velocity vectors directed towards its center. In this way the simulation gave rise to an intense clustering with lots of collisions before the objects began to spread out. Furthermore, no collision response was used; we simply counted the number of detected collisions in each discrete time step. Note that

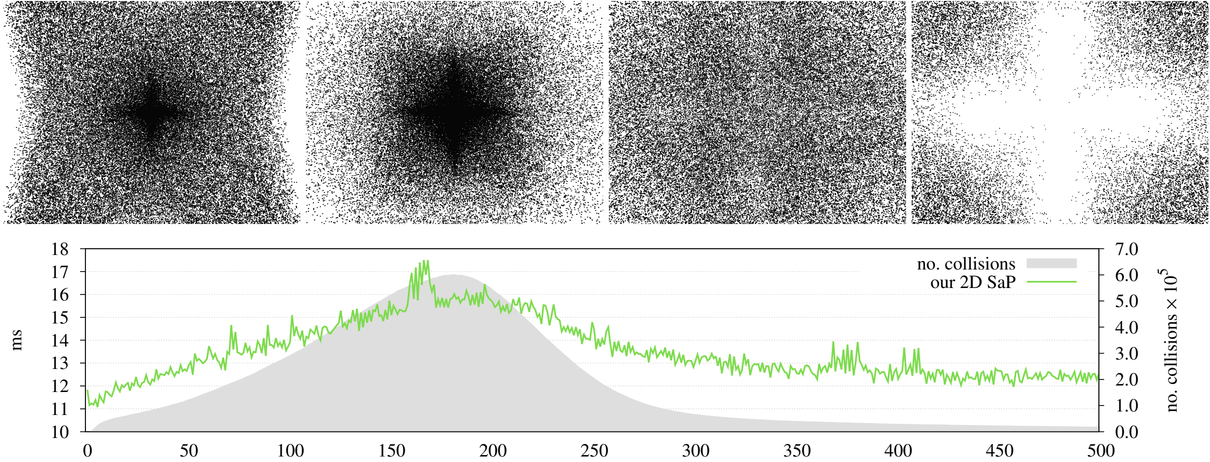


Figure 4: *Simulation of a scenario with clustering ($n = 100\,000$). The simulated objects are unisize 3D boxes, where motion is restricted to a two-dimensional surface, which means we can apply our 2D SaP method. The frames visualized are:* 100, 200, 300, *and* 400. *The plot shows collision detection times per frame (left y-axis) and the number of collisions (right y-axis).*

this choice is likely to force the algorithm to work harder, since it leads to a higher number of collisions than would be the case if the objects bounced off of each other. Four captured images of this scenario are given together with a plot of the results in Figure 4. Changes in the runtime are plotted in green, and the corresponding changes in the detected number of collisions are shown in grey. Clearly, the observed runtimes indicated real-time collision detection performance throughout the entire scenario. The worst, average, and median frame times were 17.5, 13.5, and 13.2 ms, respectively.

6. Possible Applications

We believe that there exist challenging scenarios satisfying the assumption of unisize boxes which are important in certain kinds of interactive computer graphics applications. For animated objects of roughly the same size, the size of the AABB could be set to enclose all possible poses and orientations of the objects. The size could also be extended further to enable detection of nearest neighbours, which might be useful for collision avoidance and path planning.

For instance, in large dense simulations of human crowds or animal groups, collision avoidance might become a major computational bottleneck [LM13]. As an illustration of a possible realistic scenario, consider the photograph of herding wildebeests in Figure 5. Although the movement of the herd seems quite cautious, there can be a lot of action in, e.g., wildebeest stampedes.

Besides cases that are essentially two-dimensional, there are fully three-dimensional simulations that could benefit from our unisize box assumption as well. For example, consider simulation of kinematic chains, representing objects such as ropes, cables, or protein structures, where each segment has a similar size as the others [AGN*04]. Moreover, in certain types of particle simulations, it could also be favourable to consider using a fixed box size.

Figure 5: *Photograph of a wildebeest migration [Ram10].*

7. Conclusions

The proposed modification of our more general SaP algorithm leads to an output sensitive collision detection method that is able to handle large datasets in dense environments efficiently. Straightforwardly, by adding an additional interval overlap test for box pairs overlapping in the first two dimensions, the algorithm can provide a healthy speedup also in 3D simulations of unisize boxes. The complexity analysis, however, is not transferable.

To further improve the performance of our approach, we

will turn to parallelization. Both CPU and GPU architectures offer interesting parallel features that we would like to exploit to realize a scalable parallel solution. In particular, the two sets of endpoints in Algorithm 1 can be sorted concurrently so as to reach a scalability of at most two, while the loop on Line 2 can be fairly divided among all the available CPU cores since it consists of a set of independent assignments. When we implemented such an initial parallelization, we obtained an average overall speedup of only 1.25×. Consequently, our future work will focus on finding a more complete data-parallel and scalable approach that addresses both the sorting and the pairing phase.

Another opportunity would be to evaluate the algorithm in more realistic applications, e.g., real-time simulation of fast-moving massive crowds. Moreover, it would also be interesting to further analyse the consequences of the assumption of unisize objects in the three-dimensional case as well as considering how to properly generalize our algorithm to handle continuous motion of the objects.

Acknowledgements

The authors are supported by the SSF grant no. IIS11-0060.

References

[AGN*04] AGARWAL P., GUIBAS L., NGUYEN A., RUSSEL D., ZHANG L.: Collision detection for deforming necklaces. *Computational Geometry: Theory and Applications 28*, 2-3 (2004), 137–163. 5

[Bar92] BARAFF D.: *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, 1992. 1, 2, 3

[CL16] CAPANNINI G., LARSSON T.: Efficient collision culling by a succinct bi-dimensional sweep and prune algorithm. In *Proceedings of the 32nd Spring Conference on Computer Graphics (SCCG)* (2016). 1, 2, 3, 4

[CLMP95] COHEN J. D., LIN M. C., MANOCHA D., PONAMGI M.: I-Collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (1995), pp. 189–196. 1

[CR72] COOK S. A., RECKHOW R. A.: Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing* (1972), pp. 73–80. 3

[CS06] COMING D. S., STAADT O. G.: Kinetic sweep and prune for multi-body continuous motion. *Computers & Graphics 30*, 3 (2006), 439–449. 2

[Eri04] ERICSON C.: *Real-Time Collision Detection*. Morgan Kaufmann, 2004. 1

[FLPR99] FRIGO M., LEISERSON C. E., PROKOP H., RAMACHANDRAN S.: Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science* (1999), pp. 285–297. 4

[KMZ13] KETTNER L., MEYER A., ZOMORODIAN A.: *Intersecting Sequences of dD Iso-oriented Boxes*, 4.2 ed. CGal Editorial Board, 2013. 3

[Knu98] KNUTH D. E.: *The art of computer programming, volume 3: sorting and searching (2nd Edition)*. Addison-Wesley, 1998. 2

[LHLK10] LIU F., HARADA T., LEE Y., KIM Y. J.: Real-time collision culling of a million bodies on graphics processing units. *ACM Transactions on Graphics 29*, 6 (2010), 154:1–154:8. 1, 2

[LM13] LI B., MUKUNDAN R.: A comparative analysis of spatial partitioning methods for large-scale, real-time crowd simulation. In *21st International Conference on Computer Graphics, Visualization and Computer Vision* (2013), pp. 104–111. 5

[Ram10] RAMAN T. R. S.: Photograph of wildebeest herding and following a few leading zebra in the Masai Mara Kenya, 2010. CC BY 3.0 (http://creativecommons.org/licenses/by/3.0), via Wikimedia Commons. 5

[Sam05] SAMET H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005. 1

[TBW09] TRACY D. J., BUSS S. R., WOODS B. M.: Efficient large-scale sweep and prune methods with AABB insertion and removal. In *Proceedings of the IEEE Virtual Reality Conference* (2009), pp. 191–198. 1, 2

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 24*, 1 (2005), 61–81. 1

[Wel13] WELLER R.: *New Geometric Data Structures for Collision Detection and Haptics*. Springer, 2013. 1