Computers & Graphics 57 (2016) 112–126

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on SIBGRAPI 2015

Spatial sorting: An efficient strategy for approximate nearest neighbor searching

Marcelo de Gomensoro Malheiros [a,*], Marcelo Walter [b]

[a] Center for Exact and Technological Sciences, UNIVATES, Lajeado, Brazil
[b] Institute of Informatics, UFRGS, Porto Alegre, Brazil

ARTICLE INFO

Article history:
Received 23 November 2015
Received in revised form
12 February 2016
Accepted 17 March 2016
Available online 31 March 2016

Keywords:
Spatial sorting
k-nearest neighbors
Parallel algorithms
Data structures

ABSTRACT

Many graphics and also non-graphics applications need efficient techniques to find the nearest neighbors of a given query point. There are two approaches to address this problem: space-partitioning and data-partitioning. We present a data-partitioning error-controlled strategy for solving the nearest neighbor search (NNS) problem using spatial sorting as the basic building block. We improve on the neighborhood grid method by doing an extensive study on novel spatial sorting strategies for bidimensional NNS, providing significant performance and precision gains over previous works. Experiments demonstrate that, for many dense 2D point distributions, our solution is competitive with more complex and traditional techniques, such as *k*-d trees and index sorting. We also show comparable results for the 3D case. Our primary contribution is a dynamic, simple to implement, memory efficient, and highly parallelizable solution for low-dimensional approximate nearest neighbor search.

1. Introduction

Endless graphics applications demand efficient solutions for finding the nearest neighbor, or set of neighbors, on a given set: crowd simulations [1], procedural texturing [2], remeshing and mesh simplification in geometric modeling [3,4], polygonization [5,6], volumetric reconstruction [7], fluid simulation [8], animation tasks [9], and particle-based triangular mesh generation [10]. The nearest neighbor search (NNS) is, therefore, an essential operation in many areas.

We are particularly interested in efficiently locating nearest neighbors among points in the same set. More formally, given a set S of points in a d-dimensional space and a query point P already in S, we want to find the k closest points to P in S, according to some distance metric.

Our motivation is an ongoing research on the generation of biological patterns by massive 2D cell simulations, as shown in Fig. 1. Both the NNS performance and the memory usage are our primary concerns. In each simulation step cells may move, new cells may born, and others die at random, so we also need that the underlying data structure to be updatable in parallel. In other words, as the set of points is continually changing, we must avoid rebuilding the search data structure every time.

Recently, Joselli and colleagues [11] introduced the neighborhood grid approach as an attractive alternative for low-dimensional NNS, suitable for our particular simulation problem. Such technique critically depends on the *spatial sorting* of points in space, establishing a global order. In this paper we analyze in depth such sorting operation in 2D, proposing new efficient spatial sorting strategies and experimentally measuring their performance and precision in several situations. We show that the resulting approach is *general*, behaving well for different point distributions; *memory efficient*, having very low data structure overhead; *fast*, by significantly reducing the number of comparisons needed to achieve a sorted state; and *dynamic*, adapting to a continually changing point set.

This paper is an extended version of a previous conference paper [12]. We have made comparisons to a similar approach of organizing points along a space-filling curve, added further details about the algorithms developed, discussed the final sorted states, and provided experimental measures for the adequacy of a given input point set. We also added a brief overview of comparable and consistent results when doing spatial sorting in 3D. Sample source code for algorithms and testing setups are publicly available.[1]

In Section 2 we review related work and in Section 3 we formalize the concept of spatial sorting, detailing several novel

* Corresponding author.
E-mail addresses: mgm@univates.br (M.G. Malheiros), marcelo.walter@inf.ufrgs.br (M. Walter).

[1] http://github.com/mgmalheiros/spatial-sorting

http://dx.doi.org/10.1016/j.cag.2016.03.006
0097-8493/

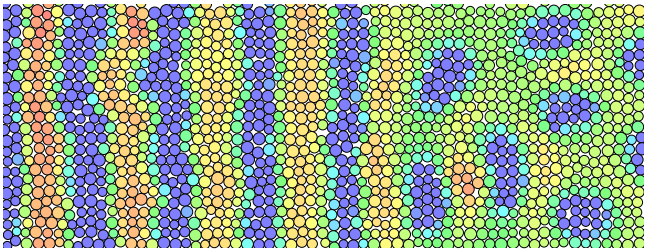

Fig. 1. Close-up detail of a biological cell simulation.

algorithms. In Section 4, we describe how to use spatial sorting as the basis for a dynamic NNS data structure. Afterward, in Section 5, we evaluate the performance and precision achieved in many scenarios. In Section 6, we summarize some best practices, providing guidance for the use of the proposed techniques. Finally, in Section 7 we present the conclusion and discuss future work.

2. Related work

Approximate nearest neighbor search (ANNS) approaches were developed as an alternative to exact Voronoi diagrams. Of particular interest is the work of Har-Peled [13], which proposes a space decomposition that approximates a Voronoi diagram and has near-linear size. Therefore, it is possible to have a trade-off between accuracy and complexity. Rong and Tan [14] present the Jump Flooding algorithm, which employs the GPU to construct an approximate Voronoi diagram in parallel. That said, we have observed that most literature employs exact NNS methods, and thus we will compare our results to such techniques.

Li and Mukundan [1] presented a review on spatial partitioning methods with the focus on 2D crowd simulations, evaluating four data structures: grid, quadtree, *k*-d tree, and Bounding Interval Hierarchy (BIH). They run their implementations of these structures on crowd simulation scenarios with a growing number of agents, up to 10,000. In their experiments, the grid performed better than the other three techniques. The paper does not mention details about the hardware used, and it is then hard to compare their results.

Considering that crowd simulations are heavy users of neighboring information, researchers have been investigating alternative partitioning methods. Vigueras and colleagues [15], for instance, explored irregularly shaped regions as a partitioning strategy in a distributed architecture. They compared the performance of such regions against R-trees [16] and against a solution using heuristics to define the rectangular partitioning. Their results showed that the convex hull regions provided better performance than the other two methods.

Recent *k*-d tree implementations make advances in GPU acceleration, overcoming limitations due to conditional computations and suboptimal memory accesses. Gieseke et al. [17] describe a buffered approach, organizing queries by spatial locality, which are then run in the same GPU core. Kofler et al. [18] use a specialized *k*-d tree to compute *n*-body simulations, built inside the GPU memory in distinct phases, by first creating nodes for large groups of particles, which are then refined.

Fluid simulations using Smoothed Particle Hydrodynamics (SPH) methods are also dependent on *k*-NN, being typically performed on uniform grids. Green [19] presents a parallel 1D sorting technique to group particles according to index similarity so that nearby particles are indexed closely, which is the base for further improvements presented by Ihmsen et al. [20], analyzing both spatial hashing and index sort.

In Kim et al. [21], the grouping of subgrid structures enables the division of work between several CPUs and GPUs, enabling the simulation of millions of particles, while overcoming a limited GPU memory space of a single GPU. Another interesting approach, somewhat similar to what is described in this paper, Connor and Kumar [22] sort the points along a unidimensional sequence, following a Z-order, and place them in a matrix. After that the *k* neighbors of a point are found by performing another local sort of $O(k \log k)$ complexity.

Gast et al. [23] note that while NNS algorithms are efficient for low dimensions, like the present case, for a large number of dimensions even specialized algorithms can give only a minor performance gain over sequential search.

Although quite efficient, these solutions still have shortcomings from our point of view, such as high memory overhead caused by the auxiliary data structures. Furthermore, as the set of cells from our biological simulation is continually changing, we must update the data structure dynamically. Thus, it is undesirable to reconstruct it at each time step. As a final demand, we also sought for an approach that is simple to parallelize on current GPUs.

The neighborhood grid was proposed by Joselli et al. [11], following early work in 2D [24] and 3D [25]. The main idea is to organize points into a matrix or tridimensional array, to accelerate the location of nearest neighbors. However, it should be noted that the approach previously described does not try to achieve a fully sorted state. Instead, only partial sorting is performed at each simulation step, mostly because of performance concerns, leading to low precision when locating nearest neighbors. In this paper, we explore ways to efficiently keep the point set always fully sorted, which significantly improves the accuracy and the usefulness of this technique.

3. Spatial sorting

Although sorting is a classic topic in Computer Science, it is almost always devoted to a single sequence of elements, that is, a unidimensional list of comparable items. For clarity, we will call this *1D sorting*. For example, we can locate the $k=2$ nearest neighbors for each real number of an array by performing two operations. First, we just sort the numbers in ascending order. Then, we can locate the nearest neighbors for a given array element i by just examining the elements with indices $i-2$, $i-1$, $i+1$, and $i+2$, as shown in Fig. 2. Naturally, we need to adjust the search when dealing with elements near the array ends.

The term spatial sorting is sometimes used to name the process of ordering *d*-dimensional points along a space-filling curve so that the result is still a unidimensional sequence. We may thus call it *1D sorting along a curve*. The more common schemes are using either the Hilbert curve or the Morton order curve (also known as Z-order) to establish a space traversal, where points are placed into discrete bins, which are then ordered when the curve is followed. Fig. 3 depicts both the Hilbert and Morton orderings for the same set of points.

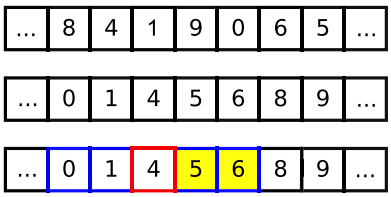

Fig. 2. Finding nearest neighbors: unsorted sequence (top), sorted array (middle), and searched elements (bottom). The query point is outlined in red, the candidates are in blue, and the two nearest neighbors are marked in yellow. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

Therefore, the same approach as in the previous unidimensional case can be performed: sort the points along a chosen curve, which gives a sequence of points that can be placed into an array, and then for each array element evaluate a few previous and following elements. Even though some spatial locality is achieved for nearby points along such ordered sequences, there can be many discontinuities between bins as inevitably the ordering curve will need to make turns to cover the bidimensional space. Such discontinuities cause some near points to get far apart on the final order, thus preventing the correct location of their nearest neighbors.

A better-suited approach of organizing points can be achieved by doing a bidimensional ordering on a given set of points, not restricted to a linear sequence, but arranged in a regular rectangular grid. If we employ a matrix as the underlying data structure, we can achieve better locality after the spatial sorting is performed. This organization can also enhance GPU memory access because a 2D texture read operation may cache nearby queries coherently in two dimensions.

Surprisingly, there are very few references in the literature that cover the problem of organizing a matrix of points by separate comparison of the x and y coordinates. It should be noted that traditional lexicographical ordering still results in a unidimensional sequence, whereas we strive to find a spatial ordering. Parallel sorting algorithms using grids of processors perform sorting along just one dimension [26,27]. A similar problem, but that still deals with single values (and not pairs of values) is the lexicographical ordering of matrices [28,29].

We call *spatial sorting* the establishment of an ordering of 2D points into a matrix, although the same strategy can be generalized to higher dimensions. An example of spatial sorting is shown in Fig. 4, where x and y integer coordinates are shown on the bottom left and top right of each matrix element, respectively. For easy visualization, the figures show color-coded points, where the normalized x and y coordinates are mapped to the red and green color channels, respectively. Also, to match the Cartesian plane, we opted to start the row numbering on the bottom of the matrix. Fig. 5 illustrates a search, where nearby elements around the query point are tested to locate the two nearest neighbors.

We say that the matrix is *spatially sorted* when the points in each row are sorted by their x coordinates, where at the same time the points in each column are sorted by their y coordinates. More formally, we establish two comparison criteria used for sorting, one for elements in a matrix row (Eq. (1)) and another for elements in a matrix column (Eq. (2)):

$$p1 \cdot x > p2 \cdot x \tag{1}$$

$$p1 \cdot y > p2 \cdot y \tag{2}$$

Previous works [24,25,11] employed a lexicographic order as comparison criteria for each coordinate, where a second coordinate was used for comparison when the first one was found to be equal. That is, when comparing elements in a given row, Eq. (3) would be used, whereas Eq. (4) would be used for elements in the

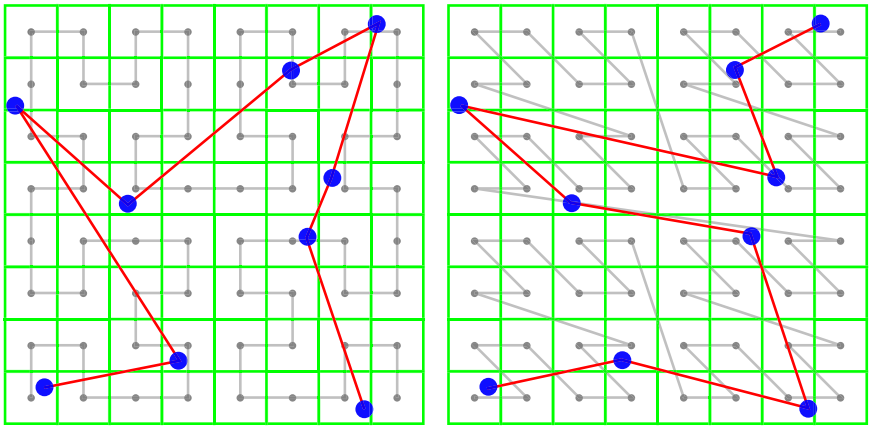

Fig. 3. Sorting 2D points (shown in blue) along a space-filling curve (drawn in gray): Hilbert curve (left) and Morton order curve (right). The resulting 1D sequence is shown in red. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

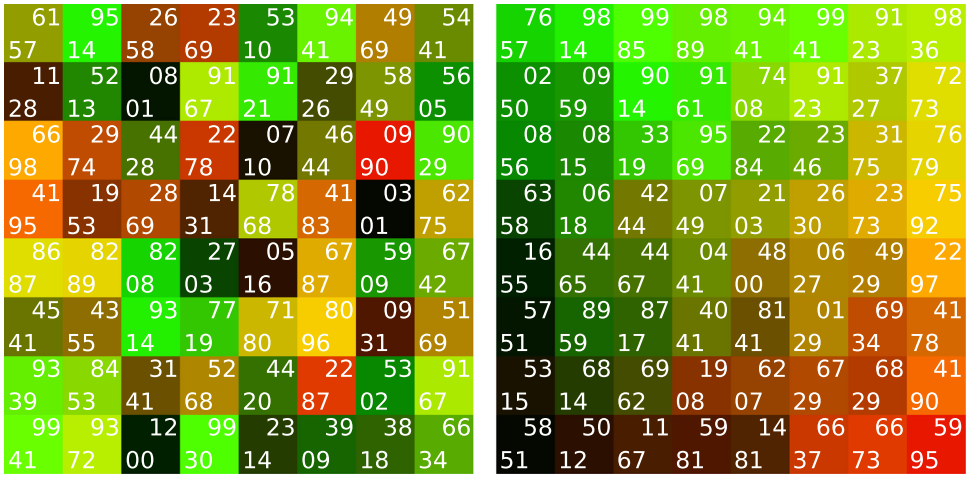

Fig. 4. Spatial sorting a matrix of 8×8 uniformly distributed 2D points: before (left) and after (right). The x coordinates are color-coded in the red component and show at the bottom-left corner of each matrix element. The y coordinates are color-coded in the green component and shown at the top-right corner. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

same column.

$$(p1 \cdot x > p2 \cdot x) \vee ((p1 \cdot x = p2 \cdot x) \wedge (p1 \cdot y > p2 \cdot y)) \quad (3)$$

$$(p1 \cdot y > p2 \cdot y) \vee ((p1 \cdot y = p2 \cdot y) \wedge (p1 \cdot x > p2 \cdot x)) \quad (4)$$

We have evaluated that this extra complexity added limited value. First, on most random point distributions tested, the amount of repeated x or y coordinates was small (being about 2% on *float* coordinates for one million points). Therefore, it is only useful to distributions with many non-distinct coordinates. Second, such complexity added a small overall increase in processing time when sorting on a CPU, but a significant penalty when running on a GPU. This increase can be explained by the parallel thread (warp) divergence introduced by the extra conditionals. Third and last, for most point distributions the criteria (3) and (4) did not improve the nearest neighbor search precision. Thus, we opted to use Eqs. (1) and (2) as the default sorting criteria throughout this work, unless noted otherwise.

In Section 3.1 we show that achieving a sorted state is always possible. However, it is not readily obvious that, from a given set of points, there are several possible sorted results. Thus, the problem is not to find *the* spatially sorted matrix, but one of the possible sorted states, by applying different sorting strategies, as discussed in Section 3.2. A more in-depth discussion for those final states is made in Section 3.3.

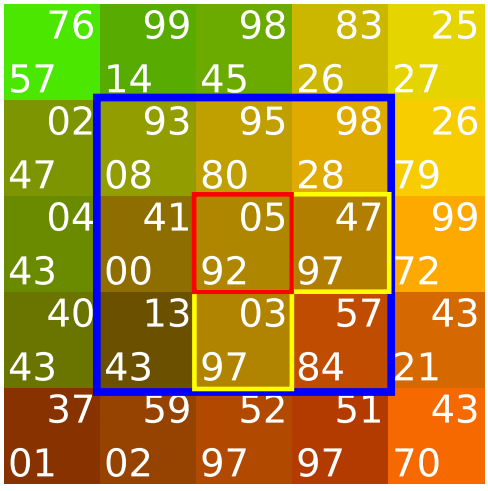

Fig. 5. Detail of previously sorted matrix: a query point is shown in red, the eight possible candidates are in blue, and the two actual nearest neighbors are marked yellow. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

3.1. Lower bound

We have devised a simple deterministic algorithm to achieve a spatially sorted state, given a random set of n 2D points.

For convenience, suppose $n = s^2$, with integer s. First, place all points into an array of length n. Then, perform a standard 1D sort on this array, comparing only the y coordinate, thus creating an increasing sequence (regarding only y). After that, the array should be viewed as a row-major matrix, where the first s elements are in row 0. We now have that all points in row 0 must have y coordinates less than or equal to the y coordinates of all points in row 1, and so on. Therefore, if now we execute a 1D sort on each row, sorting by x coordinate, we will achieve the final, spatial ordering. Provided we employ an optimal 1D sort, such as merge sort, we have that the first step is $O(n \log n)$. The second step takes $O(s^2 \log s)$, as we need to sort s rows, each one with length s. Combining the two steps and replacing s by $\sqrt{n}$, we find an overall time complexity of $O(n \log n)$. This strategy is called SIMPLE and is formalized in Algorithm 1.

Algorithm 1. Fast spatial sorting (SIMPLE).

```
run a 1D sort for all points, comparing y
for each row r do
    run a 1D sort on r, comparing x
```

Although SIMPLE reaches a sorted state, such arrangement provides consistently low precision when locating neighbors, because points with similar coordinates still can get far away. This can be seen by visual inspection after each algorithm phase, as shown in Fig. 6. The first y sorting pass makes the green component get ordered, from bottom to top. The second pass only orders points by sorting the x coordinates in each row, which also makes the red component range from darker (on left) to brighter (on right). However, we can see visual discontinuities in the overall gradient, and this is directly reflected in the precision of the later neighbor-gathering phase, as will be evaluated in Section 5.3.

We have thus established the asymptotic lower bound for the general problem of spatially sorting a set of points, given the criteria defined earlier. That is, any spatial sort is bounded below by $\Omega(n \log n)$.

This is reinforced by the equivalence between performing a bidimensional spatial sort and sorting a set of sequences of real numbers. Suppose we have the problem of sorting s unrelated lists of s real values; it has $\Omega(n \log n)$ as its established lower bound, again with $n = s^2$. We can thus build an associated spatial sorting problem by copying each unsorted list to a matrix row, setting the x coordinates. We can also define that for the i-th row, all y

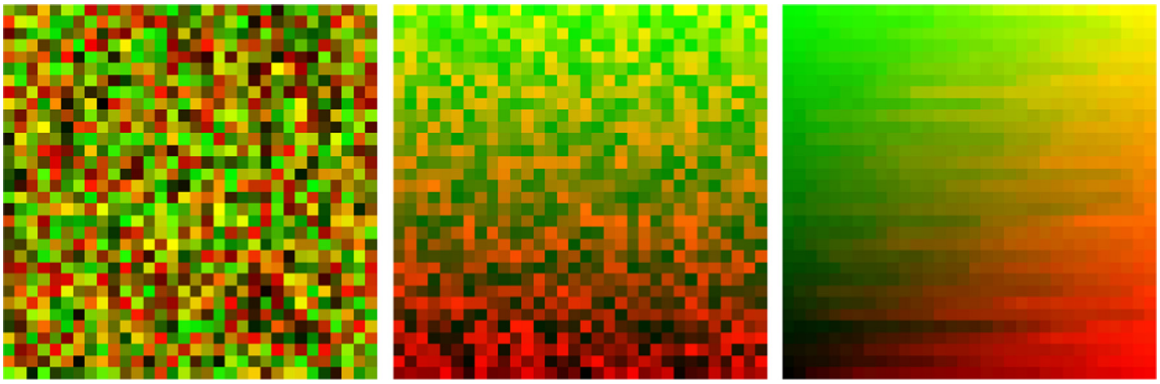

Fig. 6. Running the SIMPLE algorithm on a matrix of 32×32 uniformly distributed points: initial state (left), after the y-pass (center), and after the x-pass (right).

coordinates will be set to i. If the matrix is then spatially sorted using the SIMPLE algorithm, those lists will be ordered.

3.2. Sorting strategies

A general approach to reach a spatially sorted state can be built on repeated unidimensional sorts on the matrix, for several steps, as shown in Algorithm 2. For each step, we have two passes: a traditional 1D sort is done on each row, and then another pass, where a 1D sort is performed on each column of the matrix. After each step, the matrix will get more organized until a spatially sorted state is reached.

Algorithm 2. General approach to spatial sorting.

```
sorted ← false
while ¬sorted do
  for each row r do
    run a 1D sort on r, comparing x
  for each column c do
    run a 1D sort on c, comparing y
  if matrix is sorted then
    sorted ← true
```

Different from SIMPLE, this approach is unbiased to either the x or y coordinates. On the other side, it will take several steps to converge to a sorted state.

Passos et al. [24] proposed a variation of the general approach, based on the odd–even sort algorithm. Instead of running the standard 1D odd–even sort for each column and row, just one partial step was run. That is, for each row, a first pass would be done comparing (and swapping, if needed) adjacent pairs of values starting at odd matrix indices, and then a second pass comparing adjacent pairs starting at even indices. Then, two passes would be run for the columns. Therefore, a single step of the spatial odd–even sort would have four passes.

Such odd–even step is then repeated until a fully sorted configuration is achieved. The spatial odd–even sort (here named SOE) is simple to code and easily parallelizable, as the comparisons and exchanges within one pass can be all performed in parallel. However, this algorithm is also very slow and memory-intensive, taking many steps to converge. In fact, the 1D odd–even sort algorithm has quadratic time complexity, so this has effects on its spatial variation.

We can establish the time complexity for SOE as follows: let $n = s^2$ and note that one step will perform four passes on the matrix values. Each pass will need at most $s/2$ comparisons in each row or column, where a complete pass will perform $O(s^2)$ comparisons. A single step is still $O(s^2)$, and the number of steps will depend on the *sortedness* state of the initial matrix. For the worst case, imagine that the point P with largest x and y values is at the bottom-left element. While sorting, the points with larger coordinates move to the top and to the right, so the final P will be in the top-right corner. As a point moves at most one row or column at each step, it will take s steps for P to reach its final destination. Thus, the worst-case time complexity for SOE is $O(s^3)$ or $O(n^{1.5})$.

3.2.1. Spatial Shell sort

The unidimensional odd–even sort is *adaptive*, that is, each step takes advantage of the previous partial ordering of data. Therefore, several runs of the spatial odd–even step will gradually sort the matrix. Such order is not achieved locally, as the recursion step of a 1D merge sort, for example, but globally. In fact, we have observed that just using efficient but non-adaptive 1D sorting algorithms (like quicksort or bitonic sort) for repeatedly ordering the rows and columns alternately still takes many steps to converge. For example, if we implement a spatial quicksort algorithm SQ, which follows the general approach, by using repeated quicksort on the rows and columns, the resulting sorting performance is consistently three times slower than the best algorithms found so far. For comparison purposes, such algorithm is evaluated against other approaches in Section 5.1.

A better approach has to be adaptive, keeping the partial ordering from the last step and improving upon it as new steps are run. Also, we found that getting into a partially sorted state fast and then running another spatial sorting algorithm would converge much faster. Therefore, we employed a combination of 1D algorithms that are known to be adaptive: insertion sort and Shell sort.

Standard 1D insertion sort is known to have worst-case quadratic time complexity, but it is optimal when the input is already sorted, taking only $q-1$ comparisons in an array of size q. This fact is especially relevant as the matrix gets progressively organized because some rows and columns will get fully sorted before others, thus not changing on further steps.

Unidimensional Shell sort has time complexity that depends on the particular gap size employed. As it is based on 1D insertion sort, it is also adaptive. But Shell sort has another desirable property: it has several passes, one for each gap size. When it runs with larger gaps, it is possible to jump elements over several positions within the 1D sequence being ordered, which makes the data get more quickly into the sorted state. In fact, the last pass of Shell sort is always run with gap size 1, which means it is exactly the same as standard insertion sort. As the array is mostly ordered, this final pass is typically very efficient and ensures that the final sequence is fully sorted.

Because 1D Shell sort has several passes, we derived a spatial version, interleaving for each gap size g a parallel pass on each row, followed by another parallel pass on each column. A standard unidimensional Shell sort pass of gap g is performed in each row or column. We observed best results when using just one step of the spatial Shell sort to create a "rough" ordering, which is then refined through iterative alternating 1D insertion sort steps, as explained in Algorithm 3.

Algorithm 3. Spatial Shell + Insertion sort (SSI).

```
for each gap in sequence G do
  for each row r do
    run a gapped 1D insertion sort on r, comparing by (3)
  for each column c do
    run a gapped 1D insertion sort on c, comparing by (4)
sorted ← false
while ¬sorted do
  for each row r do
    run a 1D insertion sort on r, comparing x
  for each column c do
    run a 1D insertion sort on c, comparing y
  if matrix is sorted then
    sorted ← true
```

This algorithm, dubbed SSI, arrived at a spatially sorted state faster than any other tested variation of the general approach (Algorithm 2). That is, it performed better than repeatedly running a standard 1D sorting algorithm on each row and then on each column. Interestingly, the comparison criteria given by Eqs. (3) and (4) gave an average performance gain of 16% for SSI against simple x and y comparisons, so for this particular algorithm, the more complex comparison was used.

3.2.2. Spatial quicksort

In this work we also discuss the second-best spatial sorting algorithm found, a combination of quicksort and insertion sort, called SQI. Similar to the previous algorithm, we achieve a fast and rough initial matrix ordering, now using 1D quicksort, which is performed first on each row and then on each column. Then the sorting is continued with repeated 1D insertion sort steps until a sorted state is found. As expected, insertion sort performs much better than quicksort for those convergence steps, due to adaptive nature of the former. The outline of SQI is described by Algorithm 4.

Algorithm 4. Spatial quicksort + insertion sort (SQI).

```
for each row r do
  run a 1D quicksort on r, comparing x
for each column c do
  run a 1D quicksort on c, comparing y
sorted ← false
while ¬sorted do
  for each row r do
    run an insertion sort on r, comparing x
for each column c do
    run an insertion sort on c, comparing y
if matrix is sorted then
sorted ← true
```

Although SQI is marginally worse than SSI regarding spatial sorting performance, we opted to keep it in our experiments due to its simplicity of implementation. In fact, quicksort implementations are readily available for all architectures and programming languages. Moreover, the serial quicksort step can be directly replaced by parallel algorithms like bitonic sort or radix sort, which in fact provide an efficient parallel spatial sorting. The gapped Shell sort used by SSI, on the other hand, has a complex memory access pattern, which adversely impacts its performance on a GPU.

3.2.3. Optimizing spatial insertion sort

We have also made two simple performance improvements to the iterative 1D insertion sort steps, employed by both SSI and SQI. First, we simply count the number of write operations done during a single pass on all rows or all columns. If the count is zero, the matrix is already sorted, and the iteration should stop.

Then, we added a "dirty" boolean flag to each row and column, to mark it as already sorted. If cleared, the 1D insertion sort for that particular row or column is skipped. Such flags are set when the write operations are performed inside a serial insertion sort. For example, when running the serial insertion sort for row r, for a given write operation on the matrix element $[x, r]$ we mark the column x as dirty. After this serial insertion sort finished, we also mark the row r as clear. And the process then repeats for the columns, likewise.

Although the use of dirty flags consumes an extra amount of space and takes some additional read/write memory operations, it significantly reduces the amount of work done by the convergence steps. Due to memory caching on CPUs, we got an average performance improvement of 4% for both SSI and SQI; however for GPUs, by preventing the launch of unneeded threads, we got an average reduction of 20% on processing time for our bitonic-based spatial sort (discussed in Section 5.6).

3.3. Final sorted state

Based on the general approach for spatial sorting, given by Algorithm 2, many variations were evaluated. For this work, we opted to discuss only SIMPLE, SOE, SSI, and SQI as practical algorithms. The SIMPLE algorithm is important as it represents the fastest way to achieving a sorted state, whereas SOE was discussed in previous works [24,25,11]. Finally, SSI and SQI represent the best algorithms known so far, both in running time and precision obtained.

It should be noted that most algorithms directly derived from the general approach, like SQI, should produce the same sorted state, basically depending on the stability of the 1D sorting routines employed.

Sorting stability was not a major concern for us because of performance reasons. Unidimensional Shell sort is not stable, and the same applies to 1D quicksort. Furthermore, we would also need to differentiate between points with the same x or y coordinates, by using the comparisons defined in Eqs. (3) and (4).

More important, however, is the "quality" of the sorted state reached. We have observed that distinct spatial sorting algorithms consistently lead to different sorted states, that are more or less precise when running the nearest neighbor search phase. In other words, some algorithms result in more uniform final states of the matrix, which then gives a lower miss rate when looking for the correct nearest neighbors. The precision is numerically evaluated in Section 5.3 while we will briefly discuss here why this happens.

We can visually compare the color-coded sorted states reached by different sorting algorithms on a uniform point distribution, as shown in Fig. 7. We see that the SIMPLE algorithm, although being fast and deterministic, fails to uniformly spread the matrix elements, giving rise to visual discontinuities on the colors. A far better result is reached by SOE, but still some color banding is seen in the matrix. Both SSI and SQI reach an even better final result, very similar visually, so only SSI is shown. As we said before, this visual uniformity directly maps to the actual precision in finding nearest neighbors.

Fig. 7. Final sorted states for the same uniform point distribution and initial 100×100 matrix, using distinct algorithms: SIMPLE (left), SOE (center), and SSI (right).

The larger is the set of points, more sorted states are deemed to exist. Therefore, depending on the initial point distribution on the matrix and the actual algorithm used, we have different ways to reach a sorted state. It seems that SIMPLE performs badly because only a minimum number of comparisons are done, which prevents points from moving to more adequate positions in the matrix. The SOE algorithm provides a more uniform arrangement, but because each matrix element is only being compared to nearby elements, the migration of points is limited, still giving rise to visual discontinuities. On the other hand, both SSI and SQI, by the 1D sorting strategies, enable elements to be compared and jump to arbitrary locations in the matrix and therefore are more likely to reaching a better sorted state.

We have also experimented with an alternative method to reach the final state: randomly comparing and exchanging matrix elements, until the matrix gets spatially sorted. This led to another algorithm, dubbed RANDOM and which is discussed next.

While doing random compare-and-swap operations is obviously inefficient, it has the advantage of being completely unbiased: it does not imply any particular comparison order, so any pair of elements of the matrix have the same chance of being compared and exchanged. And, as expected, such approach indeed reaches final states that consistently give a small increase in precision in relation to SSI and SQI.

A naive approach would simply repeatedly pick two random elements from the matrix and then compare their x and y coordinates. But as there is no global ordering when doing a spatial sort, such direct comparison does not make sense, as x coordinates must be compared only within the same row, and y coordinates only the same column. Instead, we could pick at random either a row or a column, and then select two arbitrary elements in it. Thus, if a row is picked, we use Eq. (1) for a compare-and-swap operation, whereas for a column we would use Eq. (2). This procedure indeed works, but takes a long time to converge, because a single comparison made in a given row, for example, will probably disrupt some partial organization already achieved for the columns involved.

The RANDOM sorting strategy is presented in Algorithm 5, running in average four times faster than the naive approach, and still reaching the best sorted results so far, in terms of NNS precision. Albeit impractical for real applications, we think this algorithm has theoretical importance, and thus is also evaluated in Section 5.3.

Algorithm 5. Random spatial sort (RANDOM).

```
sorted ← false
while ¬sorted do
  for a large number of times do
    randomly choose i < j as column indices
    randomly choose k < l as row indices
    let A be matrix[i, k]
    let B be matrix[j, k]
    let C be matrix[i, l]
    let D be matrix[j, l]
    compare-and-swap A and B, using x
    compare-and-swap C and D, using x
    compare-and-swap A and C, using y
    compare-and-swap B and D, using y
  if matrix is sorted then
    sorted ← true
```

The insight here is to not use a pair of elements, as usual in a compare-and-swap operation for unidimensional sorts, but a set of four elements. Fig. 8 illustrates this idea. The elements are chosen as the vertices of a random rectangle inside the matrix, and compared against themselves in two passes: the first pass compares the AB and CD pairs, each in a different row, whereas the second pass compares the AC and BD pairs, now laid in different columns. This way, we use a compare-and-swap operation that tries to achieve a more consistent ordering, now ordering unambiguously four elements.

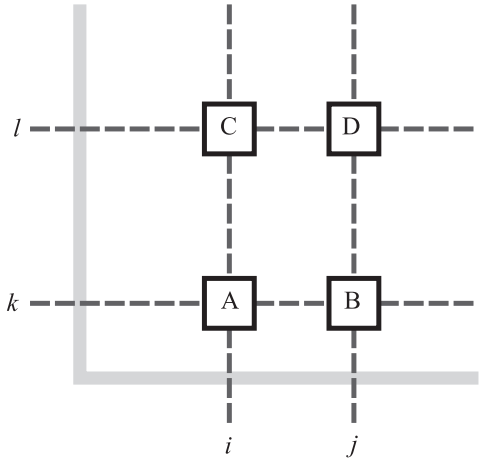

Fig. 8. Spatial compare-and-swap operation.

Visually, the result is still quite similar to what was reached by SSI and SQI. However, the search precision is slightly but consistently better. This hints at the possibility that this new spatial compare-and-swap operation can be indeed a primitive operation for the construction of other spatial algorithms. In fact, we think it could be the basis for the design of spatial sorting networks.

4. General data structure

This section provides a brief discussion of the issues that have to be handled when using spatial sorting as part of a general data structure. Such operations are needed to support the ongoing research into a massive biological cell simulation, as we strive to keep the memory overhead at a minimum and also handle the dynamic creation of cells by division. Previous works only cover the situation where the matrix is filled with a constant number of points.

However, it is necessary to account for situations when we need to dynamically add new points to the matrix, or remove existing ones. We also need a technique to find the nearest neighbors for query points that are not in the matrix.

4.1. Choice of matrix dimensions

The typical matrix arrangement is a square, but its aspect should match the point distribution. We thus need the matrix dimensions to reflect an adequate rectangle placed on the plane. The general approach is to find the ratio between the number of columns and rows that better approximates the ratio of x and y intervals in the domain. Otherwise, the precision when searching for neighbors would be negatively affected.

Suppose, for example, that we have points uniformly distributed inside an ellipse, where the minor axis coincides with the y-axis and the major axis coincides with the x-axis. Therefore, we would have a larger interval of values along the x coordinate, so to get a better sorting of values inside the matrix, we would need more columns than rows, as shown in Fig. 9.

For heterogeneous distributions, the union of the more dense regions should define the axes-aligned rectangle where most points are placed, which will give the ratio for the neighborhood matrix.

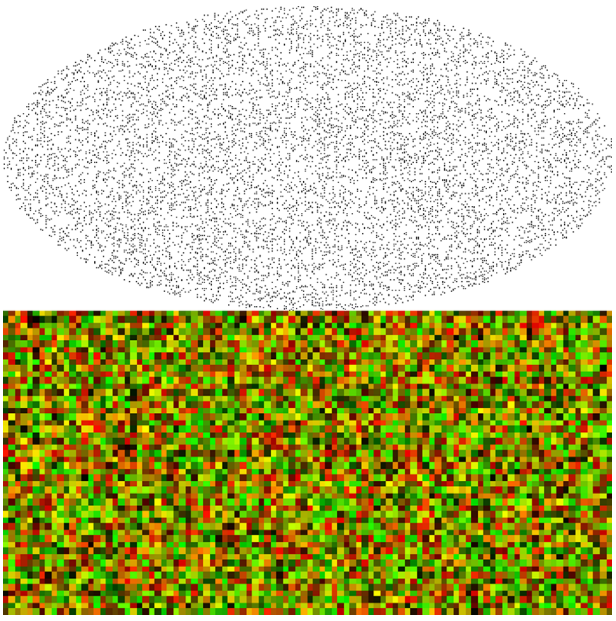

Fig. 9. An example of elliptic point distribution (top) and matching unsorted 50×100 matrix (bottom).

It should be clear by now that all points must lie in a domain which is topologically equivalent to a square. The case of points on the entire surface of a sphere, for example, cannot be handled by the current approach. In this case, even if the point coordinates are stored as a pair of angles, the proximity of points on the poles will not be correctly identified.

4.2. Matrix growth

On most situations, it is not possible to exactly fit all points into a rectangular matrix. Instead, at least one extra row or column will be needed, with some elements left empty. Empty elements are easy to handle, as they need to store the maximum floating point value for both the x and y coordinates (for C++, it would mean the `FLT_MAX` constant). They are then compared with other points as usual by the spatial sorting algorithms, which will make them accumulate on both the topmost row and the rightmost column. When testing candidates, such empty positions should be ignored.

We may allocate all matrix elements sequentially in memory, either in row-major or column-major order. A size extension would increase the row or column count, filling the newly "uncovered" positions (at the end of allocated memory) with empty elements, as illustrated in Fig. 10. After that, a full spatial sort is needed. Otherwise, all points would get shifted along the matrix by the size change. This scheme provides the minimum memory usage.

4.3. Inserting and removing points

After we created some empty elements inside the matrix, there are several ways in which a new point can be inserted. The simpler approach is just to select one of the available empty elements and set its x and y coordinates to the new point to be added. A good choice would be the leftmost empty element or the bottommost one, as the new point will already be with lower coordinate values than the empty one, which will "slide" to the correct matrix element during the next sort.

Point removal is straightforward. We should mark as empty the associated position, and after a sorting update is run, such element will move to the top or the right of the matrix.

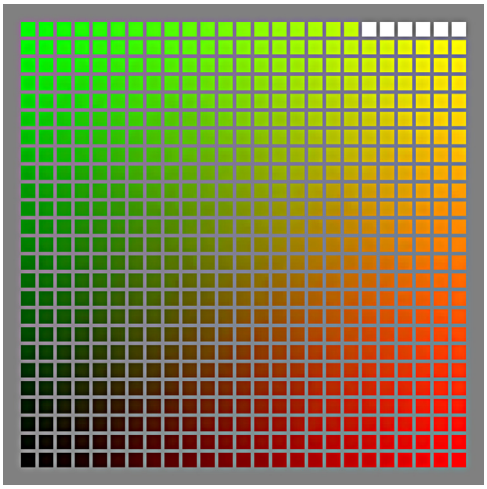

Fig. 10. Matrix with empty elements, shown in white.

4.4. Point location

Point location happens when the query point is not already associated with an element of the matrix, which is not the recommended usage for our approach. Still, such operation may be necessary, and therefore, we need to employ a search scheme that can locate the rough placement of the query point, and then perform a local search using a Moore neighborhood.

As we only have an independent ordering of rows and columns, we cannot use a spatial bisecting algorithm like a k-d tree. We also cannot walk on the matrix towards a minimum distance regarding the query point (q_x, q_y), as the relative sorting creates several local minima. Thus, we need to do a more extensive search to locate the center for a "good" candidate neighborhood. We have experimented with a procedure composed of three steps. First, for each row of the matrix, we perform a binary search on the x coordinate, to locate the element closer to q_x. From the elements selected in each row, we pick the point C that has the closest y coordinate to q_y. As this is not enough to accurately pinpoint the nearest neighbor for all situations, we finalize by analyzing candidates inside a Moore neighborhood centered at C. Such algorithm has time complexity $O(\sqrt{n} \log n)$. This follows from the fact that binary search is $O(\log n)$, which is repeated $\sqrt{n}$ times. After that, a sequential search over $\sqrt{n}$ elements is run, which is finalized by the evaluation over a fixed neighborhood of m candidates.

5. Evaluation

In this section, we evaluate the spatial sorting algorithms described earlier using both CPU and GPU implementations. We first measure the performance and precision of several 2D sorting strategies against different point distributions. We also discuss how to evaluate whether a given point distribution is amenable to the usage of spatial sorting. Then we address the particular case of dynamic updating a previously sorted state. Finally, we compare the best algorithm (SSI) to a state-of-the-art k-d tree and index sort implementations.

In order to standardize the experimental tests, we employed one million randomly generated points inside the $[0, 1] \times [0, 1]$ domain, which were then sorted by using different algorithms into a 1000×1000 matrix. For each sort, the same set of point distributions was used, generated from the same random seed values. The point distributions were generated using the Mersenne Twister implementation of the GNU Scientific Library [30].

We counted the number of actual comparisons made, together with the running time of the sorting and querying process in each case. Then, both the number of comparisons and the running time were averaged, with the standard deviation shown as a red error bar in the figures (with amplitude equals to 2σ).

The test programs were implemented in C++ on a Ubuntu 14.04 Linux system. The test machine was powered by an Intel Core i7-4500U CPU running at 1.80 GHz. Later on, parallel tests were done with the same configuration, but using an NVidia GeForce GT 750 M GPU, with 384 CUDA cores, 2 Gb of GDDR5 RAM and with CUDA runtime 7.5.

5.1. Spatial sorting performance

We have evaluated both the number of comparisons and the running time for five distinct algorithms: SIMPLE, SOE, SQ, SQI, and SSI. We note that SQ is the straightforward implementation of the general approach for spatial sorting, simply using repeated 1D quicksort on rows and columns. We have kept SQ to demonstrate that non-adaptive sorting algorithms lead to bad spatial sorting performance, despite their efficiency in 1D sorting. For the particular results shown in Figs. 11 and 12, we generated 100 different uniform point distributions.

For SSI, we have tested with the gap sequence suggested by [31], where $G = \{1750, 701, 301, 132, 57, 23, 10, 4, 1\}$. The particular complexity lower bound for this sequence has not yet been established in the literature, but we did run extensive experimental tests that confirm the efficiency of the SSI algorithm in the average case, against other strategies.

To estimate the complexity of the SOE, SQI, and SSI algorithms, we evaluated the mean number of comparisons for ten different random seeds, each generating a single uniform point distribution. This process was repeated for matrices with sizes ranging from 200×200 to 1200×1200, as shown in Fig. 13. The resulting series were then non-linearly fitted. We found as best fit a $n^{1.497}$ curve for SOE, which matches the previously estimated worst-case complexity of $O(n^{1.5})$. We also found a $n^{1.126}$ curve for SQI, and a $n^{1.088}$ curve for SSI, with n being the total number of points. Although empirical, these last two curves hint at an average comparison complexity close to $O(n \log n)$. Similar curves were also found for their running time.

5.2. Point distributions

Following [23], we can establish that there are two main types of NNS strategies: space-partitioning and data-partitioning. Space-partitioning methods divide the space according to boundaries in space, which gives rise to data structures like the k-d trees. On the other hand, data-partitioning methods divide data according to their distribution, where the typical indexing structures are derivatives of the R-tree. We can thus say that spatial sorting is a data-partitioning scheme. Therefore, the actual distribution of points is key in the assessment of precision.

We have performed experimental tests in many synthetic 2D point distributions, to evaluate how well the spatial sorting and neighbor-gathering algorithms performed in typical and not-so-common cases. Here we present 13 cases, which are individually generated from a single random seed.

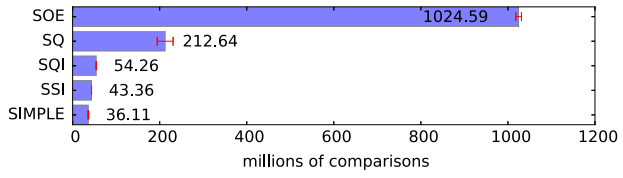

Fig. 11. Average number of comparisons for one million uniformly distributed points.

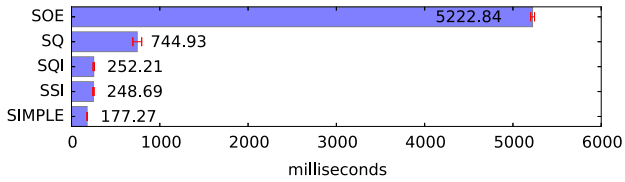

Fig. 12. Average running time for one million uniformly distributed points.

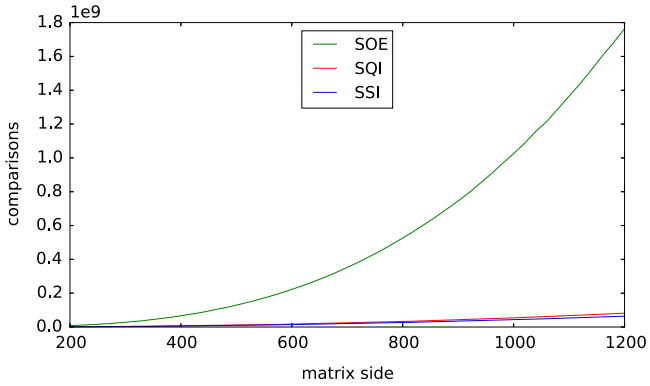

Fig. 13. Average comparisons against square matrix side, for three algorithms: SOE, SQI, and SSI.

The first set of distributions is shown in Fig. 14, which contains continuous arrangements over the domain $[0, 1] \times [0, 1]$. The *uniform* distribution generates random points with (x,y) coordinates within the $[0, 1]$ interval. The *gradient* distribution uniformly varies the density of points along the $x=y$ diagonal. The *Gaussian* distribution generates coordinates with mean 0.5 and standard deviation 0.2, clipped to the visible domain. The *Poisson* distribution generates equally spaced points on the domain, following the biased Poisson disk distribution generated by Bridson's algorithm [32].

We also created a "perfect" distribution, called *shuffle*, where points are arranged in a regular grid, thus having many repeated x and y coordinates. The points are then randomly shuffled inside the matrix before sorting. This distribution is not pictured.

Fig. 15 exemplifies some discrete distributions. The *inside* distribution selects points uniformly scattered inside a circle centered on $(0.5, 0.5)$ with radius 0.5. The *clusters* distribution generates five random and non-overlapping circles within the domain, evenly picking points that lie inside one of these circles, whereas the *holes* distribution picks points outside all those circles. The *streets* distribution places all points evenly distributed inside overlapping axis-aligned rectangles.

Another set of distributions is shown in Fig. 16, which illustrates some problematic cases for nearest neighbor queries. The *compact* distribution places all points evenly distributed in one fifth of the normalized domain, with y coordinates limited to the interval $[0, 0.2]$. The *triangle* distribution generates points only in half of the domain. The *tilted* distribution applies random rotations to an arrangement identical to the previous streets. Finally, the *diagonal* distribution places points arranged into a thin rectangle tilted by 45°, with width 0.02.

Fig. 17 shows the average running time for sorting several distributions with algorithms SQI (shown in blue) and SSI (in orange), using 100 different random seeds each. The SOE algorithm is not pictured as it performed in average 20 times slower than the other two. As we can see, most of the distributions have a similar sorting performance, with continuous distributions being faster to sort spatially. The exception is the diagonal distribution, which was found to be the pathological case for the SSI algorithm.

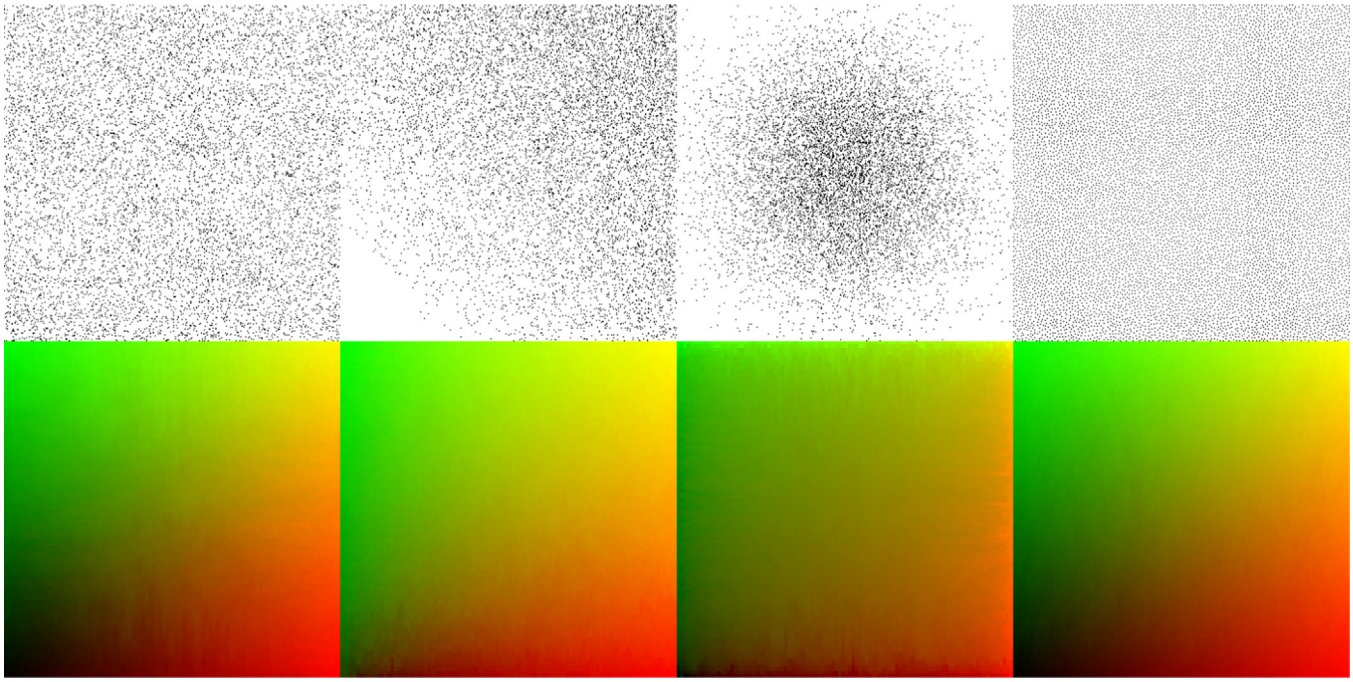

Fig. 14. Continuous point distributions (top row) and their respective color-coded spatially sorted matrices (bottom row), using SSI. The distributions are, from left to right, respectively, *uniform*, *gradient*, *gaussian*, and *poisson*. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

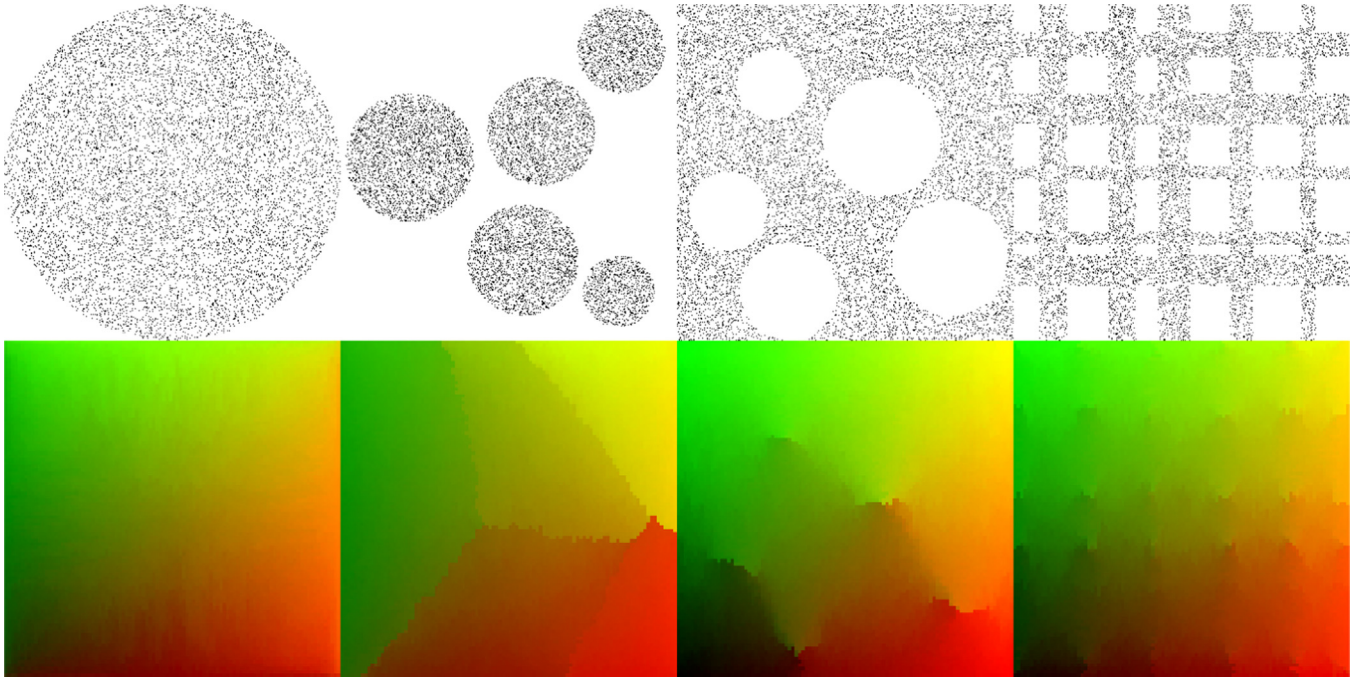

Fig. 15. Discrete point distributions (top row) and their respective color-coded spatially sorted matrices (bottom row), using SSI. The distributions are, from left to right, respectively, *inside*, *clusters*, *holes*, and *streets*. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

5.3. Nearest neighbor search

Once a full sort is established, we can select candidates to test for nearest neighbors in the vicinity of a given matrix element. We then perform the neighbor-gathering phase, where the query point is already in the dataset.

Here we have two particular situations: k-NN, when exactly k nearest neighbors are needed for each query point, and *range search*, and when all neighbors inside a radius r of the query point must be located. As these situations are similar, in this work we will focus only on the first one as it is dimension-independent. We should note that the results obtained for range search are consistently close to the ones for k nearest neighbors.

For a typical simulation application, this phase tends to be more computationally expensive than the sorting phase, because we need to test all candidates and then perform the actual calculation between near points, like the collision of particles or interaction between agents. Therefore, the neighborhood size and shape must be

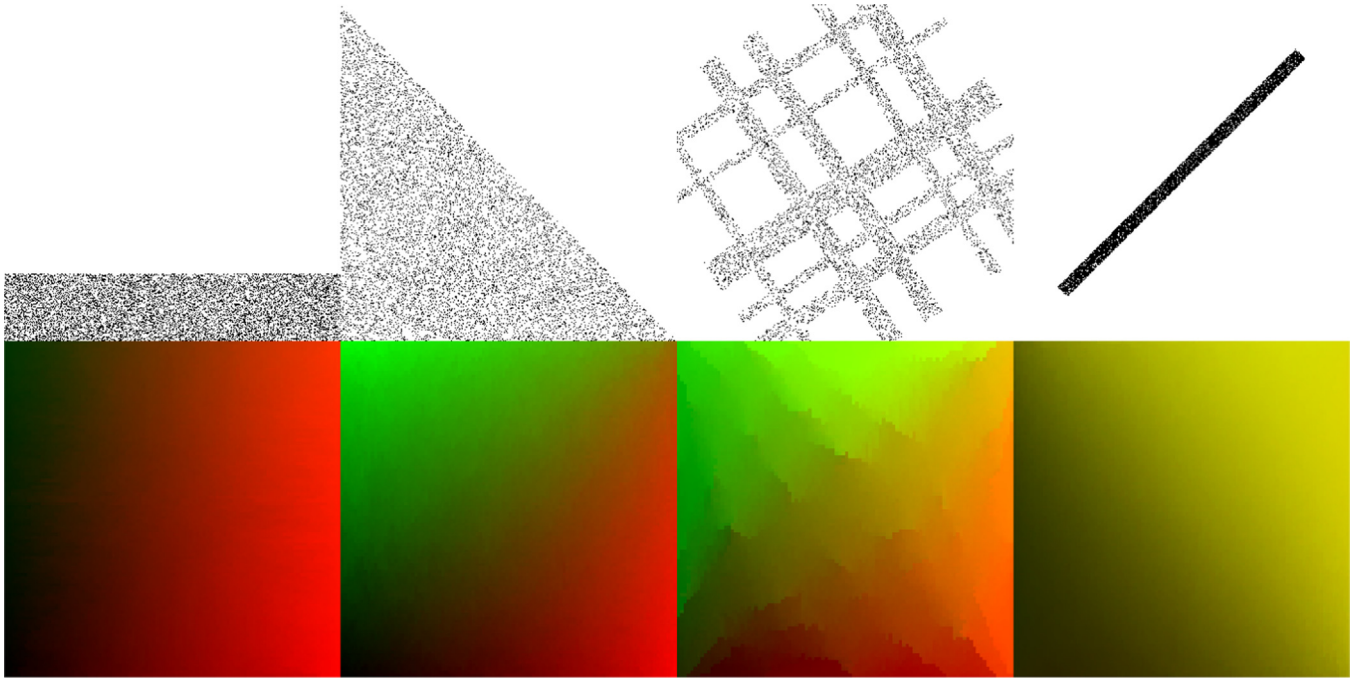

Fig. 16. Unusual point distributions (top row) and their respective color-coded spatially sorted matrices (bottom row), using SSI. The distributions are, from left to right, respectively, *compact*, *triangle*, *tilted*, and *diagonal*. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

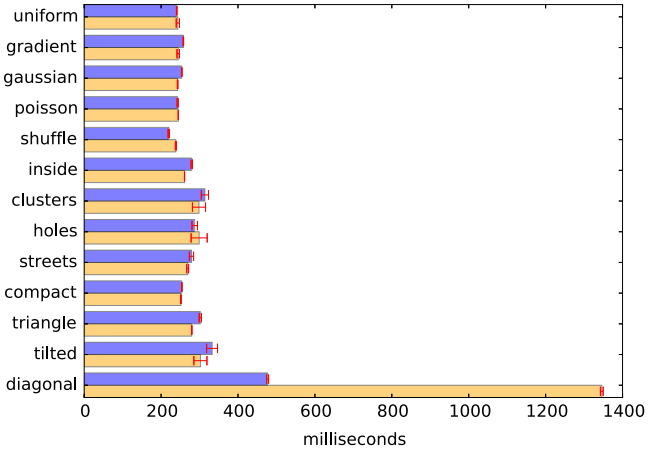

Fig. 17. Average running time for different distributions of one million points: SQI (blue) and SSI (orange). (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

chosen carefully. In these tests, we opted to use the fixed square Moore-like neighborhoods with $m = \{8, 24, 48, 80, 120\}$ neighbors.

We have measured the running time of the neighbor-gathering phase in the typical situation such that, for each point in the dataset, all k nearest neighbors are collected. Such situation is sometimes called *all nearest neighbors* in the literature. Then we compare the found set of nearest points to a "ground truth": an exact k-d tree algorithm ran on the same point distribution. We employed the implementation from CGAL [33]. This comparison makes it possible to identify precisely the number of misses for each combination of spatial sorting and candidate neighborhood size. It is also helpful to compare the total time taken, composed of the setup time (spatial sorting or k-d tree construction) added with the query time (neighbor-gathering phase or k-d tree point query).

We have also measured the precision of other approaches, to compare their effectiveness against spatial sorting. We tested the approximate k-d tree search from CGAL, which takes an *epsilon* parameter specifying the error margin to be applied when searching for nearest neighbors (we used 1.0 as it achieves a similar precision to SOE). We also tested against a partial sort, as done in [11], where a single spatial quicksort step is run before the gathering phase. Finally, we also implemented the technique that sorts points along curves, for both the Hilbert and Morton curves (following middle subdivide policy, as in CGAL). In these algorithms, after sorting, we employ a linearized neighborhood around each query point, using as candidates its immediate $m/2$ predecessors and $m/2$ successors, where m is the number of candidates. The results are shown in Fig. 18. The matrix size is again 1000×1000, with ten different random seeds for each algorithm, over a uniform point distribution. We list the comparison of setup and query time, for a reference exact k-d tree search and several other algorithms, when just one nearest neighbor is located for an m-48 neighborhood. Precision is shown as a percentage of points that had a wrong nearest neighbor in comparison with the exact results.

We see that although the setup time for the k-d tree is very small, most of the time is spent on the tree traversal to locate nearest points. On the other hand, for the SQI and SSI algorithms, the setup time is longer, and the candidate search is very efficient, thus reducing the overall time spent. Even though an approximate search is also employed for the k-d tree, the performance gains are small. We can observe that sorting along either Hilbert and Morton curves gives limited precision, as expected. The results for the SOE and SIMPLE algorithms are shown, achieving lower precision than SQI and SSI. Finally, it is clear that just a partial sorting pass is not enough to guarantee useful precision; even with an m-120 neighborhood, the miss rate would still be 90.29%.

A comparison between spatial sorting algorithms is shown in Table 1, again for a uniform distribution, and now including RANDOM. As noted before, the algorithms generate quite different final sorted results, which directly affect the search precision. Besides RANDOM, which is impracticably slow (which took about 14 min for each run), SSI is consistently the one that gives the lower miss rate.

In Fig. 19 a brief comparison is made of the running time for SSI and precision when using an m-48 neighborhood, against different point distributions. We can see that the optimal cases are the continuous distributions, even though most other discrete situations

have a miss rate around 2.5%. The only exception is again the pathological *diagonal* case.

In Table 2, we briefly summarize results for varying k while using the SSI on a uniform distribution with different candidate neighborhoods. A *hit* is achieved only when there is an exact match between the set points found. Thus, the precision measure is very strict, based on the exact k-d tree from CGAL. That said, even for a *miss*, most of the nearest points are correctly located. For example, for $k=16$ and with an m-180 search neighborhood, for all 10^6 points, only 0.1991% of them got a wrong set of neighbors. However, taking into account all $10^6 k$ neighbors located, only 0.0141% were different from the correct ones.

5.4. Assessment of point distribution

We have selected two estimates for the precision of finding nearest neighbors, given a point distribution. Instead of running the full neighbor-gathering phase and then comparing to an exact NNS search, we ought to establish simpler metrics to assess the uniformity of the input data.

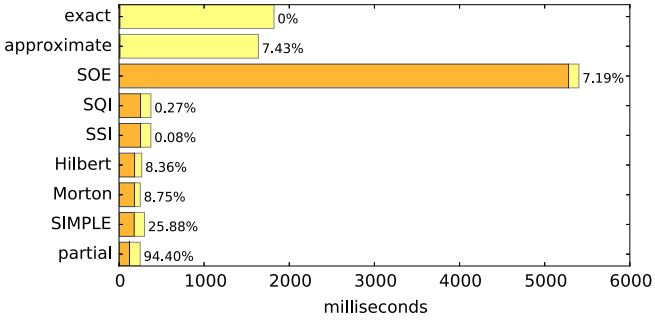

Fig. 18. Setup time (orange), query time (yellow), and precision for several algorithms, with search neighborhood m-48 and $k=1$. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

Table 1
Total time (ms) and precision for several spatial sorting algorithms and search neighborhoods, with $k=1$.

Type	SOE		SQI		SSI		RANDOM
	Time	Miss%	Time	Miss%	Time	Miss%	Miss%
m-8	5305	25.2408	278	15.5910	277	14.8581	13.2579
m-24	5340	11.9231	313	1.6832	314	1.2018	0.7519
m-48	5401	7.1885	374	0.2652	374	0.0782	0.0225
m-80	5478	4.5036	450	0.0867	454	0.0051	0.0004
m-120	5563	2.8363	536	0.0442	546	0.0003	0.0000

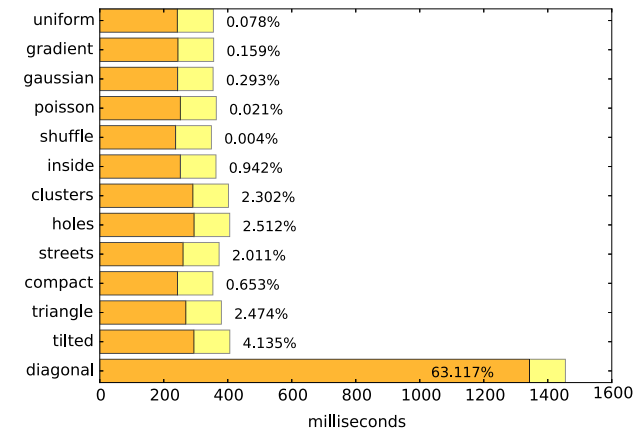

Fig. 19. Setup time using SSI (orange), query time (yellow), and precision for several distributions, with search neighborhood m-48 and $k=1$. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

A more precise measurement of uniformity is given by the Earth Mover's Distance (EMD), first described in [34]. We can compute the "work" needed to change a given point distribution into a constant distribution over a square domain. Therefore, we divided the 2D domain into $b \times b$ smaller square buckets and then counted the number of points that laid on each one. This process creates a matrix B, with integer entries. We then created another $b \times b$ matrix, called U, where each entry got the same value, resulting from the simple division of the number of points by b^2. Then we ran an EMD routine to compute the work needed to change B into U. We employed a unitary distance cost for horizontal or vertical moves between adjacent buckets, and a general Euclidean distance for the other situations. The results for several distributions with one million points, along the query precision for $k=1$, m-80 neighborhood, and using SSI, are shown in Table 3.

We can see that the amount of work done is inversely proportional to the similarity of the distributions. That is, the lower the computed EMD value, the closer it is to a constant distribution, and thus the lower the miss rate for nearest neighbors.

We also devised an even simpler metric for uniformity. After computing the number of points on the buckets for matrix B, we computed what we called a *half-mean*: we averaged the count for half of the buckets with the lowest count and divided this by the average for all buckets. So we now have a normalized metric that is 1.0 when the distribution is constant, and that tends to 0.0 as the regions with lower density increase. The resulting values are also shown in Table 3. Although not as good as EMD as an indication of suitability, we found this half-mean metric far simpler to apply and still useful as a quick check.

Table 2
Total time (ms) and precision for the SSI algorithm, when varying k.

Type	$k=4$		$k=8$		$k=16$	
	Time	Miss%	Time	Miss%	Time	Miss%
m-8	367	75.4719	400	99.8042	439	100.0000
m-24	521	11.7654	622	42.0174	741	94.9046
m-48	685	0.8329	848	4.6112	1061	28.9598
m-80	871	0.0492	1097	0.2938	1423	2.9639
m-120	1060	0.0032	1342	0.0173	1756	0.1991

Table 3
Precision, Earth Mover's distance and half-mean measure for several distributions, for $k=1$ and m-80 neighborhood.

Distribution	Miss%	EMD	Half-mean
Uniform	0.0051	0.0079	0.9921
Gradient	0.0134	1.6202	0.5373
Gaussian	0.0488	1.3740	0.2399
Poisson	0.0009	0.0022	0.9976
Shuffle	0.0006	0.0000	1.0000
Inside	0.3952	0.5240	0.7243
Clusters	1.0689	0.9332	0.1525
Holes	1.4506	0.6100	0.4145
Streets	0.9718	0.3376	0.5927
Compact	0.0607	0.0079	0.9921
Triangle	1.1545	2.3290	0.1011
Tilted	2.2393	1.0302	0.1794
Diagonal	56.8471	2.5254	0.0000

5.5. Update phase

If the points stored in the matrix change, as in a typical simulation iteration, it is straightforward to return to the sorted state by simply running a spatial sorting algorithm again. Depending on the magnitude of changes, it may be cheaper to run a few steps of alternating insertion sort (dubbed SI), as parts of the matrix could still be in an ordered state. Table 4 lists the update times for the SI and SSI algorithms, given the number of points changed and their position offset.

5.6. Comparison with other NNS algorithms

We have evaluated our proposed spatial sorting algorithms against current NNS strategies, in both serial (single-threaded CPU) and parallel (CUDA) configurations. We briefly describe the results, showing that for dense and rather homogeneous distribution of points, spatial sorting can outperform both k-d tree and index sorting.

We have built a 2048×2048 set of uniformly distributed 2D points and averaged the setup time (to build the data structure), the query time (to locate $k=10$ nearest neighbors for all given points), and the strict miss rate (that is, considering the full set of neighbors). We also computed the total time, which includes both the setup and query time. The results are shown in Table 5, ordered by average total time.

For the serial case, we tested against the Nanoflann k-d tree, which is a highly optimized C++ template for low-dimensional. In fact, it was the fastest implementation we have found. As a reference, we also kept the times given by the CGAL k-d tree, as it can scale to higher dimensions.

The parallel implementation of spatial sorting used two steps of bitonic sort, followed by several steps of spatial insertion sort (SI). As spatial Shell sort cannot be efficiently run in parallel because of its many alternating passes in x and y directions, we opted to employ bitonic sort as a standard sort. Thus, bitonic sort would be run in each row in parallel, and then on each column, again in parallel.

Table 4
Update time (ms) for two spatial sorting algorithms, given percentage of points changed and position offset.

Change (%)	SI			SSI		
	0.01	0.1	0.2	0.01	0.1	0.2
1	32.98	52.63	64.71	65.72	91.32	103.08
2	37.78	65.71	82.38	70.66	104.64	123.78
10	57.55	108.38	137.34	88.95	149.63	170.15
25	74.39	143.68	193.37	107.24	174.35	190.83
50	92.89	176.07	237.81	121.98	189.90	206.36

Table 5
Average setup time, query time, total time, and miss rate for a set of 2^{22} uniformly distributed points inside a square domain, with $k=10$.

Implementation	Setup (s)	Query (s)	Total (s)	Miss%
CGAL k-d tree	0.040	19.300	19.340	0
Nanoflann k-d tree	1.598	10.974	12.572	0
SSI m-120	1.212	5.021	6.233	0.0336
SSI m-80	1.212	3.836	5.048	0.5807
SSI m-48	1.212	2.666	3.878	8.3461
CUDA m-120	0.739	0.677	1.418	0.5018
CUDA index sort	0.045	1.322	1.367	0
CUDA m-80	0.739	0.483	1.224	1.6144
CUDA m-48	0.739	0.314	1.053	10.6954

As a reference parallel NNS, we used a finely tuned implementation of index sort for a uniform grid, based on the CUDA *particles* demo, which is described in [19]. In this case, we adapted the source to provide a 2D NNS, using on a 256×256 grid of buckets. The timings are also shown in Table 5.

Although the setup time is high for spatial sorting, it starts from an unordered matrix of points. Therefore, we can use spatial sorting to update the matrix after positions change, which can give significant gains for the repeated iterations of a simulation. For example, given a change of magnitude 0.001 on 10% of the points distributed in a unit square $[0, 1] \times [0, 1]$, the update sort would take only 0.231 s on average, against 0.739 s for the CUDA implementations, which would get a significant edge against index sort as the total time would drop from 1.418 s to 0.908 s (for the m-120 case). By using bitonic sort, we are in fact reaching a different sorted state than SSI, which is equivalent to using quicksort for each row and column, which reflects on the slight higher miss rates for the parallel implementation. Finally, it should be noted that index sort demands two complete matrices in the GPU memory, to improve memory locality when performing the query phase.

5.7. Spatial sorting in 3D

In this section, we briefly evaluate results of the previously proposed algorithms adapted to 3D.

We employed a uniform random distribution of one million points, now arranged into a tridimensional array with side 100. Again we ran each test with the same set of 10 different random seeds and averaged the timing and precision results.

In Fig. 20 we list the comparison of setup and query time for several 3D spatial sorting algorithms, against the optimized 3D k-d tree provided by Nanoflann. The precision is calculated for $k=1$ and search neighborhood m-124 (that is, a cubic neighborhood of 5^3-1 candidates). Precision is shown as a percentage of points that had a wrong nearest neighbor in comparison with the exact results. The RANDOM 3D algorithm is not depicted, as it was much slower, taking in average 79 s for each run. Yet, it achieved a miss rate of 0.23%.

We may compare this results with the ones of Fig. 18. Although the number of points is the same, in 3D the sorting time is higher. This is a direct consequence of the inclusion of the z coordinate, which increases the number of 1D sorting operations needed to run in each spatial sorting pass. Although the running time of SOE is the highest among spatial sorting strategies, now it is just two times slower than either SQI or SSI. Still, all sorting algorithms perform better than the exact Nanoflann k-d tree search. Another important point is that the query phase is now more expensive, as the square Moore neighborhood contain many more candidates to be tested. For m-124 it takes on average 394 ms, whereas for m-342 it takes 1046 ms, and for m-728 it takes 2159 ms. So the time of querying overtakes the time spent sorting the points. Overall the SSI 3D algorithm still has the better performance, whereas the unpractical RANDOM 3D strategy provides the lowest miss rate.

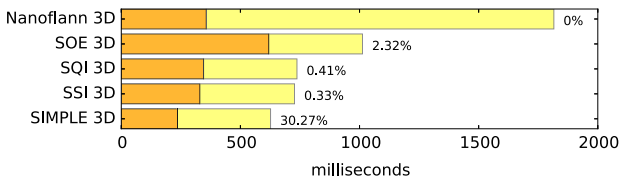

Fig. 20. Setup time (orange), query time (yellow), and precision for several 3D algorithms, with $k=1$ and search neighborhood m-124. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

Table 6
Total time (ms) and precision for both the Nanoflann 3D *k*-d tree and the SSI 3D algorithm, when varying *k*.

Type	*k*=4		*k*=8		*k*=16	
	Time	Miss%	Time	Miss%	Time	Miss%
Nanoflann 3d	2749	0	4143	0	6461	0
m-26	642	54.8384	772	88.5130	931	99.9061
m-124	1222	2.8180	1548	9.6575	2109	32.2919
m-342	2201	0.0476	2744	0.2165	3655	1.1890
m-728	3704	0.0007	4464	0.0034	5737	0.0234

In Table 6, we briefly summarize results for varying *k* while using the SSI on a 3D uniform distribution with different candidate neighborhoods.

We may compare to the results for the 2D case shown in Table 2. We can see that the precision is again mostly dependent on the search neighborhood size, as querying now dominates the total time (each SSI 3D setup only amounted to 332 ms). Therefore, to be effective, the adequate balance between precision and neighborhood size must be found. Adaptive neighborhoods could be explored to reduce the time spent on the neighbor-gathering phase.

Finally, we should note that the setup times for the evaluated algorithms were all based on an initial random state. When performing an update phase, as described in Section 5.5, we can expect much lower sorting times.

6. Summary of best practices

Here we collect practical advice summarizing how our proposed techniques could be used.

The NNS approach described in this paper should be used only when the query point is already inside the dataset. For other points, a point location algorithm can be used (Section 4.4), although this case is suboptimal.

The uniformity of the point distribution must be analyzed (Section 5.4) to estimate the resulting NNS precision. The half-mean metric is simple to implement and can give useful hints. The dimensions of the matrix should be set based on the actual point distribution (Section 4.1) to prevent distortion.

SSI is the optimal choice for CPUs, whereas SQI can be used as a simpler implementation (by reusing already existing and efficient sorting code). For the GPU case, a bitonic variation of SQI is more adequate (Section 5.6). We recommend the usage of dirty flags and write counting (Section 3.2).

The choice of candidate neighborhood size directly affects the precision and neighbor search time, therefore the many trade-offs should be evaluated for each particular situation. A square Moore neighborhood seems to be enough for most cases, as we did not measure any significant gains with a fixed round shape (Section 5.3).

Finally, a specific update step should be used for quickly adapting to changes on the point set (Section 5.5).

7. Conclusion and future work

We have presented an in-depth analysis of spatial sorting algorithms, which in turn are the basis of an efficient yet simple to implement solution for the low dimensional NNS problem. By changing the size of the candidate neighborhood, we can balance precision and performance when searching for neighbors. Besides, the presented technique is dynamic, in the sense that the data structure does not need to be recomputed from scratch for each change in the data set. Overall, this approach achieves its best performance and precision for dense and nearly uniform distributions, by using the proposed SSI algorithm in a serial implementation, or a variation of SQI in a parallel case.

We can highlight our major contributions as the proposal of novel 2D spatial sorting algorithms and the establishment of a lower bound for its time complexity. We discussed how to construct a general data structure, describing how to dynamically handle insertion and removal of points, besides the point location problem. We then have evaluated the performance and accuracy of this approach in many different and representative scenarios. Further discussion was made about the suitability of given point distributions and the importance of the update phase. We also showed that the overall results are competitive with current NNS algorithms like *k*-d tree or index sort based on uniform grids. Finally, we presented some brief results showing that the extension into 3D is straightforward and equally useful.

As future work, we plan to assess possible optimizations to GPU implementations. Another venue for research is the design of an actual spatial sorting network which could be optimal in parallel architectures. We also would like to explore this approach applied to other traditional NNS problems in Computer Graphics, such as crowd simulations. Moreover, there is also need to evaluate how to map it to other topologies (like points on the surface of a sphere).

References

[1] Li B, Mukundan R. A comparative analysis of spatial partitioning methods for large-scale, real-time crowd simulation. In: Proceedings of the 21st international conference in Central Europe on computer graphics, visualization and computer vision; 2013. p. 104–11.

[2] Walter M, Fournier A, Menevaux D. Integrating shape and pattern in mammalian models. In: Proceedings of ACM SIGGRAPH 2001. Computer graphics proceedings, Annual conference series; 2001. p. 317–26.

[3] Kammoun A, Payan F, Antonini M. Adaptive semiregular remeshing: a Voronoi-based approach. In: 2010 IEEE international workshop on multimedia signal processing (MMSP); 2010. p. 350–5.

[4] Alexa M, Kyprianidis JE. Error diffusion on meshes. Comput Graph 2015;46:336–44.

[5] Shirazian P, Wyvill B, Duprat JL. Polygonization of Implicit Surfaces on Multi-Core Architectures with SIMD Instructions. In: Childs H, Kuhlen T, Marton F, editors. Eurographics symposium on parallel graphics and visualization. Cagliari, Italy: The Eurographics Association; 2012. ISBN: 978-3-905674-35-4, http://dx.doi.org/10.2312/EGPGV/EGPGV12/089-098.

[6] Chen J, Jin X, Deng Z. GPU-based polygonization and optimization for implicit surfaces. Vis Comput 2015;31(2):119–30.

[7] Finkbeiner B, Alim UR, Ville DVD, Möller T. High-quality volumetric reconstruction on optimal lattices for computed tomography. Comput Graph Forum 2009;28(3):1023–30.

[8] Ihmsen M, Orthmann J, Solenthaler B, Kolb A, Teschner M. SPH fluids in computer graphics. In: Lefebvre S, Spagnuolo M, editors. Eurographics 2014—state of the art reports. Strasbourg, France: The Eurographics Association; 2014, http://dx.doi.org/10.2312/egst.20141034.

[9] Egges A, van Basten B. One step at a time: animating virtual characters based on foot placement. Vis Comput 2010;26(6–8):497–503.

[10] Shimada K, Gossard DC. Bubble mesh: automated triangular meshing of non-manifold geometry by sphere packing. In: Proceedings of the third ACM symposium on Solid modeling and applications. Salt Lake City, USA: ACM; 1995. p. 409–19.

[11] Joselli M, da S. Junior JR, Clua EW, Montenegro A, Lage M, Pagliosa P. Neighborhood grid: a novel data structure for fluids animation with GPU computing. J Parallel Distrib Comput 2015;75:20–8.

[12] Malheiros MdG, Walter M. Simple and efficient approximate nearest neighbor search using spatial sorting. In: 2015 28th SIBGRAPI conference on graphics, patterns and images (SIBGRAPI); 2015. p. 180–7. http://dx.doi.org/10.1109/SIBGRAPI.2015.37.

[13] Har-Peled S. A replacement for Voronoi diagrams of near linear size. In: FOCS. Las Vegas, USA: IEEE; 2001. p. 94.

[14] Rong G, Tan TS. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In: Proceedings of the 2006 symposium on Interactive 3D graphics and games. Redwood City, USA: ACM; 2006. p. 109–16.

[15] Vigueras G, Lozano M, Ordu na JM, Grimaldo F. A comparative study of partitioning methods for crowd simulations. Appl Soft Comput 2010;10(1):225–35.

[16] Guttman A. R-trees: a dynamic index structure for spatial searching. SIGMOD Rec 1984;14(2):47–57.

[17] Gieseke F, Heinermann J, Oancea C, Igel C. Buffer k-d trees: processing massive nearest neighbor queries on GPUs. In: Proceedings of the 31st international conference on machine learning (ICML-14); 2014. p. 172–80.
[18] Kofler K, Steinhauser D, Cosenza B, Grasso I, Schindler S, Fahringer T. Kd-tree based n-body simulations with volume-mass heuristic on the GPU. In: 2014 IEEE international parallel & distributed processing symposium workshops (IPDPSW). Phoenix (Arizona), USA: IEEE; 2014. p. 1256–65.
[19] Green S. Particle simulation using CUDA. NVIDIA Whitepaper; 2010.
[20] Ihmsen M, Akinci N, Becker M, Teschner M. A parallel SPH implementation on multi-core CPUs. Comput Graph Forum 2011;30(1):99–112.
[21] Kim D, Son MB, Kim YJ, Hong JM, Eui Yoon S. Out-of-core proximity computation for particle-based fluid simulations. High Perform Graph 2014:79–87.
[22] Connor M, Kumar P. Parallel construction of k-nearest neighbor graphs for point clouds. In: Proceedings of the fifth eurographics/IEEE VGTC conference on point-based graphics; 2008. p. 25–31.
[23] Gast E, Oerlemans A, Lew M. Very large scale nearest neighbor search: ideas, strategies and challenges. Int J Multimed Inf Retr 2013;2(4):229–41. http://dx.doi.org/10.1007/s13735-013-0046-4.
[24] Passos E, Joselli M, Zamith M, Rocha J, Montenegro A, Clua E, et al. Supermassive crowd simulation on GPU based on emergent behavior. In: Proceedings of the VII Brazilian symposium on computer games and digital entertainment; 2008. p. 81–6.
[25] Joselli M, Passos EB, Zamith M, Clua E, Montenegro A, Feijó B. A neighborhood grid data structure for massive 3d crowd simulation on GPU. In: 2009 VIII Brazilian symposium on games and digital entertainment (SBGAMES). Rio de Janeiro, Brazil: IEEE; 2009. p. 121–31.
[26] Lang HW. Sorting on two dimensional processor arrays. URL ⟨http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/twodim/indexen.htm⟩; 2011 [accessed 2015-09-12].
[27] Tvrdik P. Introduction to parallel sorting on mesh-based topologies. URL ⟨http://pages.cs.wisc.edu/tvrdik/15/html/Section15.html⟩; 1999 [accessed 2015-09-12].
[28] Spinrad JP. Doubly lexical ordering of dense 0-1 matrices. Inf Process Lett 1993;45(5):229–35 URL ⟨http://www.sciencedirect.com/science/article/pii/002001909390209R⟩.
[29] Haws D. Quicklexsort: an efficient algorithm for lexicographically sorting nested restrictions of a database. CoRR 2013; abs/1310.1649. URL ⟨http://arxiv.org/abs/1310.1649⟩.
[30] Galassi M, Davies J, Theiler J, Gough B, Jungman G, Booth M, et al. GNU scientific library: reference manual. Surrey, United Kingdom: Network Theory Ltd; 2003.
[31] Ciura M. Best increments for the average case of shellsort. In: Fundamentals of computation theory. Hingham, USA: Springer; 2001. p. 106–17.
[32] Bridson R. Fast Poisson disk sampling in arbitrary dimensions. In: SIGGRAPH sketches; 2007. p. 22.
[33] The CGAL Project. CGAL user and reference manual, 4.7th ed. CGAL Editorial Board. URL ⟨http://doc.cgal.org/4.7/Manual/packages.html⟩; 2015.
[34] Rubner Y, Tomasi C, Guibas LJ. The earth mover's distance as a metric for image retrieval. Int J Comput Vis 2000;40(2):99–121.