Computers & Graphics ■ (■■■■) ■■■–■■■

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

Q1 Marcelo de Gomensoro Malheiros [a], Marcelo Walter [b]

[a] Center for Exact and Technological Sciences, UNIVATES, Lajeado, Brazil

Q2 [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

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 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.

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

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

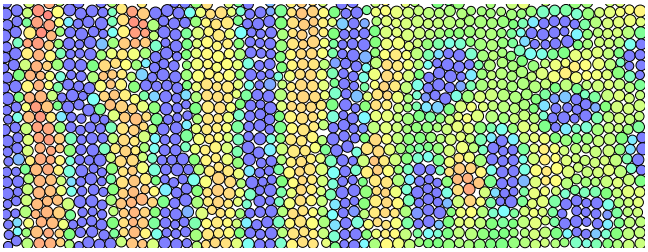

Fig. 1. Close-up detail of a biological cell simulation.

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.

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

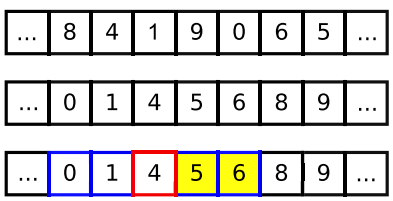

Fig. 2. Finding nearest neighbors: unsorted sequence (top), sorted array (middle), and searched elements (bottom). The query point is outlined in red, the candidates are in blue, and the two nearest neighbors are marked in yellow. (For interpretation of the references to color in this figure caption,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

Please cite this article as: Malheiros MdG, Walter M. Spatial sorting: An efficient strategy for approximate nearest neighbor searching. Comput Graph (2016), http://dx.doi.org/10.1016/j.cag.2016.03.006