Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

A Hardware Processing Unit for Point Sets

Simon Heinzle[1], Gaël Guennebaud[1,2], Mario Botsch[1,3], Markus Gross[1]

[1]Computer Graphics Laboratory, ETH Zurich
[2]Visual Computing Lab, CNR Pisa
[3]Computer Graphics Group, Bielefeld University

Abstract

We present a hardware architecture and processing unit for point sampled data. Our design is focused on fundamental and computationally expensive operations on point sets including k-nearest neighbors search, moving least squares approximation, and others. Our architecture includes a configurable processing module allowing users to implement custom operators and to run them directly on the chip. A key component of our design is the spatial search unit based on a kd-tree performing both kNN and εN searches. It utilizes stack recursions and features a novel advanced caching mechanism allowing direct reuse of previously computed neighborhoods for spatially coherent queries. In our FPGA prototype, both modules are multi-threaded, exploit full hardware parallelism, and utilize a fixed-function data path and control logic for maximum throughput and minimum chip surface. A detailed analysis demonstrates the performance and versatility of our design.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Hardware Architecture]: Graphics processors

1. Introduction

In recent years researchers have developed a variety of powerful algorithms for the efficient representation, processing, manipulation, and rendering of unstructured point-sampled geometry [GP07]. A main characteristic of such point-based representations is the lack of connectivity. It turns out that many point processing methods can be decomposed into two distinct computational steps. The first one includes the computation of some neighborhood of a given spatial position, while the second one is an operator or computational procedure that processes the selected neighbors. Such operators include fundamental, atomic ones like weighted averages or covariance analysis, as well as higher-level operators, such as normal estimation or moving least squares (MLS) approximations [Lev01, ABCO*01]. Very often, the spatial queries to collect adjacent points constitute the computationally most expensive part of the processing. In this paper, we present a custom hardware architecture to accelerate both spatial search and generic local operations on point sets in a versatile and resource-efficient fashion.

Spatial search algorithms and data structures are very well investigated [Sam06] and are utilized in many different applications. The most commonly used computations include the well known *k*-nearest neighbors (*k*NN) and the Euclidean neighbors (εN) defined as the set of neighbors within a given radius. *k*NN search is of central importance for point processing since it automatically adapts to the local point sampling rates.

Among the variety of data structures to accelerate spatial search, *k*d-trees [Ben75] are the most commonly employed ones in point processing, as they balance time and space efficiency very well. Unfortunately, hardware acceleration for *k*d-trees is non-trivial. While the SIMD design of current GPUs is very well suited to efficiently implement most point processing operators, a variety of architectural limitations leave GPUs less suited for efficient *k*d-tree implementations. For instance, recursive calls are not supported due to the lack of managed stacks. In addition, dynamic data structures, like priority queues, cannot be handled efficiently. They produce incoherent branching and either consume a lot of local resources or suffer from the lack of flexible memory caches. Conversely, current general-purpose CPUs feature a relatively small number of floating point units combined with a limited ability of their generic caches to support the particular memory access patterns generated by the recursive traversals in spatial search. The resulting inefficiency of *k*NN implementations on GPUs and CPUs is a central motivation for our work.

In this paper, we present a novel hardware architecture dedicated to the efficient processing of unstructured point sets. Its core comprises a configurable, *k*d-tree based, neighbor search module (implementing both *k*NN and εN searches) as well as a programmable processing module. Our spatial search module features a novel advanced caching mechanism that specifically exploits the spatial coherence inherent in our queries. The new caching system allows to save up to 90% of the *k*d-tree traversals depending on the application. The design includes a fixed function data path and control logic for maximum throughput and lightweight chip area. Our architecture takes maximum advantage of hardware parallelism and involves various levels of multi-threading and pipelining. The programmability of the processing module is achieved through the configurability of FPGA devices and a custom compiler.

Our lean, lightweight design can be seamlessly integrated into existing massively multi-core architectures like GPUs. Such an integration of the *k*NN search unit could be done in a similar manner as the dedicated texturing units, where neighborhood queries would be directly issued from running kernels (e.g., from vertex/fragment shaders or CUDA programs). The programmable processing module together with the arithmetic hardware compiler could be used for embedded devices [Vah07, Tan06], or for co-processors to a CPU using front side bus FPGA modules [Int07].

The prototype is implemented on FPGAs and provides a driver to invoke the core operations conveniently and transparently from high level programming languages. Operating at a rather low frequency of 75 MHz, its performance competes with CPU reference implementations. When scaling the results to frequencies realistic for ASICs, we are able to beat CPU and GPU implementations by an order of magnitude while consuming very modest hardware resources.

Our architecture is geared toward efficient, generic point processing, by supporting two fundamental operators: Cached *k*d- tree-based neighborhood searches and generic meshless operators, such as MLS projections. These concepts are widely used in computer graphics, making our architecture applicable to as diverse research fields as point-based graphics, computational geometry, global illumination and meshless simulations.

2. Related Work

A key feature of meshless approaches is the lack of explicit neighborhood information, which typically has to be evaluated on the fly. The large variety of spatial data structures for point sets [Sam06] evidences the importance of efficient access to neighbors in point clouds. A popular and simple approach is to use a fixed-size grid, which, however does not prune the empty space. More advanced techniques, such as the grid file [NHS84] or locality-preserving hashing schemes [IMRV97] provide better use of space, but to achieve high performance, their grid size has to be carefully aligned to the query range.

The quadtree [FB74] imposes a hierarchical access structure onto a regular grid using a d-dimensional d-ary search tree. The tree is constructed by splitting the space into 2^d regular subspaces. The *k*d-tree [Ben75], the most popular spatial data structure, splits the space successively into two half-spaces along one dimension. It thus combines efficient space pruning with small memory footprint. Very often, the *k*d-tree is used for *k*-nearest neighbors search on point data of moderate dimensionality because of its optimal expected-time complexity of $O(\log(n)+k)$ [FBF77, Fil79], where n is the number of points. Extensions of the initial concept include the kd-B-tree [Rob81], a bucket variant of a *k*d-tree, where the partition planes do not need to pass through the data points. In the following, we will use the term *k*d-tree to describe this class of spatial search structures.

Approximate *k*NN queries on the GPU have been presented by Ma et al. [MM02] for photon mapping, where a locality-preserving hashing scheme similar to the grid file was applied for sorting and indexing point buckets. In the work of Purcell et al. [PDC*03], a uniform grid constructed on the GPU was used to find the nearest photons, however this access structure performs only well on similarly sized search radii.

In the context of ray tracing, various hardware implementations of *k*d-tree ray traversal have been proposed. These include dedicated units [WSS05, WMS06] and GPU implementations based either on a stack-less [FS05, PGSS07] or, more recently, a stack-based approach [GPSS07]. Most of these algorithms accelerate their *k*d-tree traversal by exploiting spatial coherence using packets of multiple rays [WBWS01]. However, this concept is not geared toward the more generic pattern of *k*NN queries and does not address the neighbor sort as a priority list.

In order to take advantage of spatial coherence in nearest neighbor queries, we introduce a coherence neighbor cache system, which allows us to directly reuse previously computed neighborhoods. This caching system, as well as the *k*NN search on *k*d-tree, are presented in detail in the next section.

3. Spatial Search and Coherent Cache

In this section we will first briefly review the *k*d-tree based neighbor search and then present how to take advantage of the spatial coherence of the queries using our novel coherent neighbor cache algorithm.

3.1. Neighbor Search Using *k*d-Trees

The *k*d-tree [Ben75] is a multidimensional search tree for point data. It splits space along a splitting plane that is perpendicular to one of the coordinate axes, and hence can

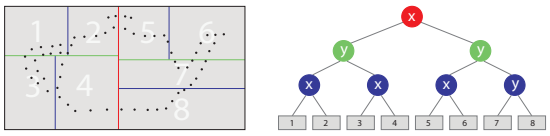

Figure 1: *The kd-tree data structure: The left image shows a point-sampled object and the spatial subdivision computed by a kd-tree. The right image displays the kd-tree, points are stored in the leaf nodes.*

be considered a special case of binary space partitioning trees [FKN80]. In its original version, every node of the tree stores a point, and the splitting plane hence has to pass through that point. A more commonly used approach is to store points, or buckets of points, in the leaf nodes only. Figure 1 shows an example of a balanced 2-dimensional *k*d-tree. Balanced *k*d-trees can always be constructed in $O(n \log_2 n)$ for n points [OvL80].

*k*NN Search

The k-nearest neighbors search in a *k*d-tree is performed as follows (Listing 1): We traverse the tree recursively down the half spaces in which the query point is contained until we hit a leaf node. When a leaf node is reached, all points contained in that cell are sorted into a priority queue of length k. In a backtracking stage, we recursively ascend and descend into the other half spaces if the distance from the query point to the farthest point in the priority queue is greater than the distance of the query point to the cutting plane. The priority queue is initialized with elements of infinite distance.

```
Point query; // Query Point
PriorityQueue pqueue; // Priority Queue of length k

void find_nearest (Node node) {
  if (node.is_leaf) {
    // Loop over all points contained by the leaf's bucket
    // and sort into priority queue.
    for (each point p in node)
      if (distance(p,query) < pqueue.max())
        pqueue.insert(p);
  } else {
    partition_dist = distance(query, node.partition_plane);
    // decide whether going left or right first
    if (partition_dist > 0) {
      find_nearest(node.left);
      // taking other branch only if it is close enough
      if (pqueue.max() > abs(partition_dist))
        find_nearest(node.right);
    } else {
      find_nearest(node.right);
      if (pqueue.max() > abs(partition_dist))
        find_nearest(node.left);
  }
}
```

Listing 1: *Recursive search of the kNN in a kd-tree.*

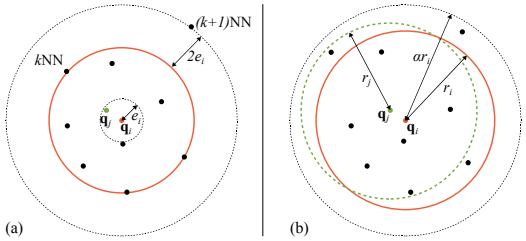

Figure 2: *The principle of our coherent neighbor cache algorithm. (a) In the case of kNN search the neighborhood of $\mathbf{q}_i$ is valid for any query point $\mathbf{q}_j$ within the tolerance distance e_i. (b) In the case of εN search, the extended neighborhood of $\mathbf{q}_i$ can be reused for any ball query $(\mathbf{q}_j, r_j)$ which is inside the extended ball $(\mathbf{q}_i, \alpha r_i)$.*

εN Search

An εN search, also called ball or range query, aims to find all the neighbors around a query point $\mathbf{q}_i$ within a given radius r_i. However, in most applications it is desirable to bound the maximum number of found neighbors. Then, the ball query is equivalent to a *k*NN search where the maximum distance of the selected neighbors is bound by r_i. In the above algorithm, this behavior is trivially achieved by initializing the priority queue with placeholder elements at a distance r_i.

Note that in high-level programming languages, the stack stores all important context information upon a recursive function call and reconstructs the context when the function terminates. As we will discuss subsequently, this stack has to be implemented and managed explicitly in a dedicated hardware architecture.

3.2. Coherent Neighbor Cache

Several applications, such as up-sampling or surface reconstruction, issue densely sampled queries. In these cases, it is likely that the neighborhoods of multiple query points are the same. The coherent neighbor cache (CNC) exploits this spatial coherence to avoid multiple computations of similar neighborhoods. The basic idea is to compute slightly more neighbors than necessary, and use this extended neighborhood for subsequent, spatially close queries.

Assume we query the *k*NN of the point $\mathbf{q}_i$ (Figure 2a). Instead of looking for the k nearest neighbors, we compute the $k+1$ nearest neighbors $N_i = \{\mathbf{p}_1, ..., \mathbf{p}_{k+1}\}$. Let e_i be half the difference of the distances between the query point and the two farthest neighbors: $e_i = (\|\mathbf{p}_{k+1} - \mathbf{q}_i\| - \|\mathbf{p}_k - \mathbf{q}_i\|)/2$. Then, e_i defines a tolerance radius around $\mathbf{q}_i$ such that the *k*NN of any point inside this ball are guaranteed to be equal to $N_i \setminus \{\mathbf{p}_{k+1}\}$.

In practice, the cache stores a list of the m most recently used neighborhoods N_i together with their respective query point $\mathbf{q}_i$ and tolerance radius e_i. Given a new query point

$\mathbf{q}_j$, if the cache contains a N_i such that $\|\mathbf{q}_j - \mathbf{q}_i\| < e_i$, then $N_j = N_i$ is reused, otherwise a full kNN search is performed.

In order to further reduce the number of cache misses, it is possible to compute even more neighbors, i.e., the $k+c$ nearest ones. However, for $c \neq 1$ the extraction of the true kNN would then require to sort the set N_i at each cache hit, which consequently would prevent the sharing of such a neighborhood by multiple processing threads.

Moreover, we believe that in many applications it is preferable to tolerate some approximation in the neighborhood computation. Given any positive real ε, a data point $\mathbf{p}$ is a $(1+\varepsilon)$-*approximate k-nearest neighbor* (AkNN) of $\mathbf{q}$ if its distance from $\mathbf{q}$ is within a factor of $(1+\varepsilon)$ of the distance to the true k-nearest neighbor. As we show in our results, computing AkNN is sufficient in most applications. This tolerance mechanism is accomplished by computing the value of e_i as follows,

$$e_i = \frac{\|\mathbf{p}_{k+1} - \mathbf{q}_i\| \cdot (1+\varepsilon) - \|\mathbf{p}_k - \mathbf{q}_i\|}{2+\varepsilon} \,. \qquad (1)$$

The extension of the caching mechanism to ball queries is depicted in Figure 2b. Let r_i be the query radius associated with the query point $\mathbf{q}_i$. First, an extended neighborhood of radius αr_i with $\alpha > 1$ is computed. The resulting neighborhood N_i can be reused for any ball query $(\mathbf{q}_j, r_j)$ with $\|\mathbf{q}_j - \mathbf{q}_i\| < \alpha r_i - r_j$. Finally, the subsequent processing operators have to check for each neighbor its distance to the query point in order to remove the wrongly selected neighbors. The value of α is a tradeoff between the cache hit rate and the overhead to compute the extended neighborhood. Again, if an approximate result is sufficient, then a $(1+\varepsilon)$-AkNN like mechanism can be accomplished by reusing N_i if the following coherence test holds: $\|\mathbf{q}_j - \mathbf{q}_i\| < (\alpha r_i - r_j) \cdot (1+\varepsilon)$.

4. A Hardware Architecture for Generic Point Processing

In this section we will describe our hardware architecture implementing the algorithms introduced in the previous section. In particular, we will focus on the design decisions and features underlying our processing architecture, while the implementations details will be described in Section 5.

4.1. Overview

Our architecture is designed to provide an optimal compromise between flexibility and performance. Figure 3 shows a high-level overview of the architecture. The two main modules, the neighbor search module and the processing module, can both be operated separately or in tandem. A global thread control unit manages user input and output requests as well as the module's interface to high level programming languages, such as C++.

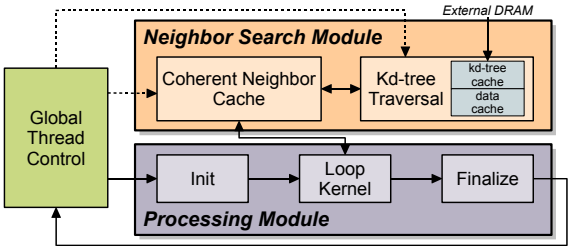

Figure 3: *High-level overview of our architecture. The two modules can be operated separately or in tandem.*

The core of our architecture is the configurable *neighbor search module*, which is composed of a kd-tree traversal unit and a coherent neighbor cache unit. We designed this module to support both kNN and εN queries with maximal sharing of resources. In particular, all differences are managed locally by the coherent neighbor cache unit, while the kd-tree traversal unit works regardless of the kind of query. This module is designed with fixed function data paths and control logic for maximum throughput and for moderate chip area consumption. We furthermore designed every functional unit to take maximum advantage of hardware parallelism. Multithreading and pipelining were applied to hide memory and arithmetic latencies. The fixed function data path also allows for minimal thread-storage overhead. All external memory accesses are handled by a central memory manager and supported by data and kd-tree caches.

In order to provide optimal performance on a limited hardware, our *processing module* is also implemented using a fixed function data path design. Programmability is achieved through the configurability feature of FPGA devices and by using a custom hardware compiler. The integration of our architecture with existing or future general purpose computing units, like GPUs, is discussed in section 6.2.

A further fundamental design decision is that the kd-tree construction is currently performed by the host CPU and transferred to the subsystem. This decision is justified given that the tree construction can be accomplished in a preprocess for static point sets, whereas neighbor queries have to be carried out at runtime for most point processing algorithms. Our experiments have also shown that for moderately sized dynamic data sets, the kd-tree construction times are negligible compared to the query times.

Before going more into detail, it is instructive to describe the procedural and data flows of a generic operator applied to some query points. After the requests are issued for a given query point, the coherent neighbor cache is checked. If a cached neighborhood can be reused, a new processing request is generated immediately. Otherwise, a new neighbor search thread is issued. Once a neighbor search is terminated, the least recently used neighbor cache entry is replaced with the attributes of the found neighbors and a pro-

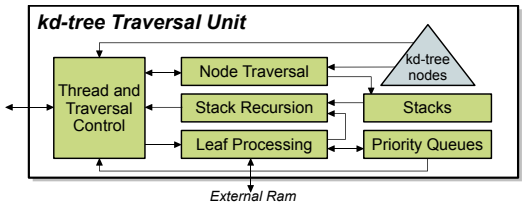

Figure 4: *Top level view of the kd-tree traversal unit.*

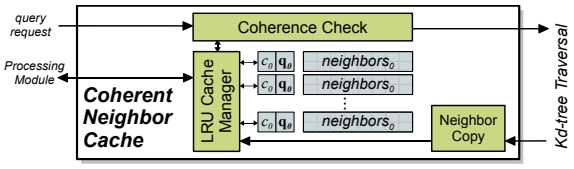

Figure 5: *Top level view of coherent neighbor cache unit.*

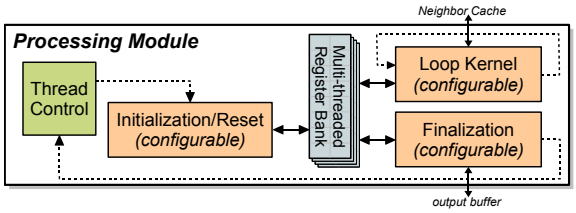

Figure 6: *Top level view of our programmable processing module.*

cessing thread is generated. The processing thread loops over the neighbors and writes the results into the delivery buffer, from where they are eventually read back by the host.

In all subsequent figures, *blue* indicates memory while *green* stands for arithmetic and control logic.

4.2. *k*d-Tree Traversal Unit

The *k*d-tree traversal unit is designed to receive a query $(\mathbf{q}, r)$ and to return at most the k-nearest neighbors of $\mathbf{q}$ within a radius r. The value of k is assumed to be constant for a batch of queries.

This unit starts a query by initializing the priority queue with empty elements at distance r, and then performs the search following the algorithm of Listing 1. While this algorithm is a highly sequential operation, we can identify three main blocks to be executed in parallel, due to their independence in terms of memory access. As depicted in Figure 4, these blocks include node traversal, stack recursion, and leaf processing.

The node traversal unit traverses the path from the current node down to the leaf cell containing the query point. Memory access patterns include reading of the *k*d-tree data structure and writing to a dedicated stack. This stack is explicitly managed by our architecture and contains all traversal information for backtracking. Once a leaf is reached, all points contained in that leaf node need to be inserted and sorted into a priority queue of length k. Memory access patterns include reading point data from external memory and read-write access to the priority queue. After a leaf node has been left, backtracking is performed by recurring up the stack until a new downward path is identified. The only memory access is reading the stack.

Search mode	***k*NN**	**εN**
$c_i =$	e_i (equation 1)	distance of top element
Skip top element:	always	if it is the empty element
Coherence test:	$\|\mathbf{q}_j - \mathbf{q}_i\| < c_i$	$\|\mathbf{q}_j - \mathbf{q}_i\| < c_i - r_j$
Generated query:	$(\mathbf{q}_j, \infty)$	$(\mathbf{q}_j, \alpha r_j)$

Table 1: *Differences between kNN and εN modes.*

4.3. Coherent Neighbor Cache Unit

The coherent neighbor cache unit (CNC), depicted in Figure 5, maintains a list of the m most recently used neighborhoods in a least recently used order (LRU). For each cache entry the list of neighbors N_i, its respective query position $\mathbf{q}_i$, and a generic scalar comparison value c_i, as defined in Table 1, are stored. The *coherence check* unit uses the latter two values to determine possible cache hits and issues a full *k*d-tree search otherwise.

The *neighbor copy* unit updates the neighborhood caches with the results from the *k*d-tree search and computes the comparison value c_i according to the current search mode. For correct *k*NN results, the top element corresponding to the $(k+1)$NN needs to be skipped. In the case of εN queries all empty elements are discarded. The subtle differences between *k*NN and εN are summarized in Table 1.

4.4. Processing Module

The processing module, depicted in Figure 6, is composed of three customizable blocks: an initialization step, a loop kernel executed sequentially for each neighbor, and a finalization step. The three steps can be globally iterated multiple times, where the finalization step controls the termination of the loop. This outer loop is convenient to implement, e.g., an iterative MLS projection procedure. Listing 2 shows an instructive control flow of the processing module.

All modules have access to the query data (position, radius, and custom attributes) and exchange data through a shared register bank. The initialization step furthermore has access to the farthest neighbor, which can be especially useful to, e.g., estimate the sampling density. All modules operate concurrently, but on different threads.

The three customizable blocks are specified using a pseudo assembly language supporting various kinds of arith-

```
Vertex neighbors[k]; // custom type
OutputType result; // custom type
int count = 0;
do {
  init(query_data, neighbors[k], count);
  for (i=1..k)
    kernel(query_data, neighbors[i]);
} while (finalization(query_data, count++, &result));
```

Listing 2: *Top-level algorithm implemented by the processing module.*

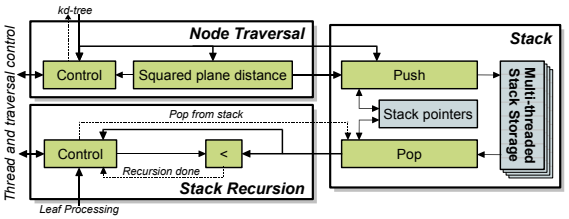

Figure 7: *Detailed view of the sub-units and storage of the node traversal, stack recursion and stack units.*

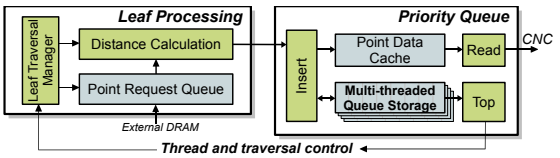

Figure 8: *Detailed view of the leaf processing and parallel priority queue units. The sub-units load external data, compute distance and resort the parallel priority queues.*

metic operations, comparison operators, reads and conditional writes to the shared register bank, and fixed size loops achieved using loop unrolling. Our arithmetic compiler then generates hardware descriptions for optimized fixed function data paths and control logic.

5. Prototype Implementation

This section describes the prototype implementation of the presented architecture using Field Programmable Gate Arrays (FPGAs). We will focus on the key issues and non-trivial implementation aspects of our system. At the end of the section, we will also briefly sketch some possible optimizations of our current prototype, and describe our GPU based reference implementation that will be used for comparisons in the result section.

5.1. System Setup

The two modules, neighbor search and processing, are mapped onto two different FPGAs. Each FPGA is equipped with a 64 bit DDR DRAM interface and both are integrated into a PCI carrier board, with a designated PCI bridge for host communication. The two FPGAs communicate via dedicated LVDS DDR communication units. The nearest neighbors information is transmitted through a 64 bit channel, the results of the processing operators are sent back using a 16 bit channel. The architecture would actually fit onto a single Virtex2 Pro chip, but we strived to cut down the computation times for the mapping, placing, and routing steps in the FPGA synthesis. The communication does not degrade performance and adds only a negligible latency to the overall computation.

5.2. *k*d-Tree Traversal Unit

We will now revisit the *k*d-tree traversal unit of the neighbor search module in Figure 4 and discuss its five major units from an implementational perspective.

There are up to 16 parallel threads operating in the *k*d-tree traversal unit. A detailed view of the stack, stack recursion, and node traversal units is presented in Figure 7. The *node traversal unit* pushes the paths not taken during traversal onto a shared stack for subsequent backtracking. The maximum size of the stack is bound by the maximum depth of the tree. Compared to general purpose architectures, however, our stacks are lightweight and stored on-chip to maximize performance. They only store pointers to the tree paths not taken as well as the distance of the query point to the bounding plane (2+4 Bytes). Our current implementation includes 16 parallel stacks, one for each thread and each being of size 16, which allows us to write to any stack and read from any other one in parallel.

The leaf processing and priority queue units are presented in greater detail in Figure 8. Leaf points are sorted into one of the 16 parallel priority queues, according to their distance to the query point. The queues store the distances as well as pointers to the *point data cache*. Our implementation uses a fully sorted parallel register bank of length 32 and allows the insertion of one element in a single cycle. Note that this fully sorted bank works well for small queue sizes because of the associated small propagation paths. For larger k, a constant time priority queue [WHA*07] could be used.

For ease of implementation, the *kd-tree structure* is currently stored linearly on-chip in the spirit of a heap, which can also be considered as a cache. The maximum tree depth is 14. The internal nodes and the leaves are stored separately, points are associated only with leaf nodes. The $2^{14}-1$ internal nodes store the splitting planes (32 bit), the dimension (2 bit) and a flag indicating leaf nodes (1 bit). This additional bit allows us to support unbalanced *k*d-trees as well. The 2^{14} leaf nodes store begin and end pointers to the point buckets in the off-chip DRAM (25 bits each). The total storage requirement of the *k*d-tree is 170 kBytes.

5.3. Coherent Neighbor Cache Unit

The CNC unit (Figure 5) includes 8 cached neighborhoods and one single *coherence check* sub-unit, testing for cache hits in an iterative manner. The *cache manager* unit maintains the list of caches in LRU order, and synchronizes between the processing module and the *k*d-tree search unit using a multi-reader write lock primitive. For a higher number of caches, the processing time increases linearly. Note, however, additional coherence test units could be used to reduce the number of iterations.

As the neighbor search unit processes multiple queries in parallel, it is important to carefully align the processing order. Usually, subsequent queries are likely to be spatially coherent and would eventually be issued concurrently to the neighbor search unit. To prevent this problem, we interleave the queries. In the current system this task is left to the user, which allows to optimally align the interleaving to the nature of the queries.

5.4. Processing Module

The processing module (Figure 6) is composed of three custom units managed by a *processing controller*. These units communicate through a multi-threaded quad-port bank of 16 registers. The repartition of the four ports to the three custom units is automatically determined by our compiler.

Depending on the processing operator, our compiler might produce a high number of floating point operations thus leading to significant latency, which is, however, hidden by pipelining and multi-threading. Our implementation allows for a maximum of 128 threads operating in parallel and is able to process one neighbor per clock cycle. In addition, a more fine-grained multi-threading scheme iterating over 8 sub-threads is used to hide the latency of the accumulation in the loop kernel.

5.5. Resource Requirements and Extensions

Our architecture was designed using minimal on-chip resources. As a result, the neighbor search module is very lean and uses a very small number of arithmetic units only, as summarized in Table 2. The number of arithmetic units of the processing module depends entirely on the processing operator. Their complexity is therefore limited by the resources of the targeted FPGA device. Table 2 shows two such examples.

Arithmetic Unit	*k*d-tree Traversal	CNC	Covariance Analysis	SPSS Projection
Add/Sub	6	6	38	29
Mul	4	6	49	32
Div	0	0	2	3
Sqrt	0	2	2	2

Table 2: *Usage of arithmetic resources for the two units of our neighbor search module, and two processing examples.*

Data	Current Prototype	Off-chip *k*d-tree & shared data cache
Thread data	1.36 kB (87 B/thd)	1.36 kB
Traversal stack	2 kB (8×16 B/thd)	2 kB
*k*d-tree	170 kB (depth: 14)	16 kB (cache)
Priority queue	3 kB (6×32 B/thd)	4 kB
DRAM manager	5.78 kB	5.78 kB
Point data cache	16 kB (p-queue unit)	16 kB (shared cache)
Neighbor caches	8.15 kB (1044B/cache)	1.15 kB (148 B/cache)
Total	206.3 kB	46.3 kB

Table 3: *On-chip storage requirements for our current and planned, optimized version of the neighbor search module.*

The prototype has been partitioned into the neighbor search module integrated on a Virtex2 Pro 100 FPGA, and the processing module was integrated on a Virtex2 Pro 70 FPGA. The utilization of the neighbor search FPGA was 23'397 slice flip flops and 33'799 LUTs. The utilization of the processing module in the example of a MLS projection procedure based on plane fits [AA04] required 31'389 slice flip flops and 35'016 LUTs.

The amount of on-chip RAM required by the current prototype is summarized in Table 3, leaving out the buffers for PCI transfer and inter-chip communication which are not relevant for the architecture. This table also includes a planned variant using one bigger FPGA only. Using only one chip, the architecture then could be equipped with generic shared caches to access the external memory, the tree structure would also be stored off-chip and hence alleviate the current limitation on the maximum tree depth. Furthermore, such caches would make our current *point data cache* obsolete and reduce the neighbor cache foot-print by storing references to the point attributes. Finally, this would not only optimize on-chip RAM usage, but also reduce the memory bandwidth to access the point data for the *leaf processing* unit, and hence speed up the overall process.

5.6. GPU Implementation

For comparison, we implemented a *k*d-tree based *k*NN search algorithm using NVIDIA's CUDA. Similar to the FPGA implementation, the lightweight stacks and priority queues are stored in local memory. Storing the stacks and priority queues in fast shared memory would limit the number of threads drastically and degrade performance compared to using local memory. The neighbor lists are written and read in a large buffer stored in global memory. We implemented the subsequent processing algorithms as a second, distinct kernel. Owing to the GPU's SIMD design, implementing a CNC mechanism is not feasible and would only decrease the performance.

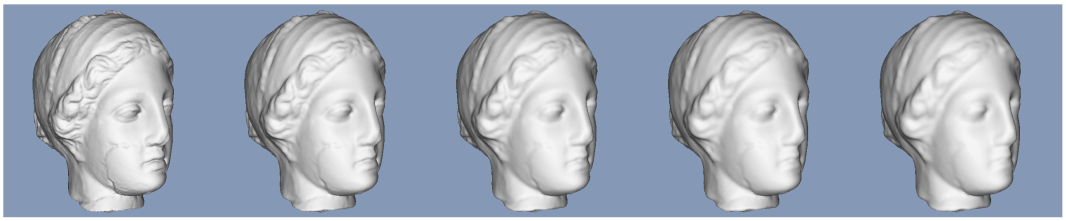

Figure 9: *A series of successive smoothing operations on the Igea model. The model size is 134k points. A neighborhood of size 16 has been used for the MLS projections. Only MLS software code has been replaced by FPGA driver calls.*

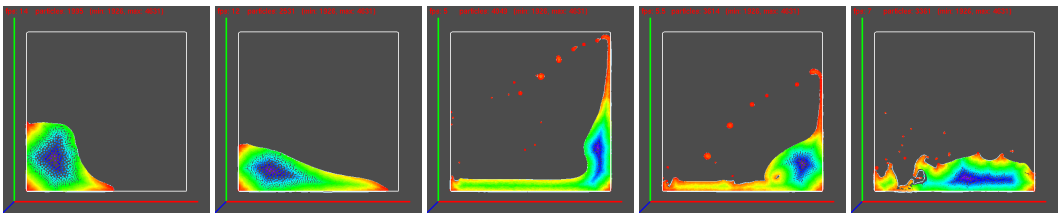

Figure 10: *A series of successive simulation steps of the 2D breaking dam scenario using SPH. The simulation is using adaptive particle resolutions between 1900 and 4600 particles, and performs kNN queries up to 30 nearest neighbors. Only kNN software code has been replaced by FPGA driver calls.*

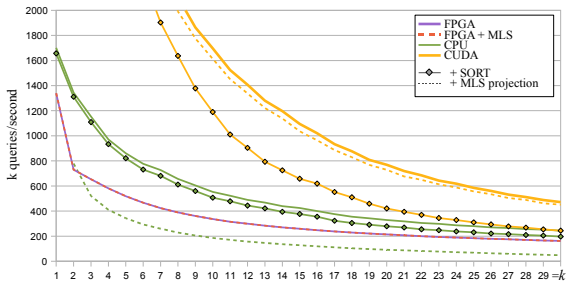

Figure 11: *Performances of kNN and MLS projection as a function of the number of neighbors k. The input points have been used as query points.*

6. Results and Discussions

To demonstrate the versatility of our architecture, we implemented and analyzed several meshless processing operators. These include a few core operators which are entirely performed on-chip: covariance analysis, iterative MLS projection based on either plane fit [AA04] or spherical fit [GG07], and a meshless adaptation of a nonlinear smoothing operator for surfaces [JDD03]. We integrated these core operators into more complex procedures, such as a MLS based resampling procedure, as well as a normal estimation procedure based on covariance analysis [HDD*92].

We also integrated our prototype into existing publicly available software packages. For instance, in PointShop 3D [ZPKG02] the numerous MLS calls for smoothing, hole filling, and resampling [WPK*04] have been replaced by calls to our drivers. See Figure 9 for the smoothing operation. Furthermore, an analysis of a fluid simulation research code [APKG07] based on *smoothed particle hydrodynamics* (SPH) showed that all the computations involving neighbor search and processing can easily be accelerated, while the host would still be responsible for collision detection and *k*d-tree updates (Figure 10).

6.1. Performance Analysis

Both FPGAs, a Virtex2 Pro 100 and a Virtex2 Pro 70, operate at a clock frequency of 75 MHz. We compare the performance of our architecture to similar CPU and GPU implementations, optimized for each platform, on a 2.2 GHz Intel Core Duo 2 equipped with a NVIDIA GeForce 8800 GTS GPU. Our results were obtained for a surface data-set of 100k points in randomized order and with a dummy operator that simply reads the neighbor attributes. Note that our measurements do not include transfer costs since our hardware device lacks an efficient transfer interface between the host and the device.

General Performance Analysis

Figure 11 demonstrates the high performance of our design for generic, incoherent *k*NN queries. The achieved on-chip FPGA query performance is about 68% and 30% of the throughput of the CPU and GPU implementations, respectively, although our FPGA clock rate is 30 times lower than that of the CPU, and it consumes considerably fewer

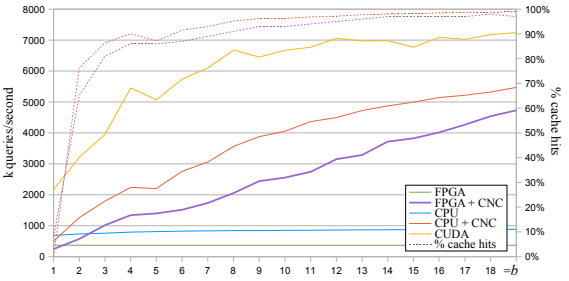

Figure 12: *Number of ball queries per second for an increasing level of coherence.*

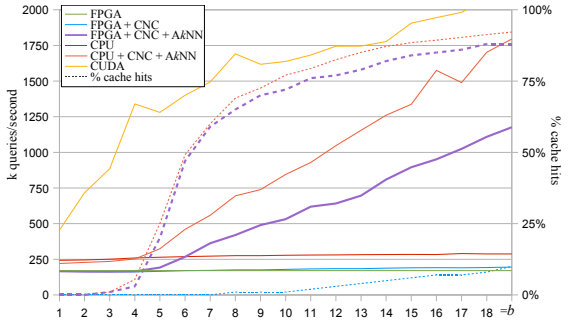

Figure 13: *Number of kNN queries per second for an increasing level of coherence ($k = 30$). Approximate kNN (AkNN) results were obtained with* $\varepsilon = 0.1$.

resources. Moreover, with the MLS projection enabled, our prototype exhibits the same performance as with the *k*NN queries only, and outperforms the CPU implementation.

Finally, note that when the CNC is disabled our architecture produces fully sorted neighborhoods for free, which is beneficial for a couple of applications. As shown in Figure 11 adding such a sort to our CPU and GPU implementations has a non-negligible impact, in particular for the GPU.

Our FPGA prototype, integrated into the fluid simulation [APKG07], achieved half of the performance of the CPU implementation, because up to 30 neighbors per query have to be read back over the slow PCI transfer interface. In the case of the smoothing operation of PointShop 3D [WPK*04], our prototype achieved speed ups of a factor of 3, including PCI communication. The reasons for this speed up are two-fold: first, for MLS projections, only the projected positions need to be read back. Second, the *k*d-tree of PointShop 3D is not as highly optimized as the reference CPU implementation used for our other comparisons. Note that using a respective ASIC implementation and a more advanced PCI Express interface, the performance of our system would be considerably higher.

The major bottleneck of our prototype is the *k*d-tree traversal, and it did not outperform the processing module for all tested operators. In particular, the off-chip memory accesses in the *leaf processing* unit represent the limiting bottleneck in the *k*d-tree traversal. Consequently, the nearest neighbor search does not scale well above 4 threads, while supporting up to 16 threads.

Coherent Neighbor Cache Analysis

In order to evaluate the efficiency of our coherent neighbor cache with respect to the level of coherence, we implemented a resampling algorithm that generates $b \times b$ query points for each input point, uniformly spread over its local tangent plane [GGG08]. All results for the CNC were obtained with 8 caches.

The best results were obtained for ball queries (Figure 12), where even an up-sampling pattern of 2×2 is enough to save up to 75% of the *k*d-tree traversals, thereby showing the CNC's ability to significantly speed up the overall computation. Figure 13 depicts the behavior of the CNC with both exact and $(1+\varepsilon)$-approximate *k*NN. Whereas the cache hit rate remains relatively low for exact *k*NN (especially with such a large neighborhood), already a tolerance ($\varepsilon = 0.1$) allows to save more than 50% of the *k*d-tree traversals. For incoherent queries, the CNC results in a slight overhead due to the search of larger neighborhoods. The GPU implementation does not include a CNC, but owing to its SIMD design and texture caches, its performance significantly drops down as the level of coherence decreases.

While these results clearly demonstrate the general usefulness of our CNC algorithm, they also show the CNC hardware implementation to be slightly less effective than the CPU-based CNC. The reasons for this behavior are twofold. First, from a theoretical point of view, increasing the number of threads while keeping the same number of caches decreases the cache hit rate. This behavior could be compensated by increasing the number of caches. Second, our current prototype consistently checks all caches while our CPU implementation stops at the first cache hit. With a high cache hit rate such as in Figure 12, we observed speed-ups of a factor 2 using this optimization on the CPU.

An analysis of the $(1+\varepsilon)$-approximate *k*NN in terms of cache hit rate and relative error can be found in Figures 14 and 15. These results show that already small values of ε are sufficient to significantly increase the percentage of cache hits, while maintaining a very low error for the MLS projection. In fact, the error is of the same order as the numerical order of the MLS projection. Even larger tolerance values like $\varepsilon = 0.5$ lead to visually acceptable results, which is due to the weight function of the MLS projection that results in low influence of the farthest neighbors.

6.2. GPU Integration

Our results show that the GPU implementation of kNN search is only slightly faster than our current FPGA prototype and CPU implementation. Moreover, with a MLS projection operator on top of a kNN search, we observe a projection rate between 0.4M and 3.4M projections per second, while the same hardware is able to perform up to 100M of projections per second using precomputed neighborhoods [GGG08]. Actually, the kNN search consumes more than 97% of the computation time.

This poor performance is partly due to the divergence in the tree traversal, but even more important, due to the priority queue insertion in $O(\log\ k)$, which infers many incoherent execution paths. On the other hand, our design optimally parallelizes the different steps of the tree traversal and allows the insertion of one neighbor into the priority queue in a single cycle.

These results motivate the integration of our lightweight neighbor search module into such a massively multi-core architecture. Indeed, a dedicated ASIC implementation of our module could be further optimized and run at a much higher frequency and could improve the performance by more than an order of magnitude. Such an integration could be done in a similar manner as the dedicated texturing units of current GPUs.

In such a context, our processing module would then be replaced by more generic computing units. Nevertheless, we emphasize that the processing module still exhibits several advantages. First, it allows to optimally use FPGA devices as co-processors to CPUs or GPUs, which can be expected to become more and more common in the upcoming years. Second, unlike the SIMD design of GPU's microprocessors, our custom design with three sub-kernels allows for optimal throughput, even in the case of varying neighbor loops.

7. Conclusion

We presented a novel hardware architecture for efficient nearest-neighbor searches and generic meshless processing operators. In particular, our kd-tree based neighbor search module features a novel and dedicated caching mechanism exploiting the spatial coherence of the queries. Our results show that neighbor searches can be accelerated efficiently by identifying independent parts in terms of memory access. Our architecture is implemented in a fully pipelined and multi-threaded manner and suggests that its lightweight design could be easily integrated into existing computing or graphics architectures, and hence be used to speed up applications depending heavily on data structure operations. When scaled to realistic clock rates, our implementation achieves speedups of an order of magnitude compared to reference implementations. Our experimental results prove the high benefit of a dedicated neighbor search hardware.

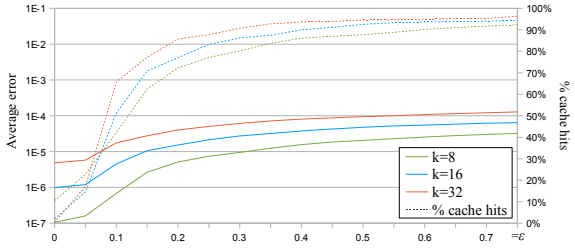

Figure 14: *Error versus efficiency of the* $(1+\varepsilon)$*-approximate k-nearest neighbor mechanism as function of the tolerance value* ε*. The error corresponds to the average distance for a model of unit size. The percentages of cache hit were obtained with* 8 *caches and patterns of* 8×8*.*

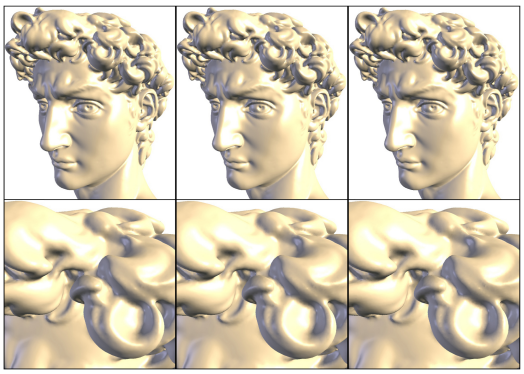

Figure 15: *MLS reconstructions with, from left to right, exact kNN (34 s), and AkNN with* $\varepsilon = 0.2$ *(12 s) and* $\varepsilon = 0.5$ *(10 s). Without our CNC the reconstruction takes 54 s.*

A current limitation of the design is its off-chip tree construction. An extended architecture could construct or update the tree on-chip to avoid expensive host communication. We also would like to investigate configurable spatial data structure processors in order to support a wider range of data structures and applications.

References

[AA04] ALEXA M., ADAMSON A.: On normals and projection operators for surfaces defined by point sets. In *Symposium on Point-Based Graphics* (2004), pp. 149–155.

[ABCO*01] ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Point set surfaces. In *IEEE Visualization* (2001), pp. 21–28.

[APKG07] ADAMS B., PAULY M., KEISER R., GUIBAS L. J.: Adaptively sampled particle fluids. *ACM Transactions on Graphics 26*, 3 (2007), 48:1–48:7.

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[FB74] FINKEL R. A., BENTLEY J. L.: Quad trees: A data structure for retrieval on composite keys. *Acta Informatica 4*, 1 (1974), 1–9.

[FBF77] FRIEDMAN J. H., BENTLEY J. L., FINKEL R. A.: An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software 3*, 3 (1977), 209–226.

[Fil79] FILHO Y. V. S.: Average case analysis of region search in balanced k-d trees. *Information Processing Letters 8*, 5 (1979).

[FKN80] FUCHS H., KEDEM Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. In *Computer Graphics (Proceedings of SIGGRAPH)* (1980), pp. 124–133.

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a GPU raytracer. In *Graphics Hardware* (2005), pp. 15–22.

[GG07] GUENNEBAUD G., GROSS M.: Algebraic point set surfaces. *ACM Transactions on Graphics 26*, 3 (2007), 23:1–23:9.

[GGG08] GUENNEBAUD G., GERMANN M., GROSS M.: Dynamic sampling and rendering of algebraic point set surfaces. *Computer Graphics Forum 27*, 2 (2008), 653–662.

[GP07] GROSS M., PFISTER H.: *Point-Based Graphics*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers, 2007.

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Symposium on Interactive Ray Tracing* (2007), pp. 113–118.

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. In *Computer Graphics (Proceedings of SIGGRAPH)* (1992), pp. 71–78.

[IMRV97] INDYK P., MOTWANI R., RAGHAVAN P., VEMPALA S.: Locality-preserving hashing in multidimensional spaces. In *Symposium on Theory of Computing* (1997), pp. 618–625.

[Int07] INTEL: Intel QuickAssist technology accelerator abstraction layer white paper. In *Platform-level Services for Accelerators Intel Whitepaper* (2007).

[JDD03] JONES T. R., DURAND F., DESBRUN M.: Non-iterative, feature-preserving mesh smoothing. *ACM Transactions on Graphics 22*, 3 (2003), 943–949.

[Lev01] LEVIN D.: Mesh-independent surface interpolation. In *Advances in Computational Mathematics* (2001), pp. 37–49.

[MM02] MA V. C. H., MCCOOL M. D.: Low latency photon mapping using block hashing. In *Graphics Hardware* (2002), pp. 89–99.

[NHS84] NIEVERGELT J., HINTERBERGER H., SEVCIK K. C.: The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems 9*, 1 (1984), 38–71.

[OvL80] OVERMARS M., VAN LEEUWEN J.: *Dynamic multi-dimensional data structures based on quad- and k-d trees*. Tech. Rep. RUU-CS-80-02, Institute of Information and Computing Sciences, Utrecht University, 1980.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Graphics Hardware* (2003), pp. 41–50.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (2007), 415–424.

[Rob81] ROBINSON J. T.: The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *International Conference on Management of Data* (1981), pp. 10–18.

[Sam06] SAMET H.: Multidimensional point data. In *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006, pp. 1–190.

[Tan06] TANURHAN Y.: Processors and FPGAs quo vadis. *IEEE Computer Magazine 39*, 11 (2006), 106–108.

[Vah07] VAHID F.: It's time to stop calling circuits hardware. *IEEE Computer Magazine 40*, 9 (2007), 106–108.

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164.

[WHA*07] WEYRICH T., HEINZLE S., AILA T., FASNACHT D., OETIKER S., BOTSCH M., FLAIG C., MALL S., ROHRER K., FELBER N., KAESLIN H., GROSS M.: A hardware architecture for surface splatting. *ACM Transactions on Graphics 26*, 3 (2007), 90:1–90:11.

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Graphics Hardware* (2006), pp. 67–77.

[WPK*04] WEYRICH T., PAULY M., KEISER R., HEINZLE S., SCANDELLA S., GROSS M.: Post-processing of scanned 3D surface data. In *Symposium on Point-Based Graphics* (2004), pp. 85–94.

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), 434–444.

[ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3D: An interactive system for point-based surface editing. *ACM Transactions on Graphics 21*, 3 (2002), 322–329.