FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Fast KNN in Screenspace on GPGPU

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Dominik Schörkhuber
Registration Number 1027470

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Mag. rer. soc. oec. PhD Stefan Ohrhallinger

Vienna, 03.04.2016

Dominik Schörkhuber

Stefan Ohrhallinger

Erklärung zur Verfassung der Arbeit

Dominik Schörkhuber
Mühlbachstrasse 21, 4451 Garsten

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Kurzfassung

Durch das Voranschreiten der 3d-scanning Technologien in den letzten Jahren wurde das Digitalisieren von Objekten sehr populär und leistbar. Heute ist es nicht nur möglich statische Objekte, sondern auch bewegte 3-dimensionale Szenen aufzunehmen. Das Rendern solcher Aufnahmen ist schwierig, da die Objekte als Punktwolken aufgenommen werden, und erst durch Vorverarbeitungsschritte in Polygonmodelle umgewandelt werden. Ein wichtiger Teil der Vorverarbeitung ist es für jeden Punkt der Punktwolke die k nächsten Nachbarn zu finden, um die Oberfläche lokal rekonstruieren zu können. In dieser Arbeit wird eine Methode präsentiert welche es erlaubt die k nächsten Nachbarn für Punktwolken in Echtzeit zu bestimmen. Der gesamte Algorithmus ist parallelisiert implementiert, und basiert auf dem Nvidia CUDA parallel computation framework. Des Weiteren arbeiten große Teile das Verfahrens ausschließlich im Bildbereich. Dadurch wird gewährleistet dass redundante und verdeckte Punkte frühzeitig erkannt und eliminiert werden. In einer beispielhaften Anwendung wird der FastKnn Algorithmus außerdem verwendet um Punkwolken in Echtzeit zu rekonstruieren, und durch Splatting zu rendern.

Abstract

Virtualization of realworld objects and scenes became very popular in recent years due to affordable laser-scanning technology. Nowadays it's not only possible to capture static frames but also realtime frame sequences. Rendering of those captures is difficult because visually appealing renderings involve the computation of local surface reconstruction from pointclouds and therefore a lot of preprocessing. This is usually not possible in realtime.

One important processing step is the computation of nearest neighbours for each 3d-point. The neighbourhood information is not only used for normal reconstruction and local surface estimation, but can also be utilized for collision detection.

In this paper we present a method for computing the k-nearest-neighbor sets for pointclouds in realtime. To achieve high frame rates we parallelize the algorithm on the GPU, using the Nvidia CUDA parallel computation framework. Furthermore computations are limited to operate in screen-space, to reduce computational complexity even further, and effectively prevent rendering invisible geometry. We also utilize the invented FastKnn algorithm to estimate local surface reconstruction for splat rendering of pointclouds in realtime and show how it compares to a state of the art algorithm.

Contents

List of Figures **viii**

1 Introduction **1**
1.1 Problem . . . 2
1.2 Method . . . 3

2 State of the art **5**

3 Methodology **7**
3.1 Overview . . . 7
3.2 Screenspace Transformation . . . 8
3.3 Quadtree construction . . . 8
3.4 K-Radius estimation . . . 10
3.5 Candidate search . . . 10
3.6 Final sorting . . . 12

4 Implementation **13**
4.1 Overview . . . 13
4.2 Projection to Screenspace . . . 13
4.3 Quadtree Layout and Construction . . . 14
4.4 k-Radius Estimation . . . 16
4.5 Quadtree Traversal . . . 16
4.6 Iterative Radius Optimization . . . 17
4.7 Sorting and Filtering . . . 17

5 Analysis **21**
5.1 Analysis . . . 21
5.2 Conclusion . . . 28

Appendix **29**
Appendix A: Tables for Figures . . . 29

List of Figures

1.1 Example for k nearest neighbours search. Blue points describe the point set P, the green point is an element of the query set Q. In this case k was set to 3 2

3.1 Each point represents an occupied pixel in screenspace. Point numbers are accumulated in each quad-tree node. Starting with the lowest quad-tree node level covering only one pixel, the size of a node quadruples in each step. On its highest level the quad-tree has only one node, containing all points. (read from left to right) 9

3.2 This illustration continues the example from figure 3.1. On the bottom we see the expected packed quad-tree buffer. The number in each node describes its index in the packed buffer, which is equal to its z-iteration order. Indices in the lowest quad-tree level point to exactly one entry. For construction of the packed buffer we compute the index offsets from right (highest-level) to left (lowest level). 9

3.3 A point P with an estimated screenspace radius r_S which projects to an estimated view-space radius r_V. Only Q_3 is inside the k-radius-sphere and is therefore considered as *k-nearest-neighbour*. 11

4.1 For each tested model, the percentage of correctly sized *candidate-sets* after n iterations is shown. We observed that three iterations are enough to generate appropriately sized sets. 18

4.2 Computation times for increasing numbers of *k-nearest-neighbours*. For $k = 8$ a specialized register based implementation was used. It shows a 20% faster sorting step compared to the general solution. 19

5.1 Total computation time for one frame in FastKNN (right) compared to the rendering time of Autosplats (left). Whereas the time needed for projection and splatting match the times of Autosplats, FastKNN computes the k-nearest-neighbour sets significantly faster. For all tests FastKNN is capped at 3 iterations. 22

5.2 The percentage of completely correct k-nearest-neighbour sets compared to an exact solution is shown after one, two, and three iterations. The colours indicate the absolute number of correct nearest neighbours for a given percentage. For comparison also the render times are shown on the bottom. 23

5.3 Scene transformations over time can reveal inconsistencies in the computed surface normals. The two images are taken at slightly different angles. Pay attention especially to the surface boundaries, where normal estimation errors tend to appear. These errors are not consistent over different transformations, and might be perceived as visual flickering artefacts. 25
5.4 Comparision of normal quality between Autosplats (left), and FastKNN (right). Dark blue areas show no error, green areas indicate an error of up to 22.5 degrees, and red areas show errors above that. 26
5.5 Figure 5.4 continued. Autosplats above, FastKNN below. FastKNN produces a similar visual quality compared to Autosplats. Especially for the dragon model, Autosplats generates many errors along the boundary, whereas FastKNN remains stable. 27

CHAPTER 1

Introduction

In recent years the development of 3d-scanning technologies made huge steps forward. While these technologies are now widely available it is still not trivial to render scanned data. Algorithms in the field of computer graphics usually work with polygonal meshes as basic geometry input. This has been done for decades and todays graphics hardware is highly optimized towards polygonal mesh data structures. However 3d-scanners do not output polygonal data.

The output of a 3d-scanner is usually a set of points in $\mathbb{R}^3$, remodelling in it's entirety the surface of the scanned object. We refer to these sets of points as point-clouds. To utilize current graphics hardware for rendering of objects or scenes represented as point-clouds, an often applied strategy is to first reconstruct the objects surface to a polygonal mesh and then render the extracted mesh. Speaking of very large data-sets in general, the surface reconstruction is computationally very expensive. Further it is difficult to map between a point-cloud and a polygonal mesh representation. For particular applications this approach is not suitable.

As first example think of continuously streamed point-cloud videos [1]. In that case the reconstruction approach does not scale very well. The 3d-information needs to be recovered for each frame in a lengthy preprocessing step. Another example are editing operations on point-cloud data. While editing point-cloud data in it's raw form is of course possible. The editor can benefit from an instantly computed closed surface representation.

Therefore we choose a more direct rendering approach. Splatting needs less preprocessing to render a closed surface representation from a point-cloud than a polygonal mesh. Instead of rendering triangles we estimate the objects surface by rendering ellipses over each point in the dataset. For each ellipse we have to estimate its orientation and extent. For a local surface reconstruction the parameter estimation for each ellipse are influenced by the nearest surrounding points. Yet it turns out that finding the k-nearest points for each point in the dataset is a very difficult task. It even takes the by far biggest chunk of time while splat-rendering objects. By the above stated reasons this thesis shows an approach to estimate the *k-nearest-neighbour* set for each point. For means of demonstration this thesis also shows how to utilize the *k-nearest-neighbour* estimation for a splat rendering application.

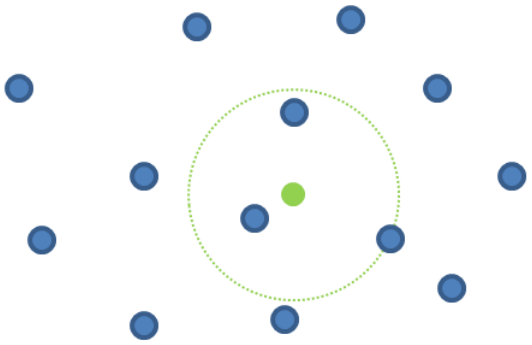

Figure 1.1: Example for k nearest neighbours search. Blue points describe the point set P, the green point is an element of the query set Q. In this case k was set to 3

1.1 Problem

Given a 3d-point-cloud we want to estimate its surface. To locally reconstruct the surface we compute the *k-nearest-neighbours* for each point in the point-cloud. Let us denote a d-dimensional metric space D and let $P = \{p_1, p_2, ..., p_n\}$ be a set of points in this d-dimensional space D. Also let $Q = \{q_1, q_2, ..., q_n\}$ be another set of query points in the same space. In order to solve the *k-nearest-neighbour* problem we assign to each query point $q \in Q$ the set of its k-nearest points in P. Figure 1.1 shows an example for *k-nearest-neighbour* search in a 2-dimensional space. For each point $k = 3$ nearest neighbours are computed and an euclidean metric is used to calculate distances.

The most trivial way to algorithmically solve the problem is by a brute force approach. Listing 1 shows a simple method to do so. But we also see that this algorithm has a huge computational complexity: $O(|P||Q|d)$. For reasonably sized point-clouds *k-nearest-neighbour* computation in real-time is not possible by this approach.

Algorithm 1 Knn Bruteforce Algorithm

```
1:  procedure COMPUTE KNN(number of nearest neighbours k, sets P and Q)
2:      for q ∈ Q do
3:          for p ∈ P do
4:              d(p) = distance(p, q)        ▷ compute a distance metric between p and q
5:          end for
6:          sorted = sortAscending(P, d)
7:          knn(q) = sorted.take(k);                          ▷ take the first k points
8:      end for
9:  return knn                                  ▷ k nearest points for each q ∈ Q
10: end procedure
```

1.2 Method

Given an unstructured set of points in 3d-space our method computes the *k-nearest-neighbours* for each point in the set. Since we want the computations to happen without any preliminary available information, except for the point positions, we have to compute the Knn for each rendered frame. If we assume a very complex scene with many parts that do not fit on the screen, or are just hidden, unnecessary work would be done. To avoid that we adopt the idea from Preiner et. al. [8] to first project the point-set into screenspace. By doing that we remove all non-visible points in the scene. Also points that would project to the same pixel position will be rejected. With common a Z-Buffering technique it is ensured that we only keep the foremost point. Be advised that this also implies that we are not computing the exact *k-nearest-neighbours*. Pruning the point-set to the screenspace projection might remove some exact nearest neighbours. Therefore the algorithm only produces an approximate solution to the *k-nearest-neighbour* problem. We then build a compressed linear quad-tree structure with the remaining points. Additionally several helping structures are generated for fast indexing. The basic structure is now built up in memory, and we are ready to issue Knn-queries for each pixel on the screen that has been covered by a projected point.

To estimate how many neighbourhood candidates we have to consider, we choose the number of points in the quad-tree nodes. We take an initial estimate by observing a leaf node in the tree, and then traverse up the quad-tree hierarchy until we reach a node containing at least k points. By this estimation we compute an initial search radius around the observed point in screenspace. Through several iterations we refine that search radius, until we receive a *k-nearest-neighbour* candidate set. For a valid solution this set contains greater or equal k points. The points in that now very small set are again considered as candidates for the *k-nearest-neighbours* of the point in question. The candidate-set gets then sorted to obtain k points with smallest distance. We have now retrieved the set of *k-nearest-neighbours* for the observed point.

Conclusively we can state several interesting benefits for our method. First of all our method is not dependent on any preliminary computed information. Only the point-clouds itself gets fed into the algorithm. This makes the algorithm attractive for dynamically changing data. Secondly the scenes complexity has only little effect on the speed of our algorithm. Since our method is working mainly in screenspace, the needed computational effort is well constrained by the screen resolution. Last but not least the chosen data structure is very well suited for parallelization on the GPU, and therefore grants a great speed-up in comparison to similar solutions.

CHAPTER 2

State of the art

The k-nearest-neighbour (Knn) method is widely used for classification [3]. The basic idea is very simple. Given two sets of points containing a set of training and a set of test data points. We then select one test data point, and compute the distance between the data point itself and all points contained in the training dataset. After obtaining the distances we sort the training points by their distance, to find the k-nearest data points. This procedure is repeated for each point in the test set. That method works well for small sets of points, but is not well suited for huge point sets as we encounter them in the fields of pattern recognition or computer graphics. One attempt to deal with that vast amount of data was to utilize spatial data structures. Due to it's simplicity the Knn algorithm happens to be very suitable for parallelization. Today GPGPU frameworks like OpenCL and Cuda enable even consumer level hardware to solve tremendously expensive tasks quickly [5].

Garcia et. al. compared different approaches on Nvidia GPUs [4]. Assuming that a brute-force approach might be suitable for the GPU, they found that also GPU solutions greatly benefit from the use of spatial data structures. Nikam and Meshram did likewise with a brute-force algorithm on OpenCL architectures [7].

Zhou et. al. demonstrate the use of SAH k/d-trees on the GPU, by rendering static scenes with Raytracing and Photonmapping in real-time [10]. Connor and Kumer state that in general k/d-trees are the best suited data structure for the Knn problem. However for well constrained problems a quad-tree data structure is better suited. Their probably most important optimization was to store the quad-tree in Z-order. By that the data itself implicitly contains information about its spatial relations [2].

The by far most important preliminary work for this thesis has been done by Preiner et. al. on Autosplats [8]. Autosplats is a point-cloud visualization pipeline, based on splatting. An important step was to abandon all sorts of pre-computation. Each frame is drawn without any preliminary knowledge. To simplify the Knn-search and to be able to handle also complex scenes all computations are designed to happen in screen-space. The implementation of Autosplats heavily utilizes the GPU. In contrast to other solutions of the Knn problem they did not use any GPGPU framework, but did all computations in OpenGL Shaders.

CHAPTER 3

Methodology

3.1 Overview

We start with a basic overview of the algorithm which consists of five steps. The input of our algorithm is an unstructured pointcloud. The pointcloud data contains only the point positions in world-space coordinates.

- Transform to screenspace
  In the first step we transform all points into screenspace. By doing that we reduce the workload of the main algorithm to visible parts of the pointcloud. Points outside get clipped against the view frustum and points projecting to the same pixel get removed in this step.
- Quad-tree construction
  For the second step we generate a compressed linear quad-tree structure from the remaining points in screenspace. The quad-tree introduces the needed spatial relation into the former unstructured data.
- K-radius estimation
  For the now structured data we perform a radius estimation on each point. This gives us an approximate search radius in which we hope to find the *k-nearest-neighbours*.
- Candidate search
  By iterating over all points in a given radius we determine the actual number of *k-nearest-neighbour* candidates, which results in a set of cardinality greater or equal than k. After several estimation steps we retrieve the final set of candidates, along with the distances to its *k-nearest-neighbour*.
- Sorting
  Finally we sort the before acquired points in the *k-nearest-neighbour* candidate set to pick the *k-nearest-neighbours* with smallest distance.

3.2 Screenspace Transformation

In the first step all points are projected to screenspace. The pointcloud data might represent a very big scene, which possibly contains many parts that are not actually visible. If the scene does not even fit on the screen we would do a lot of unnecessary work. Therefore we do a simple culling of all input points against the view-frustum. The remaining points get rasterized onto the image plane. By that we limit the maximum amount of retained points to the pixel resolution of the render-target. From all points projected on the same pixel position we only keep the foremost point. We accomplish that by simple Z-Buffering. Points in the background simply get dropped. This is an important step to constrain the scenes complexity by the size of the view-port. While rasterizing the points also the original 3d-positions get stored in the render-target. Depending on variations of the algorithm, additional point attributes like a points color can be transferred too.

3.3 Quadtree construction

As pointed out in section 1.1 the *k-nearest-neighbour* problem is very computation heavy. To complete our problem in reasonable time we speed-up the Knn queries by spatially pre-sorting the input data. Similar to Connor et. al. [2] we utilize a quad-tree data-structure.

Currently the remaining points lie unordered on the render-target. To access the points efficiently in later steps of the algorithm, we attempt to pack the points tightly into a 1-dimensional array in Z-order [6] [2]. To do so the first step of the quad-tree construction is to compute the total number of points. We assume the screenspace texture to be of square size $2^s, s \in \mathbb{N}^+$. In a first step we recursively add up the number of points in each quad-tree child-node and add the number to their common parent-node. Figure 3.1 illustrates the procedure, which is completed in s sequential steps. The point accumulation in each step is fully done in parallel as explained below. The total possible number of points is obviously constrained by $(2^s)^2 = 2^{2s}$, however the number of points on the root quad-tree node gives us the actual number N of points to process.

As stated above, the final quad-tree buffer should contain all points packed tightly in Z-order. We already know the total size of the buffer from the accumulation step, but to pack the quad-tree buffer in parallel we also need to know the position index of each quadtree-node in the final packed buffer. We can compute those indices from the already acquired numbers of points in each node.

For illustration of the problem again consider Figure 3.1. We again do the computation of indices in s steps, starting off with the root-node. By definition the root-node spans over all points in the tree. The root-nodes points start at index 0, and contains N points. We then proceed with computing indices recursively for all other nodes.

Lets assume we have a parent-node p, and we want to compute the starting indices for the four child-nodes $c_{0..3}$ of the parent-node p. Further $N(node)$ describes the number of points in a quad-tree node, and $Start(node)$ describes the starting index of a node in the quad-tree buffer. The set of points contained in $c_{0..3}$ is the same set of points as contained in the node p. By assuming Z-order iteration, we know that $Start(c_0) = Start(p)$. The second child, c_1, is offset by the number of points in the first child c_0. Which means $Start(c_1) = Start(c_0) + N(c_0)$. Further the starting index of c_2 and c_3 can be computed by accumulating the offsets. Where

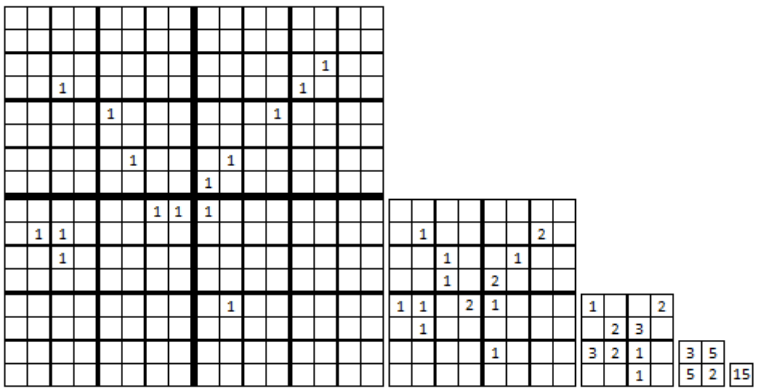

Figure 3.1: Each point represents an occupied pixel in screenspace. Point numbers are accumulated in each quad-tree node. Starting with the lowest quad-tree node level covering only one pixel, the size of a node quadruples in each step. On its highest level the quad-tree has only one node, containing all points. (read from left to right)

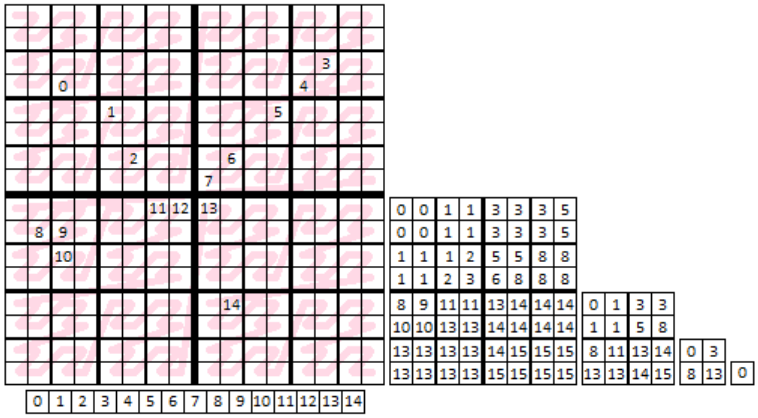

Figure 3.2: This illustration continues the example from figure 3.1. On the bottom we see the expected packed quad-tree buffer. The number in each node describes its index in the packed buffer, which is equal to its z-iteration order. Indices in the lowest quad-tree level point to exactly one entry. For construction of the packed buffer we compute the index offsets from right (highest-level) to left (lowest level).

$Start(c_2) = Start(c_1) + N(c_1)$ and $Start(c_3) = Start(c_2) + N(c_2)$ respectively. Again this computation can be done in parallel for each node on the same level. For an example of the index calculation derived from figure 3.1 please see figure 3.2.

The final step in the quad-tree construction is to copy the points from the screenspace texture to the packed quad-tree buffer. We simply read the target index for each point from the index-

buffer and copy the point to the packed quad-tree buffer.

3.4 K-Radius estimation

Now that the quad-tree structure has been built up, we estimate a search radius for the *k-nearest-neighbours*. All points within the estimated K-Radius will be considered to be in the *k-nearest-neighbour* set. Our estimation is roughly based on the number of points in the quad-tree nodes. Given a point in the quad-tree we want to get a radius estimation for, we traverse the quad-tree from the target point (leaf node) on its path to the root node. On that path we stop at the first node holding at least k_{min} points. Where k_{min} is the number of *k-nearest-neighbours* we are looking for. Since we will further refine the candidate set later on, we can also accept more than k_{min} candidates. We can collect up to an implementation dependent number of k_{max} points.

We refer to the estimated radius as the *k-radius-estimation*. Since this is only an estimation we might not hit the actual *k-radius*, but it enables us to iteratively approach the final solution. Because the quad-tree nodes cover a square region we are going to map the given point density to the search radius. Let A be the covered area of the determined quad-tree node, and r be the *k-radius-estimation*. The radius in screenspace would be computed as $r = \sqrt{\frac{A}{\pi}}$. Since we actually need only k_{min} points and the nodes covered area is usually not fully covered by points, we scale the node-area radius appropriately. Let further N be the number of points in the given node. Then the *k-radius-estimation* is computed by the following equation.

$$\frac{k_{min}}{N} \cdot A = r^2\pi \rightarrow r = \sqrt{\frac{A}{\pi} \cdot \frac{k_{min}}{N}}$$

3.5 Candidate search

Given the *k-radius-estimation*, we traverse the quad-tree nodes from the root-node down to the leafs. We leave out nodes that do not intersect the *k-radius-estimation* and further traverse nodes that are intersecting the radius. All resulting points are collected in the candidate set.

In previous steps we have reduced the initial 3-dimensional point-set to a 2-dimensional point-set. Although we have stored the world-space position of each point in the quad-tree structure, our quad-tree only considers the 2-dimensional coordinates for its spacial sorting. This leaves us with the fact that neighbouring points in the quad-tree are not necessarily near in world-space because their depth to the camera might differ a lot. Consider figure 3.3 for illustration of the described problem. By defining the *k-radius-estimation* we also define the according search sphere in view-space with radius r_V. On the one hand that means we have to compute the distances between points in world- or view-space. But even more importantly it means that we are going to reject many points in the candidate set, because they are simply too far away. Because of that, and because the *k-radius-estimation* is only approximately correct, we repeat the estimation and point-collection several times. In each step we adapt the *k-radius-estimation* to acquire the candidate-set with size between k_{min} and k_{max} points.

To adapt the *k-radius-estimation* we utilize the now known number of points V inside the search-sphere, described by r_V. We assume the points are equally distributed in screenspace.

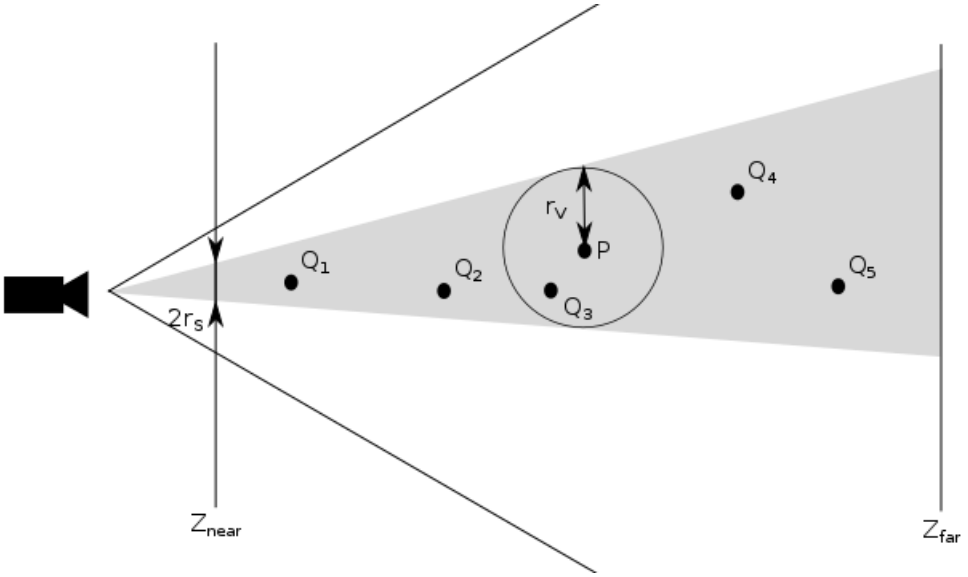

Figure 3.3: A point P with an estimated screenspace radius r_S which projects to an estimated view-space radius r_V. Only Q_3 is inside the k-radius-sphere and is therefore considered as *k-nearest-neighbour*.

Let r_s be the *k-radius-estimation* in screenspace, and let a be a multiplicative factor to adjust the *k-radius-estimation* for the next iteration, such that $r'_s = r_s \cdot a$. Assume we have a circular area A containing V valid points. To receive the new radius we alter that area proportionally to the number of expected points. Empirically we determined the number of expected points to be $2k_{min}$. The radius for the next iteration r'_s is then computed by the following equation.

$$\frac{2k_{min}}{V}A = (a \cdot r_s)^2\pi \rightarrow a = \sqrt{\frac{2k_{min} \cdot A}{V \cdot \pi}} \rightarrow r'_s = r_s \cdot \sqrt{\frac{2k_{min} \cdot A}{V \cdot \pi}}$$

For a good trade-off between runtime and correctness of the computed *k-nearest-neighbour* set we limit the number of iterations to approximate the exact radius. While the approximation is well suited for areas where the points are equally distributed; the algorithm might produce visual errors in regions where that is not the case. This is especially true for boundary regions. We can see some of the produced errors in Figure 5.3.

For our experiments we mostly used a maximum of 3 iterations. Figure 4.1 shows how that affects the *k-nearest-neighbour* sets.

If we terminate the iterative search for a point before a suitable radius and a suitable *candidate-set* were found respectively, the *candidate-set* is not optimal. If the radius estimation was too big, the *candidate-set* is capped at k_{max} points. If, on the other hand, the k-radius estimation was too small, the *candidate-set* contains less than k_{min} points.

3.6 Final sorting

After determining a set of candidate points along with its distances to the point in question, we finally determine the *k-nearest-neighbour* set. To do that we simply sort the points by their assigned distance, and pick a number of k_{min} points from the *candidate-set*. To retrieve the k_{min} closest points it is sufficient to partially sort the *candidate-set*. For this purpose we build a max-heap [9] with a maximum of k_{min} elements. Sequentially the biggest element in the heap is replaced by an element from the *candidate-set*, and the heap is updated. After all candidates have been inserted into the heap, we can retrieve the computed k_{min} nearest-neighbour points from it.

CHAPTER 4

Implementation

4.1 Overview

In this chapter we describe the implementation of the FastKnn algorithm with Nvidia Cuda. In the particular application FastKnn is utilized to provide input data for a simple splat-rendering algorithm. Besides Cuda, the demo application also uses the Opengl api for rendering. To share resources between the two apis we make heavy use of their interoperability capabilities.

FastKnn is split into multiple kernels, which are executed in serial and roughly correspond to the steps described in Section 3.1. First we upload the point cloud to device memory. The points are projected, culled and rasterized to a render target. After that we gather the remaining points and generate an indexing structure for the quad-tree by several Cuda kernels. When the indexing structure is done, another Cuda kernel packs the points tightly into a buffer in device memory. The main kernel of the algorithm then estimates the *k-nearest-neighbour* search radius, traverses the quad-tree to define a set of candidates, and finally sorts the best matches from the candidate-set into the knn-set. All used kernels run one thread per pixel on the render-target.

4.2 Projection to Screenspace

The point data is present in host memory, and is uploaded to device memory into an Opengl Vertexbuffer. Our test implementation does not dynamically change point cloud data, but is perfectly suitable for constantly renewed data that is uploaded. The first step in the algorithm is to project all input points to screenspace. An Opengl Shader for perspective projection renders the input points to floating point texture attachments in the frame-buffer. We store the transformed view-space position into a separate texture, and bind the Opengl Texture also to a Cuda texture reference, to make it available for access from Cuda kernels. Initially we followed the idea to implement the point projection also as a Cuda kernel. But the synchronization cost for an emulated Z-Buffer was too high. Because of that we use Opengl shaders for the screenspace projection step.

4.3 Quadtree Layout and Construction

To be able to access neighbouring points in screenspace very fast, we sort the projected points into a Quadtree data structure. For caching reasons it is beneficial to have spatially close points packed densely in memory. Please consider Figure 3.2 again, which illustrates how we align the points in memory.

To be able to maintain a pointer-less quad-tree data structure several memory buffers are used as outlined as below. Depending on their use these buffers are allocated with the size of the number of nodes in a full quad-tree q or the number of pixels in the view-port of size l. A view-port of $l = 4 \cdot 4$ pixels would therefore result in a buffer size of $q = \frac{4 \cdot l - 1}{3} = \frac{4 \cdot 4^2 - 1}{3} = 21$ nodes.

- NumBuffer[q] : uint32
  Stores for each quad-tree node the number of points it contains. It corresponds to the illustration in Figure 3.1. The first l points represent the point numbers of leafs in the quad-tree. Points from index l to $l + \frac{l}{4}$ represent the point numbers of one level higher into the tree, and so on.

- IndexBuffer[q] : uint32
  Contains the starting memory index of the points for a node in the PackedQuadtree buffer. See Figure 3.2 for an illustration. The IndexBuffer follows the same indexing scheme as the NumBuffer.

- PackedQuadtree[l] : 3xfloat32
  Stores the view-space positions of occupied pixels packed densely in Z-Order.

- PackedQuadtreeCoordinates[l] : uint32
  Contains the (x, y) pixel coordinate of a point in the PackedQuadtree buffer.

The first construction step is to fill the NumBuffer. The buffer is initialized with zeros. A Cuda kernel fills the leaf-level of the NumBuffer with ones where a pixel is present. The information from the leaf-level is then propagated through one kernel run per quad-tree level for all other quad-tree levels. Each thread in a kernel run sums up the point numbers of the four sub-nodes and sets the point number of one quad-tree node.

Algorithm 2 Compute number of points in each quad-tree node

```
1:  procedure PROPAGATENUMS
2:      Mark occupied pixels in the NumBuffer with 1's
3:      for all Quadtree levels from leaf level+1 to root level do
4:          for all nodes in level (in parallel) do
5:              sum = 0
6:              sum += NumBuffer(child0)
7:              sum += NumBuffer(child1)
8:              sum += NumBuffer(child2)
9:              sum += NumBuffer(child3)
10:             NumBuffer(node) = sum
11:         end for
12:     end for
13: end procedure
```

Similar we run again one kernel for each quad-tree level which computes the starting memory index of each node, and store it in the IndexBuffer. Since the PackedQuadtree buffer is arranged in Z-Order, also the IndexBuffer follows that scheme. The upper left child-node is the first node in the said iteration order. Therefore the starting address of the first sub-node is equal to the starting address of the node itself. The starting address of the second sub-node is computed by adding the number of nodes in the first sub-node to the starting address of the node itself. See Algorithm 4.3 for the full computation scheme.

Algorithm 3 Compute number of points in each quad-tree node

```
1:  procedure COMPUTEINDICES
2:      for all Quadtree levels from root level to leaf level+1 do
3:          for all nodes in level (in parallel) do
4:              IndexBuffer(child0) = IndexBuffer(node)
5:              IndexBuffer(child1) = IndexBuffer(child0) + NumBuffer(child0)
6:              IndexBuffer(child2) = IndexBuffer(child1) + NumBuffer(child1)
7:              IndexBuffer(child3) = IndexBuffer(child2) + NumBuffer(child2)
8:          end for
9:      end for
10: end procedure
```

Now that the NumBuffer and IndexBuffer are both filled with data, we are able to find the memory offset of a quad-tree node in the PackedQuadtree array. To complete the quad-tree construction we are going to copy the transformed points from the render-target to the PackedQuadtree buffer. This is possible because the leaf-level of the IndexBuffer corresponds to the render-target. For each pixel position in the render-target we look up the same position in the IndexBuffer to get the destination address. We also lookup the view-space position of that pixel and write it to that destination address. Additionally we also store the (x, y) coordinates

of a pixel in the PackedQuadtreeCoordinates buffer, to track the origin of a point in the packed buffer.

4.4 k-Radius Estimation

Computing the *k-radius-estimation* is the first step in determining the *k-nearest-neighbour* set for each point in screenspace. Section 3.4 already briefly mentioned the strategy to determine an estimated search radius. We are using the number of points contained in the quad-tree nodes as an estimate for the point density in a local region of the quad-tree. Starting at the leaf-node, which represents a pixel, we successively traverse through the parent caquad-tree nodes towards the root-node. Moving up the hierarchy we check at each node if we already reached the minimum number of needed points, to form a *k-nearest-neighbour* set.

Since we actually only need k_{min} points and the nodes covered area is usually not fully covered by points, we scale the node-area radius appropriately. Section 3.4 already shows in detail how to computethe radius.

4.5 Quadtree Traversal

Given the *k-radius-estimation* we now have to collect the points lying inside that estimated radius. The quad-tree data structure enables us to search through the data set quickly. We start the tree traversal at the root-node, and successively move through child-nodes towards the leaves. To know which nodes to traverse further, and which nodes we can safely reject, the *k-radius-estimation* is estimated by a bounding box. If the bounding box of a node intersects the *k-radius-estimation* bounding box, we follow that node, otherwise it is rejected. If the viewed node is a leaf-node, we compute its distance to the current query point, and store the distance and point index in an array in local memory with a maximum capacity of k_{max} elements. Since the target number of points while finding the *k-radius-estimation* was k_{min}, we hope to find a number of k_{min} to k_{max} points inside the search radius. This is of course not always the case, a solution to the problem is further discussed in Section 4.6.

Common quad-tree structures are mostly built of pointers referring from one node to its children. However, we have laid out our data-structure in a more beneficial way which lets one find the starting memory index of a node, and read out all of its contained nodes in sequence. As with a pointer-based data structure, it is simple to move from one node to its children. Recall the structure of the Index- and NumBuffer in Figure 3.2 and Figure 3.1 respectively. Given the (x, y) coordinates of any node, and its quad-tree level e, we are able to compute its index in the Index- and NumBuffer. For the root-node e is 0, and increases while moving towards the leaf-level. From the quad-tree level e we can determine the number of nodes in a quad-tree level $2^{2e} = 2 << e$. First we determine the index-offset i_{off} for the observed quad-tree level e. Section 4.3 explained already how to do that. The index itself is then computed by $i_{off} + x \cdot (1 << e) + y$.

The traversal steps itself are implemented in an iterative fashion, instead of being recursive. For this purpose a stack is allocated in local memory. This stack contains the level of a node, and also its (x, y) coordinates which refer to the IndexBuffer. Initially the root node is pushed onto

the stack. Each processed node is first popped off the stack. Then its four sub-nodes are tested against the bounding-box containing the estimated radius. If the bounding boxes intersect, the sub-node is pushed on the stack, otherwise it is ignored.

In practice we do not traverse to the leaf-level of the quad-tree, because this would be too computationally heavy. Instead we terminate the traversal at a specifically set level before reaching the leaf-level. In our implementation the minimal size of nodes being traversed is 8x8, so if a node of size 4x4 is hit, all of its points get processed directly. We simply iterate over all points inside that quad-tree node, which can be received from indexing the *PackedQuadtree* buffer as described before.

4.6 Iterative Radius Optimization

Traversing the quad-tree with a given radius estimate gives us a candidate-set of points, which is, at least approximately, a super set of the *k-nearest-neighbour* set. We expect a set of size k_{min} to k_{max} points. If we do not receive a set of the expected size, our estimated radius was erroneous. In that case the radius estimation is slightly adapted, and we repeat the quad-tree traversal process as a whole. A set of size $2 \cdot k_{min}$ is the target, because of that we are scaling the radius estimation by a factor of $\sqrt{\frac{2 \cdot k_{min}}{|candidateset|}}$.

This process is not guaranteed to converge in a specific number of iterations, but our experiments showed that a maximum of three iterations is already sufficient to have a *candidate-set* of the correct size for 99% of all point points. Figure 4.1 shows the percentage of points with a correctly sized *candidate-set* after n iterations.

Depending on if the radius estimation was too conservative or too generous, we have two cases of errors. Depending on the application of FastKnn, we can adjust the algorithm to tend to rather over- or underestimate the *candidate-set*. In our testing application the nearest neighbour points are used for local surface estimation, where a plane is fitted through the knn points to reconstruct a surface normal. If we get less points in the *candidate-set* we can still estimate a surface normal, but it might not be perfectly accurate. In the other case the *candidate-set* might have been too big. Since we preallocate the array to store the *candidate-set* in local memory, it is not possible to store more than k_{max} candidate points. The traversing algorithm then terminates early to save time. In that sense it is very application dependent if we can deal with that kind of error.

4.7 Sorting and Filtering

After collecting all the possible candidates, the *k-nearest-neighbour* candidate set contains (for most cases, as noted before) between k_{min} and k_{max} points. We want the *k-nearest-neighbour* set to be of size k_{min}. To obtain such a set we partially sort the candidates through a min-heap, depending on their squared distance in world space. The used distances have been already computed during the quad-tree traversal step, and reside with the *candidate-set* itself in local memory. For performance reasons the heapsort algorithm has been, just like the quad-tree traversal, implemented in a iterative way. This implementation of the sorting algorithm uses an array in

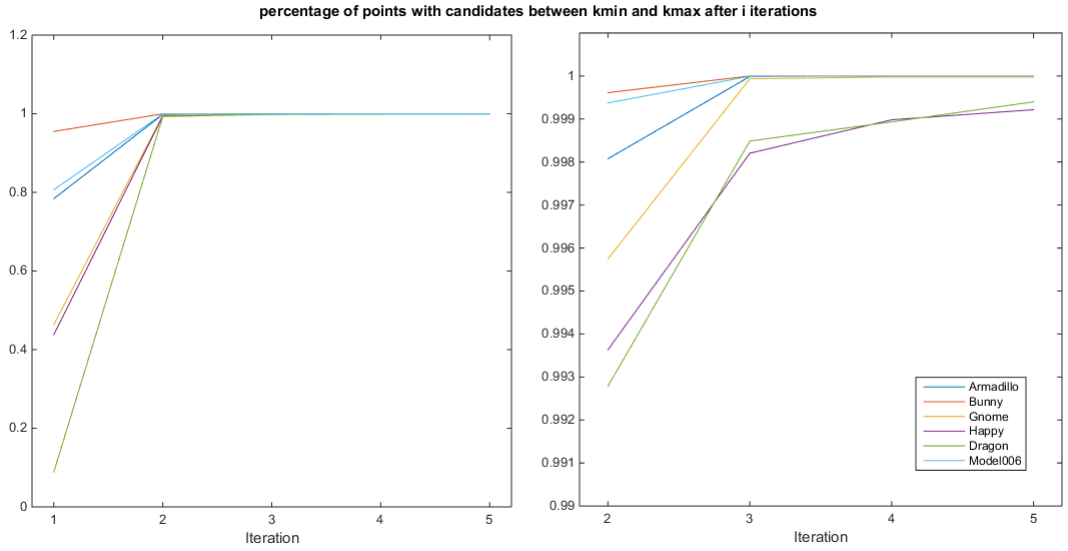

Figure 4.1: For each tested model, the percentage of correctly sized *candidate-sets* after n iterations is shown. We observed that three iterations are enough to generate appropriately sized sets.

local memory to represent the heap. To speed up the sorting even further, a specialized register-based sorting algorithm is used. Sadly Cuda does not allow to dynamically index register based arrays. Because of that it is not straight forward possible to implement the heapsort algorithm on registers only. For the sake of demonstration we implemented the register based heapsort only for a fixed heap size of 8 elements, and statically typed the tree comparisons in code. This is very inflexible, but provides a significant improvement in speed. Figure 4.2 shows the computation times of the sorting algorithm. Whereas for $k = 8$ the optimized version is depicted. The optimized version is about 20% faster than the version using a thread local array. Depending on where the results of FastKnn are needed, the output can be stored in local or global device memory. This might be a limitation for algorithms needing a dynamic range of different k values. However, if the parameter range for k is known a code generator would be able generate all the needed function implementations.

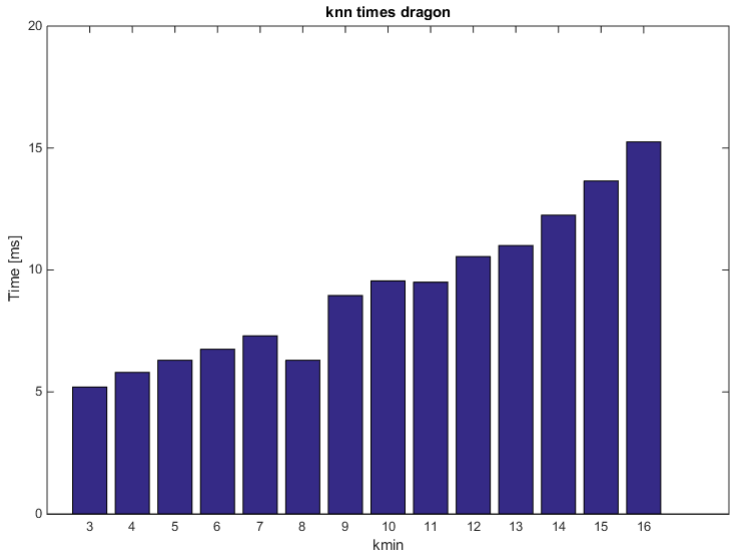

Figure 4.2: Computation times for increasing numbers of *k-nearest-neighbours*. For $k = 8$ a specialized register based implementation was used. It shows a 20% faster sorting step compared to the general solution.

CHAPTER 5

Analysis

5.1 Analysis

In this section we compare the FastKNN and Autosplats [8] algorithms in terms of speed and accuracy. To do a fair comparison, the Autosplats pipeline has been integrated into our rendering pipeline, and acts there as an alternative rendering path. This enables us to obtain detailed timings for all stages of the algorithms. We also utilize the Autosplats pipeline for the final point splatting. Since the FastKNN and the Autosplats algorithm, are only approximative solutions, we compare their results to an exact solution. This exact k-nearest-neighbour solution is computed by a kd-tree implementation on the CPU. All shown results have been computed on a Geforce GTX970, which was a mid- to high-end consumer graphics card in 2015.

Speed

In terms of speed FastKNN is able to compute the k-nearest-neighbour sets in about half of the time compared to the shader-based implementation of Autosplats. Dependent on the scene the projection step can take a significant amount of time, to project visible points, and eliminate invisible points. Through the interoperability between Cuda and Opengl resource, only little overhead is added to the overall time. Figure 5.1 shows the computation speeds for one frame for a variety of input pointclouds.

Accuracy

Since the resulting k-nearest-neighbour sets are only approximative solutions we analyzed the correctness of the computed sets, compared to an exact solution computed by a kd-tree computed on the CPU. The culling step at the beginning of the pipeline already removes many points. This helps to make the algorithm more efficient, but also creates problematic cases especially in boundary regions of a model, where the normal vectors are close to perpendicular to the image plane. In this comparison we compute the correctness of the knn-sets after they had

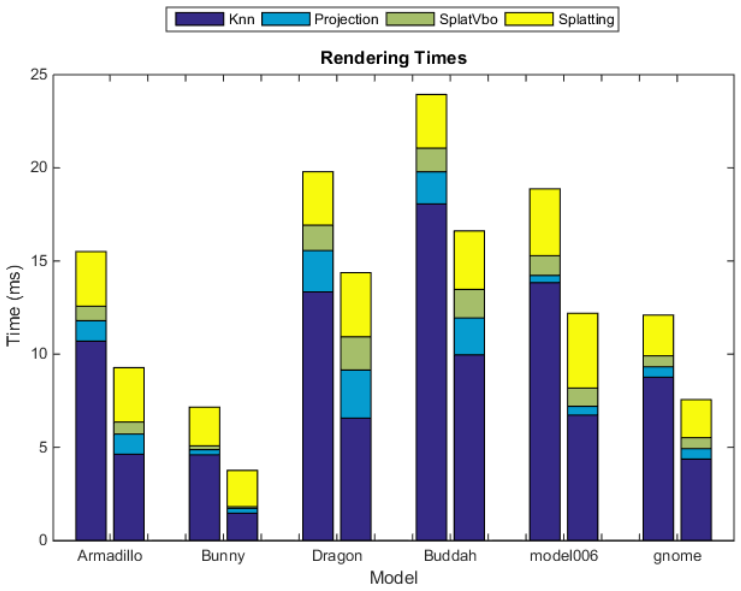

Figure 5.1: Total computation time for one frame in FastKNN (right) compared to the rendering time of Autosplats (left). Whereas the time needed for projection and splatting match the times of Autosplats, FastKNN computes the k-nearest-neighbour sets significantly faster. For all tests FastKNN is capped at 3 iterations.

been projected to screen. This removes the errors produced by the projection, but enables us to measure the errors produced by the search radius estimation and point gathering steps. Figure 5.2 shows the number of correct points after a number of $k = 8$ iterations. Figure 5.4 and 5.5 also show a visual comparison of normal qualities for Autosplats and FastKNN compared to an exact solution. To visualize the normal error, false-color rendered splat ellipses as follows: Dark blue areas show no error, green areas indicate an error of up to 22.5 degrees, and red areas show errors above that. Especially high-frequency surface regions, and regions with normals close to being perpendicular to the image plane appear to be problematic. Table 4 shows mean and standard deviation for FastKNN and Autosplats for different models. We again observe very similar results in terms of quality, while FastKNN is always slightly ahead.

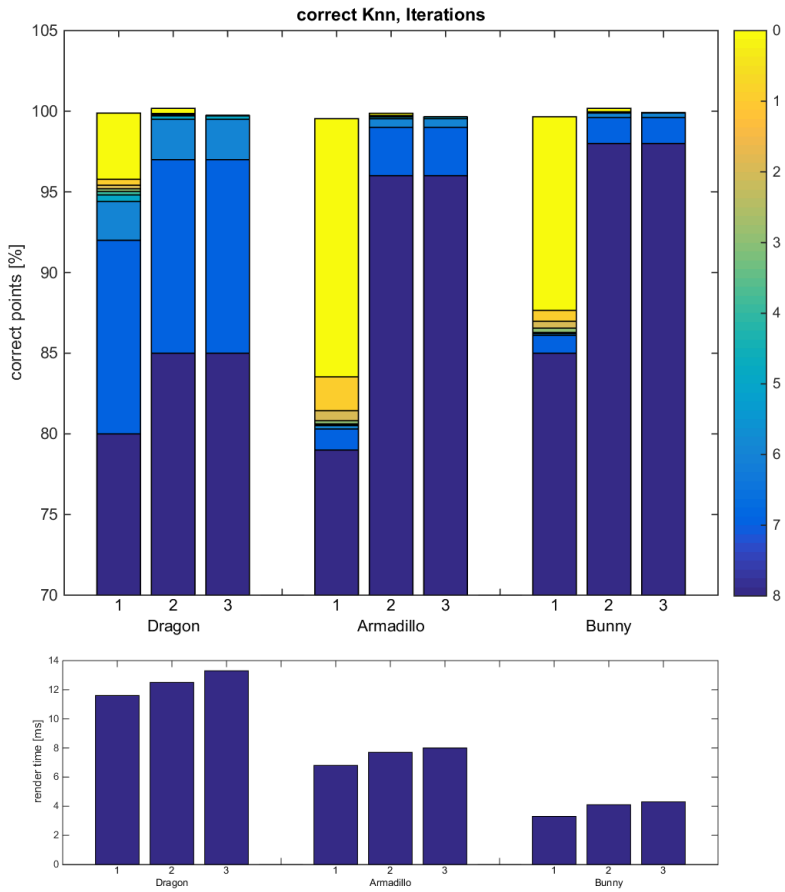

Figure 5.2: The percentage of completely correct k-nearest-neighbour sets compared to an exact solution is shown after one, two, and three iterations. The colours indicate the absolute number of correct nearest neighbours for a given percentage. For comparison also the render times are shown on the bottom.

Normal consistency

Especially for the case of surface reconstruction, we have observed the problem of surface consistency. Relative to the model, the estimated normal of the surface should be always identical. FastKNN computes the k-nearest-neighbour set only approximatively. This means that the k-nearest-neighbour set can slightly differ depending on the viewing angle and distance of the camera. While the shown normal estimation errors are usually low, and do not harm the visual quality much, scene transformations over time affect the temporal coherence. While transforming the scene the k-nearest-neighbour sets for points can change due to the approximative computation, and further produce a different normal vector estimation. While static errors for surface normals are not easily seen, change of normals over time tend to be much more visible. Figure 5.3 shows the differences in two slightly transformed frames.

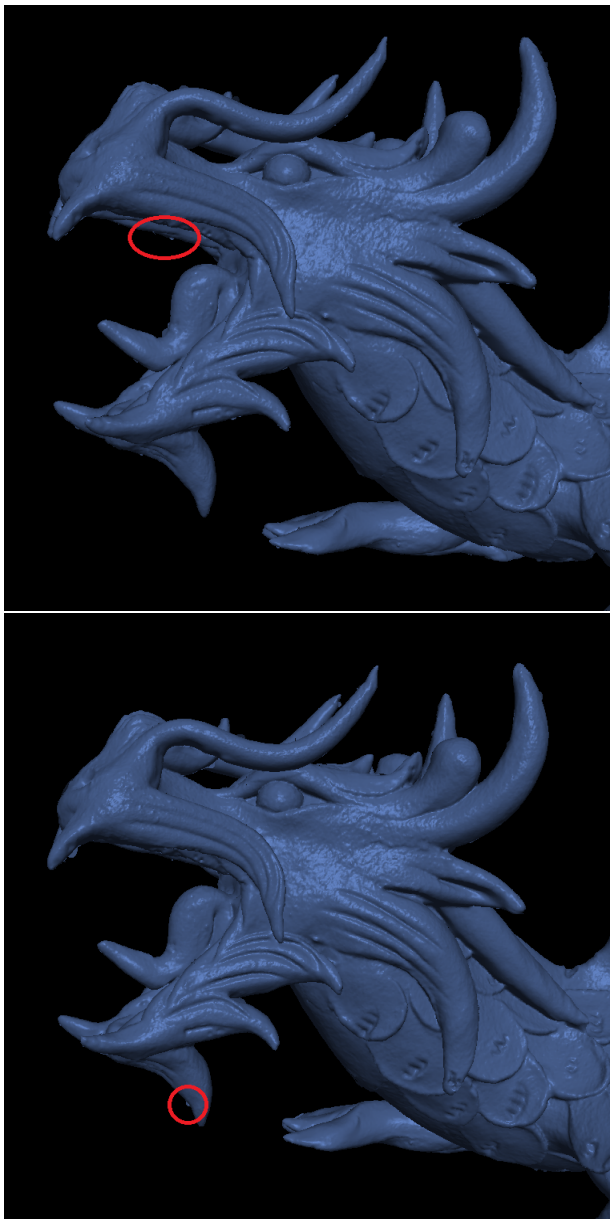

Figure 5.3: Scene transformations over time can reveal inconsistencies in the computed surface normals. The two images are taken at slightly different angles. Pay attention especially to the surface boundaries, where normal estimation errors tend to appear. These errors are not consistent over different transformations, and might be perceived as visual flickering artefacts.

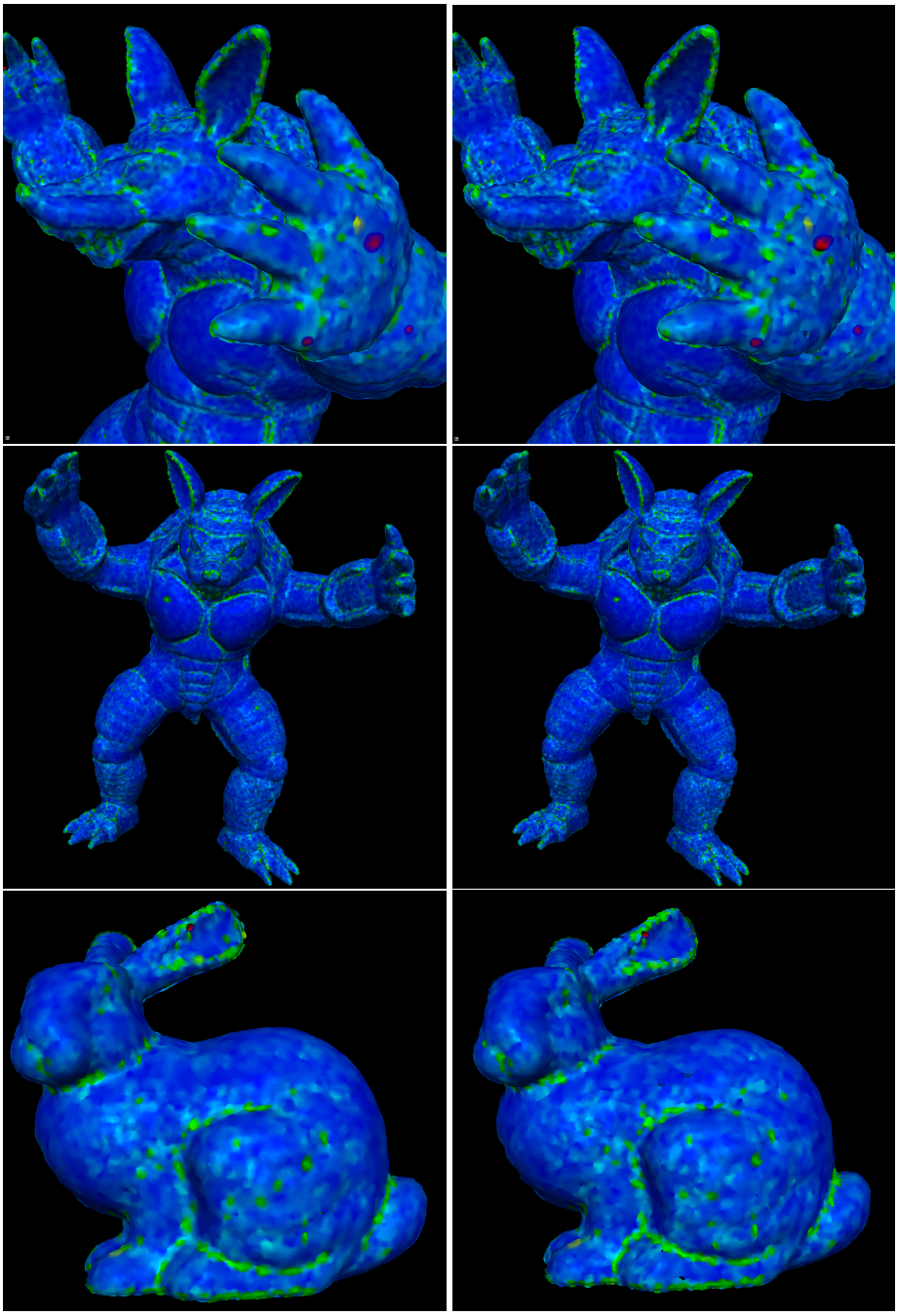

Figure 5.4: Comparision of normal quality between Autosplats (left), and FastKNN (right). Dark blue areas show no error, green areas indicate an error of up to 22.5 degrees, and red areas show errors above that.

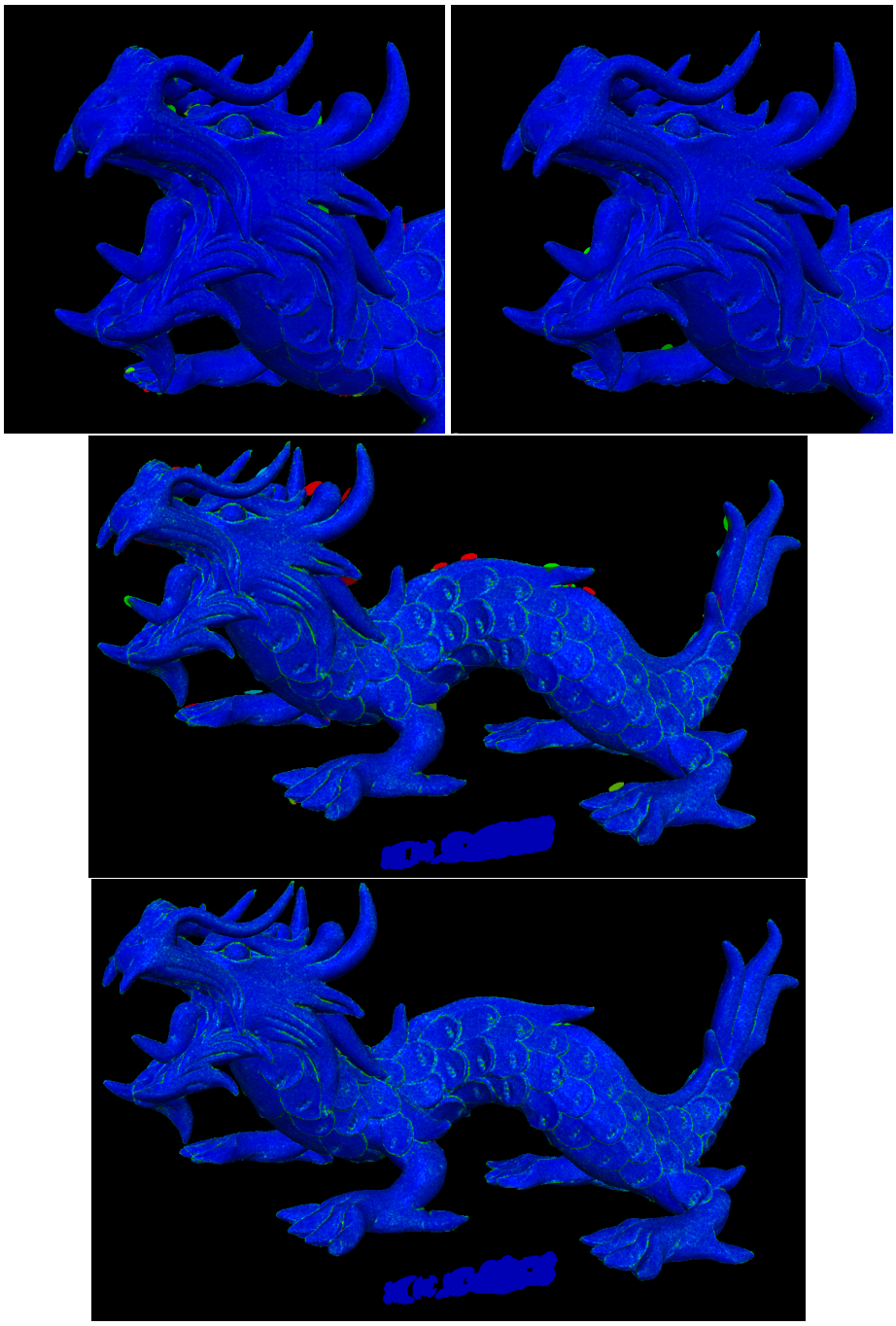

Figure 5.5: Figure 5.4 continued. Autosplats above, FastKNN below. FastKNN produces a similar visual quality compared to Autosplats. Especially for the dragon model, Autosplats generates many errors along the boundary, whereas FastKNN remains stable.

5.2 Conclusion

This paper showed how the GPU can be utilized to parallelize the *k-nearest-neighbour* problem for general unstructured pointclouds. While a very similar approach has been taken by Autosplats before, our mostly GPGPU based solution shows great improvements in terms of computation speed on consumer graphics hardware. Especially laser-scanners produce massive amounts of data which are usually difficult to render in real-time. FastKNN enables processing of such datasets without further preprocessing. However, the algorithm is limited by the scene size in the sense that all points need to be projected initially into screen-space. By this we eliminate unseen points why would project to the same pixel position. Future work could mitigate this problem by working with multiple layers of depth buffers to store the front most n pixels.

The FastKNN approach solves the *k-nearest-neighbour* only approximative. Depending on the application for FastKNN this might be suitable or not. In our implementation of FastKNN we have shown that surface estimation for splatting is a possible application of the algorithm. While the surface representations are not exact, the rendering speed is very high. This could be utilized for interactive previews and editing operations in future.

Finally we can say FastKNN is a competitive algorithm in terms of quality, and outperforms previous algorithms in terms of speed.

Appendix

Appendix A: Tables for Figures

Algorithm	Model	Total [ms]	Knn [ms]	Projection [ms]	SplatVBO [ms]	Splatting [ms]
Autosplats	Armadillo	15.50	10.69	1.09	0.77	2.93
FastKNN		9.27	4.62	1.08	0.64	2.91
Autosplats	Bunny	7.15	4.59	0.27	0.20	2.07
FastKNN		3.76	1.45	0.25	0.09	1.94
Autosplats	Dragon	19.78	13.33	2.22	1.35	2.87
FastKNN		14.37	6.56	2.58	1.78	3.44
Autosplats	Buddah	23.93	18.06	1.72	1.26	2.88
FastKNN		16.61	9.95	1.98	1.52	3.13
Autosplats	Model006	18.87	13.83	0.38	1.06	3.58
FastKNN		12.19	6.72	0.46	0.97	4.01
Autosplats	Gnome	12.09	8.76	0.55	0.58	2.18
FastKNN		7.56	4.36	0.56	0.59	2.04

Table 1: Rendering times for different models. Data for Figure 5.1

k	Time [ms]
3	5.2
4	5.8
5	6.3
6	6.75
7	7.3
8	6.3
9	8.95
10	9.55
11	9.5
12	10.55
13	11
14	12.25
15	13.65
16	15.25

Table 2: Computation time for increasing *k-nearest-neighbours*. Data for Figure 4.2.

#	Dragon			Armadillo			Bunny		
	1	2	3	1	2	3	1	2	3
0	0.0410	0.0032	0.0001	0.1600	0.0014	0.0000	0.1200	0.0021	0.0000
1	0.0037	0.0006	1.4e-5	0.0210	0.0003	0.0000	0.0068	9.7e-5	0.0000
2	0.0022	0.0003	3.8e-5	0.0062	0.0002	0.0000	0.0042	0.0002	0.0000
3	0.0017	0.0002	3.4e-5	0.0020	0.0003	2.5e-5	0.0027	0.0002	0.0000
4	0.0021	0.0002	0.0002	0.0007	0.0002	0.0001	0.0007	9.7e-5	0.0000
5	0.0041	0.0021	0.0021	0.0004	0.0011	0.0011	9.7e-5	0.0003	0.0003
6	0.0240	0.0250	0.0250	0.0021	0.0053	0.0054	0.0011	0.0028	0.0029
7	0.1200	0.1200	0.1200	0.0130	0.0300	0.0300	0.0110	0.0160	0.0160
8	0.8000	0.8500	0.8500	0.7900	0.9600	0.9600	0.8500	0.9800	0.9800

Table 3: Fraction per correct number of nearest neighbours. Data for Figure 5.2

Algorithm	Model	Mean	Stdev
FastKNN	Armadillo	8.62	6.51
Autosplats		8.74	6.97
FastKNN	Bunny	9.86	8.67
Autosplats		9.81	8.76
FastKNN	Dragon	4.20	4.35
Autosplats		4.52	5.60
FastKNN	Buddah	6.43	7.86
Autosplats		6.72	9.19
FastKNN	Model006	1.22	2.94
Autosplats		6.28	17.57
FastKNN	Gnome	6.66	7.42
Autosplats		6.95	11.05

Table 4: Comparison of mean and standard deviation of the normal errors per pixel for different models.

Bibliography

[1] Alvaro Collet, Ming Chuang, Pat Sweeney, Don Gillett, Dennis Evseev, David Calabrese, Hugues Hoppe, Adam Kirk, and Steve Sullivan. High-quality streamable free-viewpoint video. *ACM Transactions on Graphics (TOG)*, 34(4):69, 2015.

[2] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *Visualization and Computer Graphics, IEEE Transactions on*, 16(4):599–608, 2010.

[3] Thomas M Cover and Peter E Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[4] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.

[5] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[6] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[7] VB Nikam and BB Meshram. Parallel knn on gpu architecture using opencl. *Int. J. Res. Eng. Technol*, 3:367–372, 2014.

[8] Michael Wimmer Reinhold Preiner, Stefan Jeschke. Auto Splats: Dynamic Point Cloud Visualization on the GPU. IEEE CS Press, 2012.

[9] John William Joseph Williams. Algorithm-232-heapsort, 1964.

[10] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)*, 27(5):126, 2008.