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

Siegfried Reinwald
Registration Number 1126981

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, 09.04.2019

Siegfried Reinwald

Stefan Ohrhallinger

Erklärung zur Verfassung der Arbeit

Siegfried Reinwald
Pantzergasse 21/29, 1190 Wien

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

3D-Scanning Technologien sind heutzutage omnipräsent und auch mit geringem Budget zu beschaffen. Es ist inzwischen sogar möglich, dynamische Szenen zu scannen. Die Visualisierung solcher Scans ist allerdings sehr aufwändig, da die Oberfläche aus der Punktwolke die vom 3D Scanner geliefert wird, rekonstruiert werden muss. Dieser Verarbeitungsschritt macht im Allgemeinen viel Gebrauch von der Vorverarbeitung dieser Daten und ist praktisch noch nicht in Echtzeit berechenbar.

Ein Teil dieser Vorverarbeitung ist das Finden der *k-nearest-neighbors*. Mit dieser Nachbarschaftsinformation können Oberflächen und ihre Normalen konstruiert werden. Diese Rekonstruktion kann dann auch für weitergehende Berechnungen wie zum Beispiel eine Kollisionsabfrage verwendet werden.

In dieser Arbeit sehen wir uns einen parallelen Zugang zu der *k-nearest-neighbor* Suche an und verwenden dazu NVidia Cuda um die Kandidatensuche auf die Grafikkarte auszulagern. Die Punktwolke wird in den Screen-Space projiziert, so können Punkte, die für den Betrachter nicht sichtbar sind, weggelassen werden. Dies führt zu einer Reduzierung der Komplexität der Berechnung. Um die Genauigkeit dieser Lösung zu steigern, verzichten wir allerdings auf einen Pro-Pixel-Tiefentest und werten alle Punkte die im Screen-Space sind aus, selbst wenn sie aus der gegebenen Perspektive von anderen Punkten überdeckt sind.

Abstract

Nowadays 3d scanning techniques are omnipresent and affordable. It is not only possible to scan static objects, but even dynamic scenes. The visualization of these scans is very compute-intensive because the surfaces of scanned objects need to be reconstructed from the point cloud data given by the scanning devices. This usually involves a lot of preprocessing and is basically not possible in real time.

One of these preprocessing steps is the computation of the nearest neighbors. With the neighborhood information, surface estimation, and normal reconstruction are possible. The reconstructed surface can even be used for further tasks like collision detection. [6]

In this paper, we propose a parallelized approach to the k-nearest-neighbor search using Nvidia CUDA on the GPU. The computational complexity is reduced by projecting the point cloud into screen space. Points which are not in screen space can be omitted because they will not be visible by the viewer. Although fast, this approach leads to the problem, that points which are projected to the same pixel are omitted and only the point nearest to the viewer is kept because of the per-pixel-depth-test. To avoid this we implemented a self-crafted projection algorithm, so our Knn algorithm is able to consider points which are covered by other points even when operating in screen space.

Contents

List of Figures

CHAPTER 1

Introduction

Nowadays 3d laser scanners and other 3d scanning techniques are omnipresent. They range from low-cost devices to huge professional installations. For example, the release of Microsoft's Kinect device for their Xbox 360 led to many households using and playing with 3d scanning technologies for a bit more than 100$. On the other hand, there are professional devices like Lidar scanners, to scan buildings, rock formations, and other features of terrains. These scanners can even be mounted on airplanes to scan whole tracts of land. What all these devices have in common, is that they output their data not in a classical way as polygonal meshes, but rather as a set of points in $\mathbb{R}^3$ which is called a point cloud.

Due to heavy optimization of modern GPUs to render polygon based meshes, many algorithms in computer graphics are based on polygons or triangles, but there are many ways to convert point based data into polygonal mesh data. For example, the Delaunay-Triangulation or Voxel-based approaches like the marching cube algorithm which are heavily in use by medical visualization techniques. Unfortunately, the reconstruction of a surface from point cloud data is a very computationally expensive task, so mapping between a point cloud and a polygonal mesh might be not suitable for some applications, especially when it comes to real-time processing. For example, if one thinks of streaming point cloud data to render a video, one has to pre-process the 3d data for each frame, which obviously is not going to scale well for really huge point clouds. Another thing is the modification of existing point clouds. Although this is already possible, the editor will benefit from an instantly computed closed surface representation. Other things which can be easily done with closed surfaces meshes, but is rather hard with point clouds, involve for example collision detection or deformations of surfaces.

Because of this, we decided to use a more direct volume rendering approach with splatting. By splatting, the computational complexity to render a closed surface representation is significantly reduced compared to creating the whole polygonal mesh. With this technique, we do not render triangles but ellipses over the points in the data set. The computational task is to find the right orientation and extent for these ellipses, which - in case of local surface reconstruction - can be done with the k-nearest neighbor search algorithm. But finding neighbors in a huge point cloud with millions of points is a very compute intensive task. This thesis shows an approach to

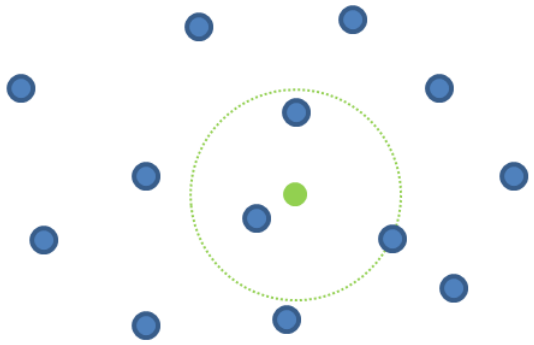

Figure 1.1: Example for k nearest neighbours search. Blue points describe the point set P, the green point is an element of the query set Q. In this case k was set to 3 [6]

estimate the *k-nearest-neighbors* for each point and shows how to utilize this data to splat render point clouds.

1.1 Motivation

We want to estimate the surface of a given 3D-point cloud, so we locally reconstruct the surface with *k-nearest-neighbor-search* for each point in the point cloud.

Given 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-neighbor* 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-neighbor* search in a 2-dimensional space. For each point $k = 3$ nearest neighbors are computed and an Euclidean metric is used to calculate distances. [6]

The naive approach, in this case, would be a brute-force approach which checks each point against all other points to find the valid neighbors. As one can easily see, in this approach the computational effort to solve the problem grows in an exponential manner with the number of points $O(|P||Q|d)$. Trying to reconstruct the surface of huge point clouds in real-time with this approach would be pointless. Listing 1 gives the pseudo code of such an algorithm.

Algorithm 1 Knn Bruteforce Algorithm [6]

procedure COMPUTE KNN(number of nearest neighbours k, sets P and Q)
  for $q \in Q$ **do**
    for $p \in P$ **do**
      $d(p) = distance(p, q)$ ▷ compute a distance metric between p and q
    end for
    $sorted = sortAscending(P, d)$
    $knn(q) = sorted.take(k)$; ▷ take the first k points
  end for
return knn ▷ k nearest points for each $q \in Q$
end procedure

1.2 Method

Given a point cloud in $\mathbb{R}^3$ the algorithm computes the *k-nearest-neighbors* for each point in the set. No preceding calculations should be needed, so we have to compute the *k-nearest-neighbors* in each frame. But when we look at how rendering works, we can see that in complex scenes many points may not even be on screen, so they will not be visible for the viewer anyway. To avoid rendering of unnecessary points we use the idea from Preiner et. al. [5] to first project the point set into screen space to only consider points which are mapped into screen space after projection. But one has to take caution because projecting on modern GPUs uses per-pixel-depth-testing, which means that after projection we only get the points nearest to the viewer and lose the points that are covered by them. To be able to use the underlying points we implement our own projection, which keeps these covered points, so we can use them later when we search for the *k-nearest-neighbors*.

After that, we build a packed linear quad-tree structure with the points in screen space. We also create helper data structures to speed up indexing in this structure. After that, the quad-tree is ready to answer our queries on the pixels which are covered by one or more projected points.

As an estimation to how many neighborhood candidates need to be considered, we use the number of points in the quad-tree nodes. We start at a leaf node and traverse up the tree until we find a node with at least k points. From this, we can compute an initial radius estimation to find the nearest neighbors. Up to two additional iterations are possible to adapt the size of the search radius. For a valid solution, we now have a set of at least k valid neighbors for each observed point.

Reconstructing surfaces this way has some interesting benefits.

- Not dependent on preceding calculations
  We don't need a preprocessing step to visualize the point clouds. This means the algorithm should be able to cope with dynamically changing point clouds and scenes.

- Computation in screen space
  Many steps of the computation can be done in screen space which means the performance

our implementation should be more bound by the screen resolution, than the sheer point cloud size.

- Computation on GPU
  The introduced data structures are well suited to be computed by a modern GPU. This means that a great speed-up can be achieved compared to similar CPU solutions.

An analysis of the performance and accuracy of our approach can be found in chapter 5.

CHAPTER 2

State of the art

The k-nearest neighbor method is a well known and widely used method for classification. The basic idea of the algorithm is very simple, which makes it well suited for parallelization. Given a set of points, we take one of them and compute it's Euclidean distance to all other points in the set. After sorting these points ascending by their distance the k first points in the result array are our k-nearest neighbors. We now repeat this calculation for all other points in the set. As one can see, this leads to a very compute-intensive algorithm, when working on huge arrays of points. Optimization can be done by introducing spatial data structures like kd-trees, b-trees, grid-structures, and BSP trees, just to mention a few. Another optimization approach could be porting the algorithm to modern GPUs or Accelerators like Intel's Xeon Phi, to harness the massive parallelism of these devices. Even low-end graphics cards may outperform CPUs nowadays with a well-optimized algorithm and the usage of APIs like CUDA, OpenCL or OpenACC becomes more user-friendly with each subsequent release.

Garcia et al. compared different parallel approaches on GPUs [2]. They assumed, that a brute force approach might be possible, but found out, that they could speed up their algorithms massively by introducing spatial data structures. Nikam and Meshra also implemented Knn algorithms on GPU but with the use of the OpenCL library [3].

Zhou et al. showed that it is possible to create kd-trees in real time on GPU harnessing the streaming architecture of modern GPUs in every stage of the kd-tree construction. They show that their algorithm is able to handle dynamic scenes including GPU ray tracing, interactive photon mapping, and point cloud modeling [8].

According to Connor and Kumer [1] it seems that kd-trees are in general the best data structures for the Knn problem, but in more special cases other spatial data structures might be faster. They also found out, that when using a quad-tree like data structure, storing the points in Z-order might significantly speed up the processing, because then the data itself implicitly contains information about its spatial relations.

But the most important preceding work for this thesis has been done by Preiner et al. on Autosplats [5]. Autosplats is a point cloud visualization pipeline based on splatting, which does not need any form of pre-computation. Autosplats does not use a GPGPU framework but is

implemented as OpenGL shaders. To simplify the calculation only points in screen space are considered, which basically constrains the complexity of the problem to the size of the framebuffer.

CHAPTER 3

Implementation

3.1 Overview

We begin with a rough overview of the algorithm, which has been implemented with the use of Nvidia CUDA, and then we will have a more detailed look at important parts of the implementation. The output of our algorithm will be the input data for a simple splat rendering algorithm implemented in OpenGL. Because of this, we make heavy use of the inter-operation capabilities of OpenGL and CUDA.

The algorithm is split into multiple kernels, that are executed in serial and correspond to the steps in this section. At the start of the algorithm, point cloud data is uploaded onto the device memory. After that, a histogram is calculated that holds the information on how many points will be mapped into each pixel. We use that information, later on, to create a prefix sum over this histogram. After histogram computation, the quad-tree data structure and indices are created and after that, points which are within screen space boundaries are projected into the quad-tree. It is worth to note, that no per pixel depth test is used in this procedure, so it is valid to have multiple points per pixel. With the points in the quad-tree, we now calculate an estimated radius to check for each pixel. After that, the Knn algorithm starts for each pixel and searches for the given amount of valid neighbors. If it finds too many or too few points for a pixel, the search radius gets modified and we start the search again. Up to 2 iterations of refining the radius are possible. After that, we feed our output data into a simple splat renderer, which renders the point clouds.

3.2 Histogram computation

Our implementation is fixed to a frame-buffer size of 1024x1024, but it is worth to mention, that the algorithm can cope with arbitrary frame-buffer sizes. So for each pixel, we count the amount of points that will be projected into this pixel. Because the GPU implementations of projections always use a per-pixel-depth-test, which is what we wanted to avoid, we created our

own projection algorithm with Nvidia CUDA. A thread is started for each point in the point cloud and checks into which pixel the given point projects. Then a counter for that particular pixel gets incremented. This needs to be implemented as an atomic operation because we want to avoid, that a race condition may occur when 2 points project onto the same pixel. We used the atomicInc(...) function of Nvidia CUDA for that. Since we are projecting into a frame-buffer with more than one million pixels, collisions should happen seldom.

Algorithm 2 Compute number of points in each pixel

```
1:  procedure PROJECTIONHISTOGRAM
2:      Count Point Cloud Elements which are projected to a particular screenspace pixel
3:      for all Elements in the PointCloud do
4:          p = p * mvMatrix;
5:          out.x = p.x + 1.f * 512.f
6:          out.y = p.x + 1.f * 512.f
7:          if PointIsVisible then
8:              count[x+1024*y]++
9:          end if
10:     end for
11: end procedure
```

3.3 Prefix-Sum Computation

Over the computed histogram we also compute the exclusive prefix sum, so we can use it later as indices for fast access into our quad-tree when we search for the valid neighbors. We do the computation with help of the Nvidia Thrust library [4], which provides parallelized methods, that are running on GPU for this task.

3.4 Quad-tree Layout and Construction

To harness the advantages of caching techniques, we project the points into a pre-created quad-tree data structure. The knowledge of how many points from the point cloud are projected per pixel is enough to create the indices and the empty quad-tree structure. Then we project the points into their given positions in the quad-tree. Spatially close points are now packed densely in memory, which should speed up our neighbor search. The chosen data structure and the algorithms for filling it was created and explained by Dominik Schoerkhuber [6] so we give only a short overview here. More about the optimizations of this data structure can be found in chapter 4 of this paper.

As stated before, the algorithm makes heavy use of buffers to maintain a pointer-free data structure. Buffers that are used before the projection step are fixed to a size equal to the number of points in the point cloud p. Buffers that are used after the projection step are fixed to a size equal to that of the quad-tree q. l means the number of pixels in screen space which is fixed to 1048576 (1024x1024) in our implementation.

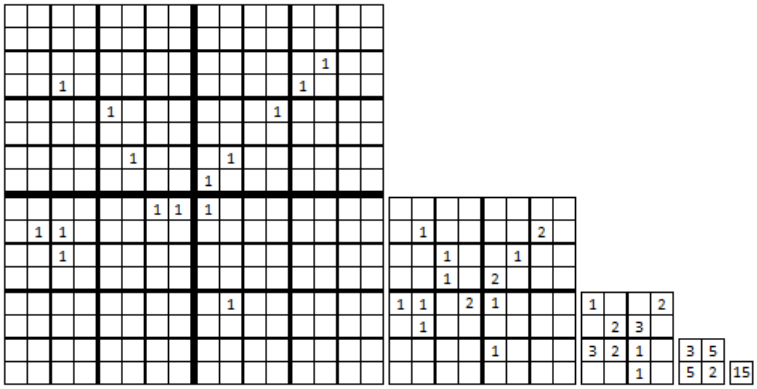

Figure 3.1: Each point represents an occupied pixel in screen space. Point numbers are accumulated in each quad-tree node. Starting with the lowest quad-tree node level covering only one pixel, the size of a node quadruples in each step. On its highest level, the quad-tree has only one node, containing all points (read from left to right). [6]

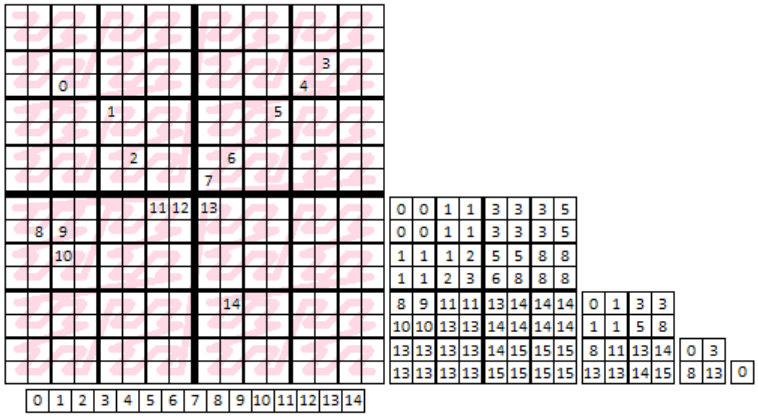

Figure 3.2: This illustration continues the example from figure 3.1. On the bottom, we see the expected packed quad-tree buffer. The number in each node describes its index in the packed buffer, which is equal to its z-iteration order. Indices in the lowest quad-tree level point to exactly one entry. For the construction of the packed buffer, we compute the index offsets from the right (highest-level) to left (lowest level). [6]

- NumBuffer[q] : uint32
  For each quad-tree node, we store the number of points it contains. It corresponds to the illustration in Figure 3.1. The first l points represent the point numbers of leaves 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. One has to keep in mind, that because z-buffering is turned off, even at the lowest level, nodes can hold multiple points. [6]

- 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. [6]

- PackedQuadtree[l] : 3xfloat32
  Stores the view space positions of occupied pixels. [6]

- PackedQuadtreeCoordinates[l] : uint32
  Contains the (x, y) pixel coordinate of a point in the PackedQuadtree buffer. This is basically the reverse of the IndexBuffer[q]. [6]

We begin constructing the quad-tree with the NumBuffer that has been filled by the projection step before. So our projection histogram is basically the leaf-level of the quad-tree. From that level, we propagate the information through one kernel run to the upper level, by summing up the points of the lower four sub-nodes. After $\log_2(1024)$ kernel runs, we have filled our NumBuffer up to the highest level.

Since we now have the indices and the quad-tree structure set up, we go through the given elements in the point cloud and project them into their correct position in the quad-tree.

Algorithm 3 Compute number of points in each quad-tree node [6]

```
1:  procedure PROPAGATENUMS
2:      Count projected pixels and write the per-pixel amount into the leaf-level of the Num-
    Buffer
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
```

Algorithm 4 Compute number of points in each quad-tree node [6]

```
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
```

3.5 Finding the point with the shortest distance to the viewer

Through the built up data structure, we are now able to retrieve the elements of the quad-tree which map to a given pixel. But to reconstruct the surface from a given point of view, we have to compute the nearest neighbors for the surface-points that are visible for the viewer not for the ones covered by other points. So if multiple points map to the same pixel, we need to find out which of them is the one with the shortest distance to the viewer. We achieve this by adding an additional kernel run. For each pixel, the element with the shortest distance to the viewer (in screen space) is computed and this element gets swapped with the element on position 0 for that pixel. Now the first point we retrieve for a given coordinate is always the one with the shortest distance to the viewer.

3.6 k-Radius Estimation

The *k-radius-estimation* is the next step in our algorithm to determine the *k-nearest-neighbour* set for each point in screen space. The initial estimation is very important because with good estimations we reduce the number of needed iterations in a later stage of the algorithm. The estimation is roughly based on the number of points in the quad-tree nodes and all points in the estimated K-Radius will be considered to be in the *k-nearest-neighbour* set. We look at a point in the quad-tree and traverse it from that point to the root node. When we find a node that holds at least k_{min} points we stop. k_{min} is, in this case, the amount of nearest neighbors we are searching for. Because of further refinement later on in the algorithm, we can also accept more than k_{min} candidates up to an implementation-dependent number of k_{max} points. Since the area covered by a node is usually not fully covered by points the node-area radius gets scaled accordingly. Details to that can be found in [6].

3.7 Bounding Box Search

Given the estimated radius we can now collect the points lying inside it. To achieve this we create a bounding box around each pixel on the screen where the edge length is 2 times our

estimated radius. So for each pixel, we check all the neighboring pixels in the given bounding box if we find k_{min} elements. A naive implementation is given here. Due to its simplicity, the algorithm can easily be run in parallel.

Algorithm 5 Search the bounding box for elements

```
1: procedure BOUNDINGBOXSEARCH
2:     for all Pixels in the bounding box do
3:         element = getElementForPixel(x,y)
4:         if DistanceToElement < EstimatedRadius then
5:             validPoints++
6:             storePoint(element)
7:         end if
8:     end for
9: end procedure
```

If we find at least k_{min} and at most k_{max} elements, our initial estimation was appropriate and we continue with the algorithm.

3.8 Iterative Radius Optimization

Since our initial assumption was only an estimation, it might happen, that we get less than k_{min} or more than k_{max} points when we search a given bounding box. In that case, the estimated radius gets slightly adopted for that given point and we restart our bounding-box-search. 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 the work of Dominik Schoerkhuber [6] 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 3.3 shows the percentage of points with a correctly sized *candidate-set* after n iterations.

3.9 Sorting and Filtering

After collecting the neighbors we need exactly k_{min} of them, but what we get is an amount between k_{min} and k_{max} points. So the points need to be sorted, to find the k-nearest ones of them. We tested our application with an own heap-sort implementation which was discussed in detail by Dominik Schörkhuber [6] and with the default sorting algorithm of the Nvidia CUDA Library (thrust::sort), but we found out, that just taking the first 8 points of the given set without sorting will not hurt the accuracy as much as one would think and leads to a significant increase in speed.

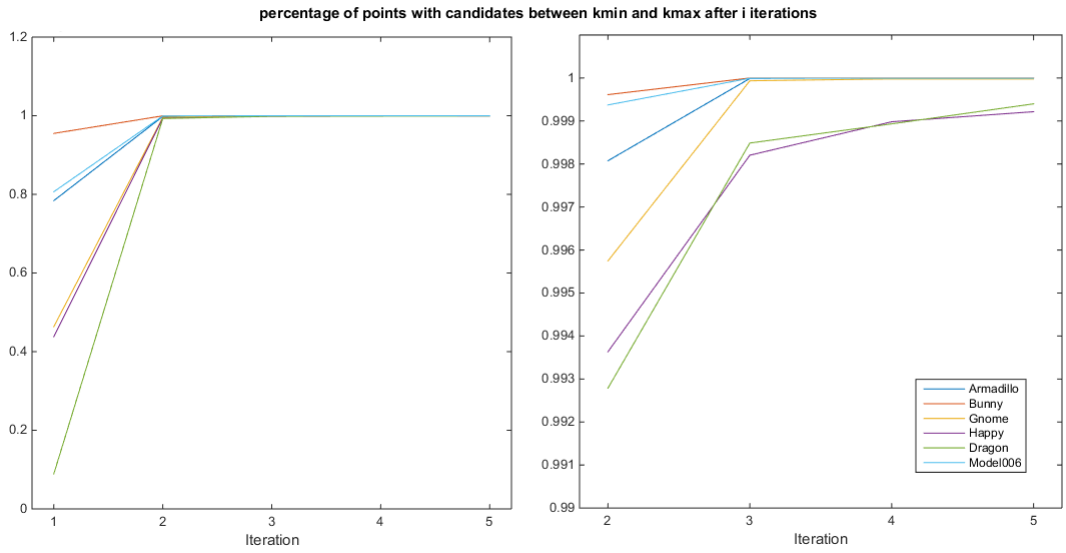

Figure 3.3: For each tested model, the percentage of correctly sized *candidate-sets* after n iterations is shown. Dominik Schörkhuber observed [6] that three iterations are enough to generate appropriately sized sets. Our observations confirmed that.

CHAPTER 4

Optimization

4.1 Overview

During the development of the algorithm, we tried to optimize different aspects of it in regard to performance as well as in regard to accuracy. This chapter discusses the outcomes of these optimizations.

4.2 Search radius refinement on CPU / GPU

As stated before in 3.8 the search radius refinement for each point used up to three iterations. In the Fastknn version of Dominik Schörkhuber, the algorithm consisted just of a for loop over the maximum iteration count for each pixel started parallelized as a CUDA kernel. This led to different run-times of the kernels because there are regions on the screen consisting of many points and regions consisting of only 1 or even no points. So some kernels exited very early, while others had to do up to 3 iterations and had a significantly longer runtime. Subsequently, this led to non-optimal performance.

Because of that, we introduced an additional buffer where we store the information if the search radius for a given point is correct or if we need to do another iteration. Through this, we could move the for loop from the kernel itself into the calling code which is running on the machines CPU and call the kernels only for points that need refinement. This approach looked promising in the beginning, but because of our implementation needing the points as a compact array, we needed to introduce an additional compacting step. This step, depending on the model, reduces or even neglects the performance gain of this optimization.

So the current solution is to do the iterations on CPU, but a CUDA kernel is started for each pixel, in each iteration and exits early if the pixel already has a correct estimated radius. This seemed to be the most promising solution and decreased the rendering times up to 10% depending on the model.

4.3 Removal of the per-pixel-depth-test

Earlier versions of the algorithm used OpenGL to project the elements of the point cloud into screen space. OpenGL uses a per-pixel-depth-test, through this all pixels which are covered by other pixels from the given point of view, are discarded. A more in-depth overview of per-pixel-depth-testing and other appliances of z-buffers can be found in Theoharis et. al. [7]. This behavior led to lost underlying point cloud elements if multiple elements were projected on the same pixel and visible errors on the edges of the reconstructed geometry. So the projection was completely rewritten in CUDA and all elements that are covered by another one in screen space can now be checked by the fastknn kernel. The performance impact of this change depends on the structure of the given point cloud because we needed to introduce an additional compute step as described before 3.5, but this led to a significant decrease of errors on the edges of the reconstructed geometry 4.1. The change also had an impact on the memory footprint of the application, because earlier versions of the algorithm only had to store the visible elements of the point cloud in the quad-tree. Now also the elements which are covered by others from the given point of view need to be stored.

4.4 Removal of Sorting

After the candidate search, we get between k_{min} and k_{max} points, but we want the nearest neighbor set to be of size k_{min}, so the results were sorted depending on their squared distance in world space. Unfortunately, the NVidia CUDA library does not allow to dynamically index register based arrays, so an optimized register based version, especially for the sorting of 8 neighbors, was created. After removing the z-buffer we also removed the sorting step at the end, because the number of elements that need to be considered can be much higher than in the old implementation. This led to a severe performance impact, as can be seen in 4.2.

4.5 Memory Layout

In the process of reworking the kernels, we tried to improve the performance by changing the memory layout of the kernel, especially by swapping the (x, y) coordinates in the for loop for the bounding box. Interestingly this led neither to a significant speedup nor a significant slow-down of the implementation. It seems, that the Nvidia-compiler is able to optimize this in our implementation by itself.

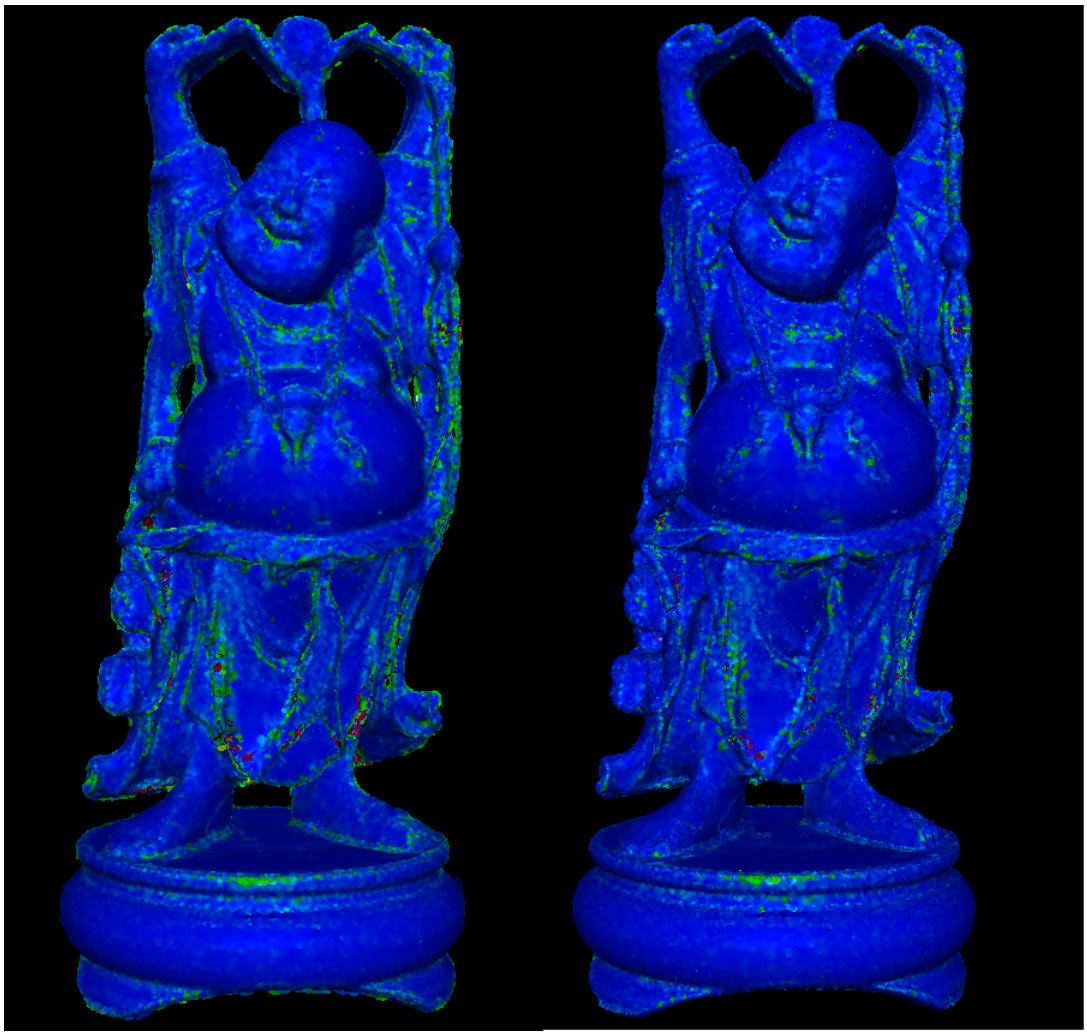

Figure 4.1: Rendering of the Happy-Buddah Model with activated (left) and deactivated per-pixel-depth test (right). The removal of the per-pixel-depth test leads to a greater accuracy especially around th edges of the reconstructed geometry.

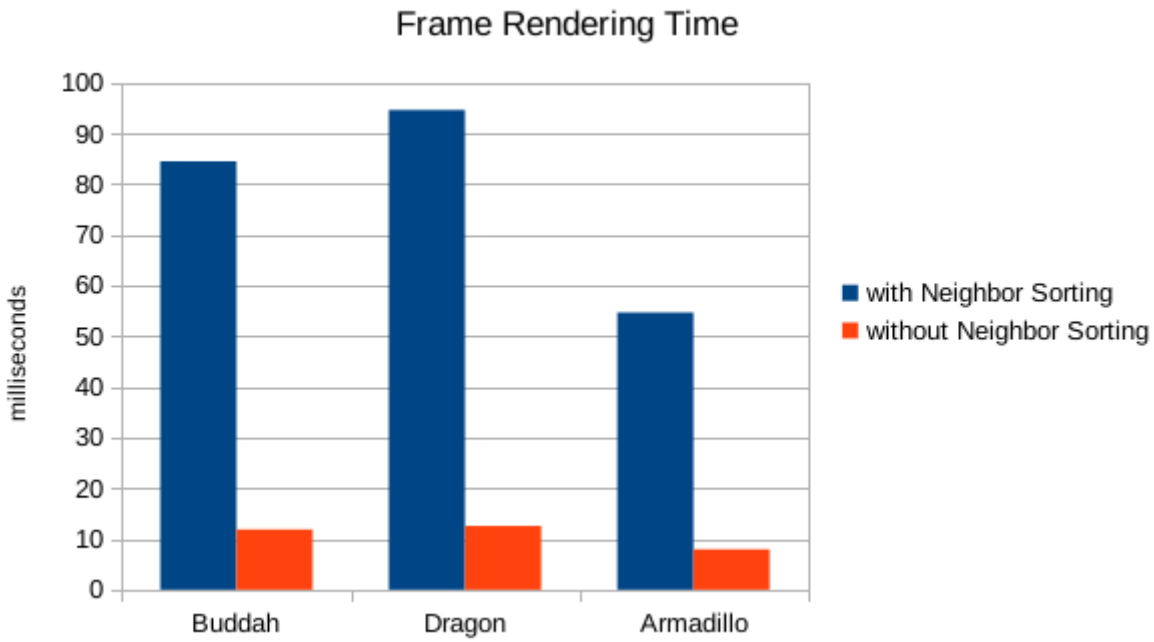

Figure 4.2: Rendering of the models with activated and deactivated neighborhood sorting. The activation leads to a heavy performance impact due to the sheer number of neighbors that need to be sorted.

CHAPTER 5

Analysis

5.1 Analysis

In this section we compare an older approach of Fastknn by Dominik Schoerkhuber [6] (Fastknn1), our new Fastknn approach (Fastknn2) and the Autosplats algorithm [5] in terms of speed and accuracy. For a fair comparison, the Autosplats pipeline was integrated into our rendering pipeline and we were able to obtain detailed timings for all the stages of the algorithm. Autosplats is also utilized to do the final splatting. Because of the Fastknn as well as the Autosplats algorithms are only approximate solutions we compare our results to an exact solution. All the results have been computed on a Geforce GTX 1080 which was a high-end consumer graphics card in 2017.

Speed

In terms of speed, our new Fastknn approach falls a little bit back behind the old one because of the decision to remove the per-pixel-depth-test. Depending on the resolution of the point cloud and the viewer's position, the knn kernel needs to check more elements if they are valid neighbors. When the whole point cloud is visible on-screen, every point cloud element gets projected into our quadtree. Figure 5.1 shows the number of elements in the quad-tree after the projection step compared to the older solution Fastknn1. Very high-resolution models can raise the element count even by one magnitude, which can be seen on the dragon2 model, that consists of approximately 4 million point cloud elements. Fortunately, the performance impact of the increased number of elements is not as huge as one would expect 5.2. This also holds true for the comparison to the Autosplats algorithm. Especially with huge point cloud sizes Fastknn2 loses performance compared to Autosplats.

Accuracy

Our resulting k-nearest-neighbor sets are only approximate solutions so we analyzed the correctness of the computed sets compared to an exact solution computed on CPU. The older Fastknn

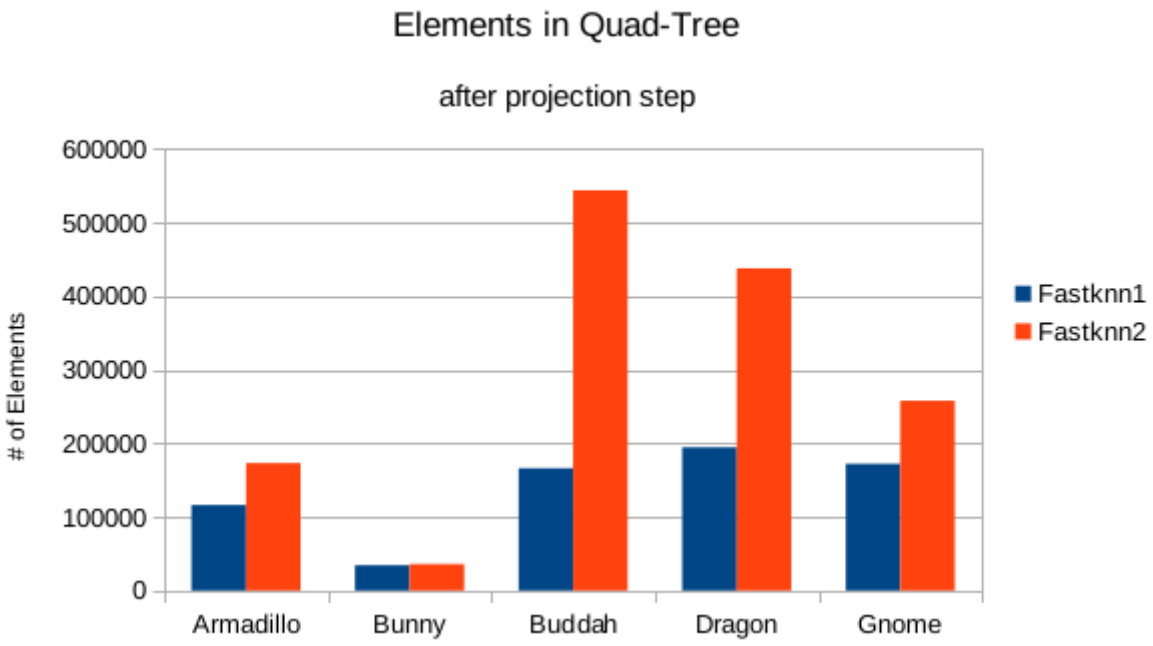

Figure 5.1: Amount of elements after the projection step is done for Fastknn1 and Fastknn2. It is clearly visible that Fastknn2 considers more points for the candidate search because of the removed per-pixel-depth-test.

and Austosplats implementations showed some problems on the boundary regions of the reconstructed geometries where the normal vectors are close to perpendicular to the image plane, so we focused on this aspect of the algorithm. With the removal of the per-pixel-depth-test, we were able to increase the accuracy in these regions significantly but as stated before at the cost of more needed computing power. 5.3 and 5.4. Figures 5.5 and 5.6 also show a visual comparison of normal qualities for the two Fastknn implementations compared to the exact normals which were part of the given point cloud data. 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.

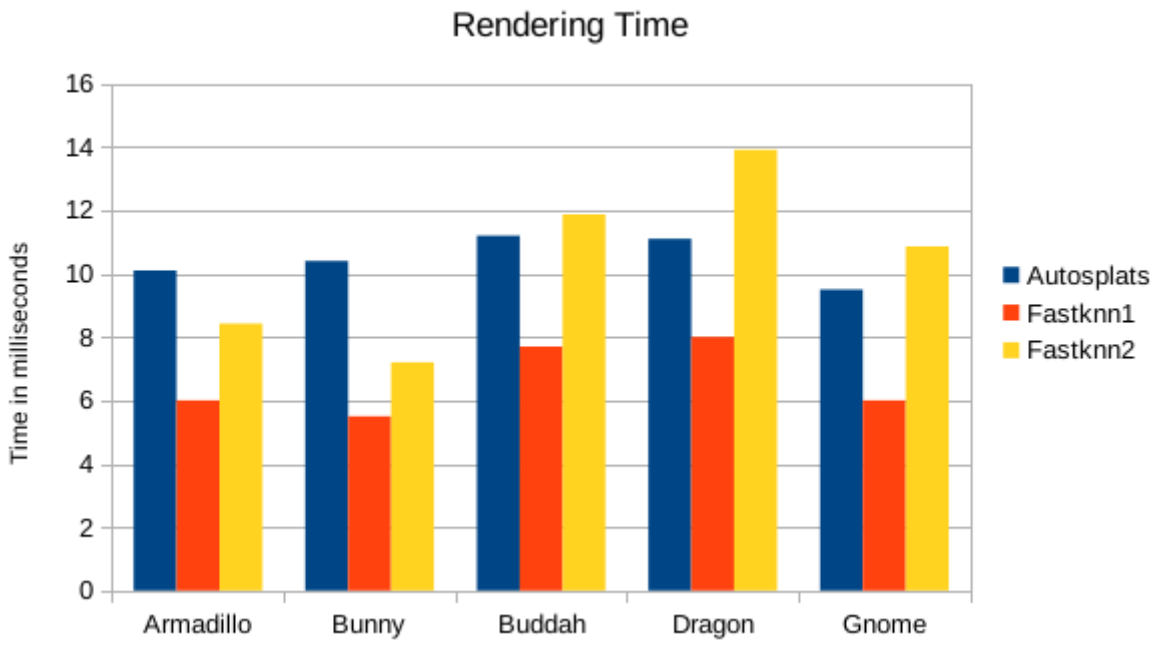

Figure 5.2: The time it takes to render the point cloud when it's completely visible on screen. Due to the increase of the element count in the quad-tree Fastknn2 performs a little slower than Fastknn1.

Normal consistency

On our reconstructed surfaces, we observed the problem of surface consistency. The estimated normals of the surface should always be identical no matter the angle or the distance to the camera. While the normal errors we showed before are usually low and might even be unnoticed on a static scene, transformations over time can reveal these errors. Again especially of the boundaries of the surfaces errors may occur, when transformations are done and these errors are much more visible for a viewer. Fastknn2 reduces these errors compared to Fastknn1 and Autosplats, but further work on accuracy might be needed if one wants to visualize a dynamic scene.

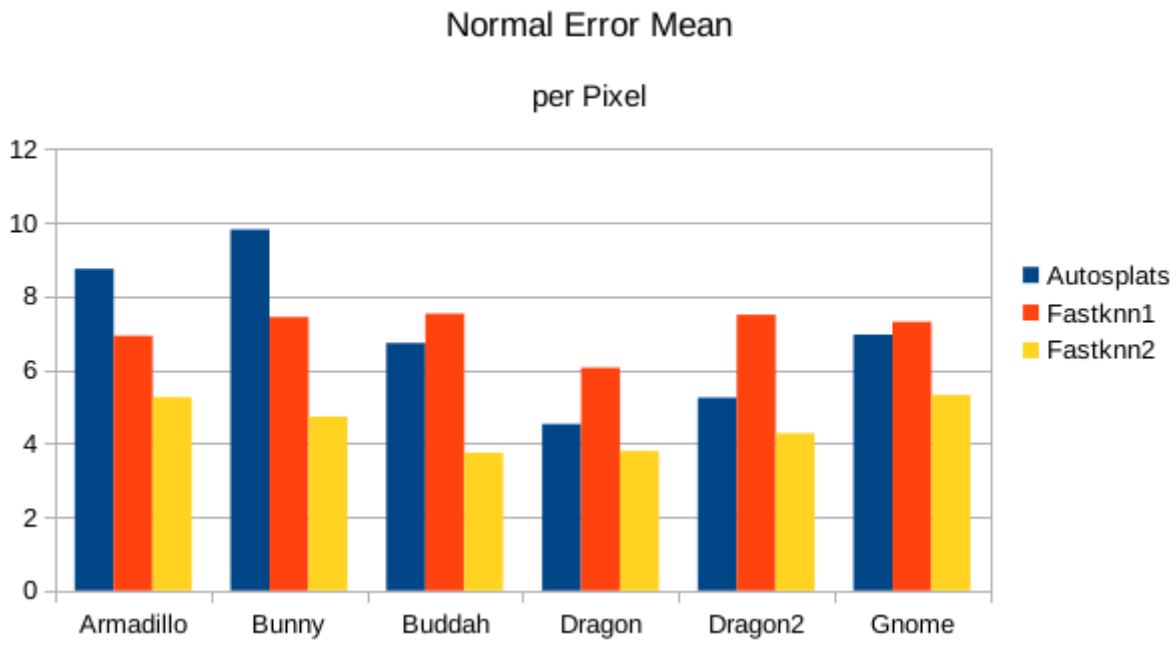

Figure 5.3: The mean of the normal error per pixel of Fastknn2 compared to Fastknn1 and Autosplats. The removal of the per-pixel-depth-test led to a significant gain in accuracy across all tested point clouds.

5.2 Conclusion

This paper shows how the GPU can be utilized to parallelize the *k-nearest-neighbour* problem for general unstructured point clouds. It also shows how different aspects of our algorithm may be written with exactness or performance in mind, but one can not have both. More accuracy means that one has to consider more points and/or increase the complexity of the initial radius estimation, which subsequently leads to more computation. On the other Hand, it's possible to improve the performance of the algorithm without hurting the accuracy of the reconstructed geometry too much. Fastknn shows, that a visually appealing reconstruction can be done in real-time, although with some limitations. Our observations show, that Fastknn2 produces less errors than Autosplats with approximately equal rendering time and Fastknn1 renders faster than Autosplats with approximately equal error rates. Thinking of the massive amount of data that 3d scanners produce, approximate solutions might be the way to go when it comes to scenes with millions or even billions of point cloud elements that need to be considered to reconstruct the geometry. But with the advent of new GPU generations every year, it might even be possible to create exact representations of point clouds in real-time in the future.

Finally, we can say, that we were able to further improve the accuracy of the Fastknn algorithm, at the cost of increased computing power.

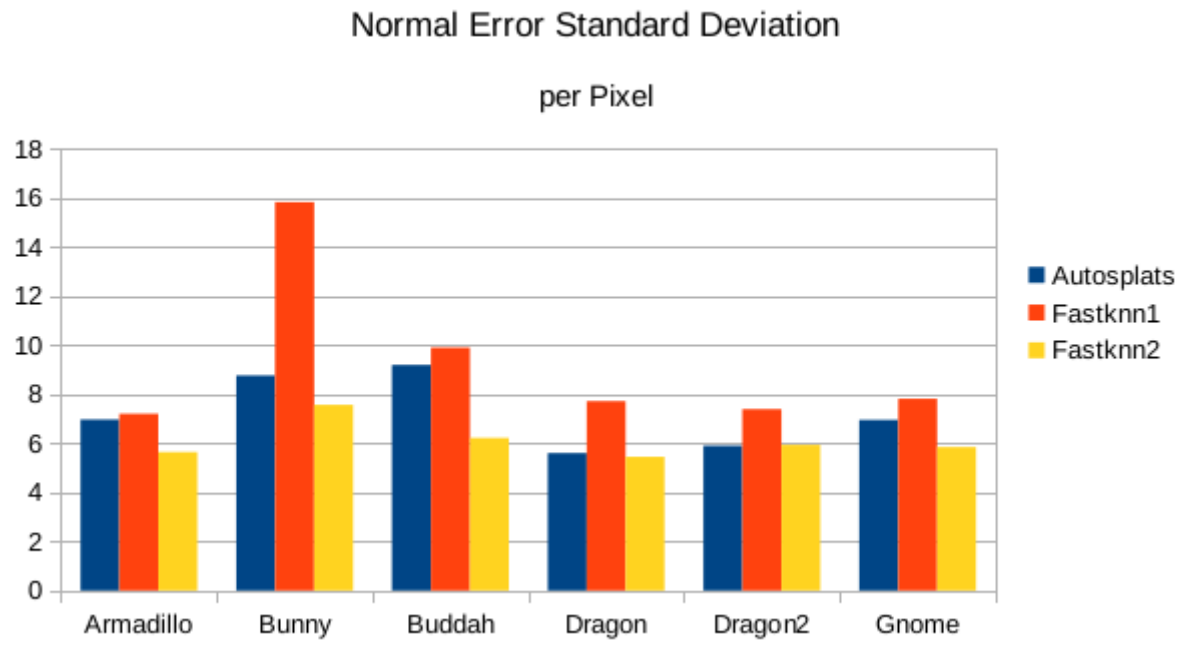

Figure 5.4: The standard deviation of the normal error per pixel of Fastknn2 compared to Fastknn1 and Autosplats. Compared to Fastknn1, the removal of the per-pixel-depth-test led to a significant gain in accuracy across all tested point clouds.

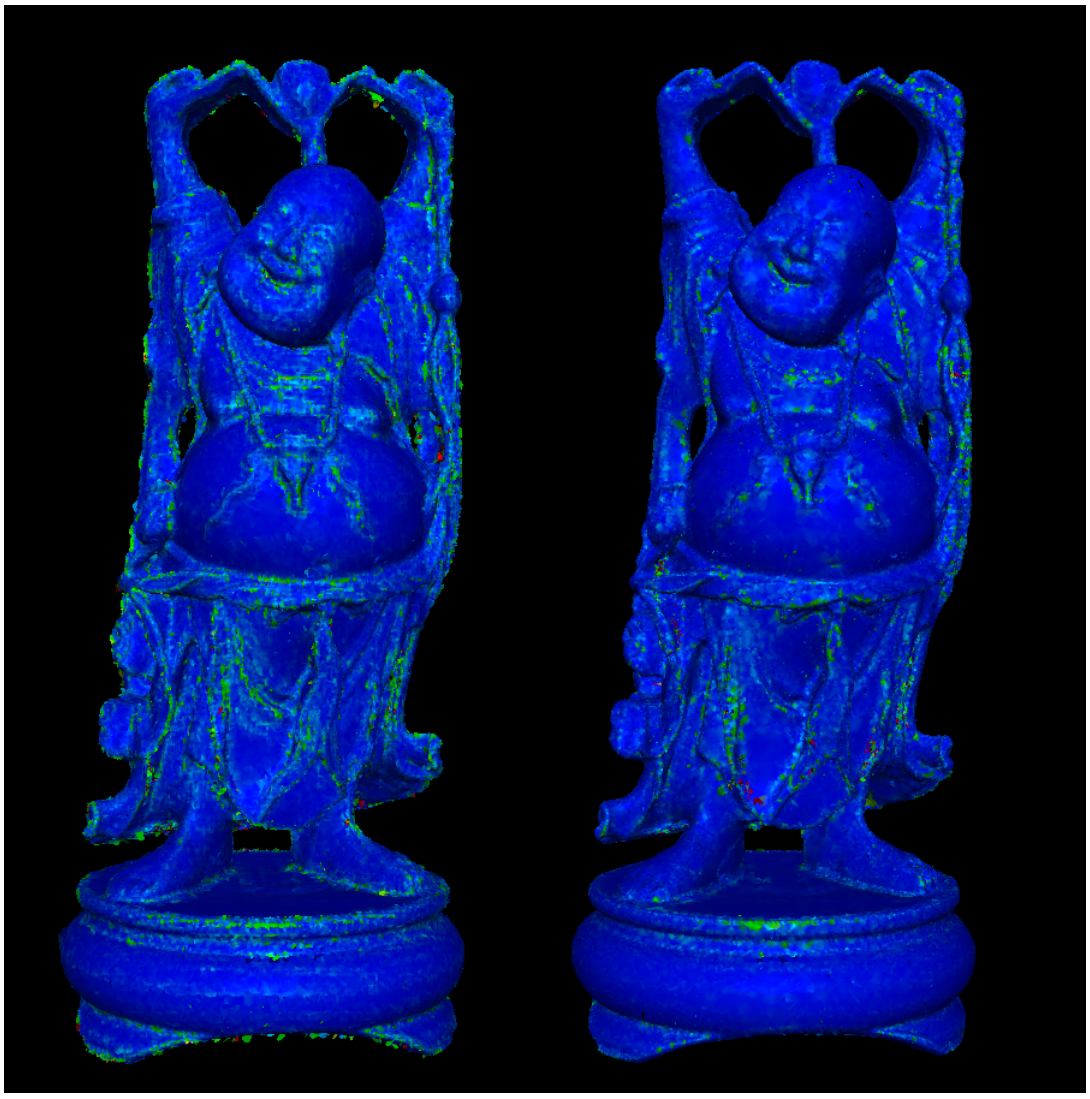

Figure 5.5: Visualization of the Buddah point cloud with Fastknn1 (left) and Fastknn2 (right). One can see, that the surface reconstruction works better especially at the borders of the geometry.

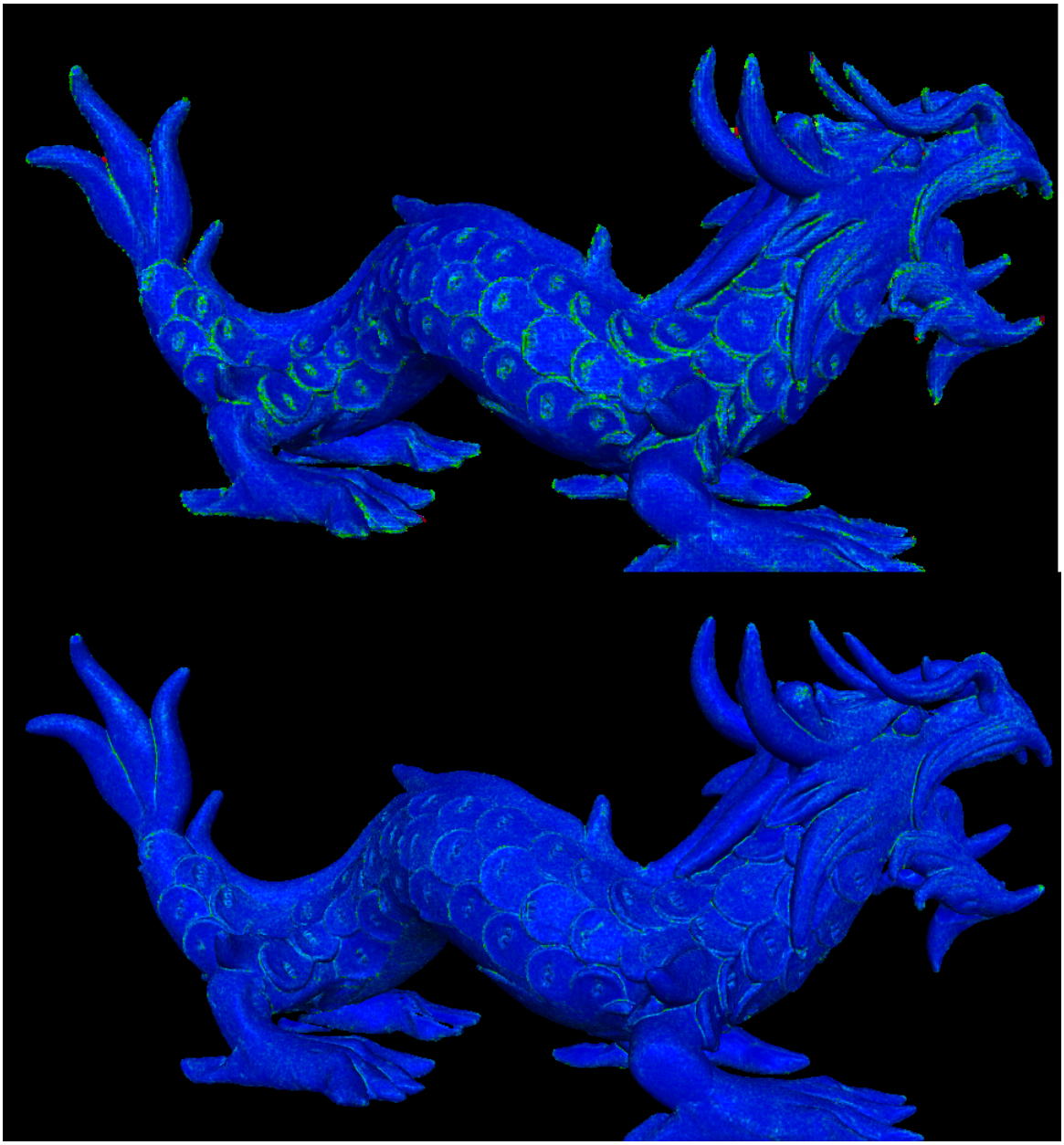

Figure 5.6: Visualization of the Dragon2 point cloud with Fastknn1 (upper) and Fastknn2 (lower). The improvement of accuracy is especially visible at the borders of the dragon scales, but nearly all parts of the geometry benefit from the increased accuracy.

Appendix

Appendix A: Tables for Figures

Algorithm	Model	Time [ms]
w/o. Sorting	Armadillo	8.42
w. Sorting		54.65
w/o. Sorting	Buddah	11.87
w. Sorting		84.48
w/o. Sorting	Dragon	13.90
w. Sorting		94.61

Table 1: Rendering times for different models. With and without activated Neighbor sorting. Data for Figure 4.2

Algorithm	Model	Element-Count
Fastknn1	Armadillo	116092
Fastknn2		172974
Fastknn1	Bunny	34355
Fastknn2		35947
Fastknn1	Dragon	194463
Fastknn2		437645
Fastknn1	Buddah	166083
Fastknn2		543652
Fastknn1	Dragon2	206028
Fastknn2		3609600
Fastknn1	Gnome	171986
Fastknn2		257706

Table 2: Elements in the quad-tree after the projection step. Fastknn1 compared to Fastknn2. Data for Figure 5.1

Algorithm	Model	Time [ms]
Autosplats	Armadillo	10.10
Fastknn1		5.96
Fastknn2		8.42
Autosplats	Bunny	10.40
Fastknn1		5.50
Fastknn2		7.20
Autosplats	Dragon	11.10
Fastknn1		7.93
Fastknn2		13.91
Autosplats	Buddah	11.20
Fastknn1		7.7
Fastknn2		11.87
Autosplats	Dragon2	9.70
Fastknn1		6.96
Fastknn2		65.13
Autosplats	Gnome	9.50
Fastknn1		6.00
Fastknn2		10.86

Table 3: Rendering times for different models. Fastknn1 compared to Fastknn2. Data for Figure 5.2

Algorithm	Model	Mean	Stdev
Autosplats	Armadillo	8.74	6.97
Fastknn1		6.92	7.2
Fastknn2		5.25	5.64
Autosplats	Bunny	9.81	8.76
Fastknn1		7.43	15.83
Fastknn2		4.73	7.56
Autosplats	Dragon	4.52	5.60
Fastknn1		6.05	7.72
Fastknn2		3.79	5.45
Autosplats	Buddah	6.72	9.19
Fastknn1		7.52	9.9
Fastknn2		3.74	6.22
Autosplats	Dragon2	5.24	5.90
Fastknn1		7.49	7.39
Fastknn2		4.27	5.93
Autosplats	Gnome	6.95	11.05
Fastknn1		7.3	7.81
Fastknn2		5.31	5.84

Table 4: Comparison of mean and standard deviation of the normal errors per pixel for different models. Data for Figure 5.3 and 5.4

Bibliography

[1] 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.

[2] 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.

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

[4] Nvidia. Thrust library. https://developer.nvidia.com/thrust. accessed 03-March-2019.

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

[6] Dominik Schoerkhuber. Fast knn in screenspace on gpgpu. Bachelor's thesis, TU Wien, 2016.

[7] Theoharis Theoharis, Georgios Papaioannou, and Evaggelia-Aggeliki Karabassi. The magic of the z-buffer: A survey. In *Proceedings of the 9-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 379–386. University of West Bohemia, 2001.

[8] 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.