Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Development of GPICP, a GPU-only registration algorithm based on Iterative Closest Point

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Yaroslav Barsukov
Matrikelnummer 0526662

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Markus Vincze
Mitwirkung: Dipl.-Ing. Aitor Aldoma

Wien, 10.06.2013

(Unterschrift Verfasser)

(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Development of GPICP, a GPU-only registration algorithm based on Iterative Closest Point

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Yaroslav Barsukov
Registration Number 0526662

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Markus Vincze
Assistance: Dipl.-Ing. Aitor Aldoma

Vienna, 10.06.2013

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Yaroslav Barsukov
Gabelsbergergasse 6/6, 1020 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)

Acknowledgements

Great thanks to my supervisors, Prof. Markus Vincze und Dipl.-Ing. Aitor Aldoma, for their active support.

Abstract

The ICP algorithm, developed by Besl and McKay, has long been the ”golden standard” for solving the registration problem. With the recent increase in programmability of GPUs, their parallel programming model has become a perfect medium for ICP. The goal of this work was to develop an ICP implementation that could run entirely on the GPU, solving the problems imposed by the GPU architecture and harvesting increased performance in an effort to make the registration more accurate.

Kurzfassung

Der von Besl and McKay entwickelte ICP Algorithmus ist längst zum Goldstandard fürs Lösen des Reqistrierungsproblems geworden. Nachdem sich die Programmierbarkeit der GPUs in der letzten Zeit erhöht hat, ist ihr paralleles Programmierungsmodel nun für ICP perfekt geeignet. Die vorliegende Arbeit setzt sich zum Ziel, eine Implementierung von ICP zu entwerfen, die völlig auf GPU ausgeführt werden könnte; die dabei entstehenden Probleme mit GPU Architektur werden gelöst, und die erhöhte Leistung wird dafür benutzt, den Algorithmus selbst präziser zu machen.

Contents

1 Introduction 1

1.1 Related work 2

1.2 Essence of this work 2

2 Algorithm Description 5

2.1 GPICP: general description 6

3 Mathematical optimizations 9

3.1 Increasing the registration accuracy 9

3.2 RANSAC set size 16

3.3 NNS 22

3.4 Optimizing the NNS 22

3.5 Reducing the set of hypothesis' points 24

3.6 Exact vs. approximate NNS 29

3.7 Approximate and depth-first searches 30

3.8 Finding the transformations 35

4 GPU optimizations 37

4.1 Introduction to CUDA architecture 37

4.2 One MP – one block – one hypothesis 39

4.3 Balanced vs. left-balanced k-d tree 40

4.4 Improved k-d tree search 41

5 Results & discussion 45

6 Conclusions 53

Bibliography 55

CHAPTER 1

Introduction

Three-dimensional reconstruction of real objects is an important topic in computer vision. The process includes two steps [1,2]:

- 3D data acquisition;
- reconstruction.

For the purpose of the first step, laser scanners and time-of-flight cameras are typically used, providing so-called *range images* of the object, taken from different viewpoints and often - at different times [1,2]. A range image (or depth map) is an image whose pixels also store distance from the sensor's origin; such pixels can thus be represented by points in 3D space, and the range image itself - by a point cloud [1,3].

The second step includes aligning the acquired point clouds in a process called "point cloud ***registration***".

The goal of registration is to align (combine) two or more point clouds into a global consistent model. The notion of alignment infers that one of the clouds acts as a reference, while the others (referred to as "hypotheses") are being transformed so that their orientation would match this reference. In effect, the point clouds are being "glued" together.

While object reconstruction has many applications (map building in robotics [4], 3D modelling for virtual museums [5] etc.), registration itself is used in even more areas. One example would be tumour growth monitoring & treatment verification, where registration ensures that the same region of human body is evaluated during the baseline and follow-up scans [6].

The ICP algorithm, developed by Besl and McKay [7], has long been the "golden standard" for solving the registration problem [8]. Despite being straightforward, it contains a computationally expensive closest point search, which, however, lends itself to parallelization [9]. With the recent increase in programmability of GPUs, their parallel programming model has become a perfect medium for ICP; however, up to this day, mostly half-GPU, half-CPU implementations exist [10]. The dual nature of these implementations significantly reduces potential performance (see Section 4.1).

1.1 Related work

Ever since the increase in programmability of GPUs, there have been attempts to implement ICP on the graphics hardware; the earliest efforts go as far as 2008, using vertex and fragment programs [11]. With release of CUDA, GPGPU (general-purpose computing on graphics processing units) became easier and more accessible, spawning a number of new ICP implementations.

Unfortunately, many of these implementations are either CPU-GPU hybrids, utilizing the GPU only for the ICP's most costly part, nearest-neighbour search (NNS) [9, 10], or are generic parallel algorithms that pay little attention to the underlying architecture, its limitations and possibilities it provides [12, 13].

While the traditional ICP utilizes k-d trees for the purpose of nearest-neighbour search, a comprehensive study on the subject by Brown and Shoeyink shows that k-d trees do not map good to the GPU [14]; the main reason for that is the non-parallel and recursive nature of a tree search. A number of researchers tried to circumvent the problem. Garcia et al. have shown that a GPU-based brute-force implementation outperforms a CPU-based k-d tree [15]; however, even with the modern GPUs' power, brute-force is still not an option when considering hundreds of hypotheses containing millions of points. A promising approach is the random ball cover (RBC) proposed by Cayton [16]; the basic principle behind the RBC is a two-tier nearest neighbour search, building on the brute-force primitive. KinectFusion framework [17] has gained much popularity, combining projective data association [18] and a point-to-plane metric [1] for rigid ICP surface registration.

Point-to-plane metric substitutes closest point search with an intricate algorithm that ultimately provides better results, but is more computationally expensive, maps worse to the graphics hardware and requires extensive pre-processing of the input point clouds [19]. Instead of searching for the closest points in the reference cloud, for each point of the hypothesis cloud, its normal is intersected with the reference's surface, and the point is then projected onto the intersection's tangent plane [1]. Such an algorithm first and foremost requires both reference and hypothesis clouds to have normals in the first place; calculating normals for a point cloud is a huge topic in itself. Furthermore, intersecting a vector with a point cloud's surface is a computationally expensive procedure; one of the acceleration techniques involves triangulation: it first searches for the closest point, and then finds the intersecting triangle from the located point's neighbouring triangles [19]. This complicates mapping the metric to the GPU.

1.2 Essence of this work

The first goal of this work was to develop GPICP - an ICP implementation that could run entirely on the GPU, solving the problems imposed by the GPU architecture. Furthermore, all of the discussed implementations retain the fundamental shortcoming of the ICP algorithm, i.e. its convergence to a local minimum; thus, the second goal was to increase the accuracy of registration. The resulting algorithm still doesn't reach the global minimum, but it tries to select a better local one, which is sufficient in most cases, as the results show (see Section 5). The algorithm is described in greater detail in Chapter 2.

A typical run of GPICP can be observed on Figure 1.1.

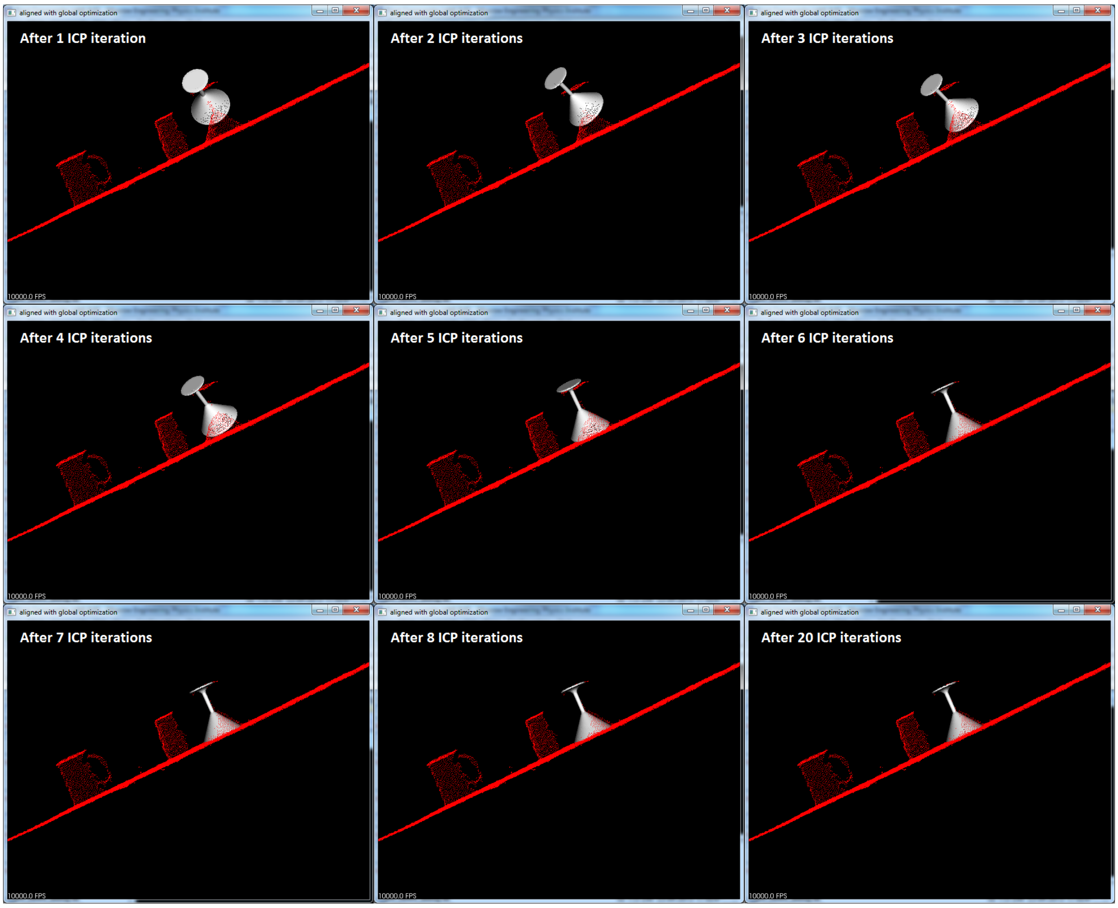


Figure 1.1: A typical GPICP run

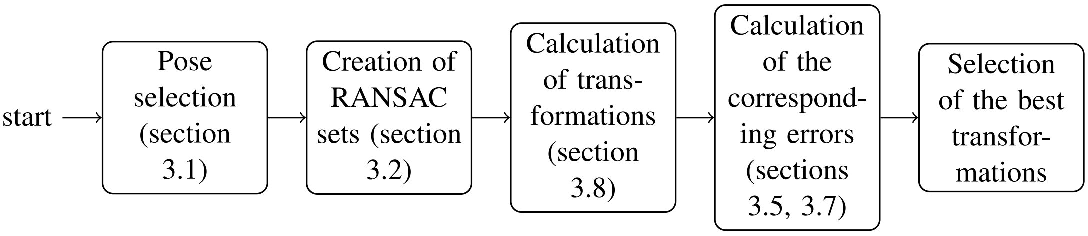

start
Pose selection (section 3.1)
Creation of RANSAC sets (section 3.2)
Calculation of trans-formations (section 3.8)
Calculation of the correspond-ing errors (sections 3.5, 3.7)
Selection of the best transfor-mations

Figure 1.2: General flow chart of GPICP

Since increased accuracy effectively means more computations, an extensive effort was put into optimizing the algorithm itself, getting rid of certain calculation redundancies. The main points of optimization can be seen at the flow chart on Figure 1.2.

Optimizations to the algorithm are of both mathematical and programmatic nature; programmatic ones facilitate the optimal mapping to the graphics hardware. Programmatic optimizations are discussed in Chapter 4.

CHAPTER 2

Algorithm Description

The ICP algorithm is iterative and works on two clouds - the ”target” (the reference cloud) and the ”hypothesis” (the cloud being transformed). In each iteration:

- for each point of the hypothesis cloud, the closest point in the target cloud is located (this step is called the ***NNS***, or nearest neighbour search);
- after the NNS is completed, there are two point sets to consider: the full set of the hypothesis cloud's points and the ”closest sub-set” of the target cloud's points; i.e. the ”closest sub-set” is such a set of points $\mathcal{P}_c$ that

$\mathcal{P}_c \subset \mathcal{P}_t$, where
$\mathcal{P}_t$ - the set of target cloud's points;
and
$\forall \vec{p}_c \in \mathcal{P}_c \; \exists \vec{p}_h \in \mathcal{P}_h : \forall \vec{p}_t \in \mathcal{P}_t$
$|\vec{p}_c - \vec{p}_h| <= |\vec{p}_t - \vec{p}_h|$, where $\mathcal{P}_h$ - the set of hypothesis' points.

- the goal of the iteration is to find such a transformation M that minimizes the distances between the points of the hypothesis cloud, $\vec{x}_i$, and their corresponding points from the ”closest sub-set”, $\vec{y}_i$;
- the function to be minimized can be written as $F = \sum\limits_{i=1}^{N} |M\vec{x}_i - \vec{y}_i|^2$, effectively making each iteration a least squares problem.

F is called the ***”error metric”*** and represents the measure of how well the iteration has performed. In the ideal case, as the iterations progress, the error metric will converge to zero, and the hypothesis will converge to the target cloud. In general case, however, the ICP algorithm converges monotonically to the nearest local minimum [7].

Convergence itself is formally proven in the original article in the form of the *convergence theorem* [7]. The fact that the point of convergence is a local rather than the global minimum is easy to show. Obviously, the notion of global minimum infers the existence of the "correct", or "ideal" mapping $\mathcal{P}_h \rightarrow \mathcal{P}_t$; it's further obvious that such a mapping cannot be obtained by considering each of the hypothesis' points independently (which is the case when using NNS to find the correspondences), or rather can be obtained in such a manner only in the most trivial cases. If the sequence of ICP steps leads to one of these trivial cases, the point of convergence will be a global minimum; usually though it will be a local one. Physically this means that ICP will arrive at a good fit of the hypothesis into the target area, but it won't be the "ideal" fit; furthermore, the obtained fit cannot be improved by considering each of the hypothesis' points independently.

During the execution of the ICP algorithm, the first few iterations converge very quickly, resulting in large transformations, whereas later iterations converge more gradually, resulting in smaller transformations [20]. Thus, the majority of the ICP's time is spent on iterations where the transformations are relatively small (cf. Figure 1.1).

The NNS is by far the slowest part of the algorithm; its brute-force implementation has the complexity of $O(N * M)$, where N is the number of hypothesis points and M is the number of target points.

RANSAC stands for RANdom SAmple Consensus and is an iterative method to estimate parameters of a mathematical model from a set of observed data which contains outliers; the algorithm was first published by Fischler and Bolles [21].

When applied to ICP, the RANSAC approach observes the fact that only 3 point correspondences are necessary to calculate a transformation; therefore, in each ICP iteration a number of random point triplets are sampled from the hypothesis, with transformations M_i being calculated for each one of them. The iteration then proceeds to select the best transformation using the error metric F.

At a first glance, the RANSAC approach significantly diminishes the amount of computations, because only triplets of points are considered, and even a large number of triplets still represents much less computations. In reality, RANSAC is even slower the normal ICP (though more accurate). Although the amount of work needed to calculate the transformations M_i is undeniably smaller, and the "closest sub-set" for each of the triplets consists of just 3 points, RANSAC still calculates the "full closest sub-set" for the whole hypothesis; moreover, it does so not once, as in the original algorithm, but k times, where k equals the number of triplets. The reason for that is that RANSAC has to calculate the error metric F ***after applying each of the transformations*** computed for different triplets, in order to select the best one [22]. Different optimizations however exist to circumvent this problem (see ***Optimizing the NNS***).

2.1 GPICP: general description

In the traditional ICP, a hypothesis is mapped to the target cloud; for the purpose of this work, a more general case is considered: the hypothesis cloud is being transformed to match a certain "target" area of the reference cloud. The only publicly available open-source ICP implementation is a part of the "Point Cloud Library (PCL)" [10]; it will thus serve as the main reference point for this work.

The GPICP algorithm is based on the RANSAC variation of ICP and includes the following steps and optimizations:

- in order to increase the registration accuracy, GPICP tries to select a ”better” initial transformation by considering 8 combinatorial and 8 random initial rotations; the ICP algorithm itself is executed 16 times (for each of the initial transformations) (see Section 3.1);
- GPICP removes the problem of iterative searching for suitable RANSAC triplets by simply selecting more than 3 points (see Section 3.2);
- GPICP performs multiple NNS searches (see Section 3.4):
  - *pre-orientation searches* to determine point-to-point correspondences for each of the RANSAC sets (small-scale NNSs);
  - *post-orientation searches* after applying each calculated transformation (large-scale NNSs);
- transformations themselves are obtained using the unit quaternions method by Berthold K.P. Horn [23]; translation and rotation are taken into account, but not the scale (since the clouds in question are the outputs of a laser scanner that functions in the same scale space) (see Section 3.8);
- since post-RANSAC searches serve to *compare* the values of the error metric F for the calculated transformations, exact numbers are redundant here, and either $(1-e)$-approximate searches or simple depth-first searches are used to determine point-to-point correspondences (see Section 3.7);
- expanding further on the idea of exact calculations redundancy, *post-orientation searches* are performed for a random subset of hypotheses’ points (see Section 3.5).

The main focus was on increasing the algorihtm’s accuracy. However, since more accuracy means more computations, an additional effort was put into optimizing the algorithm itself, getting rid of certain calculation redundancies (see above) and achieving an optimal mapping to the graphics hardware. The following chapters (Chapter 3, chapter 4) cover these modifications in greater detail.

CHAPTER 3

Mathematical optimizations

This chapter is dedicated to mathematical optimizations of GPICP that make it faster and more accurate (e.g., getting rid of redundant calculations etc.).

3.1 Increasing the registration accuracy

As was pointed out in Section 2, ICP converges to local minimum; however, in many cases the nearest local minimum proves to be an unsatisfactory result. It is easy to see, however, that the output of the ICP algorithm depends on the hypothesis' initial transformation: if we would continuously change it to match the orientation of the ”target” area more and more closely, the ICP's output would converge to the global minimum. The task of selecting the initial transformation

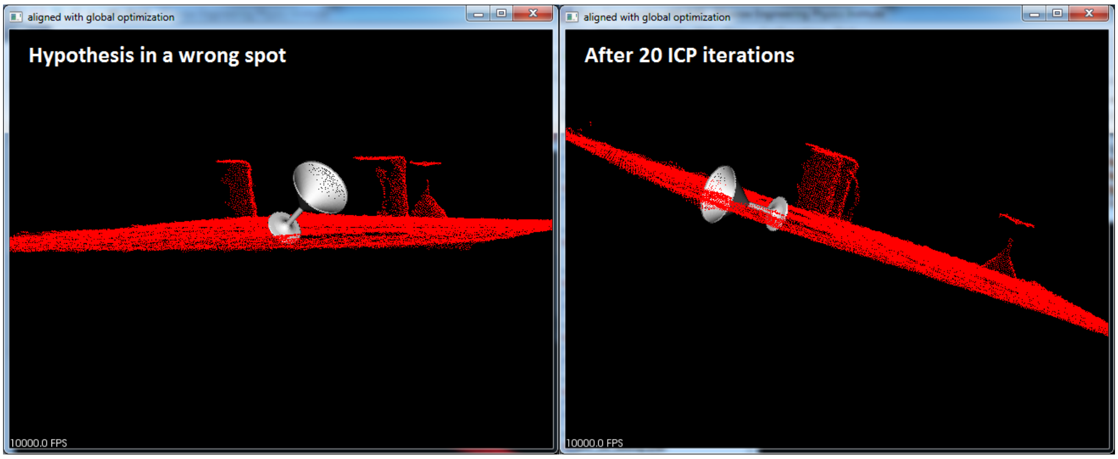

aligned with global optimization
Hypothesis in a wrong spot
10000.0 FPS
aligned with global optimization
After 20 ICP iterations
10000.0 FPS

Figure 3.1: The ”target” area is too far away from the hypothesis, the ICP algorithm fits the object into the ”most appropriate” spot in its immediate vicinity

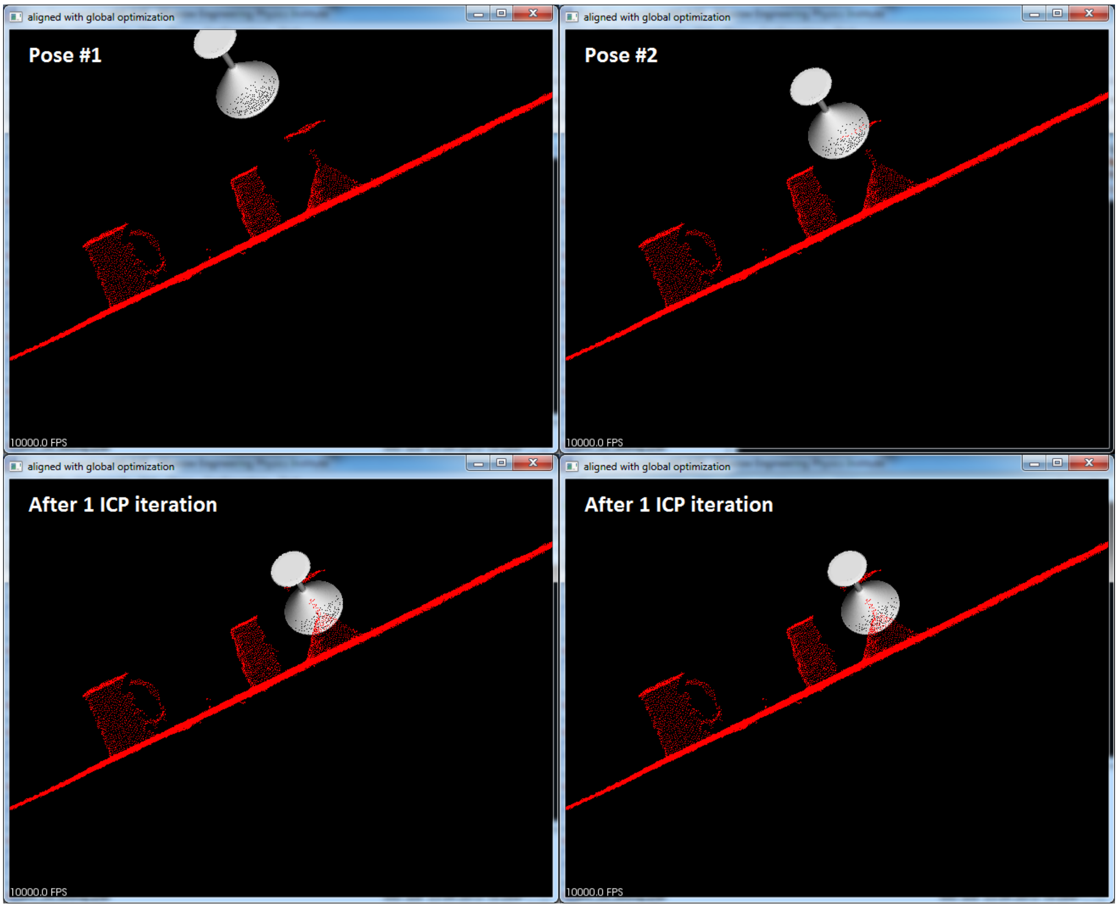

aligned with global optimization
Pose #1
10000.0 FPS
aligned with global optimization
Pose #2
10000.0 FPS
aligned with global optimization
After 1 ICP iteration
10000.0 FPS
aligned with global optimization
After 1 ICP iteration
10000.0 FPS

Figure 3.2: Two instances of a hypothesis are close enough to the ”target” area, the difference is alleviated after the first step of the ICP algorithm

can’t be solved analytically, since finding the transformation that would closely match the target area is exactly the goal of the ICP itself. It is feasible, however, to try to select a ”better” initial transformation; the ICP part can then be considered as a means of improving this selection.

When talking about the initial transformation, rotation is the only part that can really be affected, with translation being largely hit-or-miss. If the hypothesis is, for example, put into the part of the scene which is far away from the ”target” area, the ICP won’t locate the ”target” area by itself. It will instead make the hypothesis converge to the ”most appropriate” spot in its immediate vicinity (cf. Figure 3.1). Since shape recognition is an enormously costly process, it’s impossible find out analytically which part of the scene the ”target” area is and thus it’s unknown where to initially move the hypothesis to. Trying out random initial translations is fruitless since the set of potential options is non-enumerable.

If, on the other hand, the initial translation is close enough to the ”target” area of the scene, it does not matter anyway since the ICP algorithm will move the hypothesis very close to the ”target” area during the first few steps [20](cf. Figure 3.2, see Section 2).

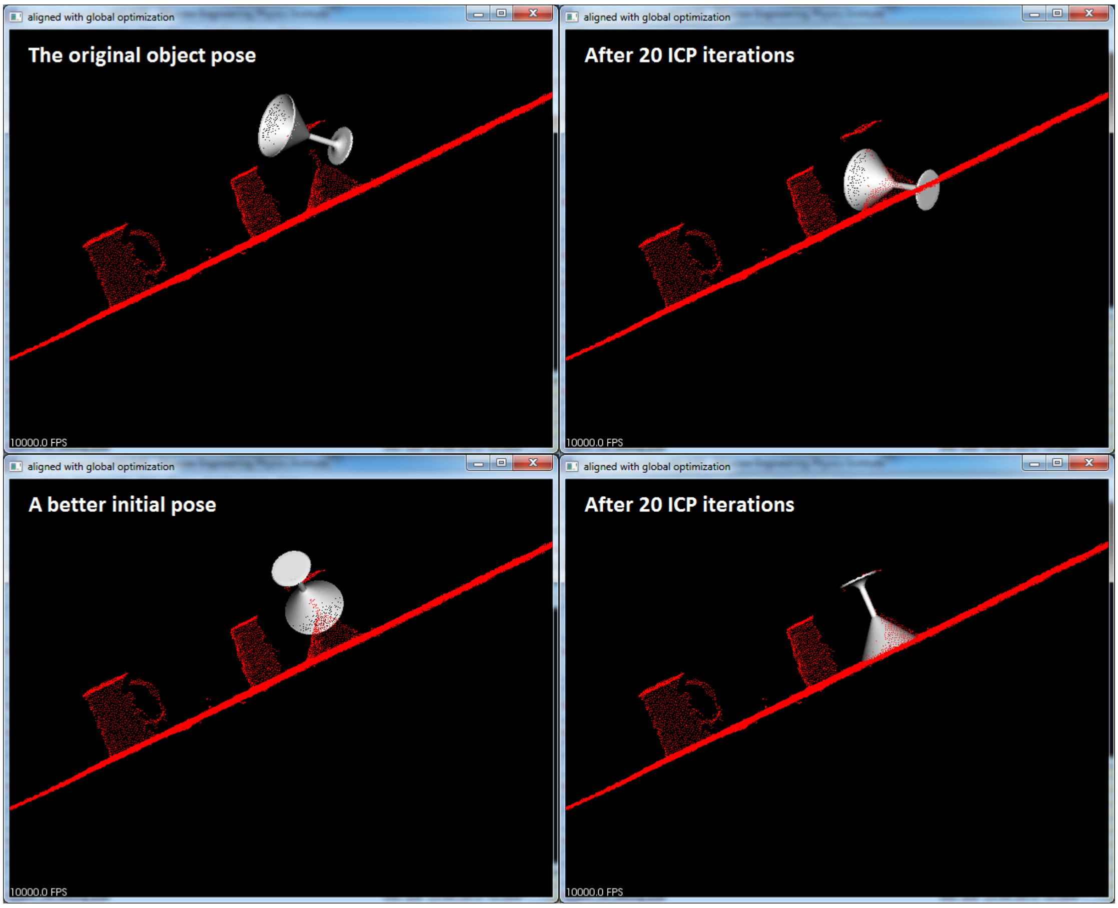


Figure 3.3: This case is easily solved by turning the liqueur glass.

Thus, as was stated above, the initial translation is largely hit-or-miss and has to be adjusted manually if needed. The initial rotation, on the other hand, is much easier to control (cf. Figure 3.3).

The discussed GPICP method proposes to select 8 combinatorial and 8 random initial rotations.

The combinatorial rotations are created in the following manner:

- for each of the 3 axes - X, Y and Z - we have 2 cases: we can either rotate the hypothesis 180 degrees around this axis, or leave it alone;
- $C^3_2 = 8$.

For reasons of symmetry, the number of random rotations is selected to mirror the number of combinatorial rotations; it can of course be increased to move the output of the ICP algorithm even closer to the global minimum. Below are experimental results showcasing dependence of the error and time on the number of initial transformations.

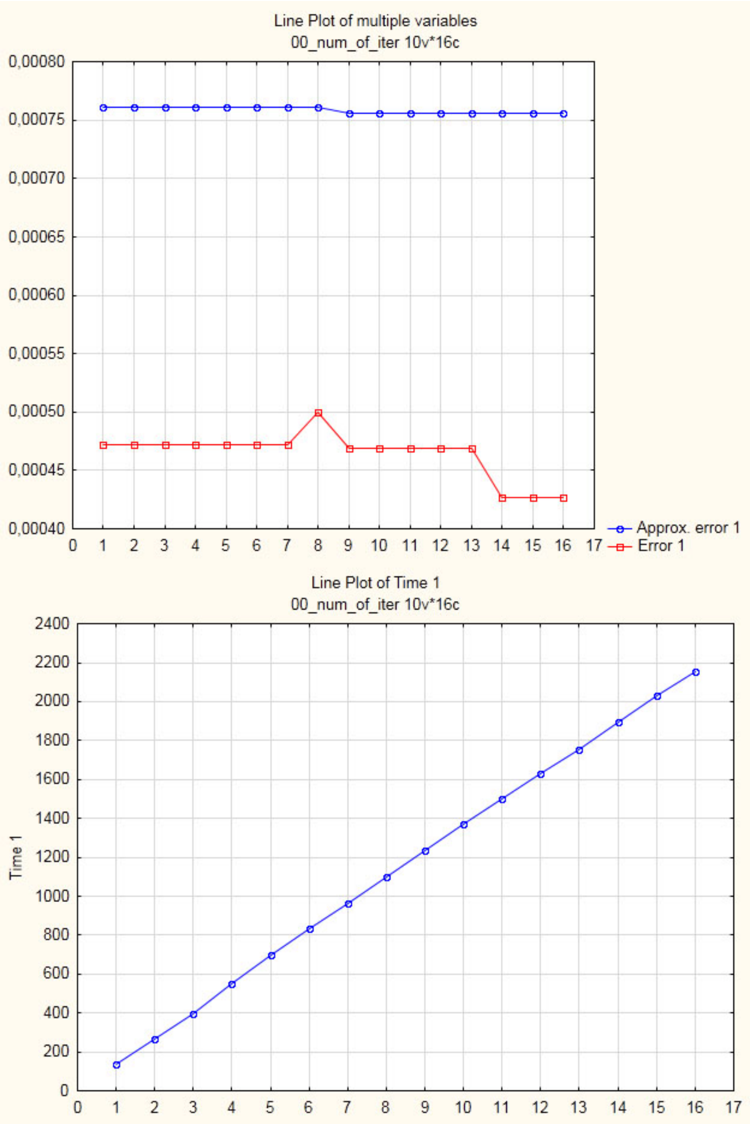


Figure 3.4: Dependence of the error on the number of initial transformations (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #1)

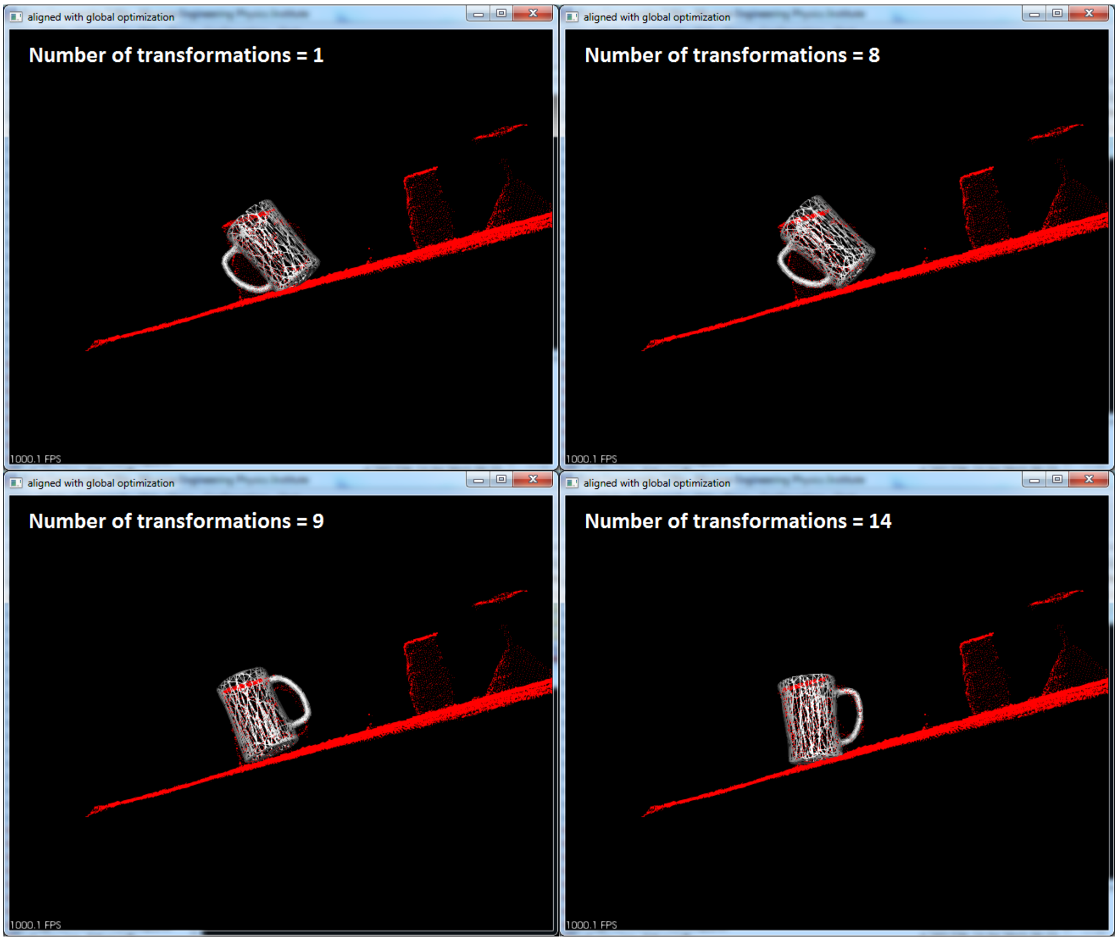

aligned with global optimization
Number of transformations = 1
1000.1 FPS
aligned with global optimization
Number of transformations = 8
1000.1 FPS
aligned with global optimization
Number of transformations = 9
1000.1 FPS
aligned with global optimization
Number of transformations = 14
1000.1 FPS

Figure 3.5: Dependence of the error on the number of initial transformations (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #1)

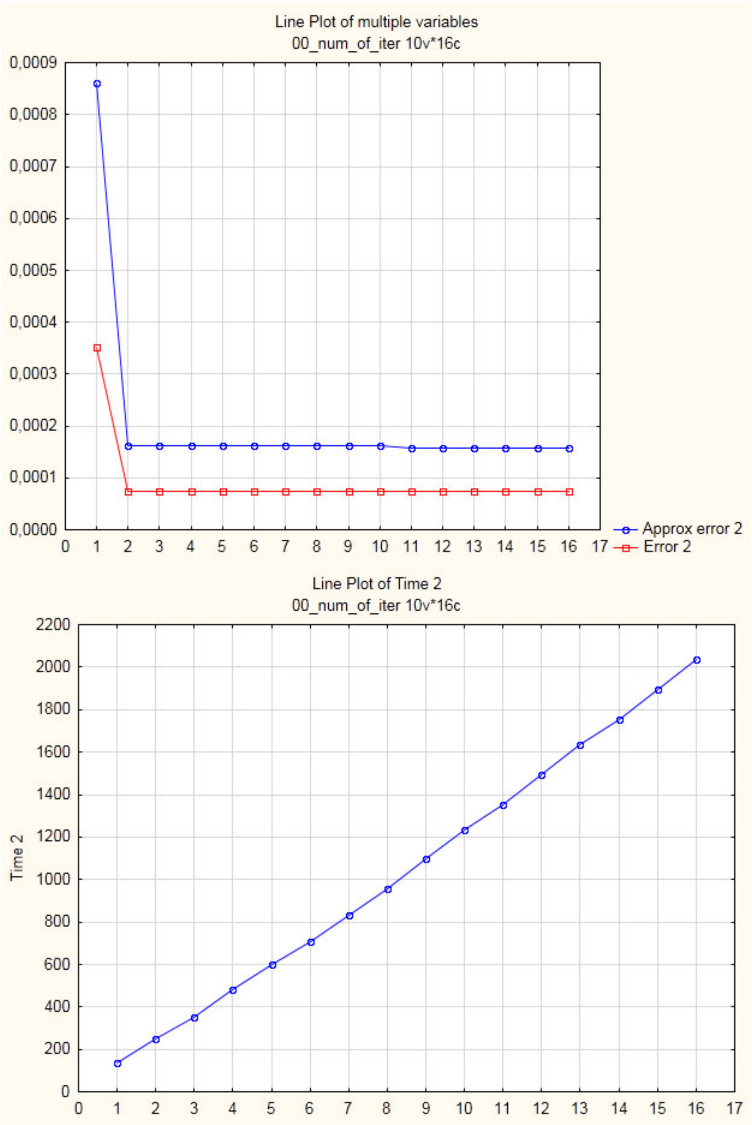


Figure 3.6: Dependence of the error on the number of initial transformations (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #2)

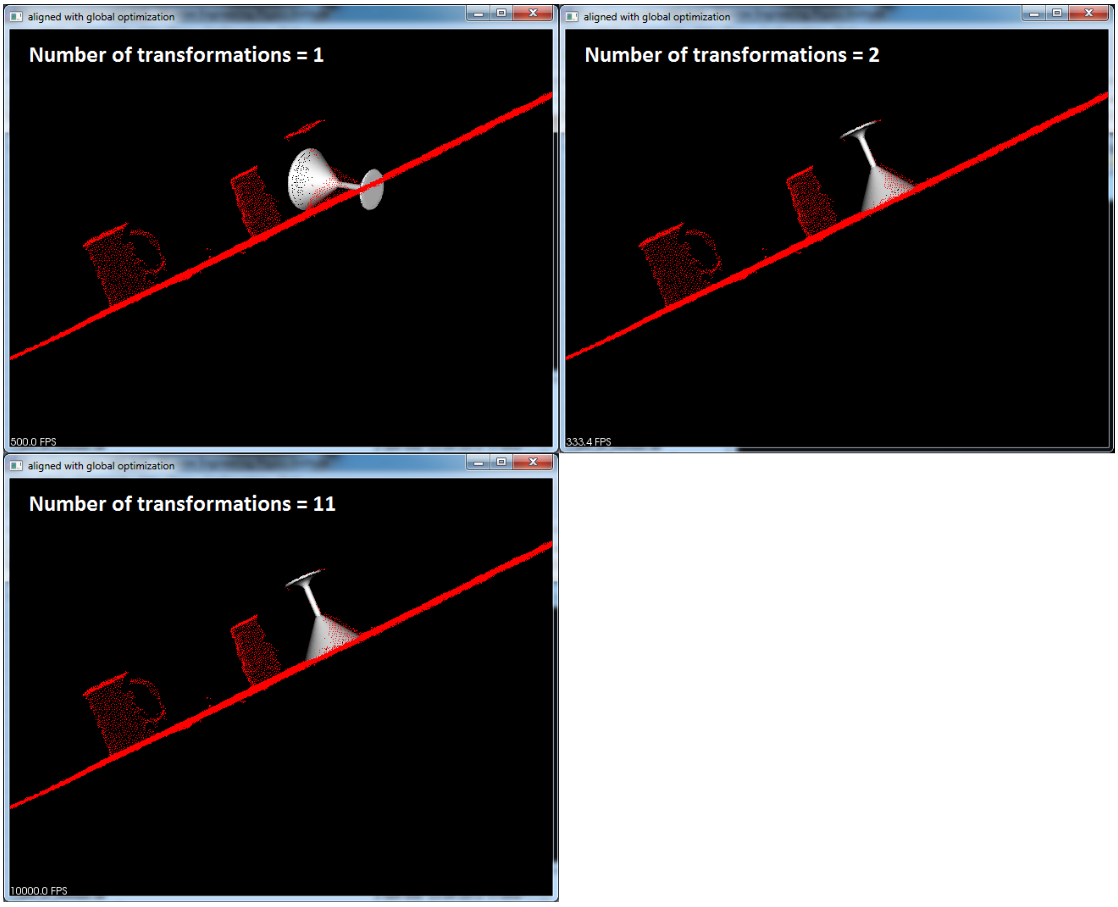

aligned with global optimization
Number of transformations = 1
500.0 FPS
aligned with global optimization
Number of transformations = 2
333.4 FPS
aligned with global optimization
Number of transformations = 11
10000.0 FPS

Figure 3.7: Dependence of the error on the number of initial transformations (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #2)

Important points:

- time grows linearly, since the same calculations are repeated for each of the initial transformations (see Figures 3.4, 3.6);
- both exact and approximate mean errors are represented by piecewise constant functions; this is explained by the fact that a drop in the error occurs only after finding a better transformation, and the new error value holds until the next better transformation (see Figures 3.4, 3.6);
- with the exception of rare fluctuations, the exact error drops together with the approximate error: of the 2 objects in question and a total of 32 initial transformation cases, only in one case did the exact error increase while the approximate one dropped - see case 8 for object #1, the mug (see Figure 3.4);
- a single initial transformation is insufficient for obtaining good results for both object #1 and #2 (see Figures 3.5, 3.7).

3.2 RANSAC set size

Application of RANSAC to ICP observes the fact that only 3 point correspondences are necessary to calculate a cloud-to-cloud transformation (see Section 2), so the usual approach is to sample a number of point triplets from the hypothesis. However, it's clear that some of the triplets will provide better results than the others; consider a degenerate case where the points of a triplet are so close to each other that the corresponding sub-set of the scene cloud consists of just one point.

The reference PCL method solves this problem by employing an iterative process:

- **preparation step #1**: a normalized 3x3 covariance matrix of the cloud points is calculated (i.e. the point cloud is effectively treated as a distribution of a 3-component random vector);
- **preparation step #2**: square roots of the covariance matrix' eigenvalues serve as estimations of axes lengths of the ellipsoid around the point cloud;
- **preparation step #3**: arithmetic mean of the these three square roots is calculated to be used as a threshold distance between any two of the sampled triplet's points;
- **iteration**: triplets are sampled iteratively until such a triplet is found that has the distance between any two points greater than the computed threshold value.

Such a process, being strictly iterative, doesn't map well to a parallel architecture; a better solution can be proposed if one observes the fact that on a CUDA GPU, mathematical operations on points are guaranteed to be executed in parallel as long as the number of points is smaller or equal to the warp size (see Section 4.1).

Thus, as long as the size of the RANSAC set is kept smaller than the warp size, selecting more than 3 points will not result in any performance hit. For the CUDA architecture-related reasons explained in Section 4.2, GPICP uses RANSAC sets of 16 points.

Below are experimental results showcasing dependence of the error and time on the RANSAC set size.

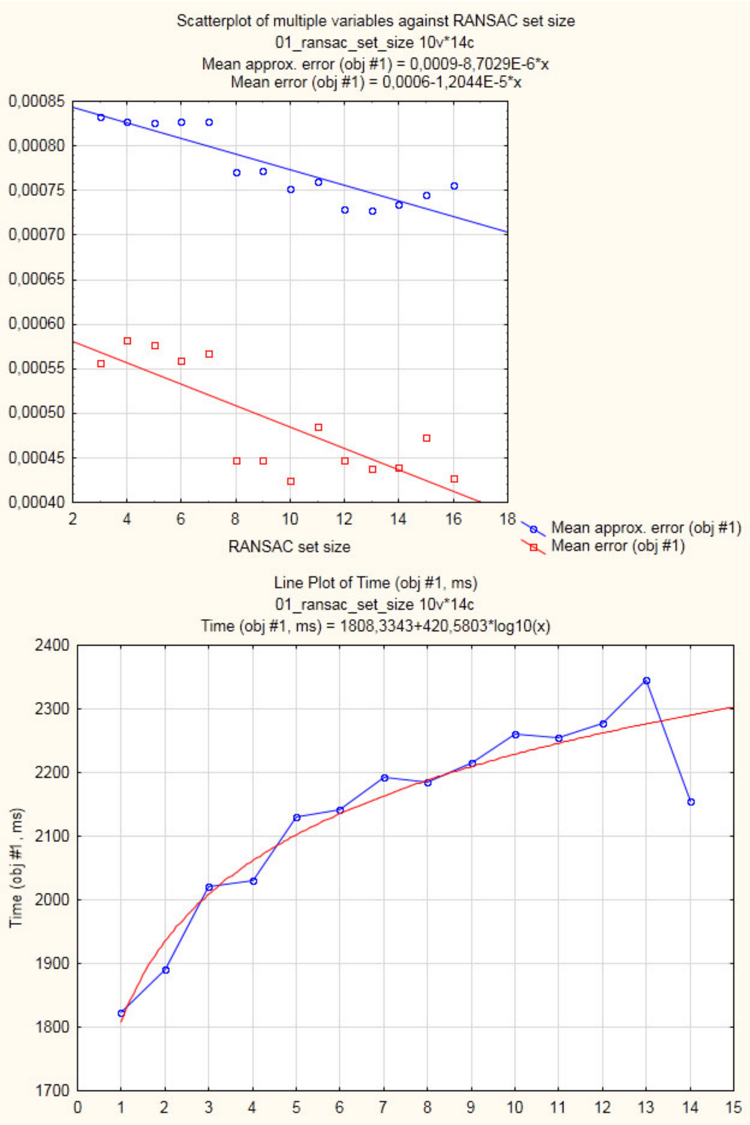
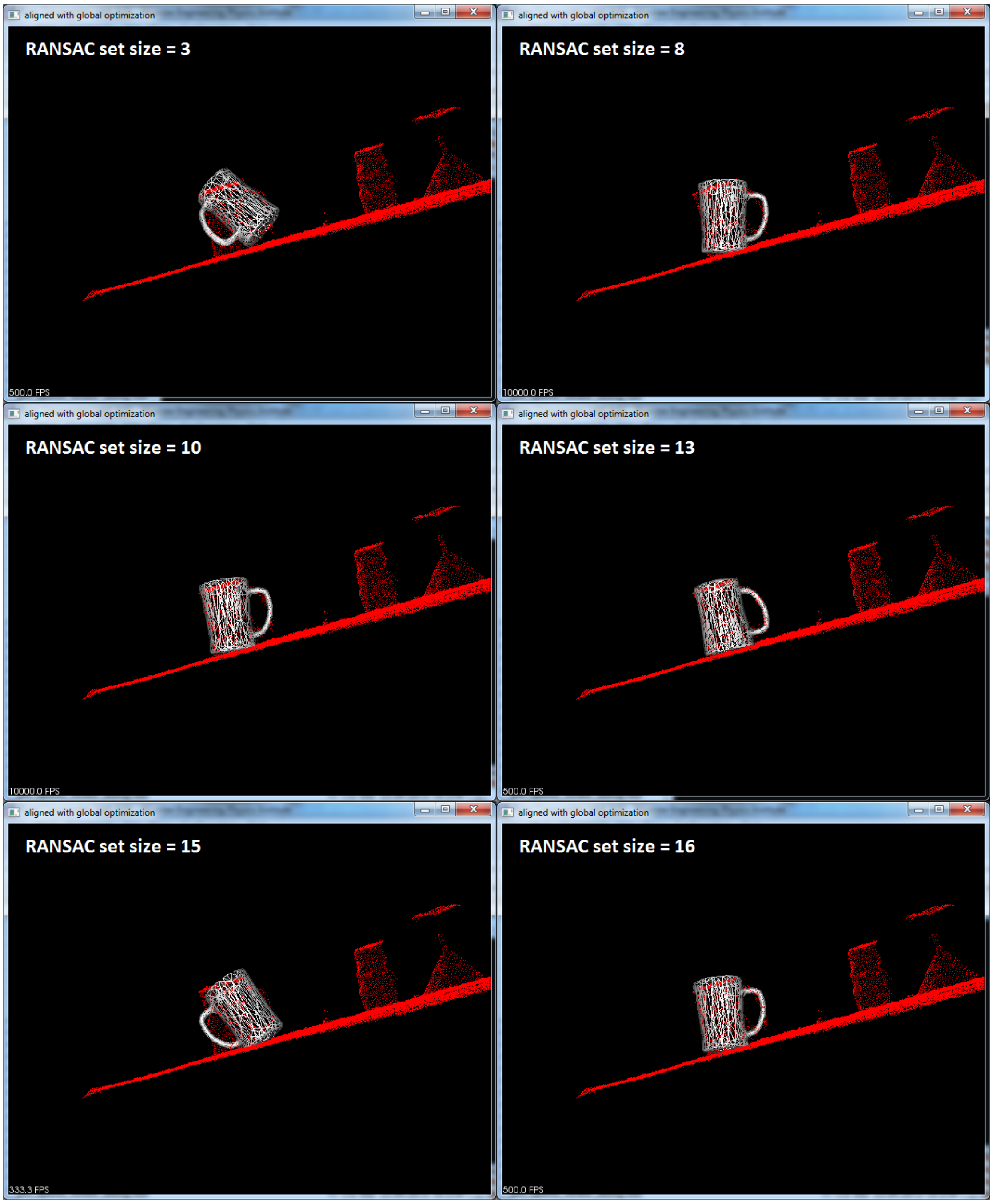


Figure 3.8: Dependence of the error on the RANSAC set size (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #1)



RANSAC set size = 3
500.0 FPS

RANSAC set size = 8
10000.0 FPS

RANSAC set size = 10
10000.0 FPS

RANSAC set size = 13
500.0 FPS

RANSAC set size = 15
333.3 FPS

RANSAC set size = 16
500.0 FPS

Figure 3.9: Dependence of the error on the RANSAC set size (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #1)

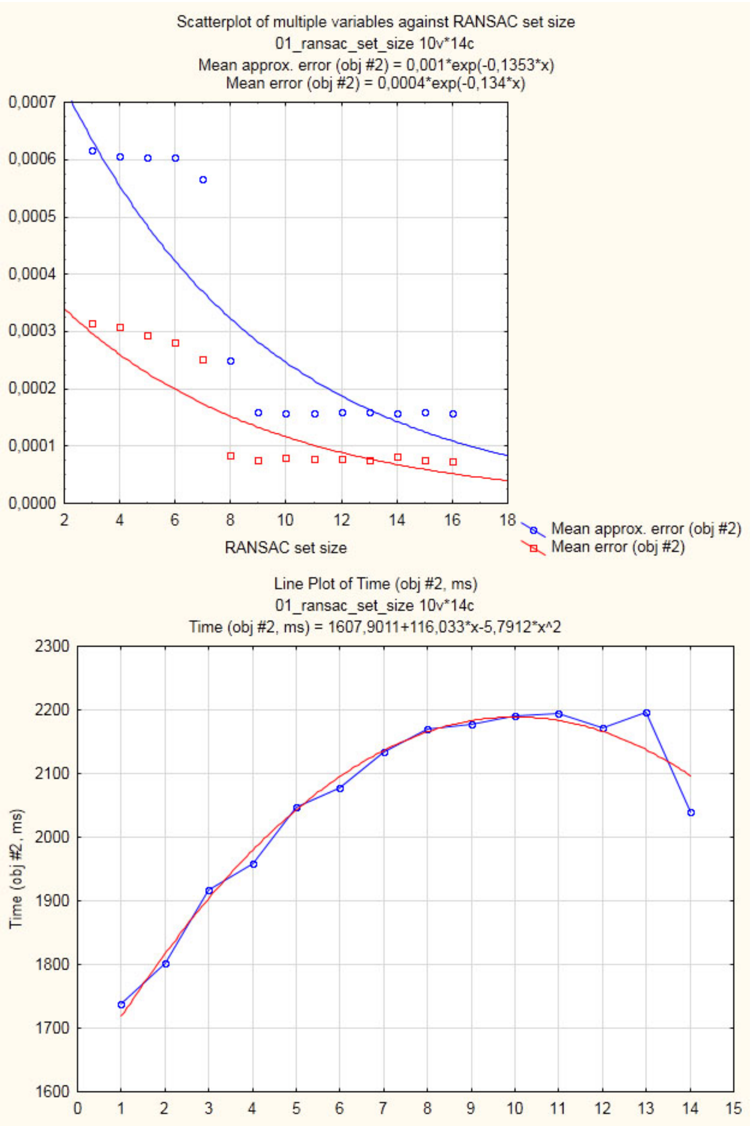


Figure 3.10: Dependence of the error on the RANSAC set size (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #2)

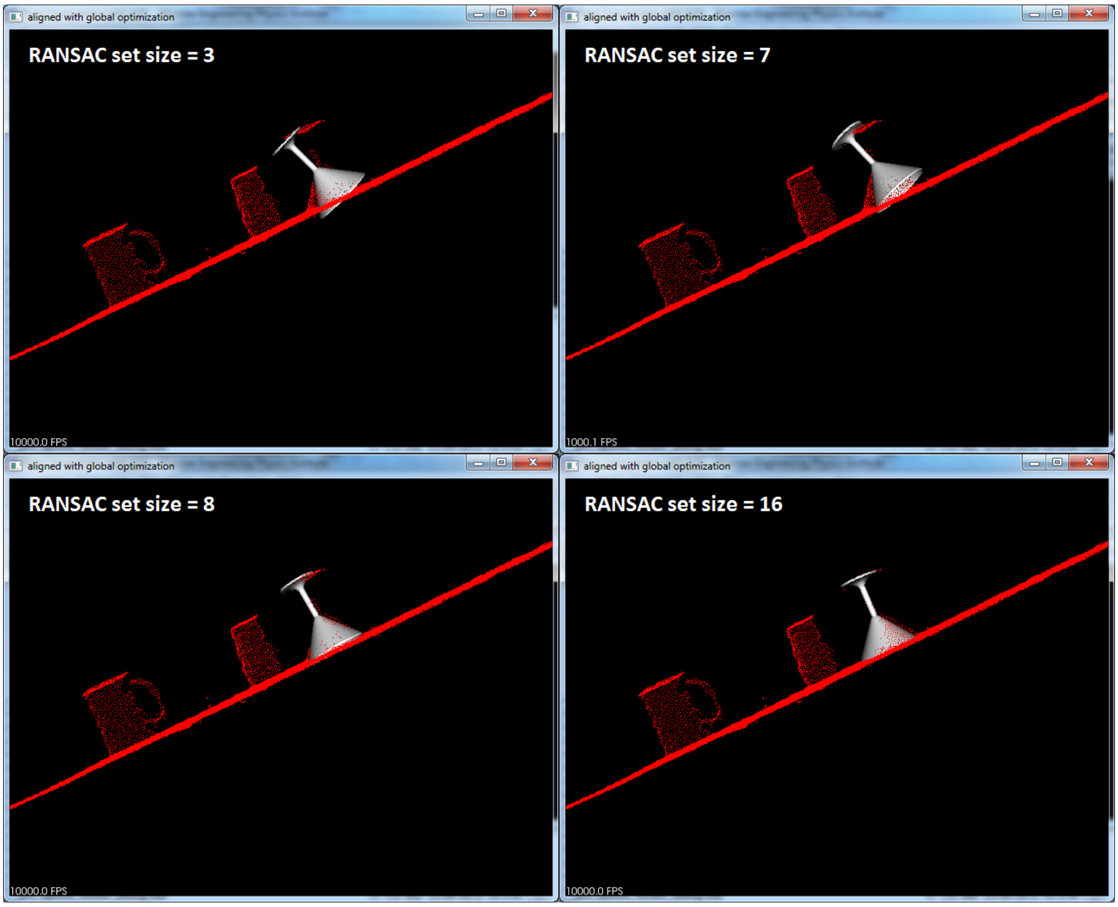

aligned with global optimization
RANSAC set size = 3
10000.0 FPS
aligned with global optimization
RANSAC set size = 7
1000.1 FPS
aligned with global optimization
RANSAC set size = 8
10000.0 FPS
aligned with global optimization
RANSAC set size = 16
10000.0 FPS

Figure 3.11: Dependence of the error on the RANSAC set size (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #2)

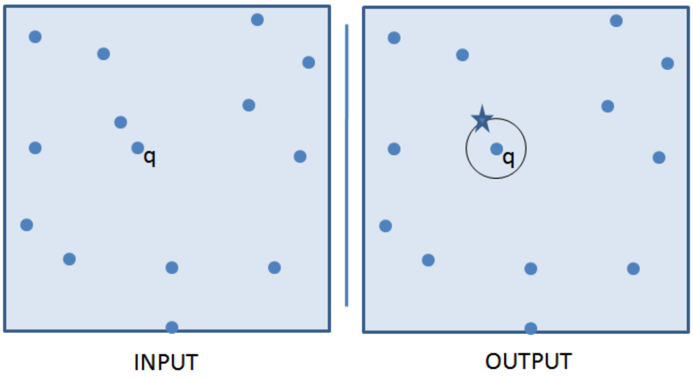


Figure 3.12: NNS

Important points:

- at least for one of the objects the trend of both mean errors (exact and approximate) is a linear function: the more points there are in a RANSAC set, the better the results (see Figure 3.8);
- as the size of a RANSAC set grows, the algorithm's time grows logarithmically, meaning that after a certain set size, the addition of new points has no significant effect on computation time (see Figures 3.8, 3.9, 3.10, 3.11);
- both cases show that the best time/error ratio occurs when the RANSAC set size equals 16, i.e. is aligned with the warp size; the reason for that is the drop in branch divergence (see Figures 3.8, 3.10 and Section 4.2).

3.3 NNS

Let $\mathcal{P}$ be a set of n points in a 3-dimensional space, and let $\vec{q}$ be a query point. A statement of the Nearest Neighbour Problem is (cf. Figure 3.12): find the point $\vec{p}_c$ in $\mathcal{P}$ which has the minimum distance from $\vec{q}$, i.e.

$|\vec{q} - \vec{p}_c| <= |\vec{q} - \vec{p}_i| \; \forall \vec{p}_i \in \mathcal{P}$

3.4 Optimizing the NNS

An ICP iteration proceeds in three principal steps:

- the ***pre-orientation step*** focuses on obtaining the RANSAC point sets and their ”closest sub-sets”;
- the ***orientation step*** focuses on calculating transformations from the former to the latter;

- the ***post-orientation step*** deals with determining which of the calculated transformations is the best one.

Obviously, point-to-point correspondences are needed in all the three steps:

- to create the ”closest sub-sets” for the RANSAC sets, and later to calculate the corresponding transformations;
- to measure how well the transformed RANSAC points coincide with the points of the ”target” area.

The reference PCL ICP implementation contains a single NNS per ICP iteration, which takes place before the iteration starts [10]. Thus, inside one iteration, the same computed point-to-point correspondences are used in all the three steps.

A more precise *multiple NNS* approach performs full NNSs on the hypothesis *after* applying each of the transformations, and then calculates the corresponding squared errors [22]. The approach taken in PCL, however, is still viable because point-to-point correspondences aren’t likely to be significantly altered after only a single ICP step, since each step represents a progressively smaller change in the hypothesis’ orientation [7]. The only exception from this rule are the first few steps where transformations are relatively large [20].

The PCL approach is obviously faster; however, it contains two major drawbacks:

- a GPU-specific drawback concerning writes to global memory: obviously, in order to be used multiple times, the point-to-point correspondences have to be saved somewhere; in general case, the amount of data is too large for local or shared memory, so it goes to global memory, prompting resource-costly global memory writes (and, later, related reads); the alternative *multiple NNS* approach, on the other hand, uses the point-to-point correspondences on the spot and doesn’t involve any global memory writes (see Section 4.1 for details on memory available to CUDA applications);
- the single NNS approach implies the same level of accuracy for orientation and post-orientation calculations; however, it will be shown in Section 3.6 that we prefer to use exact point-to-point correspondences for the former, while for error calculations approximate correspondences are sufficient.

Another important observation that will be discussed later in Section 3.5 is that it’s not necessary to use *all* the points of the hypothesis to compare the errors associated with transformations obtained for different RANSAC sets, making pre-orientation searches even more different from the post-orientation ones.

Thus, the GPICP method proposes the following approach to the NNS problem:

- perform separate pre-orientation and post-orientation NNSs;
- use exact point searches for each of the RANSAC sets; the amount of exact searches is thus equal to the number of RANSAC sets multiplied by the number of points in each set (in our implementation, the total number is 512, see Section 4.2);

- after calculating the transformations, for each one of them perform approximate point searches on the reduced sets of hypothesis points.

3.5 Reducing the set of hypothesis' points

Obviously, using multiple NNSs to determine the best transformation introduces a significant performance penalty. Furthermore, if the number of RANSAC sets is variable and serves as an input parameter for the algorithm, time complexity is increased, since the number of searches depends on the number of RANSAC sets.

However, we have to consider the fact that all the large-scale NNSs are used to calculate the error metric, values of which are then compared in order to select the best transformation. Since we're not interested in exact numbers here, we can just as well use only a portion of hypothesis' points for error metric calculation.

GPICP reduces the set of hypothesis points to a randomly samples set of power

$\frac{N}{k}$, where
N - the number of points in the hypothesis;
k - the number of RANSAC sets.

In this case, time complexity is kept constant since the total number of closest point searches for all of the RANSAC sets equals the number of searches for the complete set of hypothesis' points.

Below are experimental results showcasing dependence of the error and time on percentage of hypothesis' points used in post-orientation NNSs.

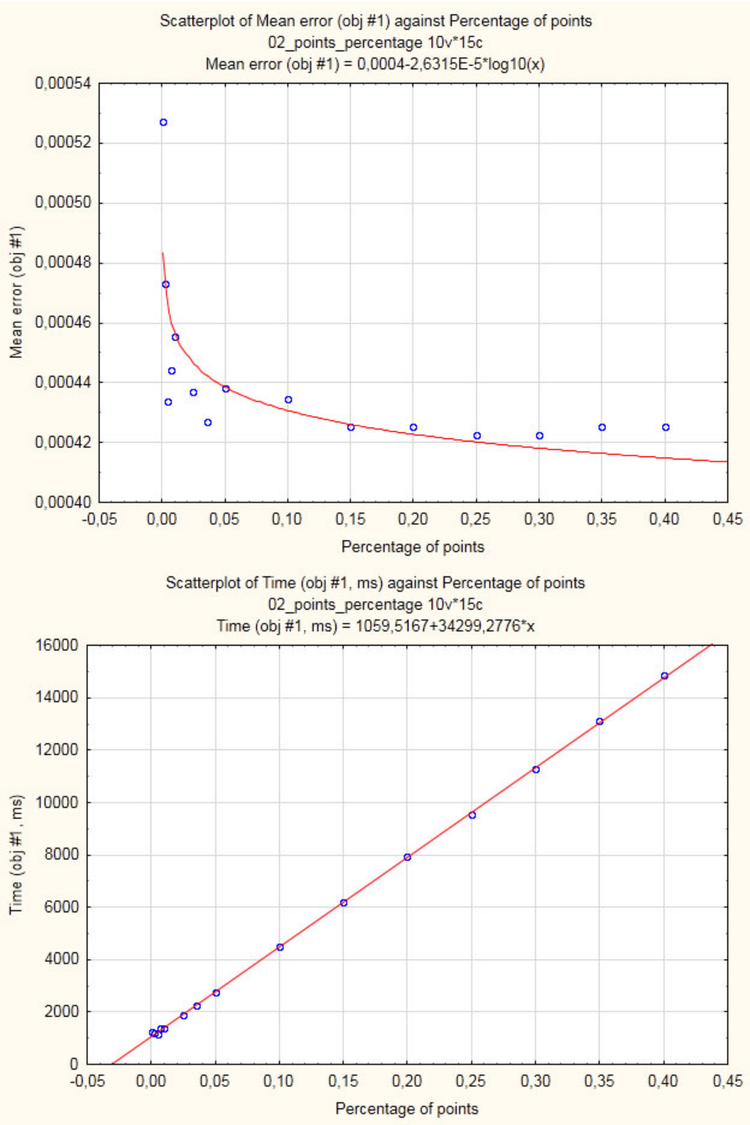


Figure 3.13: Dependence of the error on percentage of hypothesis' points used in post-orientation NNSs (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #1)

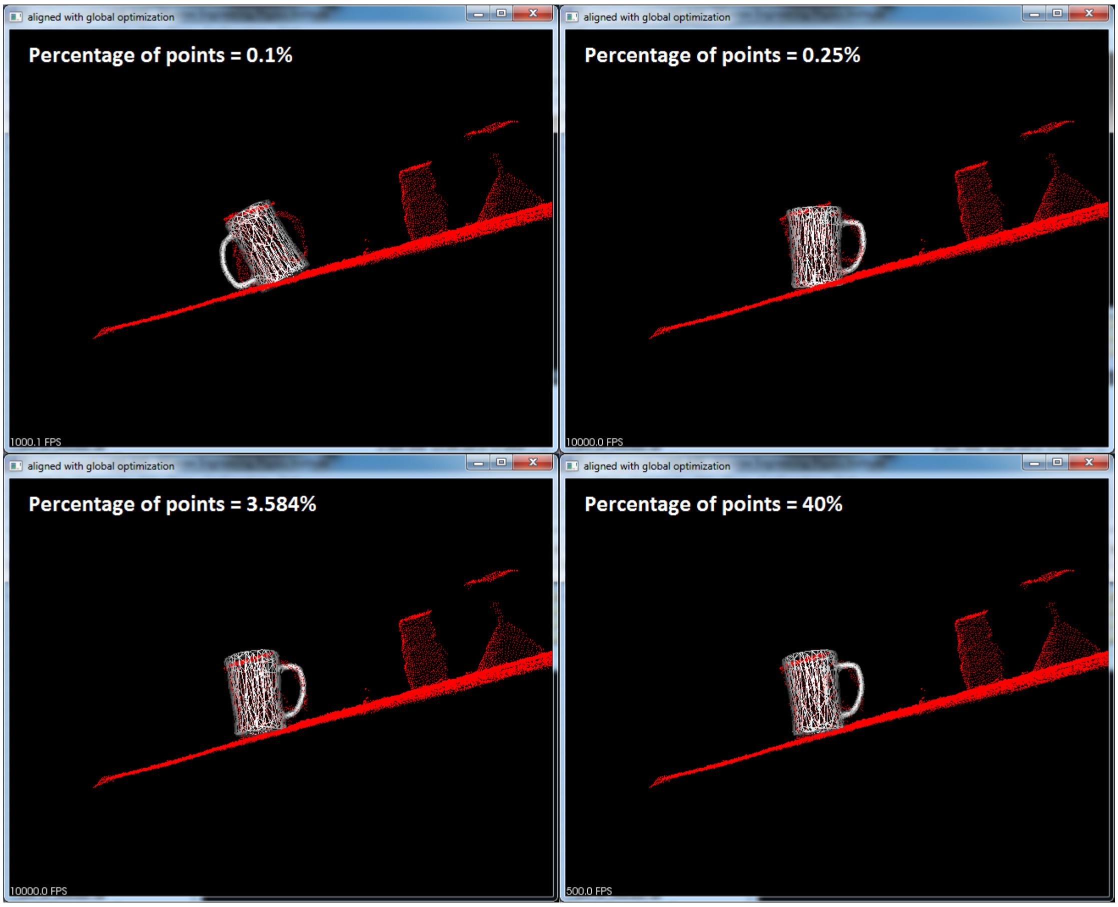

aligned with global optimization
Percentage of points = 0.1%
1000.1 FPS
aligned with global optimization
Percentage of points = 0.25%
10000.0 FPS
aligned with global optimization
Percentage of points = 3.584%
10000.0 FPS
aligned with global optimization
Percentage of points = 40%
500.0 FPS

Figure 3.14: Dependence of the error on percentage of hypothesis' points used in post-orientation NNSs (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #1)

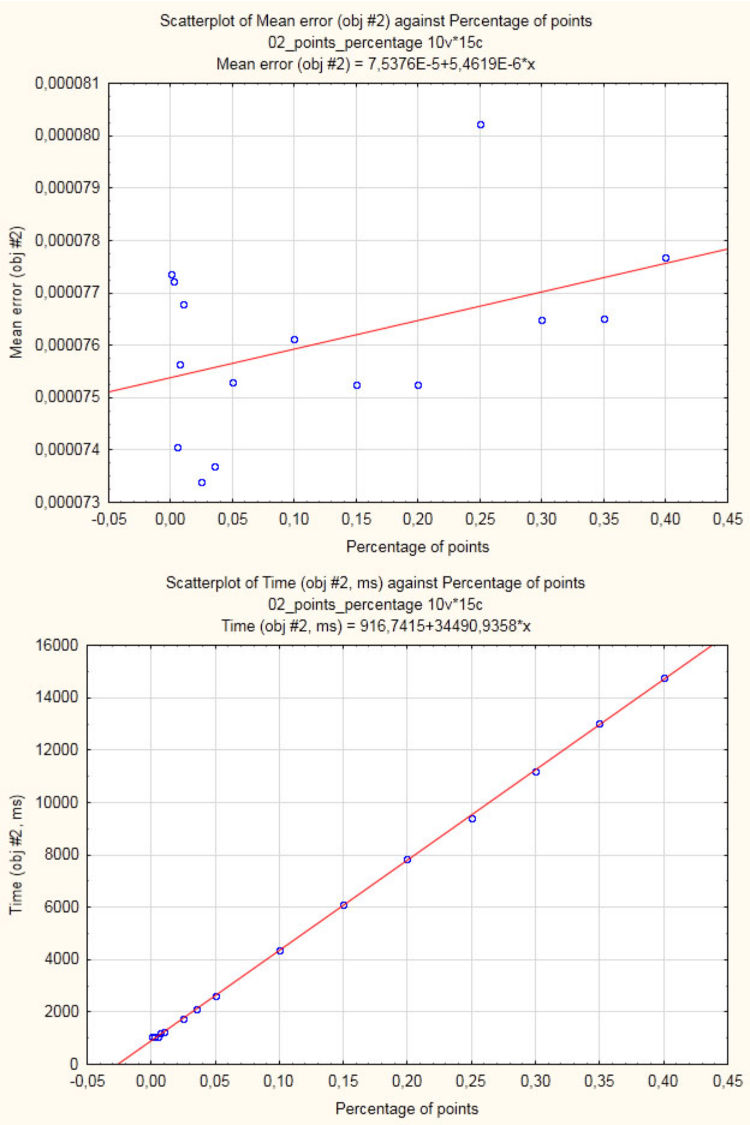


Figure 3.15: Dependence of the error on percentage of hypothesis' points used in post-orientation NNSs (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #2)

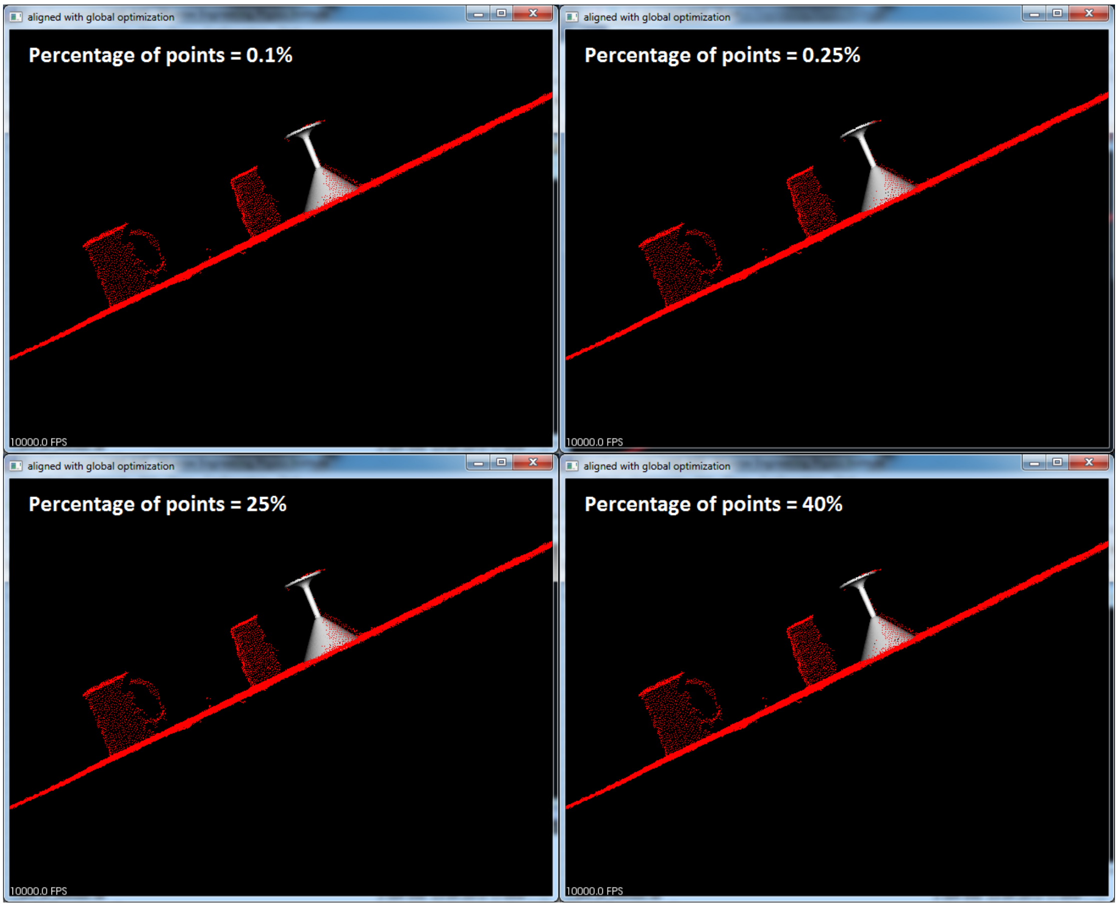

aligned with global optimization
Percentage of points = 0.1%
10000.0 FPS
aligned with global optimization
Percentage of points = 0.25%
10000.0 FPS
aligned with global optimization
Percentage of points = 25%
10000.0 FPS
aligned with global optimization
Percentage of points = 40%
10000.0 FPS

Figure 3.16: Dependence of the error on percentage of hypothesis' points used in post-orientation NNSs (100 000 points, 20 ICP iterations per pose, depth-first NNS) (object #2)

Important points:

- as the size of the post-orientation hypothesis set increases, the algorithm's time grows linearly, since the main bulk of computations is spent on post-orientation NNSs (see Figures 3.13, 3.15);
- in the best case, the exact mean error drops logarithmically, meaning that after reaching a certain size of the post-orientation hypothesis set, its further increase leads either to an insignificant drop in the error (object #1), or even to its increase (object #2) (see Figures 3.13, 3.14, 3.15, 3.16);
- the ideal percentage of hypothesis' points for post-orientation NNSs, as used in GPICP, is 3,548$\%$;
- the approximate mean error is omitted here, since it's calculated for the points of the post-orientation set, i.e. depends on this set's size.

3.6 Exact vs. approximate NNS

The quality of the orientation step's output (i.e. the transformation matrices) is directly linked to NNS precision. We can of course use approximate search methods here, but that would mean that transformation matrices would be calculated for possibly incorrect point-to-point correspondences, i.e. the output itself would be directly affected.

The post-orientation step, on the other hand, has no calculated output and focuses on determining the best transformation, thus being relative by nature. Here, we're not interested in exact numbers, but rather in comparing the error metric values for the calculated transformations. The goal is to find such a transformation $M' \in \mathcal{M}$ (where $\mathcal{M}$ is a set of calculated transformations) that

$\forall M \in \mathcal{M}$
$\sum\limits_{i=1}^{N} |M'\vec{x}_i - \vec{y}_i|^2 <= \sum\limits_{i=1}^{N} |M\vec{x}_i - \vec{y}_i|^2$, where
$\vec{x}_i \in \mathcal{X}$;
$\vec{y}_i \in \mathcal{Y}$;
$\mathcal{X}$ - the set of hypothesis cloud's points;
$\mathcal{Y}$ - the corresponding "closest sub-set";
N - the number of hypothesis' points.

A further observation regarding the approximate NNS: as the ICP progresses, the transformation gets "better", and the results of the approximate search get closer to the results of the exact search. Let's consider a degenerate case, when a non-trivial transformation M exists that maps the hypothesis cloud exactly to the target area:

$\forall \vec{x} \in \mathcal{X}\ \exists \vec{y}_c \in \mathcal{Y}$:
$|\vec{x} - \vec{y}_c| <= |\vec{x} - \vec{y}|\ \forall \vec{y} \in \mathcal{Y}$ and

$M\vec{x} = \vec{y}_c$, where
$\mathcal{X}$ - the set of hypothesis cloud's points;
$\mathcal{Y}$ - the corresponding "closest sub-set".

In this case, it doesn't matter if an exact or an approximate search is used – results would be the same. Since each ICP step represents a progressively smaller change in the hypothesis' orientation [7], and since the first few iterations converge very quickly [20], for most of the ICP steps, "approximate" search results are actually quite accurate.

Thus, to determine the best transformation, GPICP performs approximate NNSs.

3.7 Approximate and depth-first searches

A multitude of solutions exist to the NNS problem, with k-d tree likely being the most widely used [20]; in their original paper Besl and McKay [7] also suggested the use of k-d trees. The PCL approach, however, utilizes an octree for its NNS.

Regardless of the type of the tree, the search itself can be broken down into 2 parts:

- depth-first search, where we go strictly down the tree to the leaf the query point falls into; this leaf is the first logical candidate to contain the nearest neighbour;
- if a possibility exists that there are some points closer than the point from the depth-first node, we have to perform backtracking by going to the depth-first node's siblings, then to its parent's siblings etc.

Backtracking is by far the most costly part of the algorithm, and to be properly implemented, it requires one of the four following approaches:

- using recursive functions;
- utilizing a stack containing previously visited nodes;
- storing a pointer to the parent node in its children;
- using a special tree, e.g. a left-balanced binary tree, where the index of the parent can be calculated from the child's index.

It will be shown in Section 4.4 that at least the first two approaches prove to be quite costly when applied to the GPU architecture.

To reduce the amount of backtracking, two tests are usually employed: the ball-within-bounds test, or BWB, and bounds-overlap-ball test, or BOB [24]. The BWB test serves to check if we need backtracking at all (cf. Figure 3.17): after having located the closest point in the depth-first node, we define a sphere with the radius equal to the distance from the found point to the query point, centred at the query point. Obviously, if there's any point closer to the query point, it should fall into this sphere. Thus, if the sphere is completely contained within the depth-first node, no backtracking is necessary.

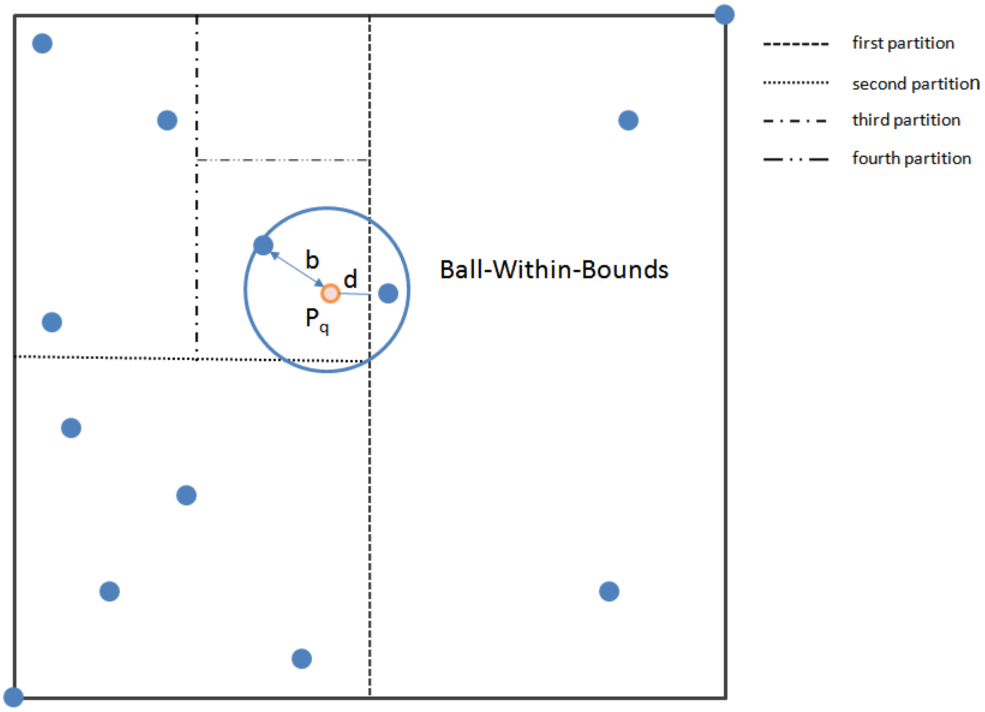


Figure 3.17: Backtracking inside a k-d tree. If the query consists of point $\vec{p}_q$, k-d tree search has to backtrack to the tree root to find the closest point [25]

This same sphere is used in the BOB test. To check if we need to go into a certain region of the tree, the sphere is intersected with that region. With each new closest point, the sphere's radius grows smaller.

There's a number of factors that further influence the amount of backtracking:

- the size of the tree's leaves;
- the tree's arity, i.e. the more children a parent node has, the more nodes should be considered for backtracking.

GPICP utilizes a left-balanced k-d tree, which is effectively a binary tree.

Recently a lot of researches have been using approximate k-d tree search for ICP algorithms [24, 25].

S. Arya and D. Mount introduce the following notion for approximating the nearest neighbour in k-d trees [26]: given a neighbourhood $\epsilon > 0$, the point $\vec{p}$ is the $(1+\epsilon)$-approximate nearest neighbour of the query point $\vec{p}_q$, if and only if $|\vec{p} - \vec{p}_q| <= (1+\epsilon)|\vec{p}_n - \vec{p}_q|$, where $\vec{p}_n$ denotes the true nearest neighbour, i.e. $\vec{p}$ has a maximal distance of ϵ to the true nearest neighbour. Using this notation, in every step the algorithm records the closest point $\vec{p}$. The search terminates if the distance to the unanalysed leaves is larger than $|\vec{p}_q - \vec{p}|/(1+\epsilon)$. Figure 3.18 shows an example where the grey cell does not need to be analysed, since the point $\vec{p}$ satisfies the approximation criterion.

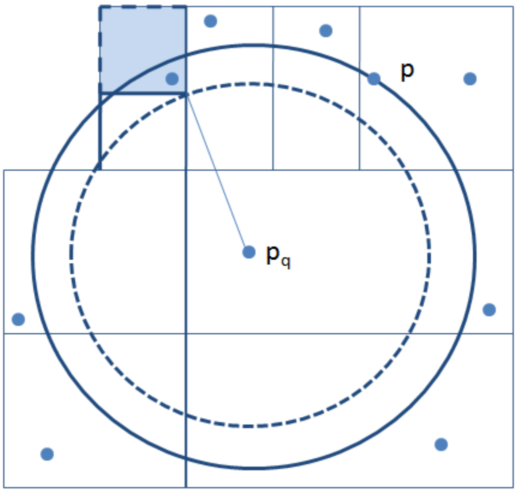


Figure 3.18: The $(1 + \epsilon)$-approximate nearest neighbour. The solid circle denotes the ϵ environment of $\vec{p}_q$. The search algorithm doesn't need to analyse the grey cell, since $\vec{p}$ satisfies the approximation criterion [25]

Some have shown that in many cases it is sufficient to use the depth-first search, thus avoiding the expensive BOB test and backtracking [24].

The PCL approach uses the $(1 + \epsilon)$-approximate method for all the point searches, even when determining point-to-point correspondences for calculating the transformations. In contrast, GPICP utilizes approximate point searches only for error metric calculations; there's an option to use the $(1 + \epsilon)$-approximate search, but the method works well with the depth-first search.

Below are experimental results showcasing dependence of the error and time on the NNS epsilon neighbourhood size.

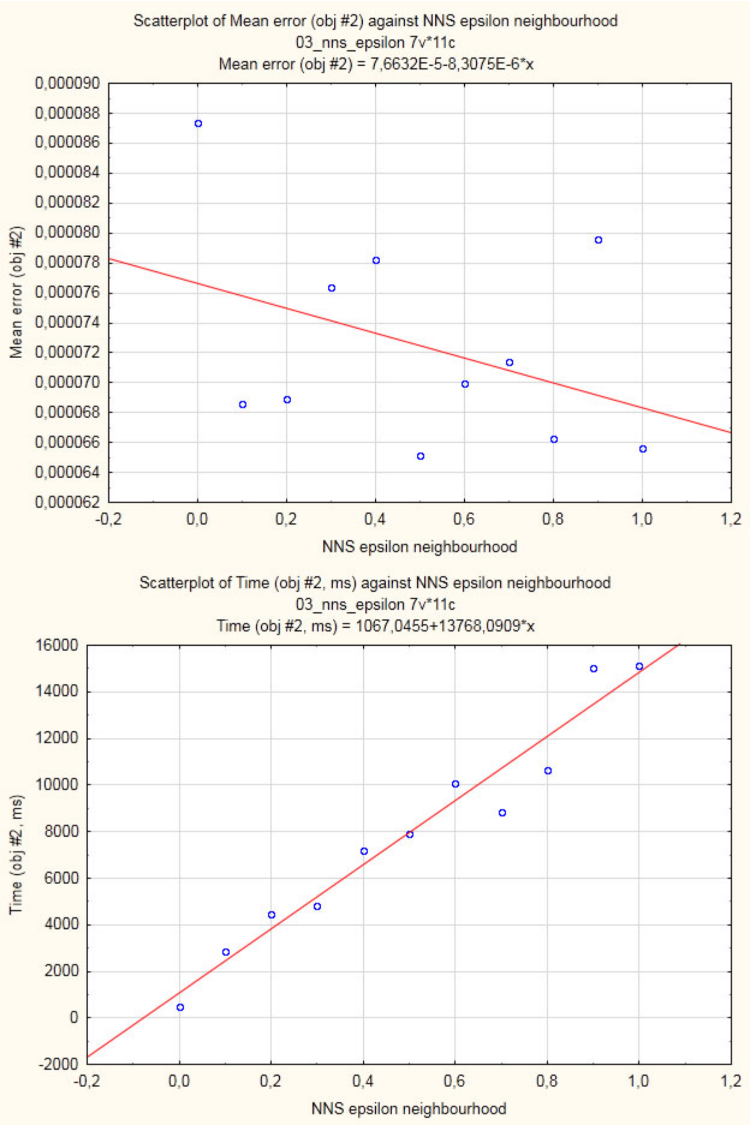


Figure 3.19: Dependence of the error on the NNS epsilon neighbourhood (1000 points, various number of ICP iterations per pose) (object #2)

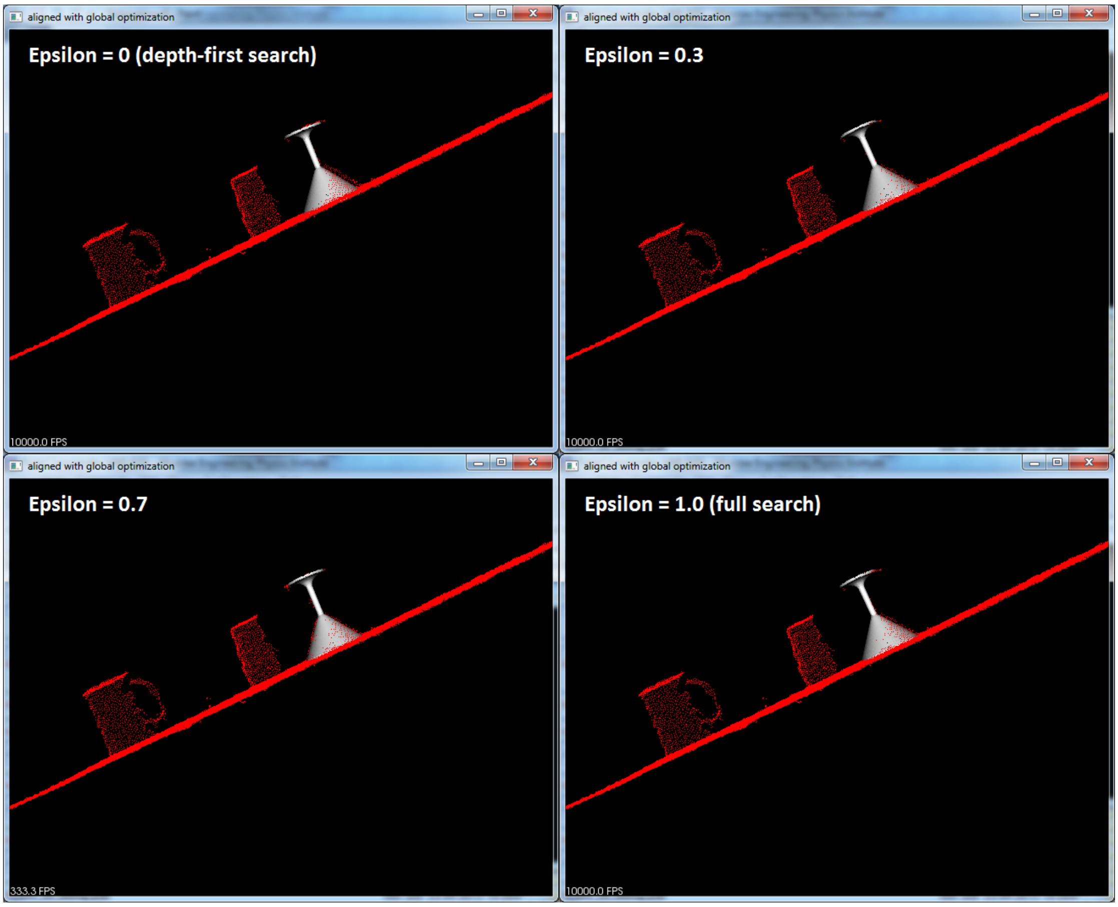

aligned with global optimization
Epsilon = 0 (depth-first search)
10000.0 FPS
aligned with global optimization
Epsilon = 0.3
10000.0 FPS
aligned with global optimization
Epsilon = 0.7
333.3 FPS
aligned with global optimization
Epsilon = 1.0 (full search)
10000.0 FPS

Figure 3.20: Dependence of the error on the NNS epsilon neighbourhood (1000 points, various number of ICP iterations per pose) (object #2)

Important points:

- time grows linearly here, i.e. the bigger the epsilon area, the longer the search takes, and the speed of the time's growth doesn't drop (see Figure 3.19);
- on the other hand, the drop in both the exact and the approximate mean errors is represented by linear functions with a small slope; the depth-first search results aren't significantly worse than that of the more exact searches (see Figures 3.19, 3.20).

3.8 Finding the transformations

The sought-after transformation can be represented by the following equation:

$\vec{a}_i = R\vec{h}_i + \vec{t}$, where
$i \in [1..n]$, n is the number of the hypothesis' points;
$\vec{h}_i$ - a point of the hypothesis;
$\vec{a}_i$ - the corresponding target area's point;
R - rotation matrix;
$\vec{t}$ - translation vector.

Note again that the scale is not taken into account, since the clouds in question are the outputs of a laser scanner that functions in the same scale space.

The error then takes the form of

$\vec{e}_i = \vec{a}_i - R\vec{h}_i - \vec{t}$, and the function to be minimized is
$F = \sum\limits_{i=1}^{n} |\vec{e}_i|^2$.

Let us define the centroid for the hypothesis, $\vec{h}_c = \frac{1}{n}\sum\limits_{i=1}^{n} \vec{h}_i$, and the target area's centroid, $\vec{a}_c = \frac{1}{n}\sum\limits_{i=1}^{n} \vec{a}_i$.

Let us further rewrite the error term as

$\vec{e}_i = \vec{a}'_i - R\vec{h}'_i - (\vec{t} - \vec{a}_c + R\vec{h}_c)$, where
$\vec{a}'_i = \vec{a}_i - \vec{a}_c$;
$\vec{h}'_i = \vec{h}_i - \vec{h}_c$.

Let us simplify the above formula:
$\vec{t}' = \vec{t} - \vec{a}_c + R\vec{h}_c$;
$\vec{e}_i = \vec{a}'_i - R\vec{h}'_i - \vec{t}'$.

Then

$$F = \sum_{i=1}^{n} |\vec{e}_i|^2 = \sum_{i=1}^{n} |\vec{a}'_i - R\vec{h}'_i - \vec{t}'|^2 = \sum_{i=1}^{n} |\vec{a}'_i - R\vec{h}'_i|^2 - 2\vec{t}' * \sum_{i=1}^{n}(\vec{a}'_i - R\vec{h}'_i) + n|\vec{t}'|^2.$$

The middle term is zero since $\sum\limits_{i=1}^{n} \vec{a}_i^{\prime} = 0$ and $\sum\limits_{i=1}^{n} \vec{h}_i^{\prime} = 0$, so

$$F = \sum_{i=1}^{n} |\vec{a}_i^{\prime} - R\vec{h}_i^{\prime}|^2 + n|\vec{t}^{\prime}|^2.$$

Since the first term does not depend on $\vec{t}$ or $\vec{t}^{\prime}$, the function is minimized with $\vec{t}^{\prime} = 0$ (i.e. $\vec{t} = \vec{a}_c - R\vec{h}_c$).

In other words, translation is just the difference between the target area's centroid and the rotated centroid of the hypothesis. Now, since $\vec{t}^{\prime} = 0$, and noting that $|R\vec{h}_i^{\prime}|^2 = |\vec{h}_i^{\prime}|^2$ (rotation doesn't change vector's length), F can be further rewritten as

$$F = \sum_{i=1}^{n} |\vec{a}_i^{\prime} - R\vec{h}_i^{\prime}|^2 = \sum_{i=1}^{n} |\vec{a}_i^{\prime}|^2 - 2\sum_{i=1}^{n} \vec{a}_i^{\prime} * R\vec{h}_i^{\prime} + \sum_{i=1}^{n} |\vec{h}_i^{\prime}|^2.$$

Thus, rotation has to be chosen so that $\sum\limits_{i=1}^{n} \vec{a}_i^{\prime} * R\vec{h}_i^{\prime}$ is maximized.

4 different methods have been devised to calculate the rotation matrix from the above condition:

- SVD (singular value decomposition) approach by Arun et al [27];
- unit quaternions approach by Berthold K.P. Horn [23];
- orthonormal matrices approach by Horn et al [28];
- dual number quaternions approach by Walker et al [29].

A thorough study by Eggert et al [30] shows that all the four algorithms share roughly the same performance and accuracy characteristics. The GPICP algorithm solves the problem of absolute orientation by employing the unit quaternions approach.

CHAPTER 4

GPU optimizations

This chapter is dedicated to achieving an optimal mapping of GPICP to graphics hardware and related programmatic optimizations of the algorithm.

4.1 Introduction to CUDA architecture

A CUDA-enabled GPU is capable of executing so-called ”kernels”; a ”kernel” is a single function (possibly containing calls to sub-functions) performing a single task. Programmer asks the GPU to execute a kernel and specifies the number of threads that need to be dedicated to it. Each thread runs the same code - the kernel's code - but has a unique id, and, using it as a basis for offset, is capable of accessing different areas of memory. Thus, each thread, while performing the same routine, works on its own subset of the input data: the GPU parallelism is of a SIMD (single instruction, multiple data) nature.

The GPU consists of a number of MPs (multi-processors), each one running its share of the kernel's threads. The distribution of threads to the MPs can be indirectly controlled by grouping the threads into the so-called ”blocks”; it is then guaranteed that each block will execute on a single MP. Since an MP has a limit on the number of threads it can be assigned, it is possible to select the block size so that each MP would execute exactly one block.

Although each thread runs the same kernel's code, execution paths will most likely diverge because of the data-dependent conditional branches. Thus, the question of synchronization arises. CUDA provides a *__syncthreads()* function which serves as a synchronization point for the threads of a single block; it is not possible to synchronize different blocks (and, thus, different MPs). Apart from the *__syncthreads()*, there's another type of synchronization provided by the CUDA architecture itself.

Although modern CUDA GPUs are capable of assigning up to 1024 threads to an MP, in reality only a small number of threads can really be executed in parallel on a single multiprocessor. The implementation is as follows: the GPU divides each block into a number of ”warps”, each one consisting of strictly 32 threads. Threads inside a warp are executed in parallel, but only one warp at a time can be run on an MP. The execution of warps is planned by a scheduler which is a complete black box from the programmer's standpoint, meaning it may schedule the execution of the first instruction of the first warp, then the first two instructions of the fourth warp, then the next two instructions of the first warp etc. However, threads within a warp are perfectly synchronized with each other, meaning they execute one common instruction at a time; if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling the threads that are not on the path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a single warp; different warps execute independently, regardless of whether they are executing common or disjoint code paths.

Branch divergence poses the first major problem for a CUDA programmer. If a conditional expression returns ”true” only for the first thread of a warp, the other 31 threads would have to wait until the first one finishes running the conditional code.

Another major problem is memory. Since the graphics card is a separate unit, it has its own memory on board, and transferring data between it and the CPU memory is an expensive process. Moreover, such transfer is only possible in between the kernels' execution.

An average modern CUDA-enabled GPU has access to 2048 MB of memory, although some recent flagship models go up to 4096 MB. 2048 MB may seem a rather small amount if two facts are taken into consideration:

- CUDA programs normally have to process extra-large amounts of data;
- when writing a CUDA kernel, programmer doesn't have the benefit of an almost infinite disk space, which rules out ”smart” memory managers that could stream data from disk when needed.

On-board GPU memory is divided into a number of spaces that include global, local, shared, texture, and registers (cf. Table 4.1).

Memory	**Is on/off chip**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	Yes	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Table 4.1: Salient features of device memory [31]

Global memory is the ”main” space, comprising almost all of the on-board memory; relative to the multi-processors it is, however, off-chip and expensive to access. Local memory is so named because its scope is local to the thread; it is actually also off-chip. Hence, access to local memory is as expensive as access to global memory. Local memory is used only when the compiler determines that there is insufficient register space to hold a thread variable. Variables that are likely to be placed in local memory are large structures or arrays that would consume too much register space and arrays that the compiler determines may be indexed dynamically.

For performance reasons, all accesses to the off-chip memory are cached through L1; the same on-chip memory is used for both L1 cache and shared memory: it can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache.

Concurrent accesses of the threads of a warp to the off-chip memory will coalesce into a number of transactions equal to the number of cache lines necessary to service all of the threads of the warp. The simplest case of coalescing: the k-th thread accesses the k-th word in a cache line. For example, if the threads of a warp access adjacent 4-byte words (e.g., adjacent float values), a single 128B L1 cache line and therefore a single coalesced transaction will service that memory access.

4.2 One MP – one block – one hypothesis

As was mentioned before, CUDA threads are grouped into so-called ”blocks”, with no synchronization possibilities existing between them, making it impossible to spread ICP calculations for a single hypothesis across multiple blocks. On the other hand, as was mentioned before in Section 4.1, 32 is the maximal number of threads that can be executed simultaneously on a single MP, rendering it useless to try to process more than one hypothesis on a single MP (or in a single block, for that matter). These considerations lead to ”one MP – one block – one hypothesis” model, i.e. the size of a single block is chosen so that the block would occupy the whole MP, and each block is given the task of processing a single hypothesis.

Due to resource demands of the implementation described in this paper, it was impossible to set the size of the block to the maximal 1024 threads. Thus, the size of 512 threads was chosen.

The execution of the GPICP algorithm for a single hypothesis can be roughly broken down into two steps.

During the first step:

- each of the RANSAC groups gets assigned exactly 16 threads;
- each thread then proceeds to randomly sample 1 hypothesis point and uses the K-D tree to find the corresponding closest point in the scene;
- after that, all the threads in a group work together to calculate the centroids for the combined set of 16 sampled hypothesis points and its corresponding closest sub-set;
- these centroids then get subtracted by each thread from the hypothesis point and the scene point the thread is responsible for; the new relative coordinates contribute to coordinate permutations;
- the final assignment of the coordinate permutations to the M matrix [23] is done by the first thread;
- the first thread is also responsible for the N matrix [23] calculations (while the other 15 threads wait);

- the first thread proceeds to calculate the most positive eigenvalue λ_m of N (while the other 15 threads wait);
- all the 16 threads then work together again to get the matrix of $(N - \lambda_m I)$'s cofactors;
- the first thread finishes the transformation calculations by finding the eigenvector corresponding to λ_m and then computing the rotation matrix & the translation vector.

The size of RANSAC groups is crucial for the algorithm. As was mentioned in Section 3.2, a RANSAC set should be big enough to escape the degenerate case. These are purely theoretical considerations. There's another hardware-related aspect which manifests itself when the algorithm is applied to the CUDA GPU architecture. Since, as was mentioned in Section 4.1, the smallest unit of parallelism on a CUDA GPU is a warp consisting of 32 threads, it is effective to keep the size of RANSAC sets under 32 points. In this case, if each thread is assigned its own hypothesis point to process, a single RANSAC group can be executed in complete synchronicity, without the need for actual synchronization primitives.

While it's a good idea to set the RANSAC group size to coincide with the warp size, i.e. 32 threads, it's an even better idea to give each RANSAC group a so-called "half-warp" consisting of 16 threads. In this case, all of the calculations performed solely by the first thread in a group (e.g., eigenvalue computation) will be executed for 2 RANSAC groups in parallel, since a warp will then comprise exactly 2 groups. Furthermore, accuracy measurements show that it's sufficient to select 16 points for a RANSAC set to escape the degenerate case (see Section 3.2, cf. Figure 3.8, Figure 3.10).

The second step begins with a synchronization point, since the GPU has to wait until all of the RANSAC groups have finished their respective calculations. It's worth noting again that the order in which half-warps corresponding to different RANSAC groups are executed is undefined, since the scheduler appears as a black box to a GPU programmer.

The goal of the second step is to calculate the error metric for each of the transformations in order to select the best one. The error metric is calculated sequentially, while each transformation is processed by all the 512 threads in parallel.

4.3 Balanced vs. left-balanced k-d tree

As was mentioned in Section 4.1, one of the major performance problems for a CUDA program is branch divergence; it occurs when threads of a warp diverge via a data-dependent conditional branch. During its first step, GPICP calls for each thread in a warp to sample one RANSAC point and then to find its nearest neighbour in the scene; effectively all 32 threads of a warp try to perform their k-d tree searches in parallel. "Try" is the keyword here since during different searches, execution paths will most definitely vary, creating a lot of branch divergence.

As the comprehensive study by Brown and Shoeyink shows, this problem can't be circumvented - k-d trees (or any other trees, for that matter) simply do not map good to the GPU [14]. The only thing that can be done to remedy the matter is to make the search itself as short as possible by keeping the k-d tree's height as small as possible. The ideal solution would be a balanced tree; however, when using it, a memory problem arises.

Unlike a left-balanced binary tree, where indices of the nodes can be calculated, the balanced tree has to actually store the pointers to child nodes (and parent nodes, if needed). In other words, each node of a balanced binary tree is represented by quite a large structure:

As was mentioned in Section 4.1, the actual amount of on-chip memory accessible to CUDA threads is quite small, and the k-d tree in the GPICP algorithm is simply too large to fit into either shared or register memory. It has to be placed into the global memory; global memory, being slow, is best read in a coalesced manner. However, during the tree search, accesses to the nodes are chaotic and cannot be coalesced, prompting a large number of global cache misses.

Since severity of cache misses is directly proportional to the amount of memory read, the use of a balanced k-d tree will result in much more misses than that of a left-balanced tree.

The following table provides a comparison of a balanced tree vs. left-balance tree approach.

4.4 Improved k-d tree search

As was mentioned in Section 3.7, a proper tree search includes backtracking to account for the possibility that there may be some points closer to the query than the point obtained from the depth-first search. Backtracking is most easily implemented by using a stack that would hold previously visited nodes; obviously, the stack could be represented by a static array with its length equal to the k-d tree's maximal height. Most applications limit the stack size to 20-28 elements [32].

The stack's statical nature is a very important property when applied to the GPU architecture, since CUDA doesn't allow for dynamic allocation of memory within a kernel. However, another problem occurs in this case. A 20 element stack is an extremely compact structure of 80 bytes, which seems like a small size until being multiplied by the number of threads in a block (since each thread needs to perform its own NNS), i.e. 512. 80 bytes * 512 = 40 KB, which would occupy almost all the shared memory available for an MP, and is certainly larger than the available register memory.

There are two solutions available for both balanced and left-balanced k-d trees:

- each node of a balanced tree would have to save a pointer to its parent, further increasing the amount of memory needed for storing the tree and, consequently, the severity of global cache misses;
- when using a left-balanced tree, the index of the parent can easily calculated from the node's own index.

However, the search algorithm cannot always backtrack to the parent, since this would create an infinite loop.

The goal here is to skip the node from which the algorithm went sideways, since processing this node would simply send the search back on the path from which it has just returned.

The following solution could be proposed:

1. Store the index of the previously visited node.
2. When backtracking, before doing the BOB test, perform the split test again.

3. Calculate the index of the node we'd have to go sideways to in case the BOB test succeeds.
4. If the index points to the same node as the previously visited one, skip the BOB test altogether and backtrack to parent.

However, in case of a left-balanced tree, step 3 could also be averted. In a left-balanced tree, left children always have odd indices, while the right ones have even indices. The proposed solution could thus be modified in the following way:

1. Store the index of the previously visited node.
2. When backtracking, before doing the BOB test, perform the split test again.
3. If we need to go left to perform the sideways movement, and the index of the previously visited node is odd, or we need to go right, and the index is even, skip the BOB test altogether and backtrack to parent.

Below is the full pseudo-code for the NNS algorithm (cf. Algorithm 4.1).

The proposed algorithm, however, has much more conditional branches than the more direct version using the stack construct, resulting in more branch divergence.

```
input: const queryPoint[]
bestDist ← $MAX\_FLOAT$;
bestNode ← $NON\_EXISTENT\_NODE$;
fromLowerNode ← $NON\_EXISTENT\_NODE$;
goDown ← $true$;
currNode ← $FIRST\_NODE$;
while (nodeExists (currNode)) do
    const splitValue ← getSplitPlaneValue (currNode);
    const goLeft ← (queryPoint [getSplitPlaneAxis (currNode)] $<$ splitValue);
    if (goDown) then
        const currDist ← vectorLength (queryPoint $-$ getPoint (currNode));
        if (bestDist $>$ currDist) then
            bestDist = currDist;
            bestNode = currNode;
        end
        const nextNode = (goLeft ? lchild (currNode) : rchild (currNode));
        if (!nodeExists (nextNode)) then  goDown ← $false$;
        else  currNode ← nextNode;
    end
    else
        const notReturnedFromBacktrack ← (isOdd (fromLowerNode) $==$ goLeft);
        if (!nodeExists (fromLowerNode) || notReturnedFromBacktrack) then
            const nextNode ← (goLeft ? rchild (currNode) : lchild (currNode));
            qpDist ← queryPoint [getSplitPlaneAxis (currNode)] $-$ splitValue;
            qpDist ← qpDist $*$ qpDist;
            const BOBTest ← (bestDist $*BOB\_EPSILON >$ qpDist);
            if (nodeExists (nextNode) && BOBTest) then
                currNode ← nextNode;
                fromLowerNode ← $NON\_EXISTENT\_NODE$;
                goDown ← $true$;
                continue;
            end
        end
        fromLowerNode ← currNode;
        if (currNode $== FIRST\_NODE$) then
            currNode ← $NON\_EXISTENT\_NODE$;
        end
        else  currNode ← parent (currNode);
    end
    return getPoint (bestNode);
end
```

Algorithm 4.1: NNS

CHAPTER 5

Results & discussion

The main goals of this work, as outlined in Introduction 1, were as follows:

- to achieve an optimal mapping of the ICP algorithm to graphics hardware, creating a GPU-only implementation (called GPICP);
- to optimize the algorithm itself, both in mathematical and programmatic ways;
- to increase registration accuracy, when compared to the standard ICP.

The main point of reference was a half-CPU, half-GPU implementation of the algorithm which is part of PCL and which represents a standard, by-the-book ICP. This particular implementation was selected because of the availability of its source code, its high performance and robustness, and popularity of PCL itself [10].

It's easy to see if the third goal was reached by visually comparing the results delivered by both PCL and GPICP implementations, as well as comparing the corresponding mean error values.

Because of the high robustness of the PCL implementation and the fact that it already shifts the most resource-consuming part of the algorithm (i.e. closest point search) to the GPU, better performance of the GPICP implementation would indicate that the first goal was reached. Furthermore, because of the GPU's parallel nature, better scaling with respect to the number of processed hypotheses would also indicate success in this area.

Since increased registration accuracy means 16x more computations in the case of GPICP, its better (or even comparable) performance on the larger number of objects would indicate that the second goal was implemented successfully.

To obtain the results, two scene clouds were used (scene #1 comprising 61841 points and scene #2 comprising 44882 points). In scene #1, three types of hypothesis objects were used, each one chosen because of its unique topological qualities:

- a liqueur glass, an object which is symmetrical with respect to the vertical axis and which has a strong representation in the reference point cloud (definite shape in the reference cloud);
- a mug, an asymmetrical object with a strong representation in the reference point cloud;
- a simple glass, an object which is symmetrical like the liqueur glass, but has a weak representation in the reference point cloud (indefinite shape in the reference cloud);

In scene #2, a clorox bottle and a milk carton were used; both are asymmetrical objects with weak representation in the reference cloud.

Of each hypothesis object, more than 5 instances were considered with different initial poses, bringing the total number of processed instances to 21 in scene #1 and 14 in scene #2:

- scene #1, 1 - 7: mugs (see Figure 5.2);
- scene #1, 8 - 15: liqueur glasses (see Figure 5.2);
- scene #1, 16 - 21: simple glasses (see Figure 5.2);
- scene #2, 1 - 7: clorox bottles (see Figure 5.4);
- scene #2, 8 - 14: milk cartons (see Figure 5.4).

Each hypothesis instance comprises 100 000 points. For dependence of the algorithm's computing time on the number of points in hypotheses, see Table 5.1 and Figure 5.5.

The tests were performed on an NVidia GeForce GTX 460 v2 (produced by Gainward, 7 multiprocessors, 2815 MB video RAM).

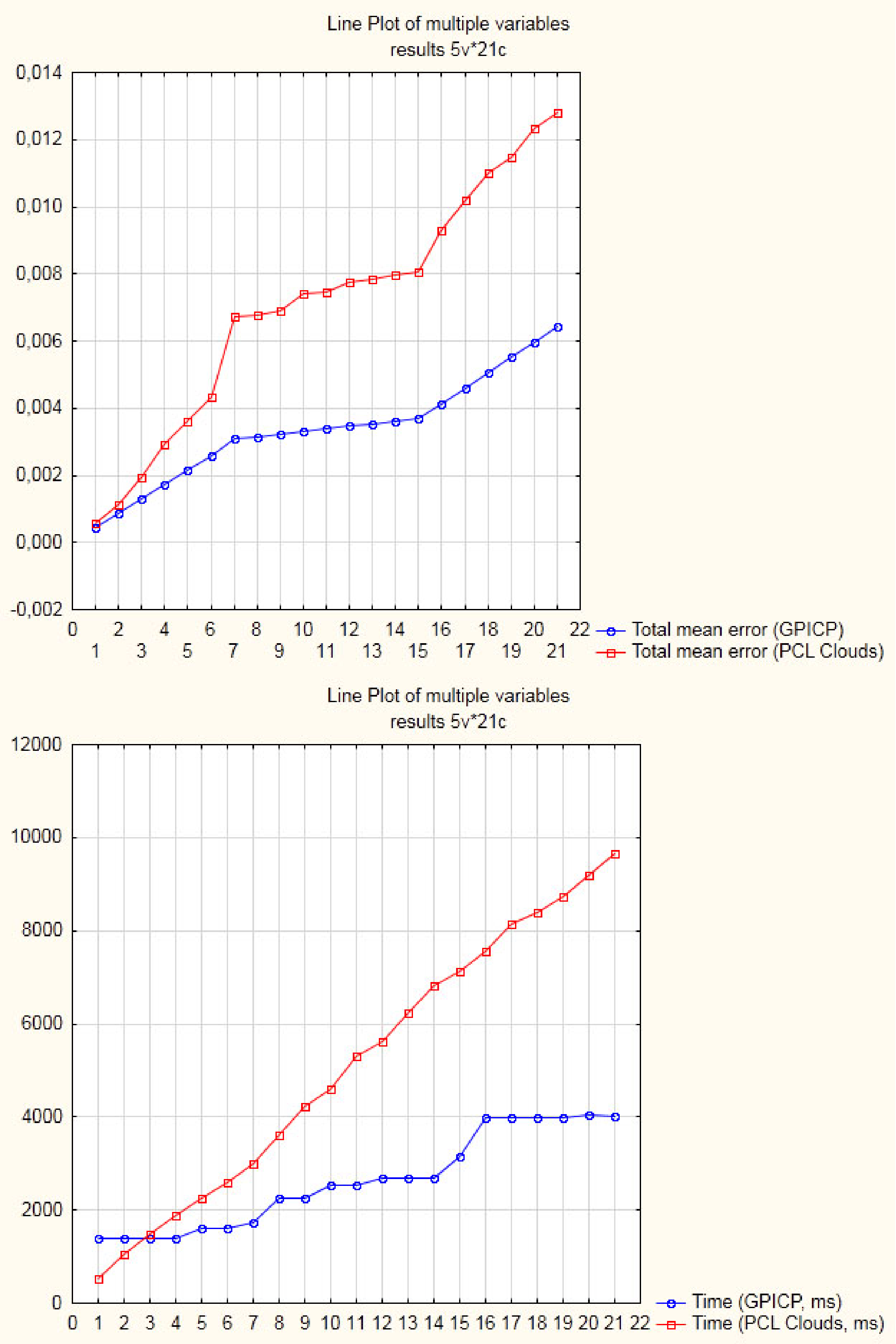


Figure 5.1: Dependence of the time / error on the number of objects (scene #1)

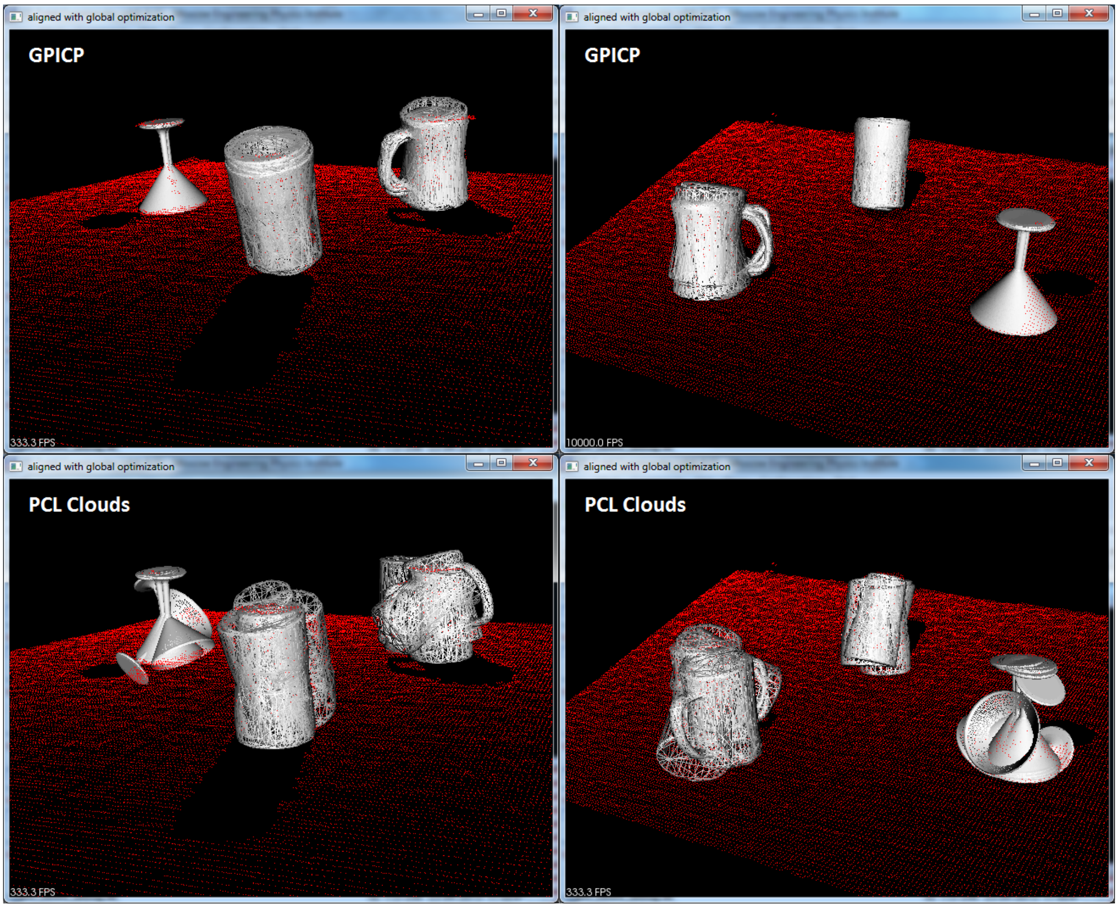

aligned with global optimization
GPICP
333.3 FPS
aligned with global optimization
GPICP
10000.0 FPS
aligned with global optimization
PCL Clouds
333.3 FPS
aligned with global optimization
PCL Clouds
333.3 FPS

Figure 5.2: Visual results (21 objects: 7 mugs, 8 liqueur glasses, 6 glasses (scene #1))

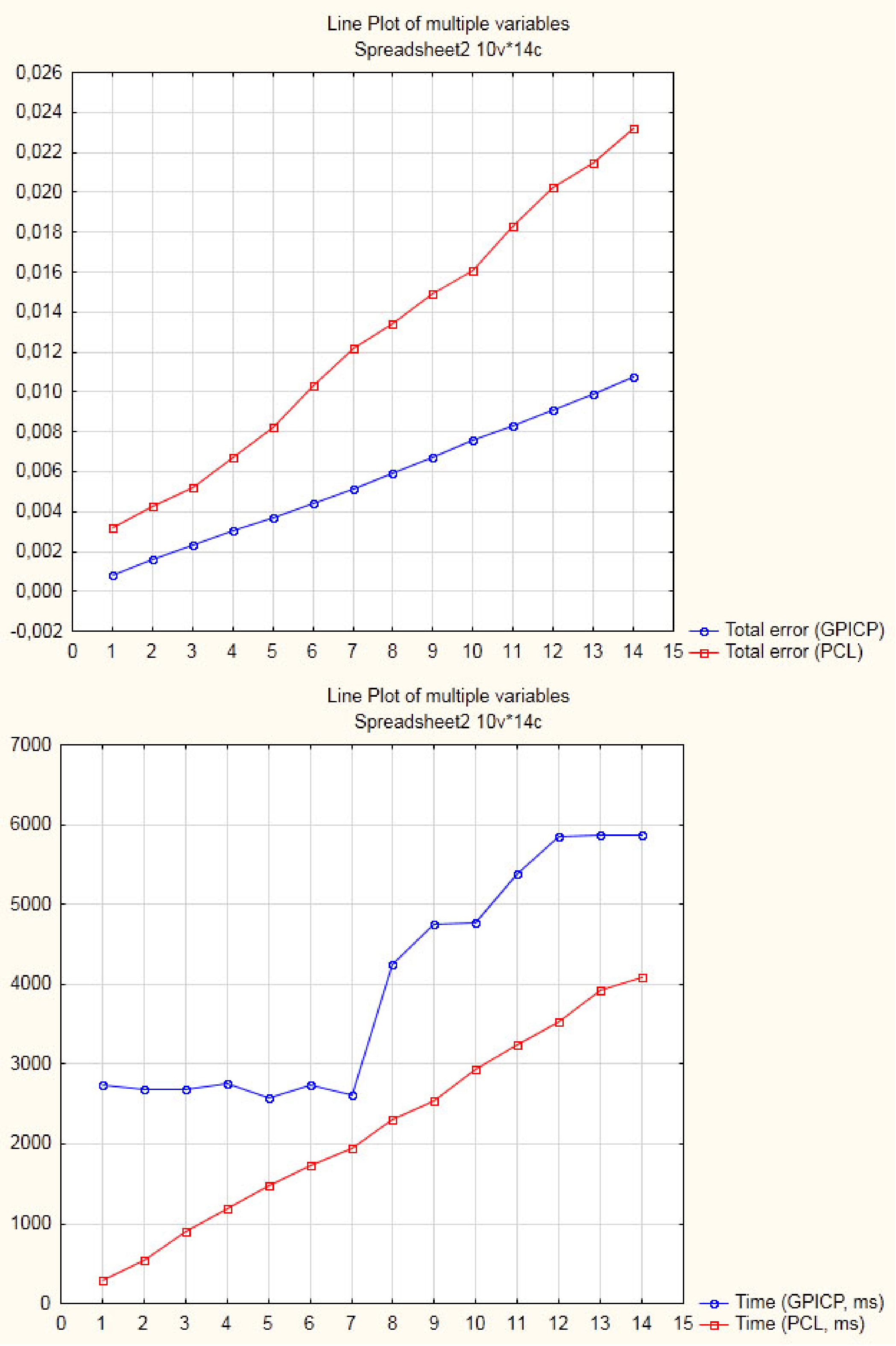


Figure 5.3: Dependence of the time / error on the number of objects (scene #2)

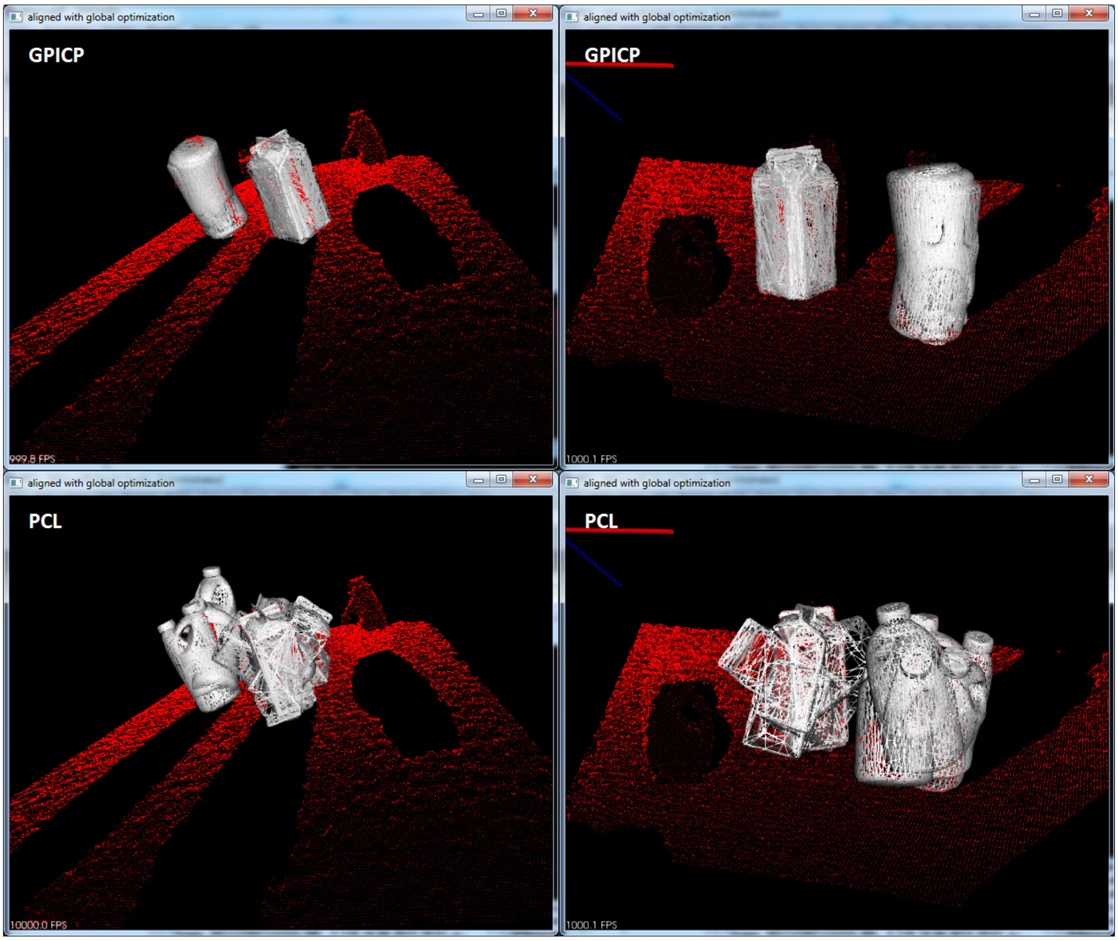

aligned with global optimization
GPICP
999.8 FPS
aligned with global optimization
GPICP
1000.1 FPS
aligned with global optimization
PCL
10000.0 FPS
aligned with global optimization
PCL
1000.1 FPS

Figure 5.4: Visual results (14 objects: 7 clorox bottles, 7 milk cartons (scene #2))

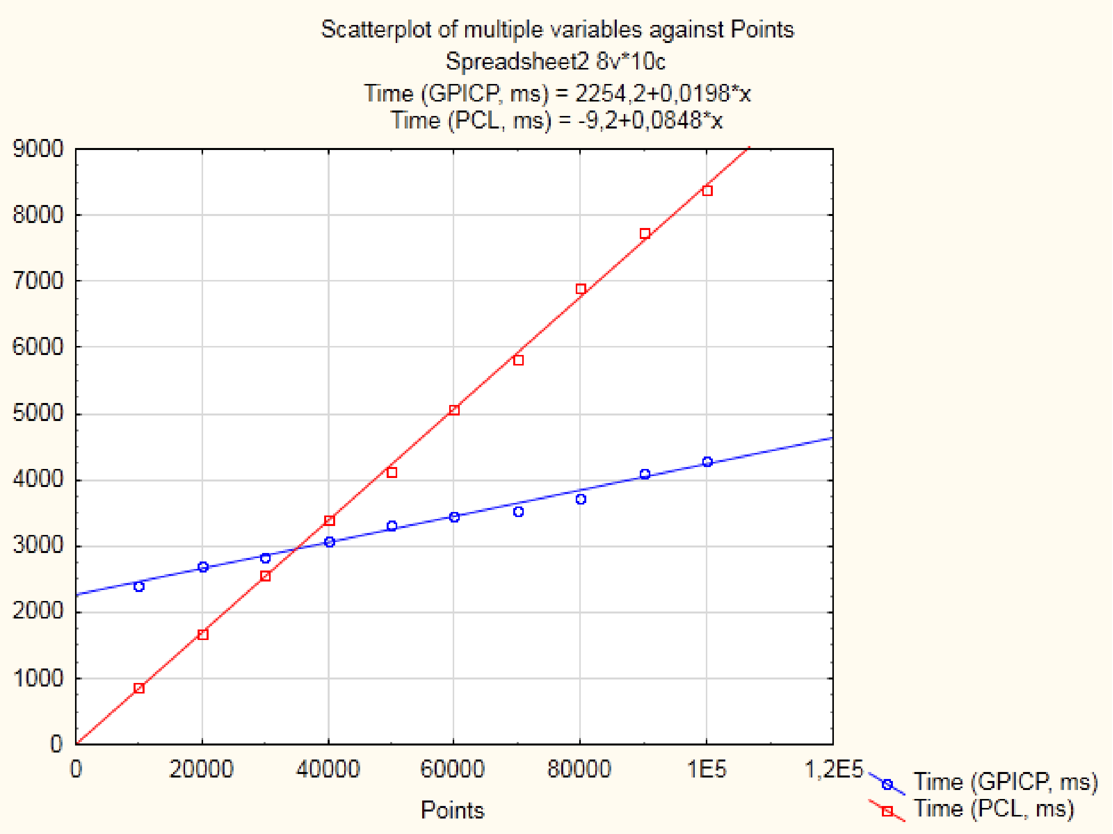


Figure 5.5: Dependence of algorithm's computational time on the number of points in a hypothesis (scene #1)

Number of points	Time (GPICP, ms)	Time (PCL, ms)
10000	2392	859
20000	2710	1686
30000	2821	2566
40000	3084	3406
50000	3329	4122
60000	3465	5056
70000	3538	5824
80000	3715	6912
90000	4104	7732
100000	4275	8376

Table 5.1: Dependence of algorithm's computational time on the number of points in a hypothesis (scene #1)

The first thing that is obvious from the above data is the fact that GPICP proves to be much more accurate, delivering better visual results and a smaller mean error (see Figures 5.1, 5.3). The second important observation lies in the fact that GPICP's results are much more stable: each next hypothesis has the same mean error as the previous one of its group (as evidenced by the piecewise linear character of the error function), while the error from the PCL implementation fluctuates: the algorithm works better on some hypotheses and worse on the others (depending on whether the initial pose was chosen favourably) (see Figures 5.1, 5.3). This indicates that the third goal (increasing registration accuracy) has been reached.

Optimality of the GPU mapping is evidenced by the fact that while the PCL implementation's processing time grows linearly with each new hypothesis, GPICP's time is almost a piecewise linear function, with intervals of roughly 7 hypotheses. This is explained by the fact that each hypothesis is processed on a single GPU multiprocessor, while the GPU on which the results were taken (see above) has 7 of these multiprocessors (i.e. it can process roughly 7 hypotheses for the cost of one, see Section 4.2). In other words, GPICP delivers slower performance only after completely exhausting the GPU's resources. Thus, the first goal has also been successfully reached.

Another interesting point lies in the fact that the more multiprocessors a GPU possesses, the better GPICP will perform; on a GPU with more than 20 multiprocessors, the above graph will degenerate into an almost straight line.

GPICP also scales better with respect to the number of points in a hypothesis; although both GPICP's and PCL's computational times grow linearly with increase of hypotheses' size, PCL's time curve is 4x as steep.

Since, considering the fact that PCL is already quite fast, and the fact that GPICP performs 16x more computations, better overall performance in scene #1 and comparable performance in scene #2 indicate that the second goal has been successfully implemented.

To summarize the results:

- GPICP delivers stable results: each next hypothesis has the same mean error as the previous one of its group (as evidenced by the piecewise linear character of the error function), while the error from the PCL algorithm fluctuates;
- GPICP delivers a smaller error overall;
- while the PCL algorithm's time grows linearly with each new hypothesis, GPICP's time is almost a piecewise linear function, with intervals of roughly 7 cases;
- GPICP is faster overall.

CHAPTER 6

Conclusions

The work resulted in development of GPICP, an ICP implementation that runs entirely on the GPU. GPICP is optimally mapped to the graphics hardware and contains a number of mathematical and programmatic optimizations (with respect to the original ICP).

Compared to the reference ICP implementation, GPICP is faster, delivers a smaller registration error and significantly more accurate results. GPICP also scales better with respect to the number of processed hypotheses and their size (i.e. point count), and will further benefit from the development of new graphics hardware.

The developed algorithm opens up an immense field of possibilities for further experiments. One of the focus areas could be feature detection, which would further improve registration for non-symmetrical objects: instead of randomly sampling the hypothesis for RANSAC and error sets (see Section 3.5), points could be selected from the object's characteristic regions.

Another focus area could be application of point-to-plane metric for finding the RANSAC correspondences; this would require, however, extensive pre-processing of hypotheses and the reference cloud in order to construct normals.

Bibliography

[1] Y. Chen and G. Medioni. Object modelling by registration of multiple range images. *Image and Vision Computing*, 10(3):145–155, 1992.

[2] C. Torre-Ferrero, J.R. Llata, L. Alonso, S. Robla, and E.G. Sarabia. 3D point cloud registration based on a purpose-designed similarity measure. *EURASIP Journal on Advances in Signal Processing*, 57, 2012.

[3] Jochen Sprickerhof. PCL (Point Cloud Library): range image. http://en.wikipedia.org/wiki/PCL_(Point_Cloud_Library)#Range_Image.

[4] L. Tamas and L.C. Goron. 3D map building with mobile robots. In *2012 20th Mediterranean Conference on Control & Automation (MED)*, pages 134–139, July 2012.

[5] S. Druon, M.J. Aldon, and A. Crosnier. Color constrained ICP for registration of large unstructured 3D color data sets. In *2006 IEEE International Conference on Information Acquisition*, pages 249–255, August 2006.

[6] Q. Jin, Y. Cheng, C. Guo, G. Li, and Y. Sato. Point to Point Registration based on MRI sequences. In *WRI Global Congress on Intelligent Systems, 2009. GCIS '09*, pages 381–384, May 2009.

[7] P. Besl and N. McKay. A method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 14(2):239–256, February 1992.

[8] A. Makadia, A. Patterson, and K. Daniilidis. Fully Automatic Registration of 3D Point Clouds. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2006.

[9] Deyuan Qiu, Stefan May, and Andreas Nüchter. GPU-Accelerated Nearest Neighbor Search for 3D registration. In *ICVS '09 Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems*, pages 194–203, 2009.

[10] Jochen Sprickerhof. PCL::Registration. http://www.pointclouds.org/assets/icra2012/registration.pdf, 2012.

[11] Seung-Hun Yoo, Yun-Seok Lee, Sung-Up Jo, and Chang-Sung Jeong. GPU Implementation of a Clustering Based Image Registration. In *Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues Lecture Notes in Computer Science*, volume 5226, pages 491–497. 2008.

[12] Y. Kitaaki and et al. High speed 3-D registration using GPU. In *SICE Annual Conference*, pages 3055–3059, 2008.

[13] Soon-Yong Park, Sung-In Choi, Jun Kim, and Jeong Sook Chae. Real-time 3D registration using GPU. *Machine Vision and Applications*, 22(5):837–850, September 2011.

[14] Shawn Brown and Jack Snoeyink. GPU Nearest Neighbor Searches using a Minimal kd-tree. http://www.nvidia.com/content/GTC-2010/pdfs/2140_GTC2010.pdf, 2010.

[15] V. Garcia, S. Antipolis, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008, CVPRW '08*, pages 1–6, 2008.

[16] L. Cayton. A nearest neighbor data structure for graphics hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[17] S. Izadi, R.A. Newcombe, D. Kim, O. Hilliges, D. Molyneaux, S. Hodges, P. Kohli, J. Shotton, A.J. Davison, and A.W. Fitzgibbon. KinectFusion: real-time dynamic 3D surface reconstruction and interaction. In *ACM Symposium on User Interface Software and Technology*, page 23, 2011.

[18] G. Blais and D.M. Levine. Registering multiview range data to create 3-D computer objects. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(8):820–824, 1995.

[19] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *Proceedings of 3D Digital Imaging and Modeling*, pages 145–152, 2001.

[20] M. A. Greenspan and G. D. Godin. A nearest neighbor method for efficient ICP. In *Proceedings of the Third International Conference on 3-D Digital Imaging and Modeling*, pages 161–168, 2001.

[21] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Magazine Communications of the ACM CACM Homepage archive*, 24(6):381–395, June 1981.

[22] Giorgio Grisetti. Least Squares and SLAM Scan-Matching using ICP and RANSAC. http://www.informatik.uni-freiburg.de/ grisetti/teaching/ls-slam/lectures/pdf/ls-slam-07-icp.pdf.

[23] B. K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America A*, 4:629, April 1987.

[24] M. A. Greenspan and M. Yurick. Approximate k-d tree search for efficient ICP. In *Proceedings of the Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003*, pages 442–448, 2003.

[25] A. Nüchter, K. Lingemann, and J. Hertzberg. Cached k-d tree search for ICP algorithms. In *Sixth International Conference on 3-D Digital Imaging and Modeling, 2007. 3DIM '07*, pages 419–426, 2007.

[26] S. Arya and D. M. Mount. Approximate nearest neigbor queries in fixed dimensions. In *Proceedings of the 4th ACMSIAM Symposium on Discrete Algorithms*, pages 271–280, 1993.

[27] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-D point sets. *IEEE Trans Pattern Anal Machine Intell*, 9:698–700, 1987.

[28] B. K. P. Horn, H. M. Hilden, and S. Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *J Opt Soc Am Ser*, A(5):1127–1135, 1988.

[29] M. W. Walker, L. Shao, and R. A. Volz. Estimating 3-D location parameters using dual number quaternions. *CVGIP: Image Understanding*, 54:358–367, 1991.

[30] D. W. Eggert, A. Lorusso, and R. B. Fisher. Estimating 3-D rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications*, 9(5-6):272–290, March 1997.

[31] CUDA C Best practices guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2012.

[32] S. Brown and J. Snoeyink. GPU Nearest Neighbor Searches using a Minimal kd-tree. *MASSIVE*, 2010.