THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

Spatial Hashing for Pose Clustering in GPU Point Pair Feature Registration

by

Nicolas Avrutin

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

09/14/2016

Professor Carl Sable, Advisor

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dean, School of Engineering - 09/14/2016

Professor Carl Sable - 09/14/2016

Candidate's Thesis Advisor

Acknowledgments

I would like to thank the following individuals for their contributions to this thesis:

- Carl Sable, my adviser, for guiding me through this project, offering technical suggestions, proofreading, and providing valuable feedback.
- John D. Long II, for his technical assistance with the computer vision, geometry, and CUDA components of this project. Without him, this project would not have gotten off the ground.
- Robert Gruener and Daniel Gitzel, for their work on the senior project that later became this project.
- David Rubinstein and Andrew Apollonsky, for proofreading and providing valuable feedback.
- NVIDIA Corporation, for donating the Tesla K40 used for this project.

Abstract

This thesis improves the point pair feature registration algorithm by introducing a new GPU-based clustering algorithm based on spatial hashing. Compared to the CPU-based agglomerative clustering algorithm used in current point pair feature registration implementations, the spatial hashing clustering algorithm has a significantly lower run time and little to no reduction in accuracy. The new clustering algorithm also allows for a completely GPU-based point pair feature registration implementation, as the clustering step is the only remaining CPU-based component in existing GPU-based implementations. Along the way, we develop the parallel hash array, an associative array data structure with support for parallel operations that can be used as an alternative to hash tables on GPUs. We provide open source implementations of both of these algorithms and, along with them, the first open source GPU point pair feature registration implementation. The source code for this project is available at `https://github.com/nicolasavru/ppf-registration-spatial-hashing` under a BSD license.

Contents

List of Figures

Listings

Chapter 1

Introduction

3D object recognition, has long had applications in industrial automation [2], [11], but the advent of cheap 3D scanners such as Microsoft's Kinect and augmented reality devices such as Google's Glass and Microsoft's HoloLens has created many new applications for it. 3D object recognition is an extensively studied problem, but the solutions remain computationally complex and most algorithms are still too slow to be practical for emerging applications such as augmented reality and autonomous robotics [15]. [15] developed a new 3D object recognition algorithm, point pair feature registration, which was able to realize significant run time improvements relative to the state of the art while maintaining comparable accuracy. Despite this, the run time of point pair feature registration is still on the order of seconds to minutes.

Graphics processing units (GPUs) designed for general-purpose computing make it possible to improve on that run time. Unlike CPUs, which are designed to run a relatively small number of complex tasks and thus contain a small number of highly-clocked cores, GPUs contain thousands of lower-clocked cores. The reason for this architecture is that each core is designed to perform relatively simple tasks on a per-pixel basis [38]. Originally, these cores were limited in what computations there were able to perform, but the early 2000s saw those cores becoming more and more programmable, to the point

where they can now be used for arbitrary computation [38]. The GPU architecture, combined with GPU-based general-purpose programming platforms such as OpenCL [47] and NVIDIA's CUDA [37], has made GPUs an excellent platform for parallelizable computations.

Some 3D object recognition algorithms, in particular point pair registration, are very parallelizable and thus are well-suited to GPU implementations. By re-implementing the point pair feature registration algorithm from [15] on a GPU, [43] was able to achieve real-time performance, a 10-100x run time improvement after controlling for the test data used in each implementation. However, one component of [43]'s implementation, a clustering algorithm, continues to be run on the CPU. This thesis develops and implements a GPU-based clustering algorithm based on spatial hashing, an algorithm commonly used for collision detection in computer graphics [28],[53]. In addition, we develop a new general-purpose associative array data structure, the parallel hash array, to use as an alternative to a hash table on a GPU.

To that end, an overview of point set registration algorithms is provided in Chapter 2 and a detailed explanation of point pair feature registration in Chapter 3. Chapter 4 introduces the parallel hash array. The GPU-based clustering algorithm is presented in 5. Finally, Chapter 6 compares the performance and accuracy of the new clustering algorithm to that of the CPU-based clustering algorithm of [15].

Chapter 2

Point Set Registration

Image registration is the process of transforming one or more target images such that they are aligned with a reference image [10], as demonstrated in Figure 2.1. One class of applications of image registration is aligning images with partial overlap to combine them and create a larger image, as in image stitching to create panoramas [49]. Another class of applications is object recognition, i.e., finding specific objects and their locations within images [15]. In that context, the reference image is referred to as the scene, the target images (the objects) are referred to as models, and the goal is to find the position and rotation, together called the pose, of the model within the scene. Formally, the pose is represented as a transformation matrix which can be applied to the model to align it with the scene. Point set registration is the application of image registration to images composed of sets of points (also called point clouds) and has applications in both two-dimensional cases (e.g., feature point alignment) and three-dimensional cases (e.g., simultaneous localization and mapping (SLAM), object recognition, medical imaging). Depending on the application, point set registration algorithms need to be robust against noise, clutter (other objects in the scene), and occlusion (only part of the model being present in the scene). These phenomena are demonstrated in Figure 2.2.

There are many different algorithms for point set registration and they use a variety

(a)

(b)

Figure 2.1 – Registration of an uncorrected aerial photo and an orthophoto [52].

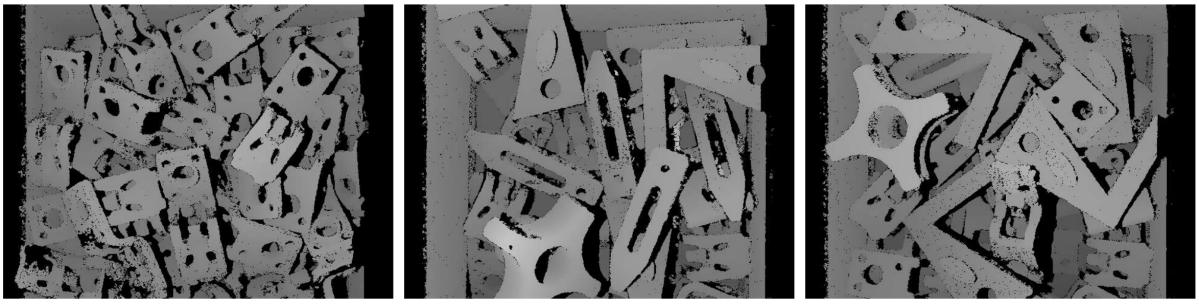


Figure 2.2 – Bins of industrial parts demonstrating noise, clutter, and occlusion [55].

of approaches. Some are special purpose, while others are general purpose. Some are designed for rigid objects, while others are designed for deformable objects. Some use only depth data (the point's spatial coordinates), while others use both color and depth (RGB-D) information. Using color data provides more discriminative power, but, unlike depth data, color data is sensitive to ambient illumination, both directly via the properties of the light source itself and indirectly via the location and orientation of an object relative to the light source. Several types of point set registration will be discussed in this chapter.

2.1 Iterative Closest Point

Iterative closest point (ICP) [8] is one of the most ubiquitous three-dimensional point set registration algorithms and has many different variations [41]. Some variations provide general-purpose performance or accuracy improvements, while others provide application-specific improvements. ICP approaches the point set registration problem by assuming that the model and subset of the scene that contains the model are identical and merely misaligned. That is, for every point in the model, there is a corresponding point in the scene. The correct alignment then is the one that creates perfect overlap between each model point and its corresponding scene point. In practice, the correspondence is not perfect because the model and the scene are captured independently. Instead of aiming for perfect overlap then, ICP tries to minimize the Euclidean distance between corresponding points. However, this still assumes that the correspondence between model

points and scene points is known. In general, this is not the case. The correspondence is not even well-defined if the object in the model and in the scene have a different number of points, for example when the scene contains a partial view of the model. ICP thus places the model at an arbitrary initial position and assumes that each point in the model corresponds to the point in the scene that is closest to it. ICP then iteratively refines the pose by minimizing the sum of the distances between corresponding model and scene points and terminates when the difference in error between successive iterations drops below a predefined threshold. During every iteration, ICP finds a correspondence for each point, computes the distances between corresponding points, finds a transformation that reduces the distance, and applies it to the model points.

ICP is accurate, simple to implement, and has decent run time. However, it is only guaranteed to find local minima, not global ones. As such, it is considered a "fine registration" algorithm and is generally not used on its own. Instead, a "coarse registration" algorithm is used to find an approximate pose and that pose is used as the initial pose for ICP.

2.2 Feature Based Methods

Feature based methods generally rely purely on the geometric properties of models and encode those properties as vectors of varying dimensionality. Feature based methods can be broadly divided into two categories: global and local [50].

2.2.1 Global Methods

Global methods encode an entire model as a single high-dimensional feature vector and then use some version of a nearest neighbor search in the feature space. Because global methods look at complete objects, applying them to object lookup in scenes requires segmenting the scene, a complicated problem in itself [5]. Global methods are often

Figure 2.3 – Two models whose spherical harmonic representations are identical [25].

insensitive to fine details of objects. Spherical harmonic descriptors, for example, are insensitive to rotations of interior parts of objects, as seen in Figure 2.3 [25]. In addition, global methods are often not robust against occlusion and clutter. Given those limitations, global methods are generally not useful for 3D video applications and are more useful for problems involving finding objects similar to the model, as opposed to objects identical to the model [5], [50], [25].

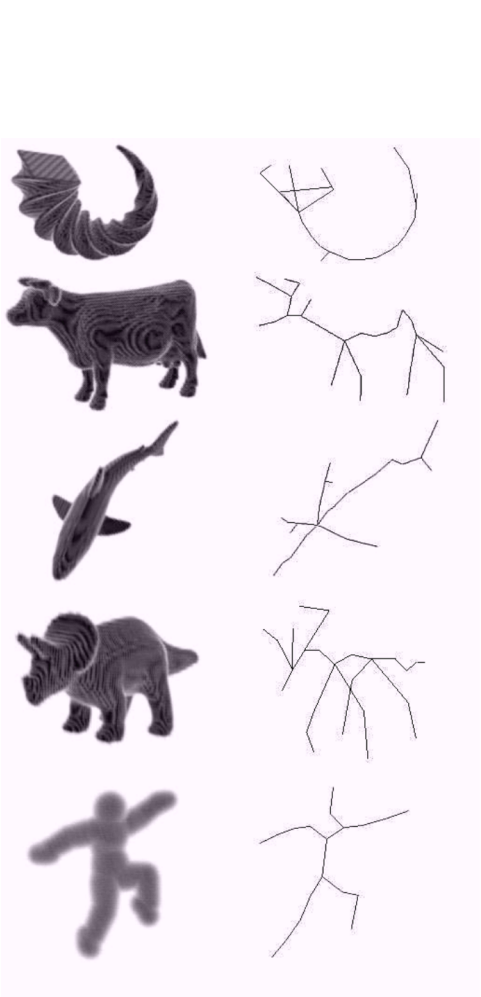
2.2.2 Local Methods

Local methods encode geometric properties of small sections (more generally, subsets of model points) of objects as low-dimensional feature vectors and combine many such vectors to create a description of a complete model. Local methods often extract local features such as keypoints [31], edges, or corners to create more useful feature vectors, though many methods use random subsets or even all subsets that match certain criteria (e.g., a partition, all pairs of points, etc.). Because the feature vector is low-dimensional and constructed from a small set of points, each individual feature vector has little discriminative power. This leads to many incorrect model point-scene point correspondences

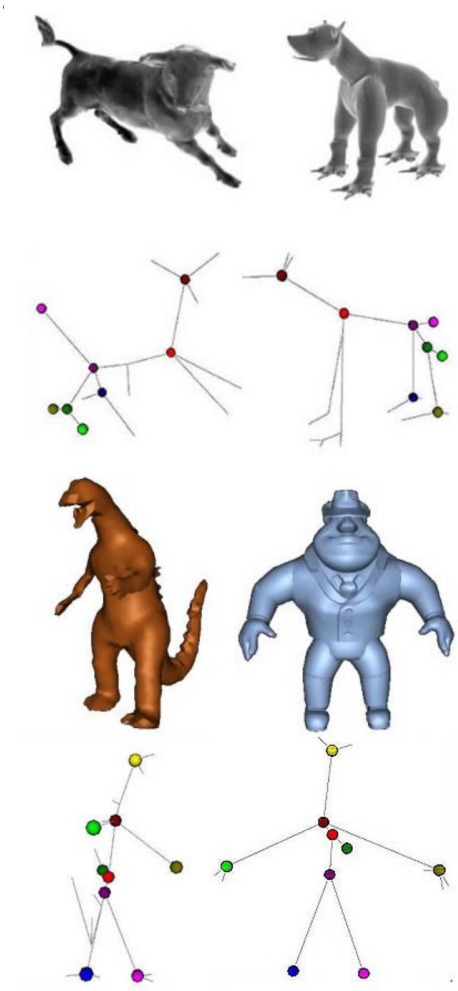
points being inferred and necessitates additional work to filter out incorrect matches. Common methods of performing this filtering are pose clustering, Hough transform-style voting [4], RANSAC [17], etc. [21]. Because local features represent subsets of objects, and because they don't require segmentation, they tend to be more robust against occlusion and clutter [21].

2.2.3 Graph Based Methods

Graph based methods encode topological information about a model by transforming it into a graph and then using graph comparison algorithms to find similar models [50]. Of this class of algorithms, only skeleton-based algorithms are interesting for general-purpose point set registration [50]. [48] generated skeletal graphs from 3D models by applying a thinning algorithm [56] to each model and then connecting the resultant set of points with a minimum spanning tree algorithm [16], creating a directed acyclic graph for each model. The model graphs are then compared by finding a minimally-weighted bipartite graph between the two graphs and evaluating the similarity between the graphs. Figure 2.4 demonstrates skeletal graphs and comparisons between them. Skeletal matching works well for different classes of objects, but cannot distinguish between objects which are topologically similar but differ in details (see Figure 2.4). Moreover, like global feature methods, skeletal matching (and graph based models in general) require scene segmentation. Indeed, graph based methods can be thought of as a special case of global feature methods where the high-dimensional feature vector is a representation of a graph.



(a) Models and their associated skeletal figures.



(b) Comparison skeletal graphs from topologically similar models.

Figure 2.4 – Skeletal graph examples [48].

Chapter 3

Point Pair Feature Registration

Point pair feature registration is a local feature based point set registration algorithm for rigid objects [15]. The general idea is to, for both the model and the scene, create feature vectors from pairs of points within each image (i.e., a model point pair feature is created from two model points and a scene point pair feature is created from two scene points) and find point pair features that are common to both. These common features are, via a Hough transform-like voting scheme [4] and pose clustering, used to identify correspondences between the model and scene and compute the pose of the model within the scene.

The algorithm consists of an "offline" phase and an "online" phase. During the offline phase, global descriptions of models to be identified in scenes are constructed. During the online phase, point pair features of a scene are generated and compared to the stored model descriptions. Based on that comparison, a multiset of model-to-scene transformations (candidate poses) is created, and the transformation with the highest multiplicity is selected. As discussed in Section 2.1, point pair feature registration is a coarse registration algorithm and the iterative closest point algorithm is usually used to refine the result of point pair feature registration.

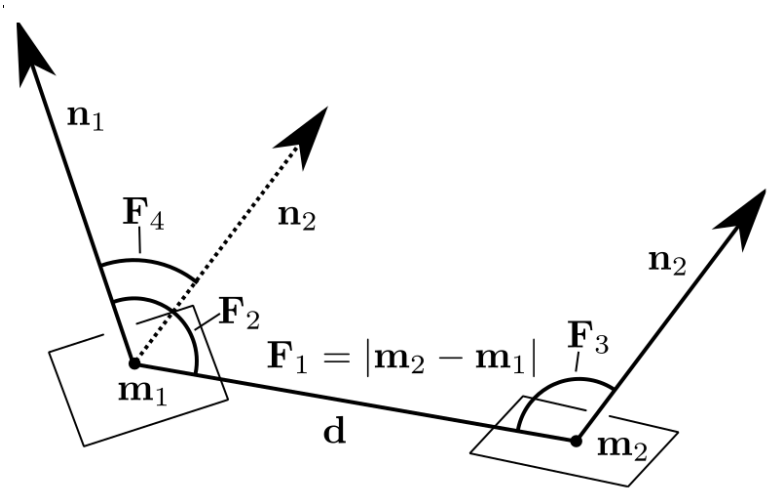


Figure 3.1 – A point pair feature [15].

3.1 Point Pair Feature

The main idea behind point pair features is to sparsely encode local structure and to use an aggregation of these local features to encode the global structure of a model. A point pair feature (PPF) is a 4-dimensional vector created from two oriented points. Given points m_1 and m_2 with normals n_1 and n_2, their point pair feature $F(m_1, m_2)$ is defined as

$$F(m_1, m_2) = (\|d\|_2, \angle(n_1, d), \angle(n_2, d), \angle(n_1, n_2)), \tag{3.1}$$

where $d = m_1 - m_2$ and $\angle(a, b) \in [0, \pi]$ is the angle between the two vectors a and b. This is demonstrated in Figure 3.1. Point pair features are specifically designed to be asymmetrical between the two points to increase their discriminative power.

3.1.1 Point Set Downsampling

Because each point pair feature is generated from two points, generating all point pair features for a set of N points results in $O(N^2)$ point pair features. This imposes a limit on the size of models that point pair features can actually be applied to. Assuming that the amount of memory available for the computation is in the tens of gigabytes, the maximum

number of points is on the order of tens of thousands of points. Given that many models will have hundreds of thousands to millions of points, they must be downsampled. Three common methods of downsampling point clouds are sequential downsampling (select every nth point), random downsampling (randomly select $\frac{n}{N}$ points), and resampling onto a voxel grid [24] (discretizing space into voxels and replacing all points in each voxel with their centroid).

Voxel grid resampling is generally the best method because it ensures uniform point density throughout the model while also preserving low-density areas, particularly object boundaries. However, when performing voxel grid downsampling, it it is difficult to estimate the resultant number of points. Furthermore, the correct voxel size is dependent on the scale of the model, which is dependent on the method used to create it. Random downsampling generally preserves uniform density, but cannot create it if the point set is not uniformly dense to begin with (which can often be the case, depending on the capture method). Sequential downsampling will preserve uniform density if the point cloud is unstructured (the points are stored in a random order), but not necessarily if it is structured (there is physical significance to the order of the points). Sequential downsampling can also cause aliasing.

3.1.2 Global Model Description

Using the computed point pair features, we can now construct global descriptions of the models. The purpose of the global model description is to, given a pair of points in the scene, find similar pairs of points in the model. These corresponding pairs can then be used to compute candidate model-to-scene transformations and find the most likely candidate.

To create a global description of a model, we generate all pairs of model points and, for each one, compute its point pair feature. To ensure that similar but non-identical point pair features are grouped together, each point pair feature is discretized

by sampling the distance component at a rate of d_{dist} and the angular components at a rate of $d_{\text{angle}} = 2\pi / n_{\text{angle}}$. Finally, a hash table mapping each discretized point pair feature to lists of point pairs which generated that point pair feature is created. Algorithm 3.1 provides pseudocode for constructing a model description.

```
model_description = {}
for point_i in model:
    for point_j in model:
        ppf = discretize_ppf(compute_ppf(point_i, point_j))
        model_description.setdefault(ppf, []).append((point_i, point_j))
```

Algorithm 3.1 – Create Global Model Description.

3.1.3 Transformation Computation

Consider a pair of scene points (S_r, S_i). If they lie on the object we are trying to register, then there will be a corresponding pair of model points (M_r, M_i) such that $T_{m\to s} M_r = S_r$ and $T_{m\to s} M_i = S_i$, where

$$T_{m\to s} = \left[\begin{array}{ccc|c} & R & & \begin{matrix} t_x \\ t_y \\ t_z \end{matrix} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \in \mathbb{SE}_3 \tag{3.2}$$

is a homogeneous transformation matrix and represents a 3-dimensional affine transformation with rotation component R and translation component $\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$.

Given (S_r, S_i) and (M_r, M_i) and the normal vectors of S_r and M_r, there is sufficient information to unambiguously compute $T_{m\to s}$. To do so, compute $T_{s\to g}$ and $T_{m\to g}$, which respectively transform S_r and M_r into a global coordinate frame by translating them onto the origin and rotating them such that their normals point along the x-axis as described by `to_global` in Algorithm 3.2. It is important to note that this algorithm depends on accurate normal vectors. If normal vectors are not available, they must be computed via plane fitting or triangulation. It is often not feasible to compute accurate normals due

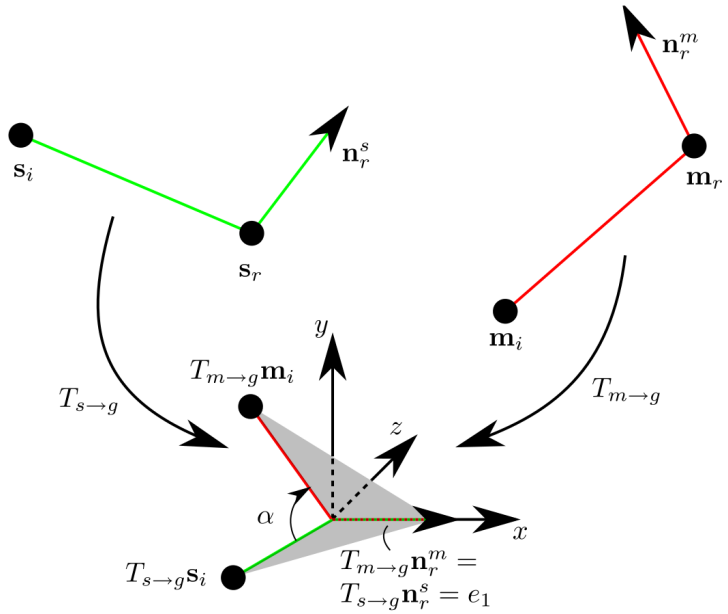


Figure 3.2 – Translation of scene and model point pairs into a global frame and their alignment with the x-axis [15].

to resource constraints, in which case worse approximations can be used, at the expense of reducing the accuracy of the registration algorithm.

The translation accounts for three of the six degrees of freedom in a 3-dimensional rigid transformation, and normal alignment rotations, which can be decomposed into rotations about the y- and z-axes, account for another two. The final degree of freedom is accounted for by rotating M_r about the x-axis such that S_i and M_i are coincident. Let that rotation be described [24] by

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}. \tag{3.3}$$

Thus,

$$T_{m\to s} = T_{s\to g}^{-1} R_x(\alpha)\, T_{m\to g}. \tag{3.4}$$

Figure 3.2 demonstrates the complete alignment procedure and Algorithm 3.2 describes its details.

```
# {x,y,z}_rot_mat(a) are functions returning transformation matricies
# representing rotations of angle a about the x, y, and z axes,
# respectively.

def to_global(point, normal):
  T_trans = -1 * point
  T_roty = y_rot_mat(atan2(normal.z, normal.x))
  n_temp = T_roty * normal
  T_rotz = z_rot_mat(-1 * atan2(n_tmp.y, n_tmp.x))
  return T_rotz * T_roty * T_trans

def compute_alpha(T_m_g, T_s_g, m_i, s_i):
    u = T_m_g * m_i
    v = T_s_g * s_i
    u.x = 0
    v.x = 0
    return atan2f(cross(u, v), dot(u, v))

def model_to_scene(m_r, m_r_n, m_i
                   s_r, s_r_n, s_i)
    T_m_g = to_global(m_r, m_r_n)
    T_s_g = to_global(s_r, s_r_n)
    alpha = compute_alpha(T_m_g, T_s_g, m_i, s_i)
    R_x = x_rot_mat(alpha)
    return inv(T_s_g) * R_x * T_m_g
```

Algorithm 3.2 – Compute Model to Scene Transformation

3.1.4 Voting Scheme

The computation of $T_{m \to g}$ from Section 3.1.3 assumed the existence of two corresponding pairs of model and scene points, but did not specify how to find those corresponding pairs. In addition, because point pair features are low-dimensional and only encode local structure, they are prone to false positives. And so, the problem becomes one of identifying true point pair correspondences. Given the information encoded in point pair features, we cannot definitively determine which point pair correspondence is correct. Instead, we can use a Hough transform-like voting scheme, which transforms the possible

correspondences into an accumulator space and selects the most commonly occurring correspondence as the best candidate.

Consider an arbitrary scene point. To represent its accumulator space, we set up a 2-dimensional $N_m \times n_{\text{angle}}$ array, where $N_m = |M|$, the number of points in the model, and n_{angle} is the number of discretized point pair feature angle values, with each entry initialized to 0. To find the possible model point correspondences for the scene point, it is paired with every other scene point and the point pair feature of the pair is computed and discretized. Each discretized point pair feature is looked up in the model description hash table. For each returned model point pair, α is computed as described in Section 3.1.3, and the value in the row of the accumulator corresponding to the model reference point and the column of the accumulator corresponding to the α index (discretized α value) is incremented. This is equivalent to voting for a particular model-to-scene transformation, but the model point-angle accumulator requires less memory to store. The highest-valued entry in the accumulator is the transformation with the most votes and is selected as the most likely correct transformation.

3.1.5 Pose Clustering

Section 3.1.4 describes how to find the best pose (model-to-scene transformation) given a scene reference point that is known to lie on the object being searched for. However, we do not know *a priori* whether any given scene point lies on the object. And so, we repeat the above voting scheme for all scene points. This results in a set of pose-vote pairs from which the best pose needs to be selected. This is done by clustering all of the poses such that their translation components do not differ by more than a distance threshold and their rotation components don't differ by more than an angle threshold. The cluster with the top score, where a cluster's score is defined as the sum of the vote counts of the transformations composing it, is selected. To improve the accuracy of the final pose, the transformations within the top-voted cluster are averaged, and the

resulting transformation is returned.

While we need many scene reference points to ensure that enough of them are actually on the model and to overcome noise from false positives, the situation is usually not so dire that we need to use every single scene point as a reference point. As an optimization, we can use every nth scene point as a reference point and pair each of them with every other scene point. This reduces the number of point pair features from N^2 to $\frac{N^2}{n}$. Based on our experience, n can be on the order of 5-20 without significant degradation in accuracy, though this depends on the particular models involved. Instead of using every nth scene reference point, we can equivalently pair every scene reference point with every nth other point. Downsampling both sets is equivalent to simply downsampling the entire model by a higher factor.

[15] does not describe the clustering algorithm they use in detail. The implementations of [42] and [11] use an agglomerative clustering method. They initialize an empty list of clusters and sort the poses in descending order based on number of votes. For each pose, they search the existing list of clusters to see whether the pose fits into any cluster based on translation and rotation thresholds. If it does, then it is averaged into the cluster; otherwise it is used to initialize a new cluster.

3.1.6 Disadvantages of Agglomerative Clustering

The agglomerative clustering approach is usually efficient, but does have several drawbacks. In the worst case, when there are a lot of isolated poses, the time complexity of agglomerative clustering is $O(n^2)$, where n is the number of poses. Depending on the implementation, it is unstable; if each pose is only added to one cluster, as is usually the case with clustering algorithms and is the case in Point Cloud Library's implementation [42], then the cluster membership can depend on the order in which the poses are tested for cluster membership. This can happen if a pose is within the radii of two distinct clusters. An additional stability problem exists if new poses are averaged into

each cluster as soon as membership is confirmed, as is the case in [11]'s implementation, because that could move the cluster center enough such that another pose which was within the radius of the cluster no longer is. In contrast, PCL's implementation maintains a membership list for every cluster and performs the averaging in a separate pass, after every pose is added to a cluster.

In addition, agglomerative clustering is difficult to efficiently implement on a GPU due to the need for a single, shared buffer to store the list of clusters as they are initialized. Another problem for GPU implementations is the possible order-dependency of the algorithm, which necessitates a serial execution for consistent results. The novel clustering approach presented in Chapter 5 address all of these problems.

3.2 Variations on a Point Pair Feature

Various papers [11], [26], [55] have attempted to address the shortcomings the original point pair feature registration algorithm presented in Section 3.1. A major weakness of the point pair features is that they are insufficiently discriminative in highly symmetric environments. One such case is when the model and the scene both contain many objects with large planar surfaces. Such environments are common in mapping and augmented reality applications[43] (chairs, tables, floors, walls), as well as industrial assembly and bin-packing applications (small, flat parts) [11], [30], [55]. In those cases, the point pair features of the planar surfaces dominate the voting scheme, resulting in incorrect point correspondences receiving a lot of votes.

3.2.1 Boundary Features

[11] introduces three variations of the point pair feature. In this context, [15]'s point pair features can be thought of as surface-to-surface features because they are based on pairs of surface points.

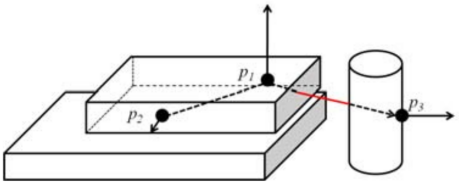
The first point pair feature variation they introduce is the boundary-to-boundary feature, which only creates features out of pairs of boundary points. Because boundary points do not have well-defined normals, lines are fitted to the boundaries and their directions (i.e., vectors pointing along the boundary) are used instead. The advantages of boundary-to-boundary features are threefold. First, intuitively, boundary points are more informative than surface points. A consequence of that higher information content is that boundary features resolve the difficulty that point pair features have with planar surfaces. Second, there are far fewer boundary points than surface points, which is especially beneficial given the quadratic nature of point pair feature algorithms. Finally, it is easier to robustly compute line segments than surface normals [11].

Boundary-to-boundary features do not perform well on objects with high curvature (i.e., objects with few, if any, well-defined linear boundaries). To that end, they introduce the surface-to-boundary feature, which attempts to overcome that problem by pairing one surface point with one boundary point. Finally, they introduce the line-to-line feature

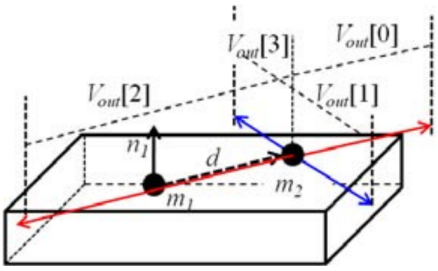
$$F\left(l_1, l_2\right) = \left(d_{\perp}, \angle\left(l_1, l_2\right), d_{max}\right), \tag{3.5}$$

where $d_{\perp}$ is the perpendicular distance between the infinite lines containing boundary line segments l_1 and l_2, $\angle\left(l_1, l_2\right)$ is the acute angle between the two line segments, and d_{max} is the maximum distance between the endpoints of the line segments.

Boundary-to-boundary features and surface-to-boundary features were both found to be more accurate and efficient than standard point pair features, though which of the two performed better varied for different objects. Line-to-line features also performed better than standard point pair features on objects with low curvature, though they were less efficient.



(a) Point pairs (p_1, p_2) and (p_1, p_3) have identical standard point pair features, but the visibility context of p_1p_3 (the red section) disambiguates it from p_1p_2.



(b) Components of vector V_{out}.

Figure 3.3 – Extended point pair features [26].

3.2.2 Point Pair Features With Visibility Context

[26] extends the point pair feature with two additional components to add discriminative power and thus reduce the number of false positive matches. The first additional component is a value representing the visibility context (surface, visible, or occluded) of the vector connecting the two points. The extended point pair feature takes the form of

$$F\left(m_1, m_2\right) = \left(\|d\|, \angle\left(n_1, d\right), \angle\left(n_2, d\right), \angle\left(n_1, n_2\right), V_{\text{in}}\right). \tag{3.6}$$

In Figure 3.3a, the standard point pair features of (p_1, p_3) and (p_1, p_3) are equal, but the extended point pair features are distinct because $d_{1,2} = p_2 - p_1$ is occluded, while $d_{1,3} = p_3 - p_1$ is visible. [26] then extended the point pair feature with another component, V_{out}, which is a 4-dimensional feature vector where each component is itself a vector:

$$F\left(m_1, m_2\right) = \left(\|d\|, \angle\left(n_1, d\right), \angle\left(n_2, d\right), \angle\left(n_1, n_2\right), V_{\text{in}}, V_{\text{out}}\right). \tag{3.7}$$

Each of those vectors extends from m_2 to the visible region, as illustrated in Figure 3.3b.

Using the additional information in V_{out}, they develop an alternate way to compute the model-to-scene transformation matrix that does not depend on the model and scene point normal vectors, which are often noisy. They add an additional pose validation step to filter out false positives, then perform a form of global accumulation and clustering

by discretizing the translation components of the transformations and selecting from every voxel the transformation with the highest number of votes. Finally, they perform a surface alignment and surface separability test to filter out any remaining incorrect poses. This allows them to omit the iterative closest point run that is usually used for fine alignment following point pair feature registration.

The combination of the methods results in improved accuracy and online performance approximately 5 times faster than Drost's algorithm [15] (9.4 s vs 54 s for their implementation of Drost's algorithm). However, this improvement comes at the cost of significantly higher algorithmic complexity. This technique ends up being faster than Drost's because the time saved by filtering out more incorrect poses early on exceeds the time spent doing the more complex filtering.

3.2.3 Weighted Voting

Based on the idea that some points have more discriminative power than others, [55] introduces a modified voting scheme where poses computed from certain points are weighted more heavily than poses computed from other points. They use a maximum-margin approach to learn model-specific weight vectors from automatically-generated training scenes and then use those weight vectors when finding the model in real scenes. Figure 3.4 shows an example of an object where most points provide very little information, but some provide a lot.

[55] was able to achieve better accuracy at high occlusion than uniformly-weighted point pair features and online performance approximately 5 times faster (15 s vs 85 s) than [15], at the expense of adding an offline training phase which took around 10 min per model. However, it is not clear where exactly the online performance difference came from (different implementation, different sampling parameters, etc.). In addition, it is not clear how well each trained vector generalizes to different scene environments.

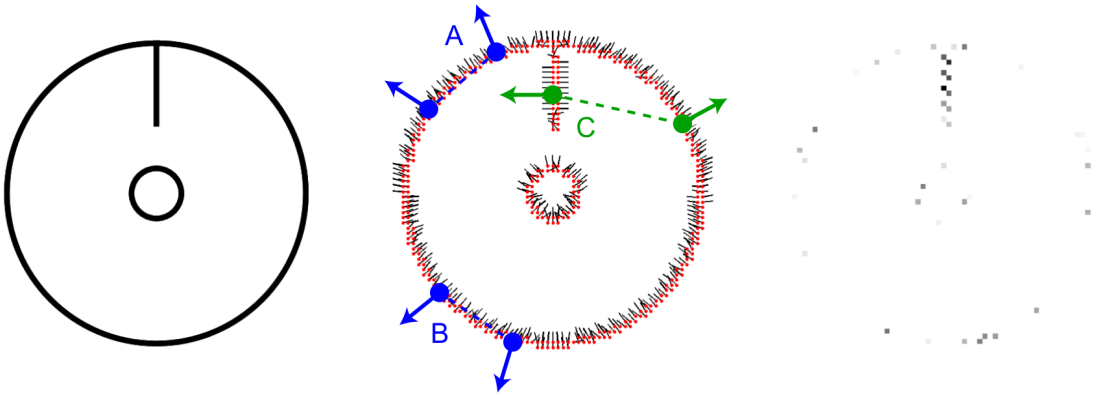


Figure 3.4 – A model where most points (the points on the circle) provide very little information and some (the points on the line) provide a lot. The point pair features of point pairs A and B are identical, while the point pair feature of point pair C can actually be used to orient the model. The image on the right represents the model point weights derived by the algorithm of [55].

3.3 SLAM++ GPU Implementation

[43] makes use of point pair feature registration in a real-time simultaneous localization and mapping (SLAM) [3] implementation. To achieve the necessary, real-time performance, they develop a GPU-based implementation.

3.3.1 Global Model Description

A GPU is very well suited to generating global model descriptions because each individual operation, computing a point pair feature from a pair of points, is computationally simple and independent of all the other computations. However, GPU hash tables, despite significant improvements in recent years, are still computationally expensive due to requiring extensive locking or atomic operations to avoid race conditions [1]. To that end, [43] develops a parallel algorithm to use for storing the model description and looking up point pair features instead of a hash table. We generalize that algorithm and develop a data structure we call a "parallel hash array", which will be described in Chapter 4.

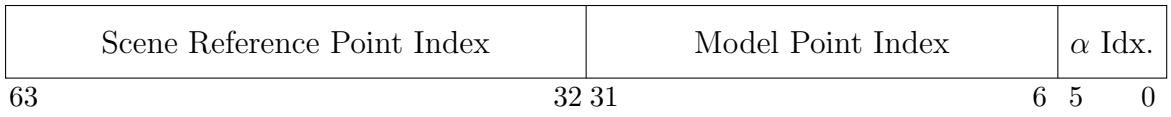


Figure 3.5 – Vote encoded as a 64-bit integer. The 32 bits $[32, 63]$ contain the scene reference point index, the 26 bits $[6, 31]$ contain the model point index, and the 6 bits $[0, 5]$ contain the α angle index

3.3.2 Voting Scheme

The voting scheme developed in [15] uses a single two-dimensional accumulator array to store votes for every scene reference point. Depending on the implementation, one could even use a single three-dimensional array to store votes for all scene reference points. As with hash tables, a large accumulator array is very expensive on a GPU due to the need for locks and atomic operations. As an alternative, [43] encodes each vote in a 64-bit integer using the structure shown in Figure 3.5. The votes can then be counted by sorting them so that identical votes are adjacent to each other and computing their histogram.

An unfortunate property of this algorithm is that, unlike the serial CPU implementation, every individual vote must be stored in memory before being reduced to counts for identical votes. This imposes a memory constraint because, in the worst case where every scene point pair feature matches every model point pair feature, the number of votes will be $O\left(N_{\text{scene}}^2 * N_{\text{model}}^2\right)$. For a modest model and scene with 2000 points each, this would generate on the order of 10^{13} votes, requiring tens of terabytes of memory to store. In practice, the number of votes tends to be less than 0.01% of that, but that still requires gigabytes of memory.

3.3.3 Application Specific Optimizations

While the GPU implementations improve the performance of the algorithm by several orders of magnitude, it is still not sufficient for real-time processing in the general case. To achieve real-time performance, [43] uses several application-specific optimizations and

simplifications.

Because [43] uses point pair feature registration as a component in a SLAM system, their scene is the persistent map of space that they are generating and the registration algorithm is applied repeatedly, after every update of the map. This makes it possible to mask out areas of the scene where objects have already been identified, reducing the number of scene points to process.

Another optimization comes from contextual knowledge of the models and scene based on the intended application of SLAM++, namely identifying furniture and similar repetitive, functional objects such as lamps for augmented reality purposes. This allows them to make certain assumptions about the poses of the objects, such as the fact that all objects are positioned in a certain way relative to a ground plane. More specifically, they assume that all objects are standing upright on the floor, which is identified implicitly based on the first detected object pose. First, this allows all points on the floor to be filtered out, significantly reducing the number of point pair features to process. Second, this allows filtering out transformations which would have the object lying sideways on the ground or floating in the air. In addition to reducing the number of transformations that need to be clustered, this also solves the planar surface problem: the seat of a chair or the top of a table cannot be matched to the floor because then its legs would be poking through the floor.

Finally, due to the nature of the models [43] are trying to identify (large and fairly distinct), they are able to obtain accurate matches with a significantly lower point sampling rate. For furniture, based on both their results and ours, a model size of approximately 400 points (160K point pair features) is sufficient. For more complex (and arbitrarily placed) models, such as those in the Mian dataset [34],[33], a minimum model size of 1600-1800 points (2.5M-3.25M point pair features) is necessary. Based on our testing, the run-time difference between those two model sizes is typically 1-2 orders of magnitude and, depending on the values of parameters that will be discussed in Section 6.1, possibly

more.

Chapter 4

Parallel Hash Array

The parallel hash array we develop is an associative array data structure designed for parallel operations. Specifically, it is optimized for batch insertions and batch lookups, and for data with a lot of hash collisions (discretized point pair features, for example). Not counting the array containing the data we wish to index, the parallel hash array consists of four arrays: a hash array, a hash to data map array, a count array, and a first hash index array.

The hash array is an array of the hashes of the data, with the ith entry being the hash of the ith element of data. The hash to data map array is an array of integers representing indices into the data being stored. It is initialized to a sequence from `0` to `len(data)-1` and is then sorted (along with the hash array) using the hash array as the sort key. By sorting the hash array, we lose the order correspondence between it and the data array. The hash to data map array preserves that correspondence because its ith element is the index of the data element whose hash is the ith hash. Constructing both of these arrays is very efficient on a GPU because computing hashes of the input data and sorting them can both be done efficiently in parallel. At this point we already have a simple search structure: a sorted array of hashes to which we can apply a binary search, and a corresponding array which maps maps indices in the sorted hash array to indices

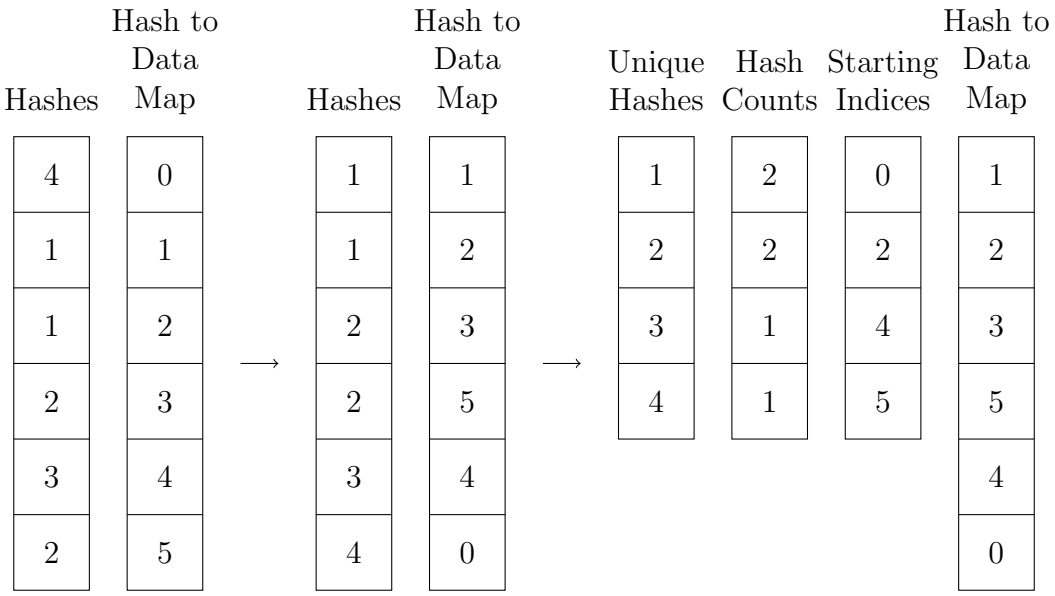


Figure 4.1 – Left: Initial, unsorted array of hashes and their indices (the hash to data map). Middle: Sorted hashes and hash to data map. Right: Unique hashes, their multiplicities, their index in the sorted hash array, and the hash to data map.

in the data array. On a GPU, the binary search can be vectorized for batch lookups.

If the input data has many hash collisions, the sorted array of hashes will have contiguous runs of identical keys. To avoid binary searching through them, we can create an array of unique hashes, an array of starting indices for each run of contiguous values, and an array of lengths of each run of values. This allows us to binary search through a significantly smaller array and then use the corresponding starting index and count to retrieve all matching data. A list of unique hashes and their counts is simply a histogram of the hashes, which can be efficiently computed on a GPU with a sort (which we already did), an inner product, and a reduce by key, each of which can be efficiently parallelized (see Appendix A.1 for details). Finally, the starting indices can be computed with a prefix scan of counts, which can also be efficiently parallelized [22]. The complete process of creating a parallel hash array from a list of data hashes is shown in Figure 4.1.

All of the operations involved in constructing the parallel hash array, including the sort because it is a radix sort[32] and our key size is fixed, are linear in the number elements being inserted, the number of elements already stored, or their sum. The

total time complexity of insertion is thus linear in the size of the parallel hash array after insertion (i.e., size of existing data plus size of data to be added). Deletions have the same complexity. Because each operation can be parallelized very efficiently, it is significantly cheaper to insert keys in large batches than to insert them individually. Point pair feature registration takes this to the extreme by inserting all of the point pair features when constructing the parallel search array and never modifying it after that.

Searching for keys in the parallel hash array is done with a binary search, which has logarithmic complexity. While binary search cannot be efficiently parallelized, it can be efficiently vectorized, so it is significantly cheaper to search for keys in large batches than to search for them individually. As with insertions, each run of the point pair feature registration searches the parallel search array exactly once.

Chapter 5

GPU Transformation Clustering

5.1 Spatial Hashing

The spatial hashing algorithm is an implementation of the spatial subdivision algorithm, which was introduced by [27] for searching geometric data and was applied to three-dimensional data by [29] in the context of molecular modeling, specifically to reduce the number of inter-atom comparisons. More generally, it is a grid-based space partitioning (or spacial index, or spatial access) method. The idea of spatial subdivision is to divide space into hyperrectangular (usually cubic) cells and to, for each cell, construct a list of objects that occupy that cell. To retrieve the neighbors of an object one simply needs to access the cell containing the object and possibly its adjacent cells, depending on the implementation. One way to implement this is by explicitly constructing a multidimensional array representing a bounded region of space and, in each cell, storing a pointer to a list of objects.

However, that implementation has several limitations that usually make it impractical: the bounds of the object coordinates must be known in advance and, unless the objects are relatively uniformly distributed (which is usually not the case), most cells will be empty, thus wasting a huge amount of memory. The latter issue is compounded hor-

ribly in higher-dimensional spaces because the volume of a space grows exponentially as the number of dimensions increases. A much more memory-efficient implementation, and one which can support arbitrarily-located objects, is maintaining a hash table mapping cells, uniquely defined by their coordinates, to lists of objects [7]. This implementation is known as spatial hashing. While spatial hashing is a general-purpose nearest neighbor algorithm, it sees a lot of use in Euclidean applications, specifically collision detection in computer graphics [28], [53].

5.2 Spatial Hash Clustering

In point pair feature registration, our goal is not to "classify" each transformation. Rather, the primary purpose of the clustering is to prune transformations which are likely incorrect. A secondary purpose is to increase the accuracy of the transformation beyond the sampling rate used in the point pair feature discretization in the same vein as super-resolution imaging [39]. In essence, we are trying to combine information (scene and model point correspondences and associated α angles) from similar transformations. Therefore, we do not necessarily need to partition the space of transformations into clusters. Instead, we can create overlapping clusters and allow each transformation to be a member of multiple clusters. This is equivalent to changing the clustering criteria from a "best fit" one to a "good fit" one: a transformation is a member of every cluster into which it fits, not only the one into which it fits best.

Given that criteria, the straightforward approach is an every-to-every comparison. For each transformation, create a cluster centered on that transformation, then iterate through every other transformation and, if it is close enough to the cluster center (i.e., their translation components do not differ by more than a distance threshold and their rotation components don't differ by more than an angle threshold), add it to the cluster. This algorithm has quadratic time complexity in the number of transformations to be

clustered and in the worst case (every transformation is near every other transformation) results in n clusters with n members each, where n is the number of transformations being clustered. However, because each cluster and each membership test is independent, this algorithm can easily be parallelized checking whether the jth transformation fits into the ith cluster in parallel. Even so, due to the quadratic number of comparisons required, this approach is too expensive in practice.

To reduce the number of comparisons required, we can, instead of comparing every transformation to every other transformation, only compare ones which we know are nearby, i.e., perform a nearest neighbor search. One way to do that is with spatial hashing. Spatial hashing is usually implemented using a hash table, but as with the GPU global model description discussed in Section 3.3.1, we replace the hash table with a parallel hash array. For efficiency reasons, we only apply spatial hashing to the translation component and not the rotation component. Applying it to both components would decrease the number of transformation comparisons required, but would exponentially increase the number of adjacent cells necessary to check because we would be working in a 6- or 7-dimensional space (depending on whether an Euler angle or quaternion representation is used) instead of a 3-dimensional space. Instead, for every nearby transformation found with spatial hashing, we do an explicit comparison for the rotational component.

We begin by dividing space into cubic cells with side length d_{dist}, which is the voxel size used to subsample the models and scene and also our clustering translation threshold. This is done sparsely, meaning that we do not construct an array that completely spans a bounded region of space, but instead store only populated cells in the parallel hash array. When searching for the neighbors of a transformation, we check its cell and its 26 adjacent cells, the union of which is a cube with an inscribed sphere of radius d_{dist}. For each transformation, we determine its cell by discretizing its translation component to d_{dist} and computing the hash of the discretized translation vector. To be able to do a batched lookup of all the cell hashes, we also compute and store the hashes of the 26

adjacent cells. The result of this is an array of hashes of length $27 * n_{\text{transformations}}$, with hashes $[27i, 27(i+1) - 1]$ corresponding to the hashes of the cell to which transformation i belongs and its 26 adjacent cells.

We can now iterate through every transformation and, for each transformation, iterate through the 27 corresponding cell hashes. For each cell hash, we find the list of transformations within each cell and, for each transformation, see if it fits in the cluster based on the translation and rotation threshold criteria. If it does, we add the transformation's vote count to the cluster's vote count. Given transformation j and the cluster centered on transformation i, the translation threshold is evaluated as

$$\|\boldsymbol{t}_i - \boldsymbol{t}_j\|_2 < d_{\text{threshold}}, \tag{5.1}$$

where $\boldsymbol{t}_i$ and $\boldsymbol{t}_j$ are respectively the translation components of transformations i and j. The rotation threshold is evaluated as

$$\sqrt{|8\left(1 - \boldsymbol{q}_i \cdot \boldsymbol{q}_j\right)|} < \theta_{\text{threshold}}, \tag{5.2}$$

where $\boldsymbol{q}_i$ and $\boldsymbol{q}_j$ are respectively the quaternion representations of the transformations i and j [20]. The quaternion representation of the rotation component is obtained from the rotation matrix with the method described in [45]. The quaternion distance metric was used instead of a matrix distance metric based one because it is significantly easier to compute a mean of quaternions than a mean of rotation matrices [20], even when including the conversions between quaternions and rotation matrices. The mean rotation of the transformation in a cluster is necessary for the weighted clustering described below, so we use the quaternion representation throughout. The complete process is demonstrated in Algorithm 5.1. In the GPU implementation, `cluster_votes` is pre-allocated and the for loop in `cluster_transformations` is parallelized.

```
def translation_distance(a, b):
    return abs(a.translation - b.translation)

def rotation_distance(a, b):
    return sqrt(abs(8*(1 - a.rotation*b.rotation)))

def sum_nearby_trans_votes(trans_idx, transformations,
                           adjacent_cell_array, cell_to_trans_map,
                           translation_thresh, rotation_thresh):
    this_trans = transformations[trans_idx]
    votes = 0
    for i in range(27):
        adjacent_cell = adjacent_cell_array[27*trans_idx + i]
        for trans in cell_to_trans_map[adjacent_cell]:
            if (translation_distance(this_trans, trans)
                < translation_thresh and
                rotation_distance(this_trans, trans) < rotation_thresh):
                cluster_votes += trans.votes

    return votes

def cluster_transformations(transformations,
                            adjacent_cell_array, cell_to_trans_map,
                            translation_thresh, rotation_thresh):
    cluster_votes = []
    for i in range(len(transformations)):
        cluster_votes.append(sum_nearby_trans_votes(
            i, transformations, adjacent_cell_array, cell_to_trans_map,
            translation_thresh, rotation_thresh))

    return cluster_votes
```

Algorithm 5.1 – Transformation Clustering

One variation on this algorithm is to omit the precise threshold for translations and consider all nearby (i.e., in the same or adjacent cell) transformations to be sufficiently close to include in a cluster. In essence, this replaces the L^2 norm in the threshold comparison with an L^1 norm. The goal of this modification is to improve performance.

Another variation on this algorithm is to, as is done in [15], adjust each cluster's position by setting its position to be the vote-weighted mean of its constituent transfor-

mations. The goal of this modification is to improve accuracy. When implementing this, care must be taken to apply the threshold check to the original position of the cluster and not a position that has been modified by the addition of transformations.

5.3 Advantages Over Tree Based Approaches

The other major class of spatial subdivision algorithms divides space into tree structures, most commonly octrees, k-d trees, range trees, and binary space partition trees, instead of grid structures [7]. Operations on these structures are typically logarithmic in time, as opposed to constant time for spatial hashing, but, by virtue of their hierarchical structure, they are able to handle objects of different sizes more efficiently [7]. For the purposes of transformation clustering, this does not matter because transformations represent orientated points in space, not objects with non-zero volume.

Chapter 6

Implementation and Evaluation

Because the source code for [15]'s implementation and the other point pair feature registration algorithms discussed in Chapter 3 [43], [55], [26] is not publicly available, this project consists of a GPU based re-implementation of the point pair feature registration algorithm based on [43] and [15]. The source code for this project is available at `https://github.com/nicolasavru/ppf-registration-spatial-hashing` under a BSD license.

6.1 Parameters

The implemented algorithm has several tunable parameters. In principle, most of them represent a tradeoff between accuracy and performance. In practice, as will be discussed in Section 6.2.1, the parameters represent that tradeoff only coarsely and can have non-obvious effects on accuracy. Furthermore, these effects can vary between models and scenes, even when the same set of models is used in all cases.

The first parameters are the scene leaf size and τ_d, a model leaf size scaling factor. The scene leaf size is the voxel size used to downsample the scene, as described in section 3.1. A higher leaf size corresponds to a less dense point cloud, while a lower leaf size corresponds to a more dense one. The name "leaf" stems from Point Cloud Library's [42] use of

the term in their implementation of voxel grid resampling. For scenes, the leaf size is directly specified. For models, the leaf size is specified indirectly with the τ_d parameter according to leaf_size $= \tau_d$ diameter (M) [15]. This allows the controlling parameter to be independent of the model scale. A lower τ_d therefore corresponds to a lower leaf size and a higher model density. The computed model leaf size is also used as the discretization rate for the distance component of the model point pair features (Section 3.1.2). The scene point pair feature discretization rate must equal that of the model so that point pair features can actually be matched between the two; therefore, the model leaf size is also used as the discretization rate for the scene point pair features.

The next two parameters are the reference point downsample factor and the vote count threshold, both introduced in [15]. The reference point downsample factor is described in Section 3.1.5. The vote count threshold determines how many candidate poses to keep and include in clustering by specifying, as a fraction of the maximum number of votes received by any pose, a minimum number of votes a pose has to receive to be included. Based on our testing, the pose vote distribution has a long tail of poses with a very small number of votes, so setting the vote count threshold to a small value can lead to significant computational savings. In addition, because the poses with very few votes are usually spurious, they can cause false positives if enough of them are sufficiently near each other to create a large cluster.

There are various additional parameters throughout that we chose based on limited experimentation to find values that worked reasonably well. The discretization rate used for the angular components of point pair features was 12°, the same value used by [15] and [43]. The distance discretization rate discussed above is not intrinsically related to the voxel grid leaf size, but the two are kept equal by [15]. In our spatial hashing implementation, the cell size is set to be equal to the distance discretization rate. When using the L^1 norm variation of the spatial hashing algorithm described in Section 5.2, that cell size is the clustering threshold. When using the L^2 norm variation, the clustering

threshold is an independent parameter, but we set it to be equal to the cell size.

6.2 Evaluation

The accuracy and performance of the spatial hashing clustering algorithm are evaluated using two datasets: a custom-made one consisting of a chair in a lab environment and the standard Mian dataset [34], [33]. This algorithm is directly compared to the CPU-based agglomerative clustering implementation in PCL [42], which was slightly modified and integrated into this project to do a direct comparison of only the clustering algorithms. Tests were run on a workstation with an Intel Xeon E5-1650 v3 CPU and an NVIDIA Tesla K40 GPU.

While the results of other point pair feature algorithms will be mentioned, they cannot be directly compared to the spatial hash clustering results. The point pair feature registration algorithm was found to be unfortunately sensitive to various parameters (primarily voxel grid sampling rate for both the model and the scene and the cluster inclusion threshold), including on a per-model basis, so a comparison without knowing the exact parameters that were used, which are not publicly available, would have little meaning. An alternative would have been to consistently use a set of parameters for all tests, but the source code for the original implementations of the other algorithms is not publicly available.

6.2.1 Cooper Union Chair

The first dataset used to evaluate the performance of this algorithm consists of a single model and two scenes captured with a Microsoft Kinect using KinFu [40], Point Cloud Library's implementation of the Kinect Fusion algorithm [36], and KinFu Large Scale, an extension of KinFu that is able to map larger environments. Due to not being able to compute reliable vertex normals with PCL, normals were computed using MATLAB's

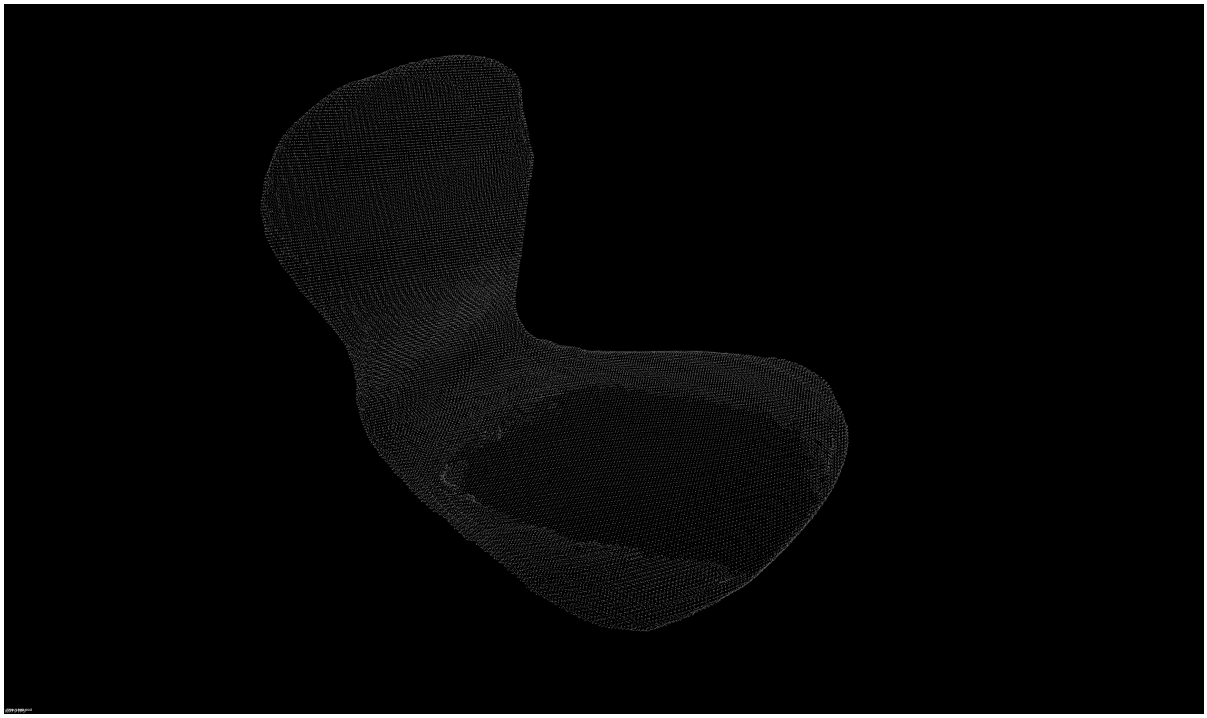


Figure 6.1 – Cooper Union Classroom Chair Model

delaunayTriangulation [51] class and Pascal Getreuer's PLY_IO MATLAB library [18]. No other processing or cleanup was performed on the scenes or model. Ground truth values for this dataset can only be obtained through manual alignment, so this dataset will be used for qualitative analysis only.

The model, shown in Figure 6.1, is a KnollStudio Sprite Stacking Chair, an obscenely expensive classroom chair from Cooper Union's 41 Cooper Square building and the single most uncomfortable chair I have ever sat upon. The chair's legs are missing because they were not reliably visible to the Kinect, due to a combination of them being very thin and reflective. The chair model contains 506,193 points. The first scene, shown in Figure 6.2, is a capture of a Cooper Union electrical engineering lab made with the KinFu Large Scale module (`pcl/gpu/kinfu_large_scale`) in PCL. This scene contains 1,204,946 points. The chair from Figure 6.1 is visible near the center of the image. The second scene, shown in Figure 6.3, is a capture of a smaller section of the same lab (viewed from a different angle in this rendering), but made with the KinFu module (`pcl/gpu/kinfu`).

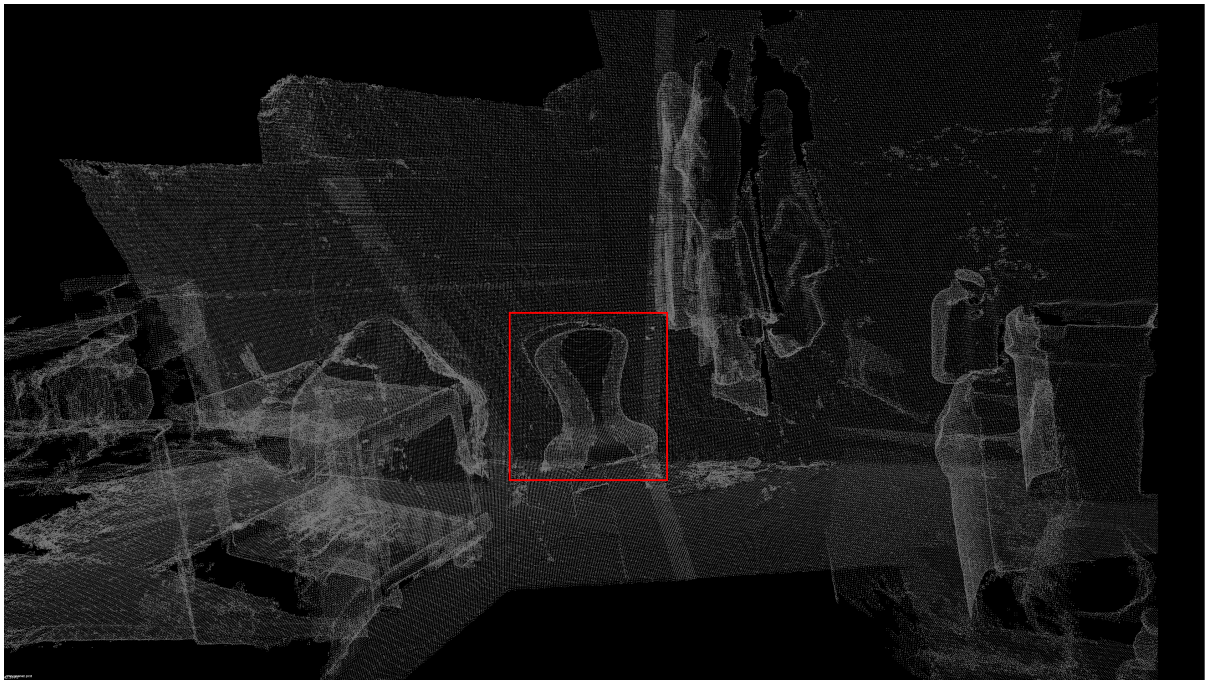


Figure 6.2 – Scene 1. The chair from Figure 6.1 is in the red box.

While covering a smaller area than the first scene, this scene contains 2,861,778 points, making it significantly denser. Scene 3, shown in Figure 6.4, is a subset of Scene 2 (viewed from a different angle) that was extracted using Meshlab [12] and has 243,964 points.

500K-1M points would require terabytes of memory to store, so we need to downsample the model and scenes first. Given the memory constraint described in Section 3.3.2, the following constraint must be satisfied in order to fit into the memory of the Tesla K40: $N_{\text{scene}} * N_{\text{model}}/n < 2000000$, where n is the scene reference point downsample factor discussed in Section 3.1.4. 2000000 is an estimate based on the actual observed memory usage.

To satisfy the memory constraint for Scene 1, we downsample the scene to 15388 points (corresponding to a voxel size of 0.051), the chair model to 1649 points (corresponding to a $\tau_d = 0.04$), and set $n = 20$. And, as seen in Figure 6.5, the chair model (in red) is not correctly aligned with the chair in the scene. Instead, the seat of the chair is aligned with a wall. This is an example of the planar surface problem described in Section 3.2. This problem could be resolved by increasing the density of the scene, but

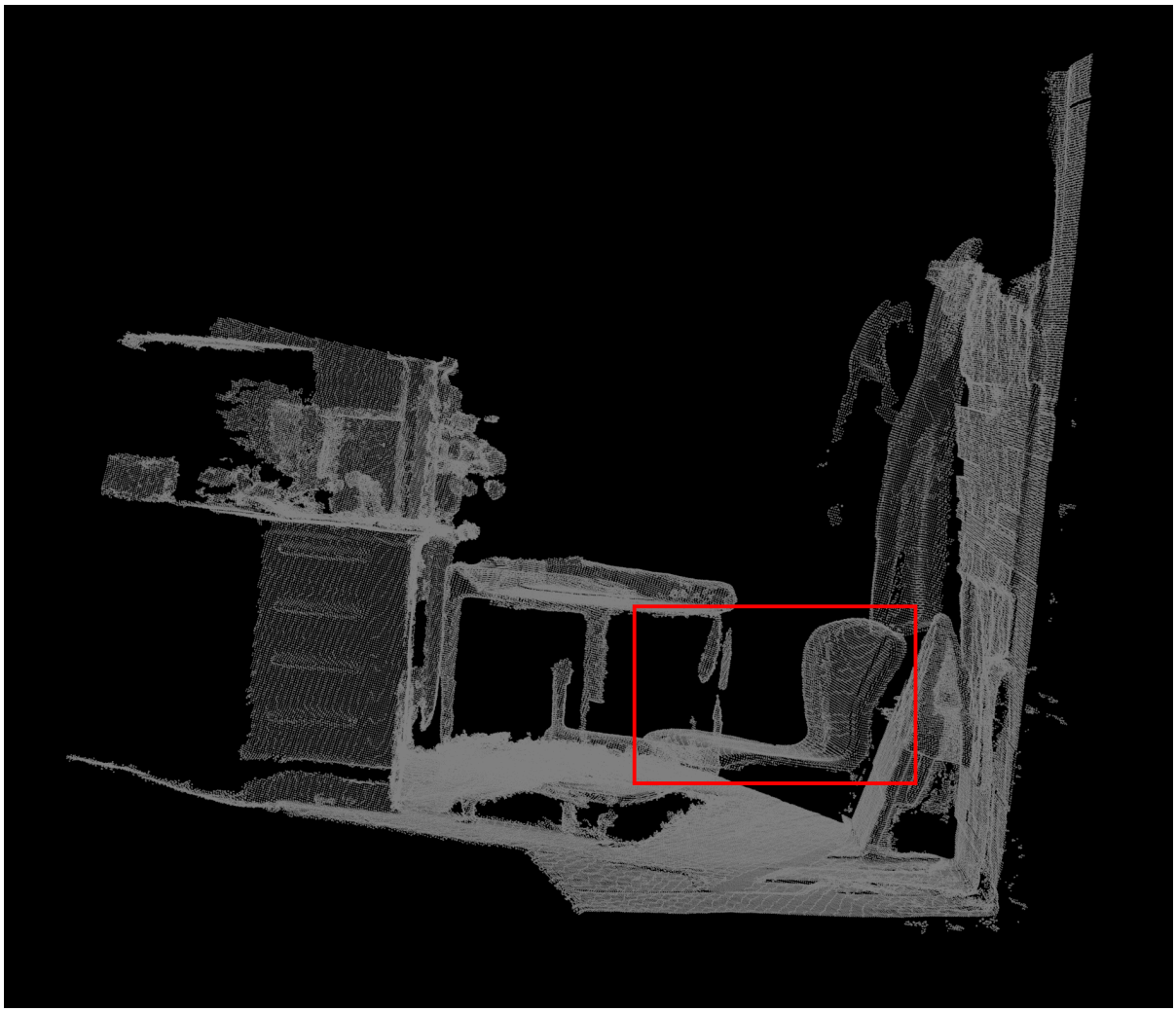


Figure 6.3 – Scene 2

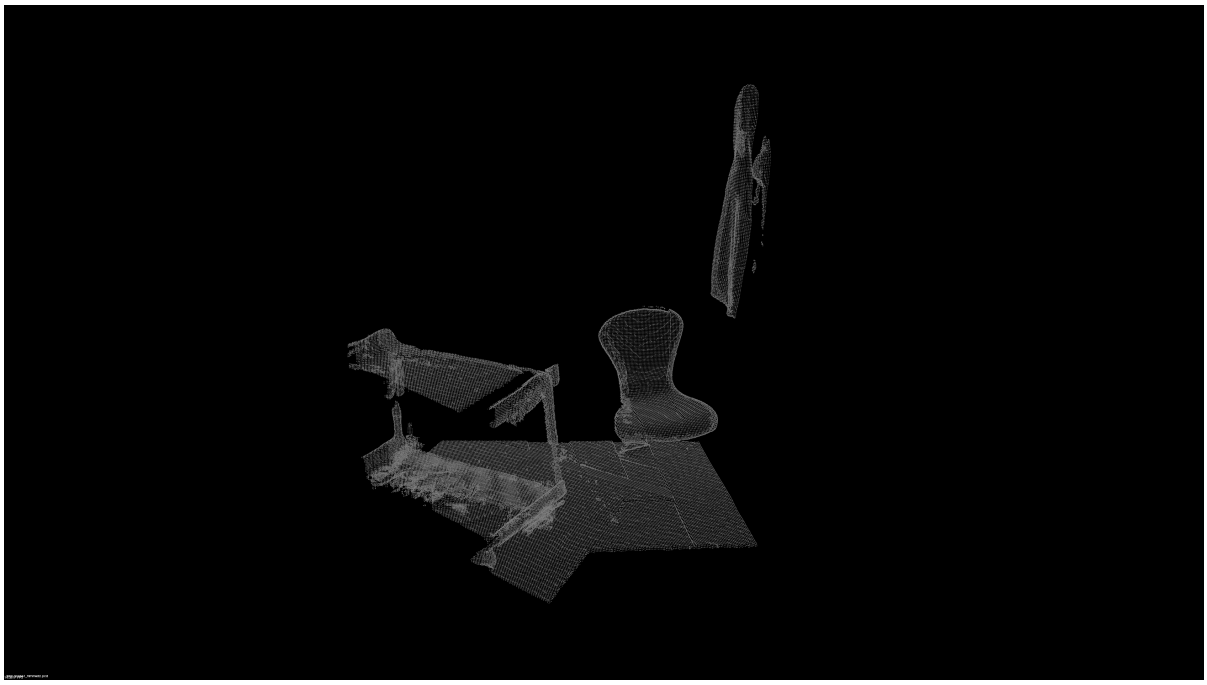


Figure 6.4 – Scene 3

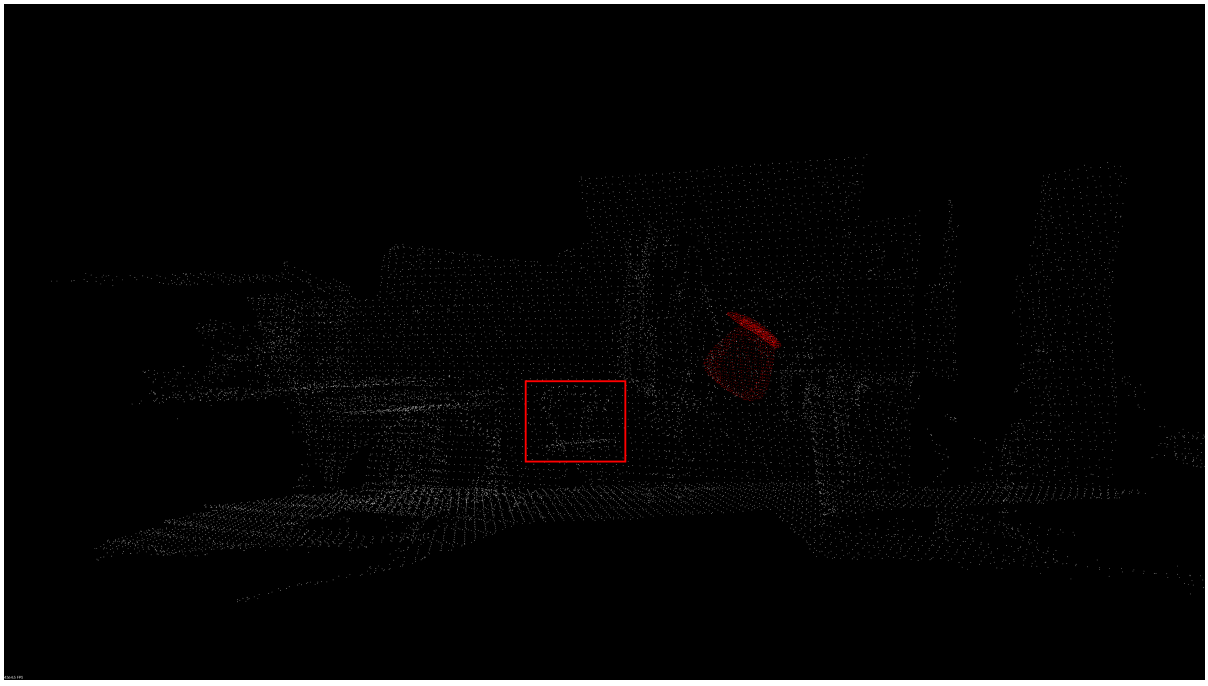


Figure 6.5 – Scene 1 incorrect alignment. The seat of the chair is aligned with the wall instead of with the chair in the scene, which is in the red box.

Figure 6.6 – Scene 2 incorrect alignment. The seat of the chair is aligned to a flat object that is leaning against the wall and the back of the chair is approximately aligned with the floor. The scene chair is in the red box.

that is not possible due to the memory constraints.

Because Scene 2 is physically smaller, we can use a higher density for the scene by decreasing the voxel size to 0.038 (resulting in 12251 points). This time, the seat of the chair is aligned to a flat object that is leaning against the wall and the back of the chair is approximately aligned with the floor (Figure 6.6). Scene 2 is still too big to get a correct match.

Scene 3 is smaller still, so we can further decrease the voxel size to 0.035 (3612 points). Now the chair is properly aligned (Figure 6.7). However, further increasing the scene density by setting the voxel size to 0.30 (4903 points) results in an incorrect match again (Figure 6.8). A correct match can be obtained again by decreasing n from 20 to 10.

Similar non-obvious results can be obtained by tweaking other parameters. For example, running the algorithm on Scene 1 with the same density parameters and changing the vote count threshold from 0.3 to 0.7 (i.e., throwing out significantly more data) re-

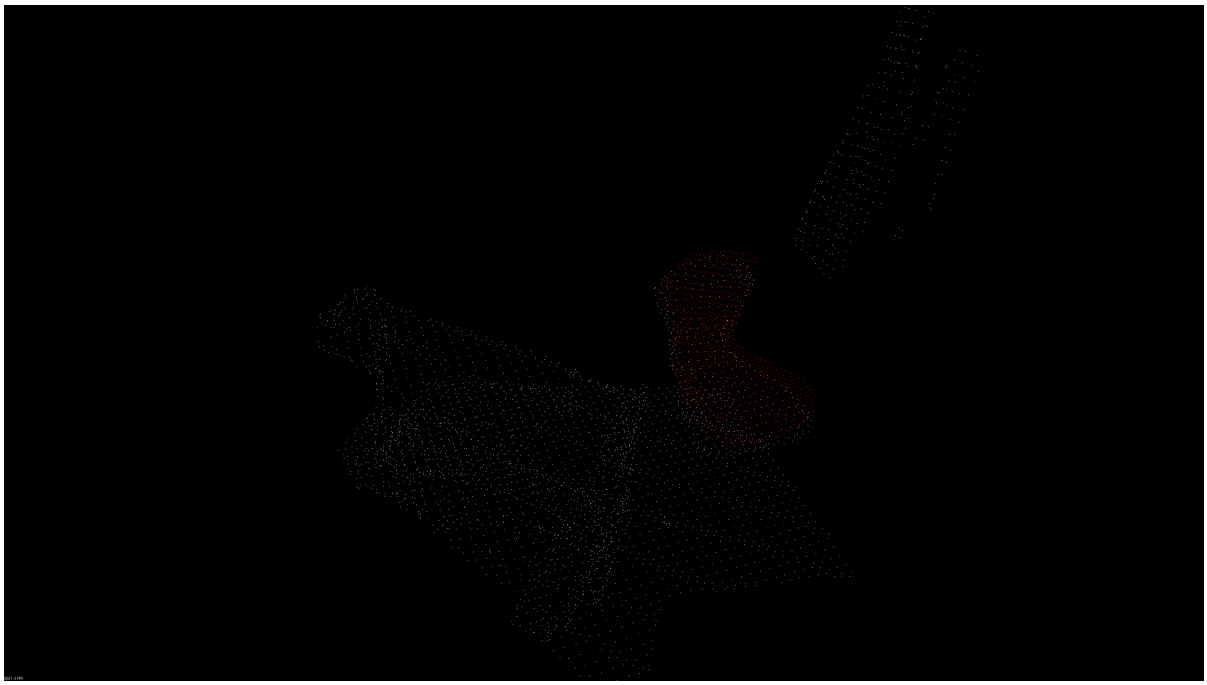


Figure 6.7 – Scene 3 correct alignment. The chair model is properly aligned with the chair in the scene.

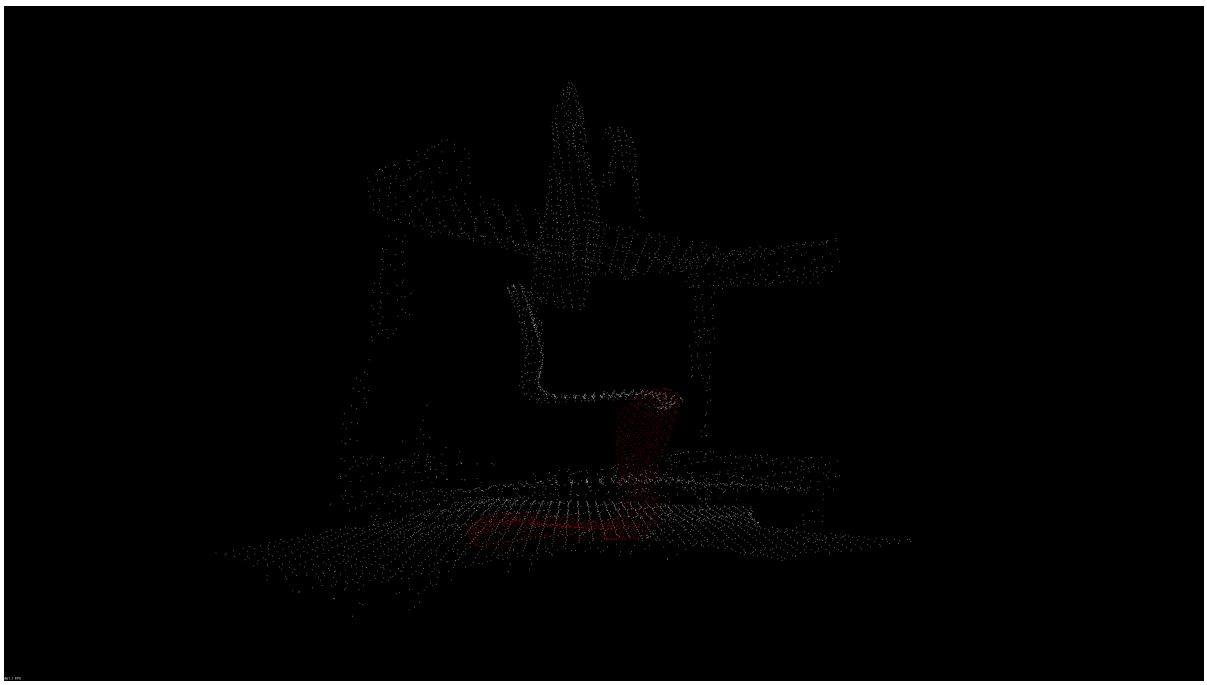


Figure 6.8 – Scene 3 incorrect alignment. The seat of the chair is aligned with the floor.

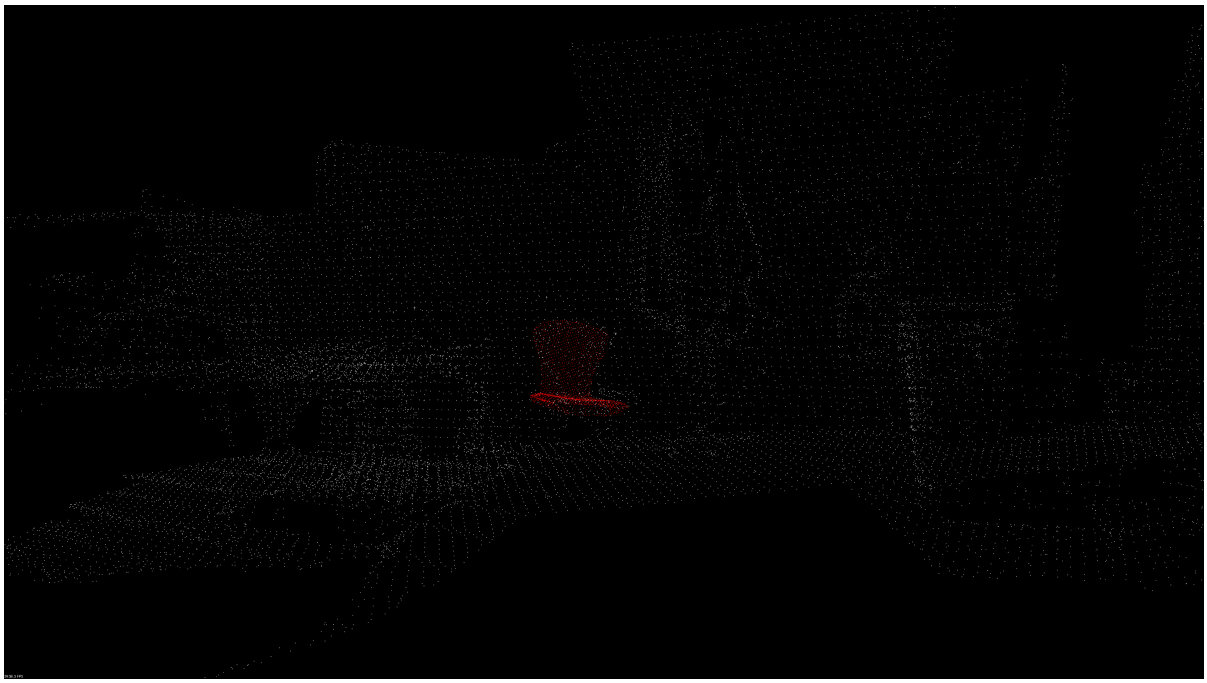


Figure 6.9 – Scene 1 correct alignment. The chair model is properly aligned with the chair in the scene.

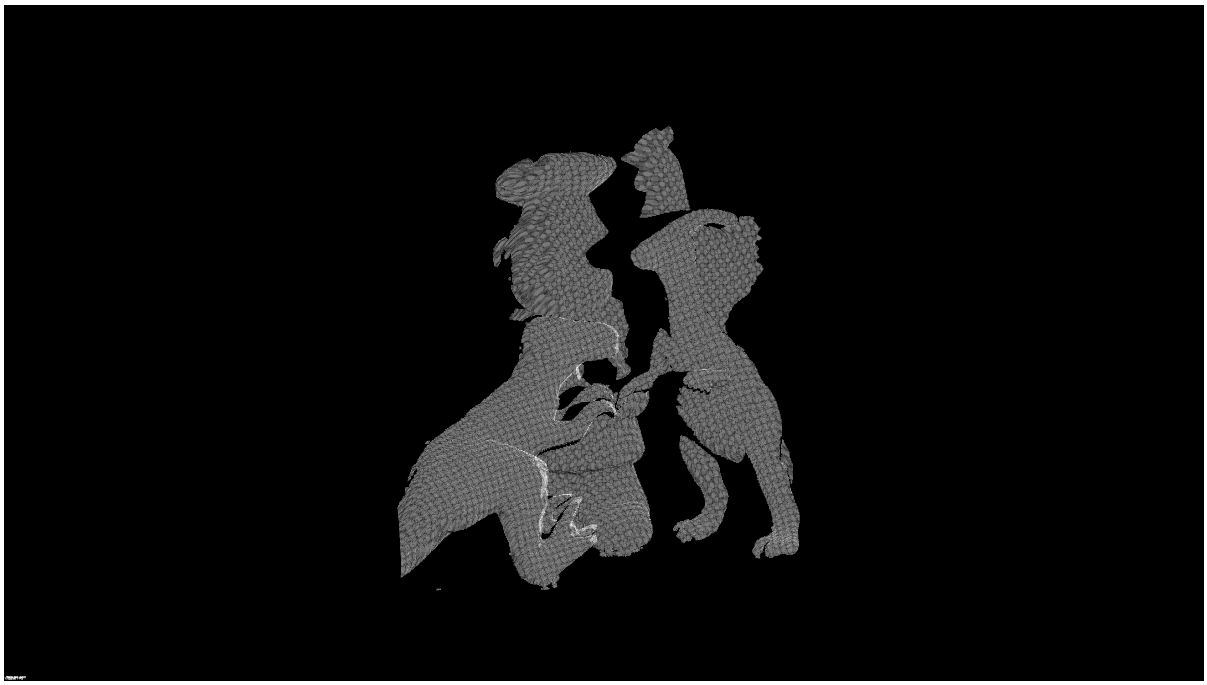
sults in a correct match (Figure 6.9). However, there is no significant difference between running the algorithm with a vote count threshold of 0.3 and of 0.7 when running it on Scene 2 with the same density parameters as above. In other cases, accuracy decreased when changing the vote count threshold from 0.3 to 0.7, as one would expect.

Re-running each of the above test cases with CPU-based agglomerative clustering instead of GPU-based spatial hash clustering results in no significant difference in accuracy and only a small difference in run time with the vote count threshold set to 0.7: 7.4–7.7 s on the GPU and 7.7–8.0 s on the CPU for Scenes 1 and 2 and 2.2 s for both GPU clustering and CPU clustering for Scene 3. With a vote count threshold of 0.3, GPU clustering ran in approximately the same amount of time, while CPU agglomerative clustering did not complete within 10 min. The parallel logarithmic complexity of GPU clustering allows it to scale to significantly larger amounts of data with no significant difference in run time.

Figure 6.10 – Mian Dataset Models [34].

6.2.2 Mian Dataset

The Mian dataset [34], [33] consists of five 3D models (chef, parasaurolophus, T-rex, chicken, rhino), of which the first four are actually used, and 50 scenes, each containing those four models in different positions and orientations and with varying amounts of occlusion. The models and scenes were captured with a Minolta Vivid 910 3D scanner. The models are shown in Figure 6.10 and the first scene from the dataset is shown in Figure 6.11. Normals were not provided, so they were computed in the same was as the EE lab dataset normals were computed. As with the EE lab dataset, the models and scenes were downsampled to between 800 and 2000 points. The Mian dataset also exhibited high sensitivity to the downsampling parameters, so, for the purposes of comparing GPU spatial hashing clustering to CPU agglomerative clustering, a set of parameters that performed decently was chosen. Beyond finding parameters that worked reasonably well on several scenes, we did not attempt to find a set of parameters that performed optimally on any or all scenes. For that reason, there is little value in comparing the



(a)

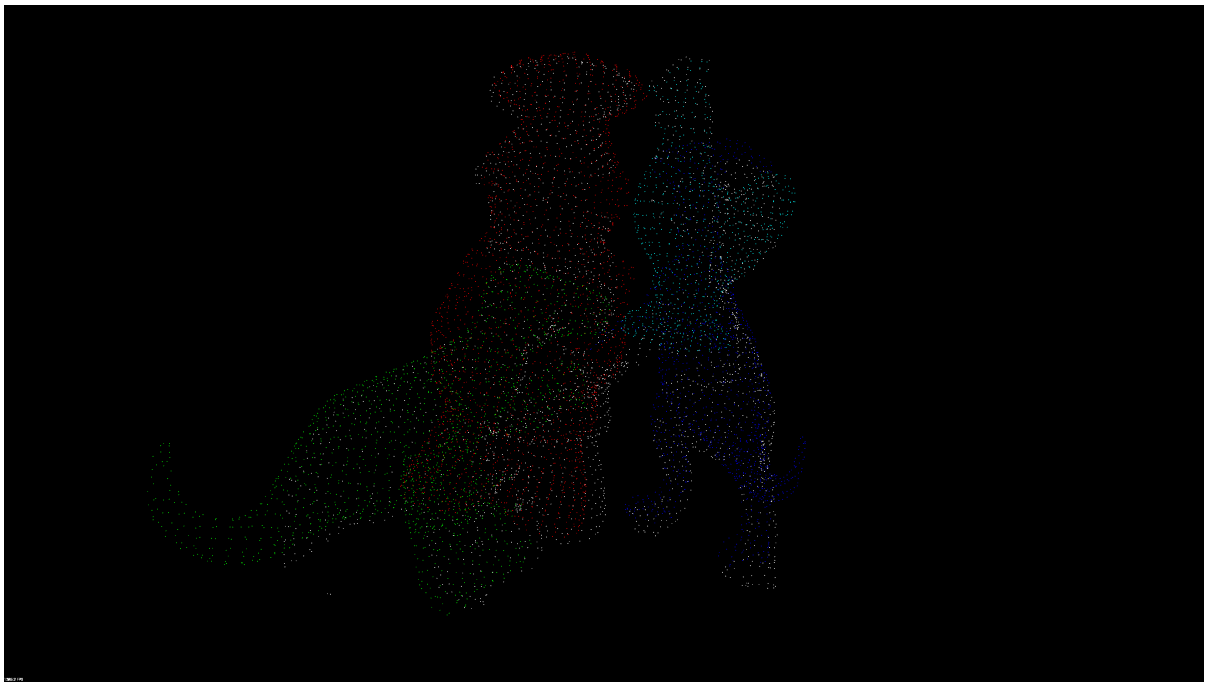
(b)

Figure 6.11 – Mian Dataset Scene 1. (a) contains a front view, with the chef in the top-left, the T-rex in the bottom left, the chicken in the top right, and the parasaurolophus in the bottom right. (b) contains the same scene viewed from the right side.

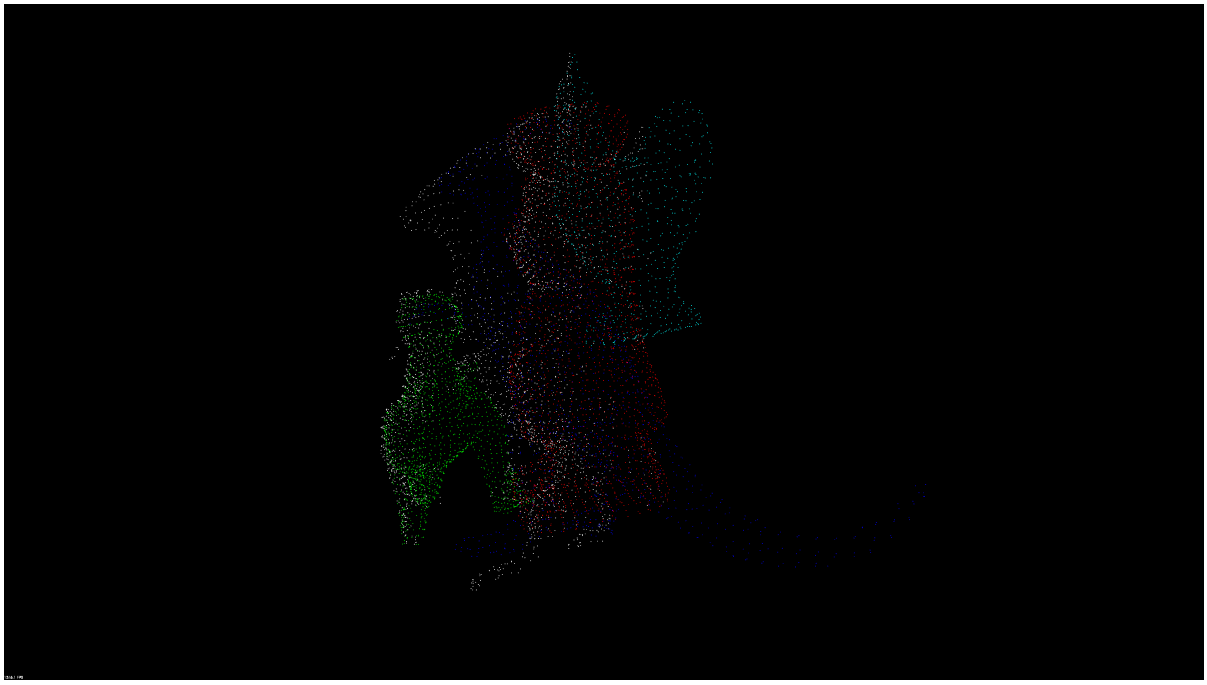
absolute accuracy to that of [15]'s implementation. Figure 6.12 demonstrates the result of matching all four models in the Scene 1.

Accuracy is evaluated based on the translation and rotation error of the found poses. As [15] does, translation errors are normalized based on the diameter of the model to account for differences in model scale. Figures 6.13 and 6.14 respectively compare the translation and rotation errors of the two clustering algorithms. CPU clustering is marginally more accurate, but because the output of point pair feature registration is usually post-processed with ICP for fine alignment, this difference is not significant in practice. However, the GPU clustering was significantly faster: the mean time to lookup a model with GPU clustering was 1913 ms, compared to 34,278 ms with CPU clustering. The median times show a smaller but still significant difference: 1592 ms with GPU clustering and 3738 ms with CPU clustering. Both of these are a significant improvement over the pure CPU implementation of [15], which required approximately 85 s per lookup.

Averaging and using an L^1 norm instead of an L^2 norm had no significant impact on performance. Both of those modifications actually decreased the accuracy by a small amount, but that difference was within the range that could be achieved by tweaking some of the algorithm's many parameters.

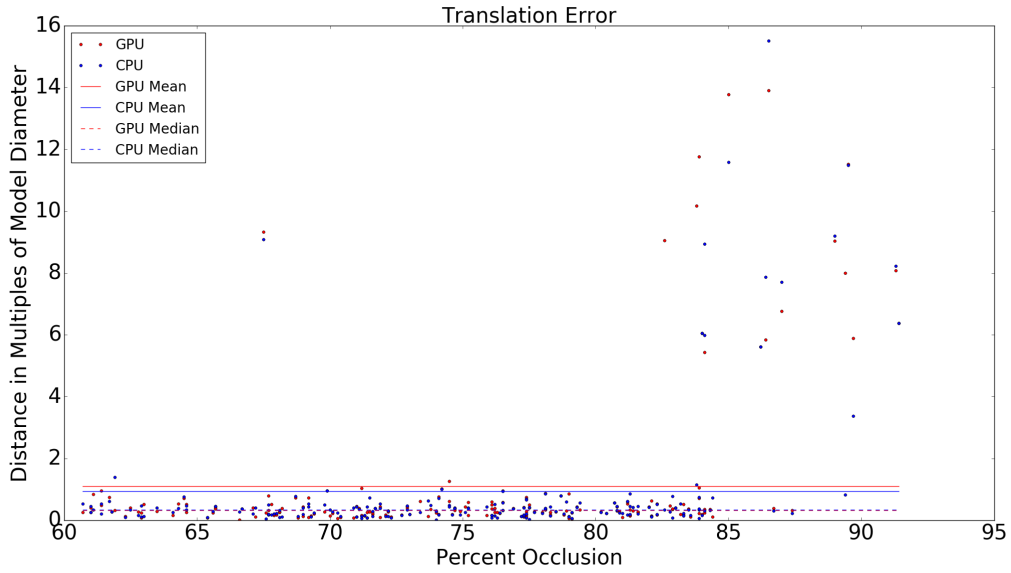


(a)

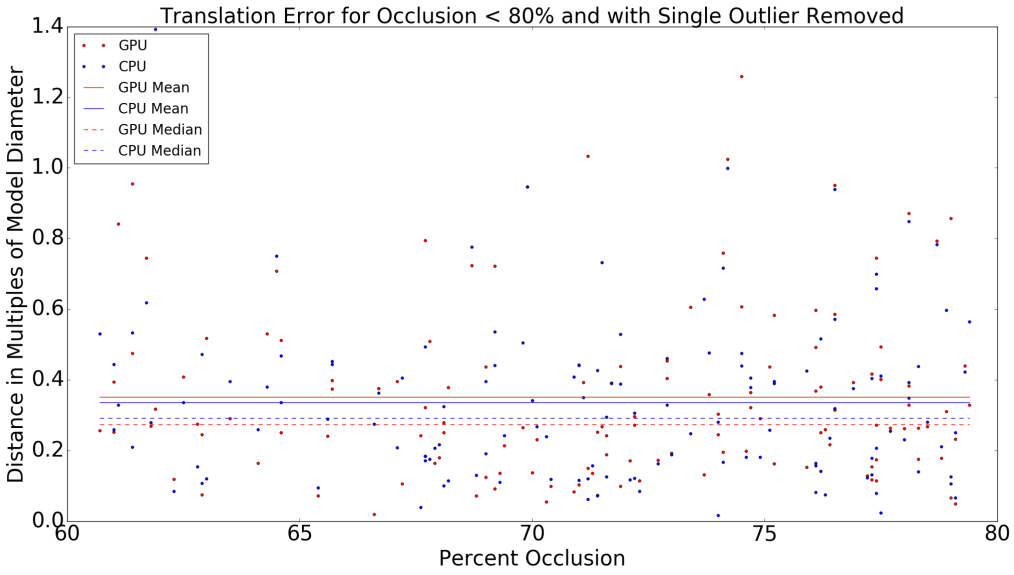


(b)

Figure 6.12 – Mian Dataset Scene 1 alignment. (a) contains a front view, with the scene in gray, the aligned chef model in red, the aligned T-rex model in green, the aligned chicken model in cyan, and the aligned parasaurolophus model in blue. (b) contains the same scene viewed from the right side.

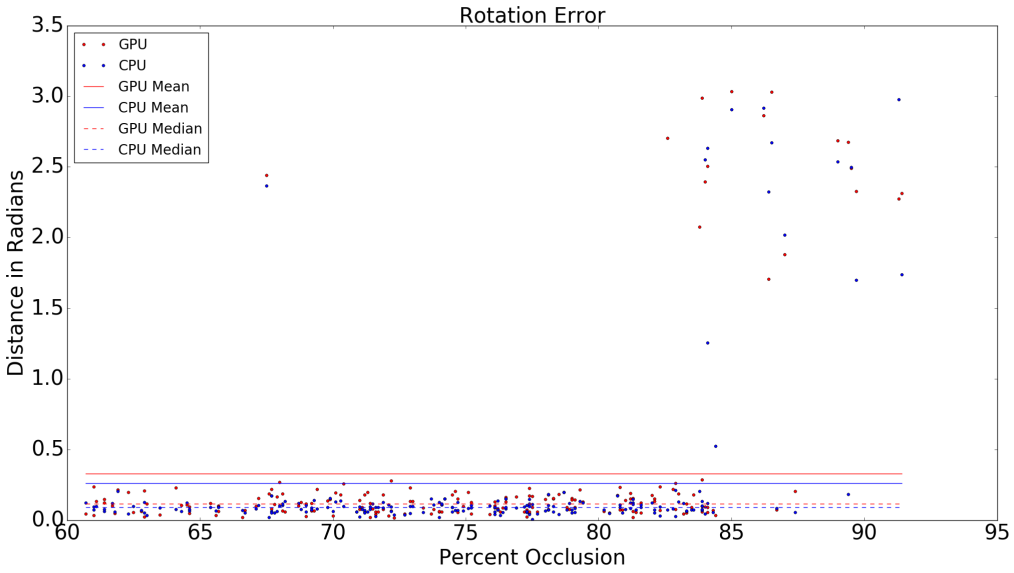


(a)

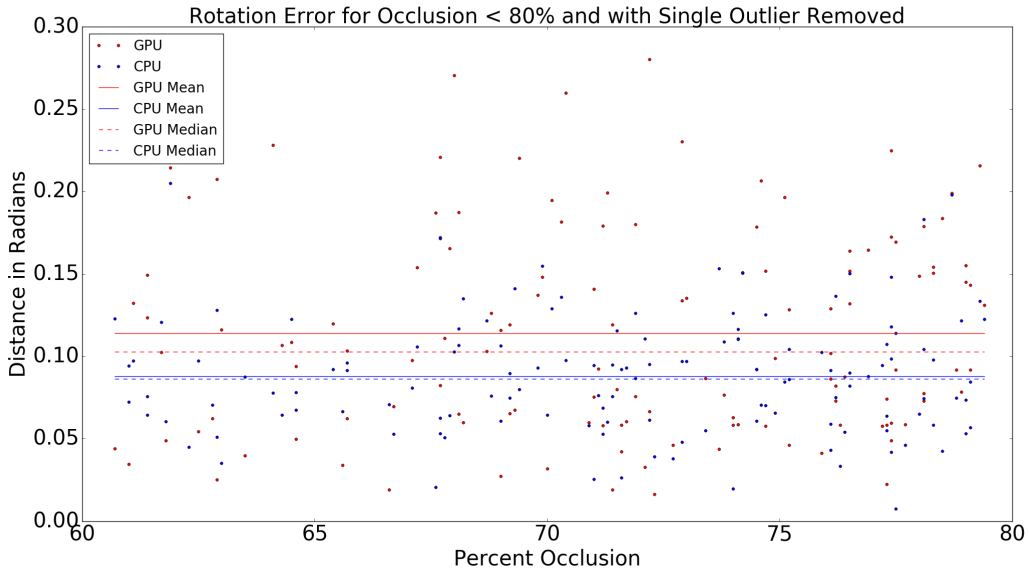


(b)

Figure 6.13 – Translation error. GPU error is in red and CPU error is in blue. Solid lines represent means and dashed lines medians. (b) restricts the calculation range to occlusion less than 80 % and removes the single remaining outlier.



(a)



(b)

Figure 6.14 – Rotation error. GPU error is in red and CPU error is in blue. Solid lines represent means and dashed lines medians. (b) restricts the calculation range to occlusion less than 80 % and removes the single remaining outlier.

Chapter 7

Conclusion and Future Work

We introduced a new GPU-based clustering algorithm based on spatial hashing for use in point pair feature registration. It replaces the CPU-based agglomerative clustering algorithm that is used in existing implementations, removing the last CPU-based component of point pair feature registration and allowing the complete algorithm to be efficiently implemented on a GPU. We also developed the parallel hash array, an associative array data structure with support for parallel operations that can be used as an alternative to hash tables on GPUs. We provide open source implementations of both of these algorithms and, along with them, the first open source GPU point pair feature registration implementation. We tested our algorithm on real-world data and showed that its accuracy is comparable to CPU agglomerative clustering, while its run time is significantly faster.

While our implementation does have some optimization, little work was spent optimizing the CUDA kernels. With additional optimization, we expect that the run time could be significantly reduced. In addition, when testing our point pair feature registration implementation, we found that point pair feature registration accuracy and run time are sensitive to a variety of parameters. While this does provide the ability to tailor the algorithm to a variety of applications by exchanging accuracy for run time, it can

also lead to suboptimal performance because the effects of some parameters are dataset-dependent and not easily predictable. Future work could investigate the parameter space more thoroughly and provide better guidance for tuning the algorithm.

Appendices

Appendix A

CUDA

CUDA [37] is a proprietary parallel computing platform developed by NVIDIA for general purpose computing on graphics processing units. CUDA supports C, C++, and Fortran frontends and extends those languages with an API and non-standard syntax to launch compute kernels, functions which are executed on the GPU. During each kernel launch, many instances of the kernel function are executed in parallel, each operating on different data. On modern GPUs with several thousand individual CUDA cores, this compute model can be used to process large amounts of data significantly faster than is possible on a CPU.

Algorithm A.1 displays an example CUDA program from [44] that copies two vectors into GPU memory, adds them in parallel, and copies the result back into CPU memory.

```
#define N  (33 * 1024)

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );

    // launch the kernel
    add<<<128,128>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

Algorithm A.1 – CUDA Vector Sum [44].

A.1 Thrust

Thrust [23] is an STL-like library for CUDA the provides GPU-based templated data structures and algorithms such as vectors, iterators, sorts, transformations, reductions, etc. Like the C++ STL, it enables writing much higher-level and more functional code than is practical with CUDA C/C++ alone. Algorithm A.2 shows how the Algorithm A.1 can be re-written using Thrust. Note how Thrust handles the details of copying memory to and from the GPU and automatically determines optimal launch parameters for the kernel.

```
#define N  (33 * 1024)

int main( void ) {
    thrust::host_vector<int> a(N);
    thrust::host_vector<int> b(N);
    thrust::host_vector<int> c(N);

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    // allocate memory and copy the arrays 'a' and 'b' to the GPU
    thrust::device_vector dev_a(a);
    thrust::device_vector dev_b(b);

    // launch the kernel
    thrust::transform(dev_a.begin(), dev_a.end(),
                      dev_b.begin(),
                      dev_c.begin(),
                      thrust::plus<int>());

    // copy the array 'c' back from the GPU to the CPU
    thrust::host_vector<int> c(dev_c);

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

Algorithm A.2 – CUDA Vector Sum Using Thrust

A more interesting Thrust example is Algorithm A.3, which is an example implementation of a histogram function. When the device_vector is constructed, Thrust automatically copies the data onto the GPU. The sort, inner_product, and reduce_by_key functions are all executed on the GPU and each one processes its data in parallel.

```
template <typename Vector1, typename Vector2, typename Vector3>
void histogram(const Vector1& input,
               Vector2& histogram_values,
               Vector3& histogram_counts){
    typedef typename Vector1::value_type ValueType; // input value type
    typedef typename Vector3::value_type IndexType; // histogram index type

    // Copy data to device: [4, 1, 2, 4, 5]
    thrust::device_vector<ValueType> data(input);

    // Sort data: [1, 2, 4, 4, 5]
    thrust::sort(data.begin(), data.end());

    // Compute number of unique values by computing
    // the inner product of [1, 2, 4, 4] and [2, 4, 4, 5]
    // using 1 as the output init value, plus as the addition operator
    // and not_equal_to as the multiplication operator:
    // 1 + not_equal_to(1, 2) + not_equal_to(2, 4) + not_equal_to(4, 4) +
    // not_equal_to(4, 5) = 1 + 1 + 1 + 0 + 1 = 4
    IndexType num_bins = thrust::inner_product(
        data.begin(), data.end() - 1,
        data.begin() + 1,
        IndexType(1),
        thrust::plus<IndexType>(),
        thrust::not_equal_to<ValueType>());
    histogram_values.resize(num_bins);
    histogram_counts.resize(num_bins);

    // Compute the multiplicity of each unique value reducing a vector
    // of ones using contiguous runs of unique values as the keys:
    // values: [1, 2, 4, 4, 5]      -> [1, 2, 4,    5]  -> [1, 2, 4, 5]
    // counts: [1, 1, 1, 1, 1, ...] -> [1, 1, 1 + 1, 1] -> [1, 1, 2, 1]
    thrust::reduce_by_key(data.begin(), data.end(),
                          thrust::constant_iterator<IndexType>(1),
                          histogram_values.begin(),
                          histogram_counts.begin());
}
```

Algorithm A.3 – Thrust Histogram

Appendix B

Evolutionary Weighted Voting

Based on [55], we attempted to implement a similar weighted voting scheme using evolutionary algorithms, specifically particle swarm optimization and differential evolution [13], instead of a maximum-margin optimization approach. Evolutionary algorithms have the advantage of being simpler to implement and are generally very parallelizable, making them ideal for GPU implementations. We applied an individual weight to every model point and, using the differential evolution implementation in libCudaOptimize [35], an open source CUDA-based metaheuristic optimization library, attempted to find an optimal weight vector. For training, we used the same strategy as [55]: dynamically generating training scenes, attempting to find the model in those scenes, and using the distance between the found poses and the ground truth poses that were used to generate the scenes as the objective function. This attempt was not successful.

Fundamentally, the problem was one of dimensionality. Because evolutionary algorithms are essentially heuristic searches, they need reasonably good coverage of the solution space. As such, they scale poorly for higher-dimensional problems because the volume of the solution space increases exponentially in the number of dimensions [13]. Maintaining good coverage requires increasing the number of particles searching the space proportionally and, because each particle has to evaluate the object function every gen-

eration, the computational complexity grows exponentially also. In addition, higher dimensional (more than a couple hundred) problems are often sufficiently complex that a heuristic search simply cannot find a good solution, even when with good coverage of the solution space [13].

In this case, the dimensionality of the problem is the number of points in the down-sampled model, which is usually between 800 and 2000 for point pair feature registration. Differential evolution usually requires a swarm size between three and ten times the dimensionality of the problem [13]. Using the low end of both ranges, we need at least 2400 particles. Our objective function is the point pair feature registration algorithm itself, which takes several hundred to several thousand milliseconds to run, as discussed in Chapter 6). Again, using the low end of that range, we have a run time on the order of $2400 * 500\,\text{ms} = 12\,\text{s}$ per generation, an unreasonably optimistic estimate. Given that hundreds or thousands or generations are necessary for a problem of this complexity [19], the training run time will be on the order of hours, an unreasonably long amount of time given the speed and performance of the non-weighted algorithm. However, even when training overnight, we were unable to obtain good results using the trained weights.

Bibliography

[1] Dan Anthony Feliciano Alcantara. "Efficient hash tables on the gpu". University of California Davis, 2011.

[2] Juan Andrade-Cetto and Michael Villamizar. "Object Recognition". In: *Wiley Encyclopedia of Electrical and Electronics Engineering* (2000).

[3] Josep Aulinas et al. "The SLAM problem: a survey." In: *CCIA*. Citeseer, 2008, pp. 363–371.

[4] Dana H. Ballard. "Generalizing the Hough Transform to Detect Arbitrary Shapes". In: *Pattern Recognition* 13.2 (1981), pp. 111–122.

[5] Neslihan Bayramoglu and A. Aydin Alatan. "Shape Index SIFT: Range Image Recognition Using Local Features". In: IEEE, Aug. 2010, pp. 352–355.

[6] Jon L. Bentley, Donald F. Stanat, and E.Hollins Williams. "The complexity of finding fixed-radius near neighbors". In: *Information Processing Letters* 6.6 (Dec. 1977), pp. 209–212.

[7] Jon Louis Bentley and Jerome H. Friedman. "Data structures for range searching". In: *ACM Computing Surveys (CSUR)* 11.4 (1979), pp. 397–409.

[8] Paul J. Besl and Neil D. McKay. "A method for Registration of 3-D Shapes". In: *Robotics-DL tentative*. International Society for Optics and Photonics, 1992, pp. 586–606.

[9] Tolga Birdal and Ilic Slobodan. "Point Pair Features Based Object Detection and Pose Estimation Revisited". In: 3D Vision. IEEE, Oct. 2015, pp. 527–535.

[10] Lisa Gottesfeld Brown. "A survey of image registration techniques". In: *ACM computing surveys (CSUR)* 24.4 (1992), pp. 325–376.

[11] Changhyun Choi et al. "Voting-based pose estimation for robotic assembly using a 3D sensor". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 1724–1731.

[12] Paolo Cignoni et al. "Meshlab: an Open-Source Mesh Processing Tool". In: *Eurographics Italian Chapter Conference*. Vol. 2008. 2008, pp. 129–136.

[13] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. "Differential Evolution: A Survey of the State-of-the-Art". In: *IEEE Transactions on Evolutionary Computation* 15.1 (Feb. 2011), pp. 4–31.

[14] Rui Pimentel de Figueiredo, Plinio Moreno, and Alexandre Bernardino. "Fast 3D object recognition of rotationally symmetric objects". In: *Pattern Recognition and Image Analysis*. Springer, 2013, pp. 125–132.

[15] Bertram Drost et al. "Model globally, match locally: Efficient and robust 3D object recognition". In: *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE, 2010, pp. 998–1005.

[16] Jason Eisner. "State-of-the-Art Algorithms for Minimum Spanning Trees - A Tutorial Discussion". In: (1997).

[17] Martin A. Fischler and Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6 (June 1, 1981), pp. 381–395.

[18] Pascal Getreuer. *PLY_IO - Read or Write a PLY File*. Version 25 February 2007. URL: `http://people.sc.fsu.edu/~jburkardt/m_src/ply_io/ply_io.html`.

[19] Matthew S. Gibbs et al. "Minimum number of generations required for convergence of genetic algorithms". In: *2006 IEEE International Conference on Evolutionary Computation*. IEEE, 2006, pp. 565–572.

[20] Claus Gramkow. "On averaging rotations". In: *Journal of Mathematical Imaging and Vision* 15.1-2 (2001), pp. 7–16.

[21] Yulan Guo et al. "3D Object Recognition in Cluttered Scenes with Local Surface Features: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (Nov. 1, 2014), pp. 2270–2287.

[22] Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Chapter 39. Parallel Prefix Sum (Scan) with CUDA". In: *GPU Gems 3*. Ed. by Hubert Nguyen. OCLC: ocn141855070. Upper Saddle River, NJ: Addison-Wesley, 2008.

[23] Jared Hoberock and Nathan Bell. *Thrust*. URL: `https://thrust.github.io/`.

[24] John F. Hughes. *Computer Graphics: Principles and Practice*. Third edition. Upper Saddle River, New Jersey: Addison-Wesley, 2014.

[25] Michael Kazhdan, Thomas Funkhouser, and Szymon Rusinkiewicz. "Rotation invariant spherical harmonic representation of 3 d shape descriptors". In: *Symposium on geometry processing*. Vol. 6. 2003, pp. 156–164.

[26] Eunyoung Kim and Gerard Medioni. "3D object recognition in range images using visibility context". In: *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011, pp. 3800–3807.

[27] Donald Ervin Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Reading, Massachusetts: Addison-Wesley, 1973.

[28] Scott Le Grand. "Chapter 32. Broad-Phase Collision Detection with CUDA". In: *GPU Gems 3*. Ed. by Hubert Nguyen. OCLC: ocn141855070. Upper Saddle River, NJ: Addison-Wesley, 2008.

[29] Cyrus Levinthal. *Molecular model-building by computer*. WH Freeman and Company, 1966.

[30] Ming-Yu Liu et al. "Fast object localization and pose estimation in heavy clutter for robotic bin picking". In: *The International Journal of Robotics Research* 31.8 (2012), pp. 951–973.

[31] David G. Lowe. "Object Recognition from Local Scale-Invariant Features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee, 1999, pp. 1150–1157.

[32] Duane Merrill and Andrew Grimshaw. "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing". In: *Parallel Processing Letters* 21.02 (2011), pp. 245–272.

[33] A. Mian, M. Bennamoun, and R. Owens. "On the Repeatability and Quality of Keypoints for Local Feature-based 3D Object Retrieval from Cluttered Scenes". In: *International Journal of Computer Vision* 89.2-3 (Sept. 2010), pp. 348–361.

[34] Ajmal S. Mian, Mohammed Bennamoun, and Robyn Owens. "Three-dimensional model-based object recognition and segmentation in cluttered scenes". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 28.10 (2006), pp. 1584–1601.

[35] Youssef SG Nashed et al. "libCudaOptimize: an Open Source Library of GPU-based Metaheuristics". In: *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. ACM, 2012, pp. 117–124.

[36] Richard A. Newcombe et al. "KinectFusion: Real-time dense surface mapping and tracking". In: *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*. IEEE, 2011, pp. 127–136.

[37] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (2008), pp. 40–53.

[38] J.D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899.

[39] Sung Cheol Park, Min Kyu Park, and Moon Gi Kang. "Super-resolution image reconstruction: a technical overview". In: *IEEE signal processing magazine* 20.3 (2003), pp. 21–36.

[40] Michele Pirovano. "Kinfu - an open source implementation of Kinect Fusion + case study: implementing a 3D scanner with PCL". In: *Project Assignment, 3D structure from visual motion, University of Milan* (2011).

[41] Szymon Rusinkiewicz and Marc Levoy. "Efficient variants of the ICP algorithm". In: *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*. IEEE, 2001, pp. 145–152.

[42] Radu Bogdan Rusu and Steve Cousins. "3d is here: Point Cloud Library (PCL)". In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1–4.

[43] Renato F. Salas-Moreno et al. "SLAM++: Simultaneous Localisation and Mapping at the Level of Objects". In: *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE, 2013, pp. 1352–1359.

[44] Jason Sanders and Edward Kandrot. *CUDA By Example: An Introduction to General-Purpose GPU Programming*. Upper Saddle River, NJ: Addison-Wesley, 2011. 290 pp.

[45] Christian Schüler. *Branchless Matrix to Quaternion Conversion*. Aug. 7, 2012. URL: `http://www.thetenthplanet.de/archives/1994` (visited on 06/20/2016).

[46] Ken Shoemake. "Uniform Random Rotation". In: *Graphics gems III*. Ed. by David Kirk. OCLC: 838123981. Boston: AP Professional, 1992, pp. 124–132.

[47] John E. Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in science & engineering* 12.1-3 (2010), pp. 66–73.

[48] Hari Sundar et al. "Skeleton based shape matching and retrieval". In: *Shape Modeling International, 2003*. IEEE, 2003, pp. 130–139.

[49] Rick Szeliski. *Image Alignment and Stitching: A Tutorial*. Oct. 2004, p. 89.

[50] Johan WH Tangelder and Remco C. Veltkamp. "A survey of content based 3D shape retrieval methods". In: *Multimedia tools and applications* 39.3 (2008), pp. 441–471.

[51] The MathWorks, Inc. *Delaunay triangulation in 2-D and 3-D - MATLAB*. Version 8.6.0.267246 (R2015b). URL: `http://www.mathworks.com/help/matlab/ref/delaunaytriangulation-class.html`.

[52] The MathWorks, Inc. *Register an Aerial Photograph to a Digital Orthophoto*. Version 8.6.0.267246 (R2015b). URL: `http://www.mathworks.com/help/images/register-an-aerial-photograph-to-a-digital-orthophoto.html`.

[53] MacDonald , Tristam. *Spatial Hashing*. Sept. 30, 2009. URL: `http://www.gamedev.net/page/resources/_/technical/game-programming/spatial-hashing-r2697` (visited on 06/19/2016).

[54] Greg Turk. "Interactive Collision Detection for Molecular Graphics". The University of North Carolina at Chapel Hill, 1990.

[55] Oncel Tuzel et al. "Learning to Rank 3D Features". In: *Computer Vision–ECCV 2014*. Springer, 2014, pp. 520–535.

[56] T. Y. Zhang and Ching Y. Suen. "A Fast Parallel Algorithm for Thinning Digital Patterns". In: *Communications of the ACM* 27.3 (1984), pp. 236–239.