2017 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI 2017)
November 16 - 18, 2017, Daegu, Korea

Real-Time Human Collision Detection for Industrial Robot Cells

Erind Ujkani[1], Petter S. Eppeland[1], Atle Aalerud[1] and Geir Hovland[1]

***Abstract*— A collision detection system triggering on human motion was developed using the Robot Operating System (ROS) and the Point Cloud Library (PCL). ROS was used as the core of the programs and for the communication with an industrial robot. Combining the depths fields from the 3D cameras was accomplished by the use of PCL. The library was also the underlying tool for segmenting the human from the registrated point clouds. Benchmarking of several collision algorithms was done in order to compare the solution. The registration process gave satisfactory results when testing the repetitiveness and the accuracy of the implementation. The segmentation algorithm was able to segment a person represented by 4-6000 points in real-time successfully.**

I. INTRODUCTION

The industry today consists of many different types of security measures regarding collision detection. Several of these applications are made up of physically isolated workspaces where human interaction is at a minimum. Work has therefore been done to remove security fences from the robot workspace, in order to include humans to the system in what is known as a collaborative working environment.

One big step in removing the traditional physical security system, has been the introduction of depth field sensors. By describing the physical world with a set of points the robots have obtained depth vision. During the recent years, the use of depth sensors in robotic applications has increased. Newer sensors offer real-time depth field acquisition, which has opened the door for various novel applications, including collision detection between humans and robots.

An essential contribution to collision detection by the use of 3D data has been made by J. Schauer et al. in [1], where an offline collision detection has been used for a train driving through a tunnel. Other studies include that of S. Chitta [2], where a collision system based on libraries for motion planning, manipulation, kinematics, control and navigation of robot manipulators was implemented.

Use of multiple 3D cameras has resulted in more advanced methods such as sensor fusion being studied. In this process, multiple sensors (e.g. barometer and GPS) are combined to estimate a parameter (e.g. altitude), which can be further improved by the use of Kalman filtering. In the field of depth sensing, J. Zhu et al. [3] researched a method for using data from both time-of-flight (ToF) and stereo cameras to enhance depth estimates. They stated that the method developed was able to combine the complementary characteristics of stereo and ToF and result in an overall depth error reduction of 50 %.

Usually, depth fields contain considerable amounts of information and thus are expensive to process. Several alternative ways of allocating data in memory have been applied to deal with this obstacle. Octree is a popular spacial data structure that reconstructs the 3D scene by the use of a hierarchical tree structure, [4]. The structure is able to compress the depth field while the desired accuracy is specified by the smallest cube size of the tree. Another way to cope with the problem has been the implementation of Graphical Processing Unit (GPU) which realizes processing of depth fields in a parallel manner. Combining Octree with GPU has been done by K. Kaldestad et al. in [5] where the potential of this combination is demonstrated to perform collision detection. Also, A. Knoll in [6] proposes an efficient iterative algorithm for ray traversal of an Octree on GPU hardware. One improvement in the work presented in this paper compared to [5] is the fact that it can be used in cluttered environments where static objects such as walls, fences etc. are removed before the actual collision detection. In addition, multiple 3D sensors and registration are used in this paper compared to only one sensor in [5].

II. PROBLEM DESCRIPTION AND APPROACH

In a robot cell consisting of a robot and static objects, collision detection was performed using 3D cameras and Octree as the spatial representation with the objective of achieving real-time human collision detection using multiple 3D cameras. By the term *real-time* it was specified that the system should run at at least 10 FPS (10 updates per second). Additionally, it was specified that the presence of a human should pause the motion of the robot. To accomplish this goal, the task was broken down into several smaller objectives. The sub problems are listed as follows:

Registration: Implementation of a satisfactory registration algorithm to combine the depth fields from the 3D cameras. The solution has to be robust, repeatable and reusable in another environment.

Segmentation: A method for removal of static objects should be implemented so that the object of interest (the human) can be extracted. The process also involves the filtration of the actuated robot.

Collision detection: Carry out algorithms to perform collision test between the robot and the human. Several approaches were evaluated in order to validate the results.

III. METHODS

In this section the system is described and the developed algorithms for the objectives are presented. Some noteworthy

[1] Department of Engineering, Mechatronics Group, University of Agder, Norway. The research presented in this paper has received funding from the Norwegian Research Council, SFI Offshore Mechatronics, project number 237896

978-1-5090-6064-1/17/$31.00 ©2017 IEEE

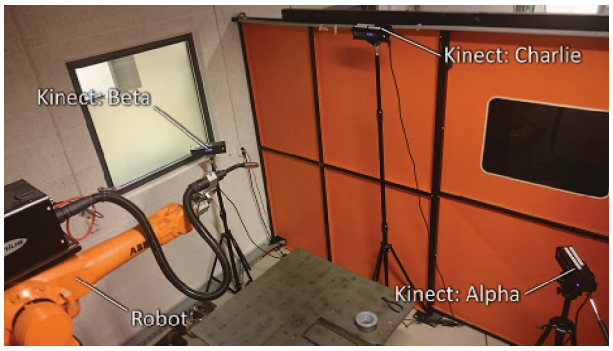


Fig. 1: Robot cell with ABB IRB1600 robot and three Kinect sensors.

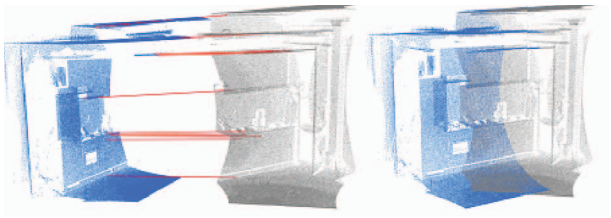


Fig. 2: Example of registration of point clouds. Before and after the alignment where the red lines show the correspondences [8].

terms are introduced, but it is assumed that the reader has some knowledge in the fields of computer vision and collision detection.

A. System Setup

The considered system consists of an ABB IRB 1600-6kg/1.45m robot with 6 degrees of freedom. The robot has a collision detection system at contact, which is triggered if abnormal torque levels are detected on any of the 6 robot axes. Inside the robotic cell, three Kinect V2 3D cameras are placed as shown in Fig. 1. Additionally, the high bandwidth requirements of the 3D cameras requires a PCI Express Card with individual and dedicated USB3.0 channels such as the PEXUSB3S44V. The hardware used consists of an Intel Core i7-6700k CPU and the NVIDIA GeForce GTX 1080 graphics card, which together with the CUDA programming platform allow for parallel computing.

B. Registration

As listed, the first obstacle is to combine the 3D cameras in a way that produces one single output. The method named registration is the process of aligning two point clouds, like two pieces of a puzzle (Fig. 2). The key idea is to find a set of correspondences between the datasets and then compute a linear transformation matrix that minimizes the distance between the corresponding points. This means that there is an area in the scene that has to take part in both point clouds. The best results are obtained with very similar clouds.

The registration algorithm used in this paper consists of a rough registration method utilizing point cloud feature information followed by a fine registration to complete the alignment. The pipeline is depicted in Fig. 3. As it can be seen, the algorithm starts with two point clouds, a source

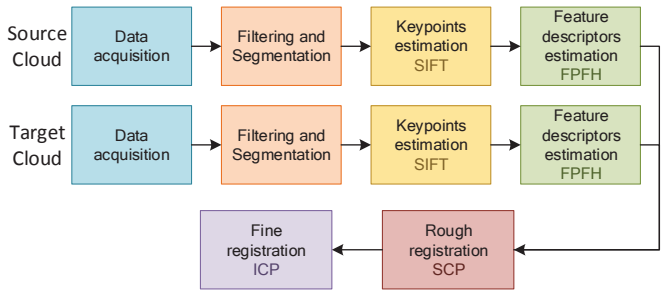


Fig. 3: The pipeline of the registration process.

and a target cloud, where the source cloud is to be aligned with the target cloud. Initially, to reduce the amount of depth field information, the point clouds are filtered using a downsampling filter and segmented by extracting common areas between the point clouds. Then, keypoints for both point clouds are acquired using the scale invariant feature transform (SIFT) keypoint estimation algorithm by D.G. Lowe in [9]. With these keypoints, the point clouds are represented in a concise but still descriptive manner.

The presented steps so far have been to save computational power for later. The next step in the pipeline is to create descriptive vectors describing every keypoint. These vectors are known as feature descriptors. The algorithm used in the work presented uses fast point cloud feature histogram (FPFH) descriptor which is known to be fast, robust and highly repeatable. The descriptor was presented by R.B. Rusu et al. in [7].

Using the feature information for the keypoints, the rough registration can be performed which is the next step in the pipeline. In short, the algorithm used in this paper consists of finding corresponding point pairs between the sets of keypoints using the feature descriptors, reject bad correspondences and then use a modified version of random sample consensus (RANSAC) to estimate the transformation between the point clouds. The RANSAC method was presented by A. Buch et al. in [10], and in this paper the algorithm is named sample consensus prerejective (SCP). Estimating a transformation using RANSAC between two point clouds starts by choosing n point pairs and then calculating the transformation between the geometry formed by the points. After this, a check is performed to see if the transformation is good or not, often done by calculating all the point pairs distances. Doing this for a lot of sets of point pairs is costly, therefore SCP introduces a simple check of the ratio between the edge lengths of the virtual polygons formed by the n sampled points. If the ratios are significantly invariant the point pairs are rejected. In [10] the results show that this algorithm is up to 15 times faster than the standard RANSAC routine.

This feature based registration is well suited for finding a quick initial alignment, but converges very slowly. That is why a second algorithm is often used to complete the registration. In this paper, the registration is finalized using the brute force method iterative closest point (ICP) which

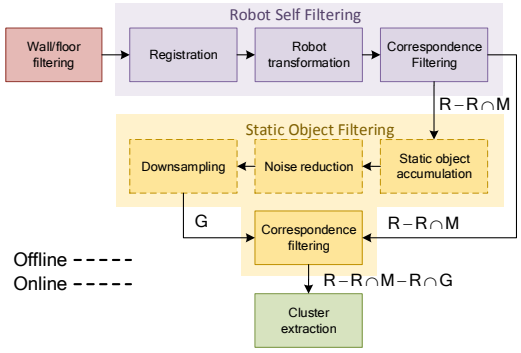


Fig. 4: The pipeline of the segmentation process.

was presented by Y. Chen and G. Medioni in [11]. The ICP algorithm works by minimizing the following cost function from [12]:

$$E(\mathbf{R},\mathbf{t}) = \sum_{i=1}^{|M|}\sum_{j=1}^{|D|}\omega_{i,j}||\mathbf{m}_i - (\mathbf{R}\mathbf{d}_j + \mathbf{t})||^2 \qquad (1)$$

where $\mathbf{R}$ and $\boldsymbol{t}$ are the rotation and translation for the affine transformation between the two point clouds. $\mathbf{m}_i$ and d_j are the points of two point clouds with the sizes M and D, respectively. $\omega_{i,j}$ is assigned to 1 if $\mathbf{m}_i$ and $\mathbf{d}_j$ describes the same point in space, otherwise $\omega_{i,j}$ is 0.

C. Segmentation

This part explains the second sub-objective which is the extraction of the human from the registrated point clouds, named *R*. Fig. 4 illustrates the pipeline, where each color marks one step in the algorithm.

The pipeline starts (red) by removing the walls and the floor. This is simply achieved by cutting off the edges of *R* using the known dimensions of the room.

Then the next step (blue) is to eliminate the robot in a process called robot self filtering. The initial step here is to use the same registration technique as previously explained to align a point cloud model of the robot (*M*) on top of the points representing the robot in *R*. After this is done, *M* continuously updates by reading the joint values of the real robot and transforming using forward kinematics. Finally, for every new point cloud of *R*, the correspondences between the transformed *M* and *R* (*R*∩*M*) are eliminated by looking for overlapping points using a k-d tree radius search.

Without the robot and the walls/floor, the next step (yellow) in the pipeline is to remove the remaining static objects in the room. The algorithm starts off by accumulating *R* over time (1 sec) and storing it in *G*. Then *G* is run through a noise reduction filter and finally down-sampled to reduce its size. The idea of accumulation is to compensate for sensor noise in the 3D cameras. *G* holds the information of the static objects and is used to remove corresponding points in *R* (*R*∩*G*) by using a k-d tree radius search.

After the first three steps, *R* contains only points that are not static and not the robot. If a person enters the room, he/she will be captured in *R*, but also unfiltered noise will be present. To cope with this issue, a final step called cluster extraction (green) is introduced. An algorithm called Euclidean clustering is used which works by selecting a point and checking its distance between other points. If the distance is within a threshold, both are considered to be part of the same cluster. Similar to a flood fill algorithm, a point in the point cloud is labeled as ”chosen” for the cluster. Then, like a virus, it spreads to all other points that are close enough. When no new point can be added, a new cluster is initialized, and the process starts over but without the already labeled points. R. Rusu in [8] presents this process for one cluster with his algorithm (algorithm 1) using a k-d tree radius search for the localization of neighborhood points.

Algorithm 1 Pseudo code for Euclidean clustering.

```
create a k-d tree representation for the input point cloud dataset P
set up an empty list of clusters C, and a queue Q of the points that need to be checked
for every point p_i ∈ P do
    add p_i to the current queue Q
    for point p_i ∈ Q do
        search for the set P_k^i of point neighbors of p_i in a sphere with radius r < d_th
        for every neighbor p_i^k ∈ P_i^k do
            if point has not been processed then
                add it to Q
            end if
            add Q to the list of clusters C, and reset Q to an empty list
        end for
    end for
end for
```

D. Collision Detection

After acquiring an object of interest in the segmentation process, a collision detection between the robot model and the human cluster can be performed. Collision occurs when a predefined Euclidean distance (eq. 2) between the robot and the human is reached.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \qquad (2)$$

The simplest collision detection algorithm is implemented as an exhaustive search, where every point of one subset is compared to every other point in the second subset (algorithm 2). This is computationally demanding, where the time complexity for this search is $O(N \cdot M)$, where N is the number of point clouds in the first cloud, and M is the number of points in the second point cloud. In order to reduce this time complexity, other methods to perform collision detection are investigated. The first method improves the exhaustive search itself by the use of parallel programming. Here, time complexity for the search task is reduced to $O(N)$. The second method is based on using the Octree data structure and PCL to perform a radius search collision

Fig. 5: Picture showing the robot model (green), human cluster (grey) and collision points on the human cluster (red).

detection (algorithm 3). This search algorithm differs from the exhaustive search because of its broad and narrow phase mechanisms. The broad phase mechanism utilizes the Octree data structure to perform an initial rejection of points that are far apart from each other based on a search sphere with radius r. The points that overlap with the search sphere are investigated further in the narrow phase by the use of exhaustive search.

Algorithm 2 Pseudo code for exhaustive search.

```
// Distance threshold: δ
// Collision threshold: ς
Set counter to 0
for each point i of cloud1 do
    for each point k of cloud2 do
        if Euclidean distance< δ then
            increase counter
        end if
    end for
end for
if counter> ς then
    set collision to true
end if
```

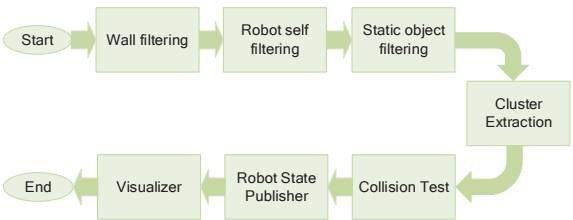


Fig. 6: Figure of the procedures included when profiling the code.

Fig. 6 shows the parts that are included when testing the system performance. Furthermore, a custom benchmark consisting of a sampled human cluster with unidirectional sinusoidal motion as shown in Fig. 7 was created. In this

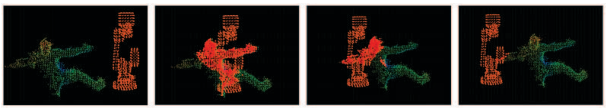


Fig. 7: Pictures of the custom benchmark test.

test the human cluster size is varied in order to study its significance on the overall system performance. This variation of point cloud size is applicable for all the algorithms, but the movement of the human cluster is only relevant for testing of the Octree radius search.

Algorithm 3 Pseudo code for the Octree radius search.

```
// Collision threshold: ς
// Radius search threshold: ξ
Create an Octree instance of the largest cloud with the chosen resolution(e.g. cloud2)
Define radius
Set counter to 0
Set collision to false
for each point n of cloud1 do
    if radius search at point n with Octree object> ξ then
        increase counter
    end if
end for
if counter> ς then
    set collision to true
end if
```

IV. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, the benchmarks of the three objectives are presented. First results specific to the implemented registration algorithm are presented, then results regarding the segmentation process and lastly the run-times of the CPU and GPU collision detection algorithms are compared with each other.

A. Registration

For the registration implementation interesting validation data include the algorithm time for the SCP and ICP and fitness scores. The fitness score is computed with eq. (3) which is the sum of square correspondences Euclidean distances divided by the number of correspondences. Specifically, it is a number for how well the point clouds aligns. Such a value is difficult to use for comparison between registrations of different sets of point clouds, because it is dependent on the ratio of the overlapping areas. But the value gives a good indication of how the alignment evolves and its repeatability. To benchmark the algorithms, 10 tests were performed. Tab. I summarizes the values with the mean values and standard deviations.

$$f_{score} = \frac{1}{n}\sum_{i=1}^{n}\left(\sqrt{(\boldsymbol{q}_1(i) \quad \boldsymbol{p}_1(i))^2 \quad (\boldsymbol{q}_2(i) \quad \boldsymbol{p}_2(i))^2 \quad (\boldsymbol{q}_3(i) \quad \boldsymbol{p}_3(i))^2}\right)^2 \quad (3)$$

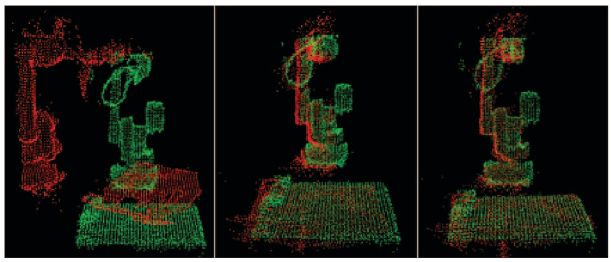
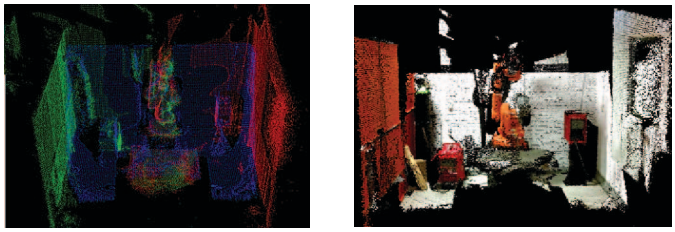


Fig. 8: The registration process using the segmented parts of the point clouds. The three images show respectively the untransformed point clouds, the transformation after SCP and the complete alignment after ICP.



(a) (b)

Fig. 9: (a) shows the point cloud alignments of the three 3D cameras. Each color represents a depth field from a camera. (b) is the same combined point cloud image, but colored using the RGB values from the cameras.

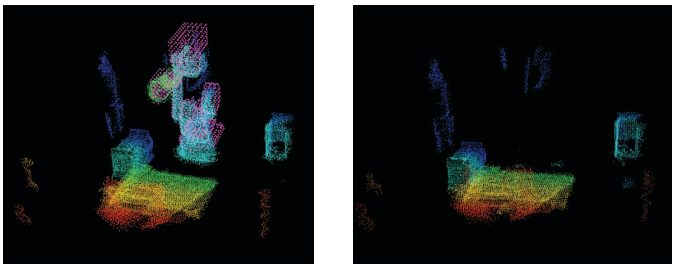
	Alpha		**Beta**	
	μ	σ	μ	σ
SCP time $[ms]$	2621	237.1	2287	171.5
ICP time $[ms]$	561	57.6	487	47.3
SCP f_{score}	$5.00e-3$	$5.93e-4$	$3.70e-3$	$1.10e-3$
ICP f_{score}	$4.10e-3$	$3.56e-4$	$2.40e-3$	$2.20e-4$

TABLE I: The registration results showing the standard deviations σ and the mean values μ.

As it can be seen from the computational times, the registration is not well suited for real-time purposes. The combined time is about 2.5 - 3.4s for each registration and ideally the time should be less than 20 ms considering other processes. Still, for a surveillance system, the cameras are most likely in fixed positions, thus there is less need for a real-time system. Hence, it was decided that it is sufficient to run the registration process as an initial step in the program, under the assumption that the cameras are stationary. If some objects in the robot cell supposed to be stationary are moved, for example a wall, the registration must be redone. The fitness scores are reasonably consistent with relatively small standard deviations (about 10 %), which show that the algorithm can produce consistent results. Figs. 8 and 9 show some results from the registration process.

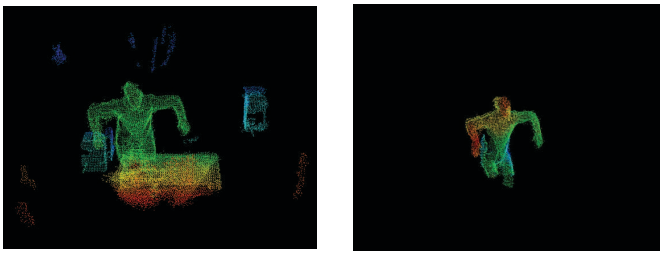
B. Segmentation

The segmentation algorithm presented was able to segment a person represented by 4000 - 6000 points while preserving



(a) Before (b) After

Fig. 10: Robot self filtering: the purple points mark the robot model which is aligned on top of the robot in *R*. The correspondences are eliminated in (b).



(a) Before (b) After

Fig. 11: Before and after the removal of static objects and the Euclidean clustering. Only the points representing the human are present in the output point cloud.

the requirement for real-time. The processing loop completes on average in approximately 40 - 50 ms. Human appearance is clearly evident in the extracted point cloud. The arms and legs are well preserved. However, in some occasions noise can be present, which means that additional points append on top of the persons surface. Stricter filters could be added to solve this, but at the expense of reducing the point cloud density and thus decrease the quality of the clustered object as previously discussed. Figs. 10 and 11 show results from the segmentation process.

C. Collision Detection

The processes included for performance testing are as specified in Fig. 6. For the desired application, the robot cloud had a resolution of 1140 points and the object of interest had a size of 1543 points. The exhaustive search on the CPU performed the test with a mean FPS rating of 10.1, whilst the same algorithm managed to get a 9.5 FPS rating by the use of GPU programming. The Octree radius search on the CPU performed better than the previous algorithm with 10.6 FPS.

Comparison between the CPU and GPU has been performed by increasing the size of human cluster size. Here the algorithms have been studied in isolation, thus defining the calculation time of the algorithm. Because of the parallelism in GPU programming the word *calculation pairs* becomes more suitable than *iterations*. A calculation pair is defined as a calculation between one point and another. As seen in the subfigure in Fig. 12, the CPU outperforms the GPU until

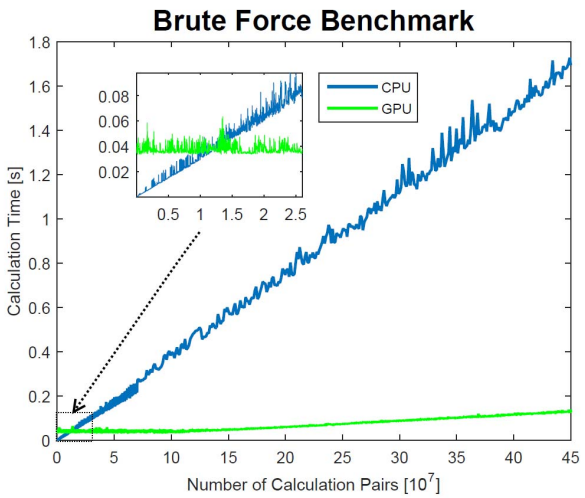


Fig. 12: Performance of the CPU and GPU with respect to the number of calculation pairs (algorithm 2).

a breaking point of around 10^7 calculation pairs. After this breaking point the CPU time increases linearly relative to the number of calculations, while the GPU calculation time remains less affected. This happens until a CPU/GPU relative saturation point at around $2 \cdot 10^8$ calculation pairs. Here the GPU algorithm manages to perform the calculations 13 times faster than the CPU.

Calc. Pairs$[10^6]$	1	2	4	10	20
CPU time $[s]$	0.0032	0.0062	0.0190	0.0370	0.0650
GPU time $[s]$	0.0340	0.0390	0.0360	0.0360	0.0430
CPU/GPU $\approx$	0.1	0.2	0.5	1.0	1.5

Calc. Pairs$[10^6]$	40	100	200	400
CPU time $[s]$	0.147	0.396	0.768	1.482
GPU time $[s]$	0.036	0.039	0.059	0.115
CPU/GPU $\approx$	4.1	10.2	13.0	12.9

TABLE II: Table showing the results of relative performance between the GPU and CPU.

V. DISCUSSION AND CONCLUSIONS

In this paper real-time human motion collision detection in an industrial robot cell has been described and results presented. By assuming that the 3D cameras were stationary and placed in fixed locations, the time-consuming registration process was performed offline and only once in the experimental results presented in this paper. The rest of the process (robot self-filtering, segmentation and collision detection) was performed online and approximately 10 frames per second (FPS) were achieved.

An approach that could ease the process of registration could be to introduce motion cap markers which are small spheres coated with a retroreflective material. These markers are highly reflective in a IR image and therefore convenient to use as keypoints in the point cloud. There are unfortunately two issues with this method. First of all, is that it requires well calibrated cameras to make sure that the IR image is perfectly matched on top of the depth field. Secondly, is that the IR image received from the topic created by the ROS bridge, iai_kinect2, saturates the areas with high intensity values, i.e. the markers. A workaround to this issue could be to use a holefilling algorithm to fill the circles and then remap the image in the depth field.

Even though fast results are achieved with ROS during the tests in this paper, no guarantees are provided about the timing of the operations. This means that ROS is not suitable for operations that have strict timing requirements such as high-frequency PID-controllers and motion control, unless there is an acceptable fallback solution. (i.e. stop machines if loop timer reaches limit). For actual safety-critical systems, it is often required to have a real-time system with high predictability (low jitter). Development of a successor of ROS called ROS 2 is ongoing for solving the real-time issue, [13]. This improvement might become important to increase the safety measures in motion planning systems such as collision avoidance.

For further work, implementation of real-time registration could be interesting. One approach would be to optimize the PCL registration functions, but an adaption to GPU is an alternative method. The need for real-time registration may not be an issue for stationary cameras in constrained environments, but rather for other applications such as offshore, where vibrations in the systems can be severe which can cause displacement of the cameras.

REFERENCES

[1] J. Schauer, and A. Nüchter, Efficient point cloud collision detection and analysis in a tunnel environment using kinematic laser scanning and k-d tree search, In Intl. Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives, vol. 40, pp. 289–295, aug 2014.

[2] S. Chitta, I. Sucan, and S. Cousins, MoveIt! [ROS Topics], IEEE Robotics & Automation Magazine, 19(1):18–19, Mar 2012.

[3] J. Zhu, L. Wang, R. Yang, and J. Davis, Fusion of time-of-flight depth and stereo for high accuracy depth maps, In 26th IEEE Conf. on Comp. Vision and Pattern Recognition, CVPR, pp.18. IEEE, Jun 2008.

[4] D.J.R. Meagher, Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer, Rensselaer Polytechnic Institute, 1980.

[5] K.B. Kaldestad, G. Hovland, and D.A. Anisi. 3D Sensor-Based Obstacle Detection Comparing Octrees and Point clouds Using CUDA. Modeling, Identification and Control, 33(4):123130, 2012.

[6] A. Knoll, Ray traversal of Octree point clouds on the GPU. RT08 - IEEE/EG Symposium on Interactive Ray Tracing 2008, Proceedings, 12(3):182, 2008.

[7] R.B. Rusu, N. Blodow, and M. Beetz, Fast Point Feature Histograms (FPFH) for 3D Registration, 2009.

[8] R.B. Rusu, Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments. Kunstliche Intelligenz, 24:345348, 2010.

[9] D.G. Lowe, Distinctive image features from scale-invariant keypoints, Intl. J. Computer Vision, 2004.

[10] A.G. Buch, D. Kraft, J.K. Kamarainen, H.G. Petersen, and N. Kruger, Pose estimation using local structure-specific shape and appearance context, Proceedings - IEEE International Conference on Robotics and Automation, 2013.

[11] Y. Chen, and G. Medioni, Object Modeling by Registration of Multiple Range Images, Sacramento, 1991. IEEE.

[12] A. Nchter, K. Lingemann, and J. Hertzberg, Extracting drivable surfaces in outdoor 6D slam, VDI Berichte, (1956):189, 2006.

[13] Open Source Robotics Foundation, Why ROS 2.0, url: http://design.ros2.org/articles/why_ros2.html, visited July 10, 2017.