Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Hallgeir Løkken

Background Subtraction on Real Time Point Clouds

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
June 2020

NTNU
Norwegian University of Science and Technology

Abstract

Many useful operations on point clouds are time consuming and can be susceptible to noise. This thesis proposes a simple system architecture to host various background subtractors while operating on real time data. This removal of unnecessary information allows further processing to focus on the task at hand. Empirical data in this thesis shows that background subtraction can reduce the size of a point cloud by more than 80% in indoor scenes with one human being the foreground object of interest. The system is implemented in C++ with a C-compliant interface, allowing it to be used as a dynamically linked library in a vast amount of environments. Also presented is a dataset, allowing further work to be done without specialized camera hardware.

Sammendrag

Mange nyttige operasjoner på punktskyer krever mye regnekraft, og kan være sårbar mot støy. Denne avhandlingen setter frem en enkel systemarkitektur som kan huse forskjellige strategier for bakgrunnssubtraksjon som kan kjøre sammen med andre prosesser i sanntid. Empirisk data presentert i denne avhandlingen viser at punktskyer fra innendørsscener med et enkelt menneske som interesseobjekt kan reduseres med mer enn 80%. Systemet er implementert i C++ med et C-kompatibelt grensesnitt som gjør det mulig å nyttegjøre seg det i form av et dynamisk lenket programvarebibliotek i svært mange situasjoner og utviklingsmiljø. Også presentert her er et datasett som muligjør videre forskning, også uten å skaffe spesialiserte kamera.

Preface

This thesis is done as the conclusion of my studies at the Norwegian University of Science and Technology (NTNU) over my final semester. I am sincerely grateful and would like to give my thanks to my supervisor Theoharis Theoharis whose help has been invaluable. I would also like to thank all my collegues at Ablemagic for their continous encouragement and for providing me not only with equipment needed and a place to work, but also a good environment in which to thrive both as a programmer and as a person.

Contents

1 Introduction and Background

This thesis describes a system for background subtraction on real time point clouds using a novel system architecture that is flexible and expandable. The system is designed to serve as a platform for advanced applications using 3D data for complex user interactions. In addition to an implementation of the proposed system, a dataset for testing, verifying, and doing further work is presented and made available.
This chapter introduces the reader to the thesis by first showing an overview of the structure, then discussing the nomenclature used and explaining some foundational background concepts.

1.1 Structure of Thesis

This thesis is structured as follows:

Chapter 1 contains an introduction to the structure of the thesis, a note on the motivation behind the project, an introduction to the nomenclature used, as well an overview of the background concepts used in this field.

Chapter 2 follows the introduction by looking at earlier related work that has been done, and considers the similarities and differences to this thesis.

Chapter 3 presents in depth the proposed system architecture for real time background subtraction on point clouds. Several options to the tweakable parameters are presented to show the flexible design.

Chapter 4 evaluates and show the results of the proposed pipeline. A publicly available, labeled dataset for use in testing, verification, training, and doing further work in this domain, without purchasing RGB-D cameras, or setting up an environment to capture usable data is described here, and can be found in Appendix B.

Chapter 5 concludes the thesis with a discussion on the proposed process for background subtraction before rounding off by looking at some possible avenues for further work into this, and related problems.

Appendices include a report done earlier on the process of capturing point clouds from RGB-D cameras and comparisons of various sensors, the dataset discussed in Chapter 4, and the source code for the project used to record, label, and inspect the included dataset, as well as to test the proposed pipeline.

1.2 Motivation

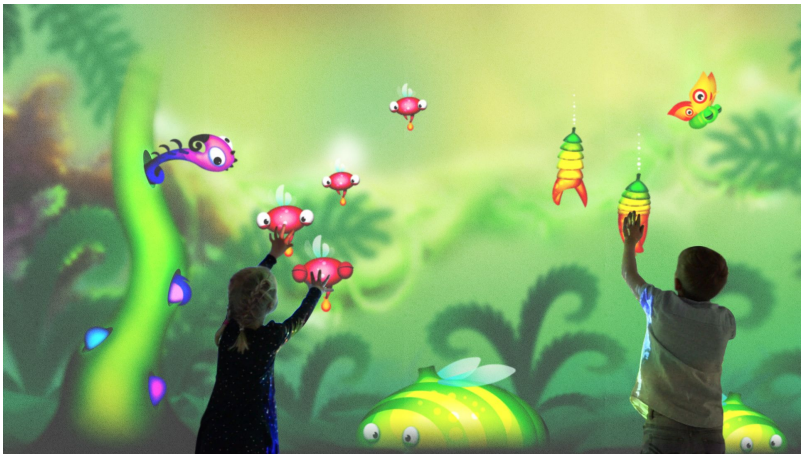


Figure 1.1: Two children interacting with the creatures in Munti Forest

This thesis is in large part motivated by work done by Ablemagic[1] on creating innovative motion technology for the purpose of art and entertainment. Early spring of 2020, during the International Film Festival in Trondheim, "Kosmorama", Ablemagic delivered an immersive experience for children called "Munti Forest"[2] that used three RGB-D cameras to create a large real time point cloud that was used to make the entirety of a large wall interactive. The system was a great success, but some technical issues was encountered during the set up and installation that no existing technology could solve. The main issue was a lack of semantic information on the generated point clouds. More precisely, the system could not differenciate between the walls of the scene, and the users interacting with it. This lead to the need for a massive amount of manual adjustments and calibration. A robust system for doing background subtraction would solve this issue completely, avoiding false interactions causing the background to trigger events that should only be triggered by the users.

1.3 Nomenclature and conventions

Here follows a list of some commonly used concepts used in this thesis with a short description and clarification of their meaning in the context of this thesis.

Background is defined by the Merriam-Webster thesaurus as "The physical conditions or features that form the setting against which something is viewed" [Merriam-Webster, 2020]. In this thesis this loose definition is used as a guide. In addition, a slightly more specific definition is written: "Everything in a scene that generally does not move". Ambiguous situations appear often. Examples may be a person bringing an object into the room and leaving it there. Another may be a car that drives and comes to a stop in the scene, and not move again in a long time. What is foreground and what is background is often disambiguated by defining objects of interest. In the examples mentioned here, the objects of interest could be humans, and the object being carried would be defined as background as soon as it was left, and the car would be defined as background from the start.

Point Cloud is a set of points containing at least spatial information, but often also other information such as color and segmentation tags. In this project each point has spatial XYZ components, as well as RGB color components. They are computed from the overlapped color and depth images from RGB-D cameras. When run through the proposed system, a 32-bit segmentation tag is also attached, holding information on which camera the point originated from, and if the point was classified as foreground or background.

RGB-D cameras are specialized cameras that capture both color images and depth information. See Appendix Appendix A for more in depth information on such sensors. All data used in this thesis was captured using an ORBBEC Astra Mini camera and an ORBBEC Astra camera.

Background Subtraction is synonymous with Foreground Extraction. It is a special case of segmentation where each datapoint is labeled either foreground or background. The goal is to be able to remove background information, in order to make various expensive processing tasks much easier because of the reduced amount of irrelevant data it needs to process. It is commonly the first stage in many security surveillance applications.

Segmentation is the process of dividing the input data into segments (pixel groups for 2D images, or clusters of points in 3D point clouds) and give each segment a semantic tag. Typically these tags denote if the segment is (part of) certain objects. As an example, a tag can be *wall*, *human*, or *sky*, or as in this thesis, the tags are *foreground* and *background*.

CNN is an acronym for a convolutional neural network. They are ubiquitous in the field of segmentation of 2D images because they take data in a known grid-like topology. They are

[1] `https://ablemagic.no/`
[2] `https://ablemagic.no/#/digitalplaygrounds/9`

also used in processing 3D point cloud data, as discussed in section 2. Importantly, CNNs are able to learn and recognize spatial, chromatic, and potentially temporal relationships in the input data. [Goodfellow et al., 2016].

FPS is an acronym for *Frames Per Second*. In most real time software, processing happens in a large core loop of reading input data, doing calculations, and producing some output data. One iteration of this loop is often referred to as a *frame*, as in animation. Different applications targets a different number of FPS.

Real Time Performance is a quality of systems that processes data fast enough to keep up with an input stream such as that of a camera or microphone, and/or output stream such as a monitor with a certain refresh rate, which determines the target FPS. For this thesis, real time performance is considered to be $30FPS$, or $\frac{1}{30}$th of a second per iteration of the core loop, as informed by Appendix A. The performance of a system is also dependent on the computer hardware it is being run on. Details on the hardware used to measure the performance of the proposed system in this thesis is shown in section 4.

Several sections of this thesis makes use of various operations on vectors. Where scalar operations are used with multidimensional vectors as input, the operation is intended to be done separately on each component. For instance $\lfloor \vec{v} \rfloor = [\lfloor v_x \rfloor \lfloor v_y \rfloor \lfloor v_z \rfloor]^T$ and likewise for scalar-vector arithmetic operations, as in $\frac{\vec{v}}{2} = \left[\frac{v_x}{2} \frac{v_y}{2} \frac{v_z}{2}\right]^T$ and functions, as in $sin(\vec{v}) = [sin(v_x)sin(v_y)sin(v_z)]^T$.

1.4 Background

This section explains some foundational concepts used by the proposed method.

1.4.1 Bounding Boxes

In this thesis, bounding boxes are axis-aligned boxes (AABBs) that have the minimum dimensions to contain a given set of points. Bounding boxes are important for many optimizations because they are easy to compute, and the common problem of determining if a 3D point p is within the given box *aabb* or not, given the points of its minimal and maximum vertices (see Figure 1.2) requires at most 6 comparisons and 5 logical AND operations as shown in Equation 1.1.

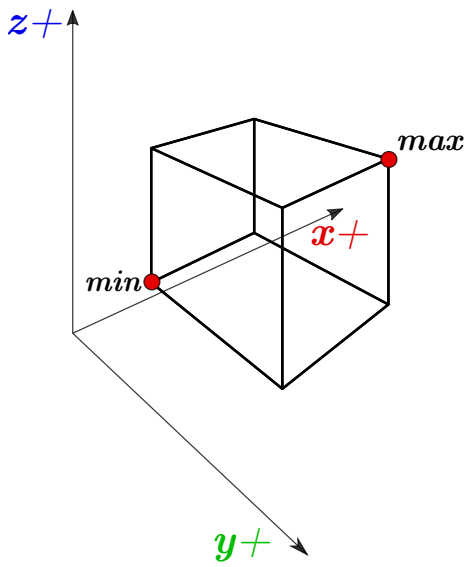


Figure 1.2: An Axis-Aligned Bounding Box

$$\begin{aligned} inside(p, aabb) =& p.x > aabb.min.x \wedge \\ & p.x < aabb.max.x \wedge \\ & p.y > aabb.min.y \wedge \\ & p.y < aabb.max.y \wedge \\ & p.z > aabb.min.z \wedge \\ & p.z < aabb.max.z \end{aligned} \tag{1.1}$$

1.4.2 RGB-D to Point Cloud

RGB-D cameras produce pairs of 2D images, one with color data, one with depth data. This is discussed more in depth in Appendix A. This section takes a look at how these images are transformed into point clouds. The depth images have a certain resolution $W \times H$ and a certain field of view in the horizontal and vertical axes FOV_H and FOV_V. The depth image given by the sensor contains the depth D_{XY} at each pixel XY.
The problem of unprojecting the pixels of the depth image into 3D space can be visualized as in Figure 1.3. It can be seen that there are a right angle triangle where the two short legs are the depth value D_{XY} and the horizontal pixel space distance from the center. tan is defined as $tan(\theta) = \frac{oppositeleg}{adjacentleg}$ and θ can be found by considering how far along the the depth image X is, and FOV_H. Thus the X component of the 3D spatial position can be found by reordering the definition of tan to $oppositeleg = tan(\theta) \times adjacentleg$. Equation 1.2 shows how the complete 3D spatial position is computed using the above knowledge. First, the X and Y pixel positions are transformed from $[0..width)$ and $[0..height)$ with origo in the top left corner of the image to $[-0.5..0.5]$ with origo in the center. They are then multiplied with their respective FOV to find how far along the image the point is. After computing the tangent, they are then multiplied with D_{XY} (the adjacent leg in Figure 1.3).
All points computed this way then form a cloud with origo at the position of the RGB-D cameras. In the case of multiple sensors the point clouds from each are combined by transforming them with both a translation and a rotation according to the real world positioning of the sensors.

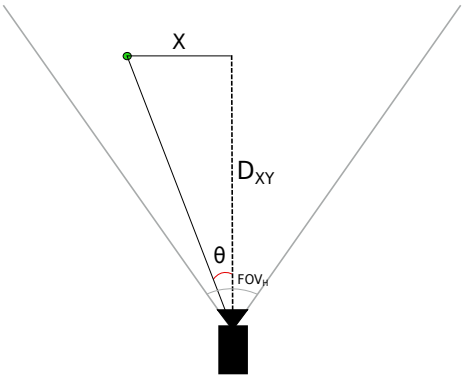


Figure 1.3: Direct top view of the camera frustum.

$$\begin{bmatrix} tan(((X/W) - 0.5) \times FOV_H) \times D_{XY} \\ tan((0.5 - (Y/H) \times FOV_V) \times D_{XY} \\ D_{XY} \end{bmatrix} \tag{1.2}$$

1.4.3 Voxels

A voxel, or *volume element* is to a volume what a pixel is to an image. Like a pixel, a voxel often contain some information such as RGB color. Together, voxels form a voxel grid of a certain dimension. Figure 1.4 shows an example voxel grid with dimensions $6x6x6$ with a single voxel colored in. Grids such as these, with empty cells are called *sparse* [Theoharis, 2008].

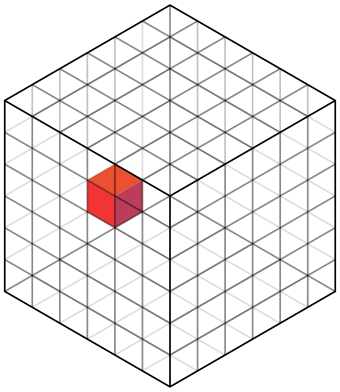


Figure 1.4: An example voxel grid.

Voxel grids in this thesis have a direct relation to world space, and require some form of indexing. In this thesis, all voxel grids are centered on the world space origo, and are indexed by *column*, *row*, and *layer* corresponding to the X, Y, and Z axes, starting at 0 and growing in the positive direction of the axes. This simplifies the implementation in code. To find the world space position v of a voxel indexed by c, r, l given the world space size V of each voxel, one first find the minimum corner B_{min} of the voxel grid bounding box. Since the voxel grid is centered on the world space origo, this is $B_{min} = \left[\frac{B_x}{-2}\frac{B_y}{-2}\frac{B_z}{-2}\right]^T$ where B is the voxel grid's AABB. Then v can be found by $v = V\left[c, r, l\right]^T + \frac{V}{2}$ where $\left[c, r, l\right]^T$ is a vector containing the indices c, r, and l.

The opposite operation is also frequently used, where the grid indices of the enclosing voxel of a given world space point P is needed. They can be found by $\lfloor\frac{P-B_{min}}{V}\rfloor$ where the *column*, *row*, and *depth* indices are the x, y, and z components of the resulting vector.

2 Related Work

This chapter will take a look at related work and discuss what is relevant for the subject of this thesis and what is different.

2.1 2D Background Subtraction

The problem of background subtraction has been well explored for 2D images. Many of the principles that have been discovered can be applied to point clouds as well, so it is worthwhile to take a look at some of the approaches that have been used.

2.1.1 Classical algorithms

First, two different approaches to background subtraction are discussed. These both work by creating a background model, and comparing each new image to the model. One creates the model based on the variance of each pixel, while the other creates a model based on local regions around pixels.

Gaussian Mixture Models , or Mixture of Gaussians (MOG) can be used to build a statistical model of the background by modeling each pixel with a mixture of K gaussian distributions, where K is some small integer. [KaewTraKulPong and Bowden, 2002] shows how this can be used to create an adaptive background model by taking a subset of B weighted distributions. A pixel is considered *foreground* if it is outside 2.5 standard deviations of any of these B distributions. The "value" of a pixel can either be it's grayscale intensity, or each color channel can have their own distributions. In order to make the model adapt to slight changes over time, the weights and the variance of each distribution change over time. Also, if there are none of the K distributions matching the pixel, the least probable is swapped with a new one, starting with a low weight, the value of the pixel in question as the mean, and a large variance.

Local SVD Binary Pattern (LSBP) is an approach described by [Guo et al., 2016] to combat the perceived high amount of false positives in the above method, and to gain better robustness to changes in illumination. MOG considers each pixel separately, but LSBP considers the local environment around each pixel to gain spatial information. The method is based on [Heikkilä and Pietikäinen, 2006] which worked on the local neighbourhoods of pixels, but was still vulnerable to local noise. The LSBP approach improved on this by introducing Singular Value Decomposition (SVD) with normalized coefficients around the local region. This makes the method more illumination invariant.

2.1.2 Deep Learning

In later years, Deep Learning has exploded in popularity, and it has also been applied to this field, chiefly in the form of CNNs. There are two main approaches to design a CNN to solve the background subtraction problem. First, one can take the same approach as the classical algorithms mentioned above, where the network is trained on the background it will subtract. This can be referred to as "Single scene training". The other, more ambitious, approach is to train the network on a vast array of scenes, and try to solve the problem in general. This can be referred to as "Multi scene training". Some of the advantages gained in multi scene training is tolerance of camera movement and rapidly changing backgrounds as the network may be able to generalize the concept of "background". However, it is shown by [Minematsu et al., 2018] that multi scene training is non-trivial, and the networks tested struggle, though they do outperform the classical algorithms in single scene training.

2.2 Real Time Data

In the case of the aforementioned classical algorithms, they all make use of temporal relationships between the input frames to solve the background subtraction problem. However, according to [Pfeuffer et al., 2019], most work done on segmentation with neural networks are done on static images and data. They show that the temporal information available in video streams and real time data can improve performance.

2.3 Point Cloud Segmentation

Though most work with CNNs for background subtraction are being done on 2D images, there have been work done with 3D data as well, mostly on the more general task of segmentation. [Qi et al., 2017] implemented a neural network called *PointNet* that is capable of processing raw point clouds directly for both shape classification and per-point segmentation. Their experiments showed that the network was able to process about one million points per second. In contrast, the proposed system can process about nine million points per second, due in part to the simpler task of binary classification, but also to the novel architecture.

PointSeg by [Wang et al., 2018] does real time point cloud segmentation by projecting the point cloud to a 2D image using spherical projection. They can then make use of the existing CNN *SqueezeNet* for segmentation. Unlike the proposed system in this thesis, their point clouds are captured by LiDAR sensors on moving automobiles, and as such their system needed to handle extremely dynamic backgrounds. Using spherical projection to transform the point cloud into 2D images allowed them to use existing datasets of spherical images, regardless if they were captured from LiDAR sensors or not. They claim their system managed to operate at $90FPS$ which is well within real time performance, as it is considered in this thesis. The spherical projection works well as long as the point cloud is captured from a single viewpoint. However, in this thesis, the clouds are amalgamations of several clouds captured from several viewpoints. In this case, points will overlap each other in a 2D projection, shadowing each other.

The proposed system is inspired by [Tchapmi et al., 2017], who proposes a system for point cloud segmentation by transforming the point cloud data into a regular voxel grid to use as input to a fully connected CNN. This allows them to take advantage of spatial relationships between points, and be able to handle any size of point cloud that fits within the spatial bounds of the voxel grid. To improve the pointwise segmentation and avoid the issue of coarse segmentation where all points within the same voxel are given the same tag, they introduce the concept of trilinear interpolation after the output layer. The interpolation takes the class probabilities from the eight closest voxels and weights them based on spatial distance to the point. This technique is also used in the proposed system.
The data presented in their article show that they surpass the state of the art, and beat *PointNet* in segmentation benchmarks. Each voxel in the grid corresponds to a 5 cm cube in real world space, and their voxel grid is one hundred voxels in all three dimensions, giving a bounding volume of a five meter cube, and one million voxels in total. They used two pooling layers in the early stages of the network to downsample to a forth of the input size combined (that is in all dimensions, so the total number of voxels went down to 15625). They used this network in combination with a Conditional Random Field as proposed by [Kamnitsas et al., 2017] to further improve the accuracy. Unlike this thesis, *Tchapmi et. al.* did not focus on real time performance, and did not publish data on the runtime of their system, nor hardware used. The system was tested and evaluated in part on the *Stanford Large-Scale Indoor Spaces 3D Dataset* [Armeni et al., 2016] which contains five areas where each area contain several large scale annotated indoor scenes captured in different rooms. Each room contains on average about one million points.

3 Method

This chapter describe the presented method of achieving background subtraction of point clouds in real time by first presenting an overview, then in closer detail, and lastly discussing some alternatives available to implementations.

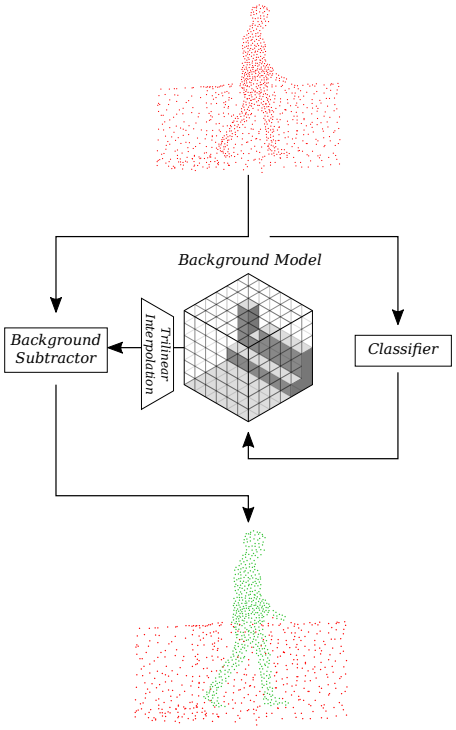


Figure 3.1: Overview of the two-threaded system.

3.1 Overview

The process is simple but two-pronged, as shown in Figure 3.1. In the main thread of execution each point P of the point cloud get a background probability b by trilinearly interpolating the values in a voxel grid M that serves as a model of the background, much like the proposed system of [Tchapmi et al., 2017]. This probability is then tresholded to get a binary classification.
The voxel grid M is created and continuously updated by a secondary thread of operation using any classifier suitable, two of which are outlined below. Because this thread is run asynchronously it is not required to conform to the same time restriction as the main thread for the system to maintain real time performance. Still, faster is better, as a faster update of the background model will adapt quicker to changes to the background. The background thread is free to choose whether to always use the latest frame available, or to implement any other suitable scheme.
The voxel grid M was chosen for this sake of the experiments of this thesis to bound real world volume of a 5 meter cube, with a resolution of $100 \times 100 \times 100$ voxels. This implies that each voxel is a 5 centimeter cube. In general, each voxel is a cube of size V, and the grid consist of $M_{width} \times M_{height} \times M_{depth}$ voxels. The outer hull of the voxel grid then forms an AABB around the area of interest in the shape of an axis-aligned parallelepiped with dimensions $VM_{width} \times VM_{height} \times VM_{depth}$.

This voxel grid format was inspired by [Tchapmi et al., 2017] and chosen to accommodate different classifiers, with the understanding that it is the classifiers that is the most advanced

part of the system, and also the most rapidly changing. From a software engineering point of view, it makes for a cleaner separation of concerns, and allows rapid iteration on the classifier without touching any other part of the system. With this architecture, the classifier is a very independent module and can be improved, replaced, or even removed with little to no effect on the rest of the environment. Additionally, the regular grid structure makes adapting existing 2D grid methods simpler, as they just need to be extended to 3D.

3.2 Getting the per-point background probability

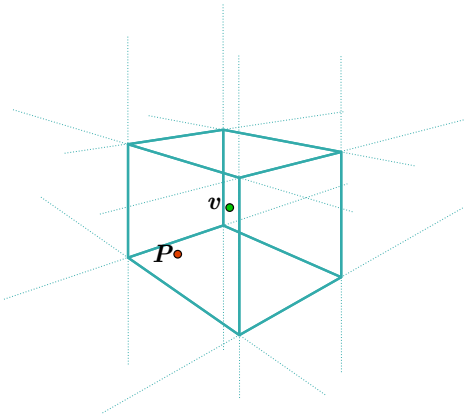


Figure 3.2: The point P and the voxel that contains it, with center v.

To get the background probability for a point P given the voxel grid background M, P is first checked against the AABB of the voxel grid using Equation 1.1. If P is outside the bounding box it is skipped (labeled neither background nor foreground). Points that exist on the inside of the bounding box get assigned a background probability b by trilinearly interpolating between the eight closest voxels, as shown in Equation 3.1. The weights $w_n, n \in \{1..8\}$ are computed for the eight surrounding voxels by normalizing the distances from P to v_n, where v_n is the center position of the n-th neighbour voxel. In the case of $P = v_n$, $w_n = 1$ while $w_m = 0, m \neq n$. Equation 3.2 show how to compute w_n in the general case. b_n is the background probability of the n-th neighbour. v_n is the center position of the n-th neighbour.

$$b = \sum_{n=1}^{8} w_n b_n \tag{3.1}$$

$$w_n = \prod_{s \in \{x,y,z\}} 1 - \frac{|v_n^s - P^s|}{V} \tag{3.2}$$

The eight surrounding voxels, or neighbours $N = \{v_1, \ldots, v_8\}$, are found by first finding the center v of the single voxel that contains P. This will be one of the eight. The full set can be found by computing a set of eight offset vectors where the n-th member is an offset from v to v_n. To find the offset vectors a vector $\vec{s} = sign(\vec{o}), \vec{o} = v - P$ is computed, where $sign$ is defined in Equation 3.3. This vector is elementwise multiplied with each of the eight possible bitstrings of length three $B = \{000, 001, 010, 011, 100, 101, 110, 111\}$.

$$sign(\alpha) = \begin{cases} -1 & \alpha < 0 \\ 0 & \alpha = 0 \\ 1 & \alpha > 0 \end{cases} \tag{3.3}$$

In summary, $N = v + \vec{s} \times B$.

3.3 Classifiers

There are several options for classifiers that can be used to create the background model. Two simple methods are implemented and discussed here, and more are discussed in subsection 5.2. Advantages and issues with each is described in section 5.
Available input data to the classifiers are the raw point cloud data containing the spatial position as well as the RGB color of each point. The classifiers must all output a voxel grid as discussed above, with a background probability score for each voxel.
Both presented classifiers voxelize the input data to a grid with same dimensions as the output grid. This grid has four channels; *occupancy*, R, G, B, where *occupancy* is the number of points contained by the voxel, and the RGB channels are the average color of each point.

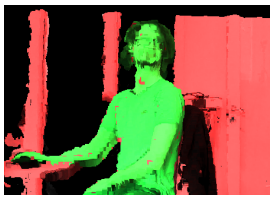


Figure 3.3: A segmented point cloud

3.3.1 Static Scene Calibration

A simple, yet efficient approach is to record the maximum occupancy values of each voxel during an initial calibration phase when the scene is known to contain only background points. This allows the background thread to only do work during the initial calibration and optionally during recalibration.
Higher maximum occupancy values during calibration leads to higher background probability. The implementation of this classifier uses a binary function, resulting in background probability $b = \begin{cases} 0 & occupancy = 0 \\ 1 & occupancy > 0 \end{cases}$. However, a linear function is also viable, such as $b = \alpha + \beta occupancy$, where α and β both are adjustable parameters that can be set by experimentation. The output should be clamped to a maximum of 1, as it is a probability.
This classifier is highly susceptible to changes to the background or changing viewpoints, as it has no way of coping. However, an advantage of this classifier, is that foreground elements that are stationary for some time does not fade or leave "shadows", as is the case with the next proposed classifier.

3.3.2 Averaging Model

Another simple approach is to average the occupancy values over a sliding window of time, giving voxels with high average occupancy a higher background probability. Equation 3.4 show how the average occupancy over d frames can be estimated, where $C = occupancy$. n is limited to $d - 1$. This estimation is a good alternative to recording d historical occupancy values for every voxel, which would scale poorly with both d and the size of the voxel grid. The average occupancy is then tresholded to give a binary probability of 0 or 1.
As mentioned above, this approach leads to an issue where stationary foreground elements starts fading into the background. The time it takes for a foreground object to fade can be increased by increasing the value of n, however by doing so, if a foreground element still does fade, the time it takes to erase the "shadow" of the element increases as well. Also, lower values of n allows quicker adaptations to changes in the background.
Another issue that arises with this approach, in 3D more so than in 2D, due to the *Hollow Hull Problem* discussed below. If the system is only using one camera, or has areas with insufficient

coverage, background voxels in the shadow of foreground elements will have point count equal to zero, making the average occupancy tend to zero as well, making them appear as foreground voxels when in view of the camera again.

$$\bar{C}_{n+1} = \frac{n\bar{C}_n + C}{n+1} \tag{3.4}$$

3.4 Classifiers for Point Clouds originating from RGB-D cameras

In the context of this thesis, all point clouds originate as 2D images from RGB-D cameras, making the earlier work done on background subtraction mentioned in section 2 viable, as it can be performed *before* the point clouds are computed. The goal of this thesis is to do background subtraction on any point cloud, regardless of origin. Published datasets of point clouds often does not contain these images in any case, even if RGB-D cameras were used to capture them, so there will not be much discussion of this approach here, however as an experiment, both of the feasibility of this approach, and to test how flexible the proposed architecture is, the implementation provided in Appendix C has the option of utilizing this strategy through the open source computer vision library OpenCV[3].
OpenCV provides implementations of several algorithms to perform background subtraction on 2D images. The implementation in Appendix C makes use of this to gather data on how effective this approach is. section 4 shows a visual inspection of a selection of the algorithms implemented by OpenCV.

3.4.1 The hollow hull problem

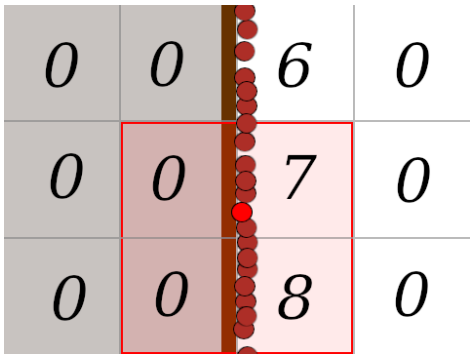


Figure 3.4: Top-down view of the voxels annotated with their occupancy values. The red region shows the voxels selected for trilinear interpolation of the bright point

An issue that arises when computing the voxel grid from point clouds captured from RGB-D cameras is that all points can only lay on the surfaces of objects. This is an issue when computing the input voxel grids, as voxels that are fully within an object will have an occupancy value of zero, even though they are part of the background. Since both the algorithms proposed above give low background probability scores for voxels with low occupancy values, the internal voxels will pull the calculated background probability scores down during trilinear interpolation on points on the wall. In Figure 3.4 this issue is illustrated by showing points along a wall and the voxels annotated with their occupancy values. As shown in the red region, half of the voxels used in the trilinear interpolation for calculating the background probability for the bright point has occupancy values equal to zero, which in both of the above algorithms would yield a background probability of zero as well. A solution is discussed in subsection 5.2.

[3] `https://opencv.org/`

4 Evaluation

This chapter discusses the dataset in Appendix B in detail, by elaborating on the motivation behind it's creation, and the how and why it was created and attached to this thesis. This chapter ends by taking a look at the implementation of the presented system that was used to produce the dataset, as well as to test and verify its quality.

4.1 Dataset

Here follows a discussion of the goals, method of creation, and detailed information on dataset in Appendix B.

4.1.1 Goals of the Dataset

Most published datasets of point clouds are static scenes with few or no foreground elements such as [Armeni et al., 2016].
For this dataset, it is intended to focus on the foreground, moving objects. Possible use cases are static analysis of raw results with RGB-D cameras, fine tuning or testing/verification of deep learning systems that work on point clouds, or simply to have data with which to work in environments where acquiring cameras, or setting them up in a useful scene, is difficult.

It is *not* the goal of this data set to be large enough to fully train a deep learning system, nor to show the performance of the specific RGB-D cameras used in every possible lighting and range situation. However, there is a note on expanding the dataset in subsection 5.2.

4.1.2 Creation of the Dataset

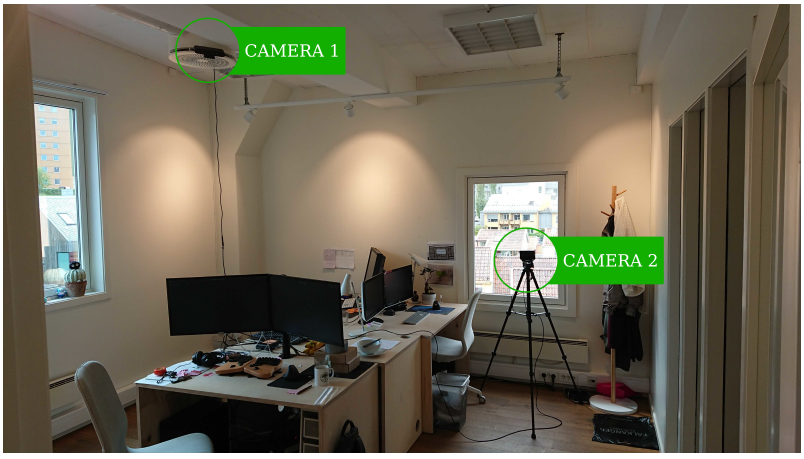


Figure 4.1: Camera setup for capturing the dataset

The dataset in Appendix B was created from two ORBBEC RGB-D cameras placed at different viewpoints as shown in in Figure 4.1. Camera 1 is an ORBBEC Astra RGB-D camera, and was placed in the ceiling, at 260cm above ground level. Camera 2 is an ORBBEC Astra Mini RGB-D camera, placed further away from the main space of interest, at 130cm above ground level. The cameras were placed so that their view volumes had as much overlap as possible, but coming from different angles. The viewing angles between the cameras were quite acute. This was due to physical constraints in the scene. The dataset contains 87 point cloud frames with an average of 323386 points per frame.
The footage shows a person (the author) moving through the space. It is divided into five phases:

1. A person moving through the room, opening a door and leaving.

2. Empty room for ten seconds.
3. A person entering the room.
4. The person standing still for ten seconds.
5. The person moving in the room, and out of view.

These phases highlight expected problem areas; foreground objects fading when still, and the background changing. Leaving the room empty and still for a while provides the opportunity for observing how quickly a system can readapt to an altered background.
The cameras did not move at all during the data capture.

The dataset is labeled by the implementation of the proposed system included in Appendix C, using the static scene calibration classifier in combination with the RGB-D specific OpenCV classifier, running an LSBP-based algorithm. It was post processed by hand by the author to remove most of the noise. This is a tedious and time consuming process. Synthetic data as mentioned in subsection 5.2 would allow one to create much larger volumes of correct data much quicker.

During recording, the system was not able to stay within the specified limits for real time performance as laid out in this thesis, due to compressing and writing large amounts of data to the file system. This explain why the dataset contains 87 frames from roughly 30 seconds of capture.

4.1.3 Dataset file format

There are currently a lack of standardized file formats for animated point clouds. There are attempts to create such file formats, such as the *XYZ* file format for describing molecular structures [OpenBabel, 2017]. However, the presented dataset has more than three hundred thousand points per frame and close to one hundred frames, making the need for some kind of compression necessary. For this reason a custom file format was chosen, in an attempt to create files both easy to parse and small of size.
Also included are the raw frames captured from the RGB-D cameras, so that the point clouds can be recomputed for more thorough verification. There are many file formats for 2D images and video available, however the nature of the coupled color and depth images, and the complexity of many popular video file formats led to the decision of implementing a similar custom format for these recordings as well.
In the following description of the formats, the point cloud recordings are referred to as *cloud recordings* and the raw frame recordings are referred to as *video recordings*. The implementation in Appendix C contains code to read and write files on both of these formats.

The file formats use binary representation of numbers, however header fields are lines of ASCII text. As a compromise between simplicity and compression, the DEFLATE algorithm was chosen to compress data. The DEFLATE algorithm is specified in RFC1951 [Deutsch, 1996]. In both file formats there are chunks of compressed data. These are an eight byte unsigned integer specifying the number of bytes in the chunk (these eight preceding bytes not included), immediately followed by the compressed data. The implementation in Appendix C use the public domain implementation *Miniz* to compress and decompress [Geldreich, 2019].
Also common for both formats is that the last eight bytes of the files contain the number of frames as an unsigned integer.

Listing 1: C code to get the number of frames in a recording

```
uint64_t
get_recording_framecount(FILE *recording_file)
{
    uint64_t result = 0;

    fseek(recording_file, -sizeof(uint64_t), SEEK_END);
    fread(&result, sizeof(uint64_t), 1, recording_file);

    return(result);
}
```

Listing 1 shows a function implemented in C to get the number of frames in a recording of any of the two formats. The cloud recordings follow the outline shown in Listing 2. Each frame is preceded by a header line in ASCII text containing the word *frame*, the frame number, and number of points in that frame. The line ends with the ASCII new-line character, value 10. This is true for all line endings in both formats.
After the header follows three compressed chunks of data, the first one being the spatial positions of each point, the second being the color data for each point, and the last being the tags for each point. Following the compressed chunk of tag data is another new-line character.

Listing 2: Dataset file format

```
frame 1 <N>\n
<Compressed frame of spatial cloud data>
<Compressed frame of color cloud data>
<Compressed frame of tag cloud data>\n
frame 2 <N>\n
...
frame M <N>\n
...
<64 bit unsigned integer with the value M>
```

Assuming the file offset is at the beginning of a frame header line, the function shown in Listing 3 will read one frame. Error checking and handling are omitted for brevity.
The spatial position data decompresses to a tightly packed array of structures representing the points. This structure contains three *IEEE 754* floating point numbers for the X, Y, and Z components of the points. [IEEE, 2019]
The color data decompressed to a structure of three bytes per point, holding the R, G, and B values of the colors. Lastly, the segmentation tags are 32 bit long little-endian bitstrings, where each bit is a binary flag. Counting from the least significant bit the flags are as follows:

1. The point originates from the first camera.
2. The point originates from the second camera.
3. The point originates from the third camera.
4. The point originates from the fourth camera.
5. The point is a foreground point.
6. The point is a background point.

The remaining bits are unused.

Listing 3: C code to read one frame from a cloud recording

```
struct CompressedBuffer
{
    uint64_t compressed_size;
    void *compressed_data;
};

struct CompressedCloudFrame
{
    size_t frame_index;
    size_t pointcount;
    struct CompressedBuffer spatial_cloud;
    struct CompressedBuffer color_cloud;
    struct CompressedBuffer tag_cloud;
};

struct CompressedBuffer
read_compressed_chunk(FILE *fd)
{
    struct CompressedBuffer result = { 0 };

    fread(&result.compressed_size, sizeof(uint64_t), 1, fd);
    result.compressed_data = malloc(result.compressed_size);
    fread(result.compressed_data, 1, result.compressed_size, fd);

    return(result);
}

struct CompressedCloudFrame
read_cloud_frame(FILE *recording_file)
{
    struct CompressedCloudFrame result = { 0 };

    size_t start_offset = ftell(recording_file);
    fscanf(recording_file, "frame %zu %zu\n",
           &result.frame_index,
           &result.pointcount);

    fseek(recording_file, start_offset, SEEK_SET);
    for(uint8_t c=0; c != '\n'; fread(&c, 1, 1, recording_file)) {}

    result.spatial_cloud = read_compressed_chunk(recording_file);
    result.color_cloud = read_compressed_chunk(recording_file);
    result.tag_cloud = read_compressed_chunk(recording_file);

    fseek(recording_file, 1, SEEK_CUR); // Skip the trailing new-line

    return(result);
}
```

Video recording files starts with a larger header detailing the number of sensors, and the specifications of each. These specifications are vendor, name, and ID on the first line, followed by a line with the resolution and field of view of the color images. Finally the last line of the sensor specifications show the resolution, field of view, and range of the depth values (in millimeters). An example of a file captured using two ORBBEC Astra Mini cameras for two frames is shown in Listing 4 as an example. The compressed chunks are abbreviated. The color data compresses in exactly the same way as the color points in the cloud recordings mentioned above. Depth data compressed to one *IEEE 754* floating point number per pixel.

Listing 4: Frame recording file format

```
2 sensors\n
Orbbec Astra 17122831108\n
640 480 1.022602\n
640 480 1.022602 0.000000 10000.000000\n
Orbbec Astra 17122730029\n
640 480 1.022602\n
640 480 1.022602 0.000000 10000.000000\n
frame 1\n
color\n
<compressed frame of color pixels>\n
depth\n
<compressed frame of depth pixels>\n
frame 2\n
color\n
<compressed frame of color pixels>\n
depth\n
<compressed frame of depth pixels>\n
<64 bit unsigned integer with value 4>
```

4.2 Results

Now the implementation of the proposed system will be introduced in more detail, and data on the performance, both in terms of execution speed and precision will be presented.

4.2.1 Implementation

The system was implemented in C++ as a shared library with a C-compatible interface, allowing it to be utilized in many environments, such as Python[4] and C#[5]. This library has been named *MagicMotion*. In addition to this library, a platform for prototyping and testing was developed, called *launchpad*. This is also developed in C++, and is a continuation of the project started with Appendix A. *launchpad* implements tools used to record the aforementioned dataset, do manual data sanitation, and do live qualitative analysis of the system performance. This separation causes the structure of the *MagicMotion* library to be clearly defined, and usable as a library. Though further discussion is out of scope of this thesis, the *MagicMotion* library has already been used by applications written in C#, with no changes required.

A key feature of the implementation of *MagicMotion* is the modular nature of the classifiers used in the secondary thread for updating the background model. At compile-time, different classifiers can be chosen. These are divided into *3D-classifiers* and *2D-classifiers*. *3D-classifiers* use only the point cloud as input data, and updates the voxel grid background model only. The *2D-classifiers* work only because the point clouds originate from RGB-D cameras, and do their computations on the images from these sensors. These classifiers produce a per-sensor mask

[4] Using ctypes
[5] Using InteropServices

image with white pixels marking high foreground probability, and black pixels the opposite. The system is able to use either type of classifier alone, or both simultaneously by running two secondary threads. *MagicMotion* contains at the time of this thesis one 2D-classifier, and two 3D-classifiers, as well as a structure that simplifies the implementation of more at a later time. The 2D-classifier makes use of the open-source free computer vision library OpenCV as mentioned before. The two included 3D-classifiers are the simple static scene calibration, and a the averaging background mentioned in section 3.

4.2.2 Visual evaluation of OpenCV background subtractors

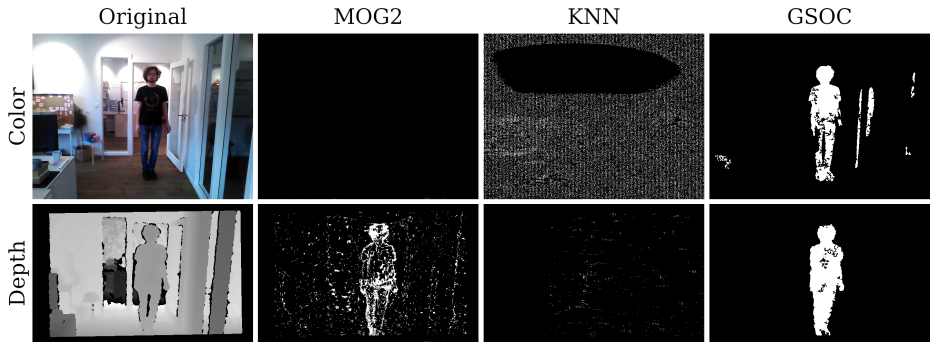


Figure 4.2: Comparison of different algorithms for background subtraction.

As mentioned previously, the open source computer vision library OpenCV implements several algorithms for doing background subtraction on 2D images. As an experiment, one of the implementations was chosen and used in the included code. In order to choose an algorithm implementation, several was run side-by-side for qualitative visual inspection. The chosen algorithms was *MOG2*[6], *KNN*[7], and *GSOC*[8]. The *GSOC* background subtractor makes use of LSBP features and was contributed to OpenCV as part of the 2017 Google Summer of Code [Samsonov, 2017].
A snapshot of the visual inspection is shown in Figure 4.2, and resulted in the *GSOC* background subtractor being chosen.

4.2.3 Classifier Performance

The dataset was created by capturing the output of the system running both a 2D-classifier, and the 3D static scene calibration classifier. Sixty seconds was spent on calibration before recording started. After recording, the dataset was manually inspected and corrected. For each captured frame, the number of True Positives TP (Points in the background labeled as such), False Positives FP (Points in the foreground labeled as background), True Negatives TN (Points in the foreground labeled as such), and False Negatives FN (Points in the background labeled as foreground) was counted by comparing the original captured frame to the manually corrected frame. Table 4.1 shows the mean, median, and standard deviation of the precision, recall, and balanced accuracy of all frames.
Precision is a metric on the probability that a point the system labeled as background, actually is part of the background, while Recall shows the fraction of true background points that was labeled as such. Balanced Accuracy ($\frac{TNR+TPR}{2}$, where $TNR = \frac{TN}{TN+FP}$ and $TPR = Recall$) was used instead of plain Accuracy ($\frac{TP+TN}{TP+TN+FP+FN}$) because of the imbalance between positive (background) points, and negative (foreground) points [Tharwat, 2018]. In the included dataset 95% of the points are part of the background, averaged over all frames. The frame

[6] `https://docs.opencv.org/3.4/d7/d7b/classcv_1_1BackgroundSubtractorMOG2.html#details`
[7] `https://docs.opencv.org/3.4/db/d88/classcv_1_1BackgroundSubtractorKNN.html#details`
[8] `https://docs.opencv.org/3.4/d4/dd5/classcv_1_1bgsegm_1_1BackgroundSubtractorGSOC.html#details`

with the minimum fraction of background points had 85%. This imbalance skews the Precision and Recall values, and evaluation should mainly consider the Balanced Accuracy.

Table 4.1: Empirical data from the dataset

	Precision	Recall	Balanced Accuracy
Mean	0.998	0.982	0.705
Median	1.000	0.978	0.500
Standard Deviation	0.003	0.010	0.241

4.2.4 Timing and Performance Profiling

A requirement of the proposed system is to be capable of running in real time. As noted before, the definition of "real time" can be loose, but in the context of this thesis it is defined as running faster than the cameras being used, which in this case is $30FPS$, or $\frac{1}{30}s = 33.333\ldots ms$ per frame.
To measure time spent at various points of interest a stop clock like functionality was implemented, and the system was run for at least 600 frames times in succession, while recording the time spent in regions of interest. In the following tables, the average, median, and standard deviation of these numbers will be discussed for a few interesting spots in the execution flow. The average and median is included to give a sense of the actual time use, while the standard deviation is included to give a sense of how stable the system is in terms of time spent. Both the CPU clock cycle counter and the time spent in seconds is included.
Naturally, all timings are very dependent on the hardware the system is run on. Table 4.2 details the specifications of the computer used, as well as the RGB-D camera used to capture the data. The depth buffer from the particular camera used is actually 640 columns by 480 rows, however, with the internal image registration algorithm in use (to properly align the depth pixels with the RGB color pixels, see Appendix A), the resolution changes somewhat.

Table 4.2: Hardware used in timing experiments

CPU	AMD® Ryzen 7 3700X 8-core processor, 16 hardware threads
CPU Clock Speed	3.8 GHz
Installed Memory	16 GiB
Camera	1x ORBBEC Astra Mini RGB-D Camera
Depth image resolution	640 columns, 400 rows
Point cloud size	256000 points (maximum)

Table 4.3 shows the timing for the entire main thread process from the sensor images arrive, through projecting the pixels to the 3D point clouds, bounds checking against the voxel grid AABB, doing trilinear interpolation of the background probability from the model, and setting the tags.

Table 4.3: Timing of the full main thread process

	CPU Cycles	Time in milliseconds
Average	55436117	14.07
Median	57325660	14.51
Standard Deviation	7481086	1.94

Not timed in the above table is the time spent ensuring proper synchronization. Table 4.4 shows the timing of the main thread process with the time spent waiting for synchronization locks, semaphores, and other unpredictable delays.

Table 4.5 contains the timing data of the secondary thread running the simple averaging classifier. The static scene calibration classifier was not timed, as it does not do any work

Table 4.4: Timing of the full main thread process, including synchronization

	CPU Cycles	Time in milliseconds
Average	61923774	15.55
Median	64467494	16.51
Standard Deviation	8357399	2.51

outside the calibration process, in which it can be assumed to perform much the same as the averaging classifier. This timing was started after synchronization locks was taken, and show that the classifier spends on average between $2ms$ and $3ms$ per iteration, and is quite stable. The maximum time spent in all of the samples taken was just above $5ms$.

Table 4.5: Timing of the simple averaging classifier

	CPU Cycles	Time in milliseconds
Average	9585040	2.44
Median	9412752	2.61
Standard Deviation	1343758	0.59

5 Discussion, Conclusion, and Further Work

This chapter concludes this thesis, by drawing some conclusions from the results presented above, and taking a look at some possible avenues for further work.

5.1 Conclusion

The proposed method is proving to be a efficient and flexible platform for experimenting with real time background subtraction on point clouds with good results even on the initial classifiers. A major feature of the proposed method is the modularity of the classifier component, which makes it easy to experiment with a wide array of different strategies. Coupled with the two-threaded architecture, this allows developers and researchers to keep experimenting on real time data without concerning themselves with premature optimizations as the work of the classifier is kept of the main thread.
The *MagicMotion* library and *launchpad* are now both valuable tools for work and experimentation in this area, and are already proving their worth in work done by Ablemagic.

5.2 Further Work

This section presents four avenues to expand the use of the presented system, or to improve its performance.

First, he Hollow Hull Problem mentioned earlier causes problems for the trilinear interpolation needed to smooth out the voxel based segmentation, and a solution is likely to improve the performance of the existing system significantly.
The problem can be more generally recognized as the problem of detecting closed hulls. A potential solution if to find a voxel v that i guaranteed to have a true occupancy score of zero, and do a flood-fill operation to find all other true zero occupancy voxels. This must be done once per sensor. Any voxels not found during this process that have an occupancy value of zero must be within a closed hull and should have a background probability score of 100%.

Secondly, more sophisticated classifiers would certainly increase the accuracy of the system. From the classical 2D background subtractors, the LSBP feature based approach should be researched closer in 3D space, to see if it can detect textures in 3D voxel grids as well as it can in 2D images. Today most modern research in segmentation and computer vision is based on deep learning. A CNN similar to the one in [Tchapmi et al., 2017] should be implemented and experimented with to see if it can be trained to do multi scene background subtraction. This network is reasonably simple and uses three residual modules as described in [for Image Recognition, 2015] with pooling layers in between and after, and showed in their paper to perform well on similar scale point clouds as used in this thesis.

Thirdly, there is a need for more well labeled data for training and evaluation. However capturing real world data and manually labeling it as done in this thesis is time consuming and tedious. I. B. Barbosa *et. al.* showed in [Barbosa et al., 2017] that synthetic data is very useful in quickly producing large amount of data for training, fully labeled, and anonymous. They showed in their paper that systems trained on synthetic data and fine tuned on a smaller amount of real data achieved state of the art performance. Similar methods of generating synthetic data are very relevant for background subtraction and other work done on point clouds as well, especially in the case of point clouds originating from RGB-D cameras such as in this thesis, as the color and depth 2D images correspond well to the color and depth buffers of traditional 3D renderers. The noise found in real life sensors can be simulated as well, if needed.

Fourth and lastly, though background segmentation has been the focus of this thesis, more fine-grained segmentation has been the focus of other related work. Many applications can benefit from adding tags such as *ground* and *wall*. As an example, the problem of aligning

several point clouds from different sensors was mentioned in Appendix A. Using assumptions about the direction of surface normal vectors of walls and floors could simplify this problem.

6 References

[Armeni et al., 2016] Armeni, I., Sener, O., Zamir, A. R., Jiang, H., Brilakis, I., Fischer, M., and Savarese, S. (2016). 3d semantic parsing of large-scale indoor spaces. In *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition*.

[Barbosa et al., 2017] Barbosa, I. B., Cristani, M., Caputo, B., Rognhaugen, A., and Theoaris, T. (2017). Looking beyond appearances: Synthetic training data for deep cnns in re-identification. *Computer Vision and Image Understanding*.

[Deutsch, 1996] Deutsch, P. (1996). Deflate compressed data format specification version 1.3. RFC 1951, RFC Editor.

[for Image Recognition, 2015] for Image Recognition, D. R. L. (2015). Kaiming he and xiangyu zhang and shaoqing ren and jian sun. *Microsoft Research*.

[Geldreich, 2019] Geldreich, R. (2019). Miniz. `https://github.com/richgel999/miniz`. Accessed: 2020-06-02.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Guo et al., 2016] Guo, L., Xu, D., and Qiang, Z. (2016). Background subtraction using local svd binary pattern.

[Heikkilä and Pietikäinen, 2006] Heikkilä, M. and Pietikäinen, M. (2006). A texture-based method for modeling thebackground and detecting moving objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[IEEE, 2019] IEEE (2019). Ieee standard for floating-point arithmetic. Standard, IEEE Computer Society.

[KaewTraKulPong and Bowden, 2002] KaewTraKulPong, P. and Bowden, R. (2002). An improved adaptive background mixture model for real-time tracking with shadow detection. *Video-Based Surveillance Systems. Springer, Boston, MA*.

[Kamnitsas et al., 2017] Kamnitsas, K., Ledig, C., Newcombe, V. F. J., Simpson, J. P., Kane, A. D., Menon, D. K., Rueckert, D., and Glocker, B. (2017). Efficient multi-scale 3d cnn with fully connected crf for accurate brain lesion segmentation. *Medical Image Analysis*.

[Merriam-Webster, 2020] Merriam-Webster (2020). `https://www.merriam-webster.com/thesaurus/background`. Accessed: 2020-05-18.

[Minematsu et al., 2018] Minematsu, T., Shimada, A., Uchiyama, H., and ichiro Taniguchi, R. (2018). Analytics of deep neural network-based background subtraction. *Journal of Imaging*.

[OpenBabel, 2017] OpenBabel (2017). Xyz (format). `http://openbabel.org/wiki/XYZ_%28format%29`. Accessed: 2020-06-02.

[Pfeuffer et al., 2019] Pfeuffer, A., Schulz, K., and Dietmayer, K. (2019). Semantic segmentation of video sequences with convolutional lstms.

[Qi et al., 2017] Qi, C. R., Su, H., Mo, K., and Guibas, L. J. (2017). Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR, abs/1612.00593*.

[Samsonov, 2017] Samsonov, V. (2017). Improvement of the background subtraction algorithm. `https://summerofcode.withgoogle.com/archive/2017/projects/6453014550282240/`. Accessed: 2020-05-25.

[Tchapmi et al., 2017] Tchapmi, L. P., Choy, C. B., Armeni, I., Gwak, J., and Savarese, S. (2017). Segcloud: Semantic segmentation of 3d point clouds. *International Conference on 3D Vision (3DV)*.

[Tharwat, 2018] Tharwat, A. (2018). Classification assessment methods. *Applied Computing and Informatics*.

[Theoharis, 2008] Theoharis, T. (2008). *Graphics & visualization : principles & algorithms*. A.K. Peters, Ltd.

[Wang et al., 2018] Wang, Y., Shi, T., Yun, P., Tai, L., and Lua, M. (2018). Pointseg: Real-time semantic segmentation based on 3d lidar point cloud. *arXiv*.

Appendices

A RGB-D cameras and their uses

Norges teknisk-naturvitenskapelige
Universitet

TDT4501

Fordypningsprosjekt

RGB-D Cameras and their uses

Author:
Hallgeir Løkken

Supervisor:
Theoaris Theoaris

December 11, 2019

RGB-D cameras and their uses

TDT4501 Fordypningsprosjekt

Abstract

RGB-D depth sensing cameras have become common in research in computer vision ever since the introduction of the Microsoft Kinect in 2010 made them cheap and accessible. This project explores todays market of RGB-D sensors, their uses, and the techonolgy that powers such uses. To make it more practical, I have begun implementing software that is able to use RGB-D sensors and some of the techniques discussed in this report to act as a playground and a jumping off point for developing applications that uses RGB-D sensors for real time computer vision. The source code is freely available on GitHub[1].

[1] `https://github.com/Istarnion/RGB-D-Lauchpad`

Contents

1 Introduction

In my work at Ablemagic I have been using various RGB-D cameras in attempts at creating interesting installations and toys suitable for public display. In this project I have sought to gain a better overview of the available RGB-D sensors on the market, their uses, and the technology that powers these uses. I have focused my attention to those cameras that are the most accessible, and the use case of generating point clouds. This project is done as a preliminary study for my master thesis, and has given me the insight needed to choose a focus area and to start my work without delay.

This chapter will introduce more precisely the nature of these cameras and some of the various models on the market, as well as some important concepts common to the applications in this field.

1.1 Vocabulary

As with any field within science, there are many words not commonly found in daily speach outside the subject. The table below provides a reference for the reader should they encounter an acronym or a term they are unfamilliar with.

RGB-D	Acronym for Red Blue Green and Depth, refering to the data channels in the images the sensors discussed here provide.
FOV	Acronym for Field Of View. The FOV of a camera is how open the viewing angle is. In this report I will use FOV for the horizontal angle only, and measure it in degrees.
FPS	Acronym for Frames Per second. This measures how many image captures a camera can make in the duration of one second, or simmilary how many images a computer graphics program can produce in that same amount of time.
Segmentation	The process of labeling each point in some data set, for instance each pixel in an image, or each point in a point cloud.
Point cloud	A potentially large set of data points that atleast contain 3D position, but often also contain information such as color and segmentation labels.
CNN	Acronym for Convolutional Neural Network, a category of neural networks made to work on multidimensional data such as images.

1.2 Examples of applications

To motivate and to inspire, I finish the introduction by listing two select applications.

1.2.1 Videogames

The most well known RGB-D sensor is the Microsoft Kinect, which I will discuss more in technical detail later. It was originally developed for the XBox 360 videogame console and enabled many games to use the players full body as a game controller. Microsoft sold 35 million Kinect sensors across all the models before they discontinued the consumer focused line in 2017 [Reisinger, 2017].

Figure 1.1: Screenshot from the game "Fruit Ninja Kinect". © Halfbrick Studios

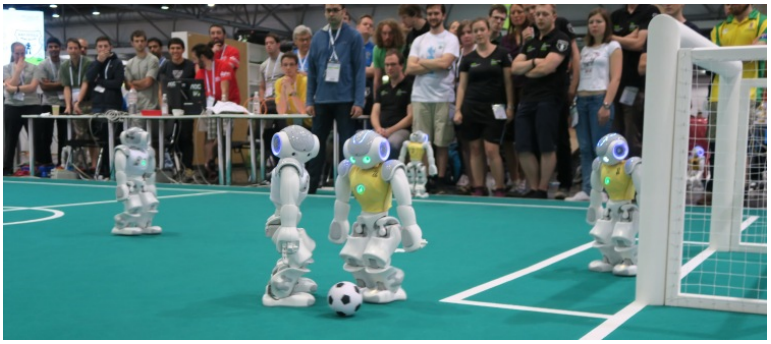


Figure 1.2: Picture from the 'Standard Platform' category in RoboCup. © RoboCup Federation

1.2.2 RoboCup

Started in 1993 by Japanese researchers, it includes both simulated and physical games of soccer played by autonomous agents. It has since evolved to include other challenges as well, such urban search and rescue. The contenders are researchers from all around the world, and many conference papers on robotics and computer vision has been published in relation to the topic. Not all robots used in RoboCup challenges use RGB-D sensors, but as the sensors became accessible, the vast majority of robots use them. [Matamoros et al., 2019]

2 Survey of existing technology

This section discusses the most commonly used hardware and software for capturing and processing data from RGB-D cameras.

2.1 RGB-D cameras

RGB-D cameras are cameras (or systems of several cameras) that provide video streams of both color and depth information. These became easily available with the PrimeSense sensors used most famously in the Microsoft Kinect, released 2010. Later, several other manufacturers have produced simmilar products. In the cameras discussed below, there are three different

Figure 2.1: The XBox One Kinect (top) and the older XBox 360 Kinect (below) © Microsoft

technologies to capture depth in these cameras: structured light, stereoscopic vision, and time-of-flight. I will explain briefly how they each work, and discuss their respective strenghts and weaknesses when we get to them. Common for all the cameras is that they have multiple lenses that are offset from eachother. This causes a misalignment of the images. Some of the cameras support transforming the streams to correct for this on board, at the expense of resolution. Now I will discuss a few relevant cameras to give a more precise view of todays assortment.

2.1.1 Microsoft Kinect

The first Kinect was released for the Microsoft Xbox 360 game console in 2010. Later versions was released for Microsoft Windows and the Microsoft Xbox One game console. Kinects are based on a depth sensor which uses an infra red projector along with a monochrome CMOS sensor to estimate depth in the on board chip. All Kinects made for gaming purposes are today discontinued, but Microsoft has announced the Azure Kinect, a revival of the project made specifically for the Azure cloud computing system, more targeted towards computer vision and serious applications.
The first Kinect, for XBox 360 used a structured light sensor for capturing depth data. The camera is built with a projector emitting a grid of dots of IR light and a sensor with a filter that only lets through IR light. If the on board processor can find a projected dot in the observed image, it can then calculate the depth using triangulation [Zeng, 2012]. The reliance on IR light and the projected dot pattern struggles in IR heavy environments such as in direct sunlight. Several structured light sensors pointed at the same location may also interfere with eachother, as they cannot distinguish their own dots from those of another camera, all though there are ways to mitigate this [Butler et al., 2012].
For the later version of the Kinect, for XBox One, they used a time-of-flight sensor. Time-of-flight sensors also projects light, but rather than trying to detect dots and triangulate, they measure the round trip time of the light and estimate distance from that.

2.1.2 Orbbec Astra

Orbbec 3D provides several RGB-D camera solutions. Most relevant for this report are their Astra series. The Astra cameras work in principle much the same as the Kinect, but the sensors are based on Orbbecs own technology. The standard Astra camera has very similar technical specifications to that of the Xbox Kinect cameras in terms of FPS, resolution and range. The Astra S camera trades lower range for increased precision, while the Astra Pro camera has an improved RGB sensor allowing it to capture higher resolution images. The Orbbec Astra line

Figure 2.2: The Orbbec Astra RGB-D camera © Orbbec 3D

uses structured light sensors much like the first Microsoft XBox Kinect, and suffers the same drawbacks.

2.1.3 Intel RealSense D4**

Intel released in early 2018 a pair of RGB-D cameras, similar to the Orbbec Astra and the Microsoft Kinect, namely the Intel RealSense D415 and D435. Both cameras capture high resolution RGB images, and offer more controll to the programmer to trade bandwidth for higher resolution and/or faster capture rates. They both contain the same Intel RealSense Vision Processor D4 on board, but use slightly different sensors for capturing depth data. The D435 provides a slightly larger field of view. The cameras have the option of directly synchronizing the capture times with each other, allowing more efficient multi-camera setups [Grunnet-Jepsen et al., 2018]. The focus on multi-camera setups is quite a unique thing for these cameras, as both structured light sensors and time-of-flight sensors are vulnerable to interference from other cameras. The Intel RealSense D4** cameras uses stereoscopic vision instead. By using two sensors, the D4** cameras estimate depth by looking at disparities between matching keypoints seen by both sensors. This removes the need for projecting a camera specific grid that can conflict with other cameras or be hard to recognize in IR heavy scenes. Rather, with stereoscopic vision, we only need some texture on the surfaces we observe to match between the sensors. In order to provide this even in dark or non-textured environments, the D4** cameras are also outfitted with projectors that emit some pattern of IR light. Having several of these overlapping is not an issue however, as they are only used to add visible texture [Dorodnicov, 2018].

Figure 2.3: The Intel RealSense D435 © Intel

2.1.4 Comparison of select RGB-D cameras

Camera	Color stream	Depth stream	Range	FOV
Microsoft XBox 360 Kinect	640x480 30 FPS	320x240	1.2 m to 3.5 m	57°
Microsoft XBox One Kinect	1920x1080 30 FPS	512x424	0.5 m to 4.5 m	70°
Orbbec Astra	640x480 30 FPS	640x480	0.6 m to 8 m	60°
Orbbec Astra S	640x480 30 FPS	640x480	0.3 m to 2 m	60°
Orbbec Astra Pro	1280x720 30 FPS	640x480	0.6 m to 8 m	60°
Intel RealSense D415	1920x1080 30 FPS	848x480	0.16 m to 10 m	65°
Intel RealSense D435	1980x1080 30 FPS	848x480	0.11 m to 10 m	87°

Note that for the Intel RealSense cameras, there are many possible configurations for resolution and capture rate. The table shows defaults only.

2.2 Available software

All camera manufacturers mentioned above provide their own software for interfacing with their cameras. These SDKs are limited to using the specific line of sensors, but provides the best ease of use and most fine grained control of the various extra features the cameras are outfitted with. There are also a few software packages available that are more or less camera agnostic, and focus on the shared functionalities of RGB-D cameras.

2.2.1 OpenNI 2

OpenNI was created by PrimeSense, the manufacturers of the sensors in the original Microsoft Kinect, and developed as open source software. PrimeSense was purchased by Apple in 2013 however, and it was announced that the website for OpenNI would be shut down. Other partners then forked the software and formed what is OpenNI 2[2]. OpenNI 2 is an open source camera independent library that provides access to basic functionalities of RGB-D cameras from different brands. OpenNI 2 does not provide point cloud data frames directly, only color and depth, but it does provide the programmer with enough information to compute this themselves. As this is the only open source and cross platform library available, it is my choice for implementing the practical part of this project. I will introduce it in more detail in subsection 3.1.

[2]More info on OpenNI 2: `https://structure.io/openni`.

2.2.2 Intel RealSense 2 SDK

Intel's RealSense 2 SDK is open source and free software[3]. It is a common SDK for interfacing with the Intel RealSense D4** line of RGB-D cameras as well as other Intel cameras, and include tools such as point cloud viewers and debug tools for device enumeration and device logs, as well as examples and wrappers to a variety of programming languages. They have a more highlevel API, as well as a more low level one. Of particular note is the focus on providing simple support of multiple camera devices running simultaniously. The SDK does not support skeleton tracking, but can provide such data if used with camera sensors that can do the processing on its on board chip. Point cloud frames with position and RGB color is available however.

2.2.3 Orbbec Astra SDK

The SDK for the Astra line of RGB-D devices is also open source and free, under the same license as the Intel RealSense 2 SDK [4]. Point cloud streams are not available, but can be computed using the depth streams and the camera information, like when using OpenNI. In addition, Astra SDK provides access to the infrared color stream, and a hand tracking stream. Full skeleton tracking is available through a proprietary plugin.

2.2.4 Nuitrack SDK

Nuitrack SDK is a third party SDK that is compatible with the Orbbec Astras, Intel RealSense D4**s, and the Microsoft XBox Kinects[5]. It is proprietary software and available under a subscription based fee. The SDK provides the normal color and depth streams, as well as skeleton tracking, a more coarse user tracking, hand tracking, and gesture detection. They have announced a new version that is to be released in the second half of 2019 called Nuitrack AI that will do skeleton tracking and face recognition using deep learning.

2.2.5 Microsoft Kinect SDK

Even if the XBox Kinects are discontinued, the Kinect SDK is still available. It offers the standard color and depth streams, as well as infrared, face, body, and body index frames. The body index frames are segmentated depth images where each pixel is classified as either background or a body index. Face tracking capabilities was added with the release of XBox One Kinect and include support for detecting glasses, closing eyes, and smiling.

2.3 Point Cloud Library (PCL)

PCL is a large open source software library for processing point clouds. It is contributed to by researchers, professionals, and hobbyists from all over the world. It provides implementations of algorithms to align point clouds, do segmentation, and query for features such as surfaces and other shapes. It also contain a module for visualization. As this report focuses mostly on the RGB-D cameras, I have chosen to not use PCL in this project, but I would strongly consider using it for more directed research into use of point clouds in further work.

[3] Intel RealSense 2 git repository is available here: `https://github.com/IntelRealSense/librealsense`.
[4] Orbbec Astra SDK git repository is available here: `https://github.com/orbbec/astra`.
[5] Nuitrack web page: `https://nuitrack.com`

3 Background

3.1 OpenNI 2 Overview

Here I will briefly introduce OpenNI 2[6], and how to use it to interface with RGB-D cameras. Let us first look at the project structure, and how the library handles various hardware. (fig shows folder structure). Drivers are developed and stored in separate dynamic libraries that expose the API defined in `OpenNI/Driver/OniDriverAPI.h`. This API consist of several callbacks and functions for initialization and finalization. This makes it easy for a camera manufacturer to provide new drivers and have the cameras work on existing OpenNI software.
OpenNI exposes both an interal C API, as well as a C++ wrapper API. The C API makes creating other wrapping APIs written in other languages such as Python easier. I will use the C++ API for this demonstration.
OpenNI has some global state that must be initialized before use, and finalized when we are done. This state includes opening the driver libraries and initializing them. Listing 1 shows the function calls for initialization and finalization.

Listing 1: OpenNI2 initialization and finalization.

```
// Initialization:
openni::Status rc = openni::OpenNI::initialize();
if(rc != openni::STATUS_OK)
{
    // Handle error
}

// Finalization:
openni::OpenNI::shutdown();
```

After initialization we can poll for connected sensors using the `openni::OpenNI::enumerateDevices()` function. This functions fills a `openni::Array<openni::DeviceInfo>` with basic information on each connected device. The information given are name (used for identification), URI (used later to uniqely identify and initialize a specific device), vendor name, USB vendor ID, and USB product ID. For functionality, applications only need to care about the URI. To initialize a camera we must create an `openni::Device` object and call its `open()` method with the wanted URI as the only parameter. Listing 2 shows how to open a device, start both color and depth streams, and clean up everything.

Listing 2: Opening a device, starting both streams, and capturing a single frame of each.

```
openni::Status rc; // return code is used to report errors.
openni::Device oni_device;
rc = opennioni_device.open(URI);
// Error handling should happen here, and after any other
// function that return a status code, but is omitted from
// this example.

openni::VideoStream color_stream;
rc = color_stream.create(oni_device, openni::SENSOR_COLOR);

openni::VideoStream depth_stream;
rc = depth_stream.create(oni_device, openni::SENSOR_DEPTH);

rc = color_stream.start();
rc = depth_stream.start();
```

[6]The OpenNI 2 source code can be found here: `https://github.com/occipital/OpenNI2`.

```
openni::VideoFrameRef color_frame;
rc = color_stream.readFrame(&color_frame);
const openni::RGB888Pixel *color_pixels =
    (const openni::RGB888Pixel *)color_frame.getData();

openni::VideoFrameRef depth_frame;
rc = depth_stream.readFrame(&depth_frame);
const openni::RGB888Pixel *depth_pixels =
    (const openni::RGB888Pixel *)depth_frame.getData();

// Use color_pixels and depth_pixels ...

// Cleanup
color_frame.release();
depth_frame.release();

color_stream.destroy();
depth_stream.destroy();

oni_device.close();
```

Since the lenses of most RGB-D cameras are horizontally offset from eachother, the depth buffer and the color does not have the same viewpoint and is not fully aligned. This can be compensated for somewhat however, and some cameras support doing this alignment in the on-board processor. OpenNI exposes this functionality first through the `openni::Device` method `isImageRegistrationModeSupported` and `setImageRegistrationMode`. Both of these methods takes a value of the `ImageRegistrationMode` enum which can be either be `IMAGE_REGISTRATION_OFF` or `IMAGE_REGISTRATION_DEPTH_TO_COLOR`.

3.2 Acceleration structures for Point Clouds

Point clouds are often quite massive data sets. [7] It is therefore frequently to build some form of acceleration structure to organize the points in to speed up later operations. This is a field of study in it of it self, so I will only cover two select structures briefly.

3.2.1 BSP and k-d trees

Binary Space Partitioning (BSP) is a way of building a binary tree where each internal node represents a plane that splits space in two. The root defines a plane that splits the entire point cloud in two, and each subtree recurses this process.
k-d trees are a specialization of this where each leaf node is a k-dimensional point, and every splitting plane is aligned to one of the k axes. There are several ways to build k-d trees, but a popular option is to cycle through the axes such that in the case of $k = 3$ the root splits by the x-axis, the first subtrees split by the Y-axis, their subtrees split by the Z-axis, and their subtrees splits the X-axis again.

3.2.2 Octrees

Octrees are trees where each node has exactly eight nodes (except leaf nodes, of course). When used as an acceleration structure, each node represents one octant of the space contained by

[7] In my system, each sensor adds approximatly 300 000 points to the cloud

its parent [Meagher, 1980]. By organizing the points of a point cloud into an octree we can quickly query for the subset of points contained in some region of space. To build an octree, one typically starts with a single node that spans all points in the cloud, then recursively subdivide it into octants. The recursion stops when some exit condition is met, such as the number of points in a bin goes below some treshold, or a maximum recursion level is reached.
When the octreee is built, we can query for subsets of points by first querying for the leaf nodes that intersect our query volume and take all points contained by those leaf nodes.

4 Practical work

I have made a program to align and visualize point clouds captured from one or several cameras at real time. It is a robust starting point for implementing existing or new algorithms, depending on the target application. The main focus area have been to create a real time system to allow for user detection and interaction.
All development and testing has been done with a late 2015 MacBook Pro and two Orbbec Astra cameras.

4.1 Architechture

The motivation was for creating the practical part of this project was to get hands on experience with various RGB-D cameras, but also to create a starting point for rapid prototyping of applications that use this technology. Figure 4.1 shows the project folder structure. To meet my goals I had 3 cirterias in mind when designing the software:

1. Flexibility. The code must be easy to change.
2. Simplicity. The code must be easy to read and understand.
3. Fun. Edit-Compile-Run cycles must be as fast as possible, and user interaction should be smooth.

To achieve point 1 I gathered most functionality in decoupled units, and tied them together in the concept of scenes. Scenes can handle user input, visualization, and data processing as it fits the current needs best. To create a new prototype or to experiment with a new idea, one simply creates a new scene or duplicates an existing one, and it is ready to include whatever existing functionality needed, or implement new features. When done, features can be extracted from scenes into new decoupled modules, ready for use in other scenes.

For point 2 I chose to code in a style more akin to C than C++ by disregarding most C++ features such as RAII, exceptions, and the STL. This makes the code easier to understand for non-C++ programmers as no knowledge of outside libraries or obscure language details are needed. This is quite controversial and mostly based on my personal experience and preference.

Lastly, for point 3 I choose to structure the code for a so-called unity build[8] to enable fast compilation. This decision might be confusing for newcomers to the project, conflicting with point 2, but in fact it simplifies the build process a great deal, and puts very little demands on the build system.

Because of the need to display large point clouds I chose OpenGL to implement GPU instanced rendering of the point cloud. This allowed me to render the point clouds efficiently and focus

[8] See this Wikipedia article for more information on unity builds: `https://en.wikipedia.org/wiki/Single_Compilation_Unit`

```
Makefile
OpenNI
OpenNI2 (Drivers)
imgui
launchpad
shaders
├── cube.glsl
├── frustum.glsl
├── full_quad.glsl
├── instanced_cubes.glsl
└── wire_cube.glsl
src
├── camera.cpp
├── camera.h
├── files.cpp
├── files.h
├── frustum.h
├── gl.h
├── imgui_impl_opengl3.cpp
├── imgui_impl_opengl3.h
├── imgui_impl_sdl.cpp
├── imgui_impl_sdl.h
├── input.cpp
├── input.h
├── magic_math.h
├── main.cpp
├── octree.cpp
├── octree.h
├── renderer.cpp
├── renderer.h
├── scene.h
├── scene_interaction.cpp
├── scene_interaction.h
├── scene_video.cpp
├── scene_video.h
├── scene_viewer.cpp
├── scene_viewer.h
├── sensor_interface.cpp
├── sensor_interface.h
├── sensor_serialization.cpp
├── sensor_serialization.h
├── stb_image.h
├── ui.h
└── utils.h
```

Figure 4.1: The project structure at the time of writing

on data processing. For quickly creating graphical user interfaces I added the Dear ImGui library[9]. This allows for quickly adding and removing controls with minimal code change, while adding little processing overhead. This meshes very well with both code criteria 1 and 3.

4.2 Point cloud alignment

When using two or more cameras in conjunction, it is neccessary to align the point clouds from each camera to some common world space. Since all sensors encountered report the depth information in millimeters, one does not have to worry about scaling the point clouds. Also, there is no need to shear, as the point projection is also the same. What we need to define is a rigid transform, that is a transform consisting of a translational component and a rotational component.
Again, as the cameras report depth in millimeters, I have chosen to use millimeters as the standard unit in my system.
In my system, I let the user do the alignment manually by providing the position X, Y, Z as well as the rotation $pitch, yaw, roll$, all relative to som arbitrary origo and identity rotation[10].
During use of the system, I have made some observations;
Rotation is much harder to align manually than position. Aligning position after rotation is correct is easy.
Measuring the distance with a ruler or tape measure in the real world gives a good first estimate for the position.
After rough positional alignment is done, the rotational part of the transform should be done, before fine tuning the positions.
To align the rotations, I select one or more flat surfaces that is seen by all the sensors I am aligning. I then adjust the rotation of all sensors until the face normals of the surfaces are the same in every sensors view. If all cameras can see a common open corner, that is ideal, because you have three flat perpendicular surfaces to align by.

4.3 User interaction

Interaction is achieved through placing axis aligned bounding boxes (AABB) in the scene, and doing collision detection on them against the point cloud. I can then fire events when something enters or leaves an AABB. To test if a single point is within an AABB, I make use of the Separating Axis Theorem (SAT) implemented especially for point vs AAB in Equation 4.1. Note that the algorithm checks if the point is not outside, rather than checking if it is directly inside. The latter would be simpler, but in most cases points in the point cloud will not be within one of the AABBs, so implementing it this way lets us shortcut the rest of the function as soon as we can see that the point is outside.
To avoid issues with noise, I set a treshold value T such that there must be more than T points within an AABB to count it as a collision.

[9] Dear ImGui git repository: `https://github.com/ocornut/imgui`
[10] $0, 0, 0$ and in the direction of the Z axis in the virtual world space, but arbitrary in relation to positions and directions in the real world.

$$\begin{aligned} inside(p, aabb) = \neg(&p.x < aabb.center.x - aabb.size.x/2 \vee \\ &p.x > aabb.center.x + aabb.size.x/2 \vee \\ &p.y < aabb.center.x - aabb.size.y/2 \vee \\ &p.y > aabb.center.x + aabb.size.y/2 \vee \\ &p.z < aabb.center.x - aabb.size.z/2 \vee \\ &p.z > aabb.center.x + aabb.size.z/2) \end{aligned} \tag{4.1}$$

These interactions provide great flexibility; By using large AABBs I can react on events such as object s or people entering or exiting the space. Or conversely, by using thin AABBs close to the walls, I can react on users touching said wall.

4.4 Optimization

The collision detection used for user interaction in the beginning iterated every collider AABB for every point. This is an $O(mn)$ algorithm where n is the number of points in the cloud, and m is the number of AABBs. I chose to improve on this by dividing the collision detection step into a broad phase and a narrow phase. In the broad phase, I check each AABB against an acceleration structure to get a set of candidate points that may lay within the AABB. I can then do linear sweep (same as before), but now on much fewer points. The acceleration structure of choice is an octree as described in subsection 3.2. I add points to the octree as I calculate them from the depth images. Then I can later query the AABBs against the octree to get a list of points that potentially lays within the AABB, and iterate only those. For a low number of AABBs, this is less efficient than the naive linear iteration because $O(mn)$ becomes $O(n)$ if m is small. But it scales better as the time complexity for querying points is $O(mlog(n))$ and building the octree is $O(nlog(n))$.
I did not manage to finalize the octree implementation before the deadline, so sadly I have no profiling data to show yet. When complete, I will try with various amounts of AABB to find when the overhead of builing the octree becomes beneficial.

5 Future work

Lastly, I will discuss some interesting avenues for future research, and improvements to the software I have written.

5.1 Noise reduction

The depth image of all sensors I have tested have noise. Figure 5.1 shows two captures taken in quick sequence with a Orbbec Astra camera of a static object. In this example 24% of the pixels changed value with an average difference of 0.9%. Maximum difference is 36.5%. Figure 5.2 visualizes the difference, and we can observe that most of the noise appears around critical areas, such as edges.

Noise in the depth image is caused by inaccuracies from the sensor, and failure to compute the depth value of some pixels. Another form of noise is banding, shown in Figure 5.3[11], which is also caused by inaccuracies in the sensor. Figure 5.4 shows the banding noise with a different

[11] This colorization is a simple normal vector estimation. It is used here because it makes the banding very obvious

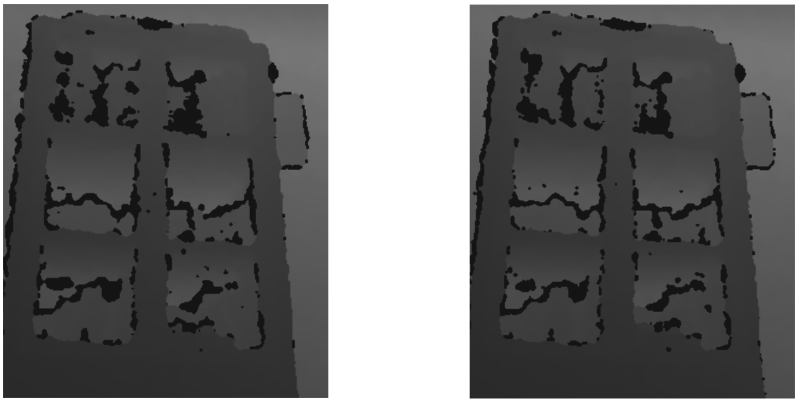


Figure 5.1: Two depth frame images taken in quick succession

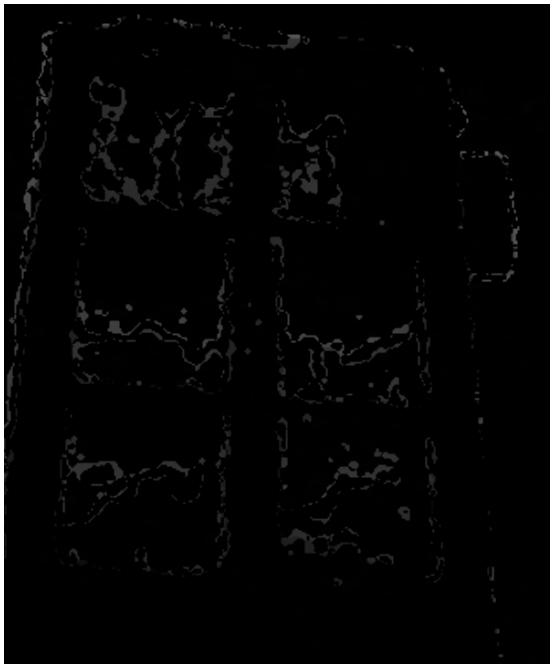


Figure 5.2: Darker pixels show less change. Bright pixels changed alot

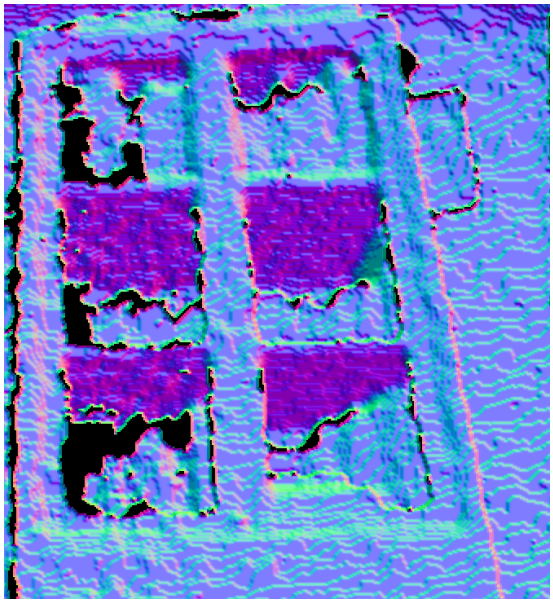


Figure 5.3: Banding issues in a depth frame

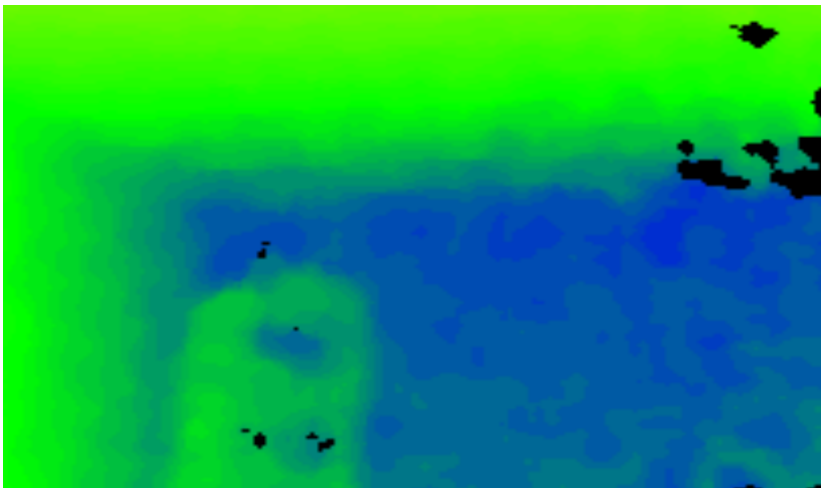


Figure 5.4: Banding issues get noticeably worse at longer distances

colorization.
While not very extreme in closer range, some procedure for noise reduction could improve performance and results of later processes, such as normal direction estimation. The two main ways we can reduce noise is

1. Spatial. Look at neighbourhoods of pixels.
2. Temporal. Look at how the pixel changes over time.

Spatial smoothing is the most obvious way to handle the issues with banding, but excessive spatial smoothing can lead to blurred images, which can lead to other artifacts down the line. More subtle smoothing is not so likely to cause issues, but might not have enough of an effect. Temporal smoothing does not have the issue with blurriness, and should work very well for static scenes. However, drastic temporal filtering would cause issues with dynamic scenes and rapid movement. As future work, I would like to try and implement the method proposed by [Yahya et al., 2015] which does both spatial and temporal filtering, and compare to more naive methods such median filtering (both spatial and temporal) in terms of both quality and computing time overhead.

5.2 Pruning

The current systems naively merges point clouds from all available cameras. In some cases when the camera view volumes overlap, many points are very close to eachother, and essentially becomes redundant. It would improve the performance of the system to merge points from different sources in a more clever way, mostly in terms of execution speed, but possibly also in reduction of noise. By building an acceleration structure as discussed in subsection 3.2 as we generate the point cloud from the depth data, we can query for existing nearby points and ignore them should they be too close.
By reducing the number of points we will reduce time spent on any subsequent iteration over the point cloud, as well as rendering. It remains as future work to implement and profile this to see if the benefits are worth the overhead of pruning.

5.3 Surface detection and segmentation

As of yet, the point cloud that gets computed only cares about the spatial position of each point. It could be useful to a add suport for labeling the points as well. As discussed in [Tchapmi et al., 2017], CNNs are a great candidate for this, but as they require a regular grid, we need to voxelize the point cloud before use. [Tchapmi et al., 2017] then proposes to do point-wise classification by using trilinear interpolation and another neural network.
Having an efficient and reliable way of separating humans from the environment will simplify a great many later operations, such as skeleton tracking and user interaction as described in subsection 4.3. Also getting major surfaces labeled such as walls, floor, and ceiling could greatly help automating the issue of multi camera point cloud alignment as discussed in the next subsection.
Training could be done on synthetic data generated from virtual 3D scenes. Since the input data is fairly coarse point clouds, it is reasonable to assume the synthetic data will be close enough to real world data to be useful, but it still remains to be seen. There has been research done before on training CNNs on synthetic data before fine tuning on real world data that have shown very positive results [Barbosa et al., 2017].

The Microsoft Kinect, Orbbec Astra SDK, and Nuitrack all do some segmentation to provide skeleton streams. Their algorithms are propriatary, so we can only guess at how they are implemented. However, Nuitrack has announced that they will soon be releasing a new SDK which will be using 'deep learning'. It is therefore reasonable to assume previous SDKs use

traditional computer vision. Skeleton tracking specifically has been done also without segmentation, such as with PoseNet [Oved, 2018]. They show that with only RGB images, they can do real time pose estimation/skeleton tracking. A future experiment could compare the performance of PoseNet on RGB images to the performance of SEGCloud on point cloud data.

5.4 Automatic multi camera point cloud alignment

The current system needs the user to align the point clouds manually by setting the rotations and positions. This can in some cases be automated, as done by [Park and Choi, 2014] by setting the camera(s) in a calibration mode and introducing a special object that is easy to recognize.
Another aproach is to let an operator define the surface normal of some automatically detected planes in the scene. For instance, it is often desirable to let the floor have a surface normal pointing up. By letting an operator tell the system that a detected surface is the floor, we can find a rotation such that the normal of the detected surface can be transformed to the desired 'up-vector'. This takes care of rotation, which as discussed in subsection 4.2 is the most difficult part to do manually. Then the operator could manually do the translation part of the required rigid transforms.
There are also several ways we could automatically do the translation part as well, such as the method with a calibration object as done in [Park and Choi, 2014].
For static scenes, we can define landmarks to remove the need to introduce a special object for calibration. Landmarks should be simple features to recognize. With three or more such landmarks, visible from all point clouds, we can use least squares plane fitting to find the complete rigid transform [Khandelwal et al., 2012].

5.5 Biometrics and re-identification

RGB-D cameras have been applied to the re-identification problem before [Barbosa et al., 2012]. By using the 3D shape of people to (re-)identify them, it is implied that there is some biometric data we could extract, such a 'somatotypes' [Barbosa et al., 2017]. With accurate skeleton tracking, it is conceivalble that we could calculate numbers such as height, shoulder width, and various proportions and combine them to create a numeric tag we could use as a biometric ID and later for re-identification. Even without a good skeleton, we can compare depth frame based re-identification to point cloud based re-identification. It has been shown that the number and variety of viewpoints have a big impact on the performance of re-identification systems [Sun and Zheng, 2018]. Helping the system to understand the 3D nature of the depth buffer by computing the point cloud first, or even better create a point cloud from multiple depth frames taken from different viewpoints, seems likely to provide a big benefit. Testing and measuring this would be interesting.

6 Conclusions and Summary

I have charted the market of RGB-D sensors, and gotten a good overview of the various applications and the technologies behind them. I have implemented a software system that is freely avaiable, and ready to use as a good base for further work and research. Lastly, I have looked at a great deal of future research to be done, and considered several improvements to existing applications (such as synthetic point cloud data for training 3D segmentation systems) and new uses for the technology (such as calculating biometric data).

This project has been quite whimsical, and I have read and learned a great deal about RGB-D cameras and the various technologies that power them and many of their uses, but also about point clouds and ways to work with them. I now feel confident in going into a masters project within this field and being more focused and thorough.

7 References

[Barbosa et al., 2012] Barbosa, I. B., Cristani, M., Bue, A. D., Bazzani, L., and Murino, V. (2012). Re-identification with rgb-d sensors. *Proc. ECCV - Workshops and Demonstrations*.

[Barbosa et al., 2017] Barbosa, I. B., Cristani, M., Caputo, B., Rognhaugen, A., and Theoaris, T. (2017). Looking beyond appearances: Synthetic training data for deep cnns in re-identification. *Computer Vision and Image Understanding*.

[Butler et al., 2012] Butler, A. D., Izadi, S., Hilliges, O., Molyneaux, D., Hodges, S., and Kim, D. (2012). Shake'n'sense: Reducing interference for overlapping structured light depth cameras. *Proc. of SIGCHI*.

[Dorodnicov, 2018] Dorodnicov, S. (2018). The basics of stereo depth vision. `https://www.intelrealsense.com/stereo-depth-vision-basics/`. Accessed: 2019-12-11.

[Grunnet-Jepsen et al., 2018] Grunnet-Jepsen, A., Winer, P., Takagi, A., Sweetser, J., Zhao, K., Khuong, T., Nie, D., and Woodfill, J. (2018). Using the intel realsense deph cameras d4xx in multi-camera configurations. Technical report, Intel.

[Khandelwal et al., 2012] Khandelwal, P., Khandelwal, P. S., and Stone, P. (2012). A low cost ground truth detection system for robocup using the kinect. *RoboCup 2011: Robot Soccer World Cup XV*.

[Matamoros et al., 2019] Matamoros, M., Seib, V., and Paulus, D. (2019). Trends, challenges and adopted strategies in robocup@home. *arXiv:1903.10882v1 [cs.RO]*.

[Meagher, 1980] Meagher, D. (1980). *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*.

[Oved, 2018] Oved, D. (2018). Real-time human pose estimation in the browser with tensorflow.js. `https://medium.com/tensorflow/real-time-human-pose-estimation-in-the-browser-with-tensorflow-js-7dd0bc881cd5`. Accessed: 2019-12-06.

[Park and Choi, 2014] Park, S.-Y. and Choi, S.-I. (2014). Convenient view calibration of multiple rgb-d cameras using a spherical object. *KIPS Transactions on Software and Data Engineering*.

[Reisinger, 2017] Reisinger, D. (2017). Microsoft has finally killed the kinect xbox sensor. `https://fortune.com/2017/10/25/microsoft-kinect-xbox-sensor/`. Accessed: 2019-12-04.

[Sun and Zheng, 2018] Sun, X. and Zheng, L. (2018). Dissecting person re-identification from the viewpoint of viewpoint. *CVPR*.

[Tchapmi et al., 2017] Tchapmi, L. P., Choy, C. B., Armeni, I., Gwak, J., and Savarese, S. (2017). Segcloud: Semantic segmentation of 3d point clouds. *International Conference on 3D Vision (3DV)*.

[Yahya et al., 2015] Yahya, A. A., Tan, J., and Li, L. (2015). Video noise reduction method using adaptive spatial-temporal filtering. *Discrete Dynamics in Nature and Society*.

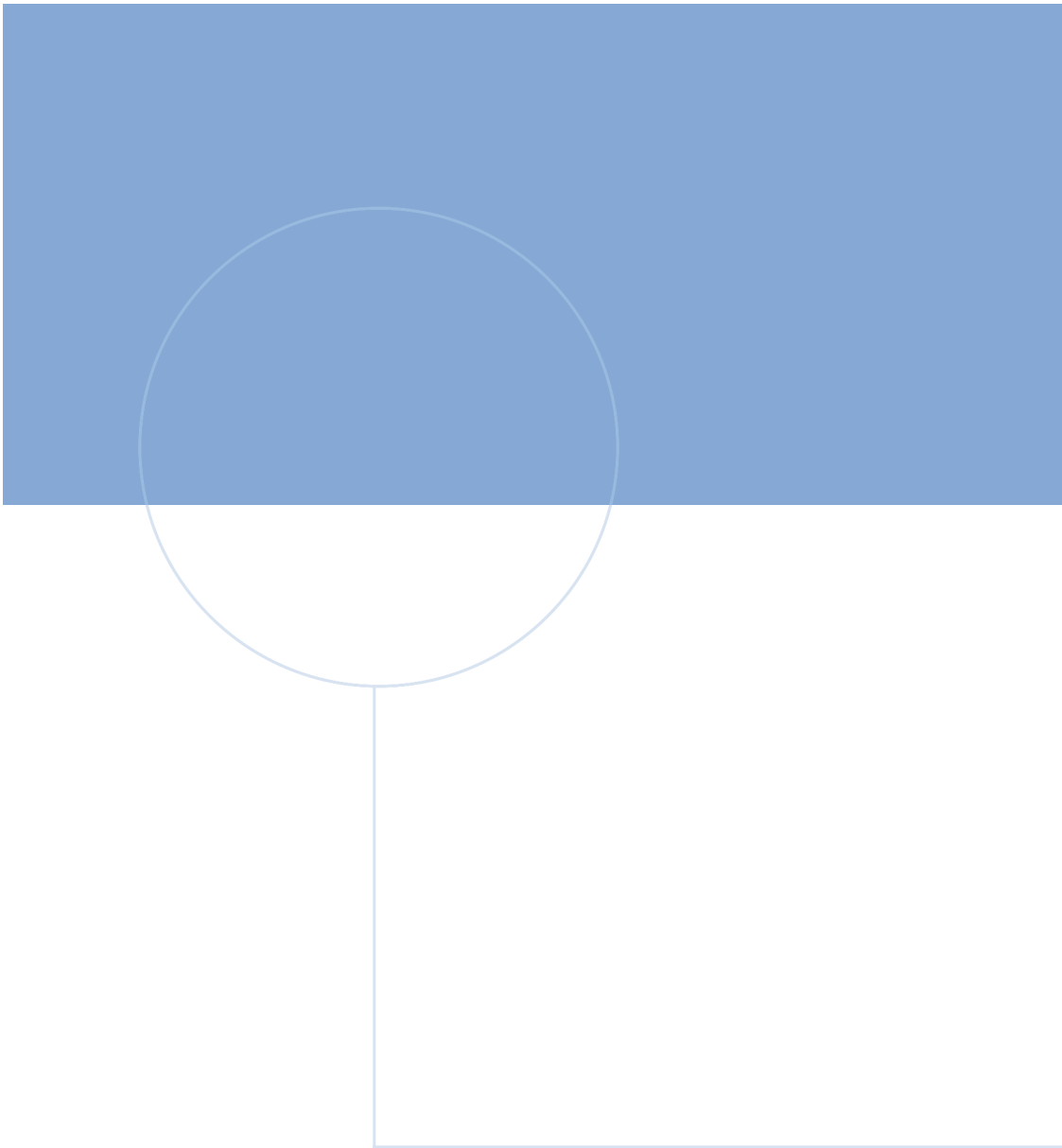
[Zeng, 2012] Zeng, W. (2012). Microsoft kinect sensor and its effect. *Multimedia at Work*.

B Dataset

The dataset is hosted at the following URL: `https://github.com/Istarnion/MagicMotion/releases/tag/v1.0`
It should also be available as an attachment to this thesis where ever it is published.

C Source code

The source code for the implementation of the suggested method is hosted on GitHub at the following link: `https://github.com/Istarnion/MagicMotion`



NTNU

Norwegian University of
Science and Technology