2018

Accelerating structured light surface reconstruction with motion analysis

Quinn Murphy
Iowa State University

Accelerating structured light surface reconstruction with motion analysis

by

Quinn Murphy

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Computing and Networking Systems)

Program of Study Committee:
Joseph Zambreno, Major Professor
Phillip Jones
Thomas Daniels

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2018

DEDICATION

I would like to dedicate this thesis to my parents Jay and Ann and my brother Drew for always encouraging me to follow my dreams and putting up with me when I get excited and talk about nerd things.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my friends Benjamin Williams and Joseph Avey for being my comrades throughout this process and reminding me that I have work to do. I would also like to thank Dr. Thomas Daniels for teaching me early in my college career not just how to program but how to think like an engineer. Finally, I'd like to thank Dr. Joseph Zambreno and Dr. Philip Jones for pushing me to challenge myself and keeping me motivated as I completed this thesis.

ABSTRACT

Structured light is the process of projecting depth-encoding features onto a surface and using a camera to build a 3D model of the surface. As 3D scanners make their way into more consumer electronics, the ability to quickly acquire 3D models has become more important. While 3D scanning has traditionally been either a slow process to acquire a high definition model or an inaccurate process to quickly grab a large model, we propose a novel implementation that concerns itself with accelerating the acquisition of 3D point clouds by pruning the search space to only objects that have moved since the last frame. By alternating between projecting a one-shot depth encoding pattern and white light, we can use the generate a motion mask using the white light frame and make the assumption that points not in motion can keep their previously decoded position. New locations will only be searched and computed for points that reside within the motion mask. This work is showcased and profiled in a software implementation running on a CPU as well as a CUDA implementation running on a GPU. This work shows significant improvements upon traditional structured light implementations for scenes with a moderate amount of motion in the camera field of view for many different classifications of motion, though these improvements are subject to diminishing returns as parallelization increases.

CHAPTER 1. INTRODUCTION

Structured light is one method of acquiring digital 3D models from real-world objects by projecting identifiable features onto a surface, using a digital camera to capture the where those features fell on the surface, searching for those features, and then extracting 3D information from the locations of the features. Branching some of the earliest work into structured light by Posdamer and Altschuler in 1981 [3], structured light has made its way into more and more consumer products in the form of the Microsoft Kinect in 2010, Google's Project Tango in 2014, and recently Apple's iPhone X in 2017 in the form of FaceID. With this proliferation of surface reconstruction technology, there is a need for higher resolution models to be generated at faster speeds.

Using structured light to acquire 3D models is useful for many different applications. As 3D printers become more affordable, there's a larger demand for online libraries of objects that people can create. Traditionally these models are created by experienced artists in 3D modelling software, though 3D scanners using structured light have allowed a larger number people to contribute to this library of things. These objects can also be imported into 3D engines to be used in virtual and augmented reality applications. Structured light is also used in industrial automation for quality control by scanning objects as they come off the assembly line to check for abnormalities. In the future, a fast structured light module could run on consumer-level devices to facilitate 3D video chat for not just faces but arbitrary objects in the scene as well.

In this work, we introduce a structured light system based on de Bruijn sequences which encodes codewords using neighborhoods of colored parallel vertical lines while also using

motion detection to reduce the number of patterns that need to be decoded at each cycle thereby accelerating the entire process. We accomplish this by alternating projecting the colored parallel line pattern and white light at 60 Hz. In the pattern frames, we search along epipolar lines in the camera view in order to find the desired codeword and triangulate its 3D location. In the white-light illuminated frames, we use a background subtractor to detect motion in the camera view and limit the codeword identification stage to only the pixels that have moved since the previous frame. If motion is detected at the last known location of a point, we attempt to search for its new position in the motion mask. We assume that if motion was not detected at the last known position, then we can keep the last decoded position for that point and save processing time. In the event that we incorrectly decode a point, we attempt to identify it using a radius outlier removal search that uses fixed-radius nearest neighbor search to ensure that a point is surrounded by enough other points to not make it an outlier. If it isolated, we remove it from the point cloud and force a search of that point's epipolar line for the next cycle so that we can attempt to find its proper place again. This step is important to this system because without it, misidentifications would propagate forward without having a chance to correct them.

This structured light system is implemented in C++ and CUDA (with the OpenCV library helping with some of the functionality). The inputs of to the algorithm are computer generated, though the system could easily be modified to support a real-world camera and projector. The evaluation shows that the use of motion analysis in structured light surface reconstruction can accelerate the process on a single-core CPU and a multi-core CPU, though not necessarily on a massively-parallel device such as a GPU. These improvements scale up as the amount of motion present in the frame goes down. We also show that while the inclusion of outlier removal is a large bottleneck in our implementation, many of the problems it solves are present in implementations without motion analysis and it is a good addition to the system regardless of whether motion analysis is used.

The rest of this thesis is organized as follows. Chapter 2 describes the mathematical

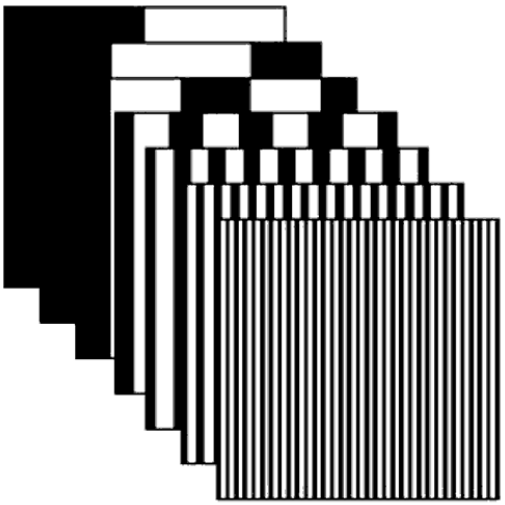
theory behind structured light and related algorithms as well as mixture-of-gaussian background subtractors which are used for motion detection. Chapter 3 covers related work to this thesis such as other structured light systems and how that work guided this thesis. Chapter 4 describes our contributions to the structured light problem and outlines the system that I will be evaluating. Chapter 5 covers the evaluation methodology so that our work can be compared to existing solutions. Chapter 6 contains the results of that evaluation. Chapter 7 discusses future work that can be done to improve the algorithm and concludes this thesis.

CHAPTER 2. BACKGROUND

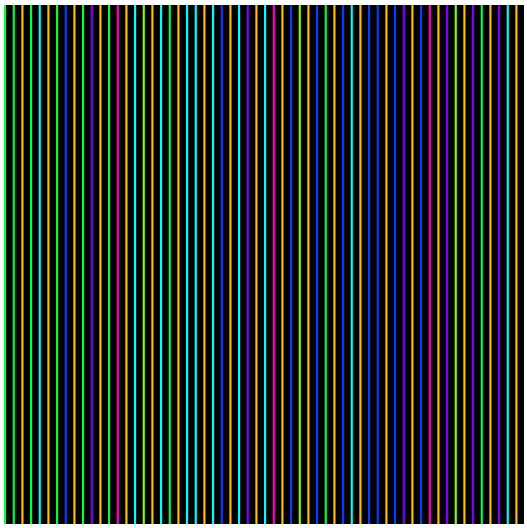
2.1 Structured Light

A structured light setup can be generalized as any system that uses one device to project light patterns that encode depth information and uses another device to detect the projected pattern to reconstruct a surface map. The patterns used to encode the depth of the surface can take many forms such as time-multiplexed patterns where sequences of patterns are projected over time (Figure 2.1a), patterns of colored lines generated with de Bruijn sequences (Figure 2.1b), patterns based on special types of matrices called m-arrays (Figure 2.1c), and more. These classifications are discussed further in 3 and work is still being done on better encoding schemes to this day. Structured light systems can also use infrared light in place of visible light. The main requirement is that the sensor can detect the light bouncing off the desired surface.

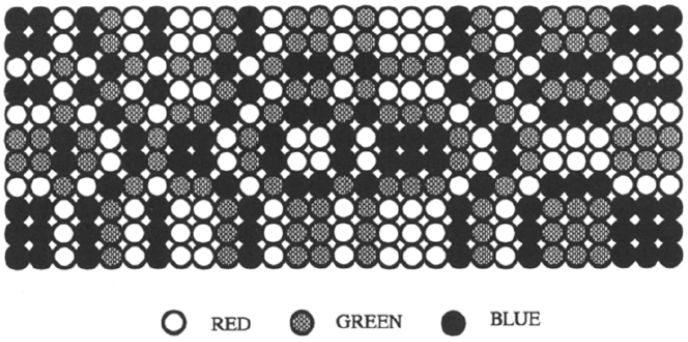
The advantage of structured light over other algorithms for surface reconstruction, such as traditional stereo vision which uses two cameras, is that it doesn't rely on the scene to have detailed features to find identical points in each field of view to calculate distance. These systems rely on these features to build disparity maps (or differences between the two fields of view). Instead, the projection system forces identifiable features onto the surface so that they only need to be identified in the camera's field of view as they are already known in the projector's field of view. Figure 2.2a depicts a traditional structured light configuration while Figure 2.2b depicts one for stereo vision.



(a) A time-multiplexed pattern used in [3]



(b) A de Bruijn sequence-based pattern encoded as colored lines used in this work

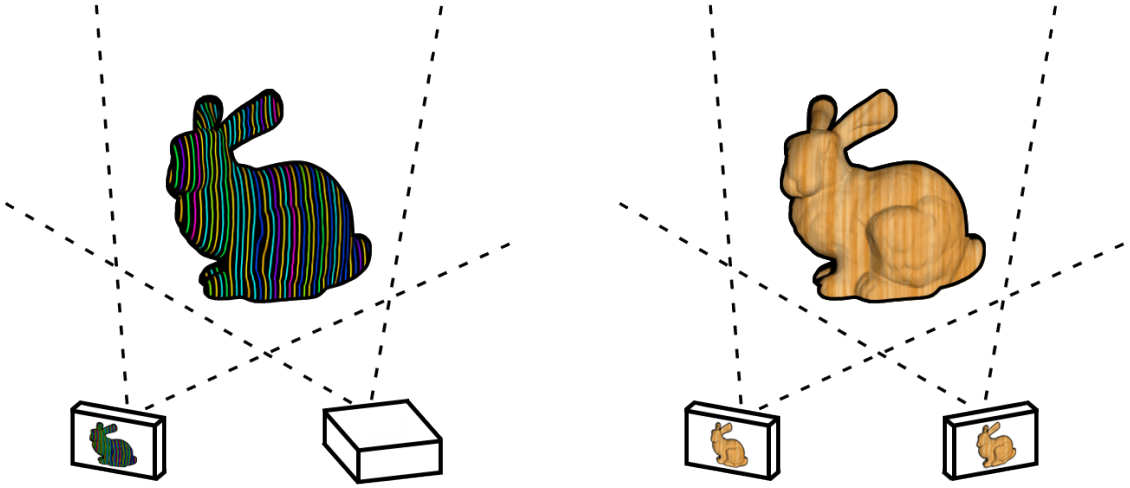


(c) An m-array-based pattern used in [4]

Figure 2.1: Various structured light encoding patterns

2.1.1 Multiple View Geometry

To reconstruct a 3D surface from a 2D camera view, we need to understand the mathematics of projections and camera imaging from not only a single camera, but from a multi-camera system. The same mathematical techniques can then be extrapolated to a camera-projector system which can be used for structured light by treating the projector as though it were a camera with a constant view.



(a) A camera and projector focusing on a subject in a structured light system

(b) Two cameras focusing on a subject in a stereo vision system

Figure 2.2: Typical surface reconstruction configurations

2.1.1.1 The Camera Matrix

Mutliple view geometry assumes that the images being processed follow the general projective camera model where a pixel on the image can be thought of as a ray starting at the projection center of the camera, moving outwards through the image plane at distance f (the focal length of the lens), and intersecting with real world objects that will then be captured in the image. This thesis specifically utilizes the pinhole camera model where all light hitting the image sensor passes through the center of the lens (Figure 2.3). Any point $(x, y, z)^T$ in $\mathbb{R}^3$ (three-dimensional coordinate space) can be mapped to $(f * x/z, f * y/z)^T$ in $\mathbb{R}^2$ (two-dimensional coordinate space) using the general projective camera model.

Homogeneous coordinates are often used in camera models to represent the fact that any point along a ray emanating from the camera center through the image plane projects onto the same coordinate of the image plane. By introducing an additional number k into the tuple and transforming $(x, y)^T$ to $(kx, ky, k)^T$, we can distinguish between points in the projection space that map to the same coordinates. Using homogeneous coordinates, the mapping from $\mathbb{R}^3$ to $\mathbb{R}^2$ is $(x, y, , 1)^T \mapsto (f * x, f * y, z)^T$. This mapping can be per-

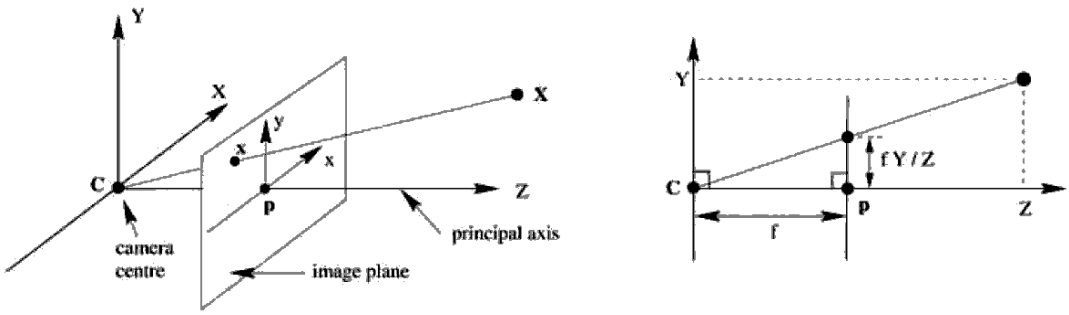


Figure 2.3: Pinhole model presented in [1]

formed by multiplying the 3x4 camera matrix P with the 3D homogeneous coordinate with Equation 2.1.

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fx \\ fy \\ z \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{2.1}$$

2.1.1.2 Camera Calibration

In physical cameras, we need a 3x3 calibration matrix K in order to account for real-world differences. This matrix contains the focal length in the x and y directions in terms of pixels (ax and ay, respectively), the principal point (center) in the x and y directions (x_0 and y_0, respectively), and a skew parameter s which is normally 0 unless the x and y directions are not perpendicular. This matrix is constructed as

$$K = \begin{bmatrix} a_x & 0 & x_0 \\ 0 & a_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Given a calibration matrix K, a rotation matrix R, the identity matrix I, and the camera's location $\widetilde{C}$, we can represent the real-world camera matrix P as

$$P = KR[I|-\widetilde{C}] \tag{2.3}$$

Figure 2.4 demonstrates the common types of radial distortions such as barrel (top) and pincushion (bottom) distortion introduced by lenses. These can be corrected for as well in order to transform the image back to the required rectilinear/Euclidean space.

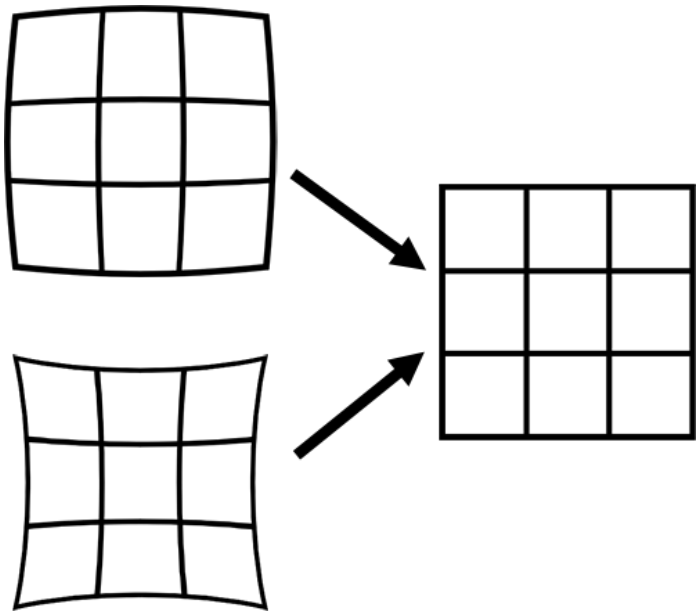


Figure 2.4: Barrel (top) and Pincushion (bottom) Distortion Correction

The correction is applied with the equations

$$\hat{x} = x_c + L(r)(x - x_c) \qquad \hat{y} = y_c + L(r)(y - y_c) \tag{2.4}$$

$$L(r) = 1 + \sum_{i=1}^{N} \kappa_i r^i \tag{2.5}$$

where $(\hat{x}, \hat{y})$ are the new, corrected coordinates, (x, y) are the old, distorted coordinates, are the center of the distortion, r is the distance from (x, y) to (x_c, y_c), and κ_i is a sequence of values computed by the camera calibration procedure. The exact values of κ are normally

calculated with a checkerboard pattern similar to Figure 2.5. In this thesis, due to the use of computer-generated images, this distortion correction does not need to be applied.

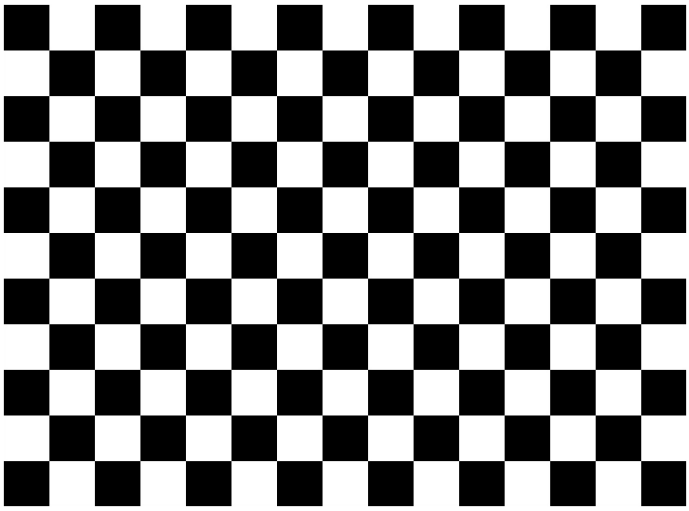


Figure 2.5: Checkerboard pattern used in camera calibration

2.1.1.3 Epipolar Geometry

Regardless of the method of finding correspondences between two fields of view, the same math is used to reconstruct 3D surfaces from 2D images. The underlying field of mathematics to do this is called epipolar geometry. If you know the specifications of the cameras/views (their intrinsic parameters), the relative pose (translation and rotation) between the cameras (their extrinsic parameters), and can identify identical points in both images (correspondence), you can triangulate the point in 3D space and reconstruct the surface.

For any point X in $\mathbb{R}^3$ and its corresponding projection in $\mathbb{R}^2$ onto the two views (x in the first view and x' in the second view), there exists a plane π that contains the spatial point X, the two image points x and x', and the centers of each view C and C'. Figure 2.6 illustrates this point. Supposing we only know the location of x in the first view, we can limit the search in the second view to only the line represented by the intersection of π and

the second view's image plane. The intersection of each view's center with each view's image plane is called the epipole. This limitation is called the epipolar constraint and such a line is called an epipolar line. Several epipolar lines in two views can be seen in Figure 2.7 The introduction of these concepts can significantly reduce the amount of searching required to find correspondence between two views.

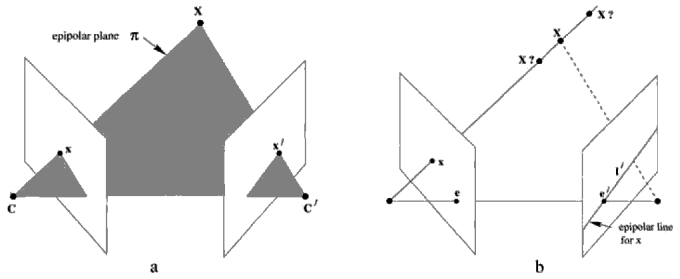


Figure 2.6: Point correspondence visualization from [1]

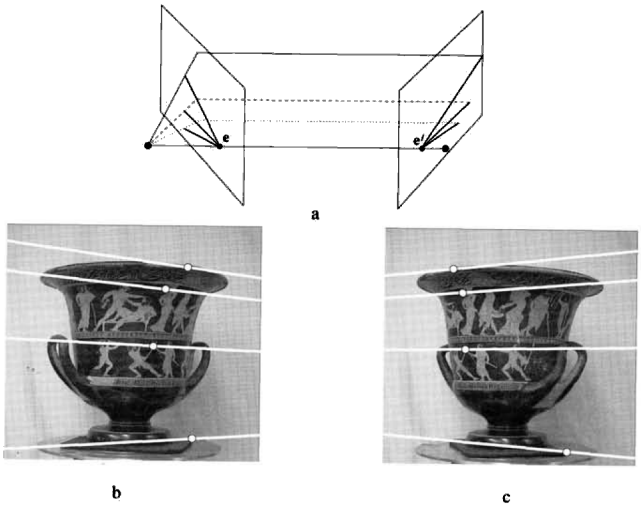


Figure 2.7: Epipolar lines visualization from [1]

Without epipolar lines, computation of correspondences between two views would be much slower and would present many more ambiguities. The epipolar constraint does not rule ambiguity out, especially in traditional stereo vision, though a carefully selected structured light pattern can significantly reduce ambiguity of correspondence.

2.1.1.4 The Fundamental Matrix

Once both cameras (or camera and projector, in the case of structured light) have camera matrices, we use something called the fundamental matrix to generate epipolar lines and determine how to map points in one field of view (e.g. a projector) to another field of view (e.g. a camera). Given a point in the second view x' and the corresponding epipolar line l' passing through x', and an epipole e', the epipolar line can be written as

$$l' = [e']_x x' = [e']_x H_\pi x = Fx \tag{2.6}$$

where $[e']_x$ is the skew-symmetric matrix of e' (Equation 2.7) and H_π is the transformation mapping each x to x'.

$$[a]_x = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{2.7}$$

With the mapping of a point x to an epipolar line l' and how it relates to F now discussed, we can state the following mapping from x to x' using F as

$$x'^T F x = 0 \tag{2.8}$$

If x and x' project to the same point in the two views, then x' lies on the epipolar line $l' = Fx$ and, algebraically, $0 = x'^T l' = x'^T F x$. The mapping from x to x' is shown in Figure 2.8. In addition to giving us faster a way to search for correspondences in other views, this relation also allows us to generate the fundamental matrix using only point correspondences. See [1] for the more information regarding the estimation of the fundamental matrix as we will be using another tool to calculate it, the essential matrix, for this thesis.

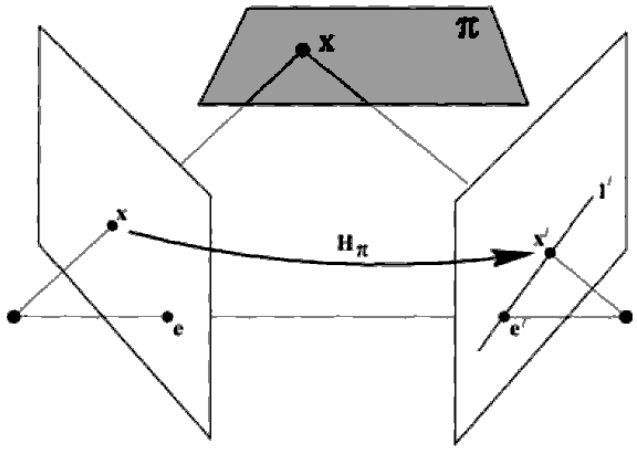


Figure 2.8: View transformation visualization using F from [1]

2.1.1.5 The Essential Matrix

The essential matrix is a special case of the fundamental matrix where we know the exact intrinsic and extrinsic parameters of the camera system. While the fundamental matrix is in terms of pixel coordinates, the essential matrix is in terms of normalized image coordinates. If a camera matrix P is equal to $K[R|t]$ and a point in three dimensional space is X, then that point projected onto its image plane is equal to PX. For any given x, can calculate the normalized coordinate $\hat{x}$:

$$\hat{x} = K^{-1}x = [R \mid t]\, X \tag{2.9}$$

A camera matrix that has been multiplied by its inverse calibration matrix is called a normalized camera matrix and can be represented as simply $[R|t]$. For a pair of normalized camera matrices $P = [I|0]$ and $P' = [R|t]$, the fundamental matrix that links the two is

$$E = [t]_x R = R\left[R^T t\right]_x \tag{2.10}$$

which is called the essential matrix. Like the fundamental matrix, we can map points from one view to the other using it, provided that we use normalized coordinates.

$$\hat{x}'^T E\hat{x} = 0 \tag{2.11}$$

Comparing Equation 2.9 for relating x and x' using the essential matrix with the Equation 2.11 for relating x and x' using the fundamental matrix, we can relate the essential and fundamental matrices with the calibration matrix K with

$$E = K'^T F K \tag{2.12}$$

Because the input data into our system is computer generated, we can skip the generation of F using correspondences with may not be precise and get an exact computation by calculating the essential matrix from the translation and rotation of the two views and using the calibration matrix to generate F.

2.1.1.6 Triangulation

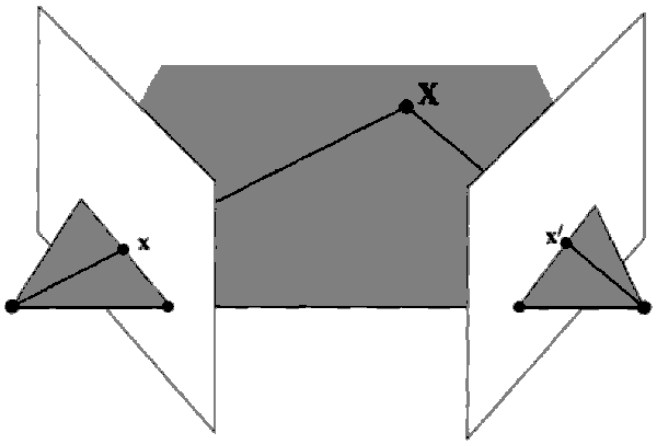


Figure 2.9: Triangulation visualization from [1]

When a correspondence is found between two points in two different views, it is possible to triangulate the points to determine the location of the correspondence in three dimensions. The most common solution is using ray projection and using least squares to find the nearest point that the rays cross and finding the midpoint between them. This technique

is shown in Figure 2.9. The only points that cannot be triangulated in this way are points that lie on the epipoles of the image planes as the rays propagating outwards from the views intersect in multiple points. This is avoidable if the epipoles of the system are outside of the field of view of each camera/projector.

2.1.2 Identifying Codewords

Structured light relies on the ability to match points of the projected image to points on the camera. In early structured light systems, codewords were sequences of white and black bars that represented bit encodings for codewords. If a pixel was black, then white, then black, its code was 010_2. Generically, for a sequence of m images, 2^m possible codewords existed. By identifying a codeword in an image, this method uniquely identifies the location of this feature in the projected image in the dimension perpendicular to the lines. Provided that the camera and projector are not aligned in the same axis as the lines, this method could use the epipolar line that crosses that location in the camera's field of view to identify the feature in the dimension parallel to the lines. Further advancements in position encoding beyond black and white lines will be discussed in Chapter 3 and our specific implementation is discussed in Chapter 4.

One challenge with structured light is how to deal with incorrect decoding of codewords. If lighting conditions aren't just right, the algorithm may perceive one codeword as another. There are solutions for decreasing the probability of incorrect decoding by accounting for lighting abnormalities, but it is also important to ensure that when errors do occur, they can, first, be detected, and then, corrected. In [5], Richard Hamming provides an answer in the form of Hamming distance. The Hamming distance of two keywords is the minimum number of symbols that can be substituted to make one identical to the other. For example, the codewords 100_2 and 111_2 have a Hamming distance of 2 because we would need to change the last two symbols from 0 to 1 in the first codeword in order for it to be misidentified as the second one; there is no shorter way to change from the first symbol to the second or

vice versa. This is visually represented by the binary cube in Figure 2.10.

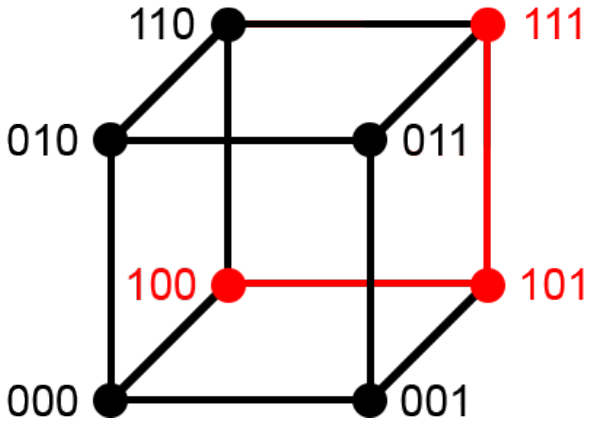


Figure 2.10: A visual representation of Hamming distance

An encoding with a minimum Hamming distance of d across all codewords means that any codeword is differentiated by at least d symbols from any other keyword in the encoding. An encoding is said to be k-errors detecting if the minimum Hamming distance of the encoding is at least $k+1$, though this is not sufficient for correcting the error. In order for an encoding to be k-errors correcting, the minimum Hamming distance between the codewords must be at least $2k+1$. Our earlier example in Figure 2.10 can detect one error, but cannot correct the error. If we replaced 100_2 with 000_2 however, the encoding could then detect up to two errors and correct for one.

2.1.3 de Bruijn Sequences

Early in structured light, the encoding methods used required sequences of images to be projected onto the surface, however many methods exist for encoding codewords onto a surface from a single image. This is sometimes referred to as one-shot encoding and can come in the form of hieroglyphs, grid patterns, colored lines, and more. This paper concerns itself in particular with colored line encodings created using de Bruijn sequences. An n-order de Bruijn sequence with an alphabet of size k is a cyclic, pseudo-random sequence that contains every possible sequence of length n in the alphabet of symbols. By identifying

n symbols in a row (or colors, in this case), a unique codeword can be formed and thus the position can be identified. As shown in Figure 2.11, codewords share encoding information by sharing colored lines between themselves. For example, the codeword n (Violet, Cyan, Orange) could then give way to codeword $n+1$ (Cyan, Orange, Violet) with Cyan and Orange being shared between the two codewords. This can be helpful in determining a codeword even when the algorithm can't quite decode a line but it knows which lines were before and after it. Smaller orders of de Bruijn sequences are easier to identify due to the shorter codewords, but to achieve high resolution scans a large alphabet must be used to avoid repeating the sequence. Larger alphabets lead to a higher probability of an incorrectly decoded codeword. Using larger ordered de Bruijn sequences can decrease the alphabet size while keeping the same resolution. Alternatively, by ignoring select codewords instead of decreasing the alphabet size when the order is increased, the minimum Hamming distance of the encoding can be increased above 1 which increases the correctability of the encoding.

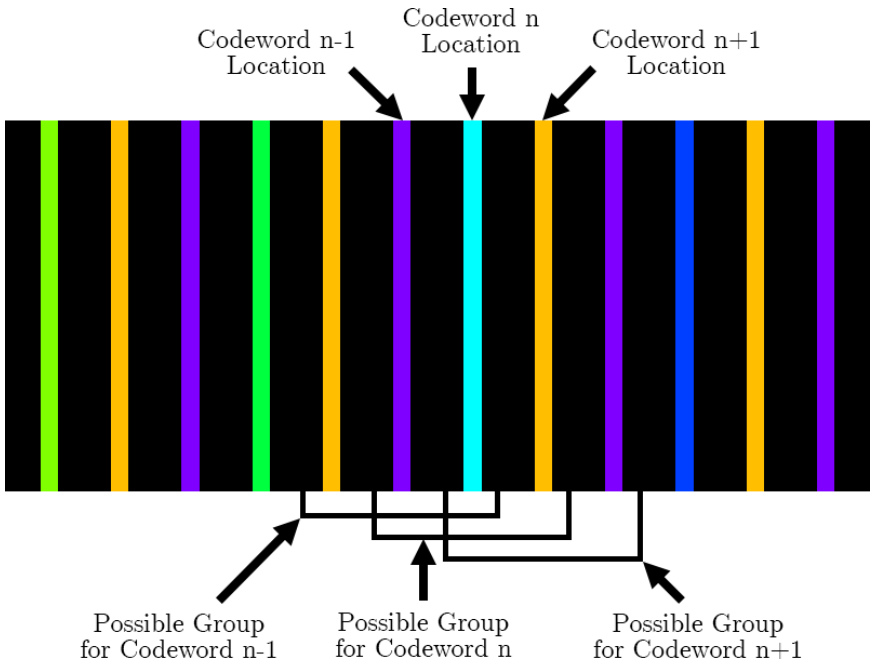


Figure 2.11: Overlapping Lines in a de Bruijn sequence

2.2 Other Computer Vision Algorithms

To accomplish the additions presented in this work, we need to utilize a few other computer vision algorithms outside of the basic matrix operations described in the previous section.

2.2.1 Background Subtraction

Background subtraction is a technique in computer vision for separating an image's foreground from its background provided that the camera is not moving. While many applications of background subtraction are for replacing the background with another scene, it can also be used to detect moving objects on a static background. Implementations of background subtraction can range from a simple difference of pixels from one frame to the next, comparing each pixel value to their historical mean average, and building combinations of gaussian models representing possible values for each pixel. This paper uses the latter of these implementations (referred to as Mixture of Gaussians, or MoG).

In MoG background subtraction, a model is trained on a sequence of images with a certain learning rate so that it might determine the mean values for each pixel and the associated variances as to be resilient to noise. If multi-modal distributions arise, a MoG background subtractor can add new gaussian distributions to the model that are outside of the current model if they persist for long enough to be determined as part of the background. The amount of time that this takes is determined by the learning rate.

Using gaussian distributions in background subtractors was introduced by [6] which can account for camera noise and other sources of pixel variation. However, swaying objects in the view like trees need multiple gaussian distributions per pixel to account for these more abstract variations. The mixture of gaussian model was introduced in [7] and many subsequent updates to the idea since then have been related to faster training methods for the model (as in [8, 9, 10]) or finding a better way to determine the ideal number of modes in the model (as in [11]). Specifically, the work of Zivkovic and van der Heijden in [11, 12] is

related to picking the proper number of modes for each pixel in an online algorithm so that it can efficiently adapt to the scene. Figure 2.12 depicts what a mixture-of-gaussian model looks like for a single pixel in Zivkovic and van der Heijden's works in the RGB color space with modes around (60, 145, 90) with a weight of 0.7 and (215, 45, 165) with a weight of 0.3.

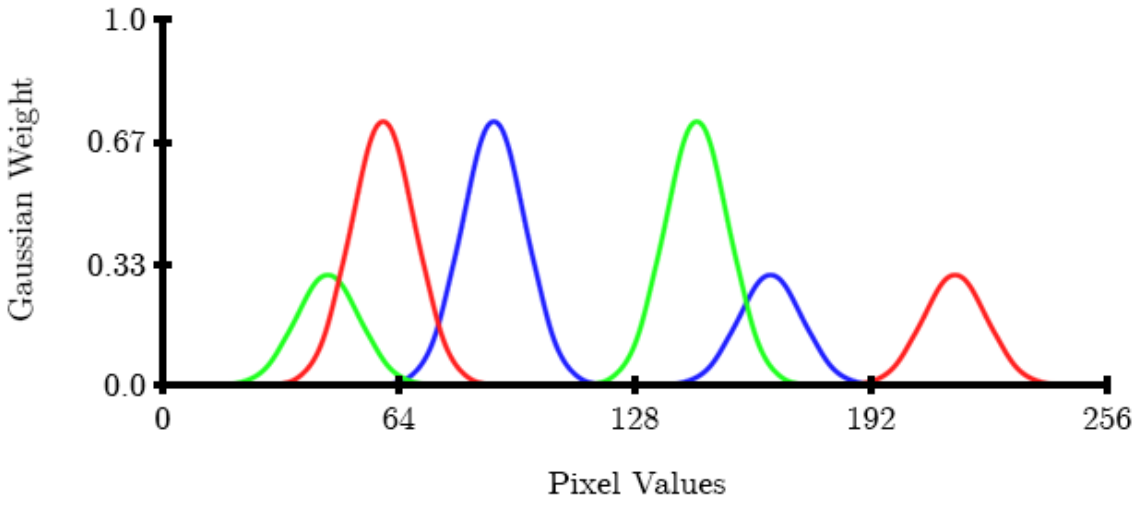


Figure 2.12: An RGB mixture-of-gaussian model with two modes

There are several parameters that must be tuned in a MoG background subtractor, with the first three being the most important:

- Learning Rate: The rate at which new modes are added to the model (this can vary over execution time)
- Variance Threshold: The threshold for which pixels will be counted as either background (described well by the model) or foreground (not described well by the model)
- Maximum Number of Mixtures: The maximum number of gaussian distributions that we will allow in the model (sometimes a memory constraint, sometimes a logical constraint)

- Background Ratio: The threshold for when a component becomes significant enough to be added to the model as its own mode
- Generation Threshold: The threshold that determines how far another value must be to generate another mode in the model
- Initial Variance: The seeded variance for each new mode
- Minimum Variance: The minimum variance that a mode can achieve
- Maximum Variance: The maximum variance that a mode can achieve
- Complexity Reduction Prior: The number of samples needed to have confidence that a mode actually exists

This method of background subtraction can also be used to identify shadows in an image, though that feature is not utilized in this thesis. The specific mathematics behind MoG background subtractors can be found in [11, 12].

2.2.2 Morphology

Dilation is a basic morphological operation on a binary image where activated pixels are expanded by a predetermined shape (called a structuring element) [13]. For any arbitrary binary image A and a structuring element B, the formal definition is

$$A \oplus B = \bigcup_{b \in B} A_b \tag{2.13}$$

Dilation can be accomplished by taking the anchor of the structuring element (usually the center) and positioning it at every previously activated pixel in an image and then activating all pixels that the structuring element now covers, as demonstrated in Figure 2.13. Dilation is very helpful for filling in small holes of a mask as well as increasing a mask's size to cover the entire edge of an object that is being masked.

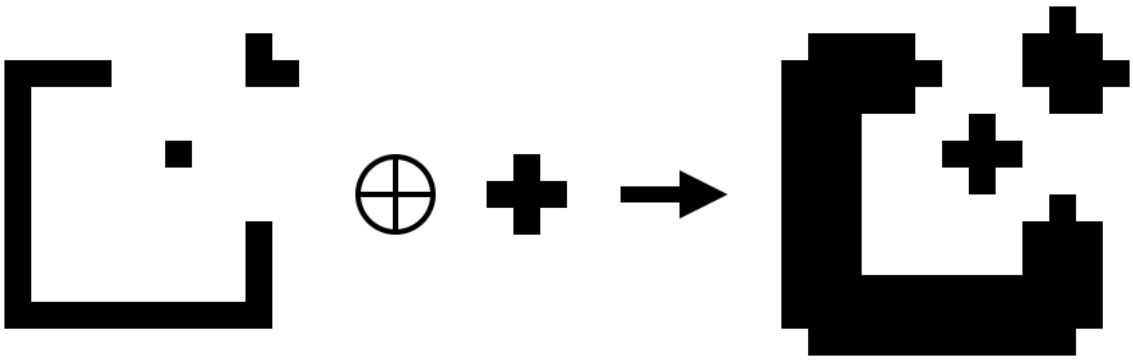


Figure 2.13: Dilation of an arbitrary design with a 3x3 cross (inverted)

2.2.3 Nearest Neighbor Search

Nearest neighbor search is the act of finding the closest point to any arbitrary point in some space. More generally, k-nearest neighbor search finds the k nearest points to any given point in some space see Figure 2.14. Nearest neighbor search is helpful when dealing with point clouds to find outliers. This problem becomes increasingly complex as the dimensionality of the search space increases. Many different algorithms exist for finding the nearest neighbors to a point and they can be grouped into two major categories: exact and approximate.

Exact methods generally take longer to find a solution but come with a guarantee that the solution is correct. The nave method for nearest neighbor search that entails calculating the distance to every other point in the search space and finding the minimum. Such an approach has a time complexity of $O(dN)$ where d is the dimensionality of the space and N is the number of points.

Another exact method is to use space-partitioning and construct a special type of binary search tree called a $k-d$ tree (where k is the number of points and d is the dimensionality of the space) linking close points for an average time complexity of $O(log(N))$. In a $k-d$ tree, each non-leaf node in the tree lies on a hyperplane (a plane in 3D and a line in 2D) that divides the remaining space. Each node has a dimensional axis associated with it

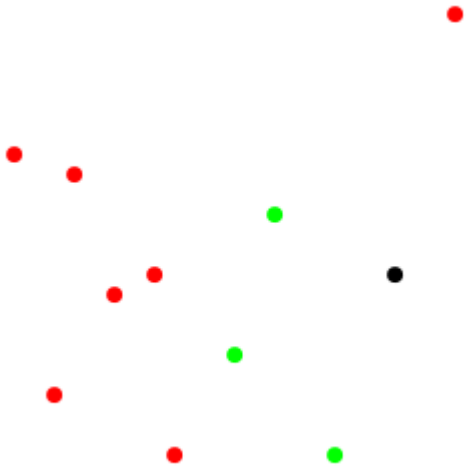


Figure 2.14: A 2D example of 3-nearest neighbor around the black dot

that tells the tree to generate the hyperplane perpendicular to that axis. An example two dimensional $k-d$ tree is shown in Figure 2.15 in its graph representation and its binary tree representation.

The canonical approach to building a $k-d$ tree is to cycle through the dimensions that determine the direction of the hyperplane as one moves down the tree and to insert points with the median of the points as the tree's root to better ensure a balanced tree. Nodes are added to the tree in the same way as any other binary search tree: traverse to where the node should exist, insert it, and set the previous node at that position as the child of the new node on the proper side. Removing a node requires either rebuilding the sub-tree starting at the node removed, or to find a child of the node to be removed, replace it with such a child, and then recursively remove the replacement node from the sub-tree. Performing nearest neighbor search on a $k-d$ tree is described in [16].

Approximate methods give plausible, but potentially not perfect, solutions to the nearest neighbor search problem. In many cases, an approximate solution is good enough and largely doesn't affect the quality of a program. A popular technique for this method of nearest neighbor search is using locality-sensitive hashing, or LSH. LSH uses a hashing

function that groups points in a similar space into similar buckets. Unlike cryptographic and checksum hashing functions that are designed to reduce the similarity of nearby inputs, locality-sensitive hashing by making nearby inputs collide with each other. Sometimes a family of hash functions are used to work on a wide variety of input data [17].

Performing approximate nearest neighbor search on LSH-binned inputs is as easy as searching the bin of the input node across the family of hash functions. If multiple hash functions are used, the bins from each hash function are searched separately. Though in this thesis we are only interested in LSH as it applies to nearest neighbor search, LSH is used as a dimensionality-reduction tool in a very wide array of applications such as DNA analysis [18], reverse image search [19], audio/video identification [20], and more.

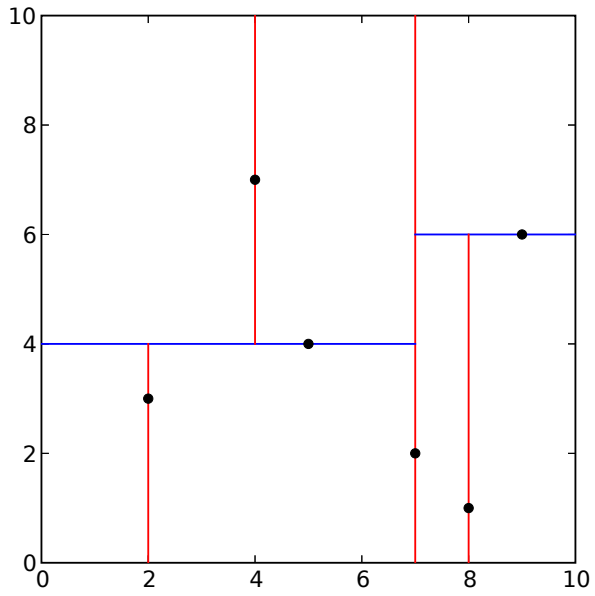
2.3 CUDA

CUDA is Nvidia's parallel computing platform introduced in 2007 as a way for programmers to more easily write code that can be accelerated on GPUs. This method of computation is called GPGPU or general-purpose computing on graphics processing units. In the past 10 years, many improvements have been added to CUDA such as support for half-precision and quadruple-precision floating point operations, increased maximum widths of arrays for computation, and increasing the size of data types that can do operations atomically. Today, CUDA is being applied to training deep neural networks, fluid dynamics simulations, and, as I apply it in this thesis, computer vision computation.

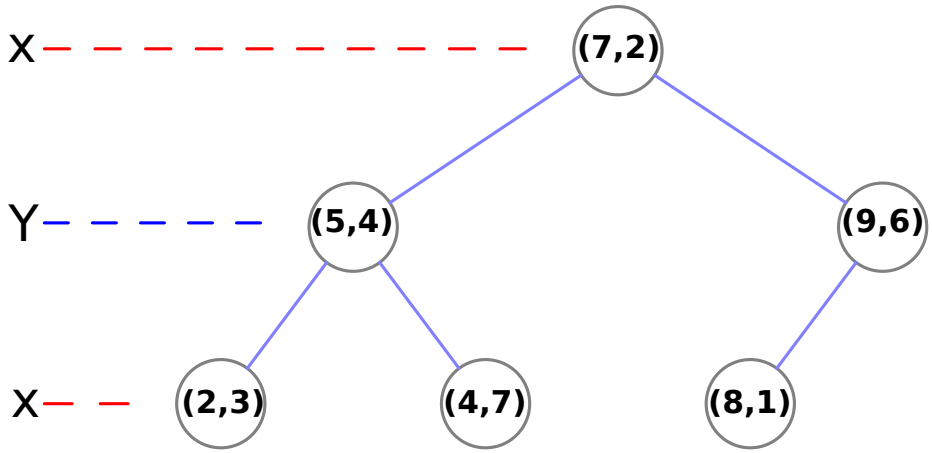
The CUDA model works by writing CUDA code called a kernel and compiling it with the nvcc compiler [21, 22]. This kernel can be called from C, C++, and Fortran code using a series of CUDA API calls. The general sequence of calls is as follows: setting up the device (GPU), allocating and copying data to the device, running the kernel, then copying data back to the host. When a kernel is called, the programmer must specify the number of threads to run in a block and then how many blocks to run. Each thread generally runs independently of the others (though not always) and each kernel can identify itself

querying its thread and block index. Using these indices, a thread can determine what part of memory it should be acting upon and then execute its instructions. Blocks are groupings of threads and all threads within a block must be running the same kernel.

CUDA comes with many benefits over standard computing including, but not limited to fast on-board memory operations, higher number of instructions per cycle (even more-so with half-precision floating point numbers), and fast, parallel libraries for dealing with linear algebra, sparse matrices, signal processing. However, no technology is without its downsides. Large amounts of host-to-device or device-to-host memory transfers will significantly bottleneck any application. Additionally, while branch instructions are allowed in kernels, performance can suffer if not all of the threads take the same branch.



(a) 2D Plane representation by [14]



(b) The binary tree representation by [15]

Figure 2.15: Visualizations of a k-2 tree built from the set { (2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2) }

CHAPTER 3. RELATED WORK

In 2004, Salvi et al. [23] analyzed several implementations of structured light up to that point and created the following high-level classifications: time multiplexed, spatial neighborhood, and direct coding. In addition, in 2011 Geng published a tutorial [24] that covers the basics of structured light that covers many of the popular implementations. This chapter discusses some advancements in the first two of those classifications over the last 30+ years as well as additions that have been made since [23] that have inspired this work.

3.1 Time Multiplexing

Time multiplexing was the initial foray into structured light, though due to its necessity to project sequences of patterns onto a surface it is always restricted to static scenes. In 1981, Posdamer and Altschuler [3] projected a temporal sequence of m binary patterns to encode codewords of length $2m$ onto a surface. Their setup used a laser reflection system to draw matrix dots onto a surface. Dark pixels corresponded to the bit 0 and bright pixels corresponded to the bit 1. By analyzing the camera image and looking at the pixel values at the same locations for each image, they found that you can read back the codeword and determine the depth of the pixel. Inokuchi et al. [25] replaced Posdamer and Altschuler's incremental binary sequence with Gray codes which give neighboring codes a Hamming distance of one to reduce the impact of errors (i.e a traditional 2-bit binary sequence goes 00 $\rightarrow$ 01 $\rightarrow$ 10 $\rightarrow$ 11, but a Gray code sequence goes 00 $\rightarrow$ 01 $\rightarrow$ 11 $\rightarrow$ 10). Mimou et al. [26] increased the number of patterns projected in order to achieve an encoding with a Hamming distance of three to allow for better error detection and

error correction. Trobina [27] showed that the location of the lines was crucial to accurate structured light surface reconstruction and introduced two methods of gaining sub-pixel accuracy: either by finding the second derivative zero crossings of the image where the lines go from white to black or by projecting both the positive and negative patterns onto the surface and comparing the two to find the edges of the stripes. Valkenburg and McIvor [28] also worked on finding the location of the stripes in their encoding to increase the accuracy of the system by fitting third degree polynomials to the intensity of 17 x 17 pixel regions using least squares to find the crossings. They also attempted this with sinusoidal waves which improved the results further. Skocaj and Leonardis [29] tackled the difficult task of scanning scenes with varying reflective properties. By projecting patterns at different light intensities, they built a radiance map of the scene and can determine which parts need different amounts of illumination to be properly decoded later. They found that while the minimum number of illumination intensities could be as low as two in a scene with varying reflective properties, though increasing that number led to much better results. Caspi et al. [30] worked on expanding Gray codes to multiple colors instead of just white and black. This addition reduced the number of patterns that needed to be projected but still came with the Hamming distance of 1 that traditional binary Gray codes had. Recently, Young et al. [31] used not multiple patterns, but multiple camera positions to encode time multiplexed data onto a scene. By moving the location of the camera and comparing the values along the epipolar lines, they found that they could achieve the same results as other time multiplexing schemes but while also handling the occlusion of subject features.

3.2 Spatial Neighborhood

Two major methods of encoding codewords in single frame patterns are using de Bruijn sequences (as this work will do) and using M-arrays. Work has progressed on each in parallel in the past years, though there is some cross-over when de Bruijn sequences are used to generate M-arrays.

3.2.1 Patterns based on de Bruijn Sequences

Hugli and Maitre [32] based their work a previous paper by Boyer and Kak [33] which also projected colored, horizontal lines, but they encoding pattern with de Bruijn sequences. However, while Boyer and Kak's system separated the colored lines with black lines, Hugli and Maitre did not which meant that they could not project the same color consecutively and thus could not achieve the maximum size that their color alphabet would generally allow. Monks and Carter [34] used a similar sequence, though added the black lines back in and used the HSV color space with a six color alphabet which gave them access to a larger list of codewords. Vuylsteke and Oosterlinck [35] introduced a binary coded pattern similar to a checkerboard that is based on de Bruijn sequences. Salvi et al. 1998 [36] combined vertical and horizontal lines. The intersections of the grid were the points to be tracked, and neighbors could easily be found just by tracing the lines. Three colors were used for the vertical lines and three for the horizontal. Zhang et al. 2002 [37] used vertical colored lines in a de Bruijn sequence with no adjacent repeating colors, but slid the pattern like a time-multiplexed system to account for lighting differences. Zhang et al. improved upon this method in 2003 [38] by adding support for moving and deforming objects. Ulusoy et al. [39] took the concept of grid based detection similar to [Salvi et al. 1998] but instead of relying on particular colors to encode the de Bruijn sequences, they encoded the sequence using the spacing of the lines and thus only uses two colors.

3.2.2 Patterns based on M-arrays

A perfect map is a matrix of size r x v where every submatrix of size n x m appears exactly once. Within a perfect map, if you know the elements within a window, you can uniquely determine your location within the larger matrix. If your perfect map is created with elements of an alphabet of size k, it is considered an M-array if and only if it contains all possible submatrices of size n x m except for the submatrix of all 0s. With these properties, M-arrays provide similar advantageous properties to de Bruijn sequences but in

two dimensions.

The use of binary M-arrays in structured light was proposed by Morita et al. in 1988 [40]. Their system projected dots that represented points that were going to be tracked so that the camera could find them and then projected the actual M-array representation of the encoding. Because of the two images that needed to be projected, it was limited to static scenes. Griffin et al. [4] used perfect submaps (perfect maps where not all possible windows appear) where each symbol could be identified by itself and its four-connected neighbors. Their encoding was generated using de Bruijn sequences. They also experimented with using colors to represent the alphabet as well as unique symbols, the latter of which proved more robust to colored surfaces. Morano et al. [41] also used perfect submaps but increased the Hamming distance of the encoding in order to allow for error correction. They also noted that any M-array based system using N colors can be converted into a binary, time multiplexed M-array system with $log_2(N) + 1$ patterns.

3.3 Multiple Cameras and Multiple Projectors

In recent years, there has been research into using multiple cameras and projectors in order to capture a full 360-degree subject using structured light. Furukawa et al. [42] used six cameras and six projectors in order to fully scan the subject. Each projector projects a set of parallel lines of a unique color set in order for each camera to differentiate between them. In 2011, Furukawa et al. [43] improve this method and scans a human body in action.

3.4 Miscellaneous Related Works

Fechteler and Eisert [44] used k-means clustering to group similarly colored pixels together to decode the sequence instead of thresholding certain colors as many other works do. This allows for this algorithm to work without being in a dark environment as only eight unique groupings need to exist in the clustering. This work also projects a white light image onto the subject that can be used after reconstruction to map a texture to the mesh.

Zhang et al. [45] used motion detection to switch between time multiplexing (TM) and spatial neighborhood (SN) schemes in order to achieve the high accuracy of time multiplexed reconstruction and low latency of spatial neighborhood reconstruction all in one system. They supported globally changing from TM to SN if motion was detection as well as only changing a region of interest to SN if motion was only detected in that region.

CHAPTER 4. THE ALGORITHM

The goal of the algorithm described in this part of the paper is to reconstruct 3D surfaces while also answering the following questions:

- Do we need to re-detect features that haven't moved?
- Where do we search for points that have moved?
- How do we detect motion?
- How do we decode the depth of moving objects?
- How do we stop incorrect decoding from propagating forward?

By combining background subtractor and a modified structured light system, this algorithm accelerates surface reconstruction by limiting the search of new points to the part of the frame that has moved and by not searching for points that haven't moved. The algorithm is implemented as a pipeline of data and a flowchart is provided in Figure 4.1. The rest of the chapter describes the algorithm and its components in more detail.

4.1 Frame Acquisition

Instead of just projecting a pattern onto a surface like a standard one-shot structured light implementation, this system alternates between projecting the pattern (Figure 4.2a) and white light (Figure 4.2b). A colorized de Bruijn sequence of order 3 and with 8 symbols, culminating in 512 unique codewords, was chosen as the encoding pattern. Each stripe is

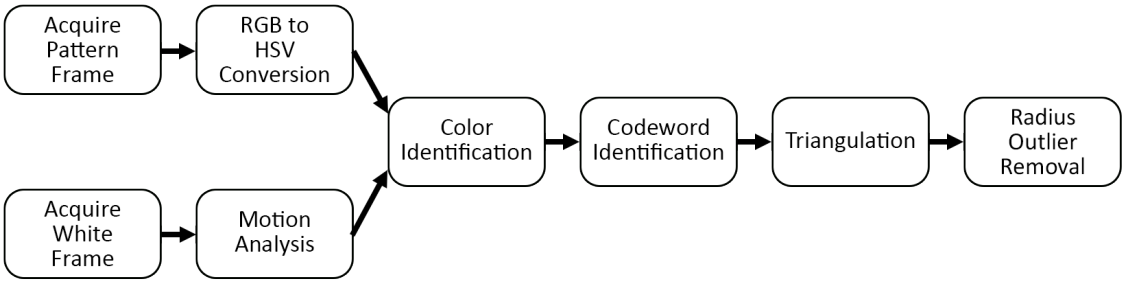


Figure 4.1: Algorithm Progression

separated in the encoding pattern by 4 pixels so we need at least 480 codewords in our pattern. The 3-8 de Bruijn sequence was chosen because it provides a good balance between number of symbols in the alphabet and only a few lines are needed to fully decode the codeword. Other alternatives were a 2-22 sequence which has too many symbols to be robustly decoded despite only needing two lines to work and a 4-5 sequence which only drops the alphabet size by 3 but requires an additional line to decode the codeword which could lead to more incorrect decodings. More complex encoding patterns with higher minimum Hamming distances are more robust, though the purpose of this work is to highlight the potential speed improvements of adding motion analysis to structured light, so a simple encoding scheme was chosen. While time multiplexed encoding schemes are even simpler, this system is designed to support moving objects which are not supported by time multiplexed encodings. The projected pattern is 1920x1440 and contains striped lines every four pixels with black space in between. We track a point in the encoding pattern at every 4 pixels in the vertical and horizontal directions and each one has its own unique epipolar line. The de Bruijn alphabet was color coded according to the HSV color space (the reasoning for which is explained in section 4.4 of this thesis). The white light frame is projected to illuminate the surface in order to get a clear picture of what is in view. This image is sent along to the motion analysis part of the algorithm to determine what parts of the image have moved since the previous white light frame was last captured. As will be discussed in Chapter 7,

including this frame in the system opens the door for even more features beyond motion analysis.

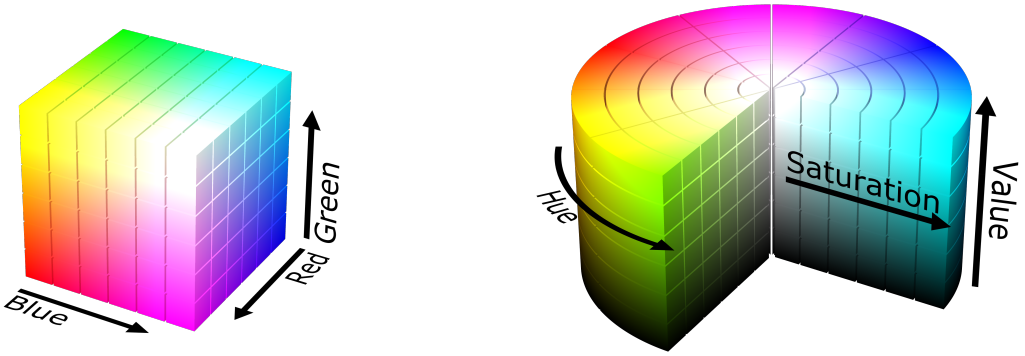
(a) Pattern Projection (b) White Light Projection

Figure 4.2: Images captured from the alternating projections in this system

4.2 RGB to HSV Conversion

HSV, or hue, saturation, and value, is an alternative color space to RGB. It stores the entirety of the color in the hue space as a degree on a circle, the saturation of that color from neutral all the way up to its full vibrance, and its value as a key from black all the way up to full brightness. It is common in computer vision algorithms to use HSV to parse color information instead of RGB when looking for objects of a certain color range. Figure 4.3 visually represents the RGB and HSV color spaces and shows why it is easier to discern different colors as well as dark pixels in the latter.

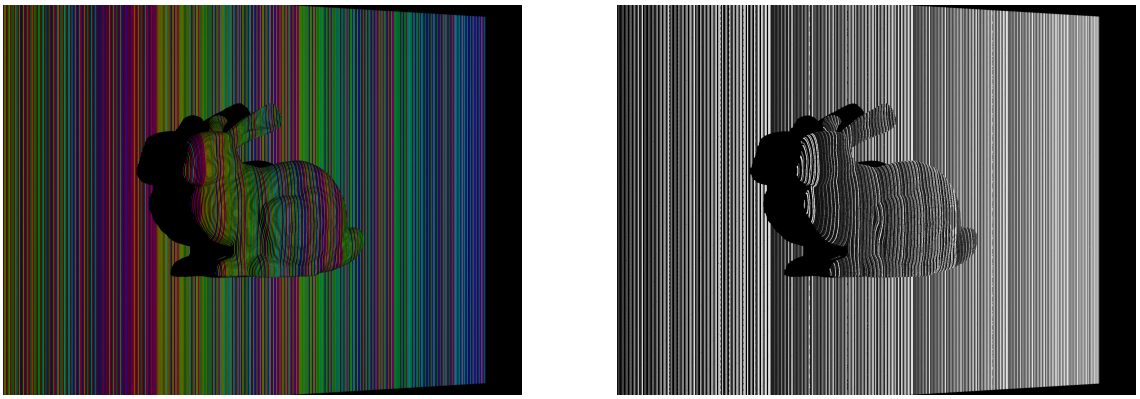
Converting from RGB to HSV is an operation that requires several branches and computing on individual values which we always try to avoid when computing on GPUs. Unfortunately, there is not a good way to avoid this during this conversion operation so we must incur the penalty when computing the HSV image on the GPU. Converting our pattern image from RGB (Figure 4.4a) to HSV (Figure 4.4b) allows us to parse our encoding scheme more easily. Because our decoding only needs the hue and value planes, we can remove the



(a) RGB Color Space by [46] (b) HSV Color Space by [47]

Figure 4.3: Visualization of the color spaces used in this system

saturation plane only copy the other two planes when running this stage. The multi-core CPU implementation divides work in this operation by rows while the GPU divides work by pixels.



(a) Pattern Projection (b) Hue Channel

Figure 4.4: An RGB frame and its corresponding Hue channel

4.3 Motion Analysis

Motion analysis is a helpful tool for determining if anything in your view isn't where it was in the previous frame and, if using an algorithm like optical flow, determining where it is in the current frame. In this thesis, we only care that an object has moved, not where it

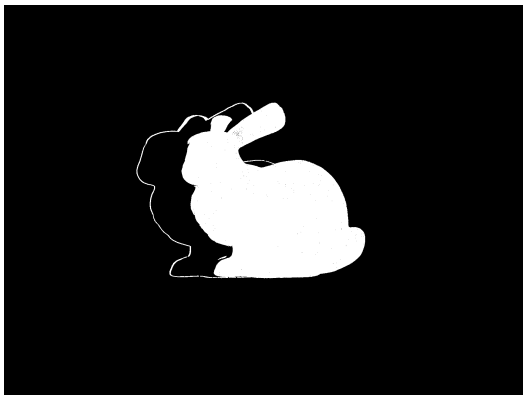
has moved to, so we use a gaussian background subtractor (MoG with only a single mode) to determine motion.

Motion analysis is run on our white light-illuminated frame because it provides the best picture of what our subject looks like at any time. We do not run motion detection on the pattern image because much of the subject is not illuminated due to the black lines in between the de Bruijn lines. Motion in that space will not be detected because it is not illuminated.

This background subtractor only uses the previous frame to determine if the new pixel is within the same gaussian curve because we want to know immediately if we need to search for a new point at that location. A longer history/slow training rate could be added if having the most up to date lines was not a concern to the user. We also use a variance threshold of 0.5 to detect noticeable but not minute changes to the subject, though for a very noisy camera or scanning environment this may need to be increased. Because the MoG background subtractor is a per pixel mode, it may be unwise to add more gaussian modes even if the location of the codeword was stored along with the gaussian information to allow for recall at a later match as similar looking pixels that are temporally distant may not refer to the same depth. All other parameters in the background subtractor were left to their OpenCV defaults [48]. The input image and the output motion mask are shown in Figures 4.5a and 4.5b. Like the previous stage, the multi-core CPU implementation divides work in this operation by rows while the GPU divides work by pixels.

Because the background subtraction as a motion detector is not perfect, we need to clean up the mask slightly as well as account for any minor movement near the edge of the subject that was not captured in the mask. To do this, we dilate the mask with a 11x11 cross two times that fills in the holes in our mask as well as increases our potential search space for new codewords. Without this expansion, we may not be able to get a large enough window to see the entirety of the codeword in the next pattern frame. The dilated motion mask and our new search space for codewords is shown in Figure 4.5c.

(a) White Light Projection



(b) Motion Mask

(c) Dilated Motion Mask

Figure 4.5: Various structured light encoding patterns

4.4 Color Identification

The choice was made to separate the colors representing the alphabet in the de Bruijn sequence by hue. As discussed earlier, the HSV color space makes it much easier to quickly discern color than the RGB color space and also makes it easy to detect the dark lines separating the colored lines. Separating the colors in the Lab color space was also considered which is sometimes used in data visualizations to increase the discernibility for human readers, but this carries little benefit for computer systems.

For each pixel in the motion mask generated by the motion detection stage, we attempt to identify the color encoded at that pixel. In the first cycle, this operation is done on all pixels. In subsequent cycles, by only identifying colors of pixels that contain motion, there

should be a direction relationship between the amount of motion in the frame and the speed of this stage. Figure 4.6 illustrates the decoding method: we add 11 to the hue, mod by 256, and divide by 22.5 in order to categorize it into one of the 8 components that represent our de Bruijn alphabet. This assumes that the hue is stored as a number in [0, 179). While this is half of the normal range of hue in [0, 360), it is necessary to fit the hue within one byte. This operation has the effect of rounding all hues to their nearest component without the need for branch statements. If the value part of the pixel in the HSV color space is less than 70, it is not put into one of the de Bruijn bins and is instead categorized as a black pixel with a code of 255. Black pixels can either be one of the black lines in between our colored lines or an occlusion from the subject blocking light from falling elsewhere on itself. In either case, we don't want to make any false guesses about what the color could be that might negatively affect decoding in the next stage. Once again, the multi-core CPU implementation divides work in this operation by rows while the GPU divides work by pixels.

Figure 4.6: de Bruijn Sequence Color Decoding Scheme

4.5 Codeword Identification

Once the colors have been identified, we have everything we need to start identifying codewords in the frame. For each codeword, we start by examining the pixel location of that it was previously identified at in the motion mask (provided that we have a valid known location for the pixel already). If no motion was detected, we skip codeword identification and put the 3D location from the previous cycle into our point cloud. The rationale is that if the pixel values were close enough to not register as motion, our codeword identification would probably have provided the same (or at least a very similar) result. Note that because the first frame's motion mask is all white, this is valid on the first run through this stage as well.

In the event that motion was detected at that keyword's previous location or we don't have a previous location stored for the codeword, we enter the process of decoding image to find the codeword's pattern. We utilize the motion mask in this step as well by skipping pixels in our search space that don't contain motion. In the case where a known codeword has moved, the codeword must reside in the motion mask if it still exists in the frame at all as the motion mask covers the previous and new locations of moving pixels. In the case where a previous location for the codeword was not known, the non-moving points don't need to be searched because we have already searched that space in previous frames and were unable to find the point and nothing is gained by searching the space again.

Recall that every codeword has an epipolar line associated with it due to our camera system being calibrated, so our search space for the codeword is the group of pixels that fall on the epipolar line associated with the codeword. Our process for finding the codeword consists of searching along the epipolar line from left to right and reading the decoded pixel values. For every non-255 and non-consecutive value that we read, we store its color and location in a FIFO buffer that we can match against our codeword that we're searching for. When we read 255, we clear the non-consecutive restriction as that means we've identified a black bar separating colors. If we read eight 255s in a row, then there's too much of a spatial

gap to rely on the previous color identifications in our FIFO to give a reliable decoding, so the FIFO is cleared. After each addition to the FIFO, we compare its contents against the codeword that we're searching for to see if we get a match. Our de Bruijn sequence has order 3, so we can identify any three neighboring lines containing our target line to uniquely identify the codeword in the frame. Figure 4.7 illustrates that for a codeword in our encoding pattern, we can identify the two lines leading up to the target, the two lines surrounding the target, or the two lines following the target and be able to decode it successfully. In each case, we store the location of the target line and discontinue the search. If none of the cases match, we continue searching along the line to find a match. Generically, for a de Bruijn sequence of order n, there are $2n-1$ different windows that can be used to identify a codeword, though as n increases, the less resilient the pattern is to discontinuities in the subject due to the larger number of lines needed for identification. In both the multi-core CPU implementation and the GPU implementation of this stage, every epipolar line gets a thread.

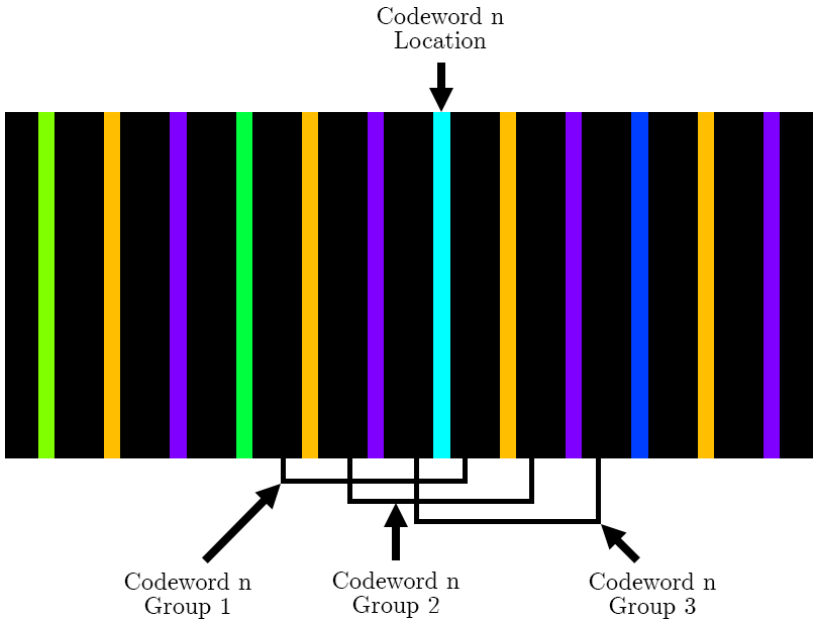


Figure 4.7: Acceptable Codeword Identification Sequences

If the codeword is found, we record its location and mark it as needing triangulation for

the next stage. If the codeword is not found, then it is either not present in the image or occluded by the subject's geometry. In this case, if we have stored a previous location for it we need to remove it from the point cloud so that it's no longer rendered and clear its stored codeword location so that we don't use that information to make assumptions about its movement or lack thereof in the future.

4.6 Triangulation

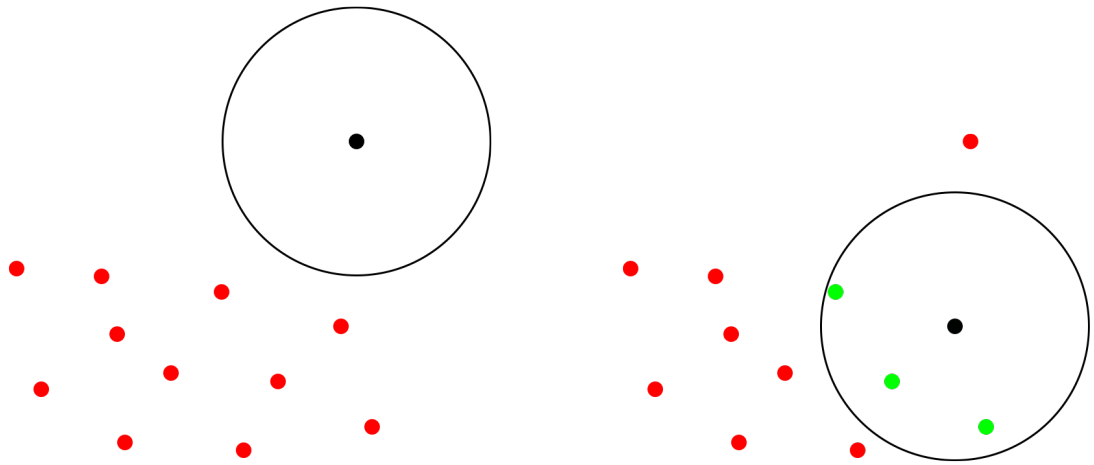
In this work, we use a close approximation heuristic of least squares triangulation (due to issues getting OpenCV's triangulation library to work) described in section 2.1.1.6 and [1] to find 3D positions of our 2D codeword locations. Using the result of the previous stage, we trace a ray out of the camera's center to codeword's location and also trace a ray out from the projector's center to where it projected that codeword and find the midpoint between the closest intersection of the two rays. We avoid the case where this is insolvable by the points overlapping the epipoles by ensuring that the two devices are not pointed towards each other.

We do this operation for any new points that were detected or any points were already known to exist but have moved. We do not triangulate for any points that were skipped or not found in the identification step. In the case where every point in the frame has moved, we do not incur any extra work that we would not have had to do otherwise in another structured light system. However, the less motion that is present in the view, the more time we save by not computing the location. In the edge case where no motion is present from one frame to the next, no calculations are needed in this step.

4.7 Radius Outlier Removal

When building a point cloud with structured light, particularly those methods that don't have high error correctability, it is possible to misidentify a codeword due to an obscured pattern or steep gradient. This is particularly dangerous in this work because the entire list

of codewords is not searched for at every cycle and an incorrect decoding could propagate forward until that point has motion again. We limit the damage of this by introducing Radius Outlier Removal to the system which is a method of identifying outliers based on their proximity to other points. It uses fixed radius k-nearest neighbor search in order to find the number of points within a given radius and if the count doesn't pass a given threshold, it marks that point for removal from the point cloud. Though our implementation is utilized in three dimensions, Figure 4.7 showcases the method in two dimensions for some radius and a count threshold of 3 by showing a point that would be eliminated (Figure 4.8a) and one that would be kept (Figure 4.8b).



(a) Point Removed (b) Point Retained

Figure 4.8: Examples of Radius Outlier Removal

The types of misidentification that we'd like to avoid are ones where the distance between the correct code and the incorrect one is great which leads to the triangulated point being far off from the truth. Because this method works by finding tight groupings of points, it works well for these types of errors. Errors where we are only off by one codeword are less of a concern since it still places the triangulated point into a region close to the truth which this method would be unable to detect.

In this work, the radius is 5 mm and the count threshold is 12. This search includes all previously identified points, not just new ones identified in the last cycle as minorly moving parts of the subject alone may not be enough to pass the threshold requirement but those points in tandem with the rest of the subject should. If a point is detected as an outlier, it is not only removed from the cloud, but its previously computed 3D point is added back to the point cloud and we mark it for forced search in the next Codeword Identification phase. If force search is marked, the point will ignore the motion mask and search its entire epipolar line for a match. This stops us from potentially missing the valid location of the codeword due to the location only existing in the previous cycle's motion mask.

We utilize the Point Cloud Library (PCL) [49] and the Fast Library for Approximate Nearest Neighbors (FLANN) [50] in the CPU implementation which utilizes k-d trees to assist in the search. The specifics of the algorithms used in FLANN are detailed in [51]. While we were unable complete our GPU implementation of fixed-radius nearest neighbor search (and thus had to use the FLANN implementation in practice), we would have based it off of the works of [2, 52] which breaks the scanning space up into a 3D spatial grid and bins each point into one of them (with a limited number of points per cell (visualized in Figure 4.9 where gray boxes are the search space). This approach was initially designed for fluid particle physics simulations, but works just as well for radius outlier removal. When searching for nearest neighbor, each point only needs to search its adjacent cells. While this approach takes more memory than a $k-d$ tree, its consistent memory layout lends itself well to parallel architectures like a GPU.

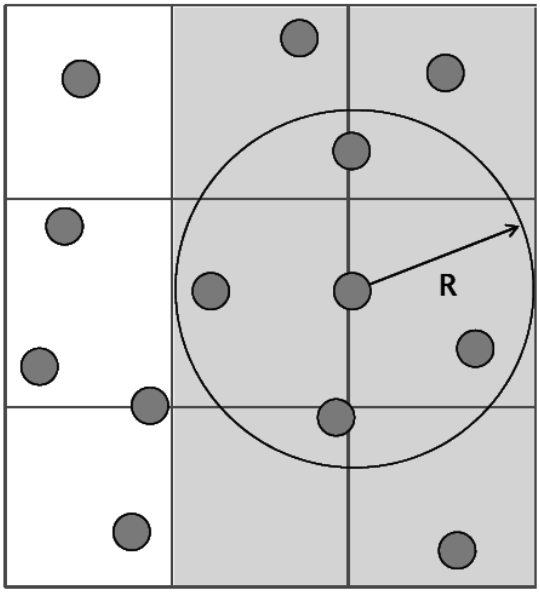
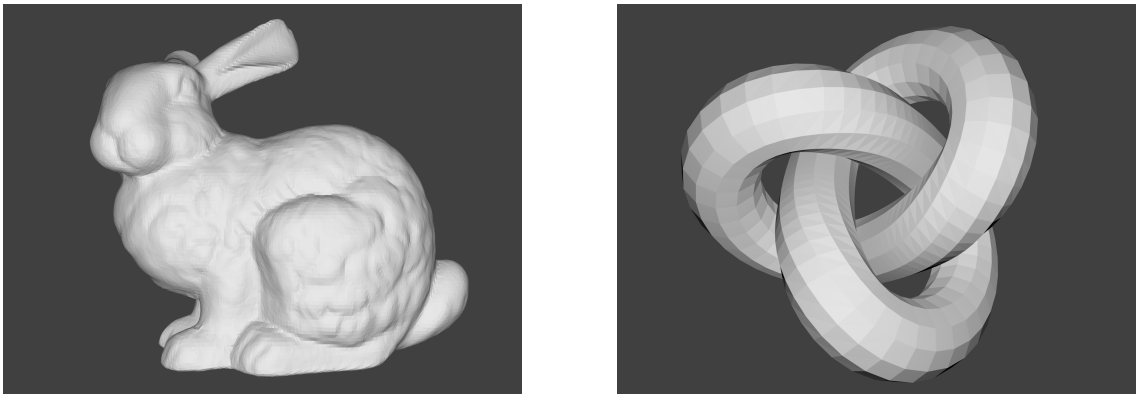


Figure 4.9: Fixed-radius nearest neighbor search in two dimensions from [2]

CHAPTER 5. EVALUATION METHODOLOGY

This system will be evaluated with the Visual Studio Profiler for the CPU execution time and the CUDA Nsight Profiler for GPU execution time. The amount of time spent running each stage of the algorithm on each platform and copying data between the CPU and GPU will be measured. Due to the lack of a physical setup, the input videos were generated using a Python script and the 3D modelling software Blender. The camera was placed at $(0, 0, 0)$ with a rotation of $(\pi/2, 0, 0)$ which faces forward. The projector was placed just to the right of at $(0.1, 0, 0)$ which is 10 centimeters to the right of the camera and angled inwards towards the subject with a rotation of $(\pi/2, 0, \pi/18)$. The subject was placed in view of both the camera and projector and given a wooden texture. The background surface was a plane positioned perpendicular and 75 centimeters away from the camera. It was assigned a concrete texture. The two subjects scanned were the Stanford Bunny (Figure 5.1a) and a simple knot that comes with the MeshLab software (Figure 5.1b).



(a) The Stanford Bunny [53]

(b) The Knot [54]

Figure 5.1: Two models used for evaluation

The subjects were recorded while rotating about all three axes (clockwise and counter-clockwise) and translating across all three axes (forwards and backwards) to determine how this methodology stood up to different types of movement. This provided 24 different input videos being created for testing. Each video is 5 seconds long with 60 frames per second. The videos were rendered in the AVI file format with full 8-bit RGB color.

All evaluation was done on a computer with the specifications in Table 5.1. The relevant software library versions that were used are listed in Table 5.2.

Table 5.1: Computer Specifications

Component	**Product**
Operating System	Windows 10 Pro, Build 16299
CPU	Intel i7-8700K
RAM	16 GB DDR4-3000
GPU	Nvidia GTX 1070

Table 5.2: Computer Specifications

Software	**Version**
CUDA Toolkit	7.5
OpenCV	3.1.0
PCL	1.8.0
VTK	7.0.0
FLANN	1.9.1

CHAPTER 6. RESULTS

This section will describe the accuracy and speed of our structured light system. The speed-ups reported in this chapter are specific to the input videos described in Chapter 5; real-world speed-ups will depend on the amount of motion in those input videos. For our evaluated videos, Figure 6.1 shows the execution time of all stages of the algorithm except for radius outlier removal on a single CPU core, twelve CPU cores, and a GPU. Radius outlier removal was not included in these plots as we could not get FLANN's multithreaded functionality to work when running k-nearest neighbor search (it's consistently around 150 ms/cycle) and it is easier to show improvements of parallelizing the algorithm.

Table 6.1 contains the full results of the system performance on a single CPU core. We can see that by adding motion analysis to structured light surface reconstruction with outlier removal, performance increases by 21.2%. When motion analysis is not used, outlier removal is less important due to the lack of error propagation. Without taking outlier removal into account, the system performance increases by 52.2%. RGB to HSV conversion can occur in parallel with the motion detection, so the motion mask is not used to accelerate that stage. We see quite significant accelerations in the Color Identification stage due to it being a per pixel operation that can be directly affected by reducing the number of pixels to do work on via the motion mask. We also see those gains propagate into the Codeword Identification and Triangulation stages by limiting the amount of codewords we need to search for and, if found, triangulate.

Table 6.2 contains the full results of the system performance on all 12 CPU cores. As with the single core case, we see generous improvements when motion analysis is used to

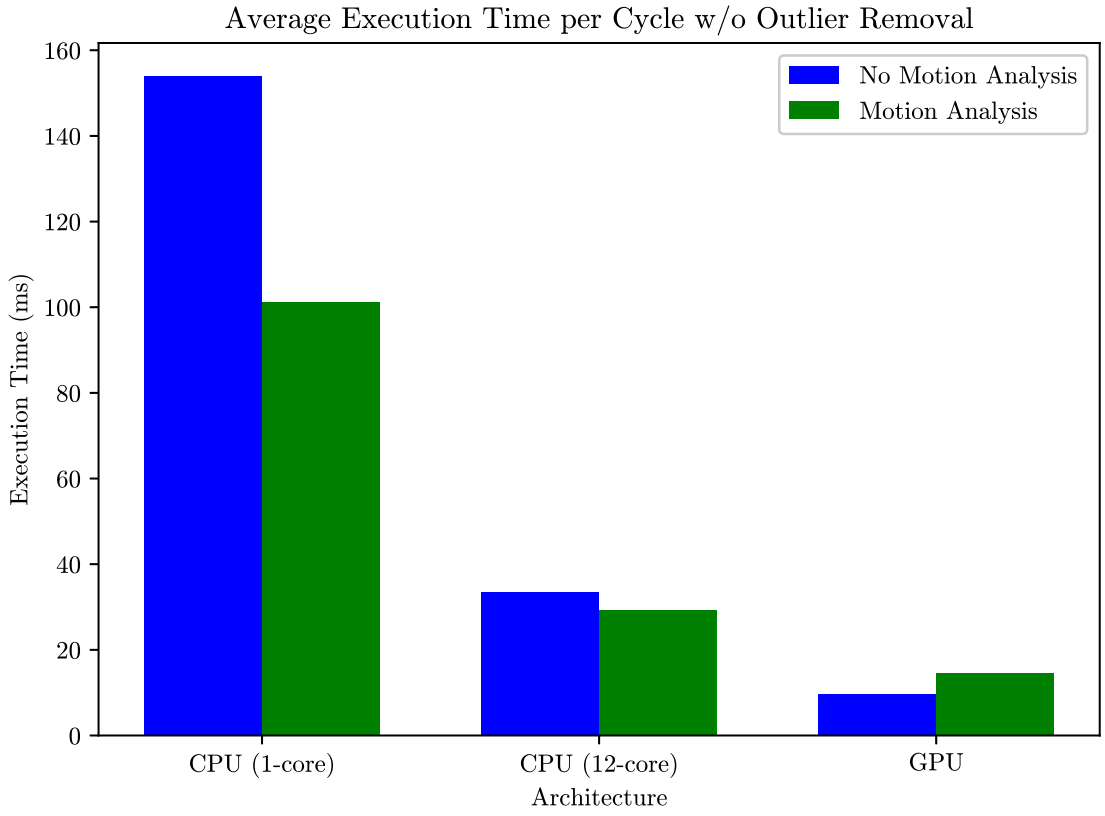


Figure 6.1: Average Execution Time per Cycle (No Outlier Removal)

reduce workload in the early steps but much of the performance gains are lost due to the slow outlier removal algorithm. While a working multithreaded radius outlier removal stage would improve the overall speed of the algorithm as the parallelization factor increases, this stage, like the RGB to HSV conversion stage, does not benefit from motion analysis as all points with known locations must be run through the filter. Even in cases where no motion is present and the early stages would have close to zero execution time, the filter must still process all points leading to diminishing returns of motion analysis. In this implementation, the system speeds up by 14.0% without outlier removal included.

Finally, Table 6.3 contains the full results of the system performance running on a GPU. The GPU implementation incurs two extra stages that the CPU implementation does not

Table 6.1: Breakdown of Figure 6.1 by Stage for CPU (1 core)

Stage	**w/o Motion Analysis**	**w/ Motion Analysis**
RGB to HSV	14.0 ms	14.0 ms
Motion Analysis	-	41.9 ms
Color Identification	13.3 ms	3.8 ms
Codeword Identification	119.8 ms	40.6 ms
Triangulation	6.9 ms	0.9 ms
Radius Outlier Removal	147.3 ms	147.4 ms
Total w/o Outlier Removal	**154.0 ms**	**101.2 ms**
Total w/ Outlier Removal	**301.3 ms**	**248.6 ms**

Table 6.2: Breakdown of Figure 6.1 by Stage for CPU (12 cores)

Stage	**w/o Motion Analysis**	**w/ Motion Analysis**
RGB to HSV	6.0 ms	6.0 ms
Motion Analysis	-	11.4 ms
Color Identification	1.9 ms	0.5 ms
Codeword Identification	18.5 ms	10.5 ms
Triangulation	7.0 ms	0.9 ms
Radius Outlier Removal	150.6 ms	150.6 ms
Total w/o Outlier Removal	**33.4 ms**	**29.3 ms**
Total w/ Outlier Removal	**184.0 ms**	**179.9 ms**

have: the transferring of inputs to the GPU and the transfer of resulting point cloud back to the CPU. The GTX 1070 averaged 4.35 GB/s in our data transfer benchmark and with sixty 1920x1440 24-bit color frames to transfer to the unit each second, we calculated a transfer time of 3.8 ms per cycle. Additionally, with 131355 potential point cloud points in our view and each point holding three floats for X, Y, and Z locations, 3 uint8s for point color, we calculated a transfer time of approximately 0.5 ms per cycle. We add this time to the total execution time of the algorithm, though the two copy engines (the GTX 1070 has one from host to device and one from device to host) can operate asynchronously from the CUDA kernels running so this time accumulation is not actually seen. We can see that despite the GPU implementation being the fastest of all of our tests and stages benefiting with motion

analysis experience a reduction in execution time, including motion analysis actually slows down the system by 53.1% without outlier removal and 37.5% with outlier removal. The time penalty incurred with the background subtraction and the morphological dilation is much greater than the time saved by only computing the later stages on the moving sections.

Table 6.3: Breakdown of Figure 6.1 by Stage for GPU

Stage	**w/o Motion Analysis**	**w/ Motion Analysis**
Frame Transfer x2	3.8 ms	3.8 ms
RGB to HSV	1.4 ms	1.4 ms
Motion Analysis	-	6.3 ms
Color Identification	0.5 ms	0.2 ms
Codeword Identification	3.3 ms	2.3 ms
Triangulation	0.2 ms	0.1 ms
Radius Outlier Removal	4.0 ms[1]	4.0 ms[1]
Point Cloud Transfer	0.5 ms	0.5 ms
Total w/o Outlier Removal	**9.6 ms**	**14.7 ms**
Total w/ Outlier Removal	**13.6 ms**	**18.7 ms**

[1] Conservative estimate based off of data available in [2].

We estimated the performance a GPU implementation of Radius Outlier Removal by looking at the performance of fluid simulation in [2]. Computing forces on 131,072 particles (very close to our number of points and a similar operation) took less than 2 ms to run on a GTX Titan (which has 2688 CUDA cores clocked at 837 MHz). Assuming similar compute capacities, converting that performance to a GTX 1070 (with 1920 CUDA cores clocked at 1506 MHz) gives us an execution time of 2.5 ms, or conservatively 4.0 ms.

Due to the focus on motion of this work, we also looked into how different types of motion affected the system. We analyze two common types of motion that would be seen by a 3D scanning system: rotational and translational, both of which are common in modern video communication systems. We restrict ourselves to rigid objects by not considering scaling movements, thought they should behave similarly to translational movements. We are primarily interested in whether different types of movement with the same subject affects

the execution time. Figure 6.2 shows the results of this experiment where the X+ direction is pointing to the right of the camera view, the Y+ direction is pointing away from the camera, and the Z+ direction is pointing to the top of the camera view. After normalizing all of our execution times across our input videos against the mean execution time of their target architecture and classifying by motion, we saw no substantial difference in execution time between our classifications. All mean execution times of each classification were within 1.1% of the total mean execution time. Even in the case of the *Translate Y* classification where the subject will either start far away from the camera and get closer or vice versa, the average amount of motion detected across the video was similar to the other videos to prevent its deviation (though execution time was less consistent from frame to frame).

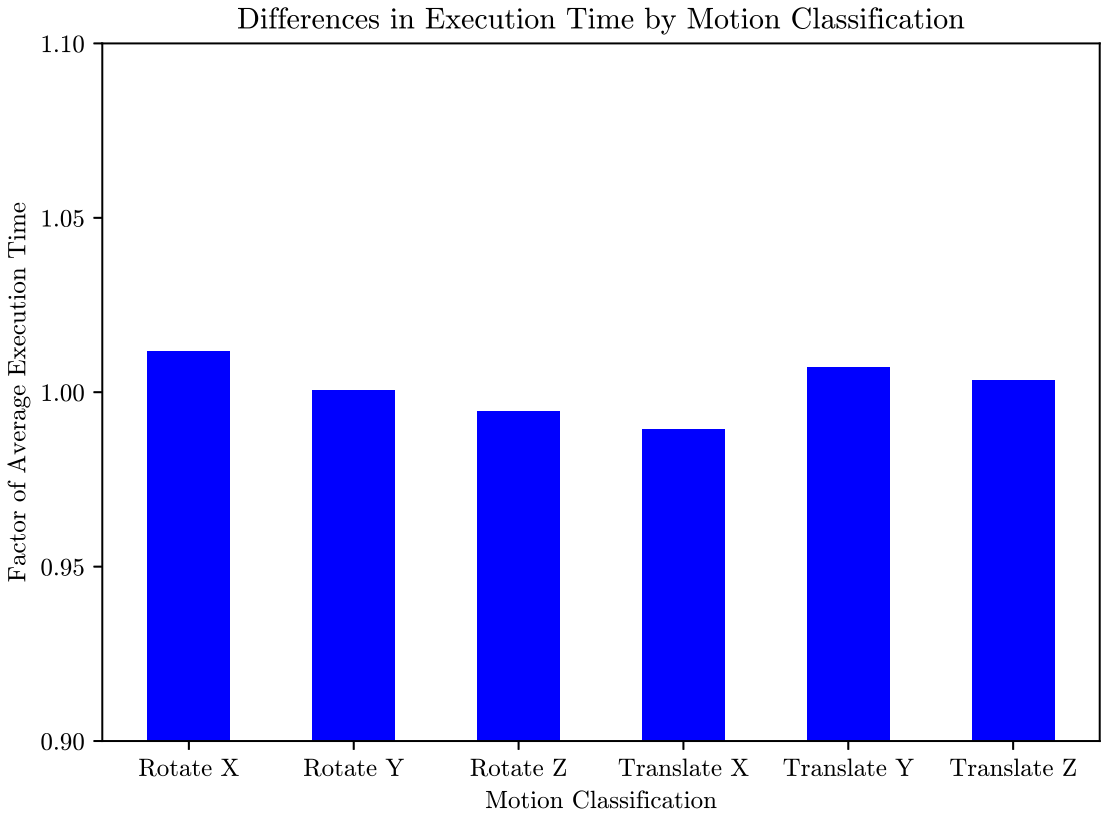


Figure 6.2: Factor of Mean Execution Time by Motion Classification

This system still contains several visual deficiencies in the resulting point that need to be addressed, though these limitations are the result of the encoding sequence chosen, not necessarily due to the addition of motion analysis. However, the use of motion analysis and point propagation can have consequences for incorrectly computed points. We will discuss a few of the limitations in our system in this section and suggest solutions in Chapter 7.

In Figure 6.3, we can see that in some locations there are holes present in our reconstruction. While in the process of turning the point cloud into a mesh with vertex triangulation, these holes would likely be un-noticable, it is still a flaw in the system caused by failing to successfully decode the pattern at that location through the pattern being obscured or identifying the pattern as another codeword. While these flaws can be propagated forward through time due to the usage of motion analysis if they are not properly detected, they can also be prevented by locking them in place after correct detection if motion is not detected in that location which is not the case in the traditional structured light system.

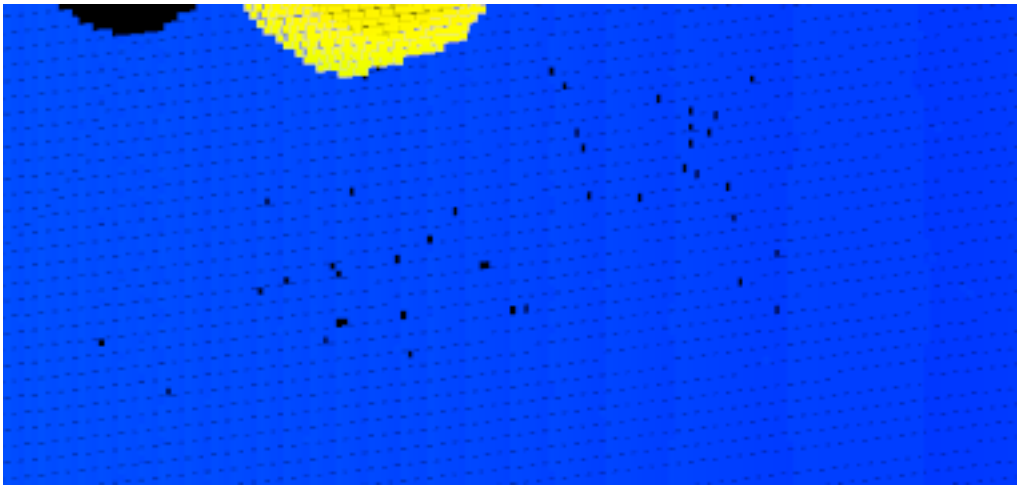


Figure 6.3: Point Cloud Holes

In Figure 6.4, you will notice a few artifacts on the edge of the subject's point cloud. One cause is that near the edge of the subject in the pattern-projected frames, shadows can start to interfere with the proper color identification and thus interfere with the codeword identification. Additionally, if a codeword that would be properly located at the edge of

the subject is mis-identified earlier in its epipolar line, then it will not appear in its proper place and we observe the other edge codewords that were properly decoded as artifacts. As with the previous example, it is possible for these errors to be propagated forward if there is no motion in that location in the near future to force the system to compute its position, though in practice they are often caught as outliers on the next cycle through the algorithm and are not left in the point cloud.

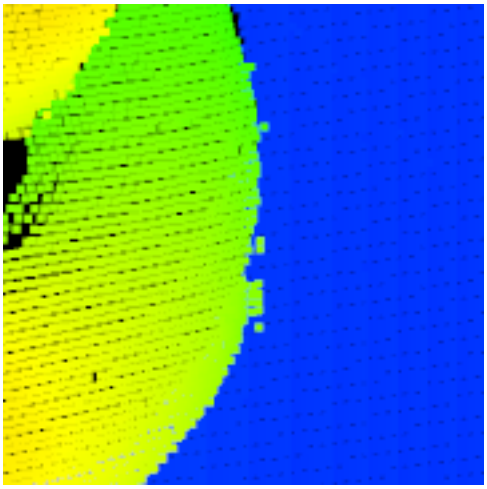


Figure 6.4: Edge Artifacts

In Figure 6.5, one of the Stanford Bunny's ears is obscuring the other one from being detected and the background plane is instead shown in that space. This is a limitation of using a structured light system with only one camera and one projector: since both components must be spatially separated in order to triangulate a point, there may be some objects like the back ear that are within the view of one component (i.e. the camera) but not the other (i.e. the projector). There is no clever encoding method that only uses one camera and one projector that can fix this phenomena, though the effects can be minimized by decreasing the distance between the two at the cost of reconstruction accuracy.

While outlier removal isn't strictly required when motion analysis is used, leaving it out can significantly affect the resulting point cloud and the effects only grow as time goes forward. Figure 6.6 is the result of rotating the Stanford Bunny about its Y axis with

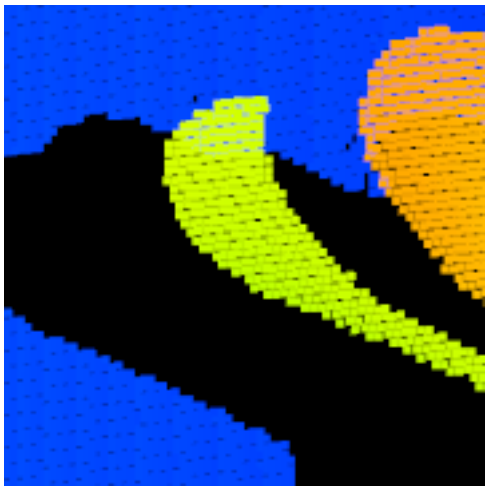


Figure 6.5: Shadow Occlusion

motion analysis turned on but keeping outlier removal off. While the subject is still mostly in tact, the background plane has several pieces missing due to the floating foreground points that were never removed from the point cloud once they were isolated from the main subject.

Figures 6.7 show the final point clouds that our system generated at the final frame of the videos where the subject rotated clockwise about the X axis. The points in these clouds are color-coded by their distance from the camera with blue being the farthest (the background plane), green being in the middle, and red being the closest. These point clouds were saved as PLY files and imported into MeshLab [54] in order to generate a 3D mesh. We used MeshLab to compute normal vectors for each of the vertices and then used the Marching Cubes (APSS) algorithm to reconstruct the surface. The filter scale was set to 3, accuracy was set to 0.01, max iterations was set to 15, sphereical parameter was set to 0, accurate normals was turned on, grid resolution was set to 150, and edge smoothing was turned off. The resulting meshes can be seen in Figure 6.8. Visually, these reconstructed meshes are close enough to the originals to be distinguishable. It is evident that sub-pixel accuracy is needed to gain further resolution into our subjects (such as the fur details on the Stanford bunny or the distinctly flat faces of the MeshLab Knot). Furthermore, the

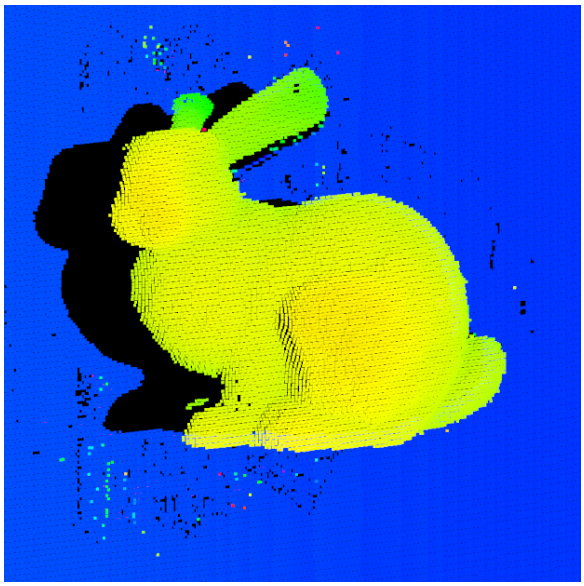
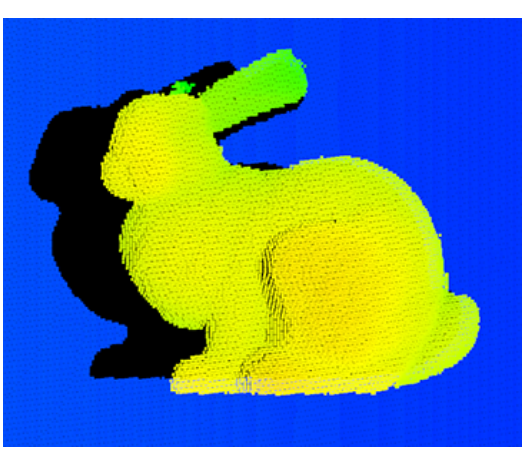
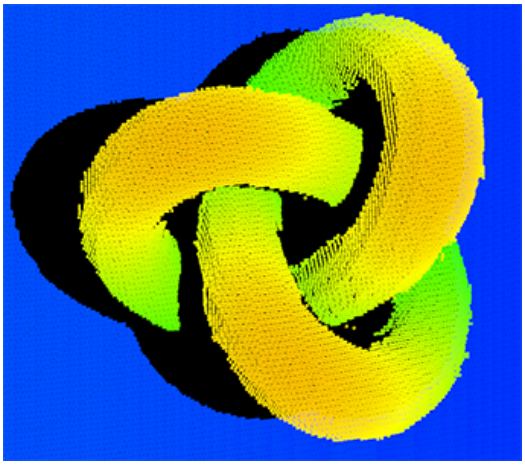


Figure 6.6: Resulting Point Cloud without Radius Outlier Removal

edge artifacts in Figure 6.4 have significantly affected the edges of the reconstructed meshes: what should have been a smooth termination of the model has turned into a set of jagged edges.

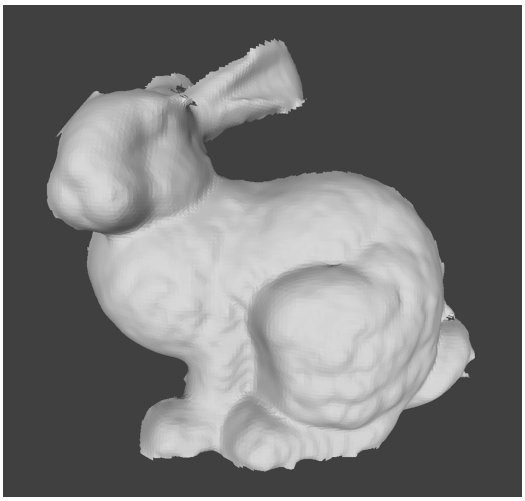


(a) Stanford Bunny Point Cloud

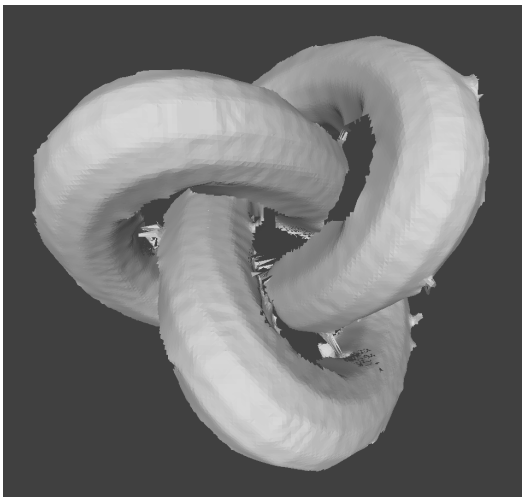


(b) Knot Point Cloud

Figure 6.7: Computed point clouds



(a) Stanford Bunny Reconstructed



(b) Knot Reconstructed

Figure 6.8: Models reconstructed from their point cloud

CHAPTER 7. FUTURE WORK AND CONCLUSION

There is much more work to be done to make it a reliable 3D scanning method. First and foremost, a physical setup is needed to scan real world objects, though because this system already implements camera calibration, it should be relatively easy.

Another addition that can be made is to increase the density of the points that we search for. By searching for the position of the black lines in between the colored lines in the projected pattern, we can double the horizontal scanning density without modifying the pattern at all.

There are several ways to increase the accuracy of the system. Firstly, we can map the intensity of a color band to a gaussian function to find a subpixel center to the band instead of finding the first location of the color to calculate a more accurate depth of the pixel. Secondly, we could choose a pattern that has a higher minimum Hamming distance than the current one to allow for not only error detection, but also error correction. Thirdly, we could use the white light-illuminated frames to normalize the patterned line frames by knowing what colors we are projecting onto. Fourthly, we could introduce multiple projectors and cameras similar to [42] to remove shadow-based occlusions in order to better scan the normally obscured portions of the subject. Finally, we could increase the robustness of the system by adding horizontal lines or switching to a different encoding pattern altogether like one based on M-arrays.

The point cloud generated by this system is currently uncolorized, but since we are also collecting a white light-illuminated frame, we could colorize the point cloud by sampling the points that we detect a correspondence at. If we ran Delaunay triangulation on the

point cloud to turn it into a 3D mesh, we could also use that frame as a texture to overlay onto the mesh. By replacing our current encoding method to a binary M-array encoding, we can replace the rapid projection switching entirely and instead project infrared symbols onto the subject for decoding via an infrared camera and then use an RGB camera to run the motion detection and point cloud colorizing.

Some errors are introduced into the system with the addition of the motion-only restricted pattern decoding. If a codeword is misidentified, we currently add its epipolar line back to the search space, though if it isn't found on the next cycle, it is removed. In the future, a smarter system for determining when to add these points back into the search space and when to keep them out would be helpful in increasing detection accuracy and making sure we don't lose points that we should be searching for. Additionally, sometimes detection errors get past the outlier removal and continue to propagate. By not stopping codeword identification when the first match is found and instead searching for more matches, it is possible to find multiple candidates for the point and check to see which one fits into the model better using fixed-radius k-nearest neighbor search and discarding the others. This would also remove the need to add a codeword's epipolar line back to the search space in the next frame. To further stop error propagation, it may be necessary to have a reset frame periodically that rescans the entire frame to remove any propagated errors. It may be worthwhile to investigate how lossy video encoders such as H.264 handle this propagation.

Other miscellaneous additions that could be made in the future are improving upon the CUDA implementation of each stage to optimize it for the architecture and finding a way to speed up slower stages such as motion analysis to make it more worthwhile on massively-parallel architectures. While there is an inter-stage dependency of the results of outlier removal being necessary for codeword identification, it may be helpful to pipeline some of the operations to take better advantage of the CUDA architecture.

In this thesis, we have demonstrated a method for reducing the amount of work needed to be done in a structured light surface reconstruction system propagating forward points

that have not moved. By alternating projecting a de Bruijn sequence-coded pattern and white light, we can encode depth information while also providing an input sequence that can be used to detect motion. The results captured with this method are visually similar to other methods that use de Bruijn sequence encodings with colored lines that don't propagate points forward through time because they are ultimately prone to the same errors that exist in our method. Our method also operates at a similar speed regardless of the types of motion (spinning, translating, etc.) that is present on the screen. Without taking outlier removal into consideration, in a single-threaded CPU implementation, we see speed-ups of 52.2%. In a CPU implementation with twelve threads, we see speed-ups of only 14.0%. Finally, in a CUDA GPU implementation, we actually see a slowdown of 53.1% due to the non-trivial amount of time spent computing motion. Since we saw the best performance on our single-threaded CPU implementation and the worst performance on our GPU implementation, we conclude that our method of surface reconstruction is useful on computation systems without access to parallelization hardware that only have a few threads to spare (if multiple at all) to the task.

REFERENCES

[1] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[2] R. Hoetzlein, "Fast fixed-radius nearest neighbors: interactive million-particle fluids," in *GPU Technology Conference*, 2014, p. 18.

[3] J. L. Posdamer and M. Altschuler, "Surface measurement by space-encoded projected beam systems," *Computer graphics and image processing*, vol. 18, no. 1, pp. 1–17, 1982.

[4] P. M. Griffin, L. S. Narasimhan, and S. R. Yee, "Generation of uniquely encoded light patterns for range data acquisition," *Pattern recognition*, vol. 25, no. 6, pp. 609–616, 1992.

[5] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[6] C. R. Wren, A. Azarbayejani, T. Darrell, and A. P. Pentland, "Pfinder: Real-time tracking of the human body," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 19, no. 7, pp. 780–785, 1997.

[7] N. Friedman and S. Russell, "Image segmentation in video sequences: A probabilistic approach," in *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 1997, pp. 175–181.

[8] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, vol. 2. IEEE, 1999, pp. 246–252.

[9] P. Kaewtrakulpong and R. Bowden, "An improved adaptive background mixture model for realtime tracking with shadow detection," 2001.

[10] D.-S. Lee, "Effective gaussian mixture learning for video background subtraction," *IEEE transactions on pattern analysis and machine intelligence*, vol. 27, no. 5, pp. 827–832, 2005.

[11] Z. Zivkovic, "Improved adaptive gaussian mixture model for background subtraction," in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 2. IEEE, 2004, pp. 28–31.

[12] Z. Zivkovic and F. Van Der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction," *Pattern recognition letters*, vol. 27, no. 7, pp. 773–780, 2006.

[13] J. Serra, *Image analysis and mathematical morphology*. Academic Press, Inc., 1983.

[14] KiwiSunset, "k-d Tree," https://commons.wikimedia.org/wiki/File:Kdtree_2d.svg, 2006, accessed: 2018-02-14. Licensed via CC BY-SA 4.0.

[15] MYguel, "k-d Tree Binary Representation," https://commons.wikimedia.org/wiki/File:Tree_0001.svg, 2008, accessed: 2018-02-14.

[16] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209–226, 1977.

[17] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge university press, 2014.

[18] J. Buhler, "Efficient large-scale sequence comparison by locality-sensitive hashing," *Bioinformatics*, vol. 17, no. 5, pp. 419–428, 2001.

[19] C. Yang, "Macs: music audio characteristic sequence indexing for similarity retrieval," in *Applications of Signal Processing to Audio and Acoustics, 2001 IEEE Workshop on the*. IEEE, 2001, pp. 123–126.

[20] Y. Jing and S. Baluja, "Visualrank: Applying pagerank to large-scale image search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 11, pp. 1877–1890, 2008.

[21] NVIDIA, "CUDA Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2017, accessed: 2017-08-07.

[22] ——, "CUDA Best Practices Guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2017, accessed: 2017-08-07.

[23] J. Salvi, J. Pages, and J. Batlle, "Pattern codification strategies in structured light systems," *Pattern recognition*, vol. 37, no. 4, pp. 827–849, 2004.

[24] J. Geng, "Structured-light 3d surface imaging: a tutorial," *Advances in Optics and Photonics*, vol. 3, no. 2, pp. 128–160, 2011.

[25] S. Inokuchi, "Range imaging system for 3-d object recognition," *ICPR, 1984*, pp. 806–808, 1984.

[26] M. MIMOU, T. Kanade, and T. SAKAI, "A method of time-coded parallel planes of light for depth measurement," *IEICE TRANSACTIONS (1976-1990)*, vol. 64, no. 8, pp. 521–528, 1981.

[27] M. Trobina, "Error model of a coded-light range sensor," *Technical report*, 1995.

[28] R. J. Valkenburg and A. M. McIvor, "Accurate 3d measurement using a structured light system," *Image and Vision Computing*, vol. 16, no. 2, pp. 99–110, 1998.

[29] D. Skocaj and A. Leonardis, "Range image acquisition of objects with non-uniform albedo using structured light range sensor," in *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, vol. 1. IEEE, 2000, pp. 778–781.

[30] D. Caspi, N. Kiryati, and J. Shamir, "Range imaging with adaptive color structured light," *IEEE Transactions on Pattern analysis and machine intelligence*, vol. 20, no. 5, pp. 470–480, 1998.

[31] M. Young, E. Beeson, J. Davis, S. Rusinkiewicz, and R. Ramamoorthi, "coded structured light," in *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE, 2007, pp. 1–8.

[32] H. Hugli and G. Maitre, "Generation and use of color pseudo random sequences for coding structured light in active ranging," in *Industrial Inspection*, vol. 1010. International Society for Optics and Photonics, 1989, pp. 75–83.

[33] K. L. Boyer and A. C. Kak, "Color-encoded structured light for rapid active ranging," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 1, pp. 14–28, 1987.

[34] T. Monks and J. N. Carter, "Improved stripe matching for colour encoded structured light," in *International Conference on Computer Analysis of Images and Patterns*. Springer, 1993, pp. 476–485.

[35] P. Vuylsteke and A. Oosterlinck, "Range image acquisition with a single binary-encoded light pattern," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 2, pp. 148–164, 1990.

[36] J. Salvi, J. Batlle, and E. Mouaddib, "A robust-coded pattern projection for dynamic 3d scene measurement," *Pattern Recognition Letters*, vol. 19, no. 11, pp. 1055–1065, 1998.

[37] L. Zhang, B. Curless, and S. M. Seitz, "Rapid shape acquisition using color structured light and multi-pass dynamic programming," in *3D Data Processing Visualization and Transmission, 2002. Proceedings. First International Symposium on.* IEEE, 2002, pp. 24–36.

[38] ——, "Spacetime stereo: Shape recovery for dynamic scenes," in *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, vol. 2. IEEE, 2003, pp. II–367.

[39] A. O. Ulusoy, F. Calakli, and G. Taubin, "One-shot scanning using de bruijn spaced grids," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on.* IEEE, 2009, pp. 1786–1792.

[40] H. Morita, K. Yajima, and S. Sakata, "Reconstruction of surfaces of 3-d objects by m-array pattern projection method," in *Computer Vision., Second International Conference on.* IEEE, 1988, pp. 468–473.

[41] R. A. Morano, C. Ozturk, R. Conn, S. Dubin, S. Zietz, and J. Nissano, "Structured light using pseudorandom codes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 322–327, 1998.

[42] R. Furukawa, R. Sagawa, H. Kawasaki, K. Sakashita, Y. Yagi, and N. Asada, "One-shot entire shape acquisition method using multiple projectors and cameras," in *Image and Video Technology (PSIVT), 2010 Fourth Pacific-Rim Symposium on.* IEEE, 2010, pp. 107–114.

[43] R. Furukawa, R. Sagawa, A. Delaunoy, and H. Kawasaki, "Multiview projectors/cameras system for 3d reconstruction of dynamic scenes," in *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on.* IEEE, 2011, pp. 1602–1609.

[44] P. Fechteler and P. Eisert, "Adaptive colour classification for structured light systems," *IET Computer Vision*, vol. 3, no. 2, pp. 49–59, 2009.

[45] Y. Zhang, Z. Xiong, Z. Yang, and F. Wu, "Real-time scalable depth sensing with hybrid structured light illumination," *IEEE Transactions on Image Processing*, vol. 23, no. 1, pp. 97–109, 2014.

[46] SharkD, "RGB Color Solid Cube," https://commons.wikimedia.org/wiki/File:RGB_color_solid_cube.png, 2008, accessed: 2018-03-11. Licensed via CC BY-SA 4.0.

[47] ——, "HSV Color Solid Cylinder," https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png, 2008, accessed: 2018-03-11. Licensed via CC BY-SA 4.0.

[48] OpenCV, "OpenCV Documentation," https://docs.opencv.org/3.1.0/, 2016, accessed: 2016-08-13.

[49] PCL, "Point Cloud Library Documentation," http://docs.pointclouds.org/1.8.1/, 2016, accessed: 2016-08-16.

[50] M. Muja and D. G. Lowe, "FLANN," http://people.cs.ubc.ca/mariusm/index.php/FLANN/FLANN, accessed: 2016-08-16.

[51] ——, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.

[52] J. M. Cohen, S. Tariq, and S. Green, "Interactive fluid-particle simulation using translating eulerian grids," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM, 2010, pp. 15–22.

[53] Stanford, "The Stanford 3D Scanning Repository," http://graphics.stanford.edu/data/3Dscanrep/, 2016, accessed: 2016-08-16.

[54] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia, "MeshLab: an Open-Source Mesh Processing Tool," in *Eurographics Italian Chapter*

Conference, V. Scarano, R. D. Chiara, and U. Erra, Eds. The Eurographics Association, 2008.