FASTER: FPGA Accelerated Simultaneous Trigonal Edge Detection and Rasterization for Robust Robotic Manipulation

Can Eren Derman

April 28, 2020

Abstract of "FASTER: FPGA Accelerated Simultaneous Trigonal Edge Detection and Rasterization for Robust Robotic Manipulation" by Can Eren Derman, Brown University, 04/28/2020.

Advancements in machine learning techniques have encouraged scholars to focus on convolutional neural network (CNN) based solutions for object detection and pose estimation tasks. Most robotic applications fail to achieve desired accuracies in real world conditions because CNNs' performance depends heavily on the similarity between the training and testing data. In their work, *Generative Robust Inference and Perception* (GRIP), the Bahar Group has introduced a second stage to assess CNN outputs and increase the robustness of the estimations, especially in adversarial conditions. Although the iterative likelihood weighting, introduced by GRIP's second stage, increases the pose estimation accuracy significantly, it is not optimized for runtime. In this thesis, we propose *FPGA Accelerated Simultaneous Trigonal Edge Detection and Rasterization* (FASTER) as a faster alternative to GRIP. On top of producing robust pose estimations in adversarial scenarios, FASTER optimizes the bottlenecks of the second stage algorithm by creating a highly parallelized custom circuit on an FPGA with a novel edge detection algorithm. We benchmark FASTER by comparing it to GRIP's accuracy on dark and occluded scenes. Our results show that FASTER achieves a significant speedup with outstanding energy savings compared to GRIP. These results are especially impressive given that GRIP is implemented on a much more sophisticated hardware with higher clock frequency.

FASTER: FPGA Accelerated Simultaneous Trigonal Edge Detection and Rasterization for Robust Robotic Manipulation

by
Can Eren Derman

A Thesis submitted in partial fulfillment of the requirements for Honors in the School of Engineering at Brown University

Providence, Rhode Island
04/28/2020

This thesis by Can Eren Derman is accepted in its present form by the School of Engineering as satisfying the research requirement for the awardment of Honors.

Date 4/28/2020

DocuSigned by:
Ruth Iris Bahar
CE39966A132C4CB...

Prof. Ruth Iris Bahar, Principal Advisor

Date 4/28/2020

DocuSigned by:
Gabriel Taubin
4006A8F4C4D0490...

Prof. Gabriel Taubin, Reader

Acknowledgements

As someone who did not write a single compilable line of code before attending Brown University, I would like to devote my thesis to those who have equipped me with the skills to comprehend and manipulate the most powerful medium of the 21^{st} century:

Dear Professor Bahar,

Meeting you on the first day of school, as you advised me to take the new Honors Intro to Engineering class, I had no clue of the frequency at which I would receive and follow your advice. I can confidently admit that my engineering education was mostly molded by your guidance, wisdom and teachings. I have learned the essence of computer engineering: designing digital systems and control theory, through you. I have had an unparalleled experience in your lab as an undergraduate for the past year. Getting to know you and having the opportunity to work with you has been the greatest privilege I had at Brown. I am mostly thankful to you, for the engineer I am today.

Dear Jasmine,

You are undoubtedly the smartest person I have ever met, even without having your Ph.D yet. I am grateful for the past year as I had the chance to work with you closely, also for your patience and understanding. I have learned from you at least as much as I have learned from my Professors at Brown. Thank you for trusting me with your idea and all your help along the way. None of this would have been possible without you.

I would like to thank Andy van Dam for making me enjoy coding, Allan Bower for showing that engineering can be fun too, Jacob Rosenstein for being my advisor on this rocky road, Stefanie Tellex for surfacing my potential, Sherief Reda for igniting a love for parallization and an appreciation for computers in me, and Gabriel Taubin for his insightful counsel throughout my thesis.

Dear Mom, Dad and Sister,

I am especially grateful for all of you, as you have inspired me with your creativity, perseverance and strong moral values. This work is a tribute to all your efforts to provide me with the best opportunities possible. I hope to pay my tributes for your love and support by contributing to the aggregated human knowledge and expanding its boundaries.

Contents

1 Introduction **3**

2 Background **5**

2.1 Edge Features and Inlier Ratio 5

2.2 Computational Complexity and Edge Detection 8

2.3 Rasterization 9

2.3.1 Perspective Projection 10

2.3.2 Edge Function 10

3 Methodology **12**

3.1 High Level Synthesis 12

3.2 Trigonal Edge Detection and Inlier Ratio Calculation 13

4 Experiments **16**

4.1 Implementation 16

4.1.1 Classifying Pixels in Triangles 16

4.1.2 Creating the Edge Map 19

4.1.3 Calculating the Inlier Ratio 20

4.2 Evaluation 21

4.2.1 Edge Map Functionality 22

4.2.2 Inlier Ratio Accuracy 22

4.2.3 Testing on the FPGA 23

5 Challenges **24**

5.1 Thresholding the Edge Function 24

5.2 Previous, Current Pixel Relation 26

5.3 Upper and Lower Edge Bounds 28

5.4 Reducing Dependencies and Memory Accesses for Pipelining Loops 31

6 Results **35**

6.1 Accuracy 35

6.2 Speed 37

6.3 Resource Usage 37

6.4 Power Consumption 39

7 Conclusion and Future Work **40**

7.1 Conclusion 40

7.2 Future Work 40

A Codes of the Implemented Functions **42**

A.1 Triangle Set 42

A.2 Create Edge Map 48

A.3 Depth Substitution 50

Bibliography **55**

Chapter 1

Introduction

Although convolutional neural networks have been in widespread use, they are not robust enough for generalized robotic applications. They fail to achieve higher levels of accuracy when the inferred scene has adveserial conditions such as limited lighting or occlusion among objects. In their GRIP paper [5] Professor Bahar and her lab are offering a three stage approach of discriminative CNN-based recognition, generative probabilistic estimation, and robot manipulation to increase the robustness of such networks.[1]. Figure 1.1 illustrates the three stages of the pipeline, where the third stage would be robot manipulation.

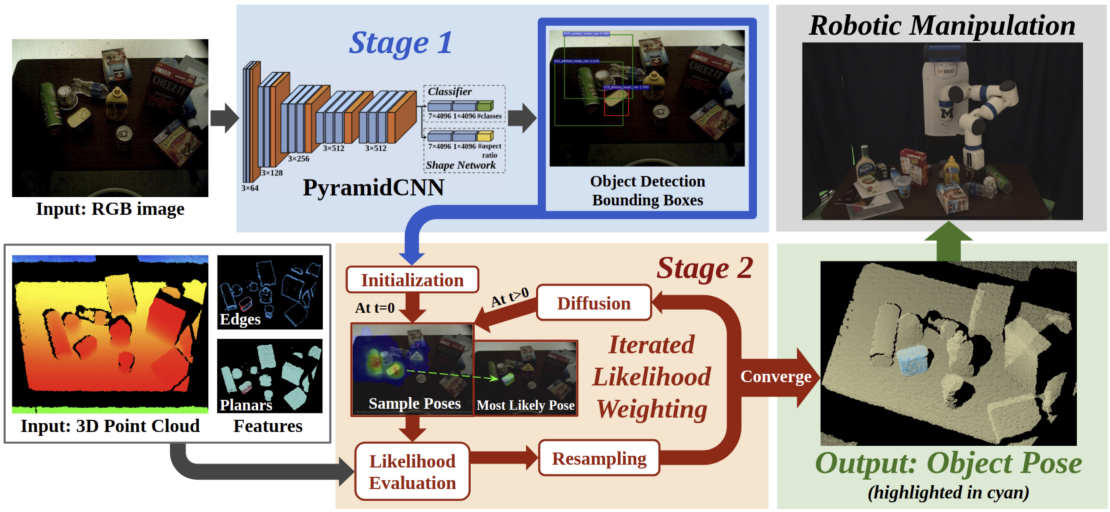


Figure 1.1: Different stages of GRIP. Figure taken from the GRIP paper [5]. Second stage initializes with the outputs of the first stage. The bounding boxes and confidence scores are subjected to an iterated likelihood evaluation, where the observed object's features are compared to rendered samples' features. Resampling generates new samples based on previous samples with high likelihood scores. Diffusion adds noise to the generated samples to increase robustness.

The first stage outputs consist of bounding boxes and associated confidence scores for each object class.

[1]This research builds off of the work on the paper "GRIP: Generative Robust Inference and Perception for Semantic Robot Manipulation in Adversarial Environments," and in particular, is focused on accelerating the feature extraction step within the second stage of GRIP

They are used as a starting point for sampling initial poses of object proposals found within each bounding box. The generated poses are compared to the observed object by assessing how the edge and planar features of the sampled object overlap with those of the observed object. Poses with higher match rate on previous iterations are repeatedly sampled until all the samples converge on a single pose. This method introduces a much more robust pose estimator on top of the CNN output. However, introducing an additional iterated likelihood evaluation in sequence to the CNN increases the inference time.

The main bottleneck of the second stage algorithm is the rendering of point clouds with potential object poses, feature extraction from these point clouds and the inlier calculation. While each of these steps is amenable to acceleration on a GPU, together they cannot be run in parallel. Because hundreds of candidate poses are tested before convergence is finalised, rasterization, edge detection and inlier calculation are highly repeated processes. This thesis describes a method called *FASTER* (**F**PGA **A**ccelerated **S**imultaneous **T**rigonal **E**dge detection and **R**asterization) to implement the edge feature extraction and inlier calculation on a Field Programmable Gate Array (FPGA) to run simultaneously with rasterization of the object models. FPGAs allow direct integration of programs into hardware by reconfigurable logic. It was theorized that this approach would decrease the compute time significantly by both parallelizing the edge detection step with object rasterization and introducing a cheap, efficient way of detecting object edges. In addition to significantly decreasing the compute time of the whole flow, the outlined approach was expected to reduce power consumption and use less resources when compared to its predecessor, GRIP.

In the upcoming chapters, the research is organized in the following manner: Chapter 2 provides in-depth background information about the important concepts used in this study. Chapter 3 introduces our methodology FASTER and its theory, while Chapter 4 discusses the experiments where the theory was implemented. Chapter 5 elaborates on the challenges faced throughout the study. Chapter 6 shows that a hardware implementation of FASTER improves runtime by 43.7% and energy consumption by 97.8% compared to GRIP. Chapter 7 concludes this thesis by discussing its implications in the field of robotics, and future improvements.

Chapter 2

Background

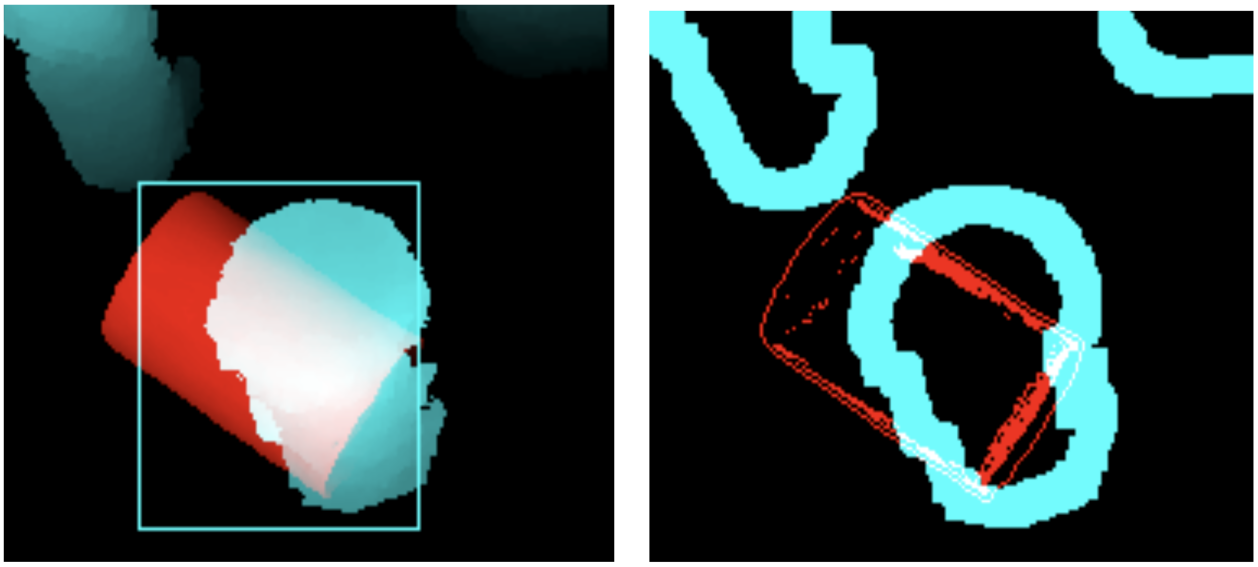
This chapter introduces important concepts upon which FASTER was built. The covered concepts are edge features and inlier ratio, computational complexity of edge detection, and rasterization. As FASTER builds off of the GRIP paper, this chapter also lays out a detailed analysis of the GRIP flow using both visualizations and mathematical explanations, establishing a foundation for proceeding sections of the thesis.

2.1 Edge Features and Inlier Ratio

The two stages of the algorithm that the Bahar Group has been working on can be described as object detection and robust pose estimation. As shown in Figure 1.1, stage two performs sample based generative inference to estimate the pose of the inferred object more confidently.

The second stage is initialized with the weighted samples $\{q(i), w(i)\}_{i=1}^{M}$ coming from the first stage object detection. Each of the weighted samples represent an object pose, where $q(i)$ represents a 6 degree of freedom (DoF) sample pose with its corresponding weight $w(i)$. Given an object pose q and knowing the object's geometry model, the second stage renders a 3D point cloud of what the object would be observed as, if it had the hypothesized pose. Here, rendering means to make visible; to draw. While, 3D point clouds consist of a set of data points in space that represent an object or a scene, after they are rasterized, they consist of pixels. Using the rasterized sample poses, the feature inlier ratios are computed. The associated weight $W(q)$ for pose $q(i)$ is calculated by the likelihood function, which takes into account the weight of the bounding boxes from the first stage output, and raw pixel-wise and feature-based inlier ratios. The computed weights are then plugged into the iterated likelihood weighing process, where each sample is continuously assigned a new weight. This process is called resampling. Then, each pose is subjected to some noise during the diffusion stage, which terminates when the weight $W(q_{final})$ is above a certain threshold or the maximum number of iterations are completed. Finally, q_{final} is accepted to be the actual pose of the observed object. This thesis focuses on the optimization of the feature-based inlier ratio calculation [5].

During the pose estimation stage, two inlier functions are used. One of them is the raw pixel-wise inlier while the other is the feature-based inlier ratio. The raw pixel-wise inlier function between two point clouds



(a) Extracted Planar Features

(b) Extracted Edge Features

Figure 2.1: Comparison of the planar features (a) and edge features (b) between the observed scene and rendered scene.

is calculated with the following formula:

$$Inlier(p, p') = \mathbf{I}(||p - p'|| < \epsilon), \tag{2.1}$$

where $\mathbf{I}$ is the indicator function, p and p' come from the two different point clouds. While the raw pixel-wise inlier is useful to determine if individual points on the rendered poses agree with the observation, the feature-based inlier ratio offers a more robust, higher level comparison of the edge and planar futures of the rendered and observed scenes. Geometry feature point clouds are extracted from the rendered samples of object point clouds and the observation object point clouds. With the geometry features, the feature inlier ratios can be calculated. By applying the feature point extraction definition from Zhang et al. [17], we can distinguish between the edges and planar points among the objects:

$$c(u, v) = \frac{||\sum_{(u',v') \in N(u,v)} p_{(u',v')} - p_{(u,v)}||}{|N(u, v)| \cdot ||p_{(u,v)}||} \tag{2.2}$$

where, $p_{(u,v)}$ represents a vector of the distance between a point in the point cloud and the base frame, and $N(u, v)$ is the set of that point's neighbouring points. So the equation basically compares the distance between the point cloud vector with pixel indices (u, v) to all of its neighbouring vectors and normalizes the result with the size of the set of neighbours and the length of the vector $p_{(u,v)}$. The calculated value of $c(u, v)$ is then compared to a threshold value to be selected either as an edge or a planar point. Although this method is successful in accurately extracting the features of an object, it is computationally expensive.

Intuitively, the method described above calculates the change in depth between the points corresponding to neighbouring pixels. Thus, it provides a good distinction between edge and planar points (points on the surface of objects) and uses the geometric information extracted from the 3D point clouds to enhance the

(a) Observed Scene (b) Extracted Edge Features of the Observed Scene

Figure 2.2: An example of an observed scene and how its edge features look after applying the feature point extraction from Zhang et al. [17]

.

robustness of the network. After the feature extraction is applied to both the observation and the rendered sample point clouds the inlier ratios are calculated using the following formula:

$$I = \frac{1}{|r|} \sum_{(u,v)} Inlier(r(u,v), o(u,v)). \tag{2.3}$$

The inlier ratio is calculated using Equation 2.3 by computing the match rate of the depth values of the edge and planar points in the observed scene $o(u,v)$ and the rendered scene $r(u,v)$. Dividing the number of matches by the number of pixels within the rendered scene, we normalize the sum and obtain the inlier ratio. As described before, the inlier ratio then plugs into the calculation of weights $W(q)$, for each pose $q(i)$:

$$W(q) = a_{box} w_{box} + a_b I_b + a_r I_r + a_e I_e + a_p I_p, \tag{2.4}$$

where the initial product term $a_{box} w_{box}$ comes from the first stage network, and w_{box} is the confidence score of the bounding box. All of the a terms represent the importance of the term they are multiplied by and add up to 1. While I_e, I_p are the inlier ratios of the edges and planar points between the rendered object and the scene object, I_r is the raw pixel-wise inlier ratio, and I_b is the inlier ratio of the part of the rendered sample that lies within its corresponding bounding box. Since the CUDA implementation of the described algorithm executes rendering, feature extraction and inlier ratio calculation in sequence, it is the perfect opportunity to pipeline computation in hardware. In addition, the proposed method to extract the edge features used in the feature based inlier calculation does not require the usage of any computationally heavy arithmetic operations.

2.2 Computational Complexity and Edge Detection

Computers are composed of transistors made of semiconducting material. Transistors act as switches; closing and opening the switch depend on the voltage applied on the gate of the transistor and the semiconductor type. More complex structures called gates are formed using these building blocks. Gates output a binary value depending on the binary inputs. Moreover, different combinations of gates create computational units and these units are used in arithmetic operations. Today, almost all computational applications depend on the usage of arithmetic operations. Numerous scholars have been exploring ways to optimize circuits to reduce the compute time for each arithmetic operation.

Arithmetic operations are designed in hardware to accommodate for different input-output sizes as well as applications. Thus, a size-agnostic measurement unit to compute run-time for any given input size was devised by computer scientists. This is called the big-O notation. According to National Institute of Standards and Technology (NIST), the big-O notation is defined as: "A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation $f(n) = O(g(n))$ means it is less than some constant multiple of $g(n)$ [1]." Table 2.1 contains the best-case (fastest) run-times of the different arithmetic operations used in the GRIP paper's edge detection implementation:

Table 2.1: Computational Complexity of Arithmetic Operations

Operation	Algorithm	Run-time (Big-O)
Addition	Carry-Select Adder [12]	$O(n), O(log(n))$
Subtraction	Carry-Select Subtractor [12]	$O(n), O(log(n))$
Multiplication	Harvey-Hoeven Algorithm [9]	$O(n * log(n))$
Division	Burnikel-Ziegler Divide-and-Conquer Division [3]	$2K(n) + O(nlog(n))$
Square Root	Newton's Method [8]	$O(log(n))$

One of the main differences of the proposed approach is the edge detection step. Even the most preliminary algorithms such as the Canny Edge Detection require many computationally heavy steps such as convolution, root mean square and gradient vector calculations [4]. These processes, especially when executed sequentially to the whole algorithm, require arithmetic operations such as multiplication and subtraction on every pixel of the image which ends up being very costly. The GRIP paper extracts edge features from an n-by-n input rendered image in $O(|N(u,v)|n^2)$ time, where $|N(u,v)|$ is the size of the set of neighbors of pixel $p(u,v)$. Since, $|N(u,v)|$ is a constant, the runtime can be simplified to $O(n^2)$. Similarly, rasterization takes $O(n^2)$ time for an n-by-n image. Because these are sequential processes, the final runtime of the two stages is $O(n^2) + O(n^2)$, but constants are omitted in sub-quadratic times finalizing the runtime as $O(n^2)$. Because the proposed approach is implemented in parallel to rasterization, its runtime will be equal to $O(n^2)$, same as rasterization. Thus, the speedups achieved in using our proposed approach will not be proportional to the input image size. While the bit size of the variables also do not change in the proposed approach, looking at the computational complexity per number of bits b for each arithmetic operation used in GRIP paper, we can get a sense of the constant speedup that can be achieved. The GRIP implementation makes use of

Equation 2.2 to extract edge features from rendered images.

To highlight the computational complexity of these operations, one can observe that Equation 2.2 computes the norm of the difference between a rendered pixel's neighbors and the observed pixel's neighbors. The norm is then divided by the product of the absolute value of the neighboring pixel and the norm of the pixel at hand from the observation scene. Using Table 2.1, one can infer that the best case run-time required to calculate the feature inlier ratios used to compute $c(u, v)$ is:

$$16 * O(log(b)) + O(b * log(b)) + O(log(b)) + O(b * log(b)) + (2K(b) + O(b * log(b))). \tag{2.5}$$

Here, b is the number of bits and $16 * O(log(b))$ is the time it takes to subtract m b-bit neighbours from the b-bit pixel $P(u, v)$ and calculate the sum of the m differences. While calculating the norm, the sum is multiplied by itself, which requires $O(b * log(b))$ compute time before taking the square root, which takes $O(log(b))$ time. The divisor is the product of the length of the neighbors and the length of the vector $p(u, v)$, which is another multiplication that takes $O(n * log(n))$ time. Finally, the division takes $(2K(b) + O(b * log(b)))$, which accounts for the largest part of the computation. Because division takes the most time, we can approximate that each pixel takes $(2K(b) + O(n * log(b)))$ time in total. This calculation is done for every pixel in the observation scene, so n^2 times. Thus, it is computationally expensive to run. The proposed FASTER method integrates the edge feature extraction with the rasterization by only using additional Boolean operations to the already existing rendering algorithm, which, in theory, would take linear $O(b)$ time to execute.

2.3 Rasterization

The rendering of a 3D object vector onto a 2D pixelated image is referred to as the *rasterization* process. This enables 3D objects to be represented as a single channel image, where each pixel value represents the depth of the corresponding point on the object. There are numerous methods and algorithms to rasterize various geometric components [2, 10, 14]. Because we use the rasterized scenes for feature extraction, the accuracy of each rasterized pixel's shade, which represents the depth of that pixel in raster space, is extremely important. Thus, the selected algorithm should be capable of distinguishing which triangles are visible from the "camera" perspective. In addition, the depth values of the pixels of the rasterized object should only be attributed to the visible triangles. These requirements compelled us to use a method that solves the visibility problem. There are two such methods: ray tracing [7] and perspective projection [11].

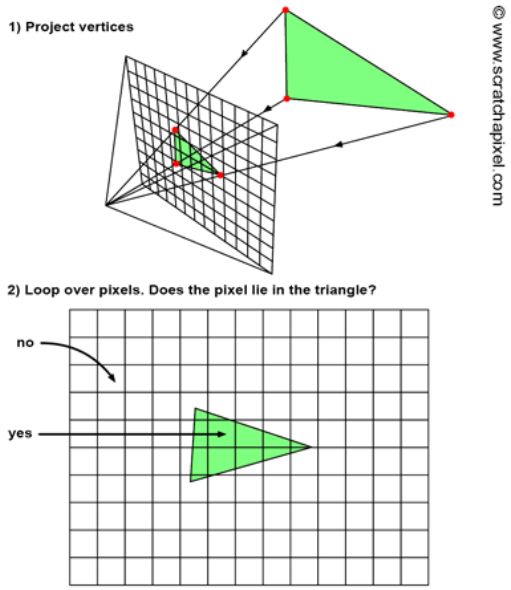


Figure 2.3: Figure taken from ScratchPixel [13] provides an intuitive visualization of the algorithm. Firstly, the vertices of the object triangle are projected onto the screen. Then the pixels are checked if they are within region bounded by the triangle.

Ray tracing traces a ray from each pixel on the screen to the 3D object and checks for intersections. This method is very costly, as it requires to trace a ray from every pixel. Thus, perspective projection was selected as the rasterization algorithm.

2.3.1 Perspective Projection

Perspective projection is an object centric method, as it starts on the object. In order to go from the 3D scene to the 2D screen vertices of the triangles on the 3D object model are projected onto the virtual screen. Then, all the pixels that lay within the three vertices are marked as a part of the rastered triangle [13].

The functionality of perspective projection is illustrated in Figure 2.3. However, as evident in the figure, it is redundant to loop over every pixel on the screen to check if it lies within the triangle as each triangle constitute a very small portion of the rasterized object. In addition, even a single object model will have hundreds of triangles contained in it, looping over every pixel in the image to check if it lies within the tree vertices is redundant and costly. Thus, a 2D bounding box can be calculated to only check the pixels around the triangle of interest. By finding the minimum and maximum coordinates of the vertices in raster space, we can choose the start and end coordinates of the bounding box.

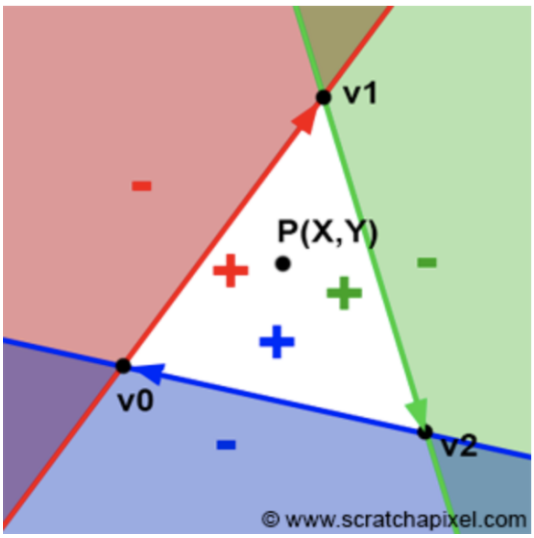


Figure 2.4: Visualization of the edge function output signs according to regions around the triangle, image taken from ScratchPixel [13].

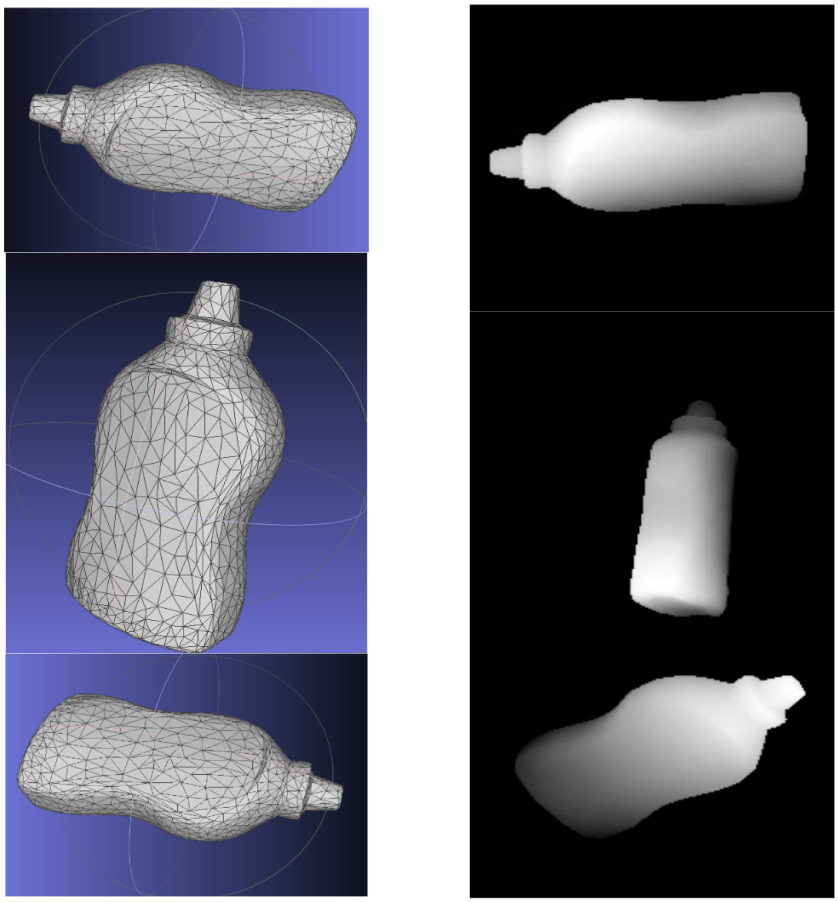
2.3.2 Edge Function

Each pixel in a given bounding box is checked to determine if it overlaps with the projected triangle. This process utilizes a technique called the *edge function* and *barycentric* coordinates [6]. Barycentric coordinates enable the edge function to efficiently determine if a 2D point lies within a projected triangle. Due to the scope of the research, we will not go into its details. On the other hand, the edge function was introduced by Juan Pineda in 1988; it outputs a negative number when point P is outside of the edges, positive while inside, and zero on the edges of the rasterized triangle [11]. This is illustrated in Figure 2.4. The edge function is defined as follows:

$$E_{01}(P) = (x_P - x_{v0}) * (y_{v1} - y_{v0}) - (x_{v1} - x_{v0}) * (y_P - y_{v0}), \tag{2.6}$$

where $E_{01}(P)$ represents the edge function for point P and edge, between vertices $v0$ and $v1$. The x and y values correspond to the x and y coordinates of the specified vertices or points. During rasterization, this method is utilized to check if a point is inside, outside or on the boarder of the triangle of interest. If the pixel is within a triangle, all edge functions - $E_{01}(P)$, $E_{12}(P)$, $E_{02}(P)$ - of that triangle will output a non-negative number for that pixel.

Starting at the top row and column of the triangle's 2D bounding box, the algorithm checks all tree edge functions of each pixel in the first row to determine if the pixel is contained within the projected triangle. In addition, to solve the visibility problem, which happens when there are several triangles that overlap the same pixel, we use a 2D array called the depth buffer. It stores the distance between the pixel and the closest triangle. If a new triangle has a shorter distance, the value is updated as the new triangle is in front of the previous one [13]. Once the first row is finished, instead of going back to the minimum column, the algorithm continues with the row below. This time, starts from the maximum column. In other words, the algorithm follows a zigzag traversal. This specific way of traversal allows the development of our novel theory of Trigonal Edge Detection.



(a) Example 3D object models of the mustard bottle class

(b) End result of rasterized 2D object scene

Figure 2.5: 3D object models and their rasterized representations after perspective projection is applied.

Chapter 3

Methodology

This chapter covers the innovative theory of Trigonal Edge Detection, and how it was implemented on hardware. We will also introduce the concept of high level synthesis, which enabled us to achieve the targeted parallelization and functionality.

3.1 High Level Synthesis

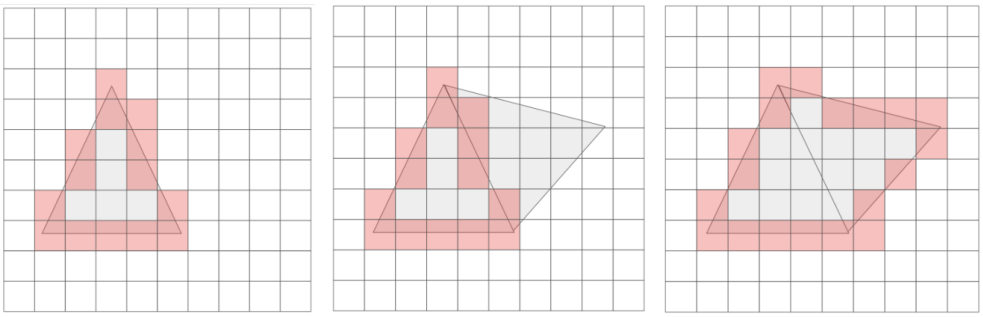
Current computer graphics libraries used for 3D rendering do not permit access to the rasterized triangles as they are being produced. Thus, it is impossible to achieve the proposed parallelization on a GPU. Due to the very nature of the project, it requires custom low level access and architecture design in order to achieve what is proposed. It is customary to use a low level design language such as Verilog in order to design hardware. However, given the scope of the whole project, it would be impossible to use Verilog to design the complete flow.

The Vivado High Level Synthesis tool is a design suite to synthesize Verilog code from C++. It makes it easier to design more convoluted hardware systems in an object oriented manner. Vivado HLS is used in this thesis to design a custom tailored hardware to achieve the proposed parallelization on the three stage process. This allowed every resource to be optimized and pipelined.

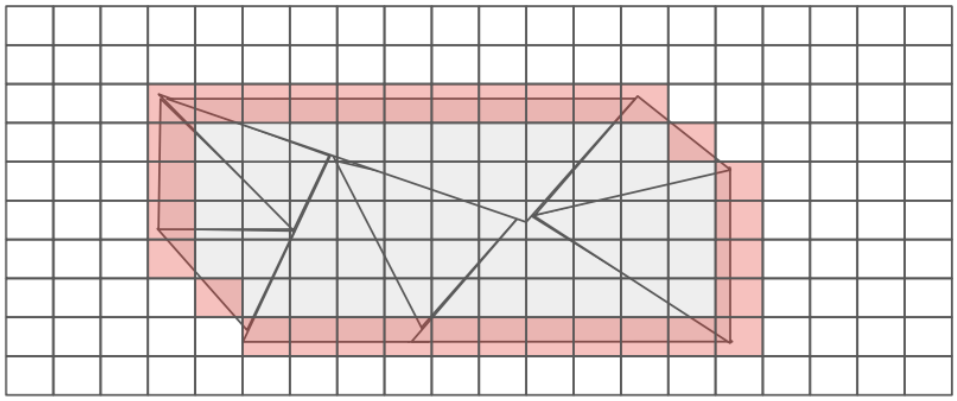
Vivado HLS Design Suite enables users to make hardware level design choices through the usage of certain syntax and protocols. This study mainly makes use of the #pragma syntax and streams. The #pragma *pipeline* is used within loops to indicate that the compiler should pipeline that loop if possible. While, the AXI protocol makes it possible to send information via streams. Streaming information enables different functions to access data without having to wait for the other function to conclude. This prevents processes from having to run sequentially.

In addition, within HLS, each variable can have a custom type with predetermined bits, decimal point or sign bit. Thus, the type of every variable used in this project was carefully crafted for its purpose. Consequently, not a single bit was wasted. All in all, HLS not only enabled this project to be possible but also empowered us to optimize speed, resource usage and power consumption.

3.2 Trigonal Edge Detection and Inlier Ratio Calculation



(a) Filtering of coinciding edges among rasterized triangles



(b) End result contour edge map of cracker_box overlaid with rasterized triangles

Figure 3.1: Proposed trigonal edge detection process and its result.

The proposed methodology to calculate edge features and inlier ratios in parallel with object rasterization was theorized by Yanqi Liu, who is a Computer Science Ph.D. candidate at Brown University and one of the authors of the GRIP [5] paper. The methodology makes use of the unique properties of the rasterization algorithm. The general idea is to combine the rastered triangles with adjacent edges to create a new polygon until all the rendered triangles are within a single polygon. Then, the edges of the polygon are used to represent the contour edges of the object. This idea is visualized in Figure 3.1.

Each rastered object is within a 200-by-200 bounding box. In theory, by introducing a 1-bit 200-by-200 bitmap to store and update the edge information, the pixels on the edges of the final polygon could be detected. This process is delineated on Figure 3.1. The pink pixels represent a 1 (HIGH) on the bitmap while the rest would be 0s (LOWs). At the end, the bitmap would represent the outline of the object once all the rastered triangles are taken into account. That's why we will refer to our contour edge detection method as Trigonal Edge Detection moving forward.

In reality two 1-bit maps were required to keep track of the object pixels and output the desired edge map. The additional algorithm needed on top of the already existing rasterization is delineated with the

following pseudo code:

```
void create_edge_map(index_stream()){
    u1_Data_T Edge_Map[200][200];
    u1_Data_T Object_Map[200][200];
    index_stream.read(index);
    for all indices {
        if index is not an edge {
            Object_Map[index.row][index.col] set 1;
            if Object_Map[index.row+1][index.col] is not 1 {
                Edge_Map[index.row+1][index.col] set 1;
            }
            if Object_Map[index.row-1][index.col] is not 1 {
                Edge_Map[index.row-1][index.col] set 1;
            }
        }
        else {
            if (Object_Map[index.row][index.col] is not 1) {
                    Edge_Map[index.row][index.col] set 1;
            }
        }
    }
}
```

The algorithm will be explained in extensive details through Chapters 4 and 5. In the meantime, one can infer from this pseudo code that the proposed edge detection methodology only depends on Boolean logic. The extracted edge features of the rendered object model can then be used with their corresponding depth values to calculate the inlier ratio. As mentioned in Section 2.2, standard edge detection algorithms need to run on complete images and require to traverse each pixel several times. Their accuracy depends on computationally heavy steps. To be specific, the GRIP paper uses Equation 2.2 to identify edges from planar points. Reiterating the equation here for convenience to the reader:

$$c(u,v) = \frac{||\sum_{(u',v')\in N(u,v)} p_{(u',v')} - p_{(u,v)}||}{|N(u,v)| \cdot ||p_{(u,v)}||}$$

One can infer that $c(u,v)$ is calculated for each pixel with row u, and column v. So after a rendered object is completely rasterized, which requires the whole bounding box to be traversed once, each row and column is traversed again for the edge detection. For an n-bit number the computational load per each n-bit pixel was calculated to be $(2K(n)+O(n*log(n)))$ in Section 2.3. So the theorized approach should perform much faster than the GPU implementation described in the GRIP paper.

In addition, the GRIP paper calculates the inlier ratios through Equation 2.3. However, the proposed approach modifies this formula to accelerate the calculation process. Instead of calculating the inlier for all the channels of the corresponding indices in both scenes, we only compare the depth channel of the edges and planar points of the rendered scene, which is denoted as r_z, to the depth channel of the edges of the

observed scene, which is represented by o_z. This simplified approach is shown in Equation 3.1.

$$I = \frac{1}{\mid r_z \mid} \sum_{(u,v)} \text{Inlier}(r_z(u,v), o_z(u,v)) \tag{3.1}$$

In this chapter, we have walked the reader through our advanced hypothesis and how it works. The next chapter provides a more detailed explanation of the experiments conducted to test the proposed theory and its end results.

Chapter 4

Experiments

This chapter is divided into two sections: implementation and evaluation. Implementation depicts the FASTER versions of rasterization, feature extraction and inlier calculation in subsequent sections. While evaluation describes how the outputs of each of these subsections are assessed.

4.1 Implementation

The final version of the algorithm consists of 3 different functions: triangle_set(), create_edge_map(), and depth_substitution(). On a high level, triangle_set() rasterizes the rendered object using triangles of various shapes and sizes, create_edge_map() creates a binary bitmap of the edges of the rendered object in parallel to the rasterization process, and depth_substitution() calculates the inlier ratio after edge features are extracted from both the observed and the rendered scenes. This flow is visualized in Figure 4.1

4.1.1 Classifying Pixels in Triangles

To begin with, the GRIP flow renders rasterized objects through the function triangle_set(). This function is where the edge function described in Section 2.5 is used to determine if a pixel belongs to the rendered object. This is the first function in which each pixel within a bounding box is identified as either edge, object or background. For the scope of this thesis, only the parts of the function that are relevant to the Contour Edge Detection will be discussed.

As an overview, triangle_set() traverses each pixel within the bounding box set by the triangle vertices. It checks whether the pixel with the previous index or the current index is within the object boundaries. The information of each pixel is stored in an edge_position struct. The struct has the following three fields: actual_row, actual_col and not_edge. Where actual_row and actual_col store the x and y coordinates of the pixel and not_edge is set true if the pixel is within the triangle. This enables the algorithm to mark each pixel as an object or an edge. Then, each of the object and edge pixels are streamed in real time to the function create_edge_map(), where the binary map of the edges are constructed. The pseudo code for the relevant section of triangle_set() is shown below:

```
void triangle_set() {
```

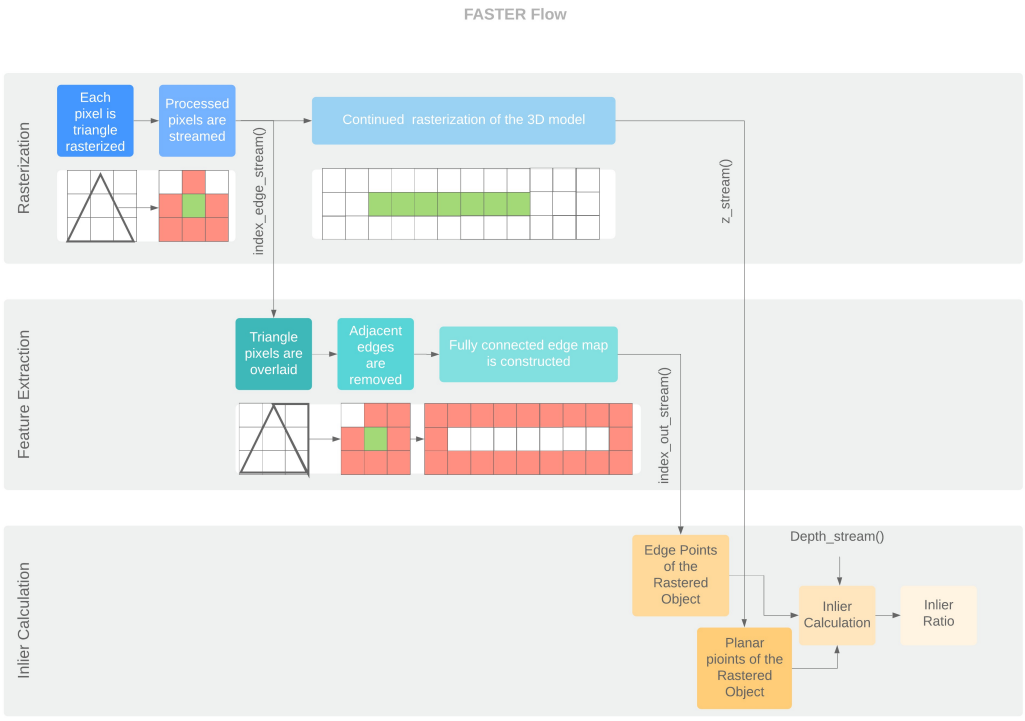


Figure 4.1: Flow of the proposed FASTER implementation, parallelized portions are vertically aligned. The object model of the sample pose is rasterized through the function triangle_set(). The pixels with planar or edge features are streamed to create_edge_map() immediately for feature extraction. This enables the rasterized object models and trigonal edge maps to be created simultaneously and streamed to depth_substitution(). Finally, the inlier ratio is calculated in the function depth_substitution().

```
...
while looping over every pixel in the bounding box {
    if prev_write is true {
        prev_write = false;
        edge_stream.write(prev);
    } else {
        if prev_is_in_triangle is not curr_is_in_triangle {
            if curr_is_in_triangle {
                curr set object;
                prev set edge;
                edge_stream.wite(curr);
                prev_write = true;
            } else {
                curr set edge;
                edge_stream.wite(curr);
                }
        } else {
            if (curr_in_triangle) {
```

```
                    curr set object;
                    edge_stream.wite(curr);
                }
            }
            prev = curr;
        }
        ...
    }
    edge_stream.write(end_index);
}
```

The simultaneous streaming of the pixels enable feature extraction to run in parallel to the rasterization. Figure 4.1 delineates the FASTER flow, for comparison Figure 4.2 depicts the GRIP flow. We can clearly see that in addition to the computational burden of GRIP's feature extraction stage, it also cannot be parallelized to rasterization. These two inefficiencies of the GRIP flow are what makes the proposed speedup with FASTER possible.

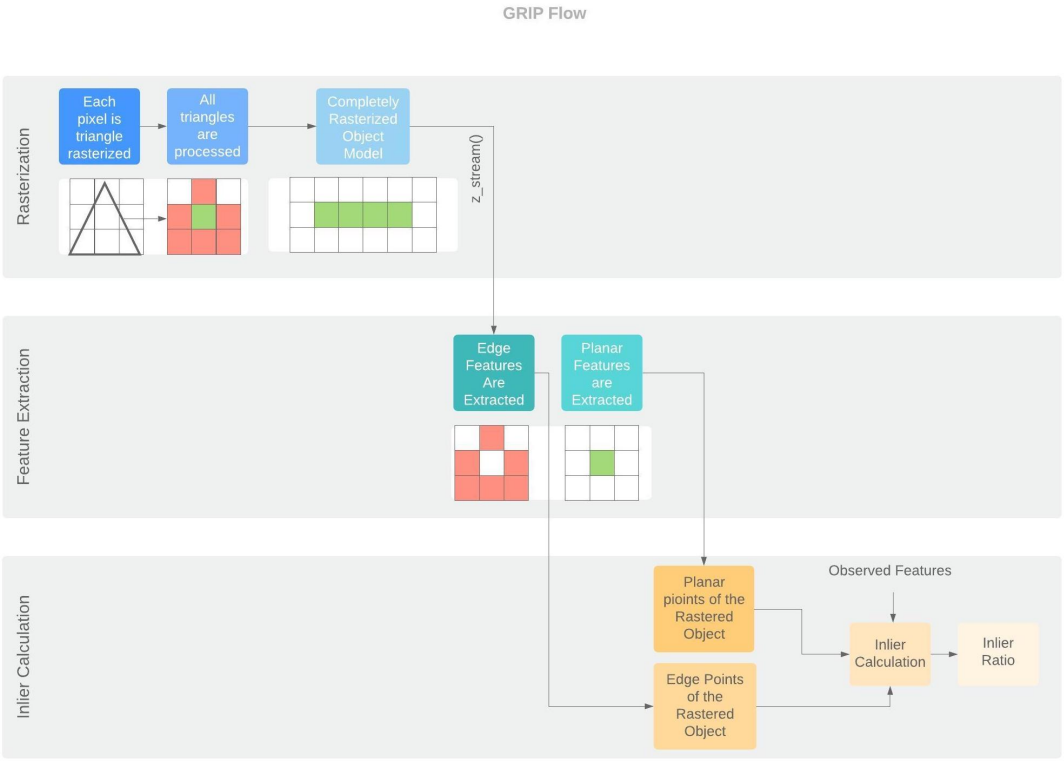


Figure 4.2: GRIP flow, where all three processes of rasterization, feature extraction and inlier calculation are executed sequentially. In addition, feature extraction is much more costly since it is using the methodology described in Zhang et al. [17]

4.1.2 Creating the Edge Map

The function create_edge_map() takes in the unorganized information streamed from triangle_set() and applies logic to merge the triangles with coincident edges. Once all the triangles are processed, and a fully-connected contour edge map is constructed, the edge information is streamed to the function depth_substitution(), where the inlier ratio is calculated. Unfortunately, these two processes cannot be parallelized since create_edge_map() is applying a sequential process on the information it receives and can only output the accurate edge map once all the input pixels are processed. The pseudo code of the final version of the function is shown below. The code will be discussed line by line in Chapter 5.

```
void create_edge_map(index_stream()) {
    u1_Data_T Edge_Map[200][200];
    u1_Data_T Object_Map[200][200];
    index_stream.read(index);
    prev_index = index;
    while index is not end_index {
        if edge_up or edge_down is true {
            if edge_up is true {
                Edge_Map[index.row-1][index.col] set 1;
                edge_up set false;
            }
            if edge_down is true {
                Edge_Map[index.row+1][index.col] set 1;
                edge_down set false;
            }
            prev_index = index

        } else {
            if index is not an edge {
                Object_Map[index.row][index.col] set 1;
                Edge_Map[idx.row][idx.col] set 0;
                if Object_Map[index.row+1][index.col] is not 1 {
                    edge_down set true;
                }
                if Object_Map[index.row-1][index.col] is not 1 {
                    edge_up set true;
                }
            }
            else {
                if (Object_Map[index.row][index.col] is not 1) {
                    Edge_Map[index.row][index.col] set 1;
                }
            }
            index = index_stream.read();
        }
    }
```

```
    for all indices in bounding box {
        output_stream.write(Edge_Map[index.row][index.col]);
    }
    output_stream.write(edge_end_index);
}
```

4.1.3 Calculating the Inlier Ratio

As described in Chapters 1 and 2, the inlier ratio is used to calculate the weight $W(q)$ for proposed pose $q(i)$ to determine the actual pose of the inferred object. The function depth_substitution in the GRIP implementation calculates the numerator and the denominator of each inlier ratio.

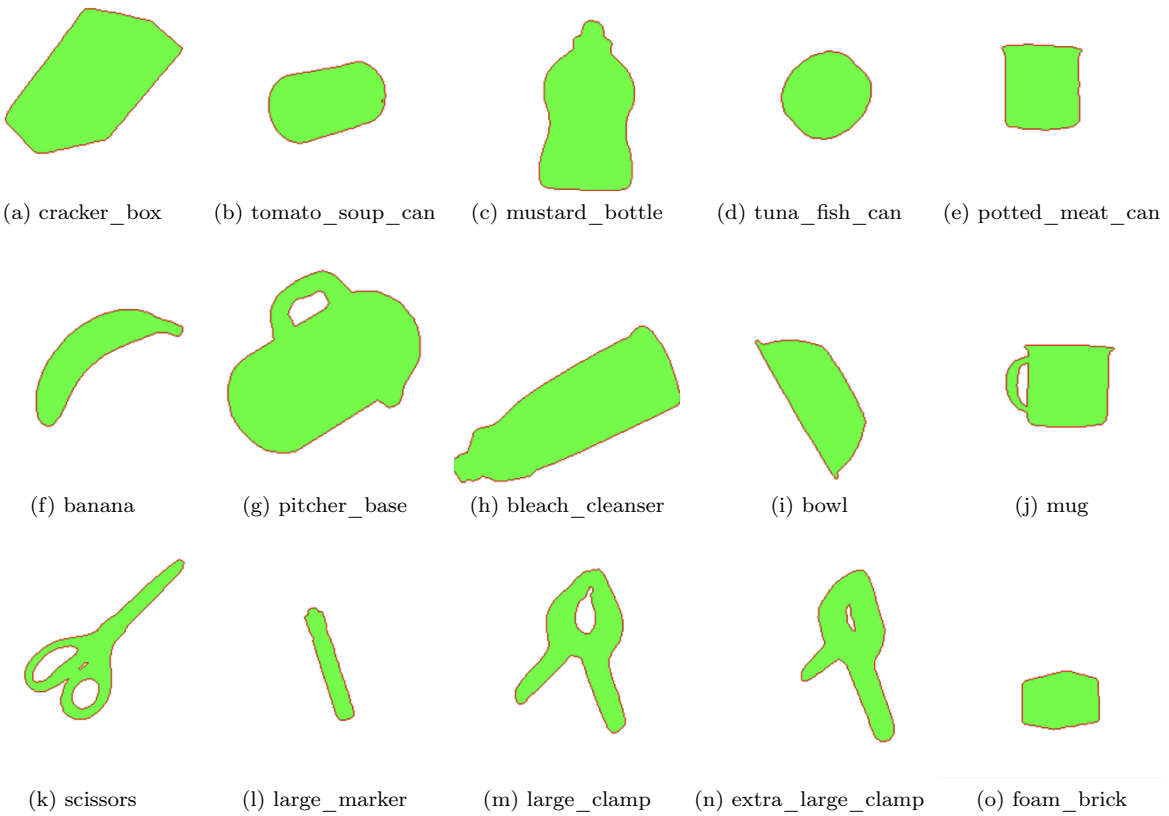
On a high level, the function takes as input the observation scene, rendered depth scene, and the bitmap of the edges of the rendered object. It checks if any of the neighbors of the rastered_edge are valid edge pixels in the actual observation_depth scene. In a depth scene, the pixels that coincide on an object have smaller depth values, while the backround pixels contain the largest possible depth value. Thus, a neighbor is considered to be a valid edge if its depth value is less than a given threshold value. Then, the validity of the inlier is determined by taking the difference of the depth value of the valid rastered edge pixel and the actual depth value of the observed scene and comparing the absolute value of the result to the threshold ϵ. If the result is less than ϵ, the depth values of the rendered_edge and the observed_edge are almost the same, which supports the possibility that the rendered pose at hand might be the actual pose of the observed object. Following is the pseudo code for the described function:

```
void depth_substitution() {
    ...
    Depth_Data_T raster_frange[200][200]; //depth values of the
                                            rastered object
    for every row and col in bounding box {
        u1_Data_T rastered_edge = index_out_stream.read();
        Depth_Data_T observation_depth = Depth_stream.read();
        u1_Data_T edge = Depth_edge_stream.read();

        for every neighbor of raster_frange[row][col] {
            u1_Data_T object_threshold = 2.0;
            if neighbor is less than object_threshold {
                z_difference = neighbor-observation_depth;
            } else {
                z_difference = raster_frange[row][col]-observation_depth;
            }
            z_difference = abs(z_difference);
        }
        Inlier_Data_T inlier_Temp;
        Depth_Non_Norm_Data_T epsilon = 0.008;
        if (z_difference <= epsilon) {
                inlier_Temp = 1;
```

```
                if (rastered_edge and edge) {
                        edge_inlier++;
                }
                inlier_in_BB++;
        }
        if (observation_depth <= Depth_Data_T(3.0)) {
                valid_depth_in_BB++;
        }
    }
    V_D_stream.write(valid_depth_in_BB);
    Inlier_stream.write(inlier_in_BB);
    edge_inlier_stream.write(edge_inlier);
    ...
}
```

4.2 Evaluation



(a) cracker_box (b) tomato_soup_can (c) mustard_bottle (d) tuna_fish_can (e) potted_meat_can

(f) banana (g) pitcher_base (h) bleach_cleanser (i) bowl (j) mug

(k) scissors (l) large_marker (m) large_clamp (n) extra_large_clamp (o) foam_brick

Figure 4.3: Examples of the drawn edge maps with red and object maps with green of each of the 15 object classes.

After the completion of each of the functions described in the Implementation section of Chapter 4, the finalized function was tested within the Vivado HLS suite, before moving onto synthesizing the code. This approach enabled each portion of the project to be validated before moving on with the time-intensive synthesis and FPGA execution stages.

4.2.1 Edge Map Functionality

Classifying the pixels of triangles during rasterization and creating the edge map are complementary processes that can be tested together. In order to test the validity of the constructed edge maps, a new function draw() was used. The function processed the input bit-map and constructed PGM images attributing different colors for specified pixel values. Backgrounds were left white, while red was used for edges and green for object pixels. The draw function was called for 620 different poses of 15 different object classes and each image was scrutinized. The original GRIP paper uses 625 poses at each iteration to reach the reported accuracy level. As the FASTER implementation can process 10 different poses concurrently, 620 total pose samples are used on every iteration. The algorithm terminates after 50 iterations. Figure 4.3 exemplifies a drawing for each object class. Then, each drawing was evaluated based on how successful the algorithm was able to generate a continuous single-pixel width contour edge map.

4.2.2 Inlier Ratio Accuracy

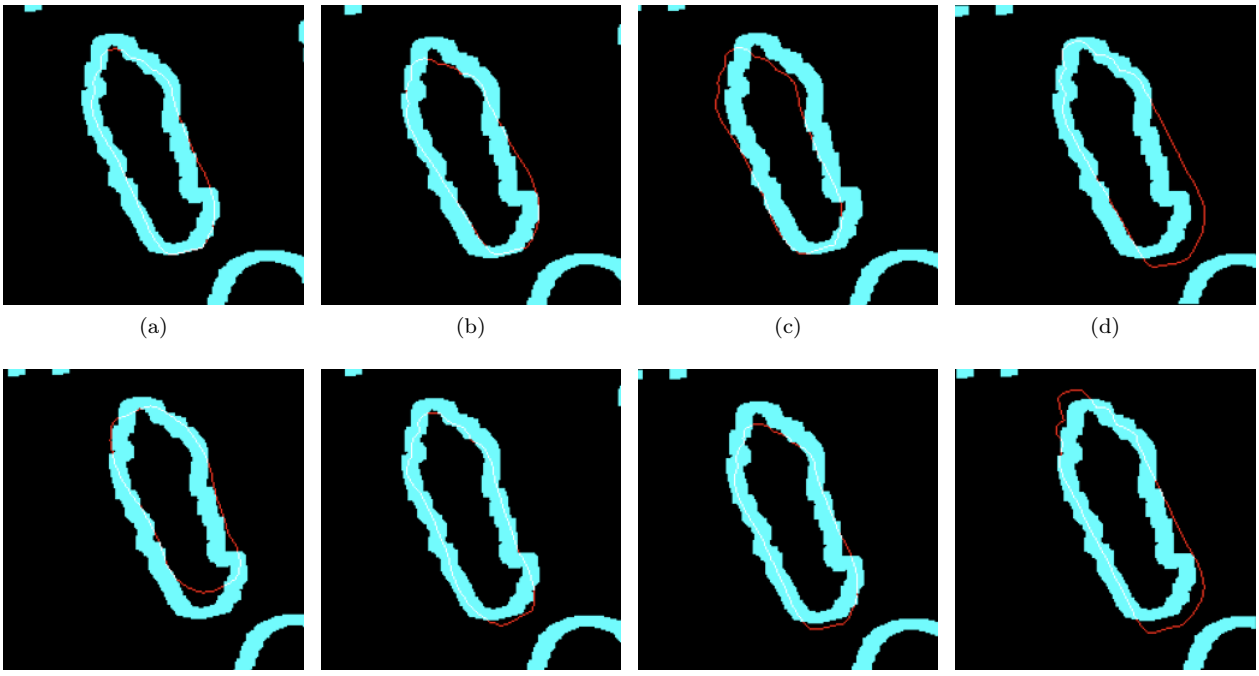


Figure 4.4: Converging results of the mustard_bottle object class after 50 iterations. Although there are slight changes in the contour edge maps of the proposed poses in red, they have converged to the actual pose in the observation scene, whose edge features are marked blue.

The inlier ratio is calculated with the purpose of accurately predicting the object pose. Although the C

simulation of the Vivado HLS Design Suite does not enable us to output the final pose of an object on a given scene, if the second stage of the flow is run for more than fifty iterations, it is observed that the resulting object poses converge around a similar pose to that of the actual object. This phenomena is illustrated in Figure 4.4. Each object among the fifteen classes in the YCB dataset were examined for convergence before proceeding with actual testing on the FPGA.

4.2.3 Testing on the FPGA

The updated flow was synthesized on Vivado HLS and implemented on a Xilinx Virtex UltraScale FPGA ZCU102 board.[1] With a 200 MHz clock frequency, the whole flow was tested for accuracy, speedup, power and resource consumption. The results obtained from our proposed method were compared to the results of the GRIP paper, and a straightforward FPGA implementation of the GRIP paper. The details of the results will be discussed thoroughly in Chapter 6.

[1]Due to restrictions caused by the COVID-19 pandemic, there was limited access to the FPGA board in the final months of this research project. As a result, the synthesized solution was uploaded onto the ZCU102 board and tested by PhD student Yanqi Liu. This project would not have been complete without her help and efforts.

Chapter 5

Challenges

The theorized approach to extract edge features during the rasterization process is a novel perspective on edge detection. Thus, the accurate implementation required several iterations and refactoring along the way. This section describes in detail the theorized algorithms, the challenges encountered in their implementations, and the final resulting flow.

5.1 Thresholding the Edge Function

Given the nature of the edge function described in section 2.4, it will output a 0 if the input pixel is located on an edge. This phenomenon inspired us to check the edge function output for each vertex of the rasterized triangles and mark each pixel in the region of interest as an edge or not edge. Streaming the marked values to a simultaneous process, a map of the resulting polygon's edges could be identified. Here is the pseudo code for this implementation:

```
void create_edge_map(stream) {
    u2_Data_T Edge_Map[200][200];
    stream.read(index);
    for each index {
        if Edge_Map[index.row][index.col] is 0 {
            Edge_Map[index.row][index.col] = 1;
        }
        else if Edge_Map[index.row][index.col] is 1 {
            Edge_Map[index.row][index.col] = 2;
        }
    }
}
```

Function create_edge_map(), takes an index stream as one of its input arguments. The stream carries the indices of the pixels with 0 edge function result. Then, for each row and column pair, the 200x200 2-bit Edge_Map BRAM stores information indicating whether the pixel is an unknown, edge or not-edge. Consequently pixels that are streamed as edges once, remain as edges. However, the pixels that were streamed more than once have to be positioned on the coinciding edges of two neighboring triangles. The pixel would

then be marked with a different index, 2, as this implies that the pixel lays within the object being rendered and it is strictly not an edge.

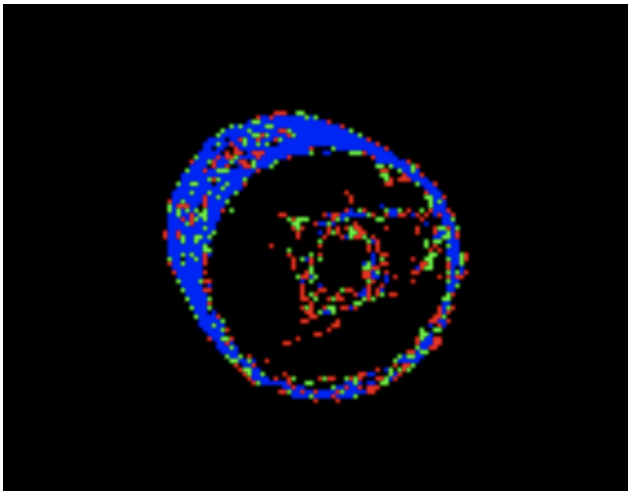


Figure 5.1: Output using our approach with an edge function close to 0. Each color represents the set of points which had an output close to 0 for one of the three vertices on the rastered triangle. It is observed that the blue points mostly cover the surface of the cylinder with rectangular cross-section, while green and red points are scattered in the middle of the surface of the cylinder with circular cross-section. As the ground truth, both of these locations only contain planar points, thus points classified as edges on these locations are false positives

Using the edge function output to determine whether a pixel is an edge was a sound theory. However, it did not work as expected. Each pixel on the image does not directly coincide with the edges of the triangles because the edges of triangles are straight lines in point space. It was realized that because the pixels are in raster space, there is a discrepancy between edge lines and pixels. The edge function is not able to output an exact 0 for any of the pixels since none of them are ever exactly on an edge line. Consequently, the condition to accept a pixel as an edge was altered. Modifying the acceptance criteria to classify pixels with edge function output close to zero as edges, we were able to produce the output in Figure 5.1.

Proceeding the initial attempt to detect edges accurately, it was theorized that carefully chosen thresholds could fine tune the edge functions of each vertex of a given triangle. This would eliminate the false positive edge pixels in Figure 5.1. In an attempt to find a correlation between the marked edges and the edge function outputs of vertices, a number of different threshold combinations were evaluated. Various thresholds were selected for each vertex's edge function. The pixels that satisfy the new acceptance criteria were streamed as edges to the same create_edge_map() function described in Section 4.1.

The edge outputs are visualized in Figure 5.2. Although this method performed slightly better than the previous attempt, the edges were too thick and the false positive rate was still higher than desired. Consequently, the output was not accurate enough to be used for the inlier ratio calculation. Again, the color coding was done in order to understand the correlation between the different edge features and the edge function of each vertex. After a significant amount of trials, no correlation was found and this method was rendered unacceptable for our purposes.

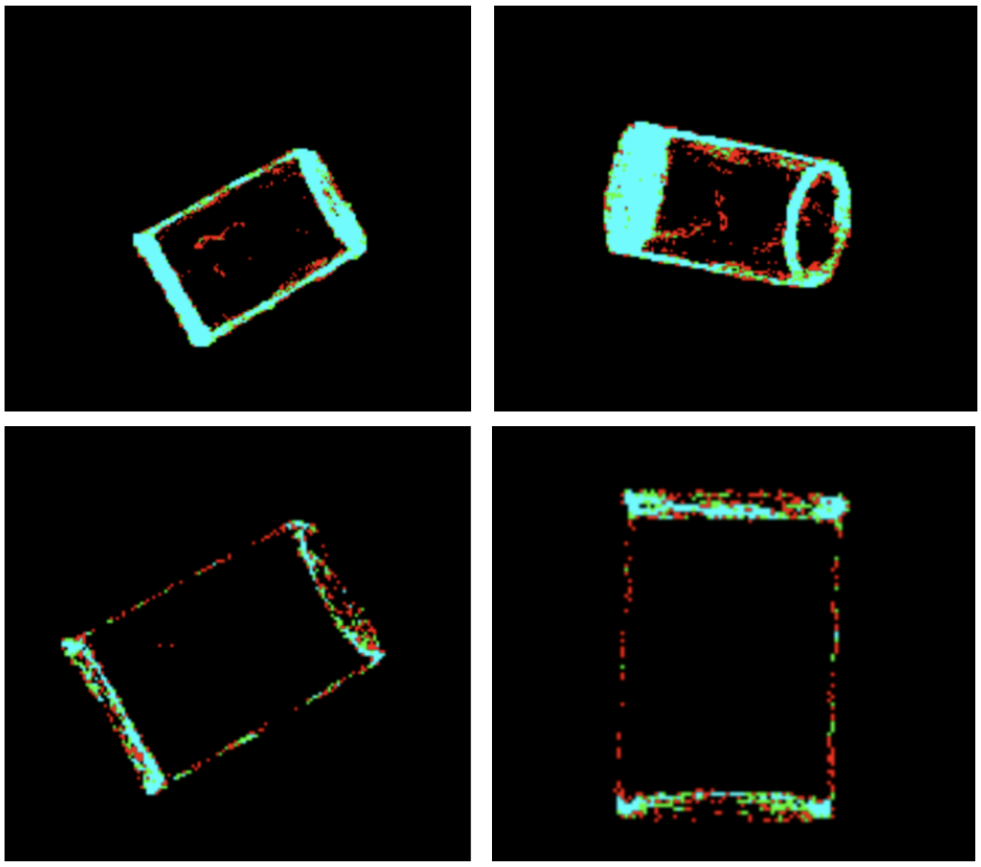


Figure 5.2: Output of the edges detected with thresholding the edge function. The points were color coded to correlate the relation between edge functions and thresholds.

5.2 Previous, Current Pixel Relation

While thresholding the edge function to detect edges during rasterization was not useful, the edge function itself was very precise in detecting the interior regions of triangles. Knowing that a pixel of interest is inside or outside of a rasterized triangle, an algorithm to identify edges could be designed. Because of the zigzag traversal, the edges occur when there is a transition between an edge pixel and a pixel that lays within the object boundaries. The pixels within the triangles' bounds will be referred as object pixels. There are only two occasions when a pixel is an edge. Both instances occur when the previous pixel is an object while the current pixel is not (and vice versa). While iterating through every pixel in the bounding box and marking each object pixel during rasterization one can simultaneously check if the previous pixel is within the triangle. Here is the pseudo code of the algorithm to select the edge pixels to be streamed to create_edge_map():

```
...
while looping over every pixel in the bounding box {
    if prev_is_in_triangle is not equal curr_is_in_triangle {
        if curr_is_in_triangle {
                curr set object;
```

```
                prev set edge;
                edge_stream.write(object);
                edge_stream.write(prev);
        } else {
            curr set edge;
            edge_stream.wite(curr);
            }
        } else {
            if (curr_in_triangle) {
                curr is edge;
                edge_stream.wite(curr);
            }
        }
    prev = curr;
}
...
```

Here, each current and preceding pixel's edge function is used to mark the indices as either an object or an edge before they are streamed to create_edge_map(). The algorithm first checks if the Boolean prev_is_in_triangle is equal to curr_is_in_triangle, where each Boolean is true if its corresponding pixel is within the set of object pixels. Then, if the current pixel is within the triangle, it is inferred that the zigzag traversal just got into the object region. Thus, the current pixel is an object, while the previous is an edge. If the previous pixel was the one within the triangle, it would indicate that the zigzag traversal just exited the object region. So, the current pixel is an edge. In addition, if previous is an edge and current is an object, the current pixel is marked as an object pixel and streamed to create_edge_map() in order to avoid false positives.

The function create_edge_map(), outputs a binary image of the rendered object. After reading the stream containing the object and edge pixel indices, it classifies the intersecting pixels on the boundaries of the coinciding triangles as object pixels. The function outputs a single-pixel width, continuous contour edge map of the complete object:

```
void create_edge_map(index_stream()){}
    u2_Data_T Edge_Map[200][200];
    index_stream.read(index);
    for all indices {
        if index is not an edge {
            Edge_Map[index.row][index.col] is 2;
        }
        else {
            if (Edge_Map[index.row][index.col] is not 2) {
                    Edge_Map[index.row][index.col] is 1;
            }
        }
    }
```

}

The function utilizes a 200x200 2-bit matrix, BRAM Edge_Map[200][200], to store the edge information. Each streamed index marked as either an edge or object is initially checked. If the index belongs to an object pixel, the matrix entry for the pixel is set to a value of 2, which indicates that this pixel was visited and marked as a non-edge. If the index is an edge pixel, it might be marked as an edge but might still be within the boundaries of another triangle. Thus, the pixel is accepted as an edge only if it was not previously stored as an object pixel. Using this simple approach, the results in Figure 5.3 were obtained.

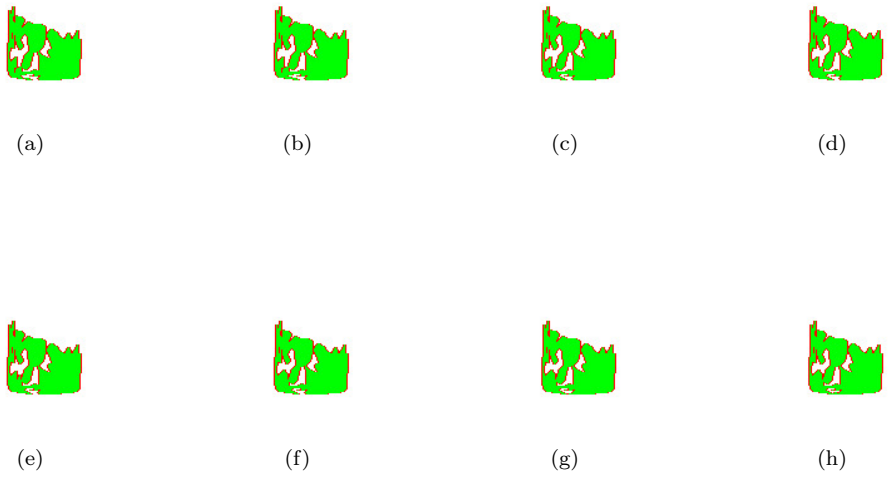
Figure 5.3: Edges detected for the tomato_soup_can after the current and previous pixel relation is implemented as described.

Although the results were much better than previous attempts, the horizontal edges were lacking a significant number of pixels, which is problematic for the inlier ratio calculation. In particular, there were no false positives but many false negatives. A pixel is considered a valid inlier, if the depth value of the rastered edge pixel matches the z_channel of the corresponding pixel in the observed scene. Missing vertical edges result in the failure of identifying valid inliers. Figure 5.3 shows that the percentage of missing edge pixels depend heavily on the orientation of the object pose. Because of its orientation, the sample pose on the left has demonstrated higher accuracy while detecting edges. The poses with less false negatives contain a higher number of detected edge pixels that can match with the edges of the observed scene. This introduces an unfair bias towards sample poses with less false negatives, as they will have a larger number of valid inlier candidates. Consequently, a lot of time and effort was spent on resolving this issue.

5.3 Upper and Lower Edge Bounds

To improve our algorithm better, we needed to identify the flaw in our logic that failed to accurate detect vertical edges. Thus, each iteration of the process was visualized and examined. Certain frames of this visualization are displayed in Figure 5.4. These visuals show that due to the very nature of the zigzag traversal, the algorithm only detects edges when there is a horizontal change between an edge pixel and an

object pixel, thus the vertical edges are missed. There are many possible solutions to this problem; however, most of them require trade offs with speed or memory usage. Since the purpose of this study is to achieve accelerated edge feature detection without increasing the memory or power consumption significantly, a new method was theorized to identify the vertical edge pixels with minimal overhead.



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 5.4: The figure illustrates how the gap at the bottom of the proposed pose is filled with object pixels, while the edges are missed. Lower bounds of the object end up with an incomplete edge set due to the deficiency of this version of the algorithm to identify the state changes among vertical neighbors.

The function create_edge_map() first checks if the streamed index is marked as an object pixel independent of what the pixel is represented as in Edge_Map[200][200]. Relying on this aspect of the algorithm, if each object region is completely encapsulated by vertical and horizontal edges as it is marked on the Edge_Map[200][200], the final object is bound to be confined with a fully connected edge. In order to accomplish this functionality, the create_edge_map() function was updated to the following version:

```
void create_edge_map(index_stream()){
    u2_Data_T Edge_Map[200][200];
    index_stream.read(index);
    for all indices {
        if index is not an edge {
            Edge_Map[index.row][index.col] set 2;
            if Edge_Map[index.row+1][index.col] is not 2 {
                Edge_Map[index.row+1][index.col] set 1;
                }
            }
            if Edge_Map[index.row-1][index.col] is not 2 {
                Edge_Map[index.row-1][index.col] set 1;
            }
        else {
```

```
                if (Edge_Map[index.row][index.col] is not 2) {
                        Edge_Map[index.row][index.col] set 1;
                }
            }
        }
}
```

When a pixel is marked as an object, if the lower and upper neighbors have previously not been classified as objects, they are marked as edges. This is done with the purple highlighted sections of the pseudo code. The logic allows the edge pixels to be overwritten as objects if needed. Thus, as each pixel is sequentially traversed, even if the state of the lower and upper neighbors are unknown at the time, they can be marked as edge pixels. If they are not, then their states will be rectified as they are processed. This results in every sub-region to be bounded by complete edges until they merge into a single object with a single-pixel width boundary. The results of the updated algorithm are visualized in Figure 5.6.

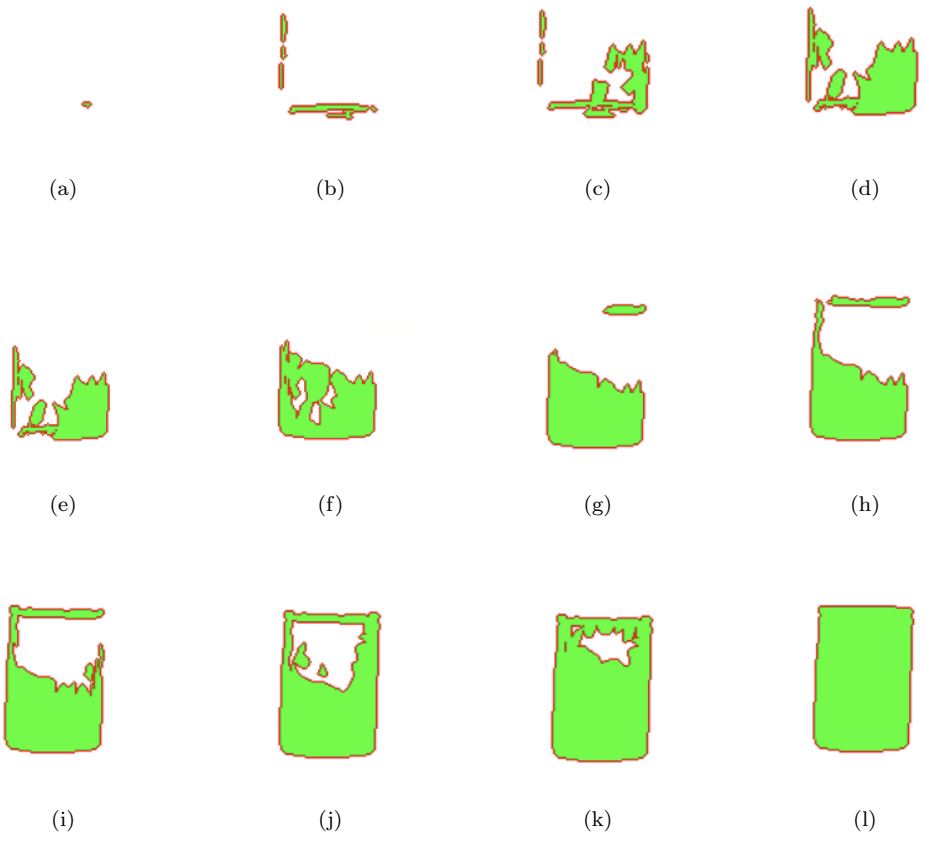


Figure 5.5: Final edge detection algorithm results encapsulating every sub-object at every iteration. Red pixels are edges, while green pixels are object pixels.

5.4 Reducing Dependencies and Memory Accesses for Pipelining Loops

Pipelining is a term used in computer architecture to describe how the central processing unit can execute different portions of subsequent instructions in parallel to complete execution of an instruction at the end of every clock cycle. In the absence of pipelining, each instruction takes several clock cycles to complete, which increases the total compute time by multiples. However, due to the very nature of pipelining, two consequitive instrucitons having dependencies result in "stalls". If one of the current instruction's variables depends on the previous instruction, the CPU needs to wait (stall) the current instruction until the processing of the previous instruction is complete.

One of the main benefits of using HLS is having access to commands that empower low level design choices such as which portions of the algorithm to pipeline. With the use of pragmas, each iteration of the loops used in the functions can be pipelined. However, when the solution is synthesized, if there are read and write dependencies between each pipelined loop iteration, the compiler would pipeline every two iterations of the loop. In addition, there are only two available memory ports to BRAMs. Thus if there are more than two read and write accesses to the same BRAM at an iteration, the compiler would need to pipeline every necessary number of iterations to access the memory a sufficient number of times. Every loop iteration that is witheld from pipelining would increase computation time by a multiple. Here is the pseudo code of the function create_edge_map() with problematic areas marked:

```
void create_edge_map(index_stream()){
    u2_Data_T Edge_Map[200][200];
    index_stream.read(index);
    for all indices {
#pragma HLS PIPELINE II=1
        if index is not an edge {
            Edge_Map[index.row][index.col] set 2;
            if Edge_Map[index.row+1][index.col] is not 2 {
                Edge_Map[index.row+1][index.col] set 1;
                }
            }
            if Edge_Map[index.row-1][index.col] is not 2 {
                Edge_Map[index.row-1][index.col] set 1;
            }
        else {
            if (Edge_Map[index.row][index.col] is not 2) {
                    Edge_Map[index.row][index.col] set 1;
            }
        }
    }
}
```

The pseudo code provided above contains read-write dependencies between the green and orange highlights.

The green highlights are reading the same location in memory the orange highlights are writing to. Memory writes need two cycles to complete execution, thus this would prevent the compiler from pipelining as desired. In addition, if all three of the initial if statements are true, there would be three memory writes, which exceeds the limit of two. In order to fix these issues, two 1-bit maps were used. All the reads from Edge_Map[200][200] check whether the pixel is an object. Thus by separating the edge and object information from each other, we can read from one and write to the other at the same time. An Object_Map[200][200] array was created to store object pixels and to be used for the read calls. In addition, edge_up and edge_down variables were introduced to prevent three read calls to the Edge_Map[200][200] array within one loop iteration. However, the compiler was still being too conservative for the flow and was not pipelining as conceptualized. Thus, pragma *dependence* was utilized to enable pipelining manually for each loop.

```
void create_edge_map(index_stream()){}
    u1_Data_T Edge_Map[200][200];
    u1_Data_T Object_Map[200][200];
    index_stream.read(index);
    prev_index = index;
    for all indices {
#pragma HLS PIPELINE II=1
#pragma HLS dependence variable = Edge_Map inter false
        if edge_up or edge_down is true {
            if edge_up is true {
                Edge_Map[index.row-1][index.col] set 1;
                edge_up set false;
            }
            if edge_down is true {
                Edge_Map[index.row+1][index.col] set 1;
                edge_down set false;
            }
            prev_index = index

        } else {
            if index is not an edge {
                Object_Map[index.row][index.col] set 1;
                Edge_Map[idx.row][idx.col] set 0;
                if Object_Map[index.row+1][index.col] is not 1 {
                    edge_down set true;
                }
                if Object_Map[index.row-1][index.col] is not 1 {
                    edge_up set true;
                }
            }
            else {
                if (Object_Map[index.row][index.col] is not 1) {
                    Edge_Map[index.row][index.col] set 1;
```

```
                }
            }
            index = index_stream.read();
        }
    }
}
```

Similarly, HLS allows only one write to the same stream per clock cycle. So the highlighted problematic portions in the triangle_set() function were changed. The two lines highlighted in purple write to the same edge_stream(). This issue was fixed by storing the previous index in a new variable and writing it to the edge_stream during the following loop iteration.

```
while looping over every pixel in the bounding box {
    if prev_is_in_triangle is not curr_is_in_triangle {
        if curr_is_in_triangle {
                curr set object;
                prev set edge;
                edge_stream.wite(curr);
                edge_stream.write(prev);
        } else {
            curr set edge;
            edge_stream.wite(curr);
            }
        } else {
            if (curr_in_triangle) {
                curr set object;
                edge_stream.wite(curr);
            }
        }
    prev = curr;
}
```

The variable prev_write was introduced to purposefully avoid writing to the same stream twice during a single loop iteration. If the previous pixel is an edge while the current is an object, current is streamed to create_edge_map(), while previous is marked to be streamed during the next iteration of the loop. The new version with the appropriate fixes is shown below.

```
while looping over every pixel in the bounding box {
#pragma HLS PIPELINE II=1
    if prev_write is true {
        prev_write = false;
        edge_stream.write(prev);
    } else {
        if prev_is_in_triangle is not curr_is_in_triangle {
            if curr_is_in_triangle {
                curr set object;
                prev set edge;
```

```
                edge_stream.wite(curr);
                prev_write = true;
            } else {
                curr set edge;
                edge_stream.wite(curr);
                }
            } else {
                if (curr_in_triangle) {
                    curr set object;
                    edge_stream.wite(curr);
                }
        }
        prev = curr;
    }
}
```

In this chapter, we have delineated the various approaches taken to prove the functionality of the theorized trigonal edge detection. In addition, explained the process of optimizing the algorithm to pipeline each loop iteration and maximize runtime. After testing the functionality for the 15 YCB classes [15] as described in section 4.2.1, the solution was configured on the Xilinx ZCU102 for testing accuracy, runtime, resource usage and energy consumption.

Chapter 6

Results

The purpose of this thesis is to prove that the proposed FASTER approach demonstrates clear improvements in run time, resource usage and energy consumption while achieving the same accuracy presented in the GRIP approach [5]. The results of FASTER were gathered after running its compiled bit stream at a 200 Mhz clock rate on the Xilinx Virtex UltraScale FPGA ZCU102. Acquired outcomes are compared to the results of different implementations of the whole flow in their appropriate sections for benchmarking.

6.1 Accuracy

(a) Occluded Scene (b) Dark Scene

Figure 6.1: Examples of Occluded and Dark Scenes

Our first goal was to match GRIP's accuracy of pose estimation in dark and occluded environments. The GRIP paper compares its accuracy on the adversarial YCB dataset [15] to those of the other state of the art methods. Each scene in the adversarial YCB dataset contains 5-7 different objects which are placed together. This causes an object clutter and occlusion. In addition, some scenes are taken with limited lighting. Thus, are considered dark scenes. Example scenes are shown on Figure 6.1.

Precision is evaluated based on the flows' pose estimation accuracies on a range of average distance thresholds between predicted and actual poses of 5 different object classes. The maximum error tolerance

is the highest acceptable distance difference between the estimated pose and the real pose, to classify the pose sample as positive. Although the GRIP paper chose 0.04m as the maximum error tolerance, we decided to extended it to 0.1m due to the inherent randomness of the iterated likelihood weighting processes used in both of the flows. ADD and ADD-S metrics [15] are utilized to calculate the pose error for the inferred objects. Each class is tested for 40 different images. For each object class, accuracy represents the percentage of images with positive pose samples for the corresponding distance threshold on the y-axis.

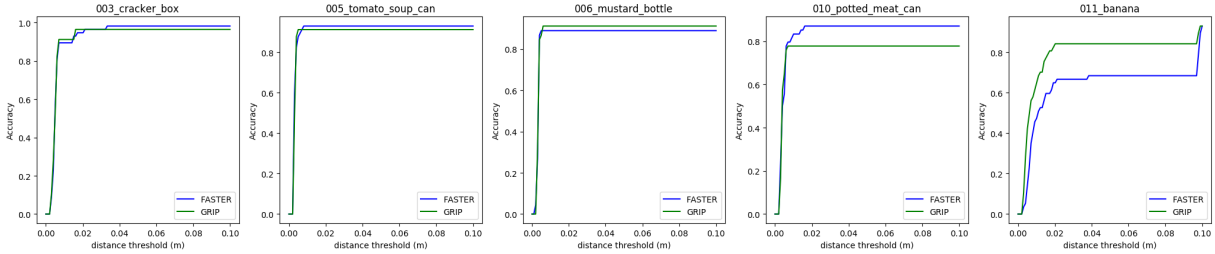


Figure 6.2: Percentage pose estimation accuracy comparison of GRIP and FASTER at various distance thresholds for the 5 different YCB objects in scenes with limited lighting.

The accuracy distance-threshold plots of the objects cracker_box, tomato_soup_can, mustard_bottle, potted_meat_can and banana are plotted on Figures 6.2 and 6.3. The vertical axes represent the percentage accuracy, while the horizontal axes reflect the average distance between the predicted pose to the actual pose. The success of both systems should be compared by the area under the accuracy-threshold metric. It is important to achieve similar levels of accuracy on all average distances ranging between 0 to 1m. Figure 6.2 delineates the accuracy and area under distance-threshold comparisons for dark settings, while Figure 6.3 shows the comparisons for occluded scenes.

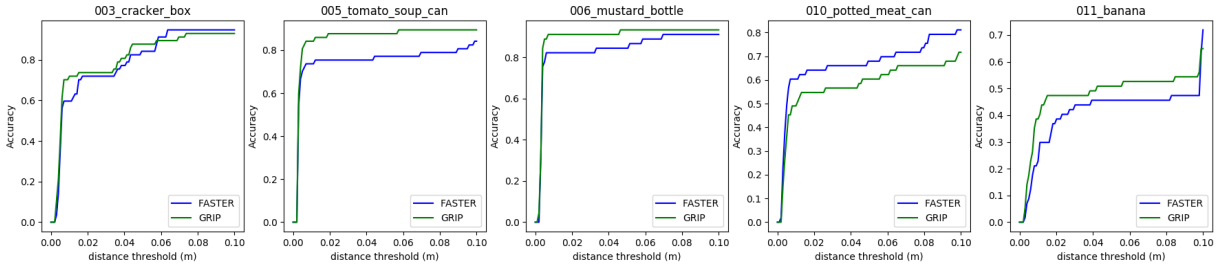


Figure 6.3: Percentage accuracy comparison of GRIP and FASTER at various distance thresholds for the 5 different YCB objects in occluded scenes.

We can see from the figures that the proposed approach performs equally well if not better for every object class except for banana in dark and tomato_soup_can in occluded settings. However, we see that these classes reach to the same level in precision for higher distance-threshold values. This might be due to two reasons. First, because of the randomness of the iterated likelihood evaluation, each flow will output different results for the same object in the same image during different trials. So in a repeated experiment, FASTER might perform better than GRIP for the banana and tomato_soup_can classes. Secondly, contrary to other classes, the curvature of the banana edge features vary immensely according to the view angle. A

larger number of sample poses might be needed to converge to the target pose in lower distance thresholds.

6.2 Speed

The entire GRIP flow is programmed on a CPU + GPU architecture. While the resampling and diffusion stages are executed on an Intel Xeon E5 CPU with 3.0 GHz clock rate, the rendering and feature extraction processes are hosted on Nvidia GTX1080/RTX2070 graphics cards, which have an average of 1.5 GHz base clock frequency. The main motivation behind this study was to offer a simpler and faster alternative to the GRIP rasterization and inlier calculation, which are done in a GPU. Since GRIP focuses on the accuracy of the system rather than the runtime, a straightforward FPGA version of GRIP's feature extraction was implemented and chosen for benchmarking. The simplified FPGA version (called FPGA_GRIP) basically imitates the edge detection methodology described in the GRIP paper on an FPGA.[1] The runtimes of the whole flows on each of the three platforms are compared in this section. The results for the 5 YCB Dataset classes and average runtimes are shown on Table 6.1.

Table 6.1: Runtime Comparison of Complete Flows

Implementation	potted_meat_can	banana	mustard_bottle	cracker_box	tomato_soup_can	Average
GRIP	124 ms	105 ms	144 ms	188 ms	114 ms	135 ms
FPGA_GRIP	97 ms	85 ms	107 ms	160 ms	93 ms	90 ms
FASTER	72 ms	66 ms	72 ms	100 ms	72 ms	76 ms

As shown in the results, our FASTER implementation has achieved 43.7% and 15.6% speedups compared to GRIP and FPGA_GRIP implementations, respectively. It should be noted that the total runtime for the GRIP algorithm includes time spent transferring data to/from the CPU and GPU. As illustrated in Figure 6.4, on average, 32.6% of GRIP's runtime was spent on data transfers on the GPU. Objectively, one of the main benefits of using an FPGA is that all the resources are configured in close proximity on the same fabric, resulting in less time spent for data transfers. Indeed, in our experiments we found that the FASTER implementation achieves significant runtime advantages over the GRIP implementation on the GPU in part because of the more efficient movement of data. However, even when the time allocated for data transfers is disregarded, our proposed FASTER approach achieves 16.5% better runtime on average and performs better for every class. Note that this speedup is achieved despite the fact that the FPGA runs at a 7.5 times reduced clock speed compared to the GPU/CPU implementation.

6.3 Resource Usage

Table 6.2 compares the resource usage of the proposed approach with that of FPGA_GRIP. It would not be fair to collate GRIP's resource utilization to that of FASTER's because the GRIP flow completes rasterization and inlier calculation on an external GPU. Almost the same number of flip flops and lookup tables are used

[1]Due to restriction in accessing FPGA and GPU resources brought about by the COVID-19 pandemic, both benchmarking implementations were compiled by Yanqi Liu (PhD student in the Bahar Lab).

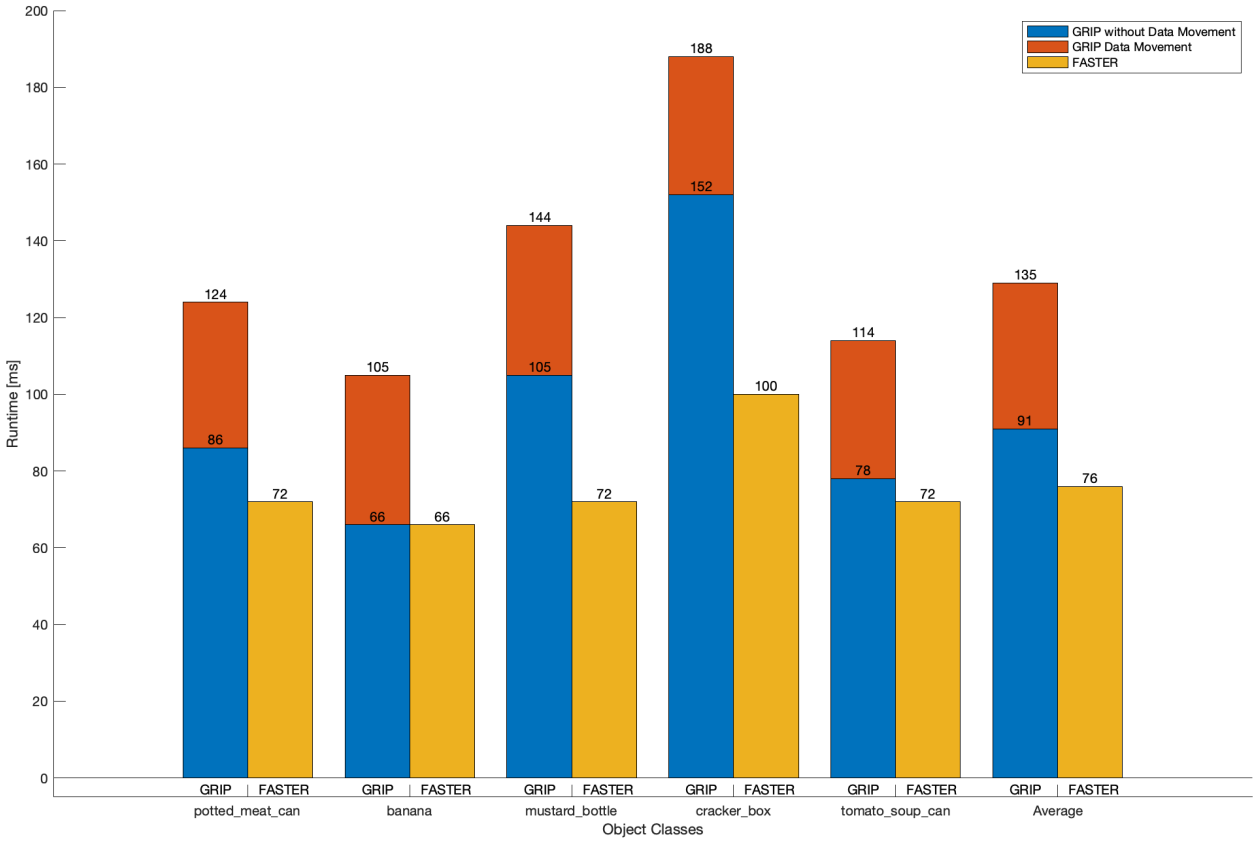


Figure 6.4: Runtimes of End-to-End Flows for each Object Class

for FPGA_GRIP and FASTER. However, the proposed approach uses 10.7% more BRAMs and 6.5% more digital signal processing slices. The supplementary BRAMs are due to the utilization of two additional 200x200 1-bit bitmaps, Edge_Map and Object_Map.

Table 6.2: Resource Usage of the Rasterization and Inlier Calculation

Implementation	BRAM	DSP	FF	LUT
FPGA_GRIP	374	614	189829	210472
FASTER	414	654	193509	212371
% Increase	10.7%	6.5%	1.9%	0.9%

On the other hand, digital signal processing units are generally allocated for heavy computations such as multiplications and divisions. Since FASTER not only gets rid of the heavy computations of the previous approach, but also uses no arithmetic logic other than addition and subtraction, a 6.5% increase in the DSP utilization was not expected. The Xilinx UltraScale DSP Slices document states that "the DSP resources enhance the speed and efficiency of many applications beyond digital signal processing, such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory-mapped I/O registers [16]." Although the DSPs are normally used for more complex tasks, the flow utilizes a considerable amount of LUTs but not DSPs, so the compiler is choosing to use the extra 8 DSPs for simple tasks such as memory

address generation and bus shifting. Unfortunately, FASTER has failed to use less resources compared to GRIP.

6.4 Power Consumption

Table 6.3: Power and Energy Consumption of the Rasterization and Inlier Computation

Implementation	Power	Energy
GRIP	174.538 W	235.6 J
FPGA-GRIP	4.249 W	382.4 mJ
FASTER	3.852 W	290.7 mJ

Power and energy comparisons among GRIP and the two FPGA implementations are shown in Table 6.3. Although our FASTER approach utilizes more resources than FPGA_GRIP, we still dissipate 9.3% less power. The Vivado Suite approximates the switching rate of the mapped hardware to calculate the power consumption of the system. Thus, the gains should be attributed to the lower switching rate of the implemented design.

Table 6.4: Percentage Decrease in Power and Energy Consumptions

	Power	Energy
% Decrease (GRIP)	97.8%	98.8%
% Decrease (FPGA_GRIP)	9.3%	24.0%

On the other hand, GPUs are much more power hungry compared to FPGAs. Consequently, FASTER achieves the same level of accuracy as GRIP by using only 2.2% of the power. A decrease in power utilization also leads to an even greater reduction in energy consumption since both runtime and power are reduced in our implementation. As shown in Table 6.4, FASTER consumes 98.8% and 24.0% less energy compared to GRIP and FPGA_GRIP respectively.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This study theorized that Trigonal Edge Detection in a task that already uses rasterization implemented on an FPGA would perform better in runtime, resource usage and energy consumption. After many iterations on the algorithm, it was demonstrated that the theorized approach performs equally in accuracy to that of its predecessors. We also showed that the proposed approach of Trigonal Edge Detection and inlier calculation, when compared to the GRIP paper and the FPGA-GRIP implementation, performs better in speed, power and energy but worse in resource usage. All of the metrics that were mentioned in the results section are equally important in robotic grasping. However, for certain applications, speed might have a priority over resource usage. Speedups of 43.7% and 15.6% are significant while operating in milliseconds. Although the described FASTER approach was implemented specifically for the two stage approach presented in the GRIP paper [5], it would be beneficial in most cases that require object rasterization and FPGA usage.

7.2 Future Work

Section 5.4 elaborates on reducing dependencies and memory accesses to pipeline more loops. Although most of the loops were pipelined optimally, some were left untouched due to time constraints. An example of such a loop is within the depth_substitution() function described in Section 4.1.3. It requires 4 memory reads from the same BRAM, which would require more ports than the available two to pipeline every loop iteration and write to the memory at the same time.

The reason why 4 neighbors of the detected edge pixels are accessed in depth_substitution() is because each edge pixel classified in create_edge_map() are the non-object pixels adjacent to the border object pixels within the object boundary. This results in the detected edge pixels of the rasterized object model to have the maximum depth value as the pixels do not correspond to any point on the object. So when the depth value of the rendered edge is compared to the depth edge in depth_substitution() to calculate the inlier ratio, the algorithm first needs to find the adjacent object pixel, which is one of the 4 neighbors of the rasterized edge in hand.

The aforementioned problem can be fixed by tweaking the create_edge_map() function to mark the boarder object pixels within the object boundary as the edge pixels. The scope of this research did not allow the proposed edit and similar fixes to be made in time. However, for future work, we aim to resolve all dependencies and memory accesses to achieve maximum possible throughput.

Appendix A

Codes of the Implemented Functions

A.1 Triangle Set

```
void triangle_set(stream<Vertice_final> & Vertice_final_stream,
ObjectName Obj_Under_Test, stream<bound_Data_T> &boundary_stream,
stream<Vec_tru_Area_Data_T> & Area_stream,
stream<Depth_Non_Norm_Data_T> & z_stream,
stream<position> &index_stream, stream<edge_position> &index_edge_stream,
stream<u2_Data_T> & comand_stream,
stream<Center_SAMPLE> & Center_SAMPLE_stream_tri,
stream<size_BB> & Size_BB_stream_tri){
// respect to the axis
// forward is ---> (go to the next)
// backward is <--- (go to the previous)
intervals row_BB;
intervals col_BB;
size_BB current_Size_BB;
Center_SAMPLE current_Centre_SAMPLE;

for (u10_Data_T i = 0; i < Num_sample_x_raster_core; i++) { //Num_sample_x_raster_core
//current_Size_BB = Size_BB_stream_tri.read();
current_Centre_SAMPLE = Center_SAMPLE_stream_tri.read();
size_BB bbox = Size_BB_stream_tri.read();
flat_loop: for (u23_Data_T x = 0; x < tri_num_[Obj_Under_Test]; x++) {

Vertice_final V[3];

V[0] = Vertice_final_stream.read();

V[1] = Vertice_final_stream.read();

V[2] = Vertice_final_stream.read();
```

```
/// RASTER STANDARD
u2_Data_T V_col_min;
u2_Data_T V_col_max;
u2_Data_T V_row_min;
u2_Data_T V_row_max;
bound_Data_T col_min = find_min(V[0].x_px, V[1].x_px, V[2].x_px,
V_col_min);
bound_Data_T col_max = find_max(V[0].x_px, V[1].x_px, V[2].x_px,
V_col_max);
bound_Data_T row_min = find_min(V[0].y_px, V[1].y_px, V[2].y_px,
V_row_min);
bound_Data_T row_max = find_max(V[0].y_px, V[1].y_px, V[2].y_px,
V_row_max);
// check if all of the triangle is outside the boundaries
if (col_max >= WIDTH || row_max >= HEIGHT || col_max < 0
|| row_max < 0 || col_min >= WIDTH || row_min >= HEIGHT
|| row_min < 0 || col_min < 0) {
Area_stream.write(0);
} else {
// OPTIMISED FOR THE Wi EVALUaATION NOT FOR AREA
Vec_tru_Area_Data_T Area = orient2d(V[0], V[1], V[2]);
Area_stream.write(Area);
if (!Area.is_neg() && !Area.is_zero()) {
position Index;
comand_stream.write(Read_area);
Index.actual_row = V[V_row_min].y_px.range(BIT_PIXEL - 1,
BIT_PIXEL - BIT_PIXEL_INT);
Index.actual_col = V[V_row_min].x_px.range(BIT_PIXEL - 1,
BIT_PIXEL - BIT_PIXEL_INT);
////////////////////////////////////
if (row_min < 0) {
row_min = 0;
}
if (row_max > HEIGHT - 1) {
row_max = HEIGHT - 1;
} else {
row_max++;
}
if (col_min < 0) {
col_min = 0;
}
if (col_max > WIDTH - 1) {
col_max = WIDTH - 1;
} else {
```

```
col_max++;
}
if (Index.actual_col > WIDTH - 1) {
Index.actual_col = WIDTH - 2;
}

// THE RASTERISATION ALGORITHM WILL START FROM THE
// COLMIN VERTICE AND SCAN THE TRIANGLE IN ONLY PLAUSIBLE DIRECTIONS
// o------> (x)
// |
// v (y)
// SET STARTING POINT

// SET THE STARTING POINT IN THE MIDDLE OF THE GRID
Vertice_final P_start = { Index.actual_col,
Index.actual_row, 0 };
Vec_tru_Area_Data_T w0_row = orient2d(V[1], V[2], P_start);
Vec_tru_Area_Data_T w1_row = orient2d(V[2], V[0], P_start);
Vec_tru_Area_Data_T w2_row = orient2d(V[0], V[1], P_start);
Vec_delta_Data_T A01 = V[1].y_px - V[0].y_px, B01 =
V[0].x_px - V[1].x_px;
Vec_delta_Data_T A12 = V[2].y_px - V[1].y_px, B12 =
V[1].x_px - V[2].x_px;
Vec_delta_Data_T A20 = V[0].y_px - V[2].y_px, B20 =
V[2].x_px - V[0].x_px;

// Start to set the basic flags
u1_Data_T raster_past_flag = no_raster;
u1_Data_T raster_actual_flag = no_raster;

u1_Data_T raster_direction = keep;
u1_Data_T raster_line = keep;
// striangle_settart going backward

u1_Data_T raster_orientation;

// select the raster direction
if (Index.actual_col == col_min) {
raster_orientation = farward;
} else {
// the max bounding box is +1 so actual col == col max is not possible
raster_orientation = backword;
}
// diagonal disabled
u1_Data_T raster_diagonal = no_diagonal;
```

```
// do a continue
u1_Data_T END_flag = continue_raster;
bool prev_in_triangle = (!w0_row.is_neg() && !w1_row.is_neg()
&& !w2_row.is_neg());
position prev_Index = Index;
edge_position prev_relative_index;
u1_Data_T prev_write = 0;
triangle_set_label0: do {
#pragma HLS LOOP_TRIPCOUNT min=0 max=6000 avg=1000
#pragma HLS PIPELINE
if (prev_write) {
prev_write = 0;
index_edge_stream.write(prev_relative_index);
} else {
// IF THE POINT IS INSIDE THE TRIANGLE RASTER IT !
bool curr_in_triangle = (!w0_row.is_neg() && !w1_row.is_neg()
&& !w2_row.is_neg());
//Calculate relative index of the point
edge_position relative_index;
offset_position current_Offset;
create_offsets(current_Offset,
current_Centre_SAMPLE);
relative_index.actual_row = Index.actual_row
- current_Offset.offset_row;
relative_index.actual_col = Index.actual_col
- current_Offset.offset_col;
relative_index.not_edge=0;

if (curr_in_triangle) {
Depth_Non_Norm_Data_T z_trunc = (w0_row * V[0].z)
+ (w1_row * V[1].z) + (w2_row * V[2].z);
comand_stream.write(Read_d_i);
z_stream.write(z_trunc);
position rel_idx;
rel_idx.actual_row = relative_index.actual_row;
rel_idx.actual_col = relative_index.actual_col;
index_stream.write(rel_idx);
//SET THE RASTER FLAG
raster_actual_flag = yes_raster;


}

if (prev_in_triangle != curr_in_triangle) {
if (curr_in_triangle) {
```

```
relative_index.not_edge = 1;

prev_relative_index.actual_row = prev_Index.actual_row
- current_Offset.offset_row;
prev_relative_index.actual_col = prev_Index.actual_col
- current_Offset.offset_col;
prev_relative_index.not_edge = 0;

index_edge_stream.write(relative_index);
prev_write = 1;
//index_edge_stream.write(prev_relative_index);
} else {
relative_index.not_edge = 0;
index_edge_stream.write(relative_index);
}
} else {//prev == tri
if (curr_in_triangle) {
relative_index.not_edge = 1;
index_edge_stream.write(relative_index);
}
}

prev_Index = Index;
if ((Index.actual_col != col_max)
&& (Index.actual_col != col_min)) {
// Inside this we are sure that we are not on the edge
// so diagonal movements are allowed
if (raster_actual_flag == yes_raster) {
// If I rasterise update the flags
raster_past_flag = yes_raster;
raster_actual_flag = no_raster;
raster_direction = keep;
raster_line = keep;
raster_diagonal = no_diagonal;
} else if (raster_past_flag == no_raster) {
// make sure that actual flag is no_raster
raster_actual_flag = no_raster;
if (Index.actual_col
== P_start.x_px.range(BIT_PIXEL - 1,
BIT_PIXEL - BIT_PIXEL_INT)) {// IN THE STARTING POINT WE WANT TO TEST AT LEAST TWO POINTS
raster_direction = keep;
raster_line = keep;
raster_diagonal = no_diagonal;
}
}
```

```
} else {
if (Index.actual_col
== P_start.x_px.range(BIT_PIXEL - 1,
BIT_PIXEL - BIT_PIXEL_INT)) { // IN THE STARTING POINT WE WANT TO TEST AT LEAST TWO POINTS
raster_direction = keep;
raster_line = keep;
raster_diagonal = no_diagonal;
} else {
// reset the flags for next line
raster_past_flag = no_raster;
raster_actual_flag = no_raster;
// if we hit the wall change line and flip the raster direction
// on edge diagonal movement is not allowed
raster_direction = change;
raster_line = change;
raster_diagonal = no_diagonal;
}
}

if (raster_line == change) {
// THIS TO GO TO THE NEXT row
// WE are forced to flip the raster direction
w0_row += B12;
w1_row += B20;
w2_row += B01;
if (Index.actual_row != row_max - 1) {
Index.actual_row++;

END_flag = continue_raster;
} else {
END_flag = end_raster;
}
P_start.x_px = Index.actual_col;
P_start.y_px = Index.actual_row;
if (raster_direction == change) {
raster_orientation = !raster_orientation;
}
} else {
// we keep the same line
if (raster_orientation == farward) {
// FOR THE NEXT COLUM
w0_row += A12;
w1_row += A20;
w2_row += A01;
Index.actual_col++;
```

```
} else {
// FOR THE PREV COLUM
w0_row -= A12;
w1_row -= A20;
w2_row -= A01;

Index.actual_col--;
}
}
prev_in_triangle=curr_in_triangle;
}
} while (END_flag == continue_raster);
}

}
}
comand_stream.write(End_raster);
edge_position end_index;
end_index.actual_row = -1024;
end_index.actual_col = -1024;

end_index.not_edge = 1;
index_edge_stream.write(end_index);
}
}
```

A.2 Create Edge Map

```
void create_edge_map(stream<edge_position> &index_edge_stream,
stream<u1_Data_T> &index_out_stream, int ID) {
u1_Data_T Edge_Map[200][200] = {};
u1_Data_T Object_Map[200][200] = {};
edge_position idx;
edge_position prev_idx;
u1_Data_T valid_idx;

for (u10_Data_T x = 0; x < Num_sample_x_raster_core; x++) {
for (bound_Data_T i = 0; i < 200; i++) {
for (bound_Data_T j = 0; j < 200; j++) {
#pragma HLS PIPELINE
Edge_Map[i][j] = 0;
Object_Map[i][j] = 0;
}
}
```

```
index_edge_stream.read(idx);
u1_Data_T object = 0;
u1_Data_T edge_up = 0;
u1_Data_T edge_down =0;
prev_idx = idx;
while ((idx.actual_row != -1024 && idx.actual_col != -1024))  {
#pragma HLS PIPELINE II=1
#pragma HLS dependence variable=Edge_Map inter false

if (object || edge_up || edge_down) {
if (object) {
Object_Map[prev_idx.actual_row][prev_idx.actual_col] = 1;
object = 0;
//prev_idx = idx;
}
if (edge_up) {
Edge_Map[prev_idx.actual_row-1][prev_idx.actual_col] = 1;
edge_up = 0;
//prev_idx = idx;
}
if (edge_down) {
Edge_Map[prev_idx.actual_row+1][prev_idx.actual_col] = 1;
edge_down = 0;

}
prev_idx = idx;
} else {
valid_idx = ((idx.actual_row >= 0) &
(idx.actual_row < 200) &
(idx.actual_col >= 0) &
(idx.actual_col < 200));
if (valid_idx) {
if (idx.not_edge == 1) { //stream_read
object = 1; //object_write
prev_idx = idx;
Edge_Map[idx.actual_row][idx.actual_col] = 0;
if (idx.actual_row != 199) {//stream_read
if (Object_Map[idx.actual_row+1][idx.actual_col]!=1) {//object_read
edge_down = 1;
prev_idx = idx;
}
}
if (idx.actual_row != 0) {//stream_read PROBLEM HERE!!
if (Object_Map[idx.actual_row-1][idx.actual_col]!=1) { //object_read
edge_up = 1;//Edge_Map[idx.actual_row-1][idx.actual_col] = 1; //edge_write
```

```
prev_idx = idx;
}
}
}
else {
if (Object_Map[idx.actual_row][idx.actual_col]!=1) { //object_read
Edge_Map[idx.actual_row][idx.actual_col] = 1; //edge_write
}
}
}
idx = index_edge_stream.read();
}
}


for (bound_Data_T i = 0; i < 200; i++) {
for (bound_Data_T j = 0; j < 200; j++) {
#pragma HLS PIPELINE II=1
index_out_stream.write(Edge_Map[i][j]);
}
}

#ifndef __SYNTHESIS__
draw_raster_edge_obj((ap_uint<1>*) Edge_Map,(ap_uint<1>*) Object_Map, "edge_", BB_MAX_SIZE, BB_MA
#endif


}
}
```

A.3 Depth Substitution

```
void depth_substitution(//stream<Center_SAMPLE> & Center_SAMPLE_stream_dep,
stream<Vec_Recip_Area_Data_T> & Area_recip_stream,
stream<Depth_Non_Norm_Data_T> & z_stream,
stream<position> & index_stream, stream<u2_Data_T> & comand_stream,
int ID, ObjectName Obj_Under_Test, stream<Depth_Data_T> & Depth_stream,
stream<ap_uint<1> > & Depth_edge_stream,
//stream<size_BB> & Size_BB_stream_dep,
stream<count_pixel_Data_T> & V_R_stream,
stream<count_pixel_Data_T> & V_D_stream,
stream<Inlier_Data_T> &Inlier_stream,
stream<count_pixel_Data_T> & V_E_stream,
stream<Inlier_Data_T> &edge_inlier_stream,
stream<u1_Data_T> &index_out_stream) {
```

```
#ifndef __SYNTHESIS__
ap_uint<1> raster_edge_frange[200][200];
ap_uint<1> depth_edge_frange[200][200];
for (int row=0;row<BB_MAX_SIZE;row++) {
for (int col = 0; col < BB_MAX_SIZE; col++) {
raster_edge_frange[row][col] = 0;
depth_edge_frange[row][col] = 0;
}
}

#endif
//       Depth_Data_T depth_frange[200][200];
//       ap_uint<1> edge_frange[200][200];
Depth_Data_T raster_frange[200][200];

Depth_Data_T z_dept_frange;
u2_Data_T Current_comand;
Vec_Recip_Area_Data_T Area_recip;
offset_position current_Offset;
position relative_index;
//position absolute_Index;
Depth_Data_T z_local;
Depth_Non_Norm_Data_T z_new;

for (int x = 0; x < Num_sample_x_raster_core; x++) {

Inlier_Data_T inlier_in_BB = 0;
count_pixel_Data_T valid_depth_in_BB = 0;
count_pixel_Data_T valid_raster_in_BB = 0;
count_pixel_Data_T valid_edge_in_BB =0;

Inlier_Data_T edge_inlier = 0;
u1_Data_T in_BB_Flag;
for (int row = 0; row < BB_MAX_SIZE; row++) {
for (int col = 0; col < BB_MAX_SIZE; col++) {
#pragma HLS PIPELINE II=1
raster_frange[row][col] = Depth_Data_T(3.99805);
}
}

Current_comand = comand_stream.read();
depth_sub: while(Current_comand != End_raster){
Area_recip = Area_recip_stream.read();
Current_comand = comand_stream.read();
```

```
while(Current_comand == Read_d_i){
#pragma HLS PIPELINE II=1
z_new = z_stream.read();
relative_index = index_stream.read();
Current_comand = comand_stream.read();

if ((relative_index.actual_row < 0
|| relative_index.actual_row > 199)
|| ((relative_index.actual_col < 0
|| relative_index.actual_col > 199))) {
// if outside the bounds reject
// kept only to be sure
} else {
Depth_Non_Norm_Data_T z_rastered = z_new*Area_recip;
// valid address for the frange
// so valid raster in the frange
if (z_rastered <= Depth_Data_T(2.0)) {
if(z_rastered <= raster_frange[relative_index.actual_row][relative_index.actual_col]){

raster_frange[relative_index.actual_row][relative_index.actual_col] = (Depth_Data_T) z_rastered;
valid_raster_in_BB++;
}
}
}
}

}
//int counter = 0;
//int edge_point = 0;
for (int row = 0; row < BB_MAX_SIZE; row++) {
for (int col = 0; col < BB_MAX_SIZE; col++) {
#pragma HLS PIPELINE II=1
ap_uint<1> edge_raster = index_out_stream.read();
Depth_Data_T Depth_tempor = Depth_stream.read();
ap_uint<1> edge = Depth_edge_stream.read();
Depth_Data_T around_raster_frange = 0;
Depth_Data_T around_raster_frange1 = raster_frange[row+1][col];
Depth_Data_T around_raster_frange2 = raster_frange[row-1][col];
Depth_Data_T around_raster_frange3 = raster_frange[row][col+1];
Depth_Data_T around_raster_frange4 = raster_frange[row][col-1];

#ifndef __SYNTHESIS__
raster_edge_frange[row][col] = edge_raster;
depth_edge_frange[row][col] = edge;
#endif
```

```
if(row ==0 || row == BB_MAX_SIZE -1 || col ==0 || col == BB_MAX_SIZE-1){
continue;
}

if (edge_raster) {
valid_edge_in_BB++;
if (around_raster_frange1 < Depth_Data_T(2.0)) {
around_raster_frange = around_raster_frange1;
} else if (around_raster_frange2 < Depth_Data_T(2.0)) {
around_raster_frange = around_raster_frange2;
} else if (around_raster_frange3 < Depth_Data_T(2.0)) {
around_raster_frange = around_raster_frange3;
} else if (around_raster_frange4 < Depth_Data_T(2.0)) {
around_raster_frange = around_raster_frange4;
}
} else{
around_raster_frange = raster_frange[row][col];
}

if(around_raster_frange < Depth_Data_T(2.0)){
Depth_Non_Norm_Data_T z_depth_difference = (Depth_Non_Norm_Data_T) (around_raster_frange)
- Depth_tempor;

Depth_Non_Norm_Data_T z_depth_abs_difference;
if (z_depth_difference < 0) {
z_depth_abs_difference = -z_depth_difference;
} else {
z_depth_abs_difference = z_depth_difference;
}
Inlier_Data_T inlier_Temp;
Depth_Non_Norm_Data_T thres = ap_ufixed<8,0>(0.008);

if (z_depth_abs_difference <= thres) {
// output 1
inlier_Temp = 1;
if (edge_raster && edge) {
edge_inlier++;
}
inlier_in_BB++;


} else {
inlier_Temp = 0;
}
}
```

```
if (Depth_tempor <= Depth_Data_T(3.0)) {
valid_depth_in_BB++;

}
}

}

////////////////////// EVALUATE INLIER
// must check the division
// it goes above 1 because we rasterise twice some pixels
// because we removed the depth check
V_R_stream.write(valid_raster_in_BB);
V_D_stream.write(valid_depth_in_BB);
Inlier_stream.write(inlier_in_BB);
V_E_stream.write(valid_edge_in_BB);
edge_inlier_stream.write(edge_inlier);


#ifndef __SYNTHESIS__
draw_raster_core((Depth_Data_T*) raster_frange, "raster_frange_", BB_MAX_SIZE, BB_MAX_SIZE, ID, x)
draw_raster_edge2((ap_uint<1>*) depth_edge_frange, (ap_uint<1>*) raster_edge_frange, "edge_and_ra
#endif


}
}
```

Bibliography

[1] Paul E. Black. big-o notation in the dictionary of algorithms and data structures. `https://www.nist.gov/dads/HTML/bigOnotation.html`, 6 September 2019.

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[3] Christoph Burnikel, Joachim Ziegler, Im Stadtwald, and D-Saarbrucken. Fast recursive division, 1998.

[4] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[5] Xiaotong Chen, Rui Chen, Zhiqiang Sui, Zhefan Ye, Yanqi Liu, R. Iris Bahar, and Odest Chadwicke Jenkins. GRIP: generative robust inference and perception for semantic robot manipulation in adversarial environments. *CoRR*, abs/1903.08352, 2019.

[6] Michael S. Floater. Generalized barycentric coordinates and applications. *Acta Numerica*, 24:161–214, 2015.

[7] A.S. Glassner. *An Introduction to Ray Tracing*. The Morgan Kaufmann Series in Computer Graphics. Elsevier Science, 1989.

[8] Thomas Hain and David Mercer. Fast floating point square root. pages 33–39, 01 2005.

[9] David Harvey and Joris Van Der Hoeven. Integer multiplication in time O(n log n). working paper or preprint, March 2019.

[10] Lancelot Perrotte and Guillaume Saupin. Fast gpu perspective grid construction and triangle tracing for exhaustive ray tracing of highly coherent rays. *The International Journal of High Performance Computing Applications*, 26(3):192–202, 2012.

[11] Juan Pineda. A parallel algorithm for polygon rasterization. In *In Proceedings of Siggraph '88*, pages 17–20, 1988.

[12] S. Sakthikumaran, S. Salivahanan, V. S. K. Bhaaskaran, V. Kavinilavu, B. Brindha, and C. Vinoth. A very fast and low power carry select adder circuit. In *2011 3rd International Conference on Electronics Computer Technology*, volume 1, pages 273–276, 2011.

[13] Scratchapixel. Rasterization: a practical implementation. www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-practical-implementation. 25 January 2015.

[14] Patryk Walewski, Tomasz Gałaj, and Dominik Szajerman. Heuristic based real-time hybrid rendering with the use of rasterization and ray tracing method. *Open Physics*, 17(1):527 – 544, 2019.

[15] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes, 2017.

[16] Xilinx. Ultrascale architecture dsp slice user guide. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf. 20 September 2019.

[17] Ji Zhang and Sanjiv Singh. Loam: Lidar odometry and mapping in real-time. In *Proceedings of Robotics: Science and Systems Conference*, July 2014.