Imperial College London

Department of Computing

3D Object Detection for Mobile Robots

by

John B. McCormac

Submitted in partial fulfilment of the requirements for the
MSc Degree in Computing Science of Imperial College London

September 2014

Abstract

A long standing goal within mobile robotics has been the ability to successfully navigate a previously unknown environment. This problem area, known as SLAM (Simultaneous Localisation and Mapping), requires a robot to build a map of its surroundings, while also localising itself within the same map. Traditional SLAM techniques have taken the approach of processing a scene as thousands of individual points. A novel approach called SLAM++ instead processes the scene as a composition of everyday objects, this improves the accuracy of the SLAM algorithm and also provides the robot with useful semantic knowledge of its surroundings. To accomplish this the objects have to be "learned" in a time-consuming offline process ($\approx$ 10 hours per object model).

This project aims to develop a system with a much faster ability to learn and detect objects in a real-world setting. It contributes a rapid three-stage prototype pipeline for mobile robots using RGB-Depth data from a consumer-grade 3D camera, such as the Microsoft Kinect. Stage one is an annotation tool integrated into a state-of-the-art SLAM system (KinectFusion), which enables a user to produce the required training data live from a Kinect device, in as little as one-click per video sequence. Stage two is an accelerated GPU implementation for training a Hough forest, capable of producing an object detector in 90 seconds on a GeForce GTX 780. On a test set which consists of three chair classes, this rapidly trained 'Fast Forest' exhibits an average precision of 61.5% for a single class, and 30.9% when scaled to the full multi-class dataset. Finally, stage three provides an object detection system which uses dense prediction on live data from a Kinect device to detect and localise the 'learned' object. Every stage of this pipeline has used the parallelism of a GPU to accelerate performance.

Acknowledgements

I would like to thank my supervisors Professor Andrew Davison and Dr Zeeshan Zia for their invaluable advice and guidance. They have generously imparted their knowledge and expertise. I am grateful for their confidence in trusting me with this project and sharing their enthusiasm for this exciting field.

Contents

1 Introduction

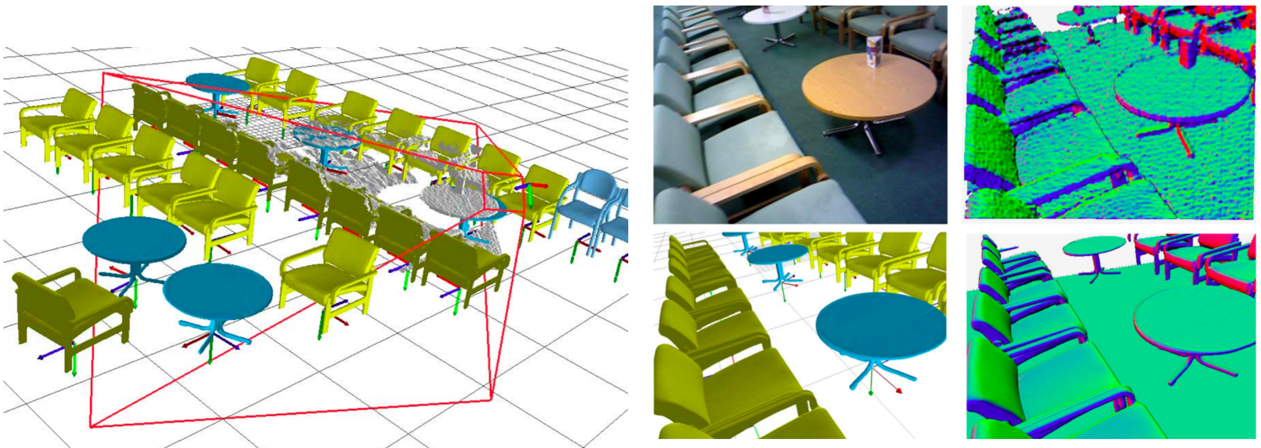
1.1 Motivation

The ability to process visual data into refined semantic and geometric knowledge is relatively straightforward for humans, yet remains extremely difficult to achieve even with today's most powerful computers. The challenge, however, is matched by the potential benefits of a robust solution, with a vast range of applications in fields as diverse as medicine, security, manufacturing, entertainment, exploration, and very directly, mobile robotics.

One of the key goals of mobile robotics is the ability to successfully navigate a previously unknown environment. This problem area, known as SLAM (Simultaneous Localisation and Mapping), requires a robot to build a map of its surroundings, while at the same time localising itself with respect to that map [42]. SLAM is an essential component in tasks ranging from domestic appliances and factory robots, to search and rescue and space exploration.

Traditional approaches to SLAM use many thousands of individual points captured from a camera or laser range finder [21]. In order to accurately determine the location of the robot with respect to the map, the SLAM algorithm needs to iteratively process the entire map representation in terms of these points. To perform this computationally expensive process in real-time, presents significant challenges [16], often requiring approximations to be made, resulting in less accurate estimates.

A novel approach called SLAM++ [48] takes an object-orientated view to mapping. This reduces the map's representation cost and introduces strong prior knowledge, in the form of a single rigid transformation (or pose) for the entire object, instead of tens of thousands of separate points which in reality are rigidly connected. This not only improves the accuracy of the SLAM algorithm itself, but also provides the robot with a useful semantic representation of the objects in its environment. If this representation is sufficiently accurate, the robot can then interact with objects in a meaningful manner, paving the way for the kind of domestic robots long imagined in science fiction, but which have yet to be made a reality. The SLAM++ system has already achieved real-time operating performance (See Figure 1.1), however to accomplish this the objects have to be "learned" in a time-consuming offline process, which takes approximately ten hours per object model.



(a) SLAM++ efficiently tracks and maps a cluttered 3D scene in real-time directly at the object level.

(b) Live RGB and depth images (top), with synthetically rendered objects and high quality scene prediction (bottom).

Figure 1.1: The SLAM++ system operating in real-time. Source [48]

A future goal of this system is the capacity to perform unsupervised real-time object learning and recognition. One way to approach this task is to separate it into two smaller sub-tasks. The first sub-task is to segment the 3D scene into 'probable objects'. This could be made possible by automatic segmentation algorithms, such as by reference to floor and wall planes in an indoor environment [29] or by a more generalised clustering algorithm [18]. Another approach would be to focus on active manipulation and exploration, in the same way infants are known to explore objects from their first year [31]. As will be discussed, the result of this segmentation process has been imitated by building an annotation tool integrated into a state-of-the art SLAM pipeline (KinectFusion [32]), while the project's primary objective is the second sub-task.

The second sub-task is to quickly learn a segmented object, in a robust manner, so that it can be recognised and located within a new context. The learning process should be fast enough to be useful when presented with real-time data, and capable of functioning on only the data available to a mobile robot. It should also be capable of learning and recognising the multitude of classes that may be encountered. The processing power required for this rapid learning has been enabled by recent advances in General-Purpose computing on Graphics Processing Units (GPGPUs), which will form the basis for acceleration efforts.

1.2 Objectives

The core objective is to develop a system capable of rapidly and robustly learning to recognise an object from real-world RGB-D data, for use in a mobile robot. This objective can be split into a number of components:

Training data collection

Data collection requires the development of software capable of capturing and annotating training data similar to the "real world" view available from a mobile robot. The focus of the project is on the learning system itself, therefore this data annotation phase will allow user input to stand-in for other unsupervised object algorithms. This modularity also allows the two phases to be separately investigated. The amount of user input is to be kept to a minimum, and the user experience should be as intuitive and frustration-free as possible.

Optimised training of a Hough forest

The learning system of choice will be a Hough forest for reasons outlined in Section 2. The central aim of the project is to minimise the amount of time it takes to produce a functioning Hough forest [22], by leveraging the power of GPGPUs. Included in this is the requirement of thorough benchmarking to ensure optimisation efforts result in real-world performance improvements.

Live object detection system

Live object detection has a dual purpose; it will ensure that the trained random forest is capable of operating in a real-time system, and will also allow the accuracy of the forest to be measured. There are a large number of parameters that can be tuned when training a random forest, and each of these parameters can affect both the training time and accuracy. A detection system allows for experimentation with these parameters to find a balance between the two competing objectives.

1.3 Contributions

This project contributes a functioning rapid object learning and detection pipeline, by building on Waldvogel's *CURFIL* library [62], and the KinectFusion system [32, 39]. The pipeline is capable of combining raw RGB-D video with user input into a ground truth training dataset, and then can use this data to generate a functioning Hough forest in less than two minutes. It also provides a live object detection system, which uses dense prediction on full resolution RGB-D video from a Kinect device to detect and localise the 'learned' object. Every stage of this pipeline has used a GPU to accelerate performance.

Previous related work by Badami et al. [3] is also capable of accurate 6 Degrees of Freedom (DOF) object detection using a Hough forest. In this work three contributions have been made with the explicit objective of use in a live setting with mobile robots. First, the requirement of a controlled turn table environment to collect the ground truth data has been done away with, instead using the KinectFusion system and minimal user input - potentially just a single click on the centre of an object for an entire video sequence. Second, the accelerated GPU implementation for training a Hough forest (a potential avenue of research proposed by Badami et al. [3]) opens up the possibility for learning new objects in the field and quickly putting them to use. Third is the creation of a ‘Fast Forest’ architecture, capable of being trained in 90 seconds on a GeForce GTX 780, with an average precision of 61.5% on a single class, and 30.9% on our multi-class dataset.

1.4 Report Outline

Chapter 2: Presents some background on training and using Hough forests for object detection. It also looks at previous work that has been done to parallelise training on a GPU.

Chapter 3: Outlines the rationale behind developing bespoke software to capture training data. It also describes the evolution of that software over time, and its basic functionality.

Chapter 4: Gives background detail on the most salient features of the CURFIL library, which formed the basis for the GPU Hough forest implementation. The additions and modification made to this library are described, as well as optimisation efforts to tune it for best performance.

Chapter 5: This chapter presents the object detection system. A description of the leaf node representation is given, followed by the object detection algorithm, and the orientation and class retrieval system. Optimisation efforts to achieve real-time object detection from a live Kinect device are outlined.

Chapter 6: Presents a brief description of the training and testing dataset, which was collected using the software developed in Chapter 3. These datasets were used to measure the performance of the Hough forest implementation.

Chapter 7: The Hough forest is systematically tested on the collected dataset using a wide range of available training parameters. The capacity for the system to operate as a multi-class forest is also evaluated.

Chapter 8: An overall discussion of the results of the analysis in the previous chapter, and an outline of the some of the specific modifications and improvements that could be investigated in the current system. The chapter concludes by exploring the broader possibilities of future research.

2 Background and Related Work

2.1 Random Forests

Random forests have been used extensively in numerous applications, and proven to be fast and robust visual processing systems [47]. Lepetit et al. [36] showed some of the early promise of random forests. Having previously used a K-mean plus nearest neighbour classifier for key point recognition, they successfully migrated to random forests as the classification technique of choice because "it is both fast enough for real-time performance and more robust." Shotton et al. [51] also used random forests in conjunction with the Microsoft Kinect system to recognise human poses in a system which was capable of running at 200 frames per second on consumer hardware.

Originally introduced by Amit and Geman [1], and later extended by Breiman [5], random decision trees act as base predictors in an ensemble method of learning known as a random forest, which utilises a collection of these trees grown independently. A decision tree is a special type of graph consisting of nodes and edges organised hierarchically [10]. Nodes are divided into split nodes, which house a function and a threshold to be applied to the incoming data, and leaf nodes, which store the final prediction values such as a histogram of most likely classes, or spatial hough voting distributions (See Figure 2.1).

The random forest consists of a number of these trees, trained independently, and averages their output to achieve a final combined result. The most commonly used random forests implementations are divided into two well-defined phases, a training phase and a test phase [10]. In the training phase, random forests use a technique known as bagging [6], to reduce the possibility of overfitting, by training a decision tree on a randomly selected subset of the training data. This process has the added advantage of speeding up training time, as only a smaller subset is processed for each tree.

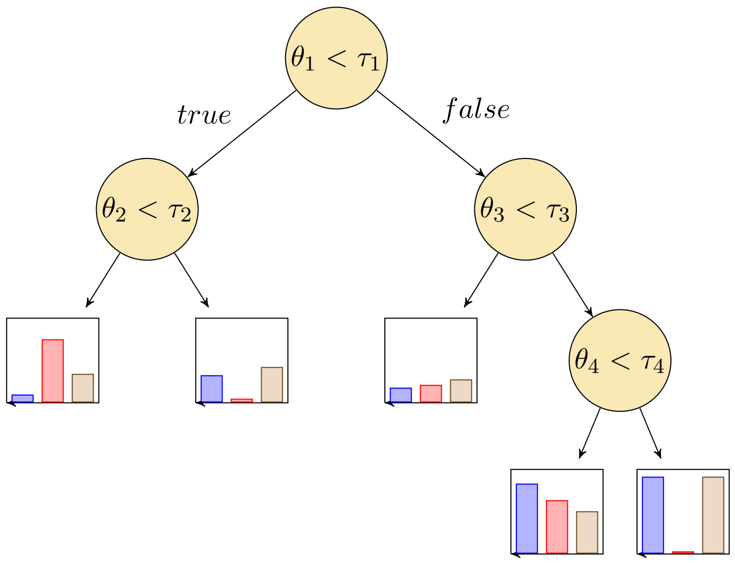


Figure 2.1: A schematic of a random decision tree. Each split node contains a feature function θ (such as intensity difference of two neighbouring patches) and a threshold τ to be applied to incoming data. The leaf nodes contain the distribution (such as object classification) that arrived at them during training. This is then used to classify new test data passed down the tree.

2.2 Hough Forests

Hough forests have proven a successful approach to object localisation [3, 22, 35]. The basis for the Hough forest comes from early work by Duda and Harthas [17] in which they developed a framework (based on a patent by Hough [30]) for detecting lines within digital pictures using a

"Generalized Hough Transform." Leibe et al. [35] drew on this idea to develop a method for detecting and localising any arbitrary object using highly flexible learned representations of the object shape. This representation known as an Implicit Shape Model (ISM) builds up a codebook of the appearance of local features of the object, and their location with respect to the object centre. When a new image is presented, evidence from these features accumulates in a probabilistic Hough voting procedure to infer the most likely location of the object centre.

Gall and Lempitsky [22] extended this idea into the Hough forest. They replaced the generative codebook with a random forest which was successfully used to efficiently map an image patch directly to its corresponding Hough vote. At test time, patches of the image are passed through the forest and maxima in the votes are sought to detect the object. The ability for random forests to cope with noisy input also allowed a bounding box approach to be taken for training data [22], rather than pixel perfect segmentation required by earlier Hough-based methods [35].

More recent work by Badami et al. [3] applied Hough forests to RGB-D data. They managed to both model the probability distribution over class labels and cast votes for the pose of an object within a full 6 DOF, with a high degree of accuracy on a benchmark dataset [34]. The availability of depth data alleviated problems with an object's scaled appearance at different distances, seen in earlier works [35], and the results of their experiments showed significant improvement over colour information alone [3]. Their implementation is CPU based [4], and they noted that for scalability to larger datasets, a GPU implementation would be required.

2.3 Training a Hough Forest

A decision tree is grown using a greedy algorithm which continues splitting nodes starting with the root node until a stopping criterion is reached. Often this stopping criterion is that either all nodes have reached the maximum allowed depth value or splitting them will not produce improved results (for example because all training instances at the node already belong to only one class). The basic algorithm, as described in [61], proceeds as follows:

1. Select a leaf node (via a depth-first or breadth-first search) that is not yet pure.
2. Select the best test that minimizes the data entropy by:
    - 2.1. Random feature candidate generation.
    - 2.2. Feature response calculation.
    - 2.3. Threshold selection.
    - 2.4. Energy model (or offset uncertainty) calculation.
3. Split the leaf node into left and right nodes according to the selected test.
4. Continue from step 1 until the stopping criterion has been reached.

The four sub-parts of step 2 form the core of the training procedure, and is where the majority of the computational expense lies [61]. Each part is described in more detail below.

Random feature candidate generation

A set of features is sampled at random from the available feature space. The most common features used with RGB-D data are the difference between intensity values on pairs of pixels [51], or pixel patches [55], surrounding the test pixel. For a given pixel q, feature generation will randomly choose an offset for each of the neighbouring pairs, the image channel to be used, and potentially the size of the rectangle to average at each offset.

Feature response calculation

In this step the feature responses are calculated for each of the randomly sampled features. Depth normalisation of features which has been shown to be highly beneficial for classification and localisation [3, 55, 61]. For given a pixel q with depth $d(q)$, the selected pair of neighbouring pixels with offsets o_i and o_j and image channels ϕ_i and ϕ_j, the feature response $\theta(q)$ can be calculated as:

$$\theta(q) = \phi_i(q + \frac{o_i}{d(q)}) - \phi_j(q + \frac{o_j}{d(q)})$$

As discussed in Section 4, the basis for the GPU implementation of a Hough forest is the CURFIL library, a result of Benedikt Waldvogel's thesis "Accelerating Random Forests on CPUs and GPUs for Object-Class Image Segmentation." Figure 2.2 illustrates depth invariant features, as depicted within that thesis [61].

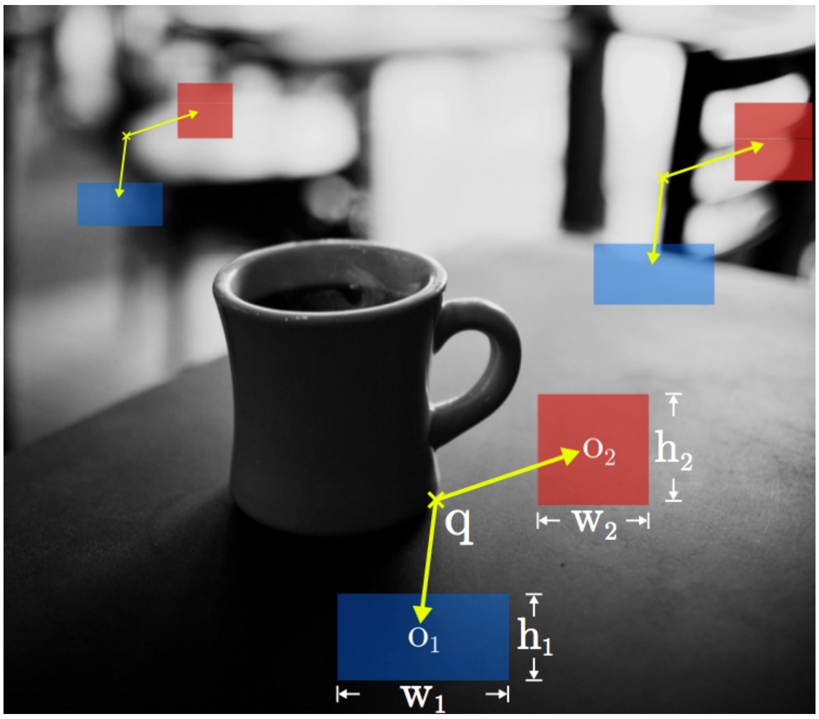


Figure 2.2: An illustration of depth invariant features at three different pixel locations in the CURFIL library. The offset locations o_i and region extents w_i, h_i are normalized with the depth $d(q)$ of the query pixel q. Source [61] Figure 3.3

Threshold selection

To create a binary question from the scalar feature response outlined above, a threshold, τ, must be selected. One method is to sample the feature response from some sensible distribution. Criminisi et al. [10] however suggest a more efficient alternative, to divide the feature responses into bins determined by a number of potential thresholds, and apply each of these bins in a linear sweep when calculating the information gain below. Although selected thresholds can be arbitrarily chosen in advance, a more refined alternative involves an initial pass of the data. In this initial pass, a subset of the responses from the data is sampled to allow a useful distribution of thresholds to be selected.

Best split calculation

For classification the energy model is used to select the candidate features and thresholds which provides the "best split." The basic concepts behind the training objective function are entropy and information gain. The information gain associated with a candidate split is defined as the reduction in uncertainty achieved by splitting the training data arriving at the node into mutliple child subsets [10]. For a subset of data S, which can be split into two subsets S^L and S^R with an entropy measure H, the information gain I is commonly defined as:

$$I = H(S) - \sum_{i \in \{R,L\}} \frac{\mid S^i \mid}{\mid S \mid} H(S^i), \tag{1}$$

where the entropy measure H can vary. For classification problems, the aim for training is to split the data in such a way as to maximise the probability of different classes arriving at separate nodes. For discrete probability distributions the Shannon entropy can be used [10], with $p(c)$ being the empirical distribution extracted from training points in the data S, and C being the set of all possible classes,

$$H(S) = \sum_{c \in C} -p(c) \cdot ln(p(c)). \tag{2}$$

The Hough forest is used for both classification and regression, the energy model only works towards the first of these objectives. To assist with the second objective, it is also useful to minimise the *offset uncertainty* [22]. For a set S with two subsets S^L and S^R containing offset vectors $\mathbf{v}$ to the object centre, the total offset uncertainty, O, is defined as $O = D(S^L) + D(S^R)$ where,

$$D(S) = \sum_{\mathbf{v} \in \{S\}} \left\| \mathbf{v} - \frac{1}{|S|} \sum_{\mathbf{v}' \in \{S\}} \mathbf{v}' \right\|_2 . \tag{3}$$

To combine these two disparate objectives, the original approach of Gall and Lempitsky [22] is to randomly pick one of the tests for each node. This interleaves nodes that decrease class uncertainty with nodes that decrease offset uncertainty, and results in leaf nodes which optimise both. In general, the objectives are chosen with equal probability unless the node is relatively pure (e.g. $> 95\%$ of one class), in which case the node is chosen to minimize the offset uncertainty. Okada [41] suggested an alternative method, starting with the root node as purely classification, and as the purity of a node reaches a given threshold switching to a purely regression objective.

Fanelli et al. [20] compared this kind of approach to the interleaved approach, above, as well as to a third approach which uses node depth d, and a steepness parameter λ to weight the regression objective according to the function $1.0 - e^{-\frac{d}{\lambda}}$. However the results of their experiments showed similar accuracy for all three methods. In this project the original interleaving approach has been maintained. Section 7 details results of experimentation with varying the likelihood of choosing one of these objectives at any given node.

2.4 Object Detection

In testing, an unseen data point $\mathbf{v}$ (such as a pixel in a new RGB-D image) passes from the root node of each trained tree, to leaf nodes, following the binary test functions chosen in training. These functions of the form $\theta < \tau$ compares the feature response θ with a threshold τ to generate a binary decision. If the result is true then the data point proceeds to the left child, otherwise it goes right, as depicted in Figure 2.1. The leaf node contains the numbers of each class that reached that node in training (for estimating the object class), and a representation of the offset vectors (for estimating the object location).

The spatial distribution of offset vectors can be represented in a number of forms. One method is to model it as a Parzen window with a Gaussian kernel [22]. As discussed in more detail in Section 5.1, a fast and simple 3D spatial histogram approach is adopted in this project, with leaf nodes storing accumulated discretised voting vectors.

Each of the reached leaf nodes for the sampled pixel from the test image have their votes collected within a 3D Hough space. Maxima within these votes can then be sought either by a mean shift clustering algorithm or by smoothing the voting space with a Gaussian filter before searching in a greedy manner [3,22,23]. In this project a mean clustering approach is adopted, and discussed in more detail in Section 5.2. For a traditional classification forest the outputs of each of the trees is aggregated using an average. For Hough voting a similar approach is taken, by simply allowing all of the trees to accumulate their votes within the Hough space.

2.5 Multi-Class Considerations

Significant research into Hough forests has focused largely on single class detection systems [3,22]. However, allowing multiple classes to be detected within a single forest comes with a number of advantages. For mobile robots it provides the flexibility to rapidly train a forest on relatively unprocessed data with a mixture of classes, and it also grants the ability to share similar features among classes, resulting in decreased memory requirements at test time [23]. These benefits often come at the cost of reduced accuracy, however two key modifications for training multi-class Hough forests have been developed by Razavi et al. [45] which aims to mitigate this accuracy degradation. The effect of these modifications was analysed in their research via a feature appearance sharing matrix and object taxonomy [45], illustrated in Figure 2.3.

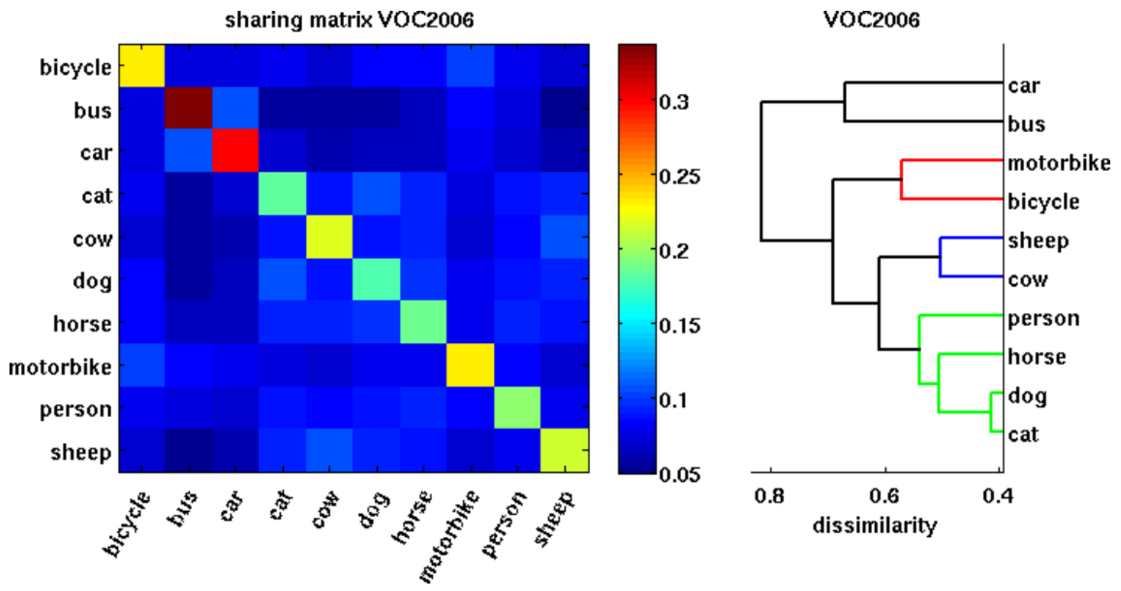


Figure 2.3: Sharing matrix and object taxonomy obtained in [45] for the VOC'06 dataset, when using the modified multi-class training criteria given in Equations 4 and 7. Source [45], Figure 4(a)

The first modification, is a relaxation of Equation 3, to limit minimisation of offset uncertainty to within classes, instead of within the entire set of samples. The effect of this is beneficial for multi-class forests, because it allows objects of quite different shapes but with similar features to 'stick together'. For example, it would be useful for a car and a bus to share the feature of a wheel, however if the offset between a bus wheel and a car wheel to their centre is required to be the same, as in Equation 3, they will naturally be separated. A relaxation of this rule to Equation 4 assists in keeping these taxonomies together while separating the components of each class, where C is the set of classes, and S_c is the set of vectors relating to a given class, c.

$$D(S) = \sum_{c \in \{C\}} \sum_{\mathbf{v} \in \{S_c\}} \left\| \mathbf{v} - \frac{1}{|S_c|} \sum_{\mathbf{v}' \in \{S_c\}} \mathbf{v}' \right\|_2 . \tag{4}$$

As shall be discussed, the test dataset consists of three classes of chairs. The shapes and offsets of these chairs are similar, and their taxonomy tree is very flat, rendering much of the advantage of this relaxation redundant. The additional computational expense and memory requirements of keeping separate offset uncertainties could not be justified, however for future work when exploring a larger more diverse dataset, this relaxation could prove much more beneficial.

The second modification provides an explicit weighting specific to the background class, c_{bg}, to ensure that it is well separated from the rest of the object classes, c_o,

$$H_{bg} = -p(c_o) \cdot ln(p(c_o)) - p(c_{bg}) \cdot ln(p(c_{bg})), \tag{5}$$

before further separating out the positive classes themselves,

$$H_o = \sum_{c \in C, c \neq c_{bg}} -p(c) \cdot ln(p(c)), \tag{6}$$

and then weighting these two with a mixing coefficient λ, which [45] set to $\|C\|$.

$$H_o(S) = H_o + \lambda H_{bg}. \tag{7}$$

This modification requires relatively little additional computation, and could be applicable to our evaluation dataset. It ensures all chair varieties are well separated from the background, before finding fine-grain features more suited to separating out different types of chair.

For detection, Razavi et al. [45] also used the generated taxonomy tree in Figure 2.3 to limit the cast votes within a 4D voting space. Unfortunately, their system required approximately $35s$ for detecting a single positive class in one image, which is approximately two orders of magnitude below real-time performance. In this project, a simplified approach is adopted, in which Hough votes are first cast into a unified 3D Hough space. Maxima within the space are sought, and then a second pass is performed to extract probable classes (and orientations) by inspecting the nodes that contributed towards the maxima.

2.6 Parallelised Random Forest Training on a GPU

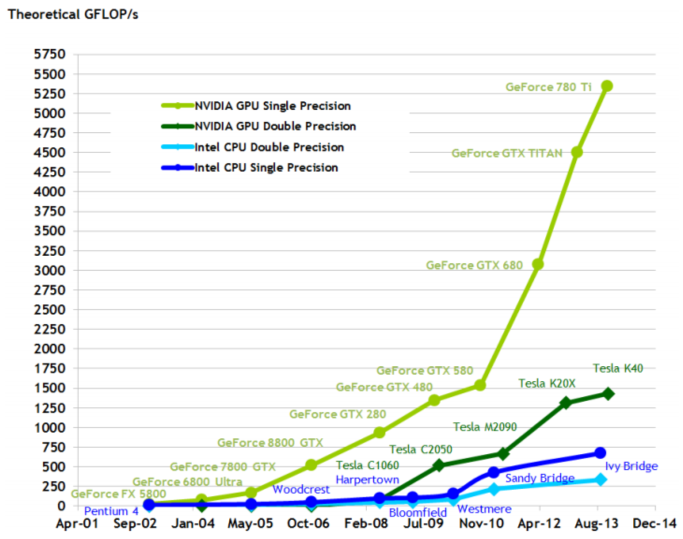
General-purpose computing on Graphics Processing Units

In recent years, fundamental limitations in the manufacturing process of integrated circuits have pushed CPU manufacturers towards multi-core processors [49]. Increasing the *number* of processors operating in parallel, instead of the speed of an individual processor is a technique taken to great lengths within GPUs. GPUs lack the flexibility of CPUs, but excel in performing computationally expensive tasks on large quantities of data by parallel processing in hundreds of independent threads, achieving a significant enhancement in floating point operations per second, and memory throughput (Figure 2.4). This is accomplished by devoting much more space to arithmetic logic units at the expense of more advanced control features (Figure 2.5).

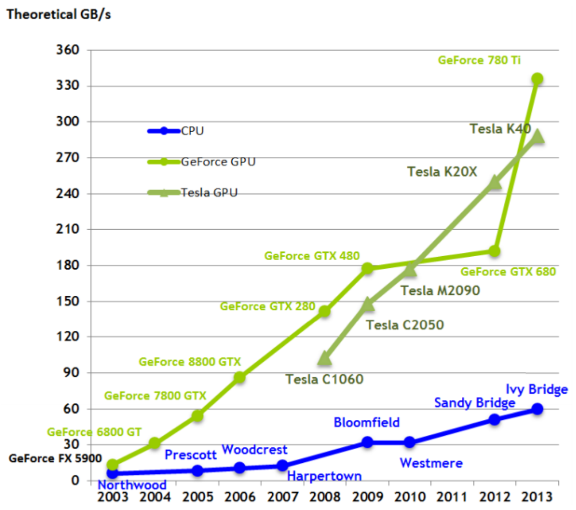
The GPU is purpose built to operate on image data, such as that being processed by a random forest. Historically, this came at the cost of an extraordinarily convoluted process of using the standard graphics APIs such as OpenGL and DirectX in an attempt to perform arbitrary computations. More recently, two general frameworks have been developed to greatly simplify General-purpose computing on Graphics Processing Units, the open source OpenCL standard and Nvidia's proprietary CUDA framework. This project uses the latter primarily because the starting point for the Hough forest implementation is the excellent CUDA Random Forests for Image Labeling (CURFIL) library [62], which (as the name would suggest) has been written in CUDA.

For optimisation efforts care must be given to the different types of memory available. CUDA devices use five different memory spaces, each with strengths and weaknesses. Figure 2.6 shows a simple schematic of the three major memory types within the hierarchy. A short description of the characteristics of each type of memory is given below:

Global memory may be accessed by any thread regardless of its block or grid. It is the largest available memory space, with 3GB in the GeForce GTX 780 used for this project, but also the slowest to access. The access performance can be improved with coalesced loads and stores when a group of concurrently running threads (known as a warp) access sequential memory locations. Global memory is persistent between kernel calls, and is allocated and freed manually.

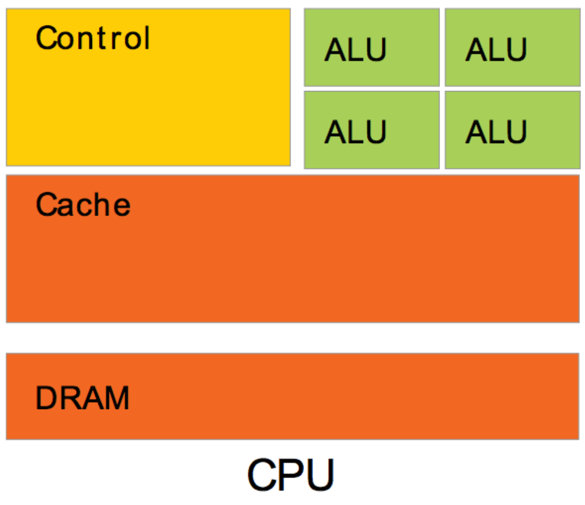


(a) Comparison of theoretical GFLOP/s

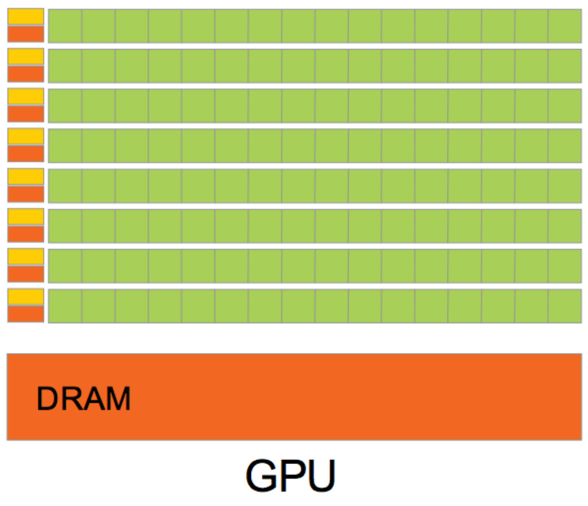


(b) Comparison of memory throughput in GB/s

Figure 2.4: The relative performance of GPU and CPU architectures. Source [13]



(a) The structure of a typical Central Processing Unit.



(b) The structure of a Graphical Processing Unit.

Figure 2.5: The GPU Devotes More Transistors to Data Processing. Source [13]

Texture memory is similar to global memory, except that it is read-only and cached in a way optimized for 2D spatial locality. A texture fetch costs one device memory read only on a cache miss; otherwise, it just costs one read from the texture cache [13].

Constant memory is again read-only memory which is cached, meaning a read only costs a memory read on a cache miss [13]. A total of 64 KB of constant memory exists on a device. The constant cache works best when threads access only a few distinct locations. If all threads of a warp access the same location, then constant memory can be as fast as a register access. However, accesses to different addresses by threads are serialized, causing the cost to scale linearly with the number of unique addresses read.

Shared memory is available to any thread within a given block, and is best viewed as a shared 'scratch pad' for those threads. It is stored on chip and provides much higher bandwidth and lower latency than global memory [13] as long as there are no bank conflicts between the threads. A bank conflict occurs when multiple locations within a single 'bank' of memory are accessed at once and must be serialised. For devices of compute capability 1.x 32-bit words are assigned to successive banks, with 16 banks. For compute capability 2.x and 3.x, devices have 32 banks, with 3.x also having the option of 64-bit word addressing.

Local memory is available to only a single thread. It is not persistent, and is used to store variables that cannot fit into the threads registers. Local memory actually is stored in the global memory space, and is therefore just as slow as a normal global memory access, unless it has been cached.

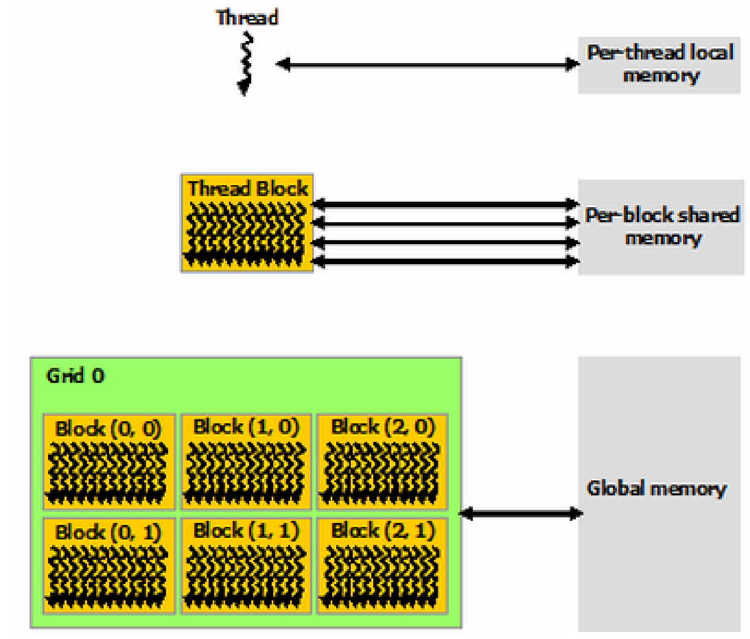


Figure 2.6: An illustration of the hierarchy of memory within an Nvidia GPU from the CUDA programming guide. Modified from Source [13]

Previous Work Parallelising Random Forests

Although potentially fast to both train and test, many results for random forests focus almost exclusively on the *test time* performance of the system [36, 51]. However for large datasets, when experimenting with forest parameters, or for time critical applications, training time plays a significant role. It has often been noted [10,50,61] that random forests have the advantageous capability of being highly parallelisable. With the advent of high power GPGPUs and associated languages such as CUDA, implementing the computationally expensive process of training a random forest on a GPU becomes the natural choice to improve performance.

Before the ready availability of CUDA-like languages, Sharp [50] successfully implemented both training and testing of random forests on a GPU using Microsoft's Direct3D and the High Level Shader Language (HLSL). In the object recognition task, [50] demonstrated the GPU produced predictions identical to that of a CPU but 100x faster. The improvement in training time performance was less substantial but still significant, with an 8x acceleration.

Recent work by Waldvogel [61] focused on a highly optimised CUDA implementation of a random forest for RGB-D label classification, which managed to accelerate training time by a factor of 28 on the NYU Depth v2 dataset [57]. A significant source of the acceleration efforts by Waldvogel [61] came from a detailed breakdown of the relative speeds and caching abilities within the memory of a modern GPU. Storing training images within texture memory reduced runtime by 35%. This speed-up was achieved by maximising usage of the texture cache (a resource of only 6-8kB in size) by sorting feature candidates and increasing the probability of a cache hit. This project combines many of the above techniques developed for use in classification forests, and extends their principles to accelerate Hough forest training with a GPU.

3 Capturing Training Data

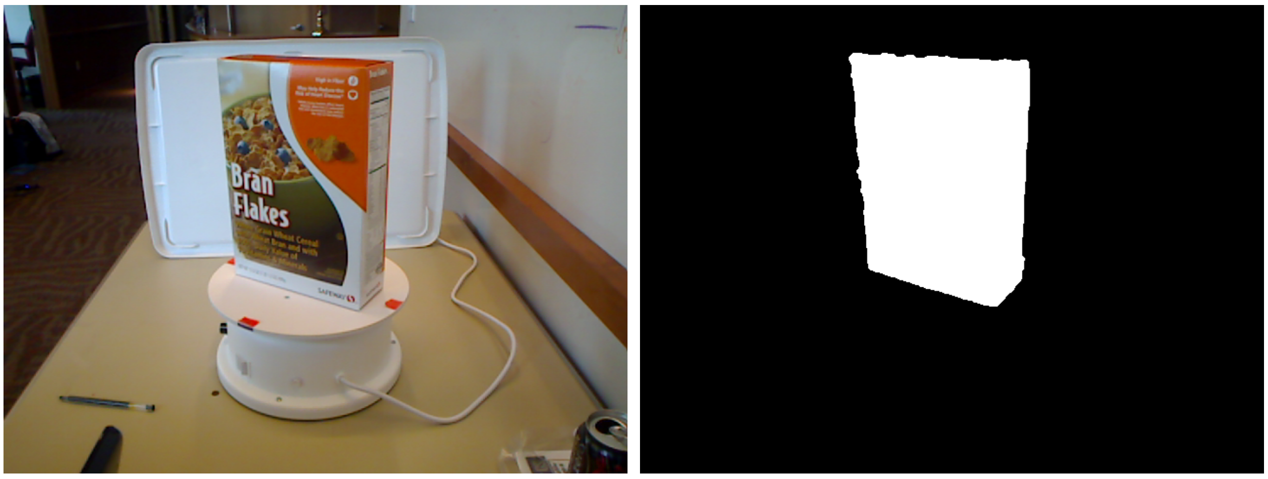
Training a random forest in a supervised fashion first requires the collection of suitable ground truth data. This chapter outlines the potential avenues available for acquiring an RGB-D dataset containing enough spatial and object segmentation data to train a Hough forest. To minimise the time required to learn a previously unseen object, and also to imitate the type of data available from a mobile robot, the decision was made to develop a tool to annotate the required ground truth data directly from the Kinect. The contribution in this chapter is an addition to the KinectFusion system [32], capable of annotating and storing Kinect data in a format compatible with the GPU Hough forest implementation.

The KinectFusion system allows a user to generate a detailed 3D reconstruction of an indoor scene, in real-time using a GPU, by freely moving a standard Kinect device within the scene. An annotation tool has been developed which uses data provided from KinectFusion to allow a user to interactively specify multiple bounding boxes around different object instances and classes to be learned. The tool then outputs, in real-time, all of the required data for training a Hough forest to learn the annotated objects.

3.1 Available Data Sources

Three potential sources of RGB-D training data were explored; publicly available datasets with both ground truth object masks and appropriate pose data, artificial training dataset generated from synthetic or scanned 3D models, or a self captured dataset along with the required software to annotate it. A discussion of the merits of each is given below.

The first option, a publicly available dataset, has the advantage of comparability with other approaches, as well as speed of procurement. There exist a number of widely used RGB-D datasets, however many of them lack a desired component. The NYU Depth Dataset [57], provides ground truth segmentation masks for different object classes, however it does not provide a consistently oriented bounding box, or object centre for each of the objects in each frame necessary for Hough voting. On the other hand, the RGB-D SLAM Dataset and Benchmark [54] and the smaller RGB-D Object Tracking Dataset [56] have accurate tracking for SLAM systems, but do not provide segmentation masks for various object classes.



(a) A colour image from the RGD-B Object Dataset (b) The ground truth object mask

Figure 3.1: An example of the data in the RGB-D Dataset. Source [34]

The RGB-D Object Dataset [34], does provide the data masks (see Figure 3.1 for an example), and enough localisation information to create a 3D Hough forest system, and this has in fact already been done for a CPU based system [3]. However, the project's broader aim would be hampered if focus is given solely to training the forest itself, ignoring the training data collection stage. If a new object is to be learned, a slow and painstaking training data collection phase would negate many of the benefits of faster training.

The second alternative, of using artificial data, has a number of additional advantages over the approach above. It allows for perfectly accurate pose tracking and object segmentation, and also allows for the generation of large quantities of training data from various poses of likely scenarios without slow manual positioning, or annotation. A dataset with these features has recently been released by Handa et al. [27], which also provides the full open source pipeline to potentially generate further test data.

Incorporating artificially generated datasets of this kind is an area of future work which could greatly enhance the accuracy and flexibility of training and testing the object detection system. However, the time and effort required to generate artificial training data of a real world object, is significant and the third, more straightforward approach has been adopted for simplicity. It also supports a key goal of the project by providing a dataset of similar quality to that available from a mobile robot.

The adopted approach was thus to develop a system to collect and provide appropriate ground truth data directly from within a SLAM (KinectFusion [32]) pipeline. This has the advantage of allowing live RGB-D video of a real life object to quickly be converted into a training dataset. It also had the advantage of allowing various 'in the wild' scenarios to be quickly put together and tested for the purposes of evaluation (See Sections 6 and 7 below). The development of the annotation tool is outlined below.

3.2 Python Prototype

To generate an initial training set from raw RGB-D images with all of the required annotations a prototype tool has been developed in Python using the basic Tkinter library [58]. This script is designed to be as simple as possible and have only the minimum functionality required. A user specifies the class type (designated by an integer) and a folder (containing the RGB-D images to be annotated) at the command line, and a basic GUI appears in which to annotate the individual training frames.

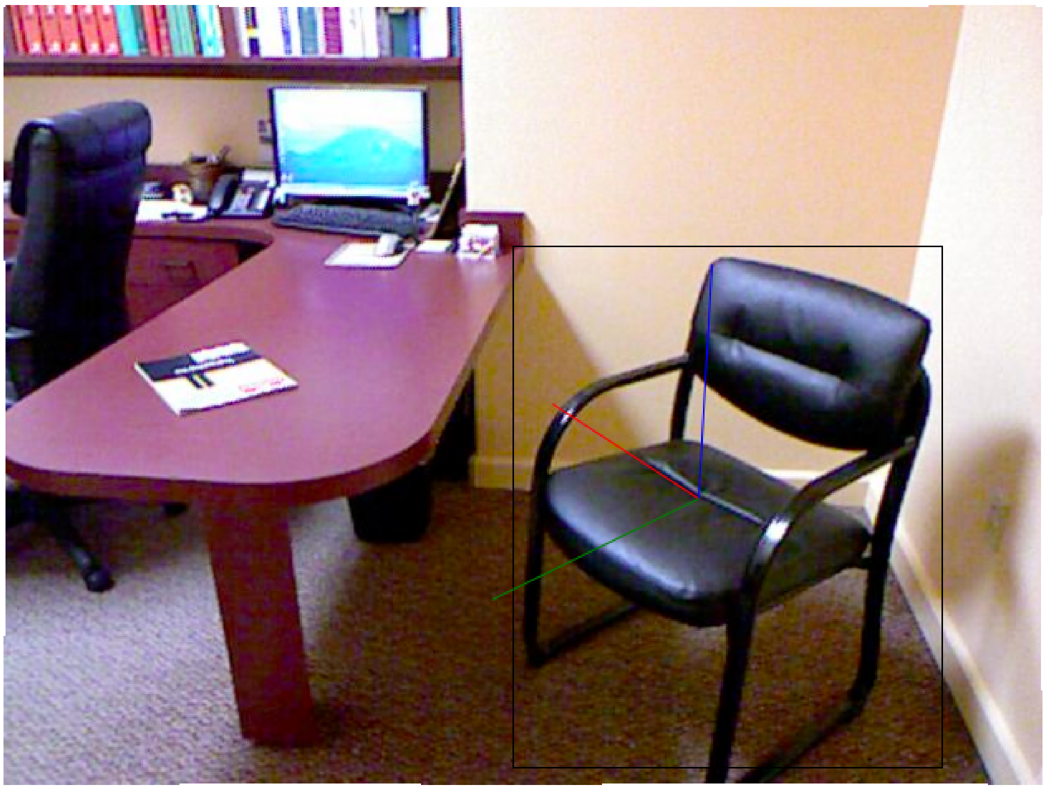


Figure 3.2: The initial Python based prototype annotation tool. Raw Image Source [57]

For each image, the user selects a rectangular bounding box around the object, denoting the pixels contained within to be of the specified class. Pixels outside the bounding box are considered to be background by default. Shift-clicking positions a set of axis at the 3D location of the click, by reading the depth value from the accompanying depth image. The user can then fine tune the position and orientation of the axis before proceeding to the next image. A screenshot of the annotation process being performed on the NYU Depth Dataset V2 [57] can be seen in Figure 3.2. The length of each axis is set to 0.5m, to give a sense of scale when positioning.

The end-result of this tagging process is written to a JSON file containing the class, object identifier, bounding box, and object pose. The object identifier is necessary because the same image could have multiple instances of the same class type tagged on successive runs. In the interest of simplicity, only a single class type can be annotated for each batch, however an image may be processed multiple times, with the output appending itself as a new entry to the array in the JSON file each time. The object pose is stored in a 4×4 transformation matrix from the camera pose to the object centre pose. An example output file can be seen in Listing 3.1.

Listing 3.1: The original ground truth JSON output from the Python annotation tool

```
[
    {
        "object_transform": [
                                [-0.57956, -0.07195, -0.81174,  0.35989],
                                [ 0.35009,  0.87750, -0.32775, -0.18820],
                                [ 0.73588, -0.47414, -0.48337,  1.88300],
                                [ 0.00000,  0.00000,   0.00000, 1.00000]
                              ],
        "bbox": {"y1": 330,"x2": 560,"x1": 315,"y2": 15},
        "id": 1,
        "class": 3
    },
]
```

This simple setup makes it possible to generate a small training set, providing all of the required information to train a Hough forest. To process this information, a JSON parser has been added to the CURFIL library [61], which converts the ground truth data into a format compatible with the existing implementation.

Although useful, the prototype annotation tool has two severe limitations. Firstly, for RGB-D video, each successive frame has to be manually annotated with both a bounding box and a pose axis; a labour intensive process, which could never be useful in a real-time system. Secondly, the simplistic rectangular bounding box failed to use any of the depth data available to provide an accurate segmentation of object from the background, and also fails to deal correctly with occlusions.

3.3 KinectFusion

The next version of the annotation tool is based on the KinectFusion system [32, 39]. KinectFusion allows a user to generate a detailed 3D reconstruction of an indoor scene, in real-time using a GPU, by freely moving a standard Kinect device within the scene. Depth data from the Kinect is used to continuously track the 6 DOF camera position using iterative closest point (ICP), and successive frames are fused together into a single volumetric surface representation [15]. The vertices are integrated into voxels using a Truncated Signed Distance Function, which specifies a relative distance to the actual surface (positive in front, negative behind, and zero at the surface itself).

Raycasting is used, not only to render the scene for users, but also to generate high quality data for the ICP camera tracking algorithm. Raycasting the signed distance function into the estimated frame provides a dense surface prediction against which the live depth map is aligned [39]. This allows issues of drift to be mitigated and reduce ICP errors. An overview of the pipeline can be seen in Figure 3.3.

The KinectFusion system provides all of the information required for a real-time, interactive, and user friendly ground truth annotation tool. Appendix A provides a detailed user guide for the tool. Here the current implementation and future directions for the tool will be discussed.

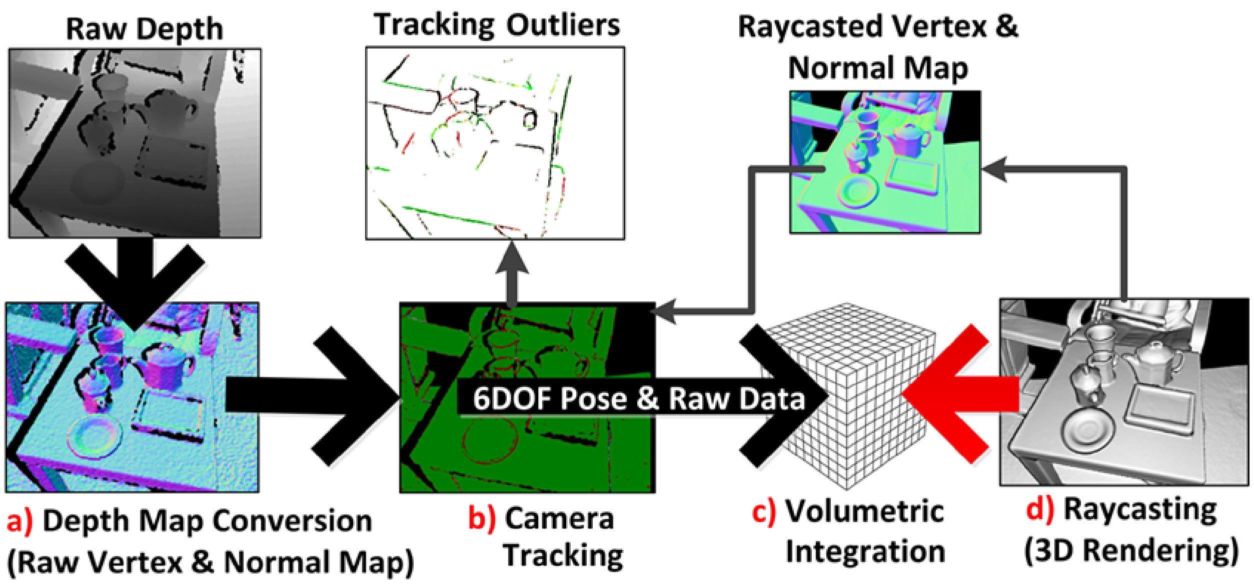

Raw Depth
Tracking Outliers
Raycasted Vertex & Normal Map
6DOF Pose & Raw Data
a) Depth Map Conversion (Raw Vertex & Normal Map)
b) Camera Tracking
c) Volumetric Integration
d) Raycasting (3D Rendering)

Figure 3.3: The pipeline of the KinectFusion system. Source [32]

3.4 Object Pose Information

The feature needed to add object annotation capability to the KinectFusion is a single user initialised 3D object bounding box. This bounding box, alongside the camera pose estimate allows for the relative transformation from the camera pose to the object pose, ${}^{O}T_C$, to be calculated in every subsequent video frame, with the simple equation,

$$ {}^{O}\mathbf{T}_C = (\mathbf{T}_C)^{-1} \times \mathbf{T}_O, $$

where $\mathbf{T}_O$ is the initial bounding box pose, and $\mathbf{T}_C$ is the camera pose transformation from its original position. The user may edit the location, orientation, and also the scale in real-time (See Figure 3.4 for an example of a positioned bounding box). At present the Hough forest does not use this scale information, it is only used for resizing the bounding box visualisation in the annotation tool. The predicted bounding boxes depicted in Section 5.3 are pre-measured and statically sized. However, scale information is stored as part of the final ground truth JSON, and could be associated to a particular class in a future version.

For memory efficiency the transformation matrix is split into three components for a total of 10 floats; a translation vector, a quaternion, and a scale vector. The translation vector ${}^{O}\mathbf{t}_C$ is easily separated from the transformation matrix ${}^{O}\mathbf{T}_C = [{}^{O}\mathbf{RS}_C, {}^{O}\mathbf{t}_C]$. Both the rotational and scalar components are contained within the ${}^{O}\mathbf{RS}_C$ matrix. As pure rotations are always orthogonal, the scale component, ${}^{O}\mathbf{S}_C$, can be extracted by transposing the original, ${}^{O}\mathbf{S}_C = {}^{O}\mathbf{RS}_C^{\mathbf{T}} \times {}^{O}\mathbf{RS}_C$, and the pure rotation can then be extracted using the inverse of the scale component,

$$ {}^{O}\mathbf{R}_C = {}^{O}\mathbf{RS}_C \times ({}^{O}\mathbf{S}_C)^{-1}. $$

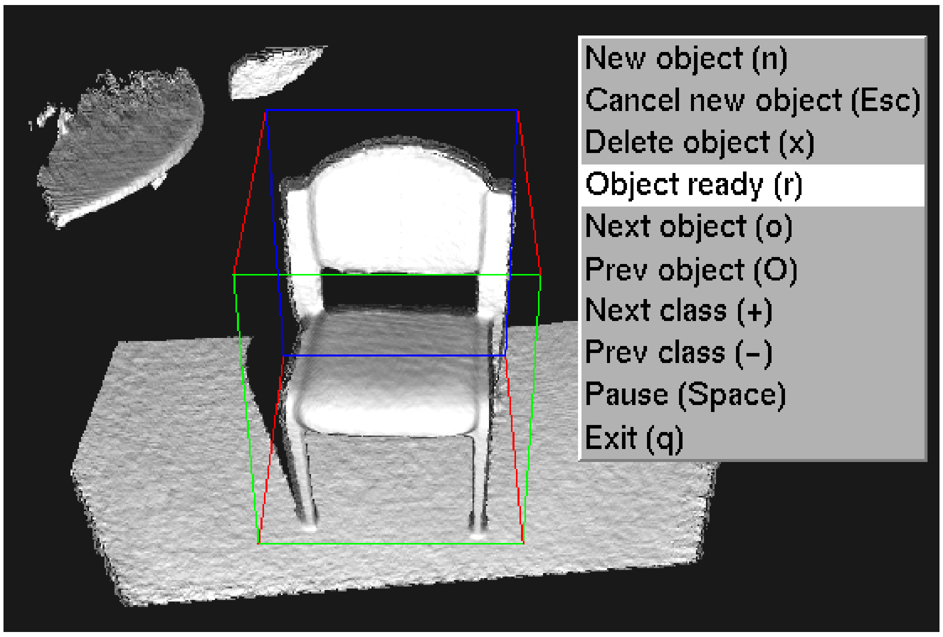


Figure 3.4: KinectFusion with the interactive menu and user positioned 3D bounding box

3.5 Ground Truth Segmentation

The KinectFusion system stores a 3D volumetric representation of the scene. This representation provides the capacity to render a pixel-wise ground truth segmentation mask, using the defined bounding box. An additional raycasting algorithm is used to render this mask. Every face of the bounding box forms part of a planar half space test, if the dot product of the plane and the hit position is greater than or equal to zero then that point is rendered with a colour associated to the object type (See Listing 3.2). Figure 3.5 shows an example of the rendered ground truth image. Any number of objects and classes can be selected for a given video (See Appendix A for a full guide).

Listing 3.2: The core ground truth segmentation kernel

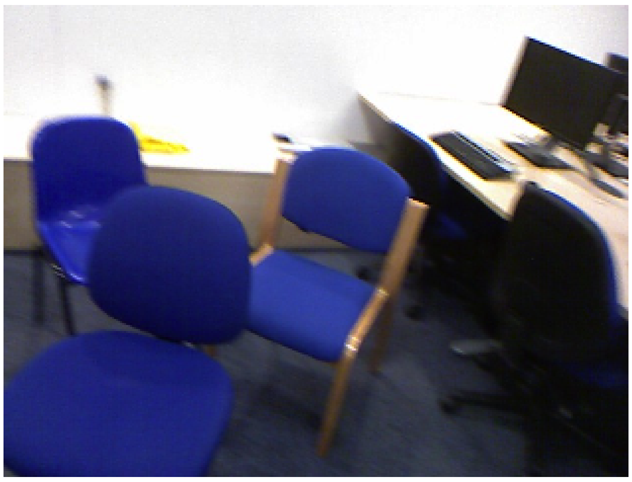
```
__device__ bool inBoundingBox(const float4& hit, ClassType& type, ObjectType& id) {
    for(size_t objectId = 0; objectId < numObjects; ++objectId) {
        // First transform the point into the Bounding Box coordinate system.
        float4 localHit = matrixData()[objectId * sizeof(Matrix4)] * hit;
        // Then perform half space checks on the 6 planes of a unit cube.
        if(dot(topPlane,    localHit) < 0) continue;
        if(dot(bottomPlane, localHit) < 0) continue;
        if(dot(nearPlane,   localHit) < 0) continue;
        if(dot(farPlane,    localHit) < 0) continue;
        if(dot(leftPlane,   localHit) < 0) continue;
        if(dot(rightPlane,  localHit) < 0) continue;
        // If this line is reached, the hit is within the bounding cuboid.
        type = classData()[objectId * sizeof(ClassType)];
        id = objectId;
        return true;
    }
    return false;
}
```

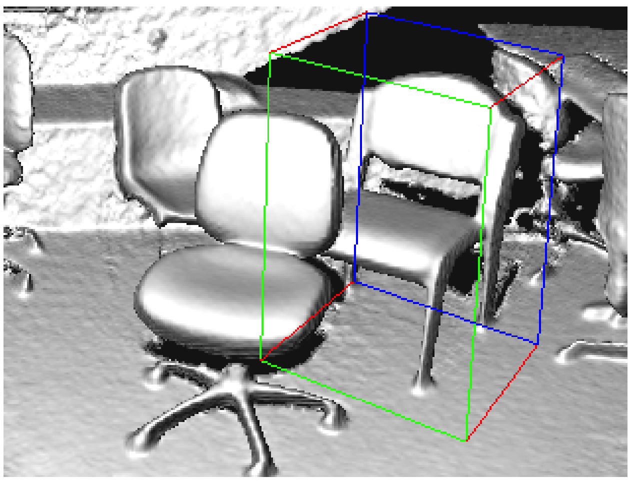
Multiple instances of a single class may be in any given frame, therefore the segmentation mask is output as the *object identifier*, in a 16-bit PNG image file. This object identifier is linked to a class type and object pose in the accompanying JSON file. An example JSON is given in Listing 3.3.

Listing 3.3: The final ground truth JSON output from the KinectFusion annotation tool

```
[{
        "object_transform": [-0.0185307,0.071515,1.27029,
                             0.0470628,-0.679116,-0.163071,0.714139,
                             0.654244,0.895703,0.620389],
        "id": 1,
        "class": 2
}]
```

For speed and simplicity the open source single-file *lodepng* library [37] is used to write all of the PNG files (both RGB and Depth, and the ground truth segmentation mask). The default PNG compression used by the library proved too slow for real-time interactive annotation. With speed rather than storage efficiency being the primary aim for the dataset, this compression has been turned off. For a test set of 100 random 640 × 480 PNG images, the average size per image increased from 387Kb compressed, to 818Kb uncompressed. On a 2.13 GHz Intel Xeon E5606, the average time for processing and writing an image decreased from 127.0ms to 47.2ms, allowing real-time interactivity at approximately 20 frames per second. After the training data is captured and written, the Hough forest training procedure may begin.


(a) A colour image from Kinect Device.


(b) The KinectFusion model, with a bounding box.

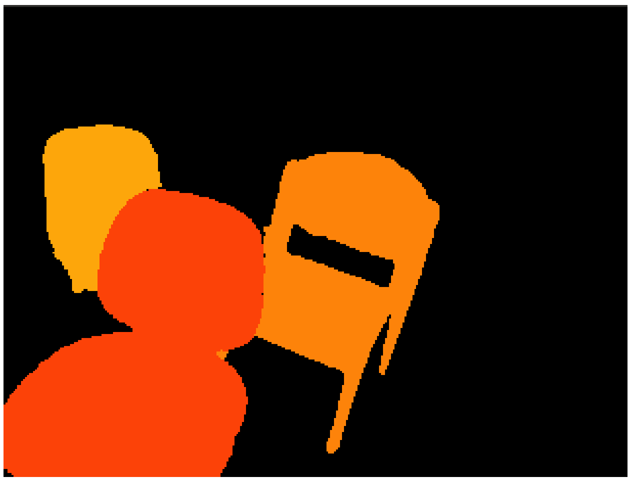

(c) The rendered ground truth object mask.

Figure 3.5: The stages leading to a pixel level ground truth segmentation mask.

4 Hough Forest Training on the GPU

The following chapter describes in detail the development of a GPU accelerated RGB-D Hough forest. The basis for this implementation is the CURFIL library [62], which provides the core components required for rapid training of an image classification forest on a GPU. To build a Hough forest from this library required significant modifications:

1. The core modification is the addition of the second objective for split nodes, to also minimise the offset uncertainty of Hough votes. Efforts were made to ensure this computationally expensive procedure would not cause a significant slow down during training.
2. To improve accuracy of the multi-class forest, the modified information gain calculation described by Razavi et al. [45] has been added, to better separate the background class from object classes.
3. The existing JSON implementation has been replaced with the *rapidjson* library. This was necessary because of the much larger leaf nodes required by a Hough forest, which resulted in extremely slow forest reading/writing.

Two additional contributions have also been made, with the aim of improving accuracy of the forest without increasing training time.

1. A post-training process has been implemented, which augments the data stored at leaf nodes by quickly passes additional training data down the tree after split tests have already been decided. This is both fast, and could help to alleviate overfitting.
2. To avoid re-training a tree with multiple maximum depths, a modification has been made which allows the maximum depth to be specified at *test time*, to retrospectively tune the optimum depth [11].

Finally, the combined results of optimisation efforts for the offset uncertainty calculation were compared against a naïve CPU implementation, and showed a 133× improvement.

4.1 The CURFIL Library

The CUDA Random Forests for Image Labeling (CURFIL) Library is an open source implementation that accelerates random forest training and prediction for labelling RGB-D images through use of a GPU [62]. CURFIL is the result of Waldvogel's [61] detailed analysis on GPU optimisations for training a random forest to perform pixel-wise classification, and it forms the core implementation for the Hough forest object detector. Presented here is a brief description of the important features implemented within the library. For a more complete description of the library and the extensive optimisation efforts made, please refer to the original thesis [61].

Training of trees proceeds in a breadth-first manner, according to Algorithm 1. Breadth-first is both conducive for parallelism, and provides the freedom to generate a single set of random features once per tree level, thereby improving efficiency.

As proposed by Stückler et al. [55], two types of RGB-D image feature responses are used, depth and colour. The feature response is calculated as the difference of two region averages that are offset around the query pixel. Images are integrated to allow for this averaging to be calculated efficiently with four memory accesses and three arithmetic operations (See Figure 4.1). Depth is always compared against depth, but any of the colour channels may be compared against any other. The colour channels can be either in raw RGB, or converted to CIELab space.

As discussed in Section 2.3, the offsets and region size use depth normalisation. As RGB-D cameras often do not guarantee depth data for every pixel, a second depth image channel is used to store valid depth values for a given region. A depth filling scheme proposed by Stückler is also available, which uses neighbouring pixels to efficiently reconstruct missing depth values.

Algorithm 1 Breadth-first training of a random decision tree. Source [61]

```
Require: $\mathcal{D}$ training instances
Require: $F$ number of feature candidates to generate
Require: $P$ number of feature parameters
Require: $T$ number of candidate thresholds to generate
Require: stopping criterion (eg. maximal depth)
 1: $D \leftarrow$ randomly sampled subset of $\mathcal{D}$ ($D \subset \mathcal{D}$)
 2: $N_{root} \leftarrow$ create root node
 3: $C \leftarrow \{(N_{root}, D)\}$                         ▷ initialize candidate nodes
 4: while $C \neq \varnothing$ do
 5:     $C' \leftarrow \varnothing$                          ▷ initialize new set of candidate nodes
 6:     for all $(N, D) \in C$ do
 7:         $(D_{left}, D_{right}) \leftarrow$ EvaluateBestSplit($D$)
 8:         if stopping criterion does not hold for $(N, D_{left})$ then
 9:             $N_{left} \leftarrow$ create left child for node N
10:             $C' \leftarrow C' \cup \{(N_{left}, D_{left})\}$          ▷ add left child to candidates
11:         if stopping criterion does not hold for $(N, D_{right})$ then
12:             $N_{right} \leftarrow$ create right child for node N
13:             $C' \leftarrow C' \cup \{(N_{right}, D_{right})\}$       ▷ add right child to candidates
14:     $C \leftarrow C'$                                     ▷ continue with new set of nodes
```

The feature parameters are randomly generated using the CUDA library `CURAND`. There are 11 feature parameters in total (such as region offsets, depth or colour, channel choice), and these are stored in an $F \times 11$ matrix, where F is the number of candidate features. Threshold candidates are then obtained by sampling feature responses of training instances, as done in the *Tuwo* implementation [40]. For every feature the threshold candidates, T, are stored in an $F \times T$ matrix.

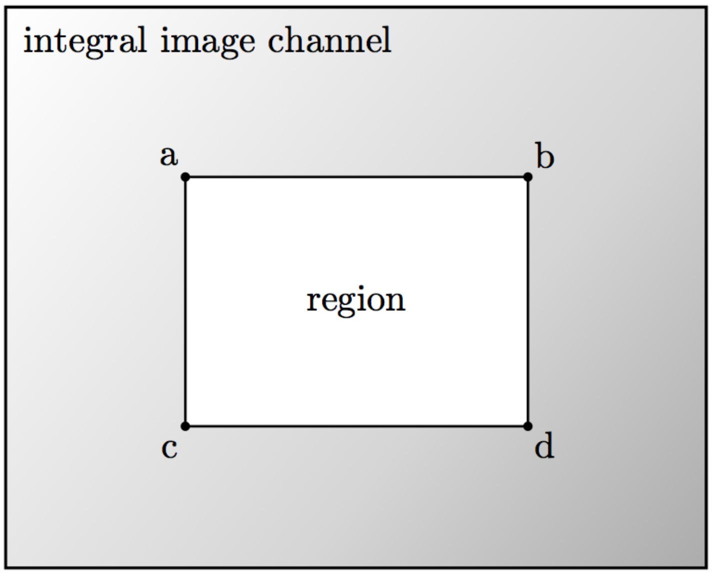


Figure 4.1: An illustration of the integration method used by the CURFIL library. Each pixel is the sum of all the pixels to its above left, therefore the region sum $= d - c - b + a$. Source [61]

For the feature response calculation, training images are stored in texture memory. Samples from training images D are stored with their corresponding image identifier, depth, and x and y offsets in a $D \times 4$ matrix. Features are sorted by type (colour or depth), channels, and offsets to increase the hit rate of the texture cache and the L2 cache by assisting sequential accesses in spatially nearby regions, and speed up the response calculation, which is stored in a $D \times F$ matrix.

To calculate the best score, counters for each class, feature, and threshold are stored in a 4D matrix of size $C \times T \times F \times 2$, with C being the number of classes, and the fourth dimension required for storing the right-left split. The algorithm keeps the samples in the innermost loop to maximise the use of shared memory (See Algorithm 2) and then subsequently performs a sum reduction step to avoid atomic operations. This is an approach followed in the offset uncertainty algorithm and discussed in more detail in Section 4.2.

Algorithm 2 Histogram aggregation on GPU. Source [61]

```
Require: $\mathbf{F} \in \mathbf{R}^{D \times F}$ feature responses for D samples and F features
Require: $\mathbf{T} \in \mathbf{R}^{F \times T}$ random threshold candidates for each feature
Require: $\mathbf{C} \in \{1..C\}^D$ class for each sample
1: $\mathbf{H} \leftarrow 0_{F,T,C,2}$                    ▷ initialise histogram counters
2: for all $f \in 1..F$ do
3:     for all $\tau \in \mathbf{T}_f$ do
4:         for all $d \in 1..D$ do
5:             $c \leftarrow \mathbf{C}_d$                 ▷ class c of sample d
6:             if $\mathbf{F}_{d,f} \leq \tau$ then
7:                 $\mathbf{H}_{f,\tau,c,0} \leftarrow \mathbf{H}_{f,\tau,c,0} + 1$   ▷ increment left histogram counter
8:             else
9:                 $\mathbf{H}_{f,\tau,c,1} \leftarrow \mathbf{H}_{f,\tau,c,1} + 1$   ▷ increment right histogram counter
```

The score kernel is then implemented in a straightforward method. Each thread calculates the information gain for a given threshold and feature, by reading the counter matrix from global memory, calculating the impurity score according to Equation 1 & 2, and the writing the result back to global memory.

4.2 Implementing the Offset Uncertainty

The additional best split objective of offset uncertainty, is the main modification required to the GPU training implementation. This calculation is the most computationally expensive training step, and so care is needed to ensure the GPU is used efficiently. Algorithm 3 describes the basic two stage offset uncertainty calculation, as defined in Equation 3. First the mean offsets for right and left split samples are calculated for a given feature and threshold. This is then used to calculate the total squared Euclidean distance from the mean for both right and left subsets. After Algorithm 3 the best split can be trivially computed by selecting the feature f and threshold τ with the smallest offset uncertainty in the 2D uncertainty matrix U.

Unfortunately an additional layer of complexity is required on top of this algorithm, resulting in the final lengthier Algorithm 4. The large number of samples $|D|$ used when training a forest of any considerable size can far exceed the available memory in the GPU. This requires samples to be processed in batches of a more manageable size. For the histogram aggregation seen in Algorithm 2 this does not present any difficulty, as the batches can simply be allowed to accumulate in $\mathbf{H}$. For Algorithm 3 however, the global mean of all batches must first be calculated, and then for every batch the squared difference must be taken for each sample. This two pass system would be computationally expensive and require each sample to be transferred to the GPU twice, slowing the training process down dramatically.

A modification exists, that allows the same offset uncertainty result to be calculated, while processing each batch only once, and then combining it into a running total. The basis for the improvement is an equation derived by Chan et al. [8] to combine the variance of two samples of arbitrary size in a single pass. Given two samples, $\{x_i\}_{i=1}^{j}$ and $\{x_i\}_{i=j+1}^{j+k}$, the totals, T and square

Algorithm 3 Offset uncertainty calculation on the GPU

```
Require: $\mathbf{F} \in \mathbf{R}^{D \times F}$ feature responses for D samples and F features
Require: $\mathbf{T} \in \mathbf{R}^{F \times T}$ random threshold candidates for each feature
Require: $\mathbf{O} \in \mathbf{R}^{D}_{c,c \neq bg}$ contains offsets, $\mathbf{o}_n$, from each positive sample
 1: $\mathbf{U} \leftarrow 0_{F,T}$
 2: for all $f \in 1..F$ do
 3:     for all $\tau \in \mathbf{T}_f$ do
 4:         $\mathbf{s}^{left} \leftarrow 0_{n+1}$          ▷ initialize the left and right totals to a zero vector
 5:         $\mathbf{s}^{right} \leftarrow 0_{n+1}$
 6:         for all $d \in 1..D$ do
 7:             if exists $\mathbf{O}_d$ then          ▷ ensure this is a positive sample with an offset
 8:                 $\mathbf{o} \leftarrow \mathbf{O}_d$
 9:                 if $\mathbf{F}_{d,f} \leq \tau$ then
10:                     $\mathbf{s}^{left} \leftarrow \mathbf{s}^{left} + [\mathbf{o}, 1]$
11:                 else
12:                     $\mathbf{s}^{right} \leftarrow \mathbf{s}^{right} + [\mathbf{o}, 1]$
13:         $\mathbf{s}^{left} \leftarrow \mathbf{s}^{left}_{1..n} / \, \mathbf{s}^{left}_{n+1}$          ▷ normalise with respect to the final counter value ...
14:         $\mathbf{s}^{right} \leftarrow \mathbf{s}^{right}_{1..n} / \, \mathbf{s}^{right}_{n+1}$          ▷ ... to arrive at the mean offset vector for this split
15:         for all $d \in 1..D$ do
16:             if exists $\mathbf{O}_d$ then
17:                 $\mathbf{o} \leftarrow \mathbf{O}_d$
18:                 if $\mathbf{F}_{d,f} \leq \tau$ then
19:                     $\mathbf{U}_{f,\tau} \leftarrow U_{f,\tau} + \left\| \mathbf{o} - \mathbf{s}^{left} \right\|_2$
20:                 else
21:                     $\mathbf{U}_{f,\tau} \leftarrow U_{f,\tau} + \left\| \mathbf{o} - \mathbf{s}^{right} \right\|_2$
```

differences, S, of the samples set, can readily be calculated;

$$T_{1,j} = \sum_{i=1}^{j} \qquad T_{j+1,j+k} = \sum_{i=j+1}^{j+k} \tag{8}$$

$$S_{1,j} = \sum_{i=1}^{j} (x_i - \frac{1}{j} T_{1,j})^2 \qquad S_{j+1,j+k} = \sum_{i=j+1}^{j+k} (x_i - \frac{1}{k} T_{j+1,j+k})^2. \tag{9}$$

It is then possible to combine these processed statistics into a single summed square difference of the entire population with the following equations,

$$T_{1,j+k} = T_{1,j} + T_{j+1,j+k} \tag{10}$$

$$S_{1,j+k} = S_{1,j} + S_{j+1,j+k} + \frac{j}{k(j+k)} \left(\frac{k}{j} T_{1,j} - T_{j+1,j+k} \right)^2 \tag{11}$$

These equations are suitable for single dimensional data, and require the storage of counters (m, n), totals ($T_{1,m+n}$, $T_{m+1,m+n}$), and the summed square differences ($S_{1,m+n}$, $S_{m+1,m+n}$). When aggregating n-dimensional vectors, the counters can be shared, and the squared Euclidean distance from the mean can be combined for all dimensions, leaving only the totals which must be maintained separately. For b batches, N values must be stored, where $N = (n+2) \times b$. As aggregation is required for both the right and left split nodes, the original matrix $\mathbf{U}_{F,T}$ must be enlarged to $\mathbf{U}_{F,T,2,N}$.

Initially, it was planned for all of the sample batch results from Equation 8 to be stored in a single matrix, and aggregated in a single parallel reduction step. However, the large number of batches made storage of even the reduced statistics unfeasible, and also profiling suggested the

Algorithm 4 Modified offset uncertainty calculation for the GPU

```
Require: $\mathbf{F} \in \mathbf{R}^{D \times F}$ feature responses for D samples and F features
Require: $\mathbf{T} \in \mathbf{R}^{F \times T}$ random threshold candidates for each feature
Require: $\mathbf{O} \in \mathbf{R}^{D}_{c,c \neq bg}$ contains offsets, $\mathbf{o}_n$, from each positive sample
Require: $n$ the number of dimensions contained in the offsets
 1: $\mathbf{U}^{total} \leftarrow 0_{F,T,2,N}$
 2: for all $B \in D$ do        ▷ take the first range B, from the complete set of samples D
 3:     $\mathbf{U}^{tmp} \leftarrow 0_{F,T,2,N}$
 4:     for all $f \in 1..F$ do
 5:         for all $\tau \in \mathbf{T}_f$ do
 6:             $\mathbf{s}^{left} \leftarrow 0_{n+1}$        ▷ initialize the left and right totals to a zero vector
 7:             $\mathbf{s}^{right} \leftarrow 0_{n+1}$
 8:             for all $b \in B_{start}..B_{end}$ do
 9:                 if exists $\mathbf{O}_b$ then        ▷ ensure this is a positive sample with an offset
10:                     $\mathbf{o} \leftarrow \mathbf{O}_b$
11:                     if $\mathbf{F}_{b,f} \leq \tau$ then
12:                         $\mathbf{s}^{left} \leftarrow \mathbf{s}^{left} + [\mathbf{o}, 1]$
13:                     else
14:                         $\mathbf{s}^{right} \leftarrow \mathbf{s}^{right} + [\mathbf{o}, 1]$
15:             $\mathbf{s}^{left} \leftarrow \mathbf{s}^{left}_{1..n} / \mathbf{s}^{left}_{n+1}$        ▷ normalise with respect to the final counter value
16:             $\mathbf{s}^{right} \leftarrow \mathbf{s}^{right}_{1..n} / \mathbf{s}^{right}_{n+1}$
17:             for all $b \in B_{start}..B_{end}$ do        ▷ store required information in the temporary matrix
18:                 if exists $\mathbf{O}_b$ then
19:                     $\mathbf{o} \leftarrow \mathbf{O}_b$
20:                     if $\mathbf{F}_{b,f} \leq \tau$ then
21:                         $\mathbf{U}^{tmp}_{f,\tau,0,0} \leftarrow U_{f,\tau,0,0} + 1$
22:                         $\mathbf{U}^{tmp}_{f,\tau,0,1} \leftarrow U_{f,\tau,0,1} + \left\|\mathbf{o} - \mathbf{s}^{left}\right\|_2$
23:                         $\mathbf{U}^{tmp}_{f,\tau,0,2..n} \leftarrow U_{f,\tau,0,2..n} + \mathbf{o}$
24:                     else
25:                         $\mathbf{U}^{tmp}_{f,\tau,1,0} \leftarrow U_{f,\tau,1,0} + 1$
26:                         $\mathbf{U}^{tmp}_{f,\tau,1,1} \leftarrow U_{f,\tau,1,1} + \left\|\mathbf{o} - \mathbf{s}^{right}\right\|_2$
27:                         $\mathbf{U}^{tmp}_{f,\tau,1,2..n} \leftarrow U_{f,\tau,1,2..n} + \mathbf{o}$
28:             for all $split \in \{0, 1\}$ do        ▷ combine matrix statistics into the total matrix
29:                 $j \leftarrow \mathbf{U}^{total}_{f,\tau,split,0}$
30:                 $k \leftarrow \mathbf{U}^{tmp}_{f,\tau,split,0}$
31:                 $\mathbf{U}^{total}_{f,\tau,split,0} \leftarrow \mathbf{U}^{total}_{f,\tau,split,0} + \mathbf{U}^{tmp}_{f,\tau,split,0}$
32:                 $\mathbf{U}^{total}_{f,\tau,split,1} \leftarrow \mathbf{U}^{total}_{f,\tau,split,1} + \mathbf{U}^{tmp}_{f,\tau,split,1} + \frac{j}{k(j+k)} \left\| \frac{k}{j} \mathbf{U}^{total}_{f,\tau,split,2..n} - \mathbf{U}^{tmp}_{f,\tau,split,2..n} \right\|_2$
33:                 $\mathbf{U}^{total}_{f,\tau,split,2..n} \leftarrow \mathbf{U}^{total}_{f,\tau,split,2..n} + \mathbf{U}^{tmp}_{f,\tau,split,2..n}$
```

combine step takes much less time than the square difference calculation or the feature response calculation (See Figure 4.2). Therefore a batch size of 2 is used, with each batch collecting statistics into one half of the U matrix and then having it immediately merged into the running total statistics in the other half.

4.3 Multi-Class Modifications

Information Gain Score

For using the forest to detect multiple classes, it is also to ensure a strong separation of the background from positive samples, before separating the classes themselves. A optional weighted

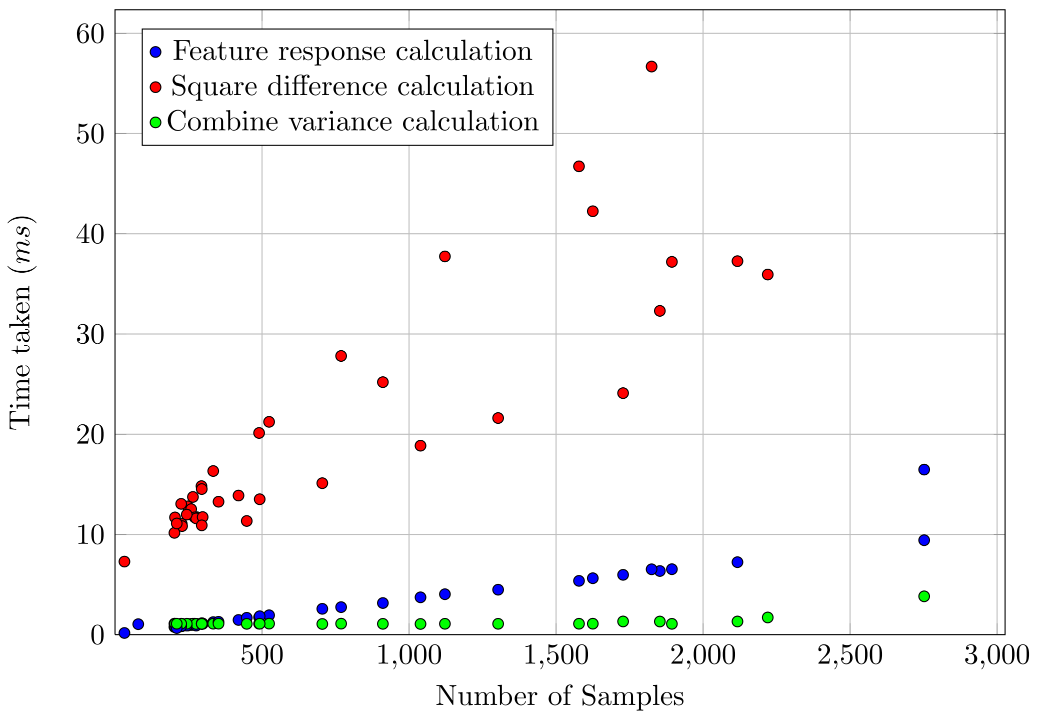


Figure 4.2: The number of samples vs the time taken to complete each stage, for 5,000 features, 20 thresholds, and a batch size of 2.

information gain score as described in Equation 7 has been implemented within CURFIL's original score kernel. The modification required only two small changes to the existing kernel, firstly to keep count of the total number of background instances and positive samples, and secondly to calculate the background separation score and combine it with the λ coefficient. The annotation tool reserves class 0 and object identifier 0 for the background, rendering this change straightforward.

Relaxed Offset Uncertainty

The offset uncertainty Algorithms 3 and 4 focus on the original single-class optimisation of Equation 3. The multi-class 'relaxed' offset uncertainty, given in Equation 4, would require a fifth dimension to also accommodate the offset uncertainty for each class C. This would be both more computationally expensive and result in significant additional memory requirements. With a focus on rapid training, and for the evaluation dataset presented here, with similar sized classes, this feature has not been implemented. However, for future versions and more varied datasets, it may result in improved accuracy worth the additional training time.

4.4 Post-Training Data

The performance of Totally Randomised Trees [24], where a single feature and threshold are chosen at random, suggests that the information stored within the leaf nodes may have a more significant impact on the accuracy of the tree, than the choice of the split node function. Storing large quantities of data at the leaf nodes comes with a corresponding increase in training time, an effect most apparent when the number of sampled training images increases past the GPU cache size. This has led to the development of a post-training step, in which new samples are passed down an already trained tree to gain an improved distribution at the leaf nodes. This is not only fast, but has the benefit of alleviating overfitting, as the trees structure is no longer being tailored to the input data. This "post-training" step has been performed entirely on the CPU, as it is bound by reading and processing training images rather than the tree traversal algorithm.

To avoid re-training a tree for exploring multiple values of maximum depth, as in Section 7.2, the suggestion by Shotton et al. in [11] has been followed on "Retrospective Tuning of Maximum

Tree Depth". This modification to CURFIL allows the tree depth to be specified at *test* time, but requires that every node, not just leaf nodes, store the aggregated information of the samples that arrive to them. The post-training tree traversal procedure can also easily accommodate training at every depth with a simple recursive functions, shown in Listing 4.1.

Listing 4.1: Post-training functions to increase sample size

```
/* Traverse a tree while adding the sample to the node statistics */
void postTrainTraverse(const Sample& sample, float4& offset, float4& quat) {
    if (isLeaf() || trainAllDepths) {
        this->addOffset(offset);
        this->addQuaternion(quat);
        this->histogram[sample.getLabel()]++;
        if(isLeaf())
            return;
    }
    if (split(instance) == LEFT)
        left->postTrainTraverse(sample, offset, quat);
    else
        right->postTrainTraverse(sample, offset, quat);
}
```

4.5 Optimisation Efforts and Benchmarking

Efficient Squared Difference Sum Reduction

To calculate the offset uncertainty as quickly as possible, the use of low latency high bandwidth shared memory is essential. Individual threads are used to accumulate three-dimensional offset vectors from training samples in global memory, using a sequential coalesced memory pattern. Each thread is given access to their own designated shared memory block with room for 8 floating point numbers. This block stored the x, y, and z offset component totals, as well as a count of vectors, for both the right and left splits, as shown in Listing 4.2. By ensuring that each thread has access to a unique block in memory, the use of atomics is not required, thus avoiding serialisation.

Listing 4.2: Each thread stores the accumulated result of its samples in the shared memory block

```
// Each thread accesses a sample in sequential order to ensure coalesced loading
for (int sample = threadIdx.x; sample < numSamples; sample += blockDim.x) {
    float xOffset = xSample[sample];
    float yOffset = ySample[sample];
    float zOffset = zSample[sample];
    // Branching statements are avoided by using the threshold function as an offset
    int right = static_cast<int>(!(featureResponse <= threshold));
    sharedMem[threadIdx.x * 8 + right * 4 + 0] += xOffset;
    sharedMem[threadIdx.x * 8 + right * 4 + 1] += yOffset;
    sharedMem[threadIdx.x * 8 + right * 4 + 2] += zOffset;
    sharedMem[threadIdx.x * 8 + right * 4 + 3] += 1.0;
    featureResponse += blockDim.x;
}
```

This shared memory then requires a reduction step to calculate the total mean of the batch for both the right and left node. The initial naïve reduction step shown in Listing 4.3 works up the thread block in powers of two, taking even threads and adding to their memory block the thread above it. This process is repeated until thread 0 stores the accumulated sum reduction of all the counter and total offsets within the left and right nodes. After each reduction the `__syncthreads()`

command is used to make sure all threads have completed the reduction before moving on.

Listing 4.3: The original naïve sum reduction kernel

```
// For every thread within repeatedly lower halfs of shared memory
for (unsigned int sumIdx=1; sumIdx < blockDim.x; sumIdx *= 2) {
    if (threadIdx.x % (2 * sumIdx) == 0) {
        for(unsigned int storageIdx = 0; storageIdx < 8; storageIdx++) {
            sharedMem[threadIdx.x * 8 + storageIdx] +=
                      sharedMem[(threadIdx.x + sumIdx) * 8 + storageIdx];
        }
    }
    __syncthreads();
}
```

An improved reduction algorithm has been implemented [28]. First the shared memory, which is eight times larger than the number of threads is compressed into two memory blocks equal in size to a single thread block, using all available threads. It then avoids divergent branching and use of the modulo operator as shown above, instead using strided sequential access in a decreasing power of two pattern, to speed up the remainder of the reduction process. This reduction is shown in the kernel in Listing 4.4. This change reduces the time taken to process 1,000 samples from 22.7ms to 20.0ms, a 11.1% reduction in the square difference kernel calculation time.

Listing 4.4: The optimised sum reduction kernel

```
// Compress sequential shared blocks into the lowest of the 8 thread blocks
// using all of the available threads simultaneously
for (unsigned int sumIdx = 7; sumIdx > 1; sumIdx--) {
    sharedMem[threadIdx.x] +=
              sharedMem[threadIdx.x + blockDim.x * sumIdx];
}
// Reduce the shared memory into the first thread only, with strided access
for (unsigned int sumIdx = blockDim.x; sumIdx >= 8; sumIdx >>= 1) {
    if (threadIdx.x < sumIdx) {
        sharedMem[threadIdx.x] += sharedMem[(threadIdx.x + sumIdx)];
    }
    __syncthreads();
}
```

Tuning the Block Size

The number of threads per block as part of the square difference calculation can be tuned, and corresponds to the number of samples in a batch which are processed simultaneously. For the reduction step, it is required to be a power of two. The number of threads was experimented on up to the maximum of 1,024, and it was found through benchmarking that 128 threads resulted in the fastest calculation (See Figure 4.3).

Benchmarking against a CPU Implementation

A single threaded CPU version of Algorithm 3 has been implemented for two reasons. First, it provides a method of ensuring the result of the more complex GPU based Algorithm 4 is identical to the basic algorithm. Second it also provides a benchmark to measure the speed-up attained for moving to a “one-pass and combine” approach on a GPU. This comparison does not strictly assess the advantages of the GPU over a CPU, as the CPU implementation has neither been multi-

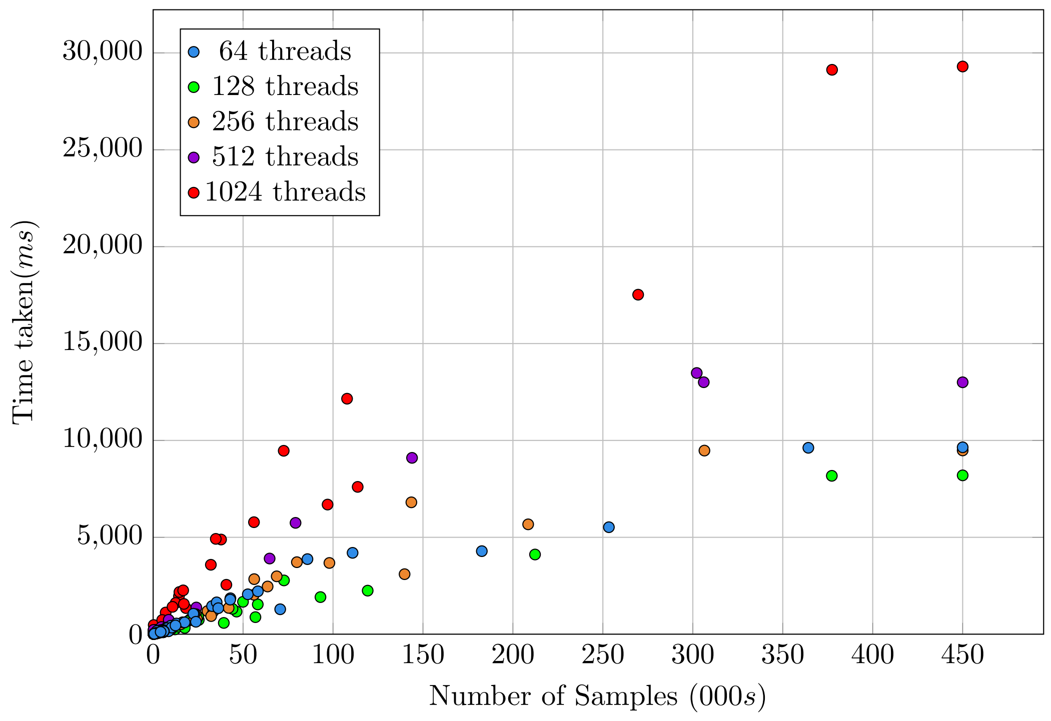


Figure 4.3: Thread block size comparison of total samples processed vs time taken to calculate the offset uncertainty for 5,000 features, 20 thresholds, and a batch size of 2.

threaded nor optimised in any significant way. Instead it aims to give an approximate measure of the total speed-up attained as a result of a combination of all of the above work on the offset uncertainty calculation, from an initial naïve C++ implementation. The result (Figure 4.4) showed that the GPU implementation witnessed a performance increase of on average 133×, when training a 20-level tree, with 500 features.

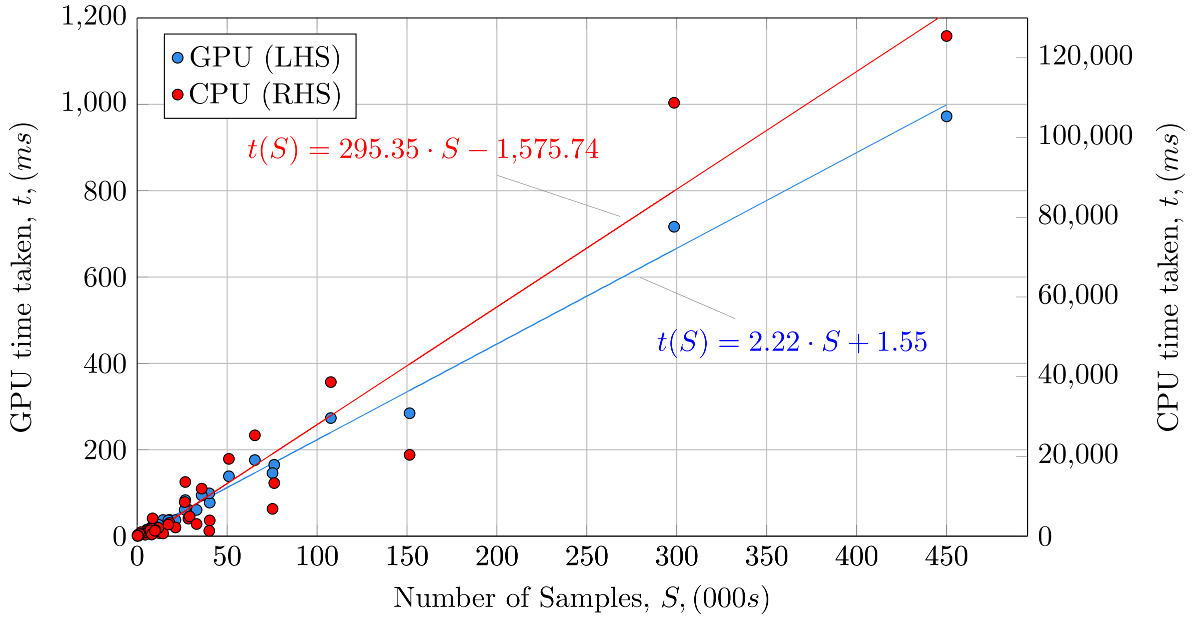


Figure 4.4: GPU comparison against naïve CPU implementation of the offset uncertainty calculation. Samples vs square difference calculation time, for 500 features and 20 thresholds.

Rapid exporting

In the original CURFIL library, the trained trees are exported in a JSON format using boost's *Property Tree* [7] library and compressed using gzip. The Property Tree library provides a generic data structure to store nested trees of values, however it is not optimised for use with the JSON format, instead aiming for generality. For storage of random trees with only histogram data at the leaf nodes, this lack of optimisation presents no difficulty, as thousands of samples boiled down into a single statistic for each class.

A Hough forest however, often requires leaf node storage which is orders of magnitude larger than a class histogram (See Section 5.1 for a detailed discussed of the leaf representation). The pressure on Boost's unoptimised Property Tree library resulted in an excessively long export time. A number of open source JSON C++ libraries exist; rapidjson, jsoncpp, YAJL to name a few. Rapidjson [43] was eventually selected, because of its simplicity and focus on high performance, which is apparent in a number of benchmarking experiments [2, 44] (See Table 4.1). The change to rapidjson resulted in a dramatic decrease in export time. For a tree of 20 levels and 135,000 samples, export time decreased from 18 *minutes* to 232 *ms*.

Table 4.1: Time in ms to parse a 672Kb test document, using an Intel Core i7 CPU 920 2.67Ghz, GCC 4.5.3 in Cygwin. Source [44]

Time Taken for Operation (ms)	rapidjson	YAJL	JsonCPP
Parsing Document Object Model	796	6,316	6,705
Stringify Document Object Model	1,289	14,040	24,757

5 Object Detection on the GPU

To ensure that the trained random forest is capable operating in a real-time system, and also to allow for rapid evaluation on a test set, object detection has also been accelerated with a GPU. To accomplish this, the following contributions have been made:

1. An efficient discretised cubic representation of offset votes has been developed and stored in GPU memory as part of a two-dimensional layered texture. Alongside this, orientation votes in the form of quaternions have been compressed and also stored for retrieving the object rotation.

2. A Hough voting procedure has been developed, building on the tree traversal algorithm already available in CURFIL. This allows each tree to vote into a unified three-dimensional Hough space, which is then normalised.

3. A rapid mean shift clustering algorithm has been designed for a GPU, which uses efficient parallelised flat cubic kernels. This processes the raw Hough space information, and retrieves the most likely object centres, within a given threshold criteria.

4. Finally a second pass is performed on the most likely object centres, and information from the nodes which voted towards them is used to estimate the class and orientation of the object.

To achieve reasonable frame rates live from a Kinect device, forest parameters have been tuned, alongside other optimisation efforts. These modifications are also outlined below.

5.1 Leaf Representation

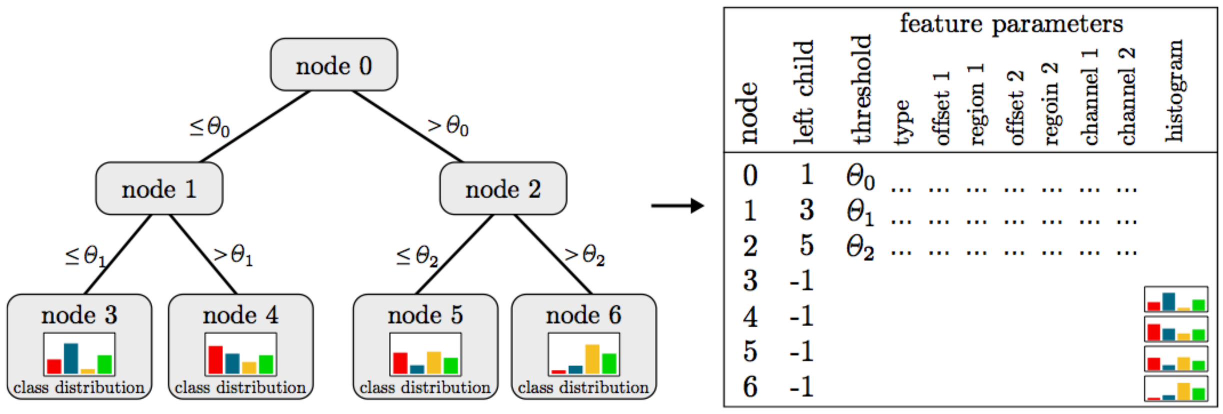
List of Vectors

There are a number of possible choices for modeling the spatial distributions of offsets at the leaf nodes. For example, a Parzen estimate with a Gaussian kernel has been used to reconstruct the distribution from the samples [22]. For the prototype implementation, the simplest possible representation was chosen to create a functioning system; a list of 3D offset vectors where stored at each node [3]. This maintains an exact representation of the distribution at the leaf nodes in continuous space, at the cost of practical memory limitations.

The CURIL library stores tree data in a column of two-dimensional layered textures. A single two-dimensional layered texture is limited to a size of $8,192 \times 8,192$ in compute capability 1.x, and $16,384 \times 16,384$ in compute capability 2.x,3.x, and 5.0. For this project, a GeForce GTX 780 with compute capability 3.5 is used, providing a maximum texture width is 16,384. To store additional data beyond this, another complete column of layers must be used.

The tree data is structured with tree nodes in successive rows in breadth-first order. Each row of a *split* node stores the left child node ID, split function threshold, feature parameters, while each row of a *leaf* node stores the aggregated histograms. Including an integer denoting the size of the list, this allows for 5,458 three-dimensional offsets to be stored with a two-class histogram (ignoring any requirements for storing orientation votes). For 5,000 samples per image, and 300 images, 1.5m offsets must be stored in a tree. If equally divided within a perfectly balanced tree of depth 10 (i.e. 1,024 leaf nodes), only 1,465 offsets should be required at any given node. In practice, however, trees are not perfectly balanced, and offsets within them come in large groups. When a list of vectors larger than the available space arrives, it is simply lost in the representation.

An additional problem with this list format is prediction time. The representation's final use is to vote into Hough space. For a list of individual vectors, each vector must individually be processed and accumulated into Hough space, causing a significant slowdown. A better system would allow for these statistics to be accumulated in a more manageable form as part of training, and then used as groups of similarly placed vectors.



node	left child	threshold	type	offset 1	region 1	offset 2	regoin 2	channel 1	channel 2	histogram
			feature parameters							
0	1	θ_0	...	...	...	...	...	...	...	
1	3	θ_1	...	...	...	...	...	...	...	
2	5	θ_2	...	...	...	...	...	...	...	
3	-1									
4	-1									
5	-1									
6	-1									

Figure 5.1: A schematic of the original CURFIL tree storage structure. Source [61]

Discretised Voting Cubes

The improved current representation of leaf votes discretises a cubic volume around the pixel into a grid of smaller 3D cubes. The size of the volume is set to $1m \times 1m \times 1m$ and the number of bins along each side is set to 8. In total each leaf node requires 512 counters to be stored to represent its surrounding cube in a linear fashion. This alleviates the two problems mentioned above, allowing any number of sample offset vectors to be stored, and also providing a few aggregate weighted vectors instead of many individual ones.

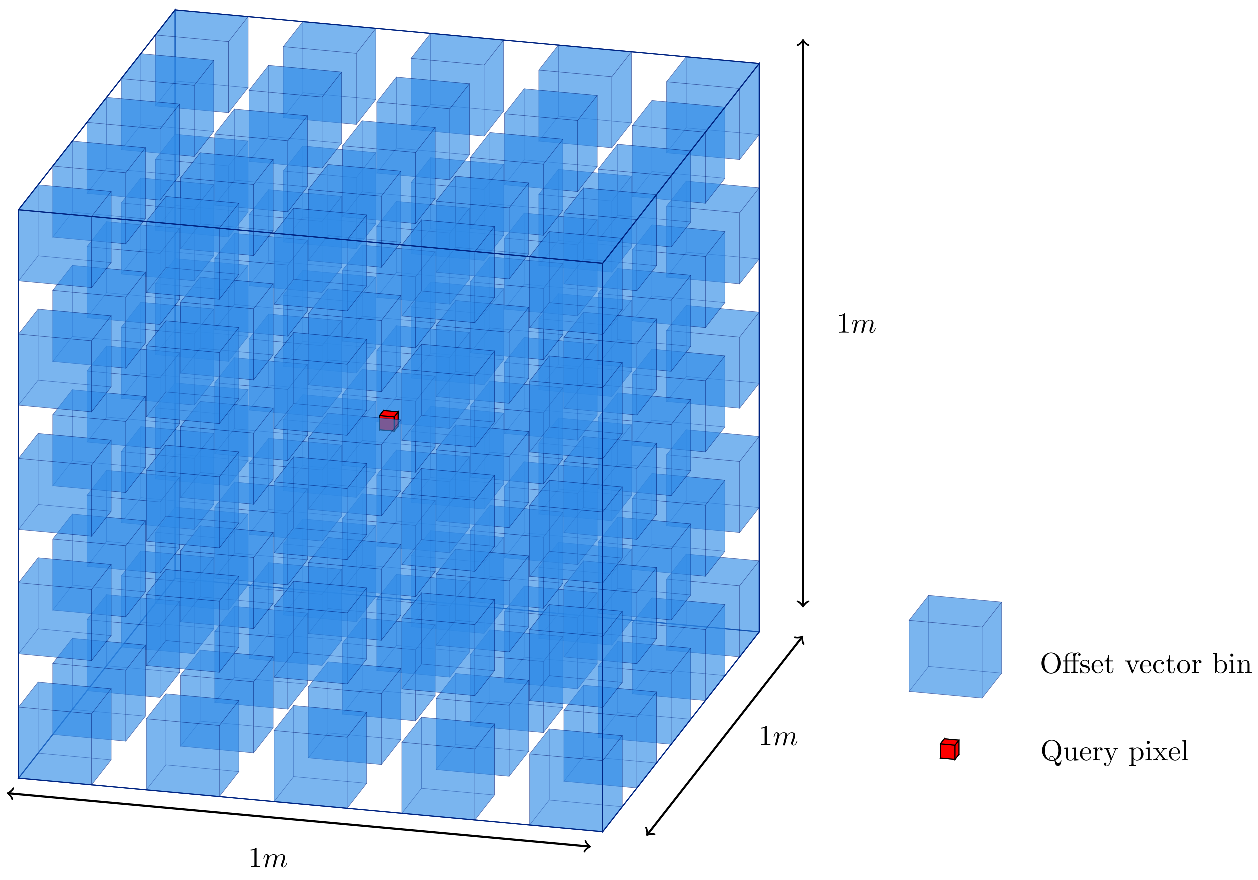


Figure 5.2: A visualisation of the voting space surrounding a pixel. For illustration purposes, a five bin division, with spaces between bins is shown.

This solution comes with a number of drawbacks. The first is that the limitation on the *number* of vectors that may be stored, has been replaced with a limitation on *which* vectors may be stored.

Any votes that fall outside of the cubic volume are ignored. For objects on the same scale as a chair this could actually be an advantage, removing outliers and giving weight only to pixels relatively close to the object centre. Shotton et al. [52] for example explicitly set a threshold to remove votes beyond a certain range. However, since all objects must share the same volume size, an object on a scale larger than 1m may not form an accurate distribution of votes, losing useful feature information as 'out-of-bounds.'

The idea of enlarging the volume to accommodate larger objects raises a second problem, that of resolution. The discrete representation use here contains bins with a side length of 12.5cm, which means that indoor objects on the scale of a coffee mug will all be placed within the single central bin of their Hough Space. This not only gives a very poor estimate of the actual distribution of votes, but also wastes prediction time reading numerous empty bins.

Future Offset Representations

Complex multi-modal distributions, such as a Gaussian Mixture Model (GMM), could help with both the scale and resolution problems highlighted above [25]. This would come with a significant increase in the training cost, as each leaf node must have a distribution estimated for it from the raw vectors. Other representations may result in a faster, more efficient, and flexible description of spatial voting data, and this constitutes an important area for potential investigation. However with a focus on real-time detection and rapid training, the static discretised cubic volume approach has been implemented, despite its limited flexibility.

Orientation Data

To estimate the 6 DOF pose of an object, orientation data must also be stored at the leaf nodes. In the present system it is stored as a 4D normalised quaternion, representing the rotation from the camera frame to the object frame. In a similar fashion to the offsets, quaternion data is discretised into bins. With 4D data, dividing into 11 equally sized sides requires 14,641 floats to be stored. However, it is possible to compress this data into a 3D space. For every rotation, two quaternions exist to represent it, $\mathbf{q}$ and $-\mathbf{q}$. A positive rotation around an axis is the same as a negative rotation about a reversed axis. Using this, the real component, $\mathbf{q}_w$, can be guaranteed to always be positive, by negating the entire quaternion if it is not. It is then possible to store only the normalised $\mathbf{q}_x$, $\mathbf{q}_y$, and $\mathbf{q}_z$ components in 3D space, and retrieve the $\mathbf{q}_w$ from them with reference to Equation 12.

$$\mathbf{q}_w^2 + \mathbf{q}_x^2 + \mathbf{q}_y^2 + \mathbf{q}_z^2 = 1, \tag{12}$$

Since each component must be between -1 and 1, it is trivial to divide the sides of a 3D cube into 11 bins within this range, resulting in an additional 1,331 floats which must be stored at any given leaf node. This representation neatly circumvents the problem of angular wrap-around, which causes averages of angles to potentially face the opposite direction. The modal bin is then sought within the 3D orientation space, and converted back into a 4D quaternion.

Although the system currently implemented functions to a reasonable degree, much future work can be done to improve this representation. The above compression system has been designed with speed in mind, with relatively little thought to the even distribution of quantised rotations. A number of possible methods of encoding 3D unit vectors have been studied, such as projection on to an octahedral [9]. These could be useful for a 4D quaternion analog, however in the same paper it is explicitly noted that "the symmetries are much more complex and it does not scale easily or obviously."

5.2 First Pass: Detection Algorithm

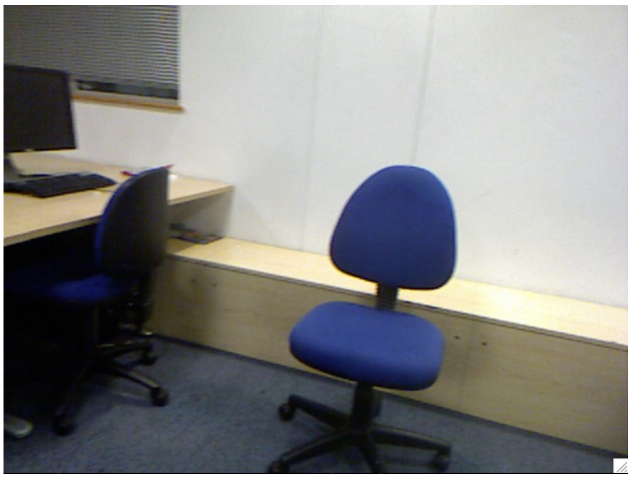
Now that the representation of the decision trees has been discussed, this section will describe the first part of the two-stage test phase [3]. The first pass operates in a unified 3D Hough space. Storing classes separately in both leaf nodes and 3D Hough space results in a significant additional memory requirement should the number of classes grow. Instead they have been combined into a single storage unit at each leaf and these then vote into a single Hough space (See Figure 5.5). It is hoped that these will then form maxima at object centres even for multi-class forests, and the class and orientation of the object can be derived from the nodes that voted for that maxima in a second pass.

A potential area of exploration, is to use the class distribution to give a weighting to the entire voting space for a leaf node for a given class. This class-weighted voting space could then be separately aggregated in a 4D Hough space with an additional dimension for each class. This technique may improve the forest's ability to distinguish classes, and requires only the information already stored in the leaf nodes, along with an expanded 4D Hough space.

Voting

For processing a test image, all decision trees are loaded into texture memory in the format outlined above. Every pixel of the test image is processed by the forest and its votes cast. The CURFIL library prediction algorithm for traversing a tree has already been optimised for 640×480 RGB-D images. Best performance is achieved by using 128 threads to classify an image block-wise with patches of size 8×16 pixels, a system followed here. Every thread processes a pixel for an entire tree from root to leaf, and as all threads access a tree level simultaneously, spatial locality and the cache hit rate are maximised. The CURFIL library also avoids branching by expressing the feature response result as an integer, either 0 for the left child, if $\theta \leq \tau$, or 1 for the right child otherwise. Each decision tree in the forest processes the image sequentially, the implementation therefore scales linearly with the number of trees, as well as the maximum depth of the trees and the number of pixels in the image.

After a leaf node has been reached, the voting space stored within the leaf is converted into weighted vectors, and accumulated in Hough space. The vectors are also weighted by the depth of the pixel as in [3], to account for the smaller projected image size of a distant object. Hough space is discretised in a similar manner to the voting cubes, but modified to represent a frustrum matching the camera viewing volume. This allows higher x-y resolution close to the camera, and lower resolution farther out, expanding to fit the visible scene. It also prevents memory being wasted on space outside of field of vision. The z-axis is limited between a near plane ($0.5m$) and a far plane ($5m$), and discretised into $0.025m$ bins. In total, a discretised space of $160 \times 140 \times 180$ bins are used, with each bin represented by a single-precision floating point format.


(a) The original colour image

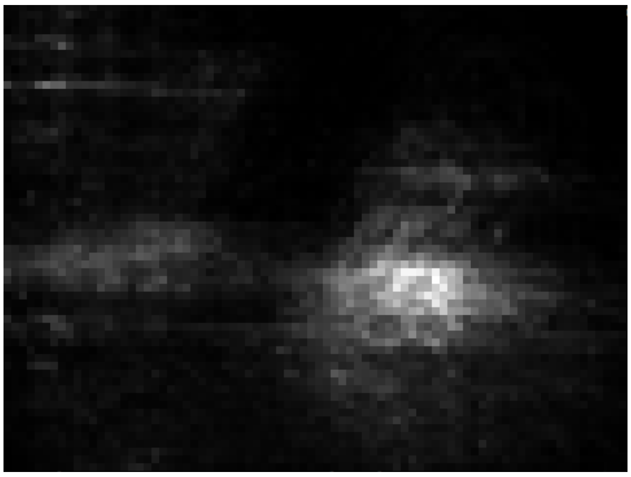

(b) The accumulated hough votes in 3D space

Figure 5.3: The accumulated 3D hough votes from a trained Hough forest on a test RGB-D image

Mean Shift Clustering

To detect object centres, a GPU mean shift clustering algorithm is used. The principle behind mean shift clustering is gradient ascent. A number of (often spherical) windows are initialised within the data space. The centroid of points inside the window is calculated and the centre point of the window is moved to this ‘mean.’ This process is repeated until the windows centre resides at a maxima and no longer moves. Groups of windows that eventually halt within a certain threshold can then be combined. Figure 5.4 illustrates this process with a square kernel.

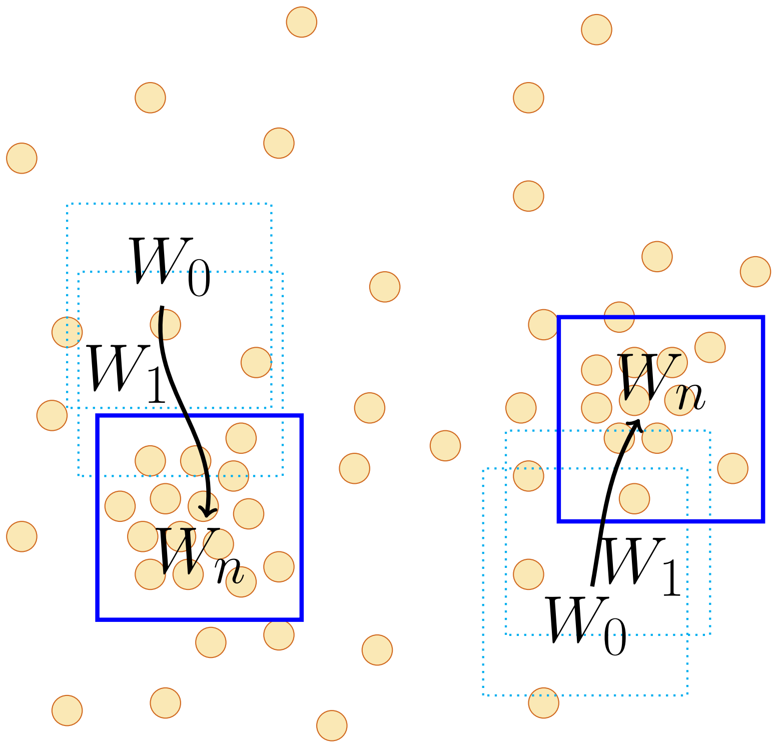


Figure 5.4: A 2D visualisation of the mean clustering procedure with square kernels. The windows W are initialised within the vote space and iteratively move towards areas of highest vote concentration

Many kernel functions exist to weight the points within the window. A Gaussian kernel function for instance gives exponentially less weight to points far from the centre. For rapid detection and simplicity, a flat cubic kernel has been implemented in this project. Each kernels is initialised in an even distribution across the grid of Hough space, and follows the frustum spatial distribution of the space itself. Each kernel is implemented as a separate block of threads, which move independently through the Hough space stored in global memory. The number of threads per block is limited to 512 in compute capability 1.x, and $1,024$ in compute capability 2.x,3.x, and 5.0. So to allow for larger cubic kernels than $8 \times 8 \times 8$, threads are assigned a two-dimensional (x, y) window, and each thread then iterates along the z-axis when calculating the mean.

After the mean clustering, a filtering procedure is done. Windows that lie within a certain minimum distance of each other (0.05m) are combined. The total hough votes within the space is counted, and the number of votes that lie within each window is normalised. The object centres above a certain minimum threshold are sorted according to their voting weight, before proceeding to the second pass.

5.3 Second Pass: Maxima Information Retrieval

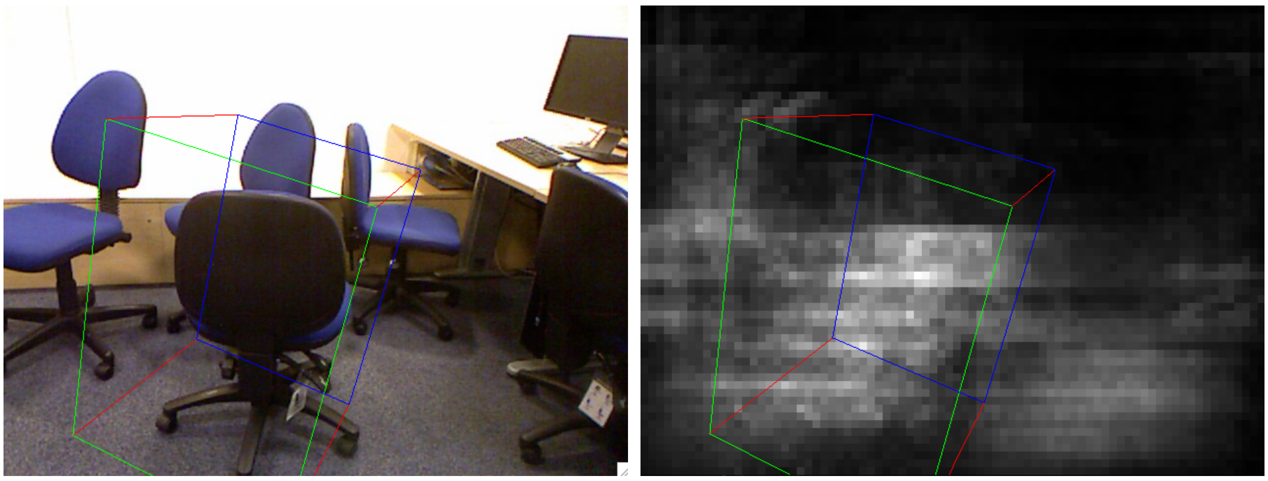
To accomplish both orientation and location voting at once would require at least a 6D voting space, with each point in 3D Hough space storing the (compressed) 3D orientation space described in Section 5.1. At the resolution outlined above this would require 2.06×10^9 4-byte floating-point numbers, which is beyond the entire global memory of a GeForce GTX 780. To overcome this challenge, and also to extract class information in a multi-class Hough forest, a second pass is performed on the leaf nodes that contributed to the maxima.

In the first pass, a 640×480 matrix stores the final leaf nodes reached by each pixel, thereby avoiding a second tree traversal. In the second pass, for each voting bin within the stored leaf nodes, it is checked whether the bin lies within one of the detected object clusters, if it does then the orientation votes stored at the leaf node are weighted by the number of votes within the voting bin, and accumulated within an orientation space stored for that object. At the same time, the objects class distribution is also accumulated, again weighted by the number of votes. This is processed is performed in parallel, again using 128 threads to process leaf nodes in patches of size 8×16.

After voting, the modal orientation bin is sought within the 3D orientation space, and the class with the maximum weighted votes is selected. The orientation bin (denoted with x, y, z coordinates within the space) is then converted back into a quaternion on the CPU with the following equations, where n is the number of voting bins per dimension:

$$\mathbf{q}_x = \frac{x \times 2}{n} - 1 \qquad \mathbf{q}_y = \frac{y \times 2}{n} - 1 \qquad \mathbf{q}_z = \frac{z \times 2}{n} - 1 \qquad \mathbf{q}_w = 1 - \mathbf{q}_x^2 - \mathbf{q}_y^2 - \mathbf{q}_y^2. \tag{13}$$

Finally a pre-measured bounding box is overlayed on the live image in the predicted location and with the predicted orientation of the object (See Figure 5.5). Ground truth annotations on the training set also provides the dimensions of the 3D cuboid in metres, which could in future be used, alongside the predicted class, to properly size the bounding box.



(a) The original colour image, with a projected bounding box (b) The accumulated hough votes in 3D space, with a bounding box

Figure 5.5: A live rendering of the pre-measured projection of the estimated chair bounding box, on a test RGB-D image

5.4 Profiling Object Detection

To enable live detection, all of the above must be accomplished within approximately 50 ms. The initial implementation of the pipeline described above required on average 263 ms to process a frame, when using a forest of 3 trees, of depth 15, and a 100 iteration mean clustering algorithm. The breakdown shown in Table 5.1 highlights areas where optimisation has been most beneficial.

Table 5.1: Performance breakdown of the object detection pipeline. Tested with a Kinect device on a GeForce GTX 780 GPU with a 2.13 GHz Intel Xeon E5606.

Average Time for Operation (ms)	Initial	Using Raw-RGB	Optimised Orientation Voting
Image Preprocessing	139.0	13.2	13.2
Hough Voting (3 trees, 15 levels)	31.1	31.8	32.1
Mean Clustering (100 iterations)	11.7	12.1	10.2
Maxima Information Retrieval	80.9	77.4	6.4
Other	1.2	1.2	1.2
Total Time (ms)	263.9	135.8	63.3
Frames Per Second	**3.9**	**7.4**	**15.8**

Minimising Image Preprocessing

The largest contributor to the slow frame rate was importing the image into a format the tree could process. For trees trained in CIELab space, the raw image from the kinect must first be converted from its native RGB. CURFIL uses the image library *Vision with Generic Algorithms* [60], which takes approximately 120ms for a 640×480 image. The CIELab space has been shown to improve the accuracy of image segmentation in [61]. In the evaluation on the Hough forest test set (See Section 7.2), results did not demonstrate a dramatic improvement in accuracy for object localisation, therefore the much more efficient directly trained RGB tree has been selected for use in live detection. This alone resulted in a 89% increase in the frame rate.

Optimising Orientation Voting

The next largest time sink was the second pass of object detection, when calculating the orientation and object class. This phase does not require tree traversal, and yet was 40ms slower than the combine time for hough voting and mean clustering. This surprising bottleneck was eventually traced to the CUDA atomic operations used for accumulating the orientation votes.

To understand the importance of atomic operations as part of a parallel system, a short example can be quite instructive. A simple program is designed to increment a given memory location by 2, using 2 threads. The threads begin running in lock-step as part of a warp, and both read the value 0. They both store this in their register, increment it to 1, and then both write the incremented value of 1 back to the memory location. Regardless of which thread writes last, a count will be missed. Atomic operations allow this missed update to be avoided, with the guarantee that multiple independent threads will safely read and write as if called sequentially, with the caveat that competing writes must be serialised, and could potentially cause slowdowns [13].

In the detection procedure, it is not known in advance which pixel will be voting where within Hough space. Atomics are therefore essential to ensure that updates are not lost. A similar requirement applies to orientation space, however unlike the Hough space, where updates are dispersed throughout bins largely at random, the *entire* set of votes are added into the detected object's orientation space. The use of atomics for this operation as in Listing 5.1 required serialisation between all of the threads and slowed the process down considerably.

Listing 5.1: The basic quaternion voting procedure

```
// Copy orientation votes into Quaternion space, weighted by the voteWeight
for(unsigned int quat = 0; quat < QuatArraySize; ++quat) {
    unsigned int quatWeight = getQuaternionWeight(quat, currentNode, currentTree);
    atomicAdd(&quatSpace[objectId * QuatArraySize + quat], voteWeight * quatWeight);
}
```

A small change to this procedure produced significant improvements. Instead of each thread starting at the same location, the location is selected by reference to the pixel location, $(loc_y * Image_{width} + loc_x) \bmod Q^3_{dim})$, and each thread is simply required to do a full 'loop' of the quaternion space. This allowed an approximately even distribution across all 1,331 locations to be incremented simultaneously, and reduced the operating time for this operation from 77ms to 20ms. After experimentation, it was discovered that the orientation error for a test set stayed consistent at approximately 65°, even with a decrease in the granularity of the orientation space to $8 \times 8 \times 8$. This change reduced the average maxima calculation time to 6.4 ms on average as shown in Table 5.1.

Listing 5.2: The improved quaternion voting procedure

```
// Chosen starting offset within quaternion space, distributed across pixel location
unsigned int qOffset = pixelId % QuatArraySize;
for(unsigned int quaternion = 0; quaternion < QuatArraySize; ++quaternion) {
    unsigned int quatWeight = getQuaternionWeight(qOffset, currentNode, currentTree);
    // Accumulate the quaternionWeight, along with the voteWeight into quatSpace
    atomicAdd(&quatSpace[objectId * QuatArraySize + qOffset], voteWeight * quatWeight);
    // If the next offset is on the edge of quaternion space, loop back to finish
    if(qOffset++ >= QuatArraySize) qOffset = 0;
}
```

6 Dataset Collection

This chapter describes the collection of a dataset using the KinectFusion-based annotation tool presented in Chapter 3. The collection of this dataset is not only required for measuring the performance of the Hough forest implementation, but is also in many ways an evaluation itself on the usability of the tool application. Two datasets were captured and annotated, both of which contain three classes of chair. The training set consists of 6,073 unoccluded images of single chairs in two different backgrounds. The test set is composed of 8 video sequences with multiple object instances and classes, often with partial occlusions. A more complete description of these datasets is given below.

6.1 Training Dataset

To test the speed and accuracy of the full object detection pipeline, an evaluation dataset has been captured. The ubiquity of chairs within an indoor environment, makes them a suitable candidate for use by a mobile robot SLAM++ system. The evaluation (See Section 7) measured the effect of a number of parameters when training a single class, and also at how scalable it may be for multiple classes, voting within the same Hough space. Three different classes of chair, shown in Figure 6.1, were selected. Class 1 (Figure 6.1a) is used in the single class evaluations, and the other two classes are used alongside the first when experimenting with a multi-class system. Their similar size and shape operates well within the confines of the statically sized voting space, and allows the forest accuracy to be evaluated on different classes with a similar appearance.

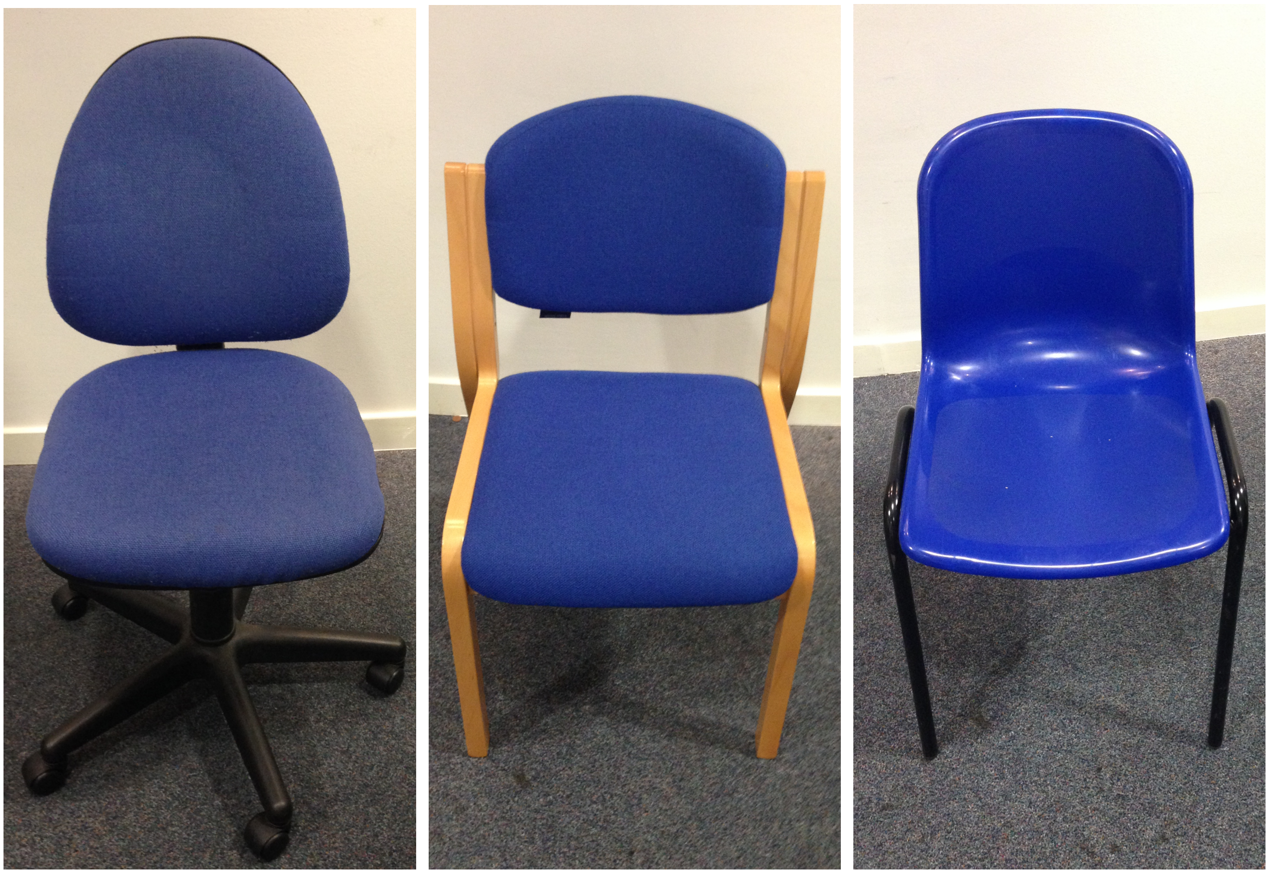

(a) Class 1 (b) Class 2 (c) Class 3

Figure 6.1: The three classes of chair used in the evaluation dataset

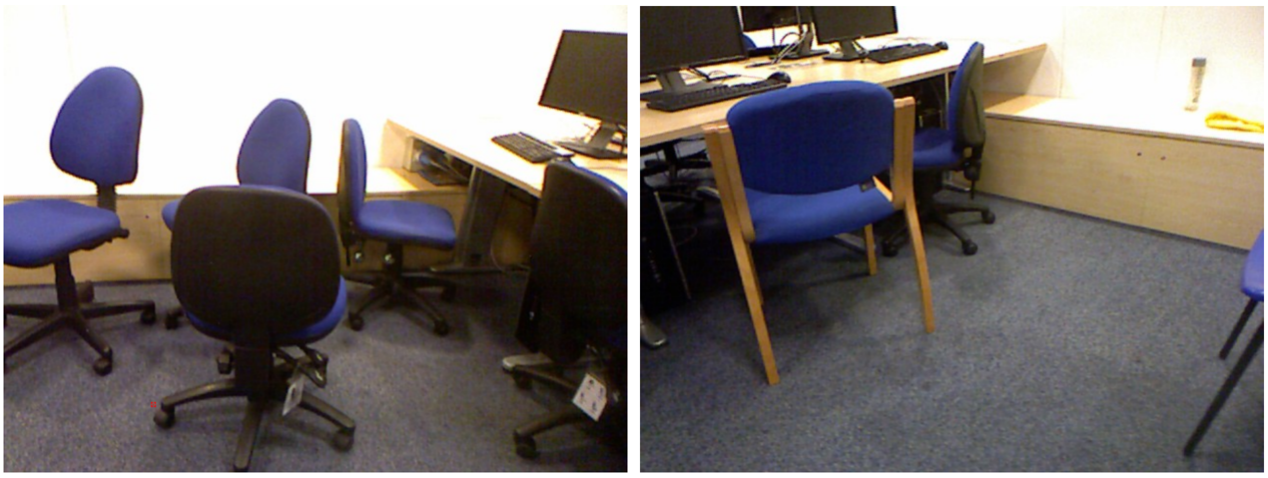
The training dataset consists of a series of RGB-D videos of the chairs within two different indoor scenes. Each training video is of a single instance of the unoccluded chair. To provide even coverage, recording was divided into four quadrants. For each quadrant, a near (1-2m) and far

(2-3m) video was captured to provide some variation in scale as well as additional training data. Distances greater than 3m resulted in tracking problems within the annotation tool, and less reliable ground truth estimates. The impact of this limitation should be negated by the depth normalisation of the feature parameters, which allows objects more distant than that seen in training data to still be recognised.

The videos were first captured in the OpenNI video format using a Kinect device. These videos were then converted into training images and annotated using the annotation tool described in Sections 3.4 and 3.5. The final training dataset consisted of 6,073 images, evenly divided between the three classes.

6.2 Testing Dataset

The test dataset is recorded in a third indoor location, different from either of the two training environments. It consists of eight separate video sequences. Scenes are designed to be similar to the kind that might be encountered by a mobile robot in the real world, and feature multiple object instances at varied orientations, often with partial occlusions and background clutter. The first two sequences where designed for testing a single class system, and only contain instances of Class 1. The remaining sequences feature a mixture of the three different chair classes. The final test dataset consists of 472 images, with 122 captured from the first two scenes.


(a) An example from the first scene, featuring multiple instances of Class 1 and partial occlusions.

(b) An example of a scene with a mixture of different classes, along with partial occlusions.

Figure 6.2: Examples taken from the evaluation test dataset.

For the testing dataset, the same limitation as for the training dataset applied when annotating a scene at distances greater than 3m. Unfortunately, this limitation prevented a systematic evaluation of the accuracy of the Hough forest at different ranges. Further work needs to be done to improve the effective range of the annotation tool. Alternative means of testing could also be examined, such as an artificial test dataset which has perfect ground truth estimates at any desired range.

7 Evaluation

Rapid training has allowed a great number of forest architectures to be tested on the evaluation dataset. The majority of the evaluation is on a single class object detection system, with Class 1 (Figure 6.1a) being used for this purpose. Evaluating on a single class removes the variability of multiple object classes voting into the same space, to gain a clearer understanding of the effect of different parameters on the accuracy of the Hough forest. After this, the capacity to learn and detect multiple objects is then evaluated. The final result of the evaluation is a multi-class 'Fast Forest' architecture, capable of being trained in 90 seconds, with an average precision of 61.5% on a single class, and 30.9% on our multi-class dataset.

7.1 Evaluation Methodology

A 'Standard' Forest

For most of the following evaluation, the effect of modifying a single parameter at a time is evaluated. It is useful to define a default set of parameters from which to vary a selected parameter, a 'standard' forest. The parameters of this forest are set out in Table 7.1.

Table 7.1: Default parameters of the standard Hough forest

Parameter	Value
Number of Trees	10
Number of Training Images	300
Number of Features Evaluated	5750
Number of Thresholds	20
Samples Per Image	4500
Max Depth	15
Min Samples Threshold	200
Max Patch Offset Radius	60
Max Patch Size	3
Probability of Uncertainty Offset Objective	50%
Colour Space	CIELab

Average Precision

For evaluation, a set detection sphere of radius 30cm around the object centre is used to define a positive detection of an object. A single object can only be detected once, any subsequent predictions which lie within the ground truth sphere are ignored; not counting towards the number of correct predictions or the number of mistakes. As predictions are sorted by weight, this means the prediction with the highest number of votes within the objects detection sphere is chosen.

To calculate the average precision, the area under a precision-recall curve has been calculated. Ordinarily, precision-recall curves can achieve recall values of 100% by simply setting the threshold weight requirement to zero, which will return every possible result, including all of the correct ones. With mean-shift clustering, it was found that setting the threshold to zero often did not result in a recall rate approaching 100%, because the available kernels sometimes did not settle in a location where the ground truth object lay. To remove the variability of the mean clustering detection algorithm to evaluate the performance of the raw tree votes, an evenly distributed grid is instead overlayed within Hough space. The total normalised weight of votes within each cube of the grid is compared to the threshold, which is varied to produce the precision-recall curves. This threshold is incremented by 0.2% until the recall fell to zero, and the area under the curve is calculated to give the average precision.

Average Orientation Error

For the orientation accuracy the angular difference, d, between the ground truth, $\mathbf{g}$ and the prediction, $\mathbf{q}$, is calculated as

$$d = cos^{-1}(2(\mathbf{g} \cdot \mathbf{q})^2 - 1). \tag{14}$$

This is then averaged for the test set. Parameter changes had relatively little impact on the orientation, which stayed in the range between 60-70° in all cases. This considerable error may be caused by a number of factors, each of which merit further investigation.

Firstly, the coarse-grained simple modal quaternion voting space could be improved with a more granular space and a second mean clustering algorithm. Some of the average error could be accounted simply by the error inherent in the discretisation process. Secondly, the 'absolute' orientation voting framework used here, where each vote corresponds to an orientation directly from the camera frame to the object frame, could be a source of inaccuracy. An alternative method used by Badami et al. [3], calculates a relative orientation from the query pixel's surface normal to the object axis. In fact, this was done for a frame of reference defined by each of the offset patches, and votes cast for every split node, not just at the leaf.

7.2 Single-class Parameter Evaluation

Maximum Depth

The chosen tree depth must be deep enough to allow a nuanced model of the training data to be captured, while avoiding going so deep as to overfit to the training data [10]. For this project, training time is of utmost importance, and for the most part, training time scales linearly with tree depth. It is therefore essential to explore the tradeoff between tree depth (and hence training time) and accuracy. Figure 7.1 shows the average precision for different depths on the test dataset. For maximum depths greater than 17 on the standard forest, a decline in accuracy occurs. Much of the benefit of depth is already apparent in trees as deep as 14-15 levels.

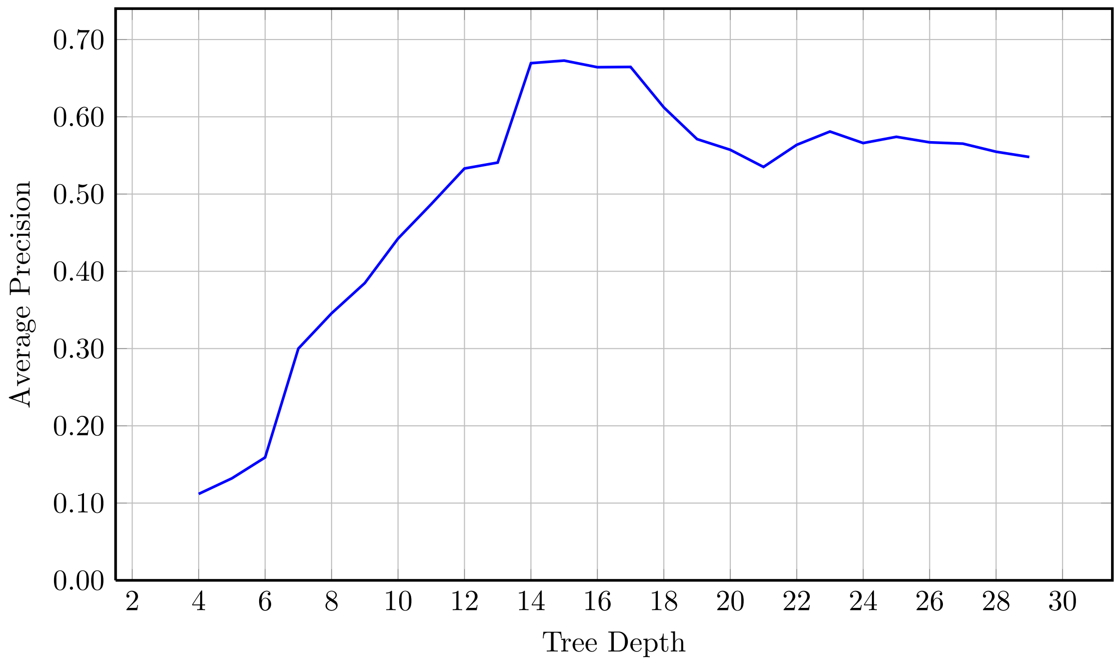
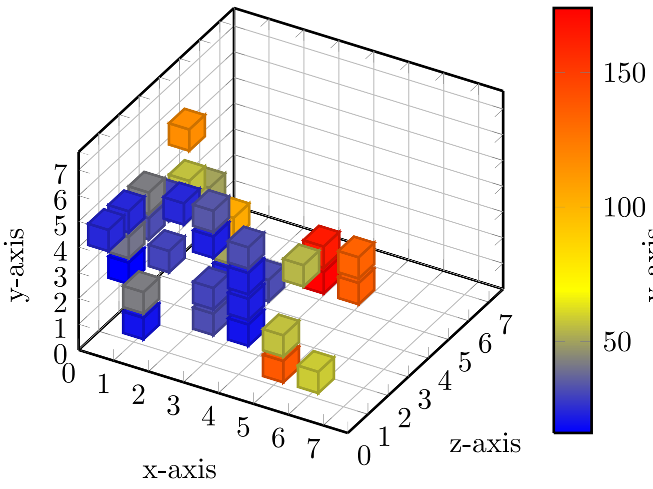


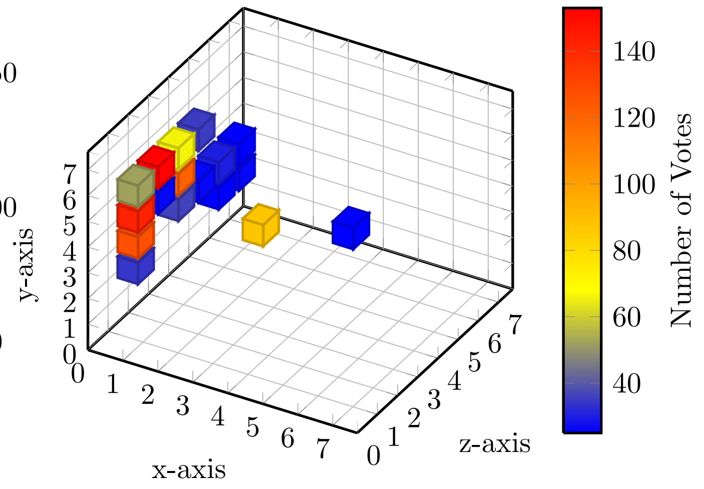
Figure 7.1: Average precision vs. Maximum Tree Depth

Offset Uncertainty Analysis

To measure the benefits of implementing the second objective, a sliding scale of probability has been implemented. At 0% every split node test is aimed at maximising information gain between classes, at 100% offset uncertainty is always minimised. Figure 7.2 shows example leaf nodes trained under each of these scenarios. The votes are shown to be much more clustered in Figure 7.2b. The average precision with varying probabilities of objective can be seen in Figure 7.3. As expected the offset uncertainty objective appears to improve the accuracy of the forest up to a certain threshold. Beyond 80% there is a notable decline in accuracy, most likely as a result of background features being poorly separated from the chair class.



(a) 0% probability of minimising offset uncertainty.



(b) 100% probability of minimising offset uncertainty.

Figure 7.2: Two example leaf node voting distributions, trained with different objective measures.

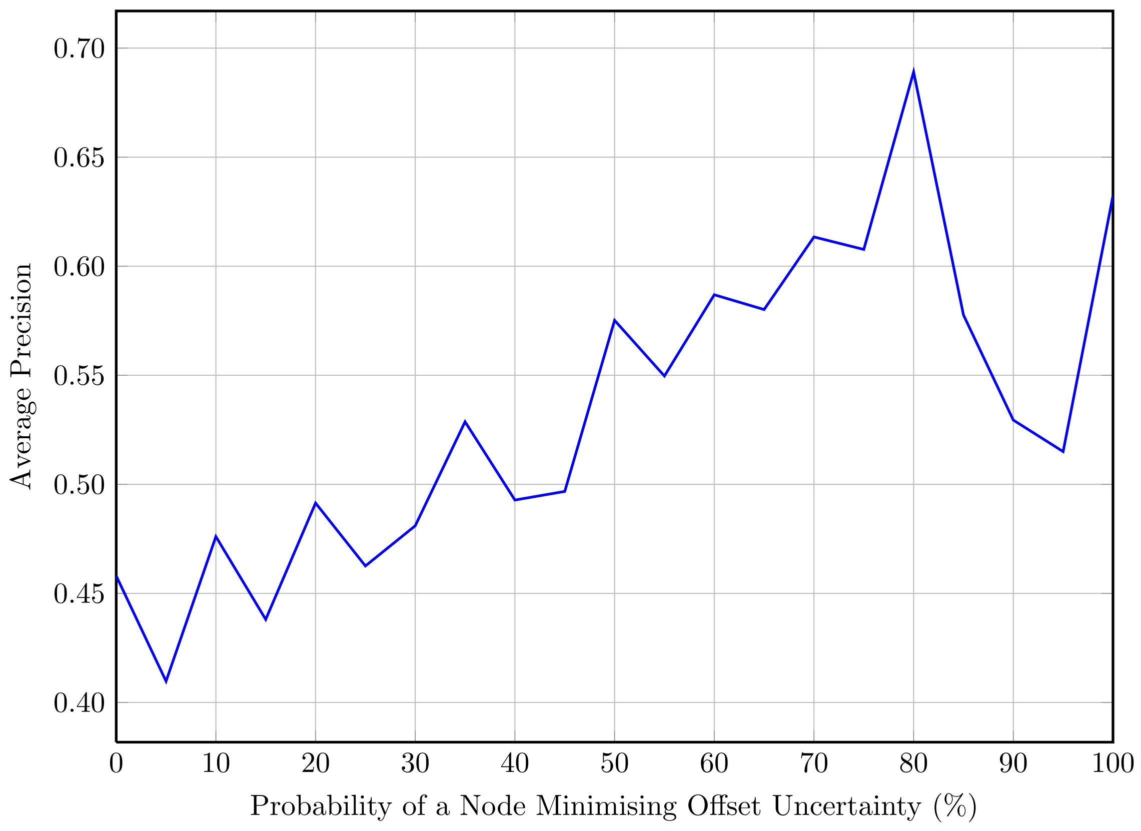
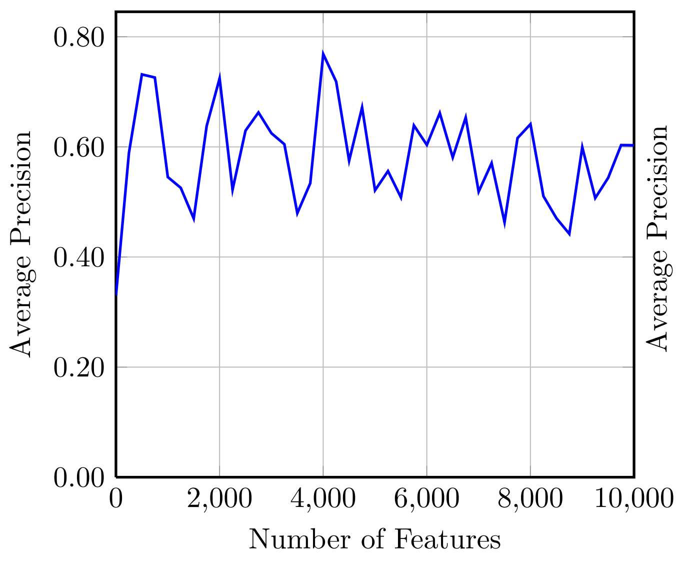


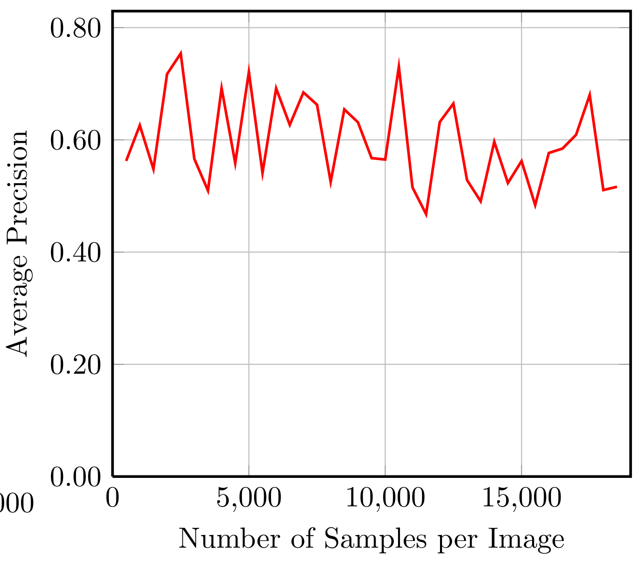
Figure 7.3: Average precision vs % Offset Uncertainty

Number of Features and Number of Samples Per Image

Training time scale linearly with both the number of features evaluated and the samples per image. Surprisingly, there does not appear to be a link between substantial increases in either of these parameters, and an improvement in precision. The one caveat to this, is that there is a noticeable reduction in average precision when only a single feature is evaluated - in a manner similar to extremely randomized forests. However above this minimum threshold, a reasonable average precision may be attainable with relatively few samples per image and features evaluated. This possibility is evaluated in Section 7.4.

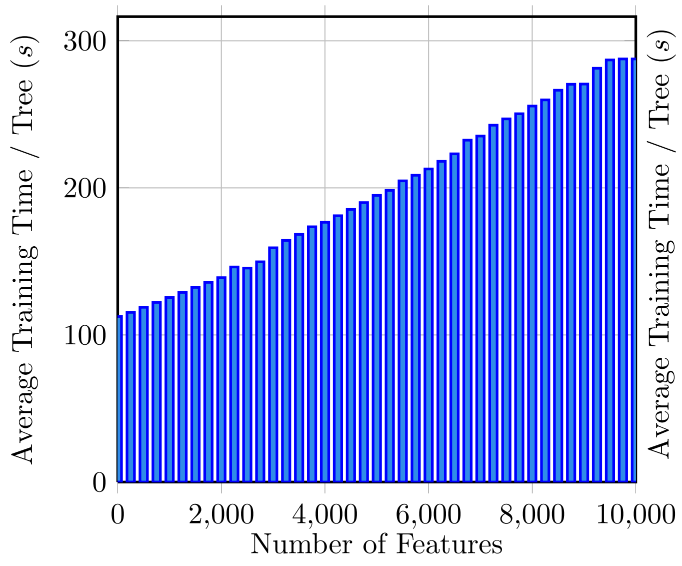


(a) Number of Features vs Average Precision.

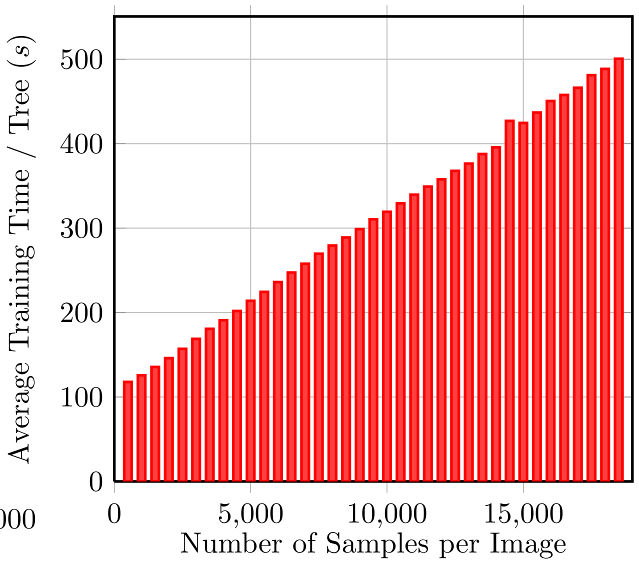


(b) Samples Per Image vs Average Precision.

Figure 7.4: The effect of feature count and samples per image do not show a consistent relationship against average precision except at particularly low levels.



(a) Number of Features vs Average Training Time.



(b) Samples Per Image vs Average Training Time.

Figure 7.5: Training time scales linearly with both feature count and samples per image.

RGB vs CIELab

In live object detection, the conversion of the raw RGB image into the CIELab space for feature response calculation results in a significant slow-down. The standard forest was evaluated using each, the RGB forest performed with an average precision of 59.7% vs CIELab's 62.8% (See Figure 7.6 for the precision-recall curves). To achieve the goal of real-time detection, the loss of 3% average precision appears warranted.

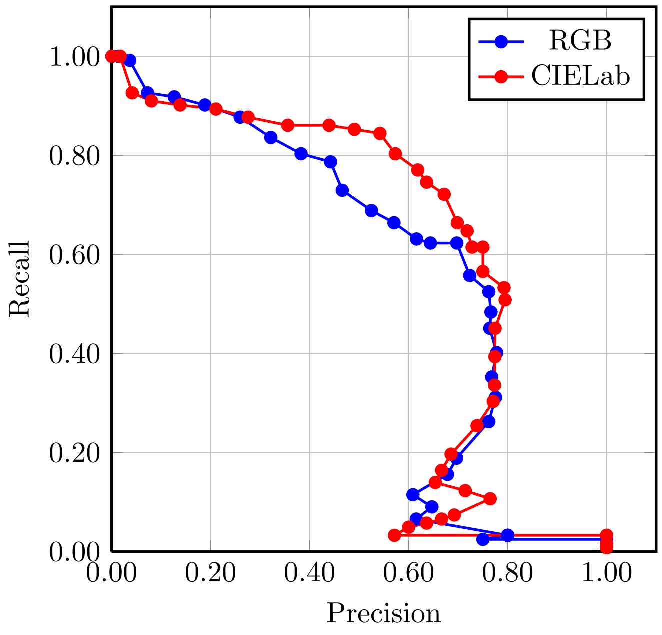


Figure 7.6: The precision-recall curves for RGB and CIE colour spaces

Post-training Leaf Augmentation

In post-training new samples are passed down an already trained tree to gain an improved distribution at the leaf nodes. This is not only fast, but could help alleviate overfitting. Unfortunately, the evaluated performance of the process is less persuasive (Figure 7.7). No improvement in precision can be seen, and in all but one of the trained forests average precision declined.

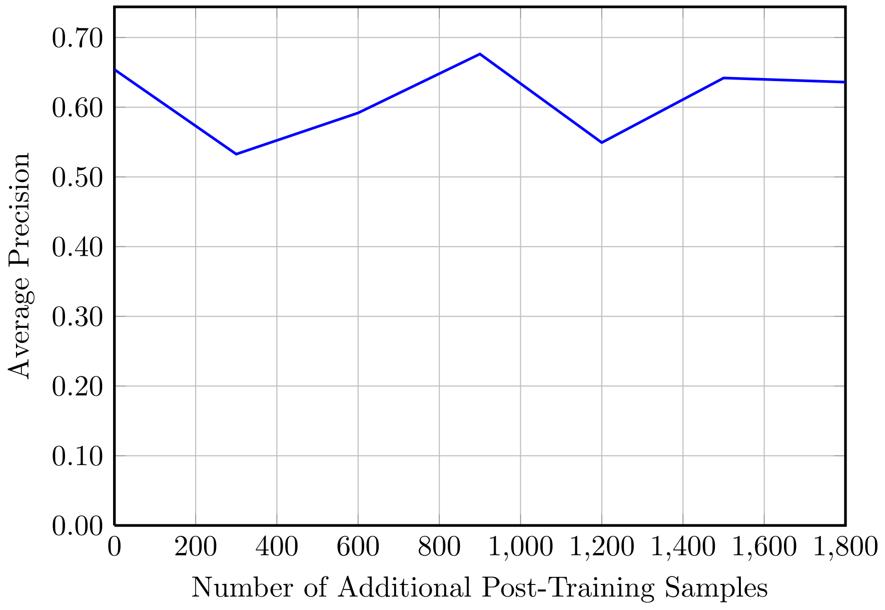
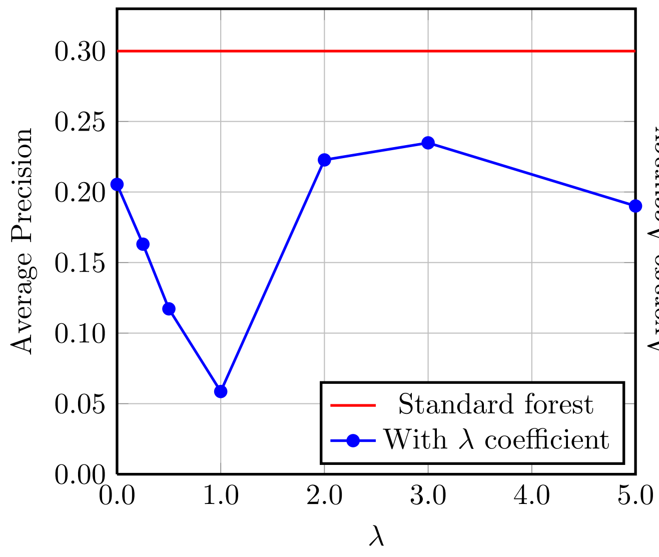


Figure 7.7: Average precision vs. Number of Additional Training Instances

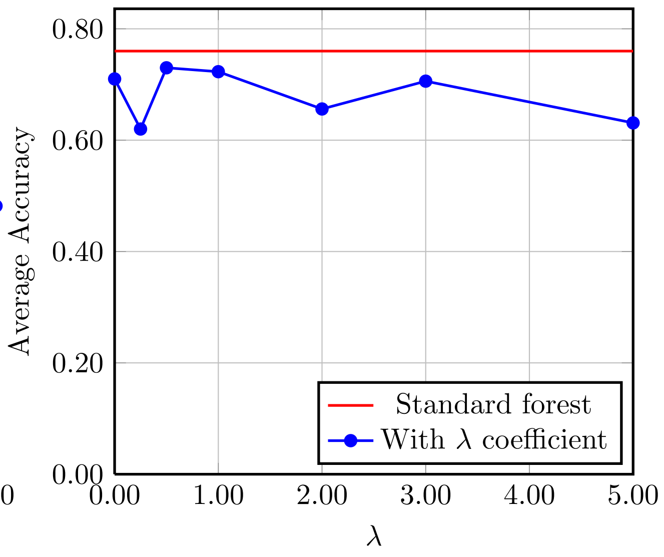
7.3 Multi-Class Evaluation

In this project, the computational overhead of additional classes is minimised by training a multi-class forest which votes into a single unified Hough space. Potential object centres are then efficiently processed to retrieve information such as class and orientation. The full 3-class training dataset was used to train a Hough forest, which is then evaluated on the test set of 472 images, which features the mixture of classes, occlusions, and background clutter typical of an indoor environment. The average precision calculation as described above is maintained in this evaluation. For the multi-class test, this means that a correctly predicted object centre must also predict the correct class. Alongside this, a class-prediction Confusion Matrix for the votes within the ground truth sphere is calculated, to measure the stand-alone accuracy of the maxima information retrieval method.

Section 4.3 details a modification made to the information gain calculation. The benefits of this modification, with different λ values was measured, and the results are shown in Figure 7.8. Unfortunately, the modification appears to marginally reduce both precision and accuracy. This could due to the similar appearance of our classes, and would be worth investigating on larger datasets. Despite this, the results for the multi-class test appear generally positive, when considering the similar appearances of the different classes, with an overall precision of 30%. The average classification accuracy of 76% appears particularly robust, supporting the efficacy of a two stage voting process.



(a) The average precision vs the background information gain score parameter, λ.



(b) The confusion matrix average accuracy vs the background information gain score parameter, λ.

Figure 7.8: The multi-class modification decreases performance on the evaluation dataset.

Label-Prediction	Class 1	Class 2	Class 3
Class 1	0.58	0.03	0.4
Class 2	0.04	0.79	0.17
Class 3	0.09	0.01	0.9

(a) The 'Standard forest' confusion matrix, with an average accuracy of 76%.

Label-Prediction	Class 1	Class 2	Class 3
Class 1	0.55	0.09	0.36
Class 2	0.05	0.79	0.16
Class 3	0.14	0	0.86

(b) The modified forest, with a tuned λ parameter of 3.0, results in slightly worse average accuracy of 73%.

Figure 7.9: The confusion matrices for the standard forest, and with the tuned information gain modification. Labels are denoted with the y-axis, and predictions along the x-axis.

7.4 Fast Forest

The above analysis exposed a number of areas which lead to significant training or testing cost, without exhibiting a significant improvement in performance, such as number of features evaluated and CIELab colour space. It has also highlighted areas that do improve performance, at least within the relatively small evaluation dataset presented here, such as depth and proportion of offset uncertainty nodes. This information has been used to develop a multi-class 'Fast Forest,' the parameters of which are given in Table 7.2.

Table 7.2: Parameters for the Fast Forest

Parameter	Value
Number of Trees	3
Number of Training Images	300
Number of Features Evaluated	250
Number of Thresholds	20
Samples Per Image	2000
Max Depth	15
Min Samples Threshold	200
Max Patch Offset Radius	60
Max Patch Size	3
Probability of Uncertainty Offset Objective	80%
Colour Space	RGB

This forest was trained in 90.4 seconds on a GeForce GTX 780, and performs with an average precision of 30.9% on the multi-class dataset (61.5% for single-class). Figure 7.10 shows the precision-recall curves of both, benchmarked against the standard forest. The average precision is 1.0% better than the standard forest (29.9%) for the multi-class evaluation, and only 1.3% worse for single-class evaluation (62.8%). However, the standard forest takes over 19 minutes to train against the fast forest's 90 seconds.

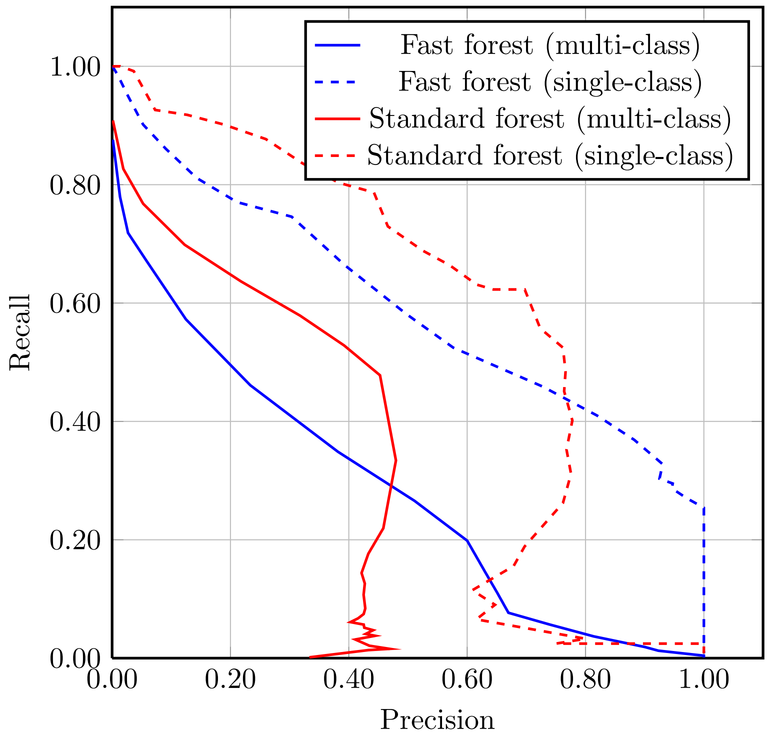


Figure 7.10: Precision-recall curves for the Fast forest and the Standard forest. A significant decline in precision occurs in the multi-class tests.

8 Conclusions and Outlook

The semantic vision community is traditionally focused on detection accuracy, with little regard to computation time, leaving much work to be done on real-time detection systems. This project aimed to produce a Hough forest for mobile robots that could be rapidly trained on RGB-D data to localise an object within a new environment. The pipeline has been developed on consumer-grade hardware, leveraging the parallel power of GPGPUs, along with increasingly popular low-cost RGB-D devices, such as the Kinect device.

Hough forests have previously been tailored for use with RGB-D data. Badami et al. [3] successfully and accurately trained a class specific forest to predict the pose of an object within a full 6 DOF on the RGB-D Object Dataset [34]. This project contributes a number of advancements on previous research. A functioning optimised GPU Hough forest has been developed along with a live object detection system which uses dense prediction on full resolution RGB-D video. The speed of this process opens up the possibility for learning new objects in the field, and quickly reaping the rewards of object recognition.

The accelerated training and testing pipeline has enabled hundreds of different forests to be systematically trained and evaluated, including a number of experimental training procedures such as the post-training leaf augmentation and retrospective maximum depth tuning. This large scale evaluation process has led to one of the key achievements of this project; the development of a 'Fast Forest,' which can learn to detect a single object with 61.5% average precision in 90 seconds on a GeForce GTX 780.

Training and testing the hough forest however, are not the only contributions made. This project has also removed the requirement of a controlled environment in which to collect the ground truth data for training such a system, replacing it with a ground truth annotation tool integrated into a state-of-the art SLAM pipeline (KinectFusion [32]). This tool requires minimal user input and operates for the duration of a video sequence.

8.1 Areas for Investigation

There still remains much future work be done, some of which has been highlighted in the course of this report. The following are some key additions that could enhance the future performance of the current system:

- If the forest proves accurate enough to initialize ICP, a post processing stage could serve to further refine the predicted position and orientation of the object.
- For live detection, each image is currently processed independently from the previous image. Better use of temporal information, such as with a Conditional Random Field, or with a Kalman filter, should result in improved performance.
- At present, only the default colour and depth features implemented in the CURFIL library are being used for detection. Additional features such a Histogram of Gradients, surfel pairs, or Sobel filters could be added and evaluated.
- The orientation system, which votes directly on the rotation from camera pose to object pose, could be replaced with a relative rotation about a frame of reference defined via the query pixel's surface normal [3].
- Only a small dataset, with relatively few similar classes have been explored here. Further experiments could be done with a larger, more varied dataset. For the forest to operate effectively under these conditions, some of the below modifications may also be required.
- The leaf node offset is static and of a relatively low resolution. Efforts could be made to evaluate other representations, such as the Gaussian Mixture Model, for use in live detection. Similarly, the distribution of orientation votes within the compressed quaternion space merits

further investigation, as well as a comparison to a projective 4D octahedral encoding approach if possible.

- The relaxed offset uncertainty equation could be implemented to evaluate the level and effectiveness of feature sharing between differently shaped objects. This should allow the system to cope more effectively with a greater number of object classes.

In addition to the above investigations are some fundamental alterations that may prove necessary to be properly incorporated into a real-time recognition system. The current system still relies heavily on batched data for accelerated training, however in real world scenarios the random forest must be capable of being trained incrementally as new images arrive. It must also be robust to extended periods of learning. Saffari et al. [47] has proposed an algorithm for growing random forests in an online manner, by combining online bagging and extremely randomized forests. A feature which may prove essential, if a tree allowed to grow indefinitely, is an ability to 'forget,' an adaptive tree structure that would allow pruning of less useful branches [47]. Again, for rapid training, these procedures will likely require modifications for acceleration on a GPU.

8.2 Future Work

The motivation expressed at the beginning of this report outlined the ideal long term goal of unsupervised real-time object recognition. The focus of this project has been on the object recognition phase, however the unsupervised first stage is the essential component for development of a fully autonomous mobile robot, and research in this area still requires a huge amount of work.

Current techniques for unsupervised object discovery predominantly use a combination of geometric segmentation and clustering algorithms. Triebel et al. [59] for example, used a Conditional Random Field to model the interdependence of segmented object parts to learn repeating objects in an indoor scene. Active learning is another technique which focuses on allowing the robot to learn objects by manipulating its surroundings in much the same way an infant learns [31]. The lack of active manipulation is a key informational asymmetry between a human's learning experience and that of a passive object segmentation system.

Some of the most exciting advances in unsupervised learning stem from research into Deep Neural Networks, which have exhibited unprecedented performance at certain tasks, and may provide just the support needed for autonomous robotics. A Convolutional Neural Network with Reinforcement Learning successfully learned to interact with, and complete, a number of Atari games without any 'game specific' modifications to the algorithm. It operated with only the raw video as input and controls as output [19]. Although still in a simple 2D setting, this ability to successfully navigate and interact with virtual realities autonomously shows the potential for this kind of learning algorithms to operate in the real world. Research into this area shows much promise, but there is still a lot to be done.

Appendices

Appendix A

User Guide: Annotation Tool

Command Line Options

This guide will walk through a typical data collection use case for the annotation tool. For details on installing the software, please see the documentation in the actual code base. To begin annotating training data directly from the Kinect a single flag (`-wtd` or alternatively `--write-training-data`) is required to the specify the directory where training data will be stored:

```
$ ./ground_truth -wtd "/path/to/training/directory"
```

It is recommended to use a new folder for each video, as subsequent annotations to the same folder will overwrite the old data. To annotate data from a pre-recorded ONI format video, the `-if` (or `--input-file`) flag is used:

```
$ ./ground_truth -if "/path/to/training/video.oni" -wtd "/path/to/training/directory"
```

Two additional commonly used flags exist to specify the size of the cubic KinectFusion model in metres (`-ms` or `--model-size`) and the resolution of the model in voxels per dimension (`-vs` or `--volumetric-size`). The default for these parameters is 2m, and 256 respectively. The following command specifies a 3m volume and 512 voxel resolution:

```
$ ./ground_truth -wtd "/path/to/training/directory" -ms 3 -vs 512
```

An exhaustive list can be viewed by running with the `-h` flag.

GUI Annotation Tool

After running the above command lines, a GUI will appear with a 3D rendering of the input. The GUI consists of a number of sub-windows (Labeled in Figure A.1), the two important sub-windows for ground truth annotation are 'Object Mask' and 'Model Render.' Arrow keys allow a user to rotate the viewpoint in the Model Render.

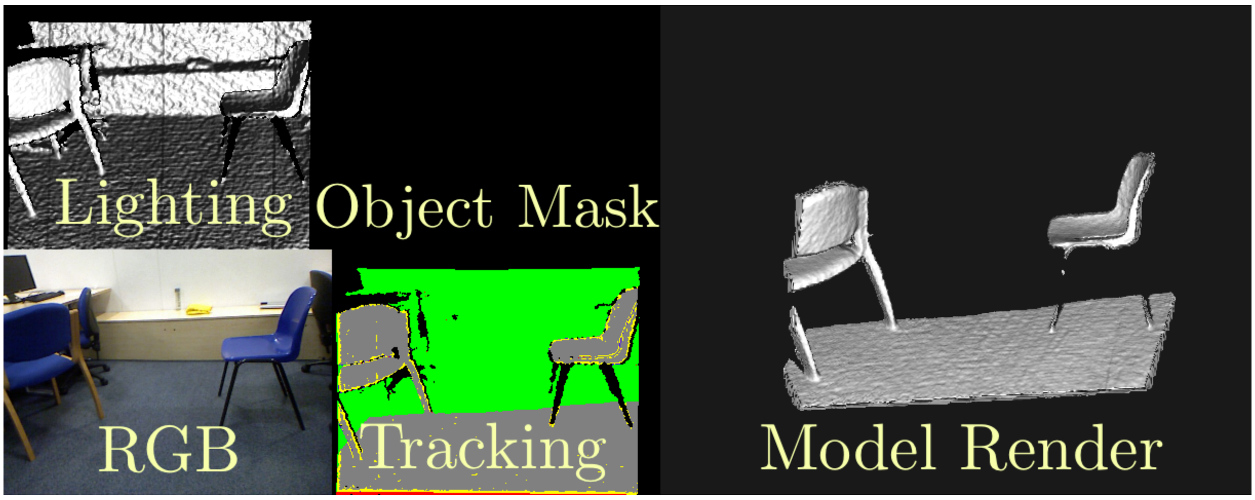
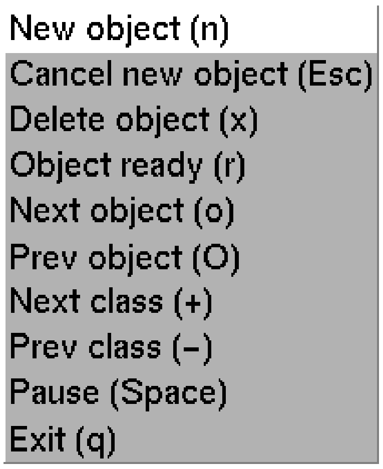
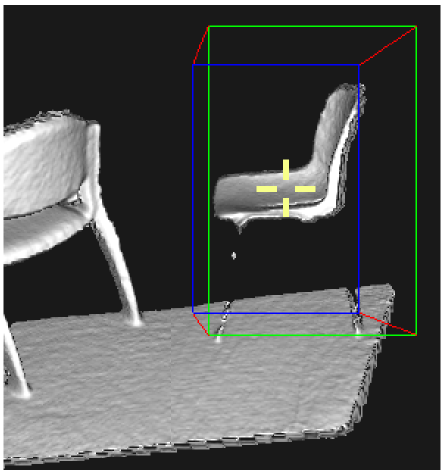


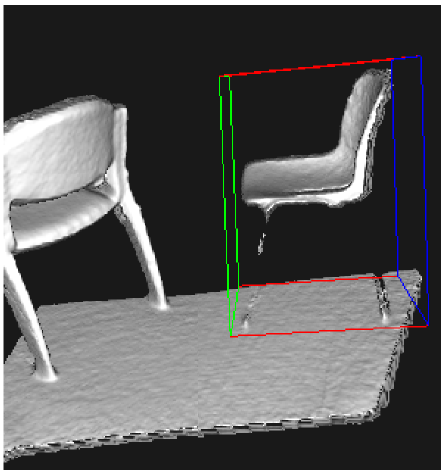
Figure A.1: The initial view of the ground truth annotation tool, with sub-windows labeled.

Right-clicking on the window provides a drop down menu (shown in Figure A.2a), from which the class type (denoted by an integer) can be incremented and decremented (or by using the `+` and `-` keys). Selecting a new object will produce a cross-hair cursor which allows a bounding box to be specified by clicking a point located on the 'Model Render' sub-window.



(a) The drop down menu.


(b) View after selecting an object.


(c) View after rotation and scaling.

The most basic annotation is now complete. If the user wishes, training data can immediately be output by specifying the object as 'ready' either by pressing `r`, or selecting the 'Object ready' option from the drop down menu. It is also possible to more precisely position, rotate, and scale the bounding box, before marking it as ready, as shown in Figure A.2c. `Left click drag` translates the bounding box in the x & y-axis, and `Ctrl-left click drag` translates along the z-axis. `Middle click`, and `Ctrl-middle click` operates in the same way, but for scale. An appropriate cursor is displayed to inform the user of the current transformation in progress. Rotating an object is performed with the `w,a,s,d` keys; `a,d` for the y-axis, `w,s` for the x-axis, and `Shift-w,Shift-a` for the z-axis.

When marked as ready, the Object Mask window will display a render of the object mask, corresponding to the view shown in the RGB window (See Figure A.2). Multiple classes can be annotated by selecting a new class with the `+` and `-` keys before creating a new object, and again marking it as ready. Each class is rendered with a unique colour. Bounding boxes can be edited after an object has been specified as ready, the `o` and `O` keys allow users to cycle through object bounding boxes and edit them, or delete them with the `x` key. It should be noted, that training data is not updated retrospectively, and only future frames will be affected by editing the bounding box. The training data is now ready for use.

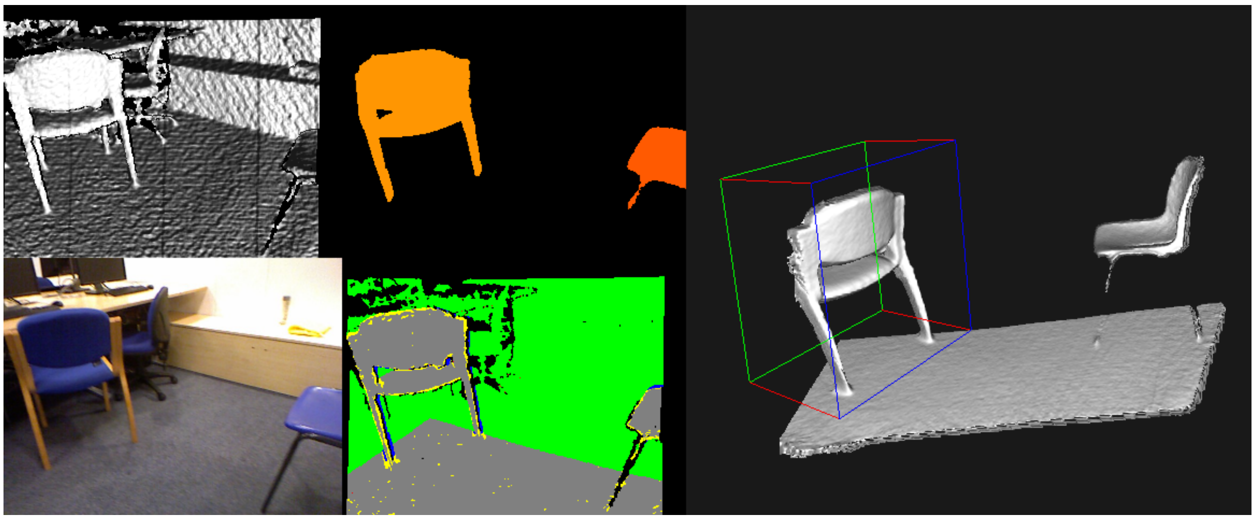


Figure A.2: The GUI view after selecting and readying two objects of different classes.

Appendix B

User Guide: Hough Forest Training

Training

This guide will walk through training a fast forest. Training is done entirely at the command line, using the `curfil_train` program. The large number of command line options make it useful to store them in a simple shell script. Listing B.1 can be copied into a `train.sh` file and updated for a users directory structure. The executable path must be updated to point towards the compiled `curfil_train` program (see the code base's `README.md` file for compilation instructions), the `--folderTraining` option is the training directory, or parent directory, given by the `-wtd` flag to the annotation tool, and the `--outputFolder` specifies the directory to save the random tree files, which follow the naming convention `tree-*.json.gz`. The updated shell script can then be made executable and ran with the following commands:

```
$ chmod u+x train.sh
$ ./train.sh
```

The output will display the current progress in training the forest. For a more verbosely timed output, the `--profile true` flag can be set. The `--trainAllDepths true` option is important if the depth is to be varied retrospectively with the live detection software, which will be described in Appendix C. Running the `curfil_train` program with the `--help` flag gives a description of all of the options available, and their purpose. After the command completes, a trained forest will be saved in the `--outputFolder` directory. For the curious, the saved JSON can be uncompressed and viewed with the following commands:

```
$ gzip -d tree-0.json.gz
$ editor tree-0.json
```

Listing B.1: An shell script for training the Fast forest

```
#!/bin/bash
/path/to/curfil/build/src/curfil/curfil_train \
            --folderTraining "/path/to/training/directory" \
            --outputFolder "/path/to/tree/directory" \
            --trees 3 \
            --maxTrainingSamples 300 \
            --featureCount 250 \
            --numThresholds 20 \
            --samplesPerImage 2000 \
            --maxDepth 15 \
            --minSampleCount 200 \
            --boxRadius 60 \
            --regionSize 3 \
            --useCIELab false \
            --cacheSizeLimit true \
            --maxAdditionalSamples 0 \
            --percentageUsageOfDispersion 50 \
            --trainAllDepths true \
            --randomBucketEachTree true
```

Testing

The Live Object Detection software is useful for visualisation, however to aggregate statistics and evaluate the performance of the trained forest on a test dataset, the `curfil_predict` program can be used. A `predict.sh` shell script similar to above can be created using Listing B.2, and removing the `#` comments. The test dataset is generated in an identical fashion to the training dataset.

Listing B.2: A shell script for evaluating the Fast forest

```
#!/bin/bash
/path/to/curfil/build/src/curfil/curfil_predict \
    15 \                                # The starting depth to begin testing
    15 \                                # The maximum depth to test up to
    "/path/to/testing/directory" \      # The directory containing annotated test data
    "/path/to/statistics/directory" \ # The directory to store the resulting statistics
    "/path/to/tree/directory/"tree-*.json.gz # The file path to the trained trees
```

When the evaluation is complete the `curfil_predict` will write out three statistical data files, with the suffix `.data`, to the folder `/path/to/statistics/directory`. These files are:

1. `performance.data`: For each depth within the range specified for testing, this file will have an entry giving the average precision, the average orientation error, and the average classification accuracy on the test set, as shown below.

```
#  Avg_Precision Avg_Accuracy Avg_Orientation_Error(Degrees)
15 0.5982        0.7543       63.34
16 0.6132        0.8123       67.12
```

2. `precision_recall.data`: This is the breakdown of the `Avg_Precision` statistic given in the `performance.data` file. For each depth within the range, this file stores the list of precision and recall values while modifying the threshold for acceptance. An example of the file format is given below.

```
#  Recall Precision
15 1.000  0.001
15 0.998  0.012
...
15 0.001  1.000

16 0.998  0.005
16 0.974  0.032
...
```

3. `confusion_matrix.data`: This gives the breakdown of the `Avg_Accuracy` statistic for each level as shown in the `performance.data` file. For each depth, a Confusion Matrix is stored, showing the performance of class predictions made by the forest on the test set.

Appendix C

User Guide: Live Object Detection

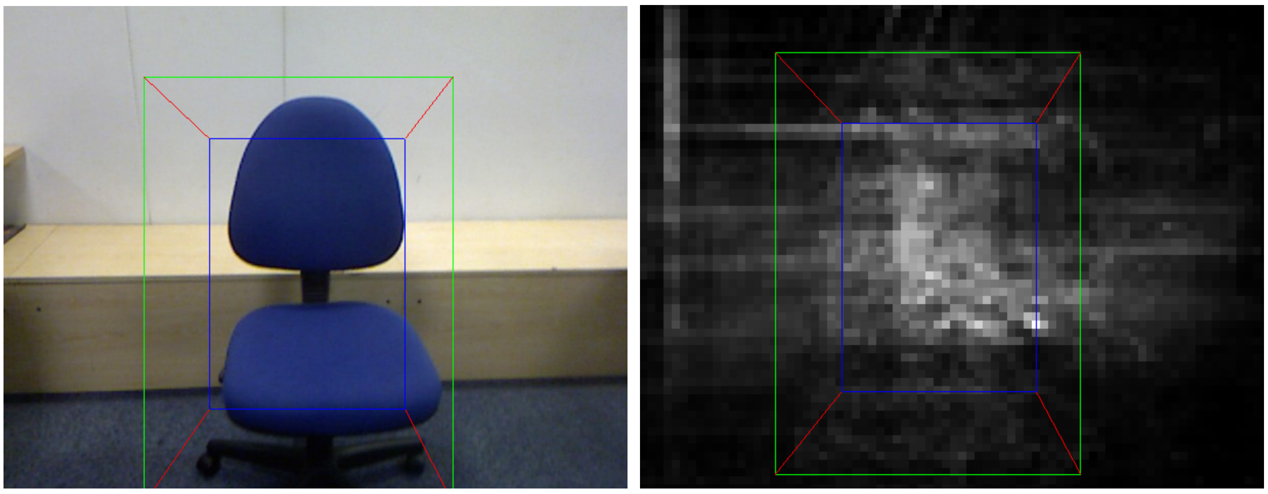
The live object detection software comes bundled with the annotation tool, as many of the features required, such as Kinect input, user controls, and OpenGL were already implemented within the KinectFusion source code. The detection system is relatively simple, with only a few options. To run the detector live from the Kinect, only the trained Hough forest files need to be specified:

```
$ ./hough_predictor -ifo "/path/to/tree/directory/"tree-*.json.gz
```

This will result in the output shown in Figure C.1a, viewed from the Kinect device. To detect objects in a pre-recorded ONI format file, the `-if` flag is again specified at the command line, and it must be placed before the `-ifo`, as below:

```
$ ./hough_predictor -if "/path/to/training/video.oni" \
                    -ifo "/path/to/tree/directory/"tree-*.json.gz
```

Two other useful options are available. The first is a live visualisation of the Hough Votes before mean clustering. This can be useful when performance is poor, and it is not known whether the mean shift clustering algorithm is the cause, or the original votes. `Space` is used to toggle this view, which is shown in Figure C.1b. The second option is a verbose readout of the predicted positions, orientations, and classes in each frame of the video. The `v` key is used for this purpose. Finally, the `q` key is used to exit.



(a) The RGB video with bounding box prediction. (b) A live view of the Hough voting space.

Figure C.1: The display from the live object detection program.

Bibliography

[1] Y. Amit, D. Geman, “Shape quantization and recognition with randomized trees” in *Neural Computation*, Vol. 9, No.7, pp.1545–1588, 1997.

[2] C. Austin, “JSON Parser Benchmarking”, URL: `http://chadaustin.me/2013/01/json-parser-benchmarking/`, last accessed 28th August 2014.

[3] I. Badami, J. Stückler, S. Behnke, “Depth-Enhanced Hough Forests for Object-Class Detection and Continous Pose Estimation” in *3rd Workshop on SPME in conjunction with ICRA*, 2013.

[4] I. Badami, *RGBD-Hough-Forest*, URL: `https://github.com/ibadami/RGBD-Hough-Forest`, last accessed 3rd June 2014.

[5] L. Breiman, “Random forests” in *Machine learning*, Vol. 45, pp. 5–32, 2001.

[6] L. Breiman, “Bagging Predictors” in *Machine learning*, Vol. 24, pp. 123–140, 1996.

[7] *Boost C++ Libraries*, URL: `http://www.boost.org/`, last accessed 28th August 2014.

[8] T. Chan, G. Golub, R. LeVeque,“Algorithms for computing the sample variance: analysis and recommendations,” in *The American Statistician*, Vol. 37, No. 3, 1983.

[9] Z. H. Cigolle, M. Mara, S. Donow, M. McGuire, D. Evangelakos, Q. Meyer, “A Survey of Efficient Representations for Independent Unit Vectors,” in *Journal of Computer Graphics Techniques*, Vol. 3, No. 2, 2014.

[10] A. Criminisi and J. Shotton, “Decision Forests for Computer Vision and Medical Image Analysis” in *Advances in Computer Vision and Pattern Recognition*, DOI 10.1007/978-1-4471-4929-3_21, ©Springer-Verlag London 2013.

[11] J. Shotton, D. Robertson, T. Sharp, “Chapter 21: Efficient Implementation of Decision Forests,” in *Decision Forests for Computer Vision and Medical Image Analysis*, DOI 10.1007/978-1-4471-4929-3_21, ©Springer-Verlag London 2013.

[12] A. Criminisi and J. Shotton, *Sherwood C++ and C# code library for decision forests*, URL: `http://research.microsoft.com/en-us/projects/decisionforests/`, last accessed 3rd June 2014.

[13] *CUDA C Programming Guide*, URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, last accessed 2nd September 2014.

[14] *CUDA Profiler User’s Guide*, URL: `http://docs.nvidia.com/cuda/profiler-users-guide/index.html`, last accessed 2nd September 2014.

[15] B. Curless and M. Levoy, “A volumetric method for building complex models from range images” in *ACM Trans. Graph.*, 1996.

[16] A. J. Davison, “Real-Time Simultaneous Localisation and Mapping with a Single Camera” in *Proceedings of the International Conference on Computer Vision*, 2003.

[17] R. O. Duda, P. E. Hart, “Use of the Hough Transformation to Detect Lines and Curves in Pictures,” in *Comm. ACM*, Vol. 15, pp. 11–15, 1972.

[18] M. Firman, D. Thomas, S. Julier, “Learning to Discover Objects in RGB-D Images Using Correlation Clustering,” URL: `www0.cs.ucl.ac.uk/staff/M.Firman/files/firman-iros-2013.pdf`, last accessed 22nd May 2014.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, "Playing Atari With Deep Reinforcement Learning" in *NIPS Deep Learning Workshop*, DeepMind Technologies 2013

[20] G. Fanelli, T. Weise, J. Gall, L. Van Gool, "Real time head pose estimation from consumer depth cameras" in *Pattern Recognition*, pp. 101–110, 2011.

[21] A. W. Fitzgibbon, A. Zisserman, "Automatic camera re-covery for closed or open image sequences" in *Proceedings European Conference on Computer Vision*, pp 311–326, June 1998.

[22] J. Gall, V. Lempitsky, "Class-Specific Hough Forests for Object Detection" in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1022-1029, 2009.

[23] J. Gall, N. Razavi, L. Van Gool, "An Introduction to Random Forests for Multi-class Object Detection" in *Outdoor Large-Scale Real-World Scene Analysis*, pp. 243-263, 2012.

[24] P. Geurts, D. Ernst, L. Wehenkel, "Extremely randomized trees" in *Machine learning 63.1*, pp. 3–42, 2006.

[25] R. Girshick, J. Shotton, P. Kohli, A. Criminisi, A. Fitzgibbon, " Efficient regression of general-activity human poses from depth images" in *International Conference Computer Vision*, 2011.

[26] B. Glocker, O. Pauly, E. Konukoglu, A. Criminisi, "Joint Classification-Regression Forests for Spatially Structured Multi-Object Segmentation" in *12th European Conference on Computer Vision*, pp. 870-881, Springer.

[27] A. Handa, T. Whelan, J. McDonald, A. J. Davison, "A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM" in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014. URL: `http://www.doc.ic.ac.uk/~ahanda/VaFRIC/index.html`

[28] M. Harris, "Optimizing Parallel Reduction in CUDA," URL: `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`, last accessed 1st September 2014.

[29] D. Holz, S. Holzer, R. B. Rusu, S. Behnke, "Real-Time Plane Segmentation using RGB-D Cameras" in *in Proc. RoboCup Symposium*, July 2011.

[30] P.V.C. Hough, "Method and means for recognizing complex patterns", *U.S. Patent 3,069,654*, 1962.

[31] S. Ivaldi, S. M. Nguyen, N. Lyubova, A. Droniou, V. Padois, D. Filliat, P. Oudeyer, O. Sigaud, "Object learning through active exploration", in *IEEE Transactions on Autonomous Mental Development*, 2013

[32] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, A. Fitzgibbon, "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera" in *UIST'11*, 16-19th October 2011,©ACM 978-1-4503-0716-1/11/10.

[33] K. Lai, L. Bo, D. Fox, "Unsupervised Feature Learning for 3D Scene Labeling" in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014. URL: `www0.cs.ucl.ac.uk/staff/M.Firman/files/firman-iros-2013.pdf`, last accessed 22nd May 2014.

[34] K. Lai, L. Bo, D. Fox, "A large-scale hierarchical multi-view RGB-D object dataset" in *Proceedings of ICRA*, 2011.

[35] B. Leibe, A. Leonardis, B. Schiele, "Robust object detection with interleaved categorization and segmentation" in *IJCV, 77(1-3)*, pp. 259–289, 2008.

[36] V. Lepetit, P. Lagger, and P. Fua. "Randomized trees for real-time keypoint recognition" in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* Vol. 2, pp. 775–781, 2005.

[37] *Lodepng*, URL: `http://lodev.org/lodepng/`, last accessed 19th August 2014.

[38] D. G. Lowe,"Object recognition from local scale-invariant features" in *Proceedings of the International Conference on Computer Vision 2*, pp. 1150–1157, 1999.

[39] R. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. Davison, P. Kohli, J. Shotton, S. Hodges, A. Fitzgibbon, "KinectFusion: Real-Time Dense Surface Mapping and Tracking" in *IEEE ISMAR*, October 2011.

[40] S. Nowozin, *Tuwo Computer Vision Library*, URL: `http://www.nowozin.net/sebastian/tuwo/`, last accessed 26th August 2014.

[41] R. Okada, "Discriminative generalized hough transform for object dectection," in *International Conference Computer Vision* 2009.

[42] *OpenSLAM*, URL: `http://www.openslam.org`, last accessed: 2nd September 2014.

[43] *RapidJSON*, URL: `https://github.com/miloyip/rapidjson/`, last accessed: 28th August 2014.

[44] *RapidJSON*, "Performance Results", URL: `https://code.google.com/p/rapidjson/wiki/Performance`, last accessed: 28th August 2014.

[45] N. Razavi, J. Gall, L. Van Gool, "Scalable multi-class object detection" in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1505–1512, 2011.

[46] M. Ristin, M. Guillaumin, J. Gall, L. Van Gool, "Incremental Learning of NCM Forests for Large-Scale Image Classification" to appear in *IEEE Conference on Computer Vision & Pattern Recognition*, June 2014. URL: `biwinas03.ee.ethz.ch/mguillau/publications/Ristin2014cvpr.pdf`, last accessed 4th June 2014.

[47] A. Saffari, C. Leistner, J. Santner, M. Godec, H. Bischof, "On-line Random Forests" in *Computer Vision Workshops (ICCV Workshops), IEEE 12th International Conference*, pp.1393-1400, 2009.

[48] R.F. Salas-Moreno, R.A. Newcombe, H. Strasdat, P.H.J. Kelly, A.J. Davison, "SLAM++: Simultaneous Localisation and Mapping at the Level of Objects" in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2013.

[49] J. Sanders, E. Kandrot, "CUDA by Example: An Introduction to General-Purpose GPU Programming," *Addison Wesley*, ISBN-13: 978-0131387683, 1st edition, 2010.

[50] T. Sharp, "Implementing Decision Trees and Forests on a GPU" in *ECCV, Part IV, LNCS 5305*, pp. 595-608. ©Springer-Verlag Berlin Heidelberg 2008.

[51] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake, "Real-time human pose recognition in parts from single depth images" in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1297–1304, 2011.

[52] J. Shotton, R. Girshick, A. Fitzgibbon, T. Sharp, M. Cook, M. Finocchio, R. Moore, P. Kohli, A. Criminisi, A. Kipman, A. Blake, "Efficient Human Pose Estimation from Single Depth Images", in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 35, No. 12, Dec. 2013.

[53] S.Schulter, C.Leistner, P.Roth, H.Bischof, L. Van Gool, "On-line hough forests" in *BMVC*, 2011.

[54] J. Sturm, N. Engelhard, F. Endres, W. Burgard, D. Cremers, "A benchmark for the evaluation of RGB-D SLAM systems", in *Proceedings of the International Conference on Intelligent Robot Systems (IROS)*, 2012.

[55] J. Stückler, N. Biresev, S. Behnke, "Semantic Mapping Using Object-Class Segmentation of RGB-D Images" in *Proceedings of the IEEE/RSJ International Conference on Robots and Systems (IROS)*, 2012.

[56] J. Stückler, S. Behnke, "Multi-Resolution Surfel Maps for Efficient Dense 3D Modeling and Tracking" in *Journal of Visual Communication and Image Representation*, 2013.

[57] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus,"Indoor segmentation and sup- port inference from RGBD images" in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2012.

[58] *TkInter*, URL: `https://wiki.python.org/moin/TkInter`, last accessed 2nd September 2014.

[59] R. Triebel, J. Shin, R. Siegwart, "Segmentation and Unsupervised Part-based Discovery of Repetitive Objects" in *Proc. of the 6th Robotics: Science and Systems Conference (RSS)*, 2013.

[60] U. Köthe, *Vigra (Vision with Generic Algorithms)*, URL: `http://ukoethe.github.io/vigra/`, last accessed 30th August 2014.

[61] B. Waldvogel, "Accelerating Random Forests on CPUs and GPUs for Object-Class Image Segmentation" *Master's Thesis*, University of Bonn, Autonomous Intelligent Systems, 2013. URL: `http://www.ais.uni-bonn.de/theses/Benedikt_Waldvogel_Master_Thesis_07_2013.pdf`, last accessed 3rd June 2014.

[62] B. Waldvogel, *CUDA Random Forest implementation for Image Labeling tasks*, URL: `https://github.com/deeplearningais/curfil`, last accessed 2nd September 2014.

[63] A. Yao, J. Gall, C. Leistner, L. Van Gool, "Interactive Object Detection" in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3242-3249, 2012.