Imperial College London
Department of Computing

KinfuSeg: a Dynamic SLAM Approach Based on Kinect Fusion

Lv Zhaoyang

September 5, 2013

Supervised by Prof. Andrew Davison

Submitted in partial fulfilment of the requirements for
the Msc Degree in Computer Science (Artificial Intelligence) of Imperial College London

To ALL PEOPLE loving Robots and Vision,
pushing AI technology
and dedicated to achieve Singularity

Abstract

Simultaneously Localization and Mapping (SLAM) is nowadays playing an important role in robot vision and localization. Existing algorithms in SLAM lay greatly on the assumption that the surrounding landmarks are static. Current SLAM research in dense representation, together with general-purpose computing in GPU, provide the chance to explore a real-time SLAM system to tackle dynamic surroundings.

In this project, based on the popular tracking and mapping system Kinect Fusion, we build our system *KinfuSeg*, which makes two main contributions. First, KinfuSeg is able to perform in a dynamic environment, which owns much less restriction on its static surrounding. Second, KinfuSeg is able to segment the moving object in a separate 3D volume. Meanwhile, KinfuSeg is able to perform in real-time in almost all the situations.

We achieve the two improvements by introducing a separate segmentation pipeline, which performs depth based segmentation first to extract motion from the surroundings, and then build 3D volumes for static scenes and motions respectively. In segmentation, we first use the point-plane model to extract all the ICP outliers as depth disparity map. Then we perform a mean filter and graph-cut method to achieve a more smooth and noise-free segmentation of motions. Finally, we perform a separate volume integration and ray casting on the static scene and dynamic scene. The models of volume integrations and ray casting are modified respectively, according to characters in static & dynamic scene.

In the results part, we demonstrate our KinfuSeg system performance. Generally, KinfuSeg is able to perform in real-time (about 15 fps). The general computation is processed as pipelines in GPU. A statistical analysis of it is therefore hard to be achieved. Instead, we generate video and image demonstrations of KinfuSeg performance in various respects. The tests include segmentation results, modelling of static & dynamic scene in different situations, and the tracking ability of SLAM system. The results show that the segmented dynamic volume is accurate when the object is moving slowly in the camera view. For two situations, either object entering or moving in the scene, KinfuSeg is able to track and segment well. When the movement of camera is slow, the tracking is capable to be accurate.

Acknowledgement

I would like to grateful acknowledge my supervisor Prof. Andrew Davison for his enthusiasm, guidance, continuous support in both inspiration and hardware through out my project. Without the high performance laptop, I would not have the chance to explore in this state-of-art robot vision research. This is also the first time that I can contribute algorithms and software to existing famous system. Prof. Davison gives me the chance to learn much knowledge in both algorithm design and programming.

Then I would like to appreciate Dr. Ankur Handa, who gave me suggestions in the project, and all my dear friends, who support me and help me with all experiments. Give my thanks to Xiao He, Pan Zilong, Sun Jiajing, Xia Lei and Wang Weikun, who took part in my final experiments.

Finally, I would like to acknowledge my dear family and girlfriend, Fang Di for their continuous encouragement and positive feedbacks. Special gratitudes I give to my parents, without whom I would not have the ability to support my study and pursue my dream in robot, computing and AI technology.

Contents

List of Figures

List of Algorithms

CHAPTER 1

Introduction

In the last several decades, scientists have been trying to build animal-like robots, from imitating their the perceptions and cognition mechanisms. Vision is an extraordinary vital sense in almost all of them. The camera, working as robots' retinal system, enables robot to perceive meaningful and consistent information from this real world with our achievement in computer vision. One of the active fields is the Structure from Motion (SFM) which simultaneously estimating both 3D geometry (structure) and camera pose (motion). A more applicable mechanism in robot system is *incrementally* SFM, known as Simultaneous Localization and Mapping (SLAM). A real-time SLAM application can help the robots to navigate and perceive in our mysterious world.

1.1 Motivation

1.1.1 Why Visual SLAM System?

Humans perform an incessant SLAM task unconsciously. While observing the environment, our brains deliver a report frame to frame. The report includes a quantitative and qualitative analysis of our own localization and the nature of surroundings. Prior SLAM systems could achieve this mechanism with a combination of sensors (laser, IMU, GPS and stereo cameras, etc.), which have already been successfully applied in domestic robots, such as an unmanned vehicle system. But there are some limitations within these multiple-sensor SLAM systems: expensive sensors (*e.g.* laser) restrict the applications in cheap domestic robots; GPS system is often denied in indoor environment; stereo vision doubles the image processing and it is also incapable to construct a remote object accurately; in a hand-held AR device, often a single camera is accessible, etc. Prior research in monocular SLAM [6][7], centring around sparse feature-based representation, has already been proved successful in a small-scale workspace. These methods held to the probabilistic propagation of states, using Extended Kalman Filter (EKF). In recent years, it was discovered that with parallel computable hardware, a new parallel framework without prior map in tracking is applicable, first proposed as Parallel Tracking and Mapping (PTAM) [15]. In PTAM, tracking and mapping are split into parallel threads on a dual-core computer. Such system makes some computational expensive techniques (bundle adjustment) applicable, and is able to produce a map

of thousand of landmarks tracked at frame-rate.

1.1.2 Why Dense SLAM System?

Both the EKF and PTAM SLAM systems are sparse-feature based, which have common limitations in accuracy and robustness in a more complicated application. To improve the tracking accuracy and robustness, state-of-the-art monocular SLAM research focuses on increasing feature numbers and more efficient methods to manage dense feature maps. Thanks to the fast development in parallel programming devices, such as Graphics Processor Unit (GPU) device, SLAM system is able to approach dense features/pixels execution in a parallel way. [23] relied on the estimation of camera pose in PTAM and used dense variational optical flow matching between selected frames. It was the first dense reconstruction system with hand-held monocular camera, built upon the commodity GPU hardware [23]. Then [25] proposed the Dense Tracking and Mapping (DTAM) system, which is able to make use of all the data in image registration. [20] presents the details about a dense visual SLAM over dense whole image alignment. The performance of such dense-feature SLAM systems outweights the sparse-feature ones in both tracking accuracy and map understanding.

The traditional dense visual SLAM frameworks are aided by a 2D monocular camera sensor. Depth information in such systems are estimated. However, the popularity of depth cameras, such as Microsoft's Kinect, makes a prior depth information accessible at measurement. Although RGB-D information is still noisy, it is possible to generate a higher-level surface geometry from the discrete 3D points (*point cloud*). [14][24] propose the KinectFusion, which can reconstruct and track a high-quality, geometrically accurate and real-time 3D model.

1.1.3 Why Dynamic SLAM System?

In the last decade, the researches of monocular SLAM system is growing more accurate and robust. Within all the SLAM approaches, they share a common fundamental assumption: the environment is static. Motions in such system are filtered out as noises. However, we are living in real world environment that is not completely static. The assumption is obviously so strong that cannot deal with our most realistic problems properly. For SLAM applications, such as in Augmented Reality (AR), much scenes are inevitable dynamic with the interaction of human

Figure 1.1: The results shown by KinectFusion [14]. B is Phong shaded reconstructed 3D model. C shows the texture mapped 3D with real-time particles simulation. D is a multi-touch interaction with the reconstructed surface. E demonstrates how it can perform a real-time segmentation and tracking. The interaction and segmentation both rely on a static environment built before motions happen.

gestures. A more robust system of monocular SLAM should begin to consider its underlying principles, and extend it to a potential dynamic environment.

The challenge in motions of a multi-sensor SLAM system is less severe than a visual SLAM system. Unmanned vehicles with SLAM systems implemented successfully in DARPA Grand Challenge can filter out motions as noises or outliers with the help other sensors, such as GPS. In their SLAM systems, multiple sensors can provide different observations for motion cues, while in visual SLAM, only vision sensor is accessible. In additions, unmanned vehicles in a wide open area observe static geographic landmarks, moving landmarks are relatively in a small-scale. However, visual SLAM is often implemented in a limited indoor environment. A moving body across the image plane may cover the most salient features in the camera, which is often forbidden in most visual SLAM framework. The success of Google car project has proven that it's essential to solve this problem for SLAM system to a gain a much wider application. In indoor small-scale environment, where GPS signal is not reliable, researches are the expected to address visual SLAM performance in dynamic and hazardous environment.

1.2 Contribution of the Thesis

In our thesis, the general contribution is that we develop the *KinfuSeg* system, based on the existing KinectFusion model, which extends the static KinectFusion SLAM model to a more dynamic scene SLAM system. Meanwhile, dynamic scene is segmented and constructed separately as accurate as possible. The tracking of camera pose in our KinfuSeg system is proven to be robust with dynamic foreground motions in our experiments. Details in KinfuSeg segmentation pipeline and software structures are all our contributions for SLAM community, which mainly include:

- Implement parallel algorithms to extract 2D depth disparity map in each frame, reducing the segmentation from global 3D space to 2D image space.
- Set up graph-cut segmentation model and implement it in parallel algorithms.
- Modify the volume integration and ray casting models in KinectFusion , which fits static volume and dynamic volumes respectively.
- The whole segmentation steps are integrated as a robust segmentation pipeline in GPU, with proper software settings for KinfuSeg system.

1.3 Thesis Overview and Layout

The rest of this thesis is divided into 7 chapters and organized as follows. Chapter 2 to 4 give a background and literature review for our system. The description of contribution starts from chapter 5.

Chapter 2 is a general discussion about background of dynamic SLAM problem, including a brief literature review for state-of-art dynamic SLAM research approaches. General-purpose computing on GPU and Kinect device are important in our approach for a dense dynamic SLAM system and they are therefore introduced.

Chapter 3 describes the basic framework of KinectFusion model. The theories and principles of each pipeline in KinectFusion are explained in detail. Meanwhile, we talk about the assumptions (eg. small motion of camera, static background) used in KinectFusion , how they are introduced and the principles behind them. A full understanding of KinectFusion model helps us to improve its performance to overcome some obstacles which prevent KinectFusion to tackle the dynamic scenes.

In chapter 4, we talk about the segmentation methods, which is important to produce a smooth and noise-free dynamic volume which our KinfuSeg system requires. Two most popular methods, level-set and graph-cut are introduced and compared.

In chapter 5, we give an overview of our KinfuSeg system. It includes an overview in KinfuSeg framework, its differences to KinectFusion and the principles that how it tackles motions and background scene. In the motion analysis, we talk about the focus of the segmentation and its current capabilities w.r.t. the conditions and difficulties, which guided our model design. The volume integration and ray casting methods are modified in modelling static scenes, which are explained with reasons and implementation details.

Chapter 6 gives a detailed explanation of the segmentation pipeline in our KinfuSeg system. This segmentation pipeline is explained step by step, including mathematical models and parallel algorithm implementations. This chapter is the key contribution of our thesis, which includes creative segmentation methods (eg. graph-cut model on depth disparity) and the modification on existing models (eg. dynamic ray casting model). A full description of the whole pipeline is listed at the last of this chapter, including initialization and clearance, which guarantees a proper and robust implementation in real-world scenes.

In chapter 7, we display the results of KinfuSeg system on our machine and analyse its performance. The experiments include a demonstration of the segmentation pipeline in steps, shaded scenes from dynamic and static scenes, etc.

Finally, chapter 8 concludes the project by discussing the strength, weakness of our KinfuSeg system and the practical future work to improve its performance.

CHAPTER 2

Background of Dynamic SLAM Problem

2.1 Related and Existing work

The reconstruction of 3D trajectory of motions from monocular camera is an ill-posed problem. The trajectory of the object and that of camera pose is coupled. The current solutions to such system pose different levels of assumptions. One of the strong assumptions require an adequate prior representation of moving object shape [17], in which the moving object geometry is known prior to tracking and mapping. A weaker assumption poses each independent moving object in the environment is a rigid 3D structure. For example, with this assumption, [27][28] proposes a one-parameter family of possible, relative trajectory of each moving object w.r.t. static background. A dynamic scenarios which doesn't pose any prior on the object is the most challenging [32]. The active research for a dynamic scene SLAM system is evolving from a sparse feature-based representation to dense pixel-based representation, from a strong prior based problem to a more general system. But state-of-art solutions are still a trade-off between the generality, real-time performance, accuracy and robustness.

2.1.1 Motion Segmentation in Sparse Dynamic SLAM

The general approach in sparse dynamic SLAM framework is in a three-step fashion. In the beginning, a visual SLAM framework is borrowed and sparse features are tracked. The second step is motion segmentation. Motion prior from optical flow and RANSAC provide clues to separate the features into foreground objects and background scene. Finally, 3D reconstruction and motion estimation are performed based on the segments. Motion segmentation is always the most challenging step in the overall performance. A simple segmentation may not fulfil its job properly and need assumptions, while a complicated segmentation is computationally expensive and hard to achieve a real-world job.

Without the real-time and monocular camera restriction, SFM with moving objects have been active for more than a decade. [12] includes a generalization of classical structure from motion related theories to methods in the challenging dynamic scenes involving multiple rigid-object mo-

tions. Given a set of feature trajectories in different independently moving objects, the dynamic SFM estimates the number of moving objects in the scene, cluster the trajectories on motions, and estimate the model as in relative camera pose and 3D structure w.r.t. to each object. [29][38] explain this method in detail. However, methods for these problems are more theoretical and mathematical than a practical solution. Only short sequences are tested, with either manually extracted or noise-free feature trajectories. A real implementation of such methods is from Ozden *et al.* [28], who proposed an sparse-feature based SFM algorithm for simultaneous tracking, segmentation and reconstruction, capable of tackling realistic sequences and achieve 3D segmentation. These methods provide inspiring theoretical methods, but are still far away from a practical approach of dynamic SLAM system.

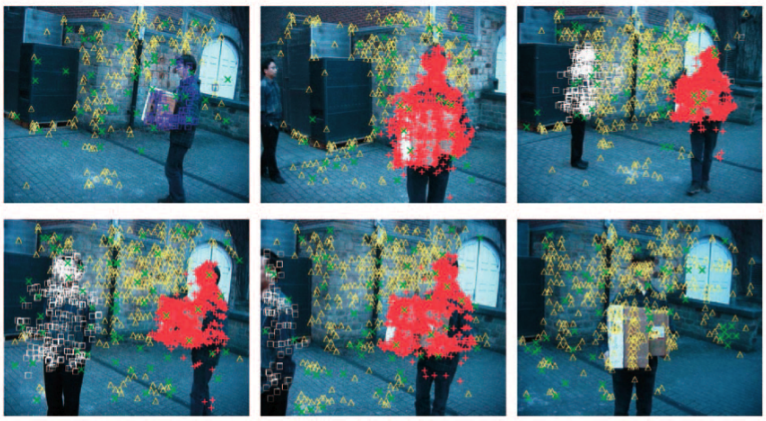


Figure 2.1: Dynamic SFM results of [28]. Even though sparse features are tracked, the runtime is still far from real-time (1 minute per frame)

SLAM with *moving object tracking* (MOT) is an active field dealing with dynamic SLAM problems, termed as SLAMMOT. Most of the solutions now are built on the sparse-feature framework and follow the same three-step fashion. In the monocular SLAMMOT approach, it is assumed that objects can be classified as stationary or moving. [40] used a 3D model based tracker running in parallel with a MonoSLAM to track a previously modelled object. Features extracted from MonoSLAM are filtered first by the object tracker. Feature evidences mutual to the object are deleted from the estimation input. This prevents the visual SLAM framework from incorporating moving features lying on the moving objects. In some practical SLAMMOT system, many researches don't perform any detection on object motions. Tracking is performed with preliminary stored features on the object. This brings other problems such as moving objects with other features will corrupt this SLAM system easily. [19] introduces a stereo SLAMMOT solution. In general, these methods of SLAMMOT have a common framework in which SLAM is built on a filter, based on which the static parts are extracted. In current SLAMMOT researches, they care more about the segmentation of static parts, rather than segmenting and tracking both motions and static scenes. Some methods, such as [19][40], still followed the EKF-based SLAM framework, which are hard to be extended to a dense SLAM system.

[1] presents a method for a live grouping of feature points into persistent 3D clusters with a single camera browsing a static scene. Although the feature grouping method is sparse clustering of static

points, this framework doesn't rely on any assumptions on object motion or shape. Agglomerative clustering is used, which approaches a graph-based segmentation method. The clustering is achieved in a real-time SLAM. It suggests the motions in dynamic SLAM system can be achieved with approaches in image segmentation.

Based on multi-body SFM framework, [16] proposed a real-time incremental multi-body visual SLAM algorithm. It integrated feature tracking, motion segmentation, visual SLAM and moving object tracking. The motion segmentation pipeline first assigns features to object and use geometric constraints, *epipolar constraint* and *Flow Vector Bound* to form a probabilistic score for each reconstructed object. In visual SLAM pipeline, objects labelled as moving are marked as invisible in background, same as the filtering in SLAMMOT. In moving object tracking, they use a set of particles to track the objects in the scene. The pose of object and its uncertainty are represented by the pose of particles and their associated weights respectively. The tracking problem of moving target's 3D position and velocity is from the notion of *bearing-only tracking* (BOT), which follows the dynamic SFM method in [27].

Figure 2.2: The motion segmentation result of [16]. The features tracked in the image sequence are still sparse. It can reach a real time performance at 14Hz.

[22] builds an improved incremental SLAM solution, adopting a graph-based framework to solve segment motions from the images. Motion potentials, defined in the graph as nodes, are composed of two parts: geometric constraints (consistent with [16]) and results from a dense optical flow. Nodes are assigned to different segmentation during a recursive estimation, in which the two nodes are merged when their differences are smaller than a threshold (set by optical flow difference). The optical flow algorithm helps them to approach a dense tracking of features and produce a dense segmentation of motions. The segmentation result is shape, especially at boundaries, in figure 2.3. The effects of optical flow in such a dense optical flow is quite obvious. From the bottom left image of figure 2.3, we can see the boundary has already been smooth in the output of optical flow. The problem with such dense optical flow is its demanding computation. [22] doesn't achieve the performance in real-time (7 minutes per frame and optical flow alone consumes 6.5 per frame, with no GPU programming).

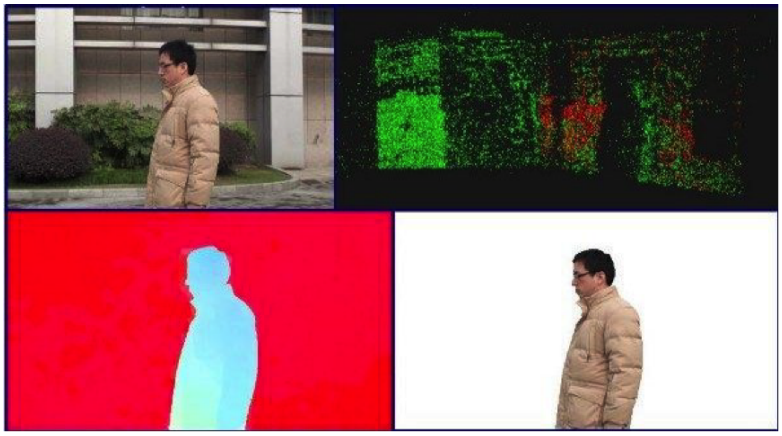


Figure 2.3: The results demonstrated by [22]. Bottom right is the extracted moving person. Bottom left is the map clues provided by optical flow, which provides smooth clues.

2.1.2 Motion Segmentation in Dense Dynamic SLAM

Although [22] is only a dense representation from features, it has already shown the merits why segmenting motion in a dense representation should be preferred. First, objects occupy only a small number of image pixels are prone to be missed or represented. Such segments are harder to be recovered in the tracking. Second, in a dense form, the region can move more freely, hard to track motions. Third, the smooth boundary extracted from the background will definitely result into a more accurate estimation in SLAM systems. The dense representation can provide a more reliable, more accurate and more robust solution than a feature-based one. Here the dense representation differs from sparse one that it should be pixel-based, rather than feature-based.

[32] proposed the first algorithm to address multi-body structure and motion problems in SLAM system, in a dense model. This work addressed a general challenge: estimate a 6-DOF transformation of each independent moving object as well as a camera with a monocular 2D camera dense video sequence. In its general frame work, there are mainly three parts to be solved: a 3D map from the 2D image sequence (including the depth map), the label of segmented objects and the motion transformation for each independent moving body. The estimation of inverse depth map follows the RGB-D visual odometry system in [37] and the tracking follows the 6-DOF camera tracking thread of DTAM [25]. Graph-cut method is used to assign the labels to pixels, during which the objects are segmented. A common framework of the three parts is that tracking and segmentation can all be solved while optimizing energy functions. [32] optimize a single energy function with hill climbing approach (*eg.* optimize label of pixels by fixing estimation of depth map and camera pose). In the calculation, GPGPU programming is involved, which enables it to estimate each frame with an average 0.7 seconds. Although not in real time, this work demonstrates well segmented results during tracking. Compared with the KinectFusion framework [14], this work is computationally demanding naturally: depth is not available and need to be estimated; not many assumptions or prior are made (only a small angle assumption), which makes optimization more expensive; no prior is used in the object tracking. However, besides demanding computation, this system is batch and reads in all the images at once after acquisition. An incremental method is suggested to be possible, but no current work has achieved it.

Figure 2.4: The segmentation result of [32]. A dense result is segmented. Before the tracking, the camera pose is unknown. In the image, the relative pose of each object is tracked w.r.t the camera pose.

[2] approached the segmentation of objects with its boundary, rather than with the dense pixels on that. In a sequence of images captured by a hand-held camera, the method segments the object shape in every image and achieves a real-time performance. Before the segmentation, it needs human interaction to 'paint' the object. Although their tracking on camera is still feature-based, the segmentation from object silhouettes can result in a dense segmentation.

2.2 Chances and Challenges with GPGPU and CUDA

The commodity GPU has emerged as an economical and fast parallel co-processor in recent years. With the popularity of GPU computing in various research fields, General-purpose Computing on Graphics Processing Units (GPGPU) is approached to solve problems by posing them as graphics rendering problems, which follows the graphics pipeline, consists of pixel processors and can be ported to the GPU. Parallel algorithms of GPU implementation can improve performance greatly in fields of linear algebra, image processing, computer vision [11][36].

Creating efficient data structures using GPU memory model is one challenge on GPGPU. Other challenges include allocating proper memory size, conflicts between parallel processors and so on. Nvidia offers an alternate programming interface called Compute Unified Device Architecture (CUDA), easing the efforts to use the parallel architecture of massive parallel processors for GPGPU. This interface is a set of library functions which can be coded as extensions to the C language. Given CUDA program, the NVCC compiler provided by Nvidia generates executable code for hosts machine and GPU devices respectively. The CUDA design has no many restrictions on memory of GPGPU. It just separates the memory into global memory, shared memory and local memory, which are friendly to use for programmers to better exploit the power of GPGPU.

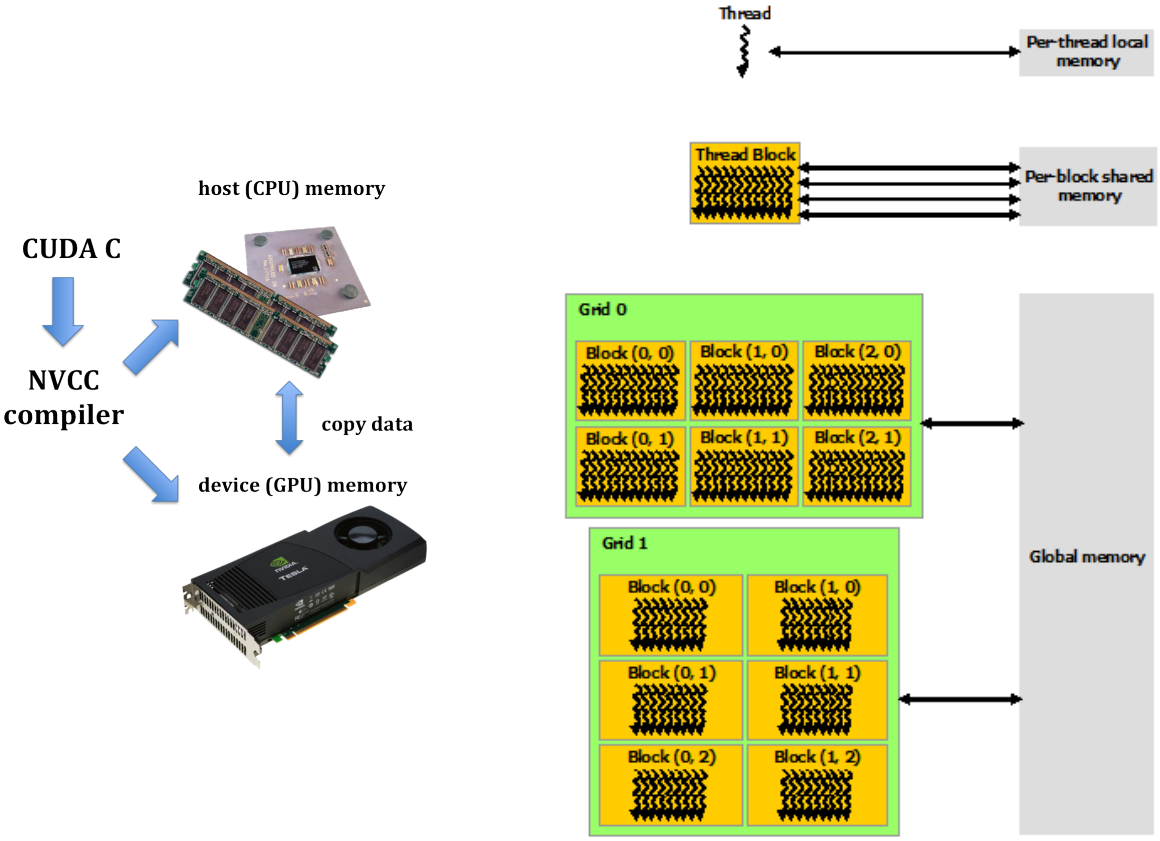


Figure 2.5: A demonstration of memory hierarchy in CUDA programming (the right one from [26]

The general CUDA model is a collection of *threads* running in parallel, which are organized in different hierarchy. A collection of threads running simultaneously is a *warp*. If the total number of threads exceeds the warp size, they are time-shared internally. The overall resources (memory, processors) are divided into *grids*, which can be further divided into *blocks*. A block is a collection of threads, in which these threads can use shared memory to accelerate access speed. Each thread executes a single instruction in parallel, called *kernel*, which is the parallel algorithm function we design.

Computation time per frame is always one of the key concern within a real-time SLAM system. With the powerful GPU processors, the tracking and mapping is now approaching a more complete, accurate and robust results in a dense SLAM system. A *Dense Tracking and Mapping* algorithm is presented in [25], which begins to exploit the matching of all the data in image registration. GPGPU not only increases the speed and accuracy of SLAM system, but it is possible to implement complicated vision or image processing techniques into SLAM system as well. For example, graph-cut method in [39] is 10 times faster thanks to GPU, which will play an vital role in our system.

The use of highly parallel GPGPU techniques is at the core of the KinectFusion design, which allows both tracking and mapping capable to perform at almost the same frame-rate of the Kinect sensor (sometimes even faster). In our system, we will also implement GPGPU to design pipelines on the base of KinectFusion , and guarantees a proper real-time performance.

2.3 The Kinect Sensor in SLAM

The Kinect sensor is a new and widely-available commodity sensor which includes a color image sensor, Infrared (IR) light source and IR image sensor for depth measurement. It can provide a UXGA 1600×1200 color image and a VGA 640×480 depth image. Since this device provides a genuine depth map within the system, SLAM system with Kinect can retrieve 3D space information without any further estimation from the traditional 2D image. It greatly improves the landmarks' positions accuracy and meanwhile saves precious computational time in every frame for a robust real-time system.

Although the quality of depth map is generally remarkable as a commodity sensor, the challenges remain vital in a vision work. Since the depth operation range is accurate only from about $0.8m$ to $3.5m$ (definitely no more than $7m$), the distance out of range is marked as "black holes" on the depth image map. Other reasons which can stop a correct measurement of IR also lead to such holes. Thin structures, smooth surfaces, transparent glass or surfaces at incidence angles can all be the causes. The problem is solved in KinectFusion by a consistent volume integration, but it will still be a very serious problem for a frame-based segmentation in our system. The device also suffers problems from a fast moving. But considering one assumption of KinectFusion is small motion, this camera weakness is not the major concern of our system.

CHAPTER 3

The Kinect Fusion Model

The real-time performance of Kinect camera tracking and surface reconstruction is based on the parallel execution on GPU device. The main system pipeline consists of four main stages: surface measurement, camera tracking, volumetric integration and ray casting. The summary of this algorithm is described below, which is also the basic algorithm of our extended dynamic SLAM model. The full model in this chapter is consistent with [14][23].

Surface Measurement

At the beginning, a *bilateral filter* is used to reduce the noise of raw depth map returned by Kinect. Then this depth values are back-projected into the sensor's frame of reference to obtain a vertex map. The cross product between the neighbouring map vertices results in a map of normal vectors. Similar to [15], a coarse-to-fine map is stored in a *3-level vertex and normal map pyramid*. With a fixed camera pose parameter at this step, the vertex and normals can be converted into global coordinates with a rigid body transformation matrix. The result of this step is a point cloud with vertices and normals.

Camera Tracking

The camera pose is tracked for each depth frame, by estimating a single 6DOF transform that closely aligns points of two consecutive frames. The alignment of frames is completed by the Iterative Closest Points (ICP) algorithm. KinectFusion uses a *projective data association* and *point-plane metric* to improve the ICP convergence speed. The basic principle is: in *projective data association*, the matching point for each image pixel is found by back-projecting it into global space and minimize the point-plane metric, given the camera pose of last frame. The estimation of pose can be reached by performing this minimization iteratively. Finally, the metric converges to produce the current estimation of camera pose at this step.

Volumetric Integration

After alignment is completed and camera pose is tracked, the raw data (without bilateral filter) is integrated at this step to reconstruct the 3D model, by using Truncated Signed Distance Function (TSDF). In each location of TSDF, it uses positive value for free space (no object), negative value for hidden space (no measurement) and zero for surface measurement.

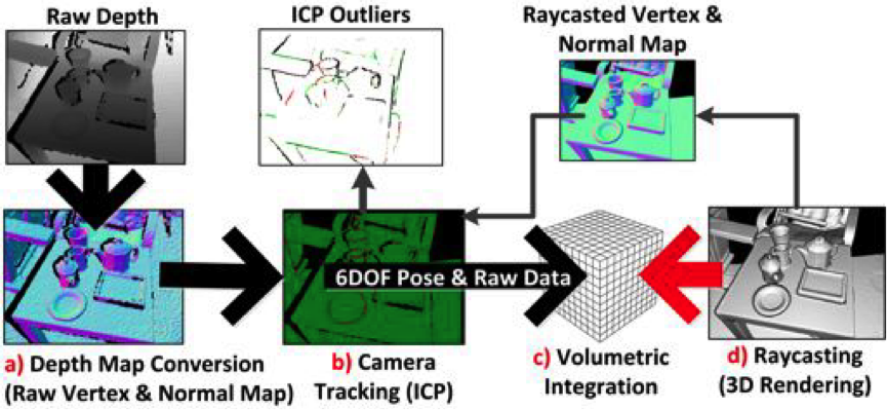


Figure 3.1: The general pipelines in KinectFusion [14]. The *Depth Map Conversion* pipeline is the *Surface Measurement* step in this thesis.

Only uncertainty of surface measurement is needed, represented by a weight factor. The global surface can be fused just by weighted averaging the TSDF's surface measurements.

Ray Casting

A prediction of the current global surface is obtained during the rendering of the global 3D model, by using ray casting. With the estimated camera pose, the camera's focal point casting rays into the surface model. The surface normals at this step is calculated from the derivative of the TSDF at the zero-crossing. The rendering result enables any 6DOF virtual camera to navigate through the 3D model. The depth and surface map generated at this step are more accurate for the next frame ICP camera tracking, than the filtered result at the step surface measurement.

3.1 Surface Measurement

At each frame k, the input data from Kinect device is an image of raw depth map $\mathbf{R}_k(\mathbf{u})$ in image domain $\mathcal{U} \subset \mathrm{R}^2$. The input raw depth data is noisy and inconsistent. First KinectFusion applies a bilateral filter to reduce noise and obtain a discontinuity preserved depth map $\mathbf{C}_k$.

$$\mathbf{C}_k(\mathbf{u}) = \frac{1}{W} \sum_{\mathbf{u'} \in \mathcal{U}} \mathcal{N}_{\sigma_1}(\|\mathbf{u} - \mathbf{u'}\|_2) \mathcal{N}_{\sigma_2}(\|\mathbf{R}_k(\mathbf{u}) - \mathbf{R}_k(\mathbf{u'})\|_2) \mathbf{R}_k(\mathbf{u'}) \tag{3.1}$$

$$\mathcal{N}_\sigma(t) = e^{-t^2 \sigma^{-2}} \tag{3.2}$$

In equation 3.1, $\frac{1}{W}$ is a normalizing constant.

Suppose $\mathbf{K}$ is the camera intrinsic matrix. The vertex map can be obtained by back-projecting $\mathbf{C}_k$ into sensor's frame, which is

$$\mathbf{V}_k(\mathbf{u}) = \mathbf{C}_k(\mathbf{u}) \mathbf{K}^{-1} \mathbf{u} \tag{3.3}$$

A fast method for corresponding normal vectors $\mathbf{N}_k$ can be calculated using neighbouring reprojected vertices.

$$\mathbf{N}_k(\mathbf{u}(x,y)) = \text{normalize}((\mathbf{V}_k(x+1,y) - \mathbf{V}_k(x,y)) \times (\mathbf{V}_k(x,y+1) - \mathbf{V}_k(x,y))) \tag{3.4}$$

To accelerate the ICP speed, we not only store a pyramid of vertex maps and normal maps at different resolution. A $L = 3$ level pyramid is chosen. First we coarse the filtered depth map $\mathbf{C}_k$ in to three levels $\mathbf{C}_k^{[1,2,3]}$. Each successive higher level of $\mathbf{C}_k^z$ is half resolution of $\mathbf{C}_k^{z-1}$, sub-sampling by averaging the block values. At each level $\mathbf{C}_k^z$, we repeat equations 3.3 and 3.4. The results are $\mathbf{V}_k^{[1,2,3]}$ and $\mathbf{N}_k^{[1,2,3]}$ at different levels.

Suppose $\mathbf{T}_k^g$ is the 6 Degree of Freedom (DoF) rigid body transform of camera pose at frame k in global coordinate, which is $\mathbf{T}_k^g = [\mathbf{R}_k^g \mid t_k^g]$. $\mathbf{R}_k^g$ is the global rotation matrix and t_k^g is the 3D translation vector. The vertex and normal map in global frame can be calculated as:

$$\begin{aligned} \mathbf{V}_k^g &= \mathbf{T}_k^g \mathbf{V}_k \\ \mathbf{N}_k^g &= \mathbf{R}_k^g \mathbf{N}_k \end{aligned} \tag{3.5}$$

3.2 Camera Tracking

The general principle of sensor pose estimation is very simple. First aligning the current frame vertices and normals with the those in global frame, a correct alignment corresponds to sensor 6 DoF transform. ICP is a well studied algorithm for this alignment and tracking.

For a real-time application, this step is often time-consuming. To guarantee a real-time performance, the KinectFusion assumes the motion between two consecutive frames is small (real-time frame rate is high). With this assumption, we can use point-to-plane metric for pose estimation, which reduces computation and can be easily parallelized by GPU computing. When point-to-plane error metric is used, the object of minimization is the sum of square distance between each source point and tangent plane at its corresponding destination point. Suppose the source point is $s_i = (s_{ix}, s_{iy}, s_{iz}, 1)$, the corresponding destination point is $d_i = (d_{ix}, d_{iy}, d_{iz}, 1)$, the normal vector at each destination d_i is n_i, the goal of each ICP iteration is to find the optimal transform $\mathbf{T}_{opt}$ for all points $i \in \mathcal{U}$ such that

$$\mathbf{T}_{opt} = \text{argmin}_{\mathbf{T}} \sum_{i \in \mathcal{U}} \|(\mathbf{T}s_i - d_i)n_i\|_2 \tag{3.6}$$

For a better association, vertices are corresponded only when a correct alignment is guaranteed ($\Omega_i \neq null$). Suppose the threshold on the distance of vertices is ε_d and threshold on difference in normal values is ε_θ suffices to reject grossly incorrect correspondences. $\mathbf{R}$ is estimated rotation and n_i' is the normal vector at source point. Thus

$$\Omega_i \neq null \quad \Longleftrightarrow \quad \begin{cases} \|\mathbf{T}s_i - d_i\|_2 < \varepsilon_d \\ \mathbf{R}n_i'(n_i)^\top < \varepsilon_\theta \end{cases} \tag{3.7}$$

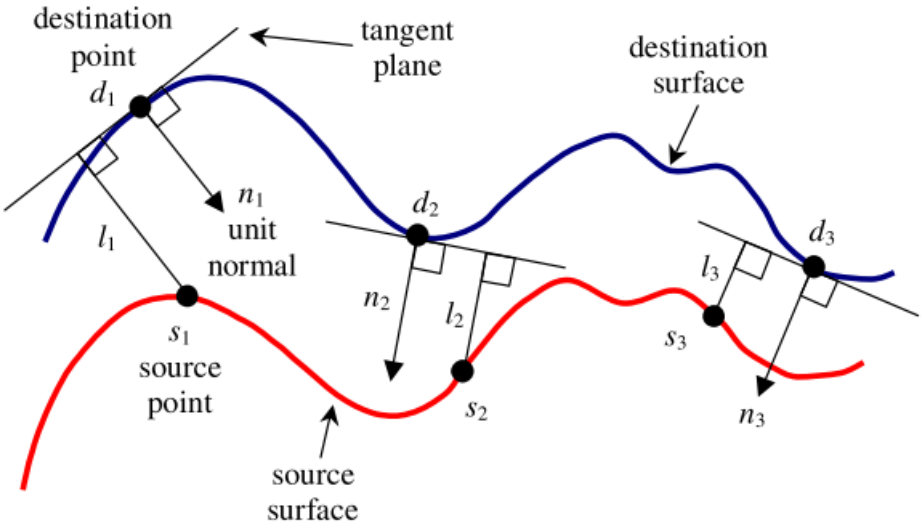


Figure 3.2: Point-to-plane error between two surfaces, in [21]

When rotation is very small (angle $\theta \approx 0$), the transform matrix $\mathbf{T}$ can be approximated as

$$\mathbf{T} = \begin{pmatrix} 1 & -\gamma & \beta & t_x \\ \gamma & 1 & -\alpha & t_y \\ -\beta & \alpha & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

The error metric term in equation 3.6 can be written as

$$(\mathbf{T}s_i - d_i)n_i = \left(\mathbf{T} \begin{pmatrix} s_{ix} \\ s_{iy} \\ s_{iz} \\ 1 \end{pmatrix} - \begin{pmatrix} d_{ix} \\ d_{iy} \\ d_{iz} \\ 1 \end{pmatrix} \right) \begin{pmatrix} n_{ix} \\ d_{iy} \\ d_{iz} \\ 0 \end{pmatrix} \tag{3.9}$$

Equation 3.8 and 3.9 can be combined as a linear expression of six prameters $\alpha, \beta, \gamma, t_x, t_y, t_z$, given all $i \subset \mathcal{U}$ pairs of correspondence (number N), the error metric $(\mathbf{T}s_i - d_i)n_i$ can be arranged into simple matrix expression as

$$\mathbf{Ax - b} \tag{3.10}$$

$$\mathbf{x} = \begin{pmatrix} \alpha & \beta & \gamma & t_x & t_y & t_z \end{pmatrix}^\top \tag{3.11}$$

$$\mathbf{A} = \begin{pmatrix} n_{1z}s_{1y} - n_{1y}s_{1z} & n_{1x}s_{1z} - n_{1z}s_{1x} & n_{1y}s_{1x} - n_{1x}s_{1y} & n_{1x} & n_{1y} & n_{1z} \\ n_{2z}s_{2y} - n_{2y}s_{2z} & n_{2x}s_{2z} - n_{2z}s_{2x} & n_{2y}s_{2x} - n_{2x}s_{2y} & n_{2x} & n_{2y} & n_{2z} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ n_{Nz}s_{Ny} - n_{Ny}s_{Nz} & n_{Nx}s_{Nz} - n_{Nz}s_{Nx} & n_{Ny}s_{Nx} - n_{Nx}s_{Ny} & n_{Nx} & n_{Ny} & n_{Nz} \end{pmatrix} \tag{3.12}$$

$$\mathbf{b} = \begin{pmatrix} n_{1x}d_{1x} + n_{1y}d_{1y} + n_{1z}d_{1z} - n_{1x}s_{1x} - n_{1y}s_{1y} - n_{1z}s_{1z} \\ n_{2x}d_{2x} + n_{2y}d_{2y} + n_{2z}d_{2z} - n_{2x}s_{2x} - n_{2y}s_{2y} - n_{2z}s_{2z} \\ \vdots \\ n_{Nx}d_{Nx} + n_{Ny}d_{Ny} + n_{Nz}d_{Nz} - n_{Nx}s_{Nx} - n_{Ny}s_{Ny} - n_{Nz}s_{Nz} \end{pmatrix} \tag{3.13}$$

Thus, when motion is small, solution of equation 3.6 approximates

$$\hat{\mathbf{x}} = \text{argmin}_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \tag{3.14}$$

Equation 3.14 is a standard linear least-square optimization problem and can be solved using Single Value Decomposition (SVD). The solution is $\mathbf{x} = \begin{pmatrix} \alpha & \beta & \gamma & t_x & t_y & t_z \end{pmatrix}^\top$.

KinectFusion uses the same point-to-plane model. Suppose the incremental transform of camera pose in global coordinate is $\mathbf{T}_{opt}$. The source point s_i is $\mathbf{T}_k^g \mathbf{V}_k(\mathbf{u})$. The destination point d_i and its normal vector are $\mathbf{V}_{k-1}^g(\mathbf{u})$ and $\mathbf{N}_{k-1}^g(\mathbf{u})$ respectively. The optimized target $\mathbf{T}_{opt}$ corresponds to the inverse of $\mathbf{T}_{opt}^{inc}$. The optimising function of KinectFusion ICP can be written as

$$\mathbf{T}_{opt}^{inc} = \text{argmin}_{\mathbf{T}^{inc}} \sum_{\mathbf{u}\in\mathcal{U}} \left\| \left(\mathbf{T}^{inc}\mathbf{T}_{k-1}^g \mathbf{V}_k(\mathbf{u}) - \mathbf{V}_{k-1}^g(\mathbf{u})\right)\mathbf{N}_{k-1}^g(\mathbf{u}) \right\|_2 \tag{3.15}$$

Since $\mathbf{T}_k^g = \mathbf{T}_{opt}^{inc}\mathbf{T}_{k-1}^g$ is estimated from $\mathbf{T}_{opt}^{inc}$, it has to be solved incrementally. It has assumed the motion is small between two frames. Suppose at the beginning of iteration, $\mathbf{T}_k^g = \mathbf{T}_{k-1}^g$. After several iteration, it will converge to a correct estimate of current transform. The iteration is performed on the pyramid we set up in section 3.1. It iterates a maximum of $z_{max} = [4, 5, 10]$ iterations in pyramid level $[3, 2, 1]$ respectively in original KinectFusion paper [23]. In our work, we reduce the iteration to be $z_{max} = [2, 3, 5]$, which has almost the same performance as the old one, but reduces about 7ms time in each frame (an important time saving for other pipelines we set up to tackle motions).

To reduce computation, equation 3.7 is slightly changed to associate the projective data only. Suppose the perspective projective point $\hat{\mathbf{u}} = \left\lfloor \pi\left(\mathbf{K}(\mathbf{T}_k^g)^{-1}\mathbf{V}_k^g(\mathbf{u})\right) \right\rfloor$, $\pi(\mathbf{u})$ is the perspective projection. Such projection will simplify ICP from global space domain into image domain, which is much easier to be parallelized and reduce computation.

3.3 Volumetric Integration

By predicting the global pose of the camera using ICP, the depth map measurement can be converted from image coordinates into single consistent global coordinate space. The KinectFusion model integrates the data by using one volumetric representation —TSDF. The TSDF volume are 3D grid of voxels stored in GPU. KinectFusion subdivides the physical space into a voxel grids with a certain number of voxels per axis (512 voxels per axis in current model). The space in the camera view is regarded as one cube. The size of the cube and the number of voxels decides the resolution of the cube. The quality of the model is also proportional to these two parameters.

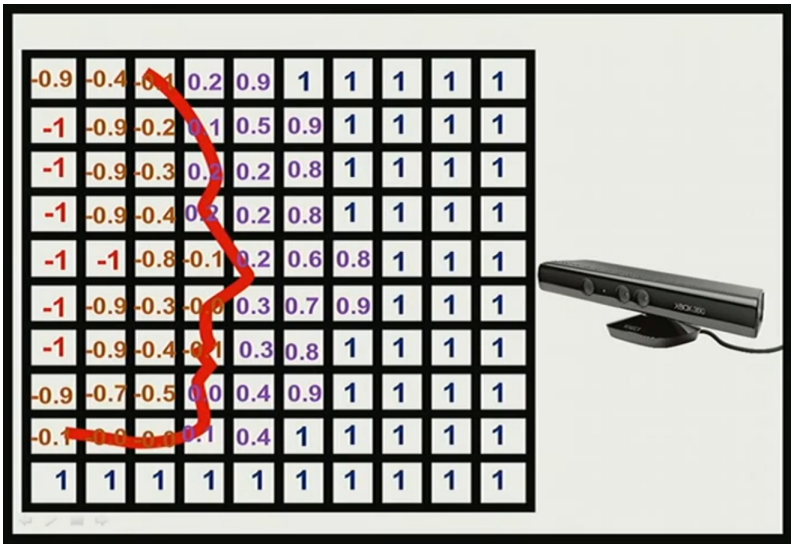


Figure 3.3: A representation of TSDF volume stored in GPU. Each grid in the image represent a voxel and its value represent the TSDF value. The *zero-crossing* place is the surface. Values increase when from far behind the surface to positive in the free-space.

Global vertices are integrated into voxels using a variant of Signed Distance Function (SDF), which specifies a relative distance to the actual point on surface. In SDF, the values are positive in front of the surface, increasing the values moving from the visible surface to free space. When values are negative, voxels are behind the surface, and decrease in values on the non-visible side. The surface interface is defined by the *zero-crossing* where the values change signs, as figure 3.3 demonstrates.

At one global frame point $\mathbf{p}$, the global TSDF $\mathbf{S}_k(\mathbf{p})$ is a fusion of the registered depths measurement from frame $1 \dots k$. There are two values stored at each voxel of TSDF: the current signed distance value $\mathbf{F}_k(\mathbf{p})$ and its weight $\mathbf{W}_k(\mathbf{p})$:

$$\mathbf{S}_k(\mathbf{p}) \mapsto [\mathbf{F}_k(\mathbf{p}), \mathbf{W}_k(\mathbf{p})] \tag{3.16}$$

The fusion in the volume is done by the weighted average of all individual TSDFs computed for each depth map. The previous fused value and weight are $\mathbf{F}_{k-1}(\mathbf{p})$ and $\mathbf{W}_{k-1}(\mathbf{p})$. At each frame, the newly projected TSDF value and weight are $\mathbf{F}_{curr}(\mathbf{p})$ and $\mathbf{W}_{curr}(\mathbf{p})$. The update can be run as

$$\mathbf{F}_k(\mathbf{p}) = \frac{\mathbf{W}_{k-1}(\mathbf{p})\mathbf{F}_{k-1}(\mathbf{p}) + \mathbf{W}_{curr}(\mathbf{p})\mathbf{F}_{curr}(\mathbf{p})}{\mathbf{W}_{k-1}(\mathbf{p}) + \mathbf{W}_{curr}(\mathbf{p})} \tag{3.17}$$

$$\mathbf{W}_k(\mathbf{p}) = \mathbf{W}_{k-1}(\mathbf{p}) + \mathbf{W}_{curr}(\mathbf{p}) \tag{3.18}$$

$\mathbf{W}_k(\mathbf{p})$ provides weighting of the TSDF proportional to the uncertainty of surface measurement. In application, [23] sets $\mathbf{W}_{curr}(\mathbf{p}) = 1$ and set a threshold $\mathbf{W}_{max}$

$$\mathbf{W}_k(\mathbf{p}) = \min(\mathbf{W}_k(\mathbf{p}), \mathbf{W}_{max}) \tag{3.19}$$

KinectFusion doesn't lunch a GPU thread for each voxel, since it's infeasible due to the large number of voxels in a 3D cube. Instead, a GPU thread is assigned to each (x, y) position on the front slice of the volume. Each GPU thread moves along each slice on the Z-axis respectively in parallel. The TSDF volume only need to represent the region of uncertainty where surface measurement exists. For the benefit of real-time performance, KinectFusion uses a projected TSDF. For the current raw depth map $\mathbf{R}_k$ with estimated know global pose $\mathbf{T}_k^g$ (the camera translation relative the origin $\mathbf{t}_k^g$), the current frame projective TSDF $\mathbf{F}_{curr}(\mathbf{p})$ is computed as

$$\mathbf{F}_{curr}(\mathbf{p}) = \Psi\Big(\lambda^{-1}\|\mathbf{t}_k^g - \mathbf{p}\|_2 - \mathbf{R}_k(\mathbf{u})\Big) \tag{3.20}$$

$$\mathbf{u} = \Big\lfloor \pi\big(\mathbf{K}(\mathbf{T}_k^g)^{-1}\mathbf{p}\big)\Big\rfloor \tag{3.21}$$

$$\lambda = \|\mathbf{K}^{-1}\mathbf{u}\|_2 \tag{3.22}$$

$$\Psi(\eta) = \begin{cases} \min\Big(1, \frac{\eta}{\mu}\Big)\text{sgn}\eta & \text{iff} \quad \eta > -\mu \\ null & \text{otherwise} \end{cases} \tag{3.23}$$

Equation 3.21 calculates the corresponding image pixel $\mathbf{u} \in \mathcal{U}$ of the global point $\mathbf{p}$. In that, the function $q = \pi(p)$ performs an perspective projection, by projecting $p = (x, y, z)$ to $q = (x/z, y/z)$. The symbol $\lfloor\cdot\rfloor$ is a nearest neighbour lookup, matching the floating result into its nearest integral pixel value. Equation 3.22 scales the measurement along each pixel ray. λ^{-1} converts the ray distance to $\mathbf{p}$ to a depth value. Equation 3.23 is a signed distance truncation of value. The truncation function Ψ ensures that a surface measurement is represented by at least one non-truncated voxel value in either side of the surface. Non-visible area (behind the surface) is not measured, and marked as *null* value.

3.4 Ray Casting

The aim of ray casting is to predict a more accurate global vertex and normal map $\mathbf{V}_k^g$ and $\mathbf{N}_k^g$ at current frame k. They will be used in the subsequent camera tracking. This is done by a per pixel ray cast.

Each pixel $\mathbf{u}$ has one corresponding ray, $t = \mathbf{T}_k^g\mathbf{K}^{-1}\mathbf{u}$. The ray t marches from the minimum depth to the *zero-crossing* and indicates the surface interface. Marching also stops when a back face is found, or exiting the working volume, both resulting a non-surface measurement at $\mathbf{u}$.

The vertex is found by block marching. KinectFusion chooses a step smaller than the TSDF threshold μ and guarantees passing through the zero crossing surface. It uses a simple approximation to the higher quality ray/trilinear intersection. Given a ray intesecting the SDF, $\mathbf{F}_t^+$ and $\mathbf{F}_{t+\Delta t}^+$ are trilinearly interpolated values along ray t and $t + \Delta t$. A more accurate approximation of ray t^* is

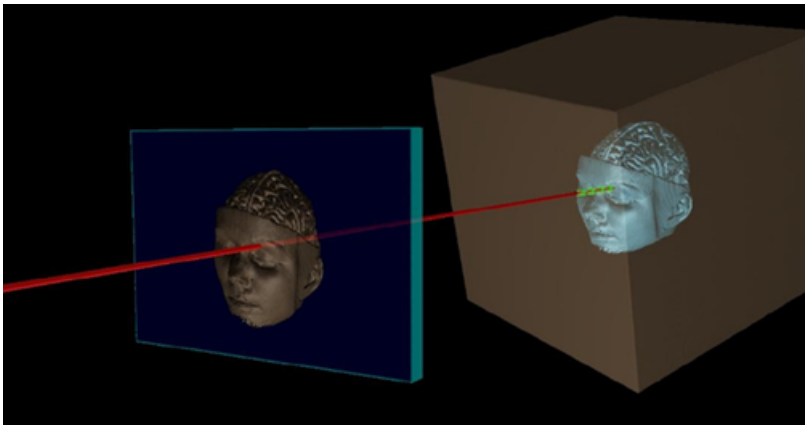


Figure 3.4: A ray cast example in global space, showing how a ray walking from a camera pixel and reach a global vertex.

$$t^* = t - \frac{\Delta t \mathbf{F}_t^+}{\mathbf{F}_{t+\Delta t}^+ - \mathbf{F}_t^+} \tag{3.24}$$

The point $\mathbf{p}$ at $\mathbf{F}_k(\mathbf{p})$ which ray t^* points to is the global vertex $\mathbf{V}_k^g$ estimated. For $\mathbf{p}$ on the surface $\mathbf{F}_k(\mathbf{p}) = 0$, it is assumed that the gradient of TSDF at $\mathbf{p}$ is orthogonal to the zero level set. Thus, the surface normal can be computed by the derivative of SDF:

$$\mathbf{N}_k^g = \text{normalize}(\nabla \mathbf{F}_k(\mathbf{p})) \tag{3.25}$$

$$\nabla \mathbf{F} = \left[\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right]^\top \tag{3.26}$$

3.5 Related Work on KinectFusion

[41] presents an extension to the KinectFusion algorithm that permits dense mesh-based mapping of extended scale environment in real-time, called *Kintinous*. The Kintinous system allows a dynamic variation of region space mapped by KinectFusion algorithm. The system achieves this by altering the KinectFusion framework such as a dynamic variation of region of space being mapped, extracting dense point cloud from the regions that leave the KinectFusion volume, and incrementally adding points to a triangular mesh representation of the environment. This system is also capable to operate in real-time.

Another advanced version of KinectFusion is the most recent SLAM++ system [35]. This system not only incorporates a pose-graph optimisation, but offers a predictive power of SLAM system as well. It takes full advantage of scenes that consists of repeated, domain specific objects. The prediction is based on a prior 3D object set up with Hough Transform. The object recognition is also real-time. The complete framework can perform SLAM in a large cluttered environment and meanwhile detecting objects well in its trip.

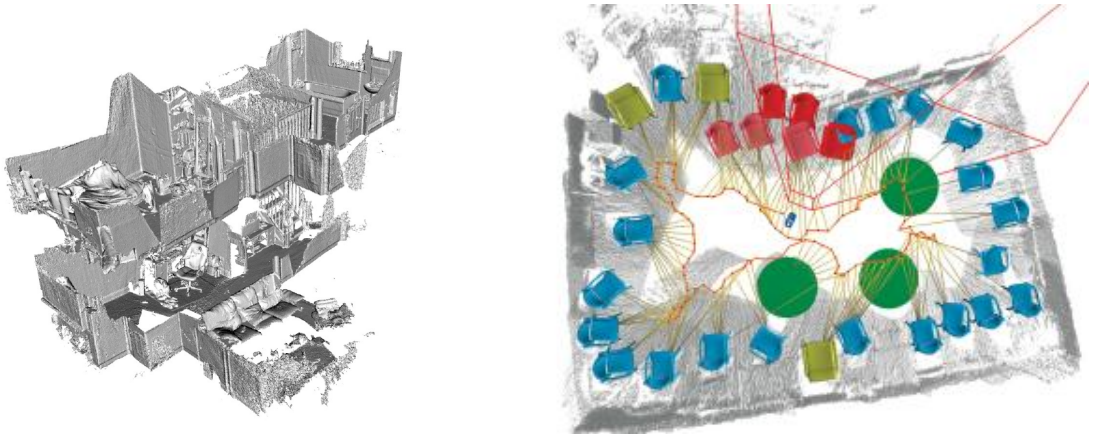


Figure 3.5: Kintinous system (left) of [41], traversing multiple rooms over two floors of an apartment and SLAM++ system (right) [35] with graph optimization and object recognition ability.

3.6 Key Assumptions in KinectFusion

The above three Kinect SLAM projects are all performed in a quite static environment. The limitations of the three frameworks are the same, which put thresholds to wider applications. The two most important assumptions are:

Static Environment

As almost all the other SLAM systems assume, during the reconstruction in a new map environment, KinectFusion still holds to the assumption that the environment should be static. Although it can tolerate a small moving object or motion in a short time, which is a great improvement, a large or longer-term scene motions will inevitably cause a tracking failure or inaccurate background surface meshing.

Small Motion between Frames

Since the algorithm can be achieved at real-time, it assumes that the changes of camera pose between frames are small. This assumption benefits the model in various parts as a reward. In the ICP algorithm, it enables a *projective data association* and point-to-plane metric possible. The energy function is linearised and solved incrementally with a *small angle* assumption in camera pose. This assumption implies two requirements. The first is the previous *Static Environment* assumption. The consecutive frames are able to be aligned correctly. The second is the *small angle* assumption, which enables incremental linearisation to predict camera pose.

3.7 Conclusion

This chapter introduces the principles of KinectFusion framework. In each pipeline of KinectFusion , we explain the detailed theory and assumptions that proposed. Some assumptions (eg. small motion) are the fundamental principle to guarantee a real-time estimation of camera pose, which will also be of vital importance in our system. But we focus on weakening others, such as totally

static scenes in the surroundings.

KinectFusion model is the foundation frame of KinfuSeg system which we will explain. All the four fundamental pipelines will be used in KinfuSeg system. Several parts of the KinfuSeg segmentation pipeline will also involve a slightly updated pipeline in KinectFusion (eg. the TSDF model).

CHAPTER 4

Methods Review for Segmentation

Section 2.1.1 and section 2.1.2 illustrate different approaches in the dynamic SLAM problem and how moving objects are segmented. In summary, there are two main approaches to such problems. First, borrow the methods of *multi-view geometry* from dynamic SFM, pixels of moving points can be detected with geometry constraints. This method is often applied to a sparse-feature representation. Sparse features that fail the constraints are excluded from the background. A more smooth and accurate result is the second: to find the correlation between pixels and segment object from the images. An object can be segmented in a batch, or extracted in every image in the incremental sequence. Such methods are similar to approaches in image/video segmentation.

Without motions clues from geometry constraints or optical flow vectors, the motion segmentation is a partitioning of an image into disjoint parts based on the image characteristics. This is an field of image segmentation. In image segmentation, there is a difference between parametric (model-driven) and non-parametric (data-driven) techniques. [1][16][22][32] all follows the non-parametric form, with no additional assumptions or prior is known.

There are several popular methods in image segmentation: snakes, graph-cut and level-set. Snake approach is an early successful active contour method and had an enormous impact in the segmentation community. But [5] points out that this method suffers from several drawbacks: sensitive to initialization, incapable to allow involving contour with topological changes, lacking probabilistic interpretation, etc. Graph-cut and level-set methods are two more successful algorithms, addressing these problems from different perspectives, which are now compared below.

4.1 Level-set Algorithm

In the variational framework, the segmentation of image plane Ω is computed by locally minimizing an appropriate energy functional, such as from its initial boundary in the direction of negative energy gradient by implementing the gradient descent algorithm:

$$\frac{\partial C}{\partial t} = -\frac{\partial E(C)}{\partial C} = F \cdot n \tag{4.1}$$

The contour can be represented either explicitly (spines or polygons) or implicitly. Level-set is an implicit representation of contour which can overcome problems in explicit ones, such as insufficient resolutions and fixed topology. In level-set, contour is defined as (zero) level line of some embedding function ϕ:

$$C = \{x \in \Omega | \phi(x) = 0\} \tag{4.2}$$

With equation 4.1 (the explicit boundary C replaced by implicit ϕ) and contour definition $n = \frac{\nabla\phi}{|\nabla\phi|}$, the level-set equation can be defined as:

$$\frac{\partial\phi}{\partial t} = -| \bigtriangledown \phi | F \tag{4.3}$$

This equation specifies the evolution of boundary ϕ and the speed function F at the location of contour. The speed function can be defined as interior versus exterior forces (liquid, viscosity, etc.). There are numerous methods to speed up the level-set[30][42]. [42] includes a complete work sheet in existing level-set algorithms. There had already been some real-time performance achieved by that. [5] gives a review of statistical approaches to level-set segmentation, which included the approaches from color, texture, and motion. Different from a SLAM problem, the condition of motion segmented there is more similar to optical flow: given camera pose and segment moving object w.r.t. their intensity change. In our dynamic SLAM problem, the speed function F can be characterised with geometry characters, such as consistency in depth, vertices, normals and even colors.

The level-set segmentation can be implemented either in global 3D space, or in image domain. In KinectFusion , one variation of level-set in global 3D space, the TSDF model, is already used in the storing and construction of surrounding scenes. The *zero-crossing* contour boundary is the corresponding contour in level-set. Equation 3.17 works in a similar way as the level-set evolution in equation 4.3. Since this method reconstructs the static whole background very successfully, it can be adopted on the moving object as well. If the moving object in the scene can be tracked well, the theory of level-set can predict it will finally evolve into its real surface. This genuine principle based on level-set in KinectFusion will bring a easier and faster segmentation, which won't affect the real-time performance too much.

To carry on such evolving in motions and static scenes separately, we still need one more step to tell foreground from the background. A cheaper method in computation is to implement level-set in image domain, rather than in global space. Such process is no different from the a pure image segmentation. More characters, such as color, intensity can be used at this step.

In image domain, level-set has its pros and cons. The advantage of level-set is that almost any energy formation can be used. But some problems may prevent level-set from being applied in image domain. First, the level-set doesn't prove to converge every time to an global optimal solution. It may even converge to only part of the object or even an empty point. A successful convergence is related to the initialisation, the characters in foreground and background. The second is the ability to cope with multiple objects. Although it is practical, solutions to such problems are slower

and more prone to failures. However, the multiple-object labelling is a non-convex optimization challenge for almost all segmentation problems. In general, level-set doesn't provide a relatively worse result compared to other methods.

4.2 Graph-cut Algorithm

Graph-cut is an optimization method for graph-based network. In many dynamic vision work, pixels/features $p \in P$ must be labelled in some set $\mathcal{L}$. In motions, such labelling can be disparity. Compared to level-set, graph-cut is more easier to optimize such disparity features. The goal is to find a labelling f that assigns each pixel $p \in P$ a label f_p where f is both piecewise smooth and consistent with observed data. Such vision problems can be naturally formulated in a graph, or in terms of energy minimization, by seeking the labelling f that optimize the energy [3]:

$$E(f) = E_{smooth}(f) + E_{data}(f) \tag{4.4}$$

In this equation, E_{smooth} measures the extent to which labelling f is not piecewise smooth and E_{data} measures the disagreement in f. E_{smooth} should be chosen carefully, or it will lead to poor results in boundaries and trapped in local minima easily. [3] suggests that one form of E_{smooth} can be ($\mathcal{N}$ is the set of pairs of adjacent pixels):

$$E_{smooth} = \sum_{\{p,q\} \in \mathcal{N}} V_{\{p,q\}}(f_p, f_q) \tag{4.5}$$

Solving such function is a NP-hard problem. The function can be solved in polinomial time if some extra limitations are imposed, such as the label $\mathcal{L}$ is restricted to 2 or a finite 1D set. [3] proposed two algorithm: α-β-*swap* and α-*expansion*. These two algorithms give a bit more flexibility to the function and guarantee a global minimum. [31] extends graph-cut to a more powerful iteration in optimization as *grab-cut*. [42] makes a summary of graph-cut related applications before year 2006. [32] in section 2.1.2 use graph-cut to segment the moving object in a SLAM system.

The advantage of graph-cut is that some solutions guarantee a global minimum, but meanwhile it is mainly subject to the computation. Problems that can be solved within polynomial time need rigid elaboration in energy function. The initialization in graph-cut is also a problem in SLAM framework. [32] uses dense optical flow as initialization for labels, but in an incremental SLAM system, an dense optical flow for each image frame is an extra burden in computation, besides the graph-cut segmentation. In KinectFusion , such initialization can from a depth disparity map give the estimated camera pose. For a better result, a good implementation should first address a good initialization and then set up models that can be solved efficiently.

4.3 Choosing between Level-set and Graph-cut

Both methods discussed above have its own segmentation merits. In combination with the original KinectFusion model, level-set can use existing structure to construct the 3D model for

static scenes or moving ones in global space, with an segmented 2D foreground image in each frame.

Both level-set and graph-cut can be implemented in image domain to separate foreground from background. Experiments prove that both algorithms can achieve good segmentation results and are possible to be achieved in a real-time application. For different situations, with various initializations, the two methods may be preferred differently. Level-set will be more useful when an explicit contour can be suggested based on the object, while graph-cut is better at pixel labelling is already given.

CHAPTER 5

Extend KinectFusion towards Modelling of Dynamic Scenes

5.1 Overview of KinfuSeg Structure

KinectFusion is able to reconstruct static scenes in an appropriate distance to the camera. Since it assumes everything it observes is static, dynamic motions will be viewed as part of the general static scenes. This inconsistency will leads to two outcomes breaking the smoothness and robustness of the model. First, while the system is performing a real-time prediction using a weighted TSDF model. When motions appear in the camera view, the foreground motion surfaces will be blended into the background scenes, which generates wrong measurement in updating vertices $\mathbf{V}_k^g$ and normals $\mathbf{N}_k^g$. Second, since the global measurement is the reference map in the next frame ICP mapping, it will lead to a wrong estimation of camera pose $\mathbf{T}_k^g$. The error in $\mathbf{T}_k^g$ will feeds back to next frame map measurements. After several iterations, the errors in $\mathbf{T}_k^g$, $\mathbf{V}_k^g$ and $\mathbf{N}_k^g$ will accumulates and lead to the failure of system.

An improved version of KinectFusion is illustrated here as the KinfuSeg system. In our system, we pose a much weaker assumption: the scene will remain static *only at the first frame reconstruction* of the background static surface. When a static surface is constructed first, any motions in the whole scenes will be separated as foreground from the background scenes. The motion will be constructed separately and will not interfere with either the background volumes or affect the tracking performance of camera.

The general system works as figure 5.1. The left part of figure 5.1 performs generally the same pipelines as KinectFusion does, except a depth disparity check after a successful tracking of camera at each frame. The right part is the segmentation pipeline, which is our innovative 5.1 plan. The result of depth disparity check will suggest whether there is motion in the system, by comparing the outlier number of ICP $\mathbf{N}_{outlier}$ to an appropriate set threshold $\varepsilon_{outlier}$. The $\varepsilon_{outlier}$ is assumed as a constant value when the system is in running. If $\mathbf{N}_{outlier} > \varepsilon_{outlier}$, the segmentation pipeline will be opened, first constructing the foreground scenes, and then returning to background scene integration. Or, a normal KinectFusion job will be performed to construct all the surrounding surfaces it assumes to be static.

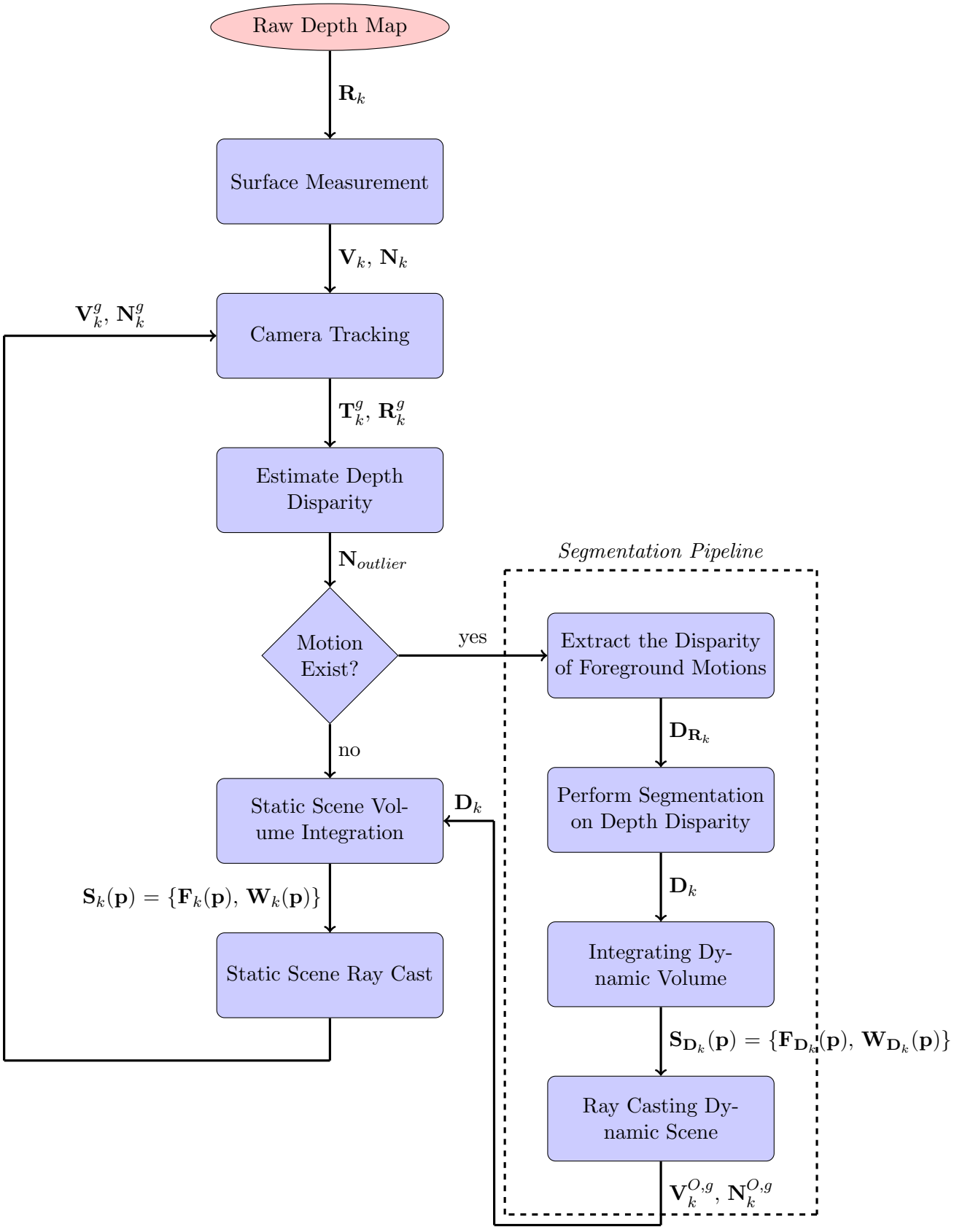


Figure 5.1: The execution process of general KinfuSeg framework in every frame. The *segmentation pipeline* is one added pipeline based on KinectFusion framework. If there is not any motion detected, it will behave like normal KinectFusion in background reconstruction.

5.2 Modelling the Motion

The aim of motion modelling is that a 3D surface of motion can be extracted from the global surface camera captured at every frame, and meanwhile it won't affect the performance of camera tracking and construction of static surface.

The focus of the motion segmentation lies on generating the moving object of current frame. Since a smooth TSDF model involves a weighted average of several frames values, the object surface will be generated from the last several frames. We don't suppose the smoothness, shapes and rigidity on the objects. However, we assume the segmented object is relatively large (eg. chair, arm, human body). Small object such as pens or mugs on the floor will be ignored. This assumption also makes sense since their movement won't cause failures in system (ICP will ignore their difference). For large moving object, such as humans' movement, they should move slowly.

In our KinfuSeg system, the segmentation of foreground doesn't track the object motion, but only construct its surface at each frame. The reason is two-fold. First, the tracking of camera pose and object is coupled. The common solution to such problem is that assuming one is static and estimate the other one's pose in each frame. When we are estimating the camera pose, we can assume the points in global background space is static (motion points are rejected). However, when we are estimating the pose of moving object, we need to suppose the camera pose is static, which is normally not in our experiment. This weak assumption will lead to a serious drift in ICP of object and lose of accuracy of TSDF. As a result, a blurred object is rendered. Second, the estimation of object pose is achieved by another local ICP of object surface. But this ICP process will be affected by the blurred object surface, generated from the the last frame. This blurring process will reject the matching of points more easily, which cause a more serious drift in return. Both of the reasons affect each other as negative feedbacks, producing a worse and worse result, and finally losing the track. In real application, the failure of tracking object is faster. Normally after one of two frame loop, the system gives up the tracking of object and fails in general.

The reconstruction of motion is completed in the segmentation pipeline in figure 5.1. The general segmentation is a four-step process. The pipeline is opened when a motion is estimated. The first two steps are mostly based on the local image. After the two steps, in each frame 2D image, the motion will be separated as foreground and the static environment as background. The depth disparity map $\mathbf{D}_k$ produced at the two steps showing their differences. With $\mathbf{D}_k$, KinfuSeg can use TSDF models to set up two separate 3D model for dynamic foreground and static background respectively. And ray casting can generate the global vertices and normals for each of them. The TSDF and ray casting of dynamic motion is completed in segmentation pipeline. After that, the disparity map $\mathbf{D}_k$ is passed back to the KinectFusion static model.

In chapter 4, we have discussed two methods in motion segmentation. In an image segmentation step, graph-cut outweights level-set in KinfuSeg . There are two reasons. First, the depth disparity map at each frame is labelled as 1 or 0 and displayed as blocks. The map is more easily initialized in energy function with graph-cut. The second reason is the about the noise. Since there might be several small blocks of noises, these block contours may mislead the contour initialization or its

evolution. In the TSDF model and ray casting model of segmentation pipeline, there are several changes in their original models. For example, the TSDF model adds in the depth disparity $\mathbf{D}_k$ and the ray casting model adds in a comparison of static vertex map $\mathbf{V}_k^g$ and object vertex map $\mathbf{V}_k^{O,g}$ against noise effects.

5.3 Modelling the Static Scenes

In general the static scene construction sticks to the original KinectFusion system. If there is no signal for motion detected, this step is the same as section 3.3 and 3.4. If motion exists, the modelling of static scenes are based on a successful segmentation of dynamic scenes. The depth disparity map from the segmentation pipeline is passed to the Volume Integration step. The general steps are almost the same as section 3.3. One difference is that the static background should not integrate the points which are labelled as dynamic. Now the input is not only a raw depth map $\mathbf{R}_k$, but also with the depth disparity $\mathbf{D}_k$. Equation 3.20 is now changed to the equation 5.1 as

$$\mathbf{F}_{curr}(\mathbf{p}) = \Psi\Big(\lambda^{-1}\|\mathbf{T}_k^g - \mathbf{p}\|_2 - \mathbf{R}_{\mathbf{D}_k}^{static}(\mathbf{u})\Big) \tag{5.1}$$

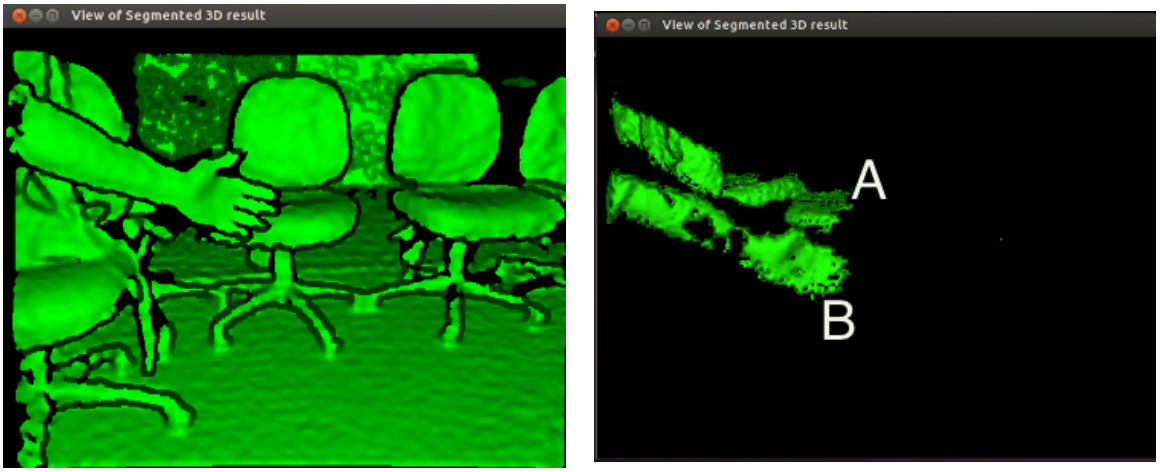
$$\mathbf{R}_{\mathbf{D}_k}^{static}(\mathbf{u}) = \begin{cases} \mathbf{R}_k(\mathbf{u}) & \mathbf{D}_k(\mathbf{u}) = 0 \\ null & \text{otherwise} \end{cases} \tag{5.2}$$

Sometimes, the disparity map $\mathbf{D}_k$ can be misleading. Figure 5.2a shows a scene the Kinect camera captures at the first several frames. In that scene, an arm is captured when system start and will move in the next several frames. According to our system assumption, the first several frames are assumed to be static. When the arm moves, $\mathbf{D}_k$ will contain two parts waiting to be constructed: the area where arm moves to (constructed as area **B** in figure 5.2b) and the area where arm moves away (area **A**). The result is that the dynamic arm which now moves away, is falsely assumed to static and still in the background scene.

To address this problem, we introduce a check on the TSDF model. Suppose foreground motions are always nearer to camera than background scene in global space. We also suppose the TSDF volume constructed in the last several frame is accurate (which proves to be normally true in our experiment). If object in background scene moves, the area once occupied by foreground motion is the *zero-crossing* in the previous TSDF model ($\mathbf{F}_{k-1}(\mathbf{p}) \approx 0$) and will now be the empty space ($\mathbf{F}_{curr}(\mathbf{p}) \approx 1$). Thus, this problem can be solved by comparing the TSDF values to the two threshold $\epsilon_{low}, \epsilon_{upper}$ on $\mathbf{F}_{k-1}(\mathbf{p})$ and $\mathbf{F}_{curr}(\mathbf{p})$. This can be concluded as:

$$\mathbf{S}_k(\mathbf{p}) \text{ is not updated } \Longleftrightarrow \begin{cases} \mathbf{F}_{k-1}(\mathbf{p}) < \epsilon_{low} \\ \mathbf{F}_{curr}(\mathbf{p}) > \epsilon_{upper} \end{cases} \text{ is false} \tag{5.3}$$

In our experiment, ϵ_{low} is chosen as 0.1 and ϵ_{upper} is 0.5. After TSDF integration, the ray casting step of background static scene is the same in both situations.



(a) The starting static scene

(b) the dynamic model extracting from the scene

Figure 5.2: The failure mode to construct object when a existing object assumed static in the background scene begins to move. In the left image, the arm is captured when the system begins and is assumed to static. The right images how the dynamic scene is when the arm moves. Area **A** is the original arm position, and area **B** is the new arm position. Both positions are shown in the disparity map and constructed falsely in the dynamic volume.

Launch **threads** $t[x][y]$ for x,y coordinate of volumes **Vol**, **in parrallel do:**

for *voxels* $\boldsymbol{g}$ *with* $z \in$ **Vol do**

- $v^g \leftarrow translate(\mathbf{g})$; `// convert g into global space, normally no rotation`
- $v \leftarrow \pi(\mathbf{K}(\mathbf{T}_k^g)^{-1}v^g)$; `// perspective projection of` v^g `into` v
- **if** $v \in \mathcal{U}$ **then**
  - $\lambda \leftarrow \|\mathbf{K}^{-1}v\|$;
  - $\mathbf{F}_{sdf} \leftarrow \lambda^{-1}\|\mathbf{t}_k^g - v^g\| - \mathbf{R}_k(v)$;
  - **if** $\boldsymbol{D}_k(\boldsymbol{v}) = true$ **then**
    - **if** $\boldsymbol{F}_{k-1} < \epsilon_{low}$ **and** $\boldsymbol{F}_{curr} > \epsilon_{upper}$ *is false* **then**
      - continue;
    - **end**
  - **end**
  - **if** $\boldsymbol{F}_{sdf} > 0$ **then**
    - $\mathbf{F}_{curr} \leftarrow \min(1, \frac{\mathbf{F}_{sdf}}{\mu})$;
  - **else**
    - $\mathbf{F}_{curr} \leftarrow \max(-1, \frac{\mathbf{F}_{sdf}}{\mu})$;
  - **end**
  - $\mathbf{W}_k(\mathbf{p}) = \min(\mathbf{W}_{k-1}(\mathbf{p}) + \mathbf{W}_{curr}(\mathbf{p}), \mathbf{W}_{max})$;
  - $\mathbf{F}_k = \frac{\mathbf{F}_{curr}\mathbf{W}_k(\mathbf{p}) + \mathbf{F}_{k-1}\mathbf{W}_{k-1}(\mathbf{p})}{\mathbf{W}_k(\mathbf{p}) + \mathbf{W}_{k-1}(\mathbf{p})}$;
- **end**

end

Algorithm 1: The parallel algorithm of static TSDF model on GPU

CHAPTER 6

Motion Segmentation in KinfuSeg

The general segmentation pipeline is described in figure 5.1. In this chapter, we will talk about the details in building the segmentation model.

6.1 Depth Disparity Generation from ICP Outliers

In each frame process, the successful ICP will generate the estimated camera pose $\mathbf{T}_k^g$. With an estimated camera pose $\mathbf{T}_k^g$ at current frame, current local vertices $\mathbf{V}_k$ and normals $\mathbf{N}_k$, the global static scene vertices $\mathbf{V}_k^g$ and normals $\mathbf{N}_k^g$, it is able to use the same projective data association method to find whether each depth point in local frame corresponds to a correct match in the global scene.

6.1.1 Extract ICP Outliers as Raw Depth Disparity

The estimation generally follows the same point-plane metric in section 3.2. The difference is that in depth disparity estimation we only need an estimation of correspondence for each local depth point, rather than a estimation of global correspondence energy minimization. Since the current camera frame $\mathbf{T}_k^g$, $\mathbf{R}_k^g$ parameters have already been estimated from ICP, there is no need to perform an iteration in checking the transform. $\mathbf{D}_{\mathbf{R}_k}$ is the raw depth disparity map generated from the correspondence check between vertices and normals comparison. ε_d and ε_θ are the same thresholds of distance and normal difference for each surface point. Similar to the rejection criterion in equation 3.7:

$$\mathbf{d}_{corresp}(\mathbf{u}) = \text{true} \quad \Longleftrightarrow \quad \begin{cases} \|\mathbf{T}_k^g \mathbf{V}_k(\mathbf{u}) - \mathbf{V}_k^g(\mathbf{u})\|_2 < \varepsilon_d \\ \mathbf{R}_k^g \mathbf{N}_k(\mathbf{u}) \mathbf{N}_k^g(\mathbf{u}) < \varepsilon_\theta \end{cases} \tag{6.1}$$

$$\mathbf{D}_{\mathbf{R}_k}(\mathbf{u}) = \begin{cases} 0 & \mathbf{d}_{corresp}(\mathbf{u}) = \text{true} \\ 1 & \mathbf{d}_{corresp}(\mathbf{u}) = \text{false} \end{cases} \tag{6.2}$$

An ideal estimation of depth disparity requires two conditions. First, the measurement of vertex map and normal map at both local and global frame ($\mathbf{V}_k$, $\mathbf{N}_k$, $\mathbf{V}_k^g$, $\mathbf{N}_k^g$) should be very accurate. Second, the estimation of camera pose $\mathbf{T}_k^g$ and $\mathbf{R}_k^g$ should be accurate too.

In our application, both of the conditions are not perfect satisfied. Although the KinectFusion model is able to construct an accurate global model $\mathbf{V}_k^g$ and $\mathbf{N}_k^g$ of surrounding scenes, the local estimation of each frame, is often inaccurate. Since the accurate measurement distance $Dist_k$ of Kinect device is often from $1m$ to $7m$, an estimation out of this range is quite inaccurate, even after bilateral filtering of depths. To reject these points, we update equation 6.2 to

$$\mathbf{D}_{\mathbf{R}_k}(\mathbf{u}) = \begin{cases} 0 & Dist_k(\mathbf{u}) \not\subset [1m, 7m] \\ 0 & \mathbf{d}_{corresp}(\mathbf{u}) = \text{true} \\ 1 & \text{otherwise} \end{cases} \tag{6.3}$$

In equation 6.3, we simply assume inaccurate measurement of $Dist_k(\mathbf{u})$ will never generate a correct depth disparity, which will be labelled as 0. This will resolve most of the problem in condition one. he measured depths at current frame is now able to select out the foreground region, which display the area of motions.

About the second condition, camera pose can often be estimated very accurately, as long as the moving object doesn't occupy the whole image frame. But even a very small error in tracking estimation will bring a serious negative effect in disparity checking. Figure 6.1 shows the raw depth disparity generated from equation 6.3 when all the scenes are still static. The image window on the right is the depth disparity generated from raw depth map (left window). The white pixel points in the black background suggests that is the area where depth disparity exist, although the area is purely static.

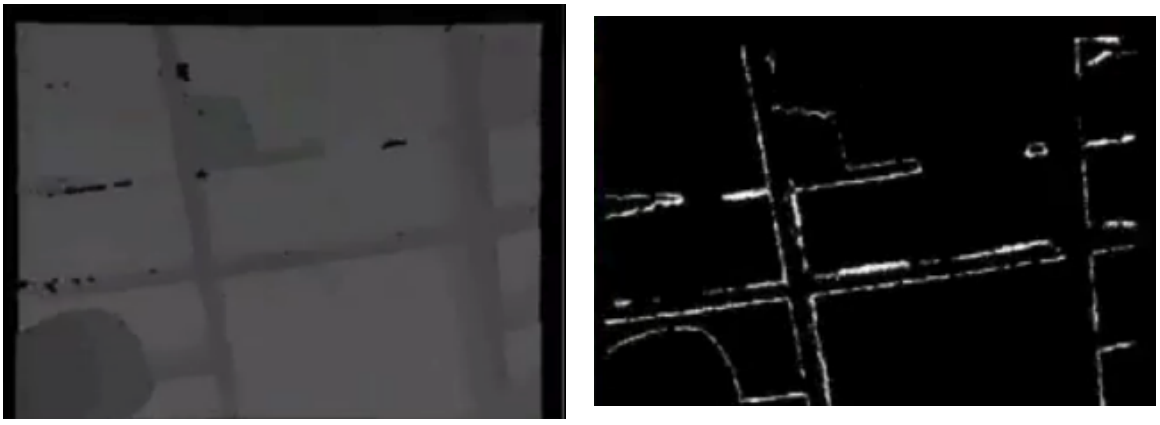


Figure 6.1: The raw depth disparity generated from the segmentation (right). The left image is the raw depth map (purely black points indicate inaccurate depth information).

6.1.2 Reduce Noises in Raw Disparity Map

This estimating errors in $\mathbf{D}_{\mathbf{R}_k}$ mostly occur at objects margin areas in the surroundings. Since there are often great difference of vertices depth in these areas, a small misalignment will break equation 6.1. For foreground region of motions, this points are noises.

One character of such background disparity errors is that they are often thin, seldom aggregating into blocks. A fast filtering method is to apply a mean filter. Define the window size to be n_w, the total number pixels in window is $N = n_w^2$. Define a threshold ε_{mean} (say 0.6) the $\mathbf{D}_{\mathbf{R}_k}(\mathbf{u})$ can be updated as

$$\mathbf{D}_{\mathbf{R}_k}(\mathbf{u}) = \Upsilon\left(\frac{1}{\mathbf{N}} \sum_{\substack{\mathbf{v}\in\mathcal{U} \\ |\mathbf{u}-\mathbf{v}|\leq\frac{n_w}{2}}}^{\mathbf{N}} \mathbf{D}_{\mathbf{R}_k}(\mathbf{v})\right) \tag{6.4}$$

$$\Upsilon(x) = \begin{cases} 1 & x > \varepsilon_{mean} \\ 0 & x \leq \varepsilon_{mean} \end{cases} \tag{6.5}$$

6.1.3 Parallel Implementation in GPU

This process is highly parallel. Between different pixel threads, there aren't much communication. The sum step in mean filter can be solved by atomic functions API in CUDA. The ICP check and mean filter are written in two kernels in GPU units. In mean filter, a faster implementation can be importing the raw disparity map into shared memory, which accelerate the frequent read process from them.

6.2 Extracting Motion Disparity with Graph-cut

The depth disparity map generated from the last frame still contains some noises, which are mostly noises of small blocks. Some object details, such as margin on the surface, are not quite smooth. In addition, for small objects, some parts of them might be rejected as difference from the background. The graph-cut method is expected to solve such problems, or reduce their effects to some extents.

The segmentation of the image is expressed as an array of *label* values $f(\mathbf{u})$ at each pixel. Generally $0 \leq f(\mathbf{u}) \leq 1$. Our problem is a hard segmentation problem $f(\mathbf{u}) \in \{0, 1\}$ (object is either foreground or background). The segmentation task is to infer the final label of each pixel from the given information.

6.2.1 Energy Function for Disparity

Section 4.2 describes the models. The raw depth disparity map $\mathbf{D}_{\mathbf{R}_k}$ is the input. The raw depth disparity map $\mathbf{D}_{\mathbf{R}_k}$ can be treated either as maps in the space domain(together with $\mathbf{R}_k$), or as color image in image domain. In space domain, either the raw depth map $\mathbf{R}_k$ or constructed maps $\mathbf{V}_k$, $\mathbf{N}_k$ can be used. In image domain, $\mathbf{D}_{\mathbf{R}_k}$ can be treated as color in RGB channels (0 as black and 1 as white). Both methods have their pros and cons. In 3D space, the depths or vertices of

Launch **threads** $t[x][y]$ for each image pixel $\mathbf{u} \in \mathcal{U}$, **in parrallel do:**
$\hat{\mathbf{u}} \leftarrow \pi(\mathbf{K}(\mathbf{T}_k^g)^{-1}\mathbf{V}_k^g(\mathbf{u}))$; `// û is the perspective projected vertex of u`
if $\hat{\boldsymbol{u}} \in \boldsymbol{V}_k$ **then**
  $v \leftarrow \mathbf{T}_k^g\mathbf{V}_k(\hat{\mathbf{u}})$;
  $n \leftarrow \mathbf{R}_k^g\mathbf{N}_k(\hat{\mathbf{u}})$;
  if $\|v - \boldsymbol{V}_k^g\| < \varepsilon_d$ **and** $|n\boldsymbol{N}_k^g| < \varepsilon_\theta$ **then**
    $\mathbf{d}_{corresp} \leftarrow$ true;
  else
    $\mathbf{d}_{corresp} \leftarrow$ false;
  end
else
  return;
end
if $Dist_k(\hat{\boldsymbol{u}}) > 1$ **and** $Dist_k(\hat{\boldsymbol{u}}) < 7$ **then**
  $\mathbf{D}_{\mathbf{R}_k}(\hat{\mathbf{u}}) \leftarrow 0$;
else if $\boldsymbol{d}_{corresp}(\hat{\boldsymbol{u}}) = true$ **then**
  $\mathbf{D}_{\mathbf{R}_k}(\hat{\mathbf{u}}) \leftarrow 0$;
else
  $\mathbf{D}_{\mathbf{R}_k}(\hat{\mathbf{u}}) \leftarrow 1$;
end
for $|\hat{\boldsymbol{v}} - \hat{\boldsymbol{u}}| = n_w$ **and** $\hat{\boldsymbol{v}} \in \mathcal{U}$ **do**
  $sum \leftarrow sum + \mathbf{D}_{\mathbf{R}_k}(\hat{\mathbf{v}})/\mathbf{N}$;
end
if $sum > \varepsilon_{mean}$ **then**
  $\mathbf{D}_{\mathbf{R}_k}(\hat{\mathbf{u}}) \leftarrow 1$;
else
  $\mathbf{D}_{\mathbf{R}_k}(\hat{\mathbf{u}}) \leftarrow 0$;
end
return

Algorithm 2: The parallel algorithm of extracting disparity from image, including checking ICP outlier and a mean noise filter

foreground and background are often one of the best one to distinguish their difference. Foreground vertices often aggregate as one block in the space. The disadvantage is that noises at the same depth level often has a small energy difference from the object block. But in image domain, this problem can be reduced to some extent. The difference in neighbouring pixels are constant 0 or 1 in values.

An energy function $\mathbf{E}$ can be expressed as below. This energy should respond to the measurement in segmentation quality, in the sense that it should be guided by both the observed foreground and background information. $\mathbf{E}$ is related to its labelling $f(\mathbf{u})$ at each pixel $\mathbf{u}$, $f(\mathbf{u}) \in \{0, 1\}$, local vertex maps $\mathbf{V}_k$ and the raw disparity map $\mathbf{D}_{\mathbf{R}_k}$. Similar to equation 4.4, $\mathbf{E}$ is a combination of data smooth term and a data term as:

$$\mathbf{E}\big(f_{\{0,1\}}, \mathbf{V}_k\big) = E_{smooth}\big(f_{\{0,1\}}, \mathbf{V}_k\big) + E_{data}\big(f_{\{0,1\}}, \mathbf{V}_k\big) \tag{6.6}$$

In which the smooth term E_{smooth} and data term E_{data} can be expressed as the sum of smooth energy $V_{\mathbf{u},\mathbf{v}}$ and data energy $D_{\mathbf{u}}$:

$$E_{smooth}\big(f_{\{0,1\}}, \mathbf{V}_k\big) = \sum_{\substack{(\mathbf{u},\mathbf{v})\in\mathcal{N} \\ (\mathbf{u},\mathbf{v})\in\mathcal{U}}} V_{\mathbf{u},\mathbf{v}}\big(\mathbf{V}_k \mid f(\mathbf{u}), f(\mathbf{v})\big) \tag{6.7}$$

$$E_{data}\big(f_{\{0,1\}}, \mathbf{V}_k\big) = \sum_{\mathbf{u}\in\mathcal{U}} D_{\mathbf{u}}\big(\mathbf{V}_k \mid f(\mathbf{u})\big) \tag{6.8}$$

In smooth term, $\mathcal{N}$ is the set of neighbouring pixels. Good results are often obtained when by including neighbours either horizontally/vertically (8-way connectivity). Both energy terms include vertex information and image information. In $D_{\mathbf{u}}$, the two terms are added as a weighted sum:

$$D_{\mathbf{u}}\big(\mathbf{V}_k \mid f(\mathbf{u})\big) = \begin{cases} \omega_1 \cdot \|\mathbf{V}_k(\mathbf{u}) - \bar{\mathbf{V}}_{back}\|_2 + \omega_2 \cdot \|f(\mathbf{u}) - 0\|_2 & f(\mathbf{u}) = 0 \\ \omega_1 \cdot \|\mathbf{V}_k(\mathbf{u}) - \bar{\mathbf{V}}_{fore}\|_2 + \omega_2 \cdot \|f(\mathbf{u}) \cdot 255 - 255\|_2 & f(\mathbf{u}) = 1 \end{cases} \tag{6.9}$$

$\bar{\mathbf{V}}_{back}$ and $\bar{\mathbf{V}}_{fore}$ are the average vertex value of background/static vertices and foreground/dynamic vertices respectively. The vertex energy is a comparison of its value with the average. For color term, $f(\mathbf{u}) = 0$ is viewed as black and $f(\mathbf{u}) = 1$ as white. At initialization process $f(\mathbf{u}) = \mathbf{D}_{\mathbf{R}_k}(\mathbf{u})$. For background pixels, foreground pixels will have larger $D_{\mathbf{u}}$ term, and the opposite for foreground pixels. Both terms can be normalized into histograms, but this process consumes time in real-time application. Here, two terms are simply added with different weights.

The smooth energy encourages the coherence in regions with similar information, which is modelled as:

$$V_{\mathbf{u}}\big(\mathbf{V}_k \mid f(\mathbf{u}), f(\mathbf{v})\big) = \begin{cases} \gamma \cdot \|\mathbf{u} - \mathbf{v}\|_2^{-1}\Big(1 - e^{-\beta \cdot f_{pair}}\Big) & f(\mathbf{u}) \neq f(\mathbf{v}) \\ 0 & \text{otherwise} \end{cases} \tag{6.10}$$

$$f_{pair} = \|\mathbf{V}_k(\mathbf{u}) - \mathbf{V}_k(\mathbf{v})\|_2 + \alpha \cdot \|f(\mathbf{u}) - f(\mathbf{v})\|_2 \tag{6.11}$$

$$\beta = \big(2\langle f_{pair}\rangle\big)^{-1} \tag{6.12}$$

The main part of the smooth model is the pairwise labelling of pixel neighbours f_{pair}. It is also a weighted sum of vertices difference and labelling difference. β is a weight constant. When $\beta = 0$, the smoothness term is simply the well-known Ising prior [31]. Here β is chosen similar to methods in [3][31], which ensures them exponential term switches appropriately between high and low contrast. The $\langle\cdot\rangle$ denotes an expectation of f_{pair}. The function $1 - e^{1-\beta x}$ ensures a smooth switch of f_{pair} value. When vertices are similar, it approaches 0 and grow bigger when difference increase. And this stops a exponential explosion when vertices are significant different, when the vertex falls on the surface margin. γ is a weight of the smooth term, which decides its relative importance w.r.t. the data term.

One important character for the model $V_{\mathbf{u}}(\mathbf{u}, \mathbf{v})$ is that it satisfies the *semi-metric* character. For any pair of labels $\mathbf{u}, \mathbf{v}$, $V_{\mathbf{u}}(\mathbf{u}, \mathbf{v})$ is *semi-metric* if it satisfies two properties: $V_{\mathbf{u}}(\mathbf{u}, \mathbf{v}) = V_{\mathbf{u}}(\mathbf{v}, \mathbf{u})$ and

$V_{\mathbf{u}}(\mathbf{u},\mathbf{v}) = 0 \iff \mathbf{u} = \mathbf{v}$. [3] proves that graph with such characters can be solved via graph-cut more efficiently, which will be explained in section 6.2.2.

Now the energy model $\mathbf{E}$ is fully defined, the segmentation can be estimated as a global minimum:

$$f(\mathbf{u})_{opt} = \text{argmin}_{f(\mathbf{u})}\mathbf{E}(f(\mathbf{u}), \mathbf{V}_k(\mathbf{u})) \tag{6.13}$$

When the minimization process finishes, the expected $f(\mathbf{u})_{opt}$ is a correct label classification of each pixel, which should be same as the final disparity map $\mathbf{D}_k(\mathbf{u})$: $\mathbf{D}_k(\mathbf{u}) = f(\mathbf{u})_{opt}$.

6.2.2 Solving Graph-cut: Push-Relabel Algorithm

The major difficulty for equation 6.13 lies in the computational costs and local minima. There have been numerous attempts to design fast algorithms for energy minimization, such as simulated annealing [9], min-cut/max-flow algorithms [8][10][3][4]. Simulated annealing is popular since it can optimize an arbitrary energy function, but it requires exponential time. In practice, annealing method is inefficient because at each step, it changes the value of only a single pixel. Since graph-cut is only one step in a KinfuSeg with high real-time requisite, such algorithm with slow iteration speed and unlikely parallalized will not be a candidate solution.

[3] proposes that energy function with *semi-metric* smooth term can be solved via graph-cut more efficiently with some proved theorems. In section 6.2.1, the term $V_{\mathbf{u}}$ is *semi-metric*. The corresponding graph can be constructed as figure 6.2. Suppose $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a directed weighted graph. It consists of a set of node $\mathcal{V}$ (rectangle in figure 6.2), connected by a set of directed edges $\mathcal{E}$ (directed lines in figure 6.2). The nodes correspond to pixels $\mathbf{u}$ in the depth disparity image $\mathbf{D}_{\mathbf{R}_k}$. There are two additional nodes called terminals (ellipses in figure 6.2). Terminals correspond to a set of labels that can be assigned to pixels. The two terminals are called *source* and *sink*, which stands for no-disparity (0) and disparity (1) respectively.

According to the types of node, there are two corresponding type of edges in the graph: **n**-link (neighbourhood link) and **t**-link (terminal link). **n**-link connects pairs of pixels, which is directed edges $\mathcal{E}$. These edges represent a neighbourhood system in the image. The cost of **n**-links stand for the penalty for discontinuity between the pixels, usually corresponding to the smooth term $V_{\mathbf{u}}$ term in equation 6.10. The **t**-links are the edges connecting pixels with terminals, which correspond to a penalty for assigning the corresponding label to the pixel. According to [3], an example of **t**-link and **n**-link weights can be set as table 6.1. $\mathbf{t}_{\mathbf{u}}^{source}$ and $\mathbf{t}_{\mathbf{u}}^{sink}$ are **t**-links connecting *source* and *sink* to nodes respectively. $\mathbf{cap}(\mathbf{u},\mathbf{v})$ is the **n**-link of node $\mathbf{u}$ pointing to $\mathbf{v}$. An opposite direction edge is $\mathbf{cap}(\mathbf{v},\mathbf{u})$.

Such graph can be solved efficiently with min-cut/max-flow algorithm. There are several algorithms are popular [4][8][10]. [8] repeatedly computes the augmenting paths from *source* to *sink* in the graph through which flow is pushed until no augmenting path can be found. This method is improved by Boykov in [4], which is a most popular method in graph-cut solution. However, this method is not easily parallelizable. The method proposed in [10] solves this problem by manipulating a *preflow* on the network, pushing flow from vertex to one of its neighbours an relabelling its

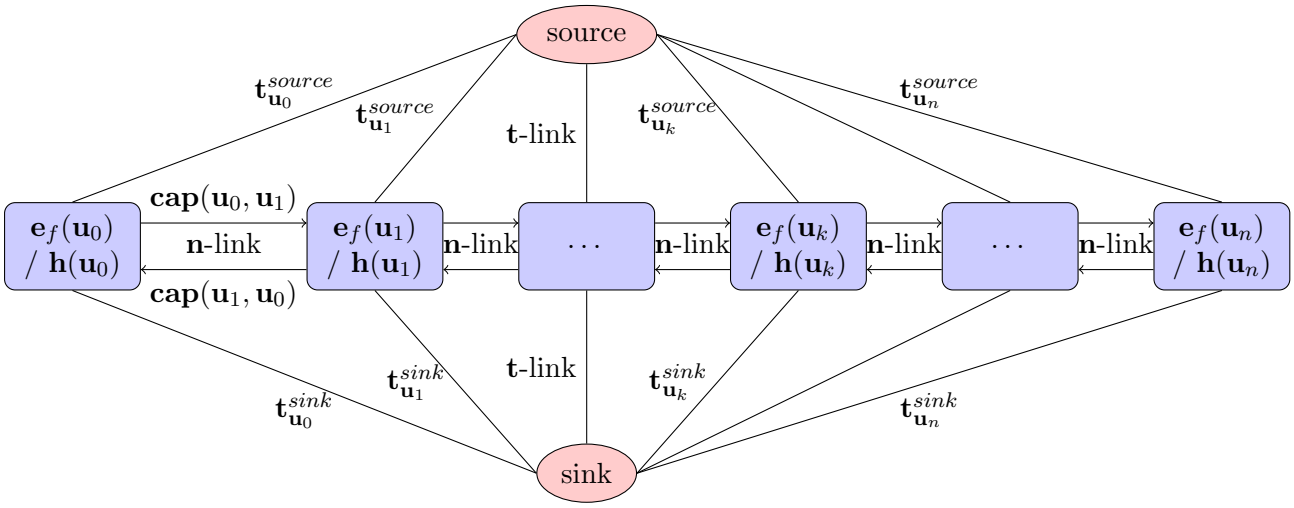


Figure 6.2: An example of the graph (simplified 1D image) set up for graph-cut algorithm. *source* and *sink* are two terminal nodes. The rectangle in the middle are pixel nodes. Lines connecting pixel nodes are **n**-links and that connecting pixel nodes with terminal nodes **t**-link.

Edge	Weight	For
$\mathbf{t}^{source}_{\mathbf{u}}$	$D_{\mathbf{u}}(\mathbf{V}_k \mid f(\mathbf{u}) = 0) + \sum_{\substack{f(\mathbf{v})\in\{1\} \\ f(\mathbf{v})\notin\{0,1\}}} V_{\mathbf{u}}(\mathbf{V}_k \mid f(\mathbf{u}) = 0, f(\mathbf{v}))$	$f(\mathbf{u}) \in \{0, 1\}$
$\mathbf{t}^{sink}_{\mathbf{u}}$	$D_{\mathbf{u}}(\mathbf{V}_k \mid f(\mathbf{u}) = 1) + \sum_{\substack{f(\mathbf{v})\in\{0\} \\ f(\mathbf{v})\notin\{0,1\}}} V_{\mathbf{u}}(\mathbf{V}_k \mid f(\mathbf{u}) = 1, f(\mathbf{v}))$	$f(\mathbf{u}) \in \{0, 1\}$
$\mathbf{cap}(\mathbf{u}, \mathbf{v})$	$V_{\mathbf{u}}(\mathbf{V}_k \mid f(\mathbf{u}), f(\mathbf{v}))$	$\{f(\mathbf{u}), f(\mathbf{v})\} \in \mathcal{N}$ $\{f(\mathbf{u}), f(\mathbf{v})\} \in \{0, 1\}$

Table 6.1: Weights allocated in **t**-link and **n**-link

distance to the sink, usually called as *Push-Relabel* algorithm. A parallel version of *Push-Relabel* called *CudaCuts* is achieved on CUDA GPU in [39]. We follow this method to solve our graph-cut problem.

In graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, the vertices $\mathcal{V} = \{\mathbf{u} \in \mathcal{U}, source, sink\}$, edges $\mathcal{E} = \{\mathbf{cap}(\mathbf{u}, \mathbf{v}), \mathbf{t}^{source}_{\mathbf{u}}, \mathbf{t}^{sink}_{\mathbf{u}}\}$, the *Push-Relabel* algorithm maintains two quantities: the excess flow $\mathbf{e}_f(\mathbf{u}')$ at each vertex and height $\mathbf{h}(\mathbf{u}')$ of each vertex, $\mathbf{u}' \in \mathcal{V}$. The excess flow $\mathbf{e}_f$ is the net income flow at node $\mathbf{u}'$. During push and relabel, the algorithm examines the vertices other than *source* and *sink* node with positive $\mathbf{e}_f(\mathbf{u}')$ and push excess from them to nodes closer to *sink* target. The goal is getting as much excess as possible to *sink* target and eventually all the nodes other than *source* and *sink* have zero excess. The conservative estimate of each node to the *sink* target is the height $\mathbf{h}(\mathbf{u})$.

Initially, at *source* point, $\mathbf{h}(source) = n, n = |\mathcal{U}|$, while at all other points, $\mathbf{h}(\mathbf{u}'), \mathbf{u}' \neq source$. The algorithm begins with a *preflow* that is equal to the edge capacity on each edge leaving the source ($\mathbf{t}^{source}_{\mathbf{u}}$).

The basic implementation of the push-relabel requires two kernels. In *Push* kernel (Algorithm 3), at each node $\mathbf{u}$, the graph first pushes the maximum flow ξ, which is the minimum of $\mathbf{e}_f(\mathbf{u})$ and

$\mathbf{cap}(\mathbf{u}, \mathbf{v})$, to its neighbour $\mathbf{v}$ and then updates the net excess flow at each node. After the push, either the node $\mathbf{u}$ is saturated ($\mathbf{e}_f(\mathbf{u}) = 0$), or the edge $(\mathbf{u}, \mathbf{v})$ is saturated ($\mathbf{cap}(\mathbf{u}, \mathbf{u}') = 0$). In the *Relabel* kernel (Algorithm 4), it applies a local relabelling process and adjusts the height $\mathbf{h}(\mathbf{u})$ at each pixel. If any potential flow is possible ($\mathbf{cap}(\mathbf{u}, \mathbf{v}) > 0$), the height of $\mathbf{u}$ is one more than the minimum height of its neighbouring nodes ($\mathbf{h}(\mathbf{u}) = \mathbf{height}(\mathbf{v}) + 1$). Both kernels are operated in parallel only on active nodes ($\mathbf{u} \in \mathcal{V} - \{south, sink\}, \mathbf{h}(\mathbf{u}) < \mathbf{h}_{max}$, and $\mathbf{e}_f(\mathbf{u}) > 0$). Two kernels are iterated until no active nodes exist. Finally, the excess flows in the nodes are then pushed back to the *source* and the saturated nodes of the final graph gives the min-cut of the the graph.

```
Launch threads t[x][y] for each node u ∈ V − {source, sink}, in parrallel do:
if active(u) = false then
    return;
end
if all (u, v) ∈ N, (u, v) ∈ V − source, sink, cap(u, v) > 0, h(u) = h(v) + 1 then
    for all (u, v) ∈ N do
        if cap(u, v) > 0 and h(v) = h(u) − 1 then
            ξ ← min(cap(u, v), e_f(u, v));
            e_f(u) ← e_f(u) − ξ;
            e_f(v) ← e_f(v) + ξ;
            cap(u, v) ← cap(u, v) − ξ;
            cap(v, u) ← cap(v, u) + ξ;
        end
    end
end
```

Algorithm 3: The parallel algorithm of *push* kernel

```
Launch threads t[x][y] for each node u ∈ V − {source, sink}, in parrallel do:
if active(u) = false then
    return;
end
if all (u, v) ∈ N, (u, v) ∈ V, cap(u, v) > 0, h(u) = h(v) − 1 then
    h_new = h_max;
    for all (u, v) ∈ N do
        if cap(u, v) > 0 then
            h_new = min(h_new, h(v) + 1);
        end
    end
    height(u') = height_new;
end
```

Algorithm 4: The parallel algorithm of *relabel* kernel

6.2.3 Parallel Implementation in GPU

Since both *push* and *relabel* processes are local methods (check only on its neighbours), they are able to be designed as two separate kernels in CUDA GPU, as shown in Algorithm 3 and

Algorithm 4. In both kernels, the height of each pixel $\mathbf{h}(\mathbf{u})$ is frequently retrieved and changed, a faster implementation can load the height map $\mathbf{h}$ into the shared memory in each block, rather than retrieve their values from global GPU memory each time.

Different from kernels before, in these two kernels, a bank conflict problem may occur when each thread changes values of capacity or of their neighbours (eg. $\mathbf{cap}(\mathbf{u}, \mathbf{v})$). The atomic functions in the most recent CUDA5.0 toolkit [26] is able to solve this problem. In our work, we didn't have enough time to dig into optimizing the parallel code. Since bank conflicts problem is serious in this application, potential hazard behaviours of threads are forbidden and atomic functions are used fully in solving these problem. But it may not be the best strategy in exploiting GPU throughput. For example, the threads are synchronized too often and block margin are disabled which caused too many idle threads. A powerful graph-cut on 2D image might contribute more if future work can improve the efficiency in algorithm and reduce its model complexity.

Although in [39], the fast graph-cut algorithm on GPU is acclaimed to be able to reduce the whole process time to around 0.1 second in 10 iterations. The time (0.1 second) still occupies much more resources than the total of other pipelines in KinfuSeg (KinfuSeg can finish each frame process in around 0.05 seconds with graph-cut). In our system, we reduce the number of iterations. The whole process will iterate in a maximum number of three. Of course, this reduces the capability of graph-cut algorithm, but generally it works well in our system. The noises left don't affect greatly. Since the estimation of depth disparity $\mathbf{D}_{\mathbf{R}_k}$ delivers a good stating point, the convergence speed is faster and the result is less likely to get trapped into local minima. After this step, the output $\mathbf{D}_k$ can provide a relatively good estimation for the dynamic TSDF volume and its surface estimation.

6.3 Dynamic Volume Construction

The output of graph-cut process is a new disparity map $\mathbf{D}_k$, which labels out all the pixels indicating motions. Similar to the modelling of static scenes, the input depth map of TSDF is filtered by $\mathbf{D}_k$. An updated TSDF equation is now

$$\mathbf{F}_{curr}(\mathbf{p}) = \Psi\Big(\lambda^{-1}\|\mathbf{T}_k^g - \mathbf{p}\|_2 - \mathbf{R}_{\mathbf{D}_k}^{dynamic}(\mathbf{u})\Big) \tag{6.14}$$

$$\mathbf{R}_{\mathbf{D}_k}^{dynamic}(\mathbf{u}) = \begin{cases} \mathbf{R}_k(\mathbf{u}) & \mathbf{D}_k(\mathbf{u}) = 1 \\ null & \text{otherwise} \end{cases} \tag{6.15}$$

The λ and Ψ is the same as equation 3.22 and 3.23.

In section 5.2, we describe the aim of a dynamic TSDF volume is to preserve the most current actions. The expected dynamic TSDF volume is able to erase the past occupied voxels and fill into new values. Similar to a dynamic occupancy map. We hope the probability of voxel reduces to 0 when the dynamic voxel is no longer occupied. Since the probability of each voxel is controlled by the their weight $\mathbf{W}_k(\mathbf{p})$. By adding one extra term $\mathbf{W}_m$, which is chosen to be half of $\mathbf{W}_{max}$ in our system. Thus the equation 3.18 and 3.19 are updated as

$$\mathbf{W}_k(\mathbf{p}) = \min(\mathbf{W}_{k-1}(\mathbf{p}) + \mathbf{W}_{curr}(\mathbf{p}) + \mathbf{W}_m, \mathbf{W}_{max}) \tag{6.16}$$

Launch **threads** $t[x][y]$ for x,y coordinate of volumes **Vol**, **in parrallel do:**
for *voxels* $\boldsymbol{g}$ *with* $z \in$ **Vol do**
  $v^g \leftarrow translate(\mathbf{g})$; `// convert g into global space, normally no rotation`
  $v \leftarrow \pi(\mathbf{K}(\mathbf{T}_k^g)^{-1}v^g)$; `// perspective projection of` v^g `into` v
  if $v \in \mathcal{U}$ **and** $\boldsymbol{D}_k(v) = 1$ **then**
    $\lambda \leftarrow \|\mathbf{K}^{-1}v\|$;
    $\mathbf{F}_{sdf} \leftarrow \lambda^{-1}\|\mathbf{t}_k^g - v^g\| - \mathbf{R}_k(v)$;
    if $\boldsymbol{F}_{sdf} > 0$ **then**
      $\mathbf{F}_{curr} \leftarrow \min(1, \frac{\mathbf{F}_{sdf}}{\mu})$;
    else
      $\mathbf{F}_{curr} \leftarrow \max(-1, \frac{\mathbf{F}_{sdf}}{\mu})$;
    end
    $\mathbf{W}_k(\mathbf{p}) = \min(\mathbf{W}_{k-1}(\mathbf{p}) + \mathbf{W}_{curr}(\mathbf{p}) + \mathbf{W}_m, \mathbf{W}_{max})$;
    $\mathbf{F}_k = \frac{\mathbf{F}_k\mathbf{W}_k(\mathbf{p}) + \mathbf{F}_{k-1}\mathbf{W}_{k-1}(\mathbf{p})}{\mathbf{W}_k(\mathbf{p}) + \mathbf{W}_{k-1}(\mathbf{p})}$;
  end
end

Algorithm 5: The parallel algorithm of dynamic TSDF model on GPU

The other part of the algorithm is the same as the TSDF model in KinectFusion system. For a full description, the details of dynamic TSDF model is shown as Algorithm 5.

6.4 Ray Casting for Dynamic Scenes

The aim of ray casting for dynamic scenes is to predict the global vertex and normal map $\mathbf{V}_k^{O,g}$ and $\mathbf{N}_k^{O,g}$ of the moving object at current frame k. The general steps pixel ray cast of dynamic volume is the same as the static ones, which also shoots one ray from each pixel $t = \mathbf{T}_k^g\mathbf{K}^{-1}\mathbf{u}$ and marches from minimum depth to *zero-crossing* and use equation 3.24 to find the exact ray and use equation 3.25 and 3.26 to estimate normal map.

To further reduce the noise effect, in our KinfuSeg , we shoot a ray from the camera pixel into both the static volume $\mathbf{S}_k(\mathbf{p})$ and dynamic volume $\mathbf{S}_{\mathbf{D}_k}(\mathbf{p})$. Both of the two volume has the same translation and no rotation w.r.t. the origin. Thus the values of the ray t_D walks across can be compared directly. We assume the moving object always appear in the foreground scenes and static scenes always lie in the background. When t_D walks to the *zero-crossing* point of $\mathbf{S}_{\mathbf{D}_k}(\mathbf{p})$, where the TSDF value $\mathbf{F}_{\mathbf{D}_k}(\mathbf{p}) = 0$, the corresponding TSDF value $\mathbf{F}_k(\mathbf{p})$ should be much greater than 0, or at least greater than $0 + \epsilon$. Suppose the $F^+_{\mathbf{D}_k,t}$ and $F^+_{\mathbf{D}_k,t+\Delta t}$ are the intersection points of ray t_D with dynamic SDF model, while F_t^+ and $F_{t+\Delta t}^+$ are the ones with static SDF model. The ray t_D is valid if:

$$t_D = \mathbf{T}_k^g \mathbf{K}^{-1}\mathbf{u} - \frac{\Delta t \mathbf{F}^+_{\mathbf{D}_k,t}}{\mathbf{F}^+_{\mathbf{D}_k,t+\Delta t} - \mathbf{F}^+_{\mathbf{D}_k,t}} \tag{6.17}$$

$$\mathbf{F}_k(t_D) > \epsilon \tag{6.18}$$

If equation 6.18 can be satisfied, the ray is valid and will generate the vertex on the surface. Otherwise the t_D is discarded, and all the voxels $\mathbf{F}_{\mathbf{D}_k}(\mathbf{p})$ which t_D walks through are reset as 1. This means these voxels reset are all empty area, not occupied by any objects. In our system, the threshold ϵ is set as 0.5, rejecting points in foreground too close to the background surfaces.

Launch **threads** $t[x][y]$ for each pixel $\mathbf{u} \in \mathcal{U}$, **in parallel do:**
$\mathbf{ray}^{start} \leftarrow \mathbf{T}_k^g \mathbf{K}^{-1}[\mathbf{u}, 0]$;
$\mathbf{ray}^{position} \leftarrow \mathbf{T}_k^g \mathbf{K}^{-1}[\mathbf{u}, 1]$;
$\mathbf{ray}^{dir} \leftarrow$ normalize$(\mathbf{ray}^{position} - \mathbf{ray}^{start})$;
$\mathbf{step} \leftarrow 0$;
$\mathbf{g}^{dynamic} \leftarrow$ first voxel on $\mathbf{ray}^{start}$; `// The first voxel in dynamic volume VolD`
$\mathbf{g}^{static} \leftarrow$ first voxel on $\mathbf{ray}^{start}$; `// The first voxel in static volume VolS`
for *all* $\boldsymbol{g}^{dynamic} \in$ ***VolD*** **and** $\boldsymbol{g}^{static} \in$ ***VolS*** **do**
  $\mathbf{step} \leftarrow \mathbf{step} + 1$;
  $\mathbf{g}^{dynamic}_{\mathbf{step}-1} \leftarrow \mathbf{g}^{dynamic}$;
  $\mathbf{g}^{dynamic} \leftarrow \mathbf{g}^{dynamic} + \mathbf{ray}^{dir}$;
  if zero-crossing *between* $\boldsymbol{g}^{dynamic}_{\boldsymbol{step}-1}$ *and* $\boldsymbol{g}^{dynamic}$ **then**
    $\mathbf{ray}_D =$ trilinear_interpolate$(\mathbf{ray}^{dynamic}, \mathbf{step})$;
    $\mathbf{g}^{static} \leftarrow \mathbf{g}^{static} + \mathbf{step} \cdot \mathbf{ray}^{dir}$;
    if $\boldsymbol{F}_k(\boldsymbol{ray}_D) > \epsilon$ **then**
      $\mathbf{V}_k^{O,g} \leftarrow \mathbf{ray}_D$;
      $\mathbf{N}_k^{O,g} \leftarrow$ normalize$(\nabla \mathbf{F}_{\mathbf{D}_k,k}(\mathbf{ray}_D))$;
    else
      for *all voxels* $\boldsymbol{g}'$ *on* $\boldsymbol{g}^{static} + \boldsymbol{ray}^{dir} \cdot \lambda$ **do**
        $\mathbf{F}_k(\mathbf{g}') \leftarrow 1$;
      end
      break;
    end
  end
end

Algorithm 6: The parallel algorithm of dynamic ray cast on GPU

6.5 Segmentation Initialization and Clearance

The full segmentation pipeline of KinfuSeg is demonstrated in figure 6.3. Since the moving object doesn't constantly appear in our scenes, containers related to segmentation should be initialized at a proper time and updated when dynamic scenes exist and change, and finally clear all the existing dynamic TSDF volumes when the moving scenes disappear.

Figure 6.3 shows the initialization and clearance procedures. At initialization step, a new dynamic TSDF volume is created. Meanwhile, containers used in graph-cut are initialized. At clearance step, the existing dynamic TSDF volume is destroyed. For a proper visualization, the $\mathbf{V}_k^{O,g}$ and $\mathbf{N}_k^{O,g}$ copies the values from $\mathbf{V}_k^g$ and $\mathbf{N}_k^g$. Thus, when there is no dynamic scene, the dynamic scene window just shows the same result as the static scene one.

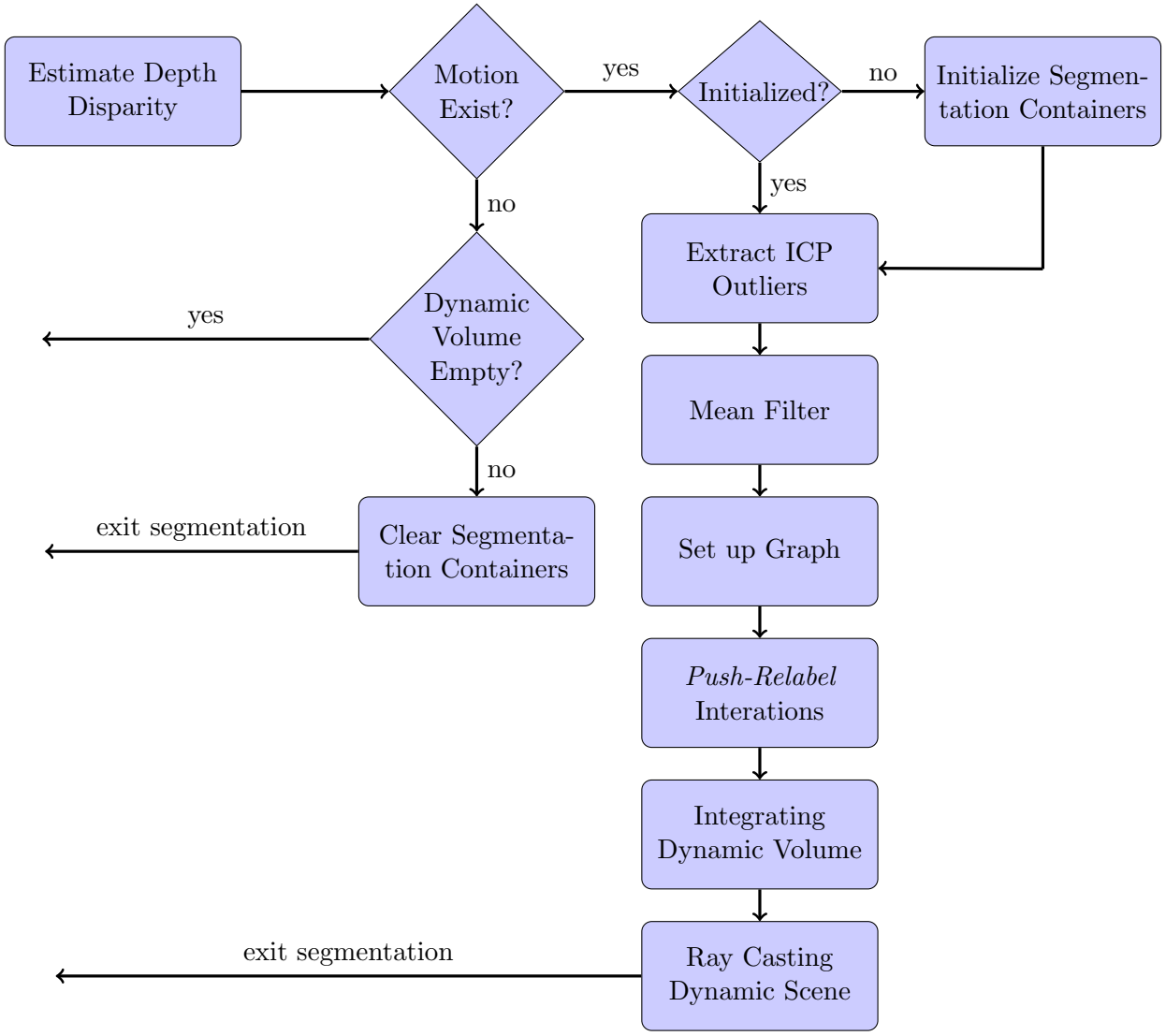


Figure 6.3: A complete description of KinfuSeg segmentation pipeline

6.6 Conclusion

In this chapter, we describe the details of every step in KinfuSeg segmentation pipeline. When segmentation pipeline is opened, initially, with the camera pose estimated at current frame, we perform a ICP matching of depth information and extract all the outliers. These outliers are filtered first and saved as a raw depth disparity $\mathbf{D}_{\mathbf{R}_k}$. Some noises are still preserved at this step. Then we use a graph-cut algorithm to extract the foreground disparity area and produce the new disparity map $\mathbf{D}_k$. After that, the $\mathbf{D}_k$ along with the raw depth map $\mathbf{R}_k$ comes into the dynamic TSDF model and set up the dynamic TSDF volume. Finally, a pixel ray cast method walks through

both the dynamic TSDF volume and static TSDF volume, getting $\mathbf{V}_k^{O,g}$ and $\mathbf{N}_k^{O,g}$. Meanwhile, the dynamic TSDF volume is filtered. The segmentation pipeline ends at this step and returns to a construction of static volume in main static pipelines in KinfuSeg . And then waits for the next call.

A proper initialization and clearance mechanism ensures the robustness of the KinfuSeg system. In the next chapter, we will demonstrate the performance of KinfuSeg in various situations and analyse its strength and weakness w.r.t. to the principles of the system.

CHAPTER 7

Results and Analysis

7.1 Project Development Tools

System and hardware

For all programs and tests we run our KinfuSeg system on a standard laptop PC on a 64-bit Ubuntu 12.04 system. The laptop is equipped with an Intel(R) Core(TM) i7 4700MQ 2.40GHz CPU, 16GB DDR 1600MHz RAM and a nVidia GeForce GTX 780M GPU card. The GPU owns 4GB global memory and a total of 1536 CUDA Cores (8 multiprocessor $\times$ 192 CUDA cores/MP). The sensor we use is a single standard RGB-D Kinect camera, with 640×480 resolution and frame-rate up to 30Hz.

Software

The software implementation is based on the open-source library Point-Cloud Library (PCL) [34][33]. The PCL 1.70 version incorporates a KinectFusion module[18], which contains all the KinectFusion pipelines. This module is based on the KinectFusion [23], and is improved (in several pipeline details) with Kintionous work [41]. It is able to construct scenes in a $3 \times 3 \times 3m^3$ TSDF cube and perform tracking at a frame-rate about 20Hz. Texture mapping is disabled because of unsolved program bugs in RGB display. In parallel GPU programming, we use the CUDA5.0 toolkit [26], which has built-in atomic functions.

7.2 General Real-time Performance of KinfuSeg System

7.2.1 System Speed

All the processes are completed on a single laptop in real-time, including the data capturing, system operation and visualization. The data sets we use are captured by Kinect camera at every frame. The system performance is not affected by the types of scenes. But since a basic full constructed background model is the pre-requisite for the dynamic model, we often choose small rooms with no smooth reflection surfaces and transparent glasses (infrared from Kinect device can't measure depth of such conditions). A comparison of computational performance between

KinectFusion and KinfuSeg on our machine is presented in table 7.1.

	KinectFusion	**KinfuSeg**
minimum time (ms/frame)	30	48
maximum time (ms/frame)	49	70
average time (ms/frame)	35	62

Table 7.1: The comparison between computational performance of KinectFusion System and that of KinfuSeg system

	average time (ms)
extract ICP outliers	1.5
mean filter	2
graph-cut (up to 3 iterations)	7
dynamic TSDF volume integration	8
dynamic ray casting	6

Table 7.2: The computational performance of KinfuSeg

Table 7.2 shows the average time each step consumes in KinfuSeg segmentation pipeline. Graph-cut method, volume integration and ray casting the three steps that slow down the whole process. The changes in TSDF volume and ray casting don't impair the performance greatly. Since the two steps are the necessary parts in the pipeline and are hard to optimized further, the graph-cut algorithm ceils KinfuSeg system real-time performance. Convergence speed normally slows down greatly when iteration exceeds three. Thus, we perform the graph-cut method with a maximum of three iterations. In our system, we don't expect the energy function in equation 6.6 to achieve a global minimum value in every frame. In such a system, the job of graph-cut is similar to a high-level filter. This allows the KinfuSeg system able to run at around 15 Frames Per Second (FPS), which ensures the system assumptions and the system robustness.

7.2.2 Segmentation Accuracy

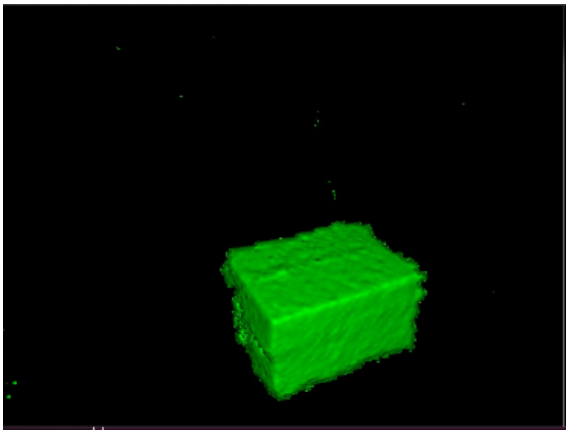
An alternation to the segmentation pipeline is use bilateral filter to smooth the raw vertex map and then render it. But its drawback is obvious compared with the 3D ray casting from TSDF volume. Figure 7.1 shows this difference. Figure 7.1b is a 3D map with a bilateral filter and shaded display while figure 7.1a is the ray casting view from 3D dynamic volume. The dynamic volume stores a more complete information in 3D. This not only leads to a better, more accurate segmentation from moving object to the static scene, but also improves the accuracy in future disparity estimation and camera pose tracking.

The segmented volume cuboid is measured, compared to the real box cuboid in figure 7.1c. The result is shown in table 7.3. The voxel in TSDF volume is $3cm$ for each. The measurement of segmented volume cuboid is approximated by comparing its lengths in three edges in the image window to the static volume scene. The length and width of dynamic volume have approximated

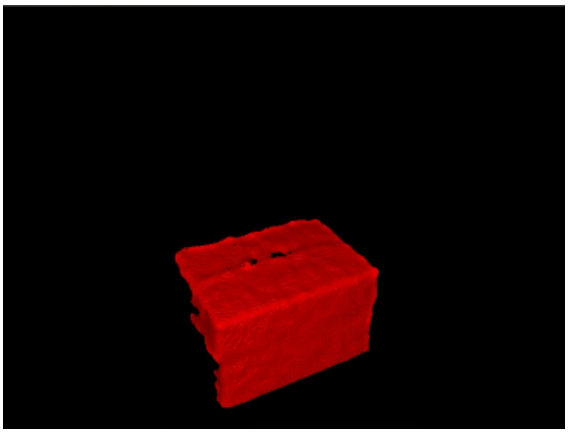
	width(cm)	**length**(cm)	**height**(cm)
real box	25	40	35
constructed dynamic volume	24	39	28

Table 7.3: The accuracy of dynamic TSDF volume compared to a real box

the same result to the real values but the height is shortened. The reason is that part of the box which is extremely close to the ground is not segmented out. This distance loss also approximates our distance threshold ($8cm$) in ICP outlier checking. If the box is lifted, the height can still be estimated correctly.



(a) 3D view from ray casting segmentation of object in dynamic TSDF volume



(b) Shaded view of segmentation in each image frame with bilateral filter

(c) The original RGB image

Figure 7.1: A comparison the segmentation of dynamic TSDF volume and that with bilateral filter in image domain.

In the following sections, we will demonstrate and analyse the capabilities of KinfuSeg system in various aspects. All the image results below are snapshots from our real-time videos, which can be accessed on this channel:

`http://www.youtube.com/channel/UCx-DbdC03CZiQqMhtliAwhg/videos` .

A general final demonstration of the whole system can be accessed here:

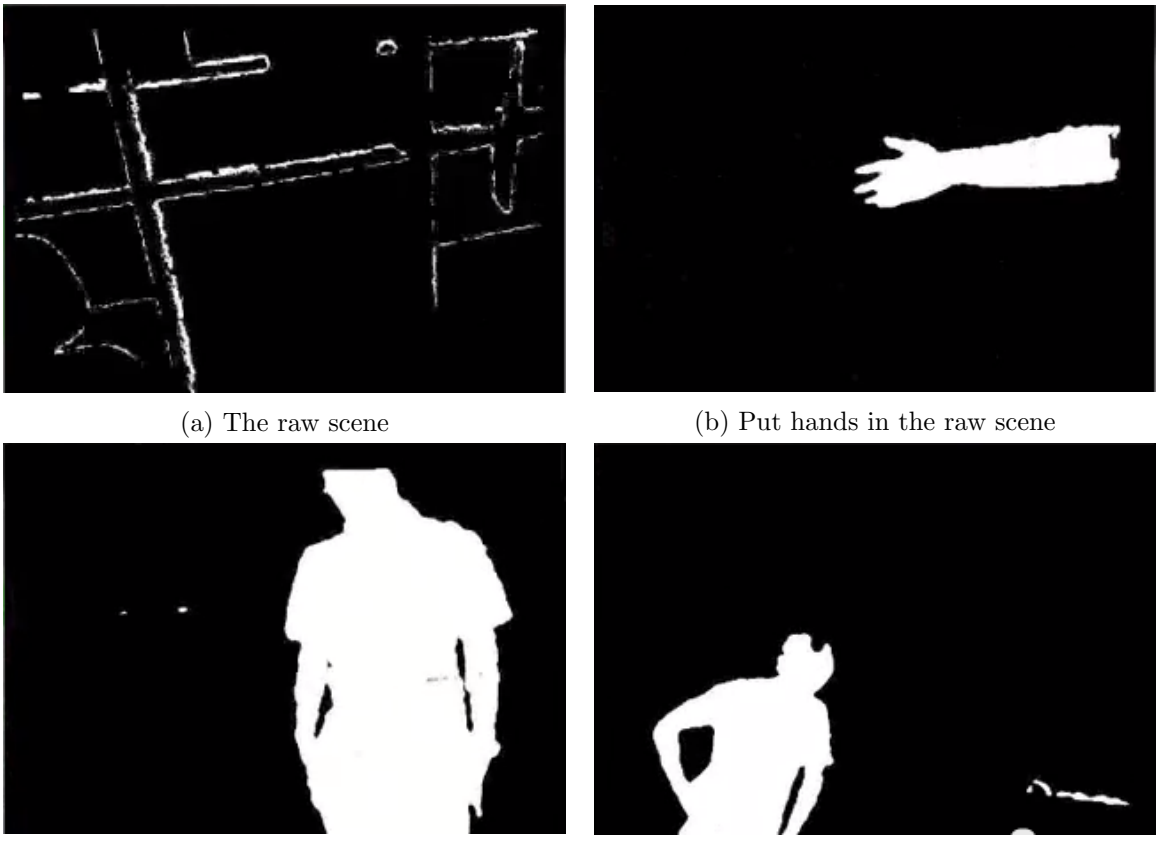
`http://www.youtube.com/watch?v=-ElbSDHQMYI`

7.3 Segmentation of Depth Disparity Map

The performance of segmentation on depth disparity map $\mathbf{D}_k$ is shown in figure 7.2. Figure 7.2a shows the disparity generated after the ICP outlier check. The other three images show the segmentation results after graph-cut. Hands, body and small motions can be extracted before a well constructed background. Details such as fingers can be shown clearly. Most background noises (edges, margins) are removed from the scene.

The segmentation result is not always perfect in every situation. Some noises still preserve (some small white points in the background in figure 7.2). If the moving object is large in scene, the disparity doesn't ensure to perfect segment all the details on the object. In graph-cut, we add in the image term in energy functions to reduce this effect. In some situations, this happens especially when the depth difference between foreground and background is too small or we haven't constructed the background scene which is now occupied by the foreground. A great graph-cut method can solve this problem, which will also be one of the focuses in our future work.

Although some noises still preserve, in the dynamic ray-cast step, some of them can be removed by the ray casting algorithm we described in algorithm . In next section, we will see the 3D view result from ray casting in KinfuSeg system.



(a) The raw scene

(b) Put hands in the raw scene

(c) Man enters the raw scene

(d) Multiple objects enter the raw scene

Figure 7.2: The raw depth disparity

7.4 Separate Modelling of Dynamic Motion and Static Scenes

While KinfuSeg is running, ray casting scenes of both static and TSDF volumes are downloaded to host machine and displayed at every frame. The TSDF volume are stored in GPU device. This section will demonstrate the real-time scene of ray casting view. For simplicity, the static scene are displayed as grey volume and the dynamic scene as green.

7.4.1 Modelling with Outer Object Entering into the Scene

Figure 7.3 demonstrates the situation that an object enters the static scene. In this situation, Kinect camera captures and constructs the static scene at the first several frames (figure 7.3a). Then an object out of the constructed static space enters in. In this situation, when object is detected as depth disparity, segmentation pipeline is opened to segment the foreground moving object. Meanwhile, KinfuSeg system will track the background scene and camera pose.

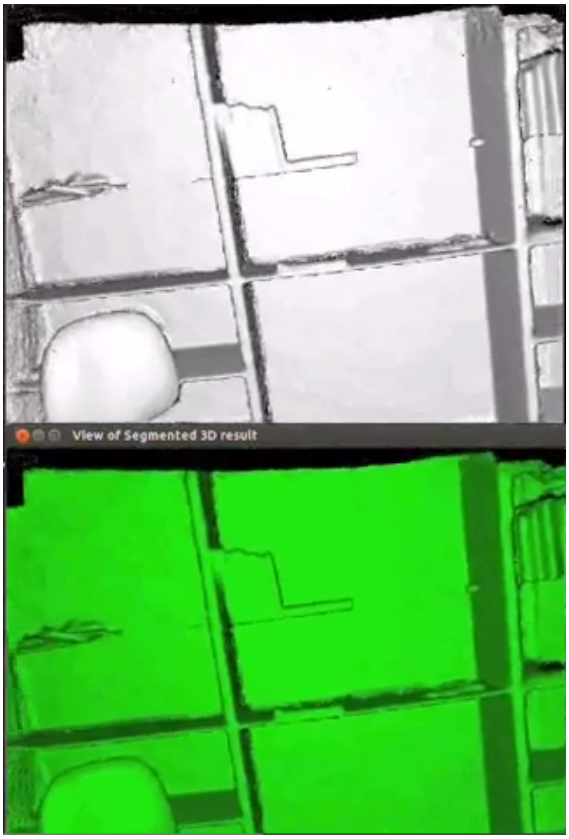
The four images in figure 7.3 are reconstruction snapshots from a motion sequence while tracking the camera. In the whole sequence, camera keeps its 6 DoF travel while new object keeps coming into the scene. Arm, human and books are tested. In figure 7.3c, a man taking a thin book from the shelf. The book, clothes and hair style can be seen clearly from the dynamic ray casting scene (color green). And meanwhile, the background scene is still smooth and keep updating, which involves more scenes when camera is moving.

7.4.2 Modelling with an Inner Object Moving in the Scene

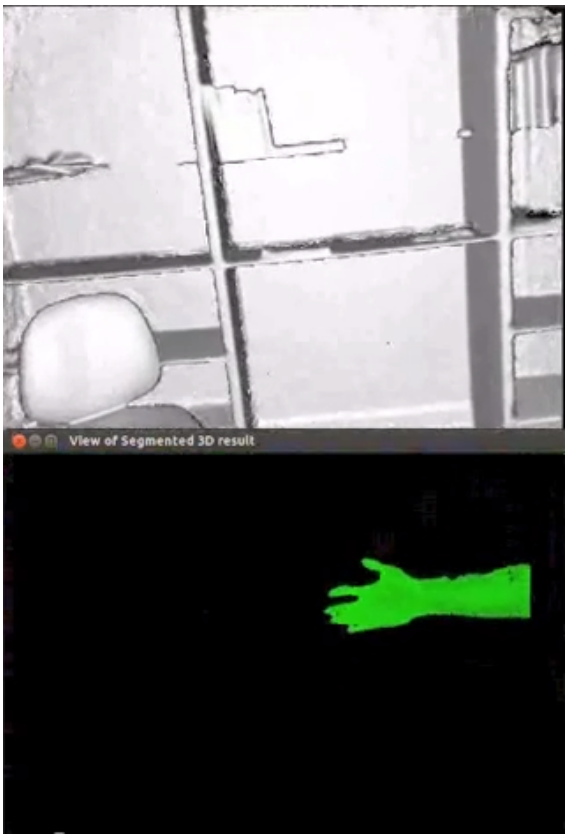
In section 5.3, we talked about how the situation is different when an inner object moves in the scene and described the solutions. A small example in figure 7.4 shows the performance of our system in this situation. When Kinect begin to capture its first frame image, a small box lying on the ground is included into the static background scene (figure 7.4a). Then the box is moved by a man to a new position. Box is detected as the moving object is thus segmented. After several frames, the original constructed box surface in the static scene fades away and never appears while the dynamic volume approaching a full reconstruction of the box volume.

The depth disparity map in figure 7.4 can help to explain how KinfuSeg processes in this situation. When the box is moved by a man, the man, the areas which the box move to and move away, are all included in disparity (figure 7.4b). With algorithm 1, static TSDF volume detects the change is a foreground motion and updates the surface in that area. Thus, surface in old box position begins to fade, which is now approaching the true ground. Meanwhile, false disparity area and begin to fade. So does that in that in dynamic volume. After tens of frames, the update of both volume approach the positive true surface in real life.

In this whole trip, the camera is also moving. The movement of camera contributes to a more accurate TSDF volume. Our KinfuSeg system is able to deal with a much more complex environment. One drawback is that these detection and update steps need about ten frames to successful modelling both the static scene and object. If a fast and consistent movement exist in the scene, KinfuSeg may produce a delayed static TSDF volume and blurred dynamic volume.

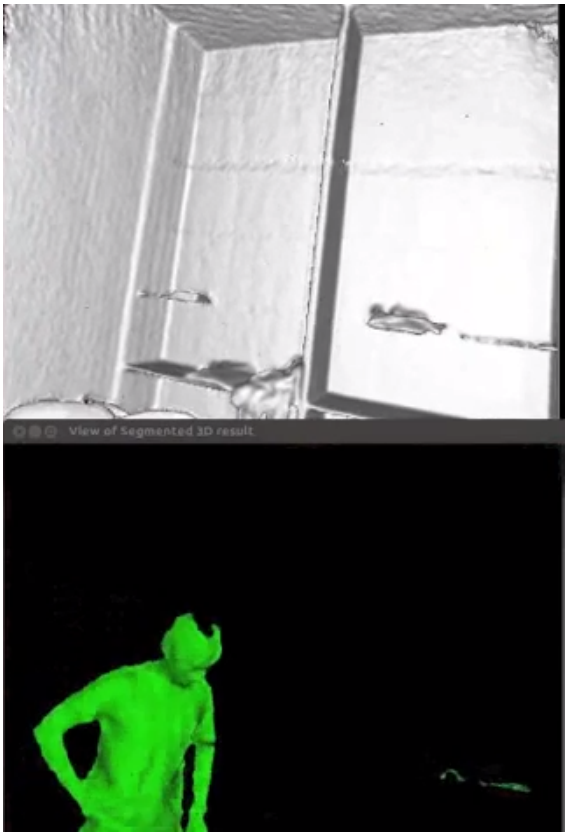


(a) The starting background. Both windows show the same scene.



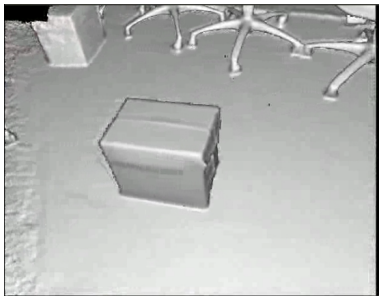
(b) A foreground hand segmented. Background scene is not affected.

(c) A foreground body segmented. Details such as holding book are shown.



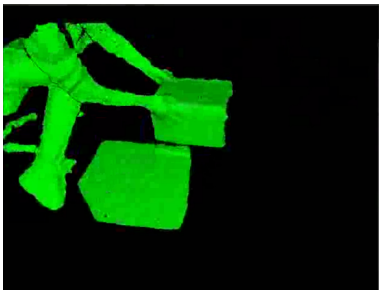
(d) A moving object segmented in the scene, when the motion speed is slow.

Figure 7.3: The constructed 3D scenes of static background (above) and dynamic foreground (below), without texture mapping



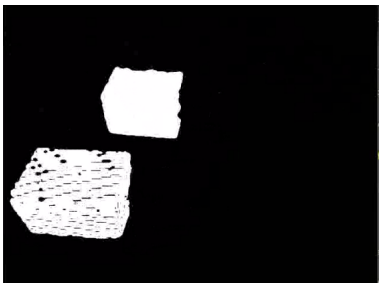
(a) The original static view

(b) The disparity map when a man moves the box

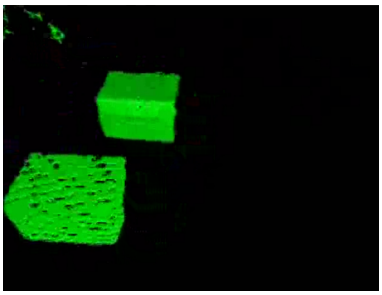


(c) The dynamic view from ray cast when a man moves the box

(d) The background scene when box is moved to a new position. Static scene begins to construct the new surface.



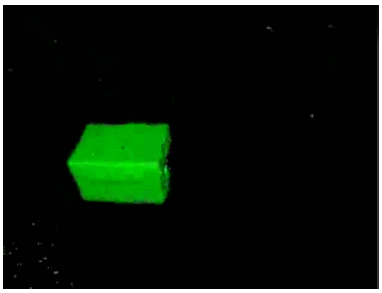
(e) The depth disparity map when box is moved. Two area with disparity exist at first, but one begin to fades away



(f) The dynamic view from ray cast when box is moved, corresponding to depth disparity map.

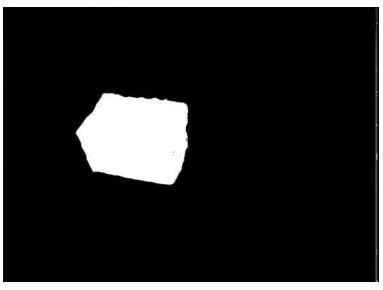
(g) The box fades in the static scene.

(h) The area where box moves away fades, depth disparity shows where the box now exist

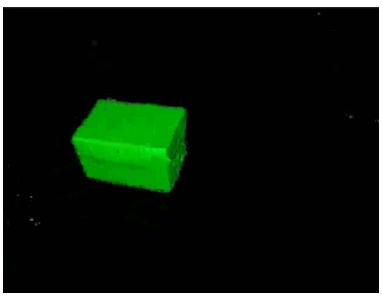


(i) The dynamic view from ray cast when box is now reconstructed.

(j) The final static scene.



(k) Final depth disparity map.



(l) The final view from ray cast of the box.

Figure 7.4: The steps show how the a box is moved from one place in the static scene and finally segmented. During the whole trip, camera is moving and tracked. The full video link can be accessed here: `http://www.youtube.com/watch?v=DCAu4aaIzIs`

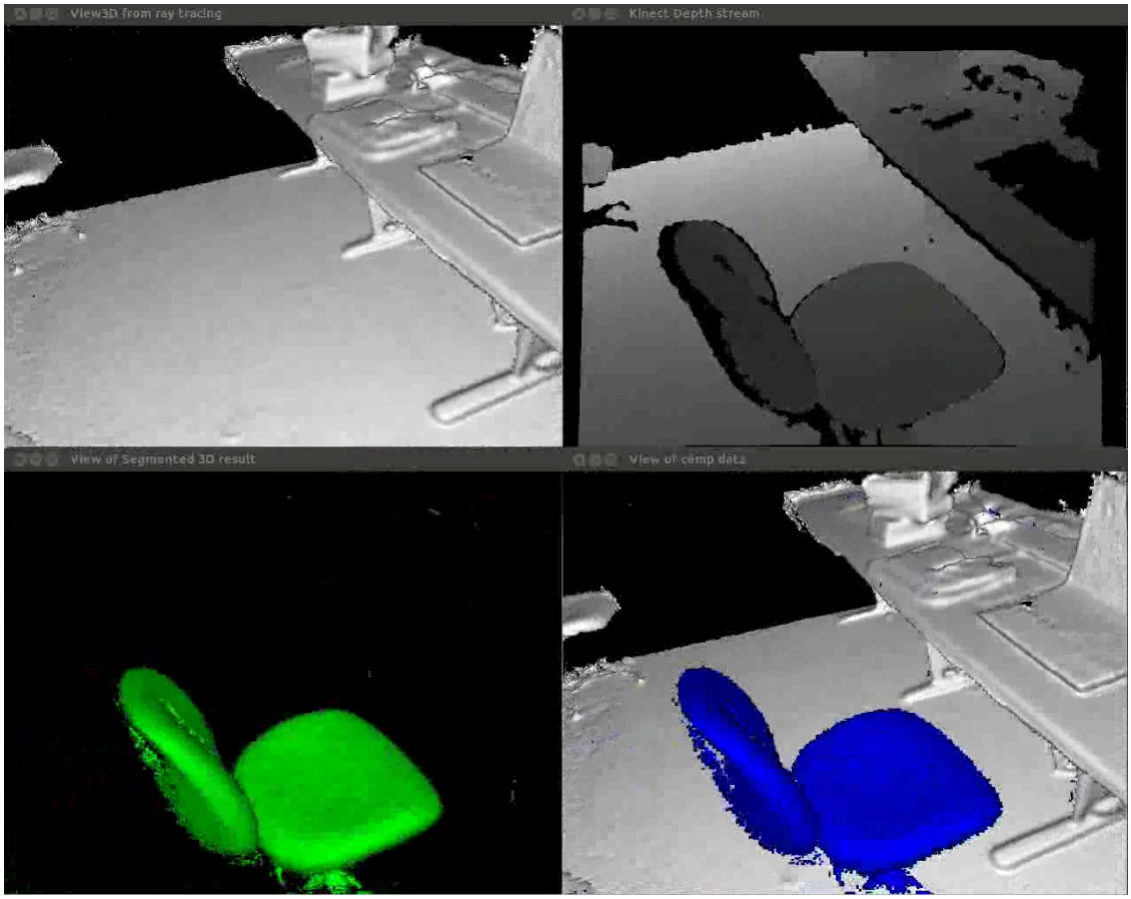
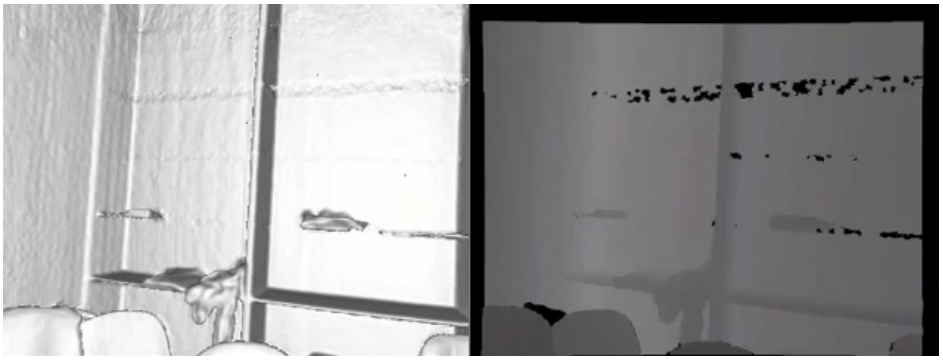


Figure 7.5: A general test of KinfuSeg system. The background surface is scanned first, and then move the chair to the desk.

7.5 Performing SLAM in a Dynamic Environment

In previous sections, the camera is always moving while mapping and construction are operating. If any inaccuracy in SLAM system, it will lead to drifts in all parts of the system: camera pose estimation, ICP mapping, surface construction, etc. If drift happens, it can be observed from the ray casting 3D view (surface meshes correspond to wrong positions). In our KinfuSeg system, when the camera motion is careful and small, ICP converges perfectly and drifts normally approximate to zero.

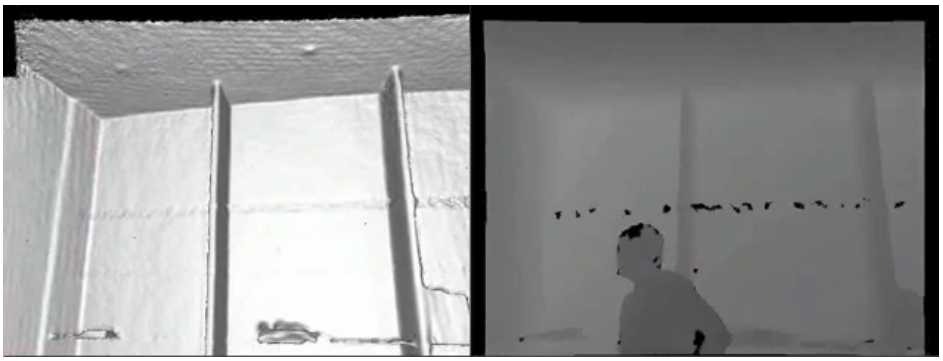
We perform a SLAM experiment in an indoor room to test the robustness of SLAM system. Figure 7.6 shows some snapshots in the results. In the whole test, a man keeps moving across the camera view (shown on the depth map in right column). In the left column of figure 7.6, a consistent mapping and construction is operated. In the whole trip, the body motions don't affect the surface estimation of background, and the tracking on camera is successful (there no sign of drifts that may exist).



(a) Starting background scene, with no motions inside

(b) The same background scene with small camera rotation, while a new human body enters in

(c) Background 3D constructed scene while the camera is travelling rightwards, and the body in dynamic foreground keeps moving



(d) The last 3D background scene when the camera move upwards, and the body is still inside

Figure 7.6: The backgrounds scenes while KinfuSeg tracking the camera. In all four images, the left one is the constructed scenes without texture mapping, and the right one is original depth captured by Kinect camera. The depth scene shows the real world condition (whether there is motions inside the scene).

7.6 Failure Modes and Exceptions

In the current software system, there are some situations which may lead our KinfuSeg system fail. There are mainly three failure conditions:

1. Object is not detected.
2. Object is segmented with great noise.
3. System lose tracking while constructing.

The first situation normally happens when segmented object is small in size and near to existing surface. This is because the distance difference of object to the background is smaller than the ICP threshold (about $5 - 8cm$). For example, a small mug or thin book on the desk will be ignored by the KinfuSeg system. For large objects, such as human motions, chairs, desk, can be detected successfully.

The second situation happens when the estimation of camera pose or surface is inaccurate in the last frame. The ICP converges, but has not reached the global minimum. The surface planes and points not aligned correctly will appear as noises in the current frame. This situation can be avoided when camera is moving carefully and system working in a relatively complex environment with multiple planes, not just one plane surface like a wall.

The third problem mostly lies in the current KinectFusion framework. The PCL KinectFusion is not robust enough to provide a successful tracking and mapping each time. This will definitely bring troubles into our KinfuSeg system. To avoid this, we don't perform KinfuSeg in an open space. Move camera slowly and carefully in the tracking. If the construction of the background fails, we need to restart the system and perform again.

There are also some existed bugs in PCL KinectFusion which haven't been solved. For example, color information cannot be correctly accessed. Occasionally the mapping kernel will report a exception when ICP doesn't converge. These problem affects our system performance. But in general, if conditions above can be avoided, our system can track well and produce good segmentation.

CHAPTER 8

Conclusions and Future Work

8.1 Conclusion of Work

The SLAM field is pushed by scientists and researchers over last two decades. In this three-month project, we focus on exploring methods to improve the performance of SLAM in a dynamic environment, based on the state-of-art dense SLAM KinectFusion system. As discussed in related and existing work in section 2.1, there is very little progress and improvement in methods to tackle a dynamic navigation and tracking in the visual SLAM system. In our thesis, we present a dense visual SLAM system KinfuSeg which bases on the robust dense RGB-D SLAM system KinectFusion . We try to approach a dynamic scene tracking and segmentation, by focusing on a proper improvement on KinectFusion system.

Our KinfuSeg system tackles the dynamic problem by introducing a separate segmentation pipeline, which not only successfully tracks the camera pose by removing the ICP outliers, but also reconstructs the segmented dynamic object in another TSDF volume. The implementation of such system is more than tracking of background scenes. This makes it possible to track and classify objects in the scene, which can benefit various interaction applications in AR field.

In our segmentation pipeline, we introduce two methods, mean filter and graph-cut to segment the object, and achieve a real-time performance. Although the reconstructions of both background and foreground scene are in global space, we implement the graph-cut on a 2D image, which reduces the computation greatly. The graph-cut model we set-up incorporating the depth and image information is also novel. With this pipeline, we are able to detect and segment smooth and small object motions in the scene. In our segmentation, the object is not restricted to be rigid. Dynamic object such as human motions, can be constructed smoothly within the scene.

The KinectFusion model only works on depth information from Kinect sensor, whose operation range is limited by a depth detection range of sensor. Since our work requires a prior construction of static scene of the background, the depth restriction problem quite often lead to a incomplete reconstruction in the first several frames. Hazardous situations for KinfuSeg can be indoor wide

open space, transparent glass windows, extreme smooth surface. In other conditions, KinfuSeg can work well within small indoor environments.

8.2 Future Research

The research on visual SLAM system in dynamic scenes is still in its early stage. This work can be approached from various aspects and many methods are waiting to be tested. I hope to move in this direction in my future PhD study if possible. Some of the immediate improvements related to this thesis that can be applied to improve our KinfuSeg system are listed as the following.

Current segmentation pipeline in KinfuSeg needs to be optimized. Although CUDA provides the API that makes GPGPU much easier to implement on GPU, the efficiency of program is still affected by the software implementation, such as consideration of GPU throughput (not well designed in our program). For example, In our implementation, to avoid bank conflict problems, we use many atomic functions or synchronize threads too often. Some more trivial parallel algorithms can be implemented to improve the efficiency. KinfuSeg is a real-time online SLAM system, which also has the small-motion assumption between frames. Higher running frequency means more accuracy results.

In KinfuSeg , object motions in our program are successfully segmented, but not tracked. In multi-body research of [22][16], a sparse SLAM system tracks the body movement. Such SLAM system can have better understanding of the surroundings. Tracking of object and camera is a coupled problem in any SLAM system. In our KinfuSeg system, if efficiency can be improved (more than 20 FPS), the tracking of object is possible by using an object ICP and assuming camera static in each frame. Or we can set landmarks in static scenes, and explore other alignment methods to reach the goal. After achieve tracking, we can also implement classifications and clustering on multiple-object. This will make robot approach human cognition in the real-world. [13] explores a multiple-object classification problem with graph-cut, which might be combined into our graph-cut algorithm.

The graph-cut method in our system can be improved, by including more characters in energy functions. In our system, only depth disparity is used. This brings some limitations that moving object is expected to have salient depth/vertex difference. At the object edge which is connected to static surface in the background, such segmentation might lose its accuracy. Graph-cut in image-processing has been proved to be robust and accuracy. If future work can incorporate color information, the object segmentation can be more smooth, more accuracy and more robust.

In addition, texture mapping in current KinfuSeg system is disabled since some bugs on the KinectFusion PCL framework. This and some other color visualization problems can be solved in the near future.

Bibliography

[1] Adrien Angeli and Andrew Davison. Live Feature Clustering in Video Using Appearance and 3D Geometry. *Procedings of the British Machine Vision Conference 2010*, pages 41.1–41.11, 2010.

[2] John Bastian, Ben Ward, Rhys Hill, Anton van den Hengel, and Anthony Dick. Interactive modelling for AR applications. In *2010 IEEE International Symposium on Mixed and Augmented Reality*, pages 199–205, Seoul, October 2010. IEEE.

[3] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001.

[4] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–37, September 2004.

[5] Daniel Cremers, Mikael Rousson, and Rachid Deriche. A Review of Statistical Approaches to Level Set Segmentation: Integrating Color, Texture, Motion and Shape. *International Journal of Computer Vision*, 72(2):195–215, August 2006.

[6] Andrew J Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1403–1410 vol.2, Nice, France, 2003. IEEE.

[7] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. MonoSLAM: real-time single camera SLAM. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1052–67, June 2007.

[8] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[9] S Geman and D Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE transactions on pattern analysis and machine intelligence*, 6(6):721–41, June 1984.

[10] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[11] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High performance computingHiPC 2007*, pages 197–208, Goa, India, 2007. Springer-Verlag.

[12] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[13] Hossam Isack and Yuri Boykov. Energy-Based Geometric Multi-model Fitting. *International Journal of Computer Vision*, 97(2):123–147, July 2011.

[14] Shahram Izadi, David Kim, and Otmar Hilliges. KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568, New York, NY, USA, 2011. ACM.

[15] Georg Klein and David Murray. Parallel Tracking and Mapping for Small AR Workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 1–10, Nara, November 2007. IEEE.

[16] Abhijit Kundu, K Madhava Krishna, and C. V. Jawahar. Realtime multibody visual SLAM with a smoothly moving monocular camera. *2011 International Conference on Computer Vision*, pages 2080–2087, November 2011.

[17] Bastian Leibe, Nico Cornelis, Kurt Cornelis, and Luc Van Gool. Dynamic 3D Scene Analysis from a Moving Vehicle. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, Minneapolis, MN, June 2007. IEEE.

[18] Point-Cloud Library. Kinfu extension to large scale. `http://www.pointclouds.org/blog/srcs/fheredia/index.php`.

[19] Kuen-han Lin and Chieh-chih Wang. Stereo-based simultaneous localization, mapping and moving object tracking. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3975–3980, Taipei, October 2010. IEEE.

[20] Steven Lovegrove. *Parametric dense visual SLAM*. PhD thesis, 2012.

[21] KL Low. Linear least-squares optimization for point-to-plane icp surface registration. *Chapel Hill, University of North Carolina*, (February):2–4, 2004.

[22] Rahul Kumar Namdev, Abhijit Kundu, K Madhava Krishna, and C. V. Jawahar. Motion segmentation of multiple objects from a freely moving monocular camera. *2012 IEEE International Conference on Robotics and Automation*, pages 4092–4099, May 2012.

[23] Richard a. Newcombe and Andrew J. Davison. Live dense reconstruction with a single moving camera. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1498–1505. Ieee, June 2010.

[24] Richard a. Newcombe, Andrew J. Davison, Shahram Izadi, Pushmeet Kohli, Otmar Hilliges, Jamie Shotton, David Molyneaux, Steve Hodges, David Kim, and Andrew Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, Basel, October 2011. IEEE.

[25] Richard a. Newcombe, Steven J. Lovegrove, and Andrew J. Davison. DTAM: Dense tracking and mapping in real-time. In *2011 International Conference on Computer Vision*, pages 2320–2327, Barcelona, November 2011. IEEE.

[26] nVidia Incorporation. Cuda 5.0 tool-kits official documents. `http://docs.nvidia.com/cuda/index.html`.

[27] Kemal Egemen Ozden and Kurt Cornelis. Reconstructing 3D trajectories of independently moving objects using generic constraints. *Computer Vision and Image Understanding*, 96(3):453–471, 2004.

[28] Kemal Egemen Ozden, Konrad Schindler, and Luc Van Gool. Multibody structure-from-motion in practice. *IEEE transactions on pattern analysis and machine intelligence*, 32(6):1134–41, June 2010.

[29] Shankar R. Rao, Allen Y. Yang, S. Shankar Sastry, and Yi Ma. Robust Algebraic Segmentation of Mixed Rigid-Body and Planar Motions from Two Views. *International Journal of Computer Vision*, 88(3):425–446, January 2010.

[30] Mike Roberts and Jeff Packer. A work-efficient GPU algorithm for level set segmentation. In *Proceedings of the Conference on High Performance Graphics Pages*, pages 123–132, Saarbrucken, Germany, 2010. Eurographics Association.

[31] Carsten Rother, V Kolmogorov, and Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics (TOG)*, 23(3):309–314, 2004.

[32] Anastasios Roussos, Chris Russell, Ravi Garg, and Lourdes Agapito. Dense multibody motion estimation and reconstruction from a handheld camera. In *International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 31–40, Atlanta, GA, November 2012. IEEE.

[33] Radu Bogdan Rusu. Point-cloud library. `http://pointclouds.org/`.

[34] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *International Conference on Robotics and Automation*, Shanghai, China, 2011 2011.

[35] Renato F Salas-moreno, Richard A Newcombe, and Paul H J Kelly. SLAM ++ : Simultaneous Localisation and Mapping at the Level of Objects. In *Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2013.

[36] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[37] F. Steinbrucker, J. Sturm, and D. Cremers. Real-time visual odometry from dense RGB-D images. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 719–722, Barcelona, 2011.

[38] René Vidal, Yi Ma, Stefano Soatto, and Shankar Sastry. Two-View Multibody Structure from Motion. *International Journal of Computer Vision*, 68(1):7–25, April 2006.

[39] Vibhav Vineet and P. J. Narayanan. CUDA cuts: Fast graph cuts on the GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–8, Anchorage, AK, June 2008. IEEE.

[40] S. Wangsiripitak and D.W. Murray. Avoiding moving outliers in visual SLAM by tracking moving objects. In *2009 IEEE International Conference on Robotics and Automation*, pages 375–380, Piscataway, NJ, May 2009. IEEE.

[41] Thomas Whelan, Michael Kaess, and Maurice Fallon. Kintinuous: Spatially extended kinectfusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, Sydney, Australia, 2012.

[42] Vladimir Kolmogorov Yuri Boykov, Daniel Cremers. Tutorial: Graph-cuts versus level-sets in european conference on computer vision, May 2006.