

## ABSTRACT

Title of Thesis:                   A REAL TIME IMPLEMENTATION OF 3D  
  SYMMETRIC OBJECT RECONSTRUCTION

  Liangchen Xi, Master of Science, 2017

Thesis Directed By:            Professor Yiannis Aloimonos  
  Department of Computer Science

KinectFusion is a surface reconstruction method to allow a user to rapidly generate a 3D model of an indoor scene by moving a standard Kinect camera. With a GPU-based pipeline, the system takes in live depth data from a moving Kinect camera and can create a high-quality 3D model in real time.

In this thesis, in order to increase the utility of 3D reconstruction for the symmetric objects, we augment the reconstruction algorithm by adding an occupancy mapping step on top of the surface reconstruction. This is achieved by combining the GPU-based KinectFusion with the CPU-based occupancy mapping algorithm Octomap. The resulting system runs in real time and is used as an input to the symmetry detection algorithm.

Tested by the Kinect video stream, our results demonstrate an accurate colored 3D surface reconstruction, which is useful for symmetry detection.

A REAL TIME IMPLEMENTATION OF 3D SYMMETRIC OBJECT  
RECONSTRUCTION

by

Liangchen Xi

Thesis submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park, in partial fulfillment  
of the requirements for the degree of  
Master of Science  
2017

Advisory Committee:  
Professor Yiannis Aloimonos, Chair  
Professor Cornelia Fermuller  
Professor Gang Qu

© Copyright by  
Liangchen Xi  
2017

## Acknowledgements

I owe my gratitude to all the people who have made this thesis possible.

First and foremost, I thank Prof. Yiannis Aloimonos for his great scientific guidance. Next, I would like to thank Prof. Cornelia Fermuller for her continuing and unfailing support throughout the work on this thesis.

Last but not least, I want to thank Aleksandrs Ecins for supporting my work on this thesis. His insightful comments and suggestions highly influenced both design and performance of the presented system.

# Table of Contents

Acknowledgements .....	ii
Table of Contents .....	iii
List of Tables .....	iv
List of Figures .....	v
Chapter 1: Introduction .....	1
Chapter 2: Reconstruction model.....	3
2.1 Camera space .....	3
2.2 Image space.....	3
2.3 Global space.....	3
2.4 Depth map.....	4
2.5 Point clouds.....	6
2.6 Volumetric TSDF.....	6
Chapter 3: KinectFusion .....	7
3.1 Bilateral filtering .....	7
3.2 Depth conversion .....	8
3.3 Camera tracking .....	9
3.3.1 Overview.....	9
3.3.2 Implementation .....	12
3.4 Volumetric Integration.....	12
3.4.1 Overview .....	12
3.4.2 Implementation .....	12
3.5 Color rendering .....	14
3.6 Ray casting.....	15
3.6.1 Overview.....	15
3.6.2 Implementation .....	16
Chapter 4: Octomap .....	17
Chapter 5: System design.....	19
5.1 Image correction .....	19
5.2 Multithreading.....	19
Chapter 6: Surface extraction.....	21
6.1 Marching cubes .....	21
6.2 Point cloud down sampling.....	22
Chapter 7: Experiments and results .....	23
7.1 Initialization.....	23
7.2 Surface reconstruction .....	24
7.3 Octomap.....	25
7.4 Down sampling .....	25
Conclusion .....	26
Appendices.....	27
Bibliography .....	28

## List of Tables

1. Camera pose estimation .....	11
2. TSDF integration without color .....	13
3. TSDF integration with color .....	14
4. Ray casting .....	16
5. Octomap .....	18
6. Marching cubes .....	21
7. Down sampling a point cloud .....	22

## List of Figures

1.1 GPU pipeline of KinectFusion.....	2
2.1 Camera space/Image space/Global space .....	3
2.2 Depth map.....	5
2.3 Raw data from Kinect.....	5
3.1 Depth pyramid/camera tracking system.....	10
3.2 Ray casting.....	16
4.1 Octree/Point cloud/Octomap.....	17
4.2 System for real-time surface reconstruction .....	19
6.1 Volume and camera initialization.....	23
6.2 Surface reconstruction from video A .....	24
6.3 Surface reconstruction from video B .....	24
6.4 Surface reconstruction from video C .....	25
6.5 Octomap from video C.....	25
6.6 Point cloud down sampling.....	25

## Chapter 1: Introduction

Algorithms used for 3D surface reconstruction based on consecutive depth maps were well studied. With a core GPU pipeline of four stages described by figure 1.1, KinectFusion was the first system to achieve real-time 3D reconstruction with a low-cost handheld Kinect camera scanning. However, neither of those existing systems could provide both the real-time 3D surface reconstruction and occupancy mapping, which are necessary for symmetric detection algorithm introduced by [1].

In this work, in order to increase the utility of real time 3D surface reconstruction for the symmetric objects, we combine the GPU-based pipeline of KinectFusion with the CPU-based occupancy mapping algorithm Octomap.

Firstly we implement and test the real-time GPU-based KinectFusion system. Compared to [2], on the one hand, we use two strategies to ensure the accuracy of camera tracking: (i) Apply the bilateral filtering module to remove the noise on the raw depth data. (ii) A power-of-two depth pyramid is constructed to increase the efficiency and range of convergence. On the other hand, for steady TSDF integration, the KinectFusion integration equation is modified both for coloring and non-coloring vertices.

Secondly, we augment the surface reconstruction algorithm by adding an occupancy-mapping step Octomap on top of the surface reconstruction KinectFusion. The resulting system runs in real time and is useful for symmetric detection.

Finally our implementation adds more capabilities to extract the reconstruction surface, for example, (i) visualize the fusion process frame by frame in



the interactive mode, (ii) visualize and export the point clouds, surface normal, meshes, and (iii) down-sample the point clouds result by GPU.

The remainder of this work is structured as follows: After introducing the basic mathematical background for KinectFusion in chapter 2, our GPU-based KinectFusion implementation is described in details in chapter 3. In chapter 4, we introduce the Octomap and then in chapter 5, we add this occupancy mapping step on top of surface reconstruction by multithreading. In chapter 6, we provide two GPU-based methods to do surface extraction: marching cubes and point cloud down sampling. We finish with demonstrating our experiment results in chapter 7 and a conclusion.

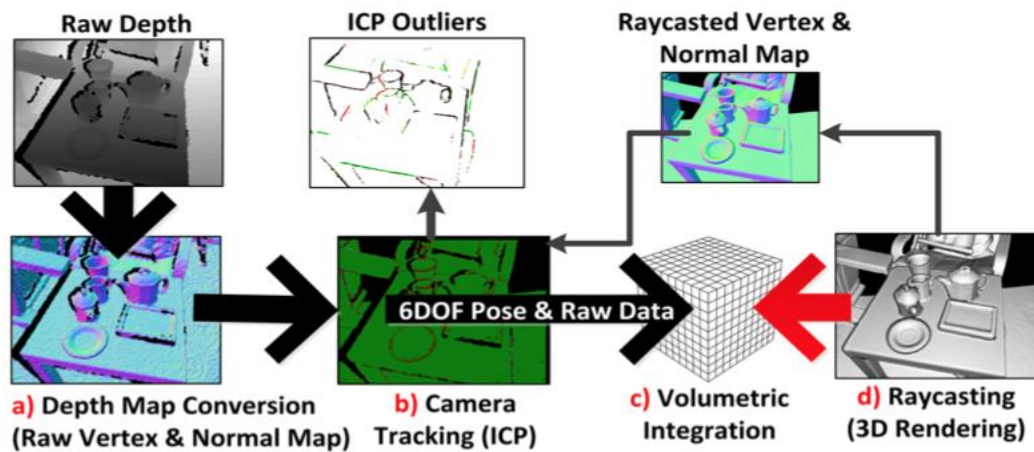


Figure 1.1: GPU pipeline of KinectFusion, image from [2]

## Chapter 2: Reconstruction model

This chapter gives a short overview about the scene's geometry described by figure 2.1. Then we introduce the Kinect camera and volumetric TSDF model used in this work. The advantages and disadvantages of our surface reconstruction model are discussed.

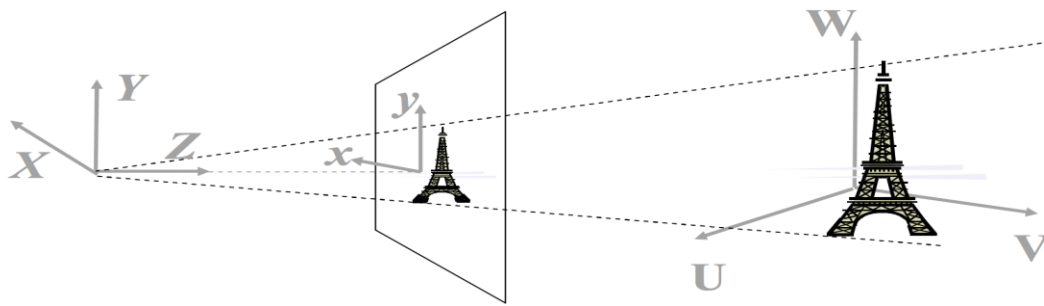


Figure 2.1: Camera Space / Image Space / Global Space, images from [3]

### 2.1 Camera Pose

The camera space is a 3D camera coordinate system. The Kinect camera is placed at the origin  $(0, 0, 0)$  in this system and looking down the Z axis. The position of camera is also pre-defined and can be adjusted. The mapping of camera space to global space is a 6 DOF problem (3 for rotation and 3 for translation), which can be described in matrix form  $T$  as following:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{0,0} & r_{0,1} & r_{0,2} & t_x \\ r_{1,0} & r_{1,1} & r_{1,2} & t_y \\ r_{2,0} & r_{2,1} & r_{2,2} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.1)$$

Given the camera pose  $T$ , a vertex  $v_g$  in the global space can be projected into the camera space vertex  $v$ , by  $v = T * v_g$ .

## 2.2 Image space

The image space is a 2D image coordinate system. The image consists of pixels and is pertained by the camera's view frustum. Each pixel can be projected to the camera space by projection matrix  $\mathbf{K}$ , which is defined as following:

$$\mathbf{K} = \begin{bmatrix} -f_x & 0 & c_x \\ 0 & -f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2.2)$$

$(c_x, c_y)$  is the center of image and  $(f_x, f_y)$  is the focal length. These 4 parameters can be obtained when calibrating the Kinect camera. So the equation for projecting an image space pixel  $\mathbf{u} = (x, y)$  to camera space vertex  $\mathbf{v}$  is:

$$\mathbf{v}(\mathbf{u}) = D(\mathbf{u})\mathbf{K}^{-1}(\mathbf{u}, 1) \quad (2.3)$$

## 2.3 Global space

The global space is a 3D world coordinate system. All of the reconstruction will be implemented within a volume, of which the size and volume number is pre-defined and can be adjusted. The center of volume is placed at  $(0, 0, 0)$  and the coordinates are measured in millimeters.

## 2.4 Depth map

From 2D pixel coordinates alone, it is impossible to reconstruct a unique 3D model, because all points on a line through the origin will be projected to the same pixel [4]. To get a unique solution, a third parameter, the depth, is necessary.

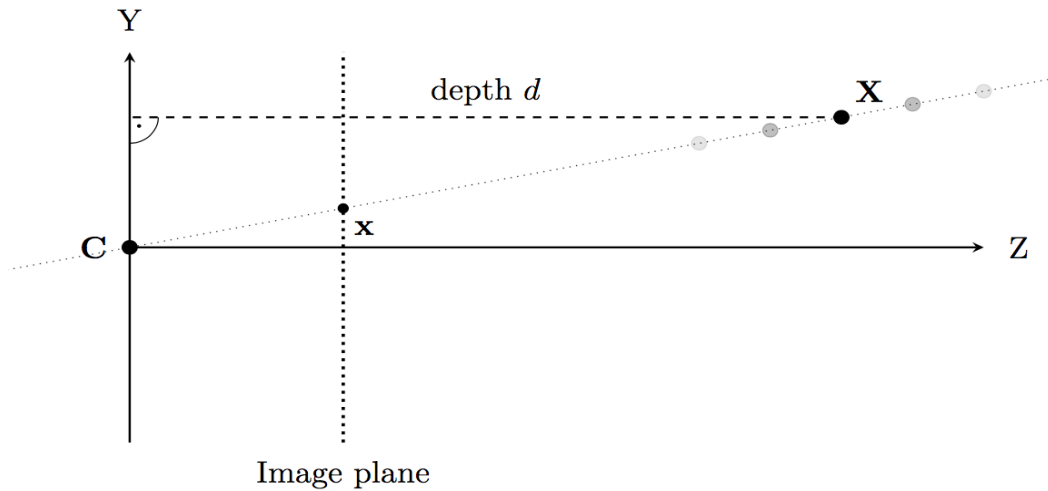


Figure 2.2: The depth  $d$  of pixel  $x$  is the distance of the point  $X$  to the camera plane  $CY$ , and it is normal to the viewing direction  $CZ$ . Varying  $d$  changes the position of  $X$  along the line  $CX$ , but its projected pixel  $x$  remains the same, image from [4].

The Microsoft Kinect RGB-D sensor, like other RGB-D sensors, provides color information as well as the estimated depth for each pixel. The Kinect Camera has 30 fps, and the resolution for the depth map and RGB image is both 640 by 480. With a single Kinect camera, we can reconstruct a unique 3D surface for the objects by the depth and RGB images from the Kinect video stream.



Figure 2.3: Raw data from Kinect. Depth map (Noisy) / RGB image

## 2.5 Point clouds

A point cloud is a loose collection of 3D points representing object surfaces in the scene [4]. A Point cloud can be colorless or annotated with color. Our KinectFusion implementation reconstructs the object surface by point clouds both with color and without color.

## 2.6 Volumetric TSDF

The point cloud model is enclosed within a 3D volume, where each voxel contains a truncated signed distance (TSDF) function value to represent the voxel's distance [4]. Then we can find the object surface by finding the TSDF's zero crossing (TSDF sign changes), which needs to be interpolated from the voxel grids. More details will be discussed in Chapter 3.7.

The key advantage to use a volumetric TSDF is that multiple camera views can be combined and extended each other, thus smoothing out the depth noise and filling holes on the depth map [4]. Also for symmetry detection, multiple views of a same object is necessary to ensure the symmetry property.

However, the main disadvantage is that this reconstruction model must be within a fixed size volume, so computational resources need to be assigned for each voxel inside the volume [5]. The zero crossing might only occupy a small part of the volume, but memory has to be allocated for processing each voxel, which is not memory efficient [6].

## Chapter 3: KinectFusion

In this chapter, the GPU-based KinectFusion algorithm is discussed and implemented for real time 3D surface reconstruction.

### 3.1 Bilateral filtering

The raw depth map received from the camera is noisy and there may be black holes indicating no depth data available, thus filtering is necessary to smooth the raw depth signals [7]. Bilateral filtering is a good method to smooth the depth image and remove the noise, while still preserving the edges. The update formula is as follows:

$$g(x, y) = \frac{D(x, y)w(x, y, i, j)}{\sum_{i, j} w(x, y, i, j)} \quad (3.1)$$

In equation 3.1,  $D(x, y)$  is the depth value and  $g(x, y)$  is the pixel value. The pair  $(x, y)$  indicates the pixel position, while  $(i, j)$  indicates the filter radius, which is also known as the kernel size. The weighting  $w(x, y, i, j)$  is the product of spatial Gaussian  $G_{\sigma_d}$  and range Gaussian  $G_{\sigma_r}$ , as following:

$$w(x, y, i, j) = \exp\left(-\frac{(x - i)^2 + (y - j)^2}{2\sigma_d^2} - \frac{\|D(x, y) - D(i, j)\|^2}{2\sigma_r^2}\right) \quad (3.2)$$

With the predefined parameters as discussed above, each GPU thread operates in parallel on the pixel  $\mathbf{u} = (x, y)$  on the depth map  $D(\mathbf{u})$  of current frame, and the depth value for each pixel is then filtered with the weight function inside the kernel window.

## 3.2 Depth conversion

Given the intrinsic calibration matrix  $K$  of the depth camera, the re-projected 3D vertex in the camera space is defined as following:

$$\mathbf{v}(\mathbf{u}) = D(\mathbf{u})K^{-1}(\mathbf{u}, 1), \mathbf{u} = (x, y) \quad (3.3)$$

In other words, for a point  $\mathbf{u}$  in the depth map, the corresponding 3D vertex  $\mathbf{v}$  can be calculated by the following equation, where  $D$  represents the depth map.

$$\begin{cases} v.x = (u.x - c.x) * D(\mathbf{u}) / f.x \\ v.y = (u.y - c.y) * D(\mathbf{u}) / f.y \\ v.z = D(\mathbf{u}) \end{cases} \quad (3.4)$$

By using the neighbors of the re-projected vertices, we can calculate the normal vectors by the following equation, after which the value is normalized.

$$n_{g,i}(x, y) = (v_{g,i}(x + 1, y) - v_{g,i}(x, y)) \times (v_{g,i}(x, y + 1) - v_{g,i}(x, y)) \quad (3.5)$$

By equation 3.4 and 3.5, we can convert the depth map into the distance map, then the normal map.

Moreover, in order to increase the efficiency and convergence for the ICP process, we use a coarse-fine depth pyramid over a power of two. The lowest level is the original depth map after bilateral filtering, and each level is half of the previous level by down sampling, then each pixel value is averaged in a pre-defined window. As indicated by [8], this optimization is more stable than 6DOF estimation when the number of pixels considered is low, helping to converge for large pixel motions, even when the true rotation is not strictly rotational. With the depth pyramid, we also build a normal pyramid corresponding to depth map in each pyramid level.

## 3.3 Camera tracking

### 3.3.1 Overview

The ICP (Iterative Closest Point) algorithm is widely used for geometric shape alignment of 3D models. Given the camera pose in the global space, the stream of depth maps can be correctly fused into a single 3D model [2]. In the global space, the camera pose consists of 6 unknown parameters, 3 for translation and 3 for rotation. In this stage, we estimate the camera pose for each frame in the global space by estimating a single rigid 6DOF transformation between two consecutive frames.

The input of ICP is the consecutive points and normal in two consecutive frames, and the output is the 6DOF transformation matrix  $T$ , which describes the translation and rotation of the camera in the global space.

We set the first frame correspond to the global coordinates. Every frame will be aligned to the previous one, thus eventually all the frames can be aligned to the first frame [2]. The ICP method for camera tracking can be described as an optimization problem, and further a linear optimization problem as follows. Assume:

- Previous Frame: Source point  $s_i = (s_{ix}, s_{iy}, s_{iz})$
- Current frame: point correspondence for  $s_i$ :  $d_i = (d_{ix}, d_{iy}, d_{iz})$
- Surface Normal at point  $d_i$ :  $n_i = (n_{ix}, n_{iy}, n_{iz})$

Given the point correspondence, the goal of ICP is to find matrix  $T$  that minimize the squared distances between source point and tangent plane at the corresponding destination point [2], and  $T$  is the global camera pose estimation.

$$\sum_{D_i(u)>0} ((T * s_i - d_i) * n_i)^2$$



Based on the assumption of slight movement between consecutive frames, the target function can be transformed into linear least square problem by approximation, and then solved as a 6-by-n linear system using SVD algorithm as mentioned in [9] and [10].

### 3.3.2 Implementation

With the distance and normal pyramid described in chapter 3.2, then ICP is done iteratively at different pyramid levels from the coarsest level to the full to incrementally update the transformation matrix  $T$ , by minimizing the sum squared errors during iteration in each pyramid level. At each ICP iteration, the first step of the ICP algorithm is to find the point correspondence, then the error is minimized to get the increment change of camera pose matrix  $T$  by SVD decomposition. The point correspondence can be found by pixel parallel on GPU with the ray casting result as previous reference frame, while the SVD decomposition by OpenCV library on CPU, as described in Algorithm 1.

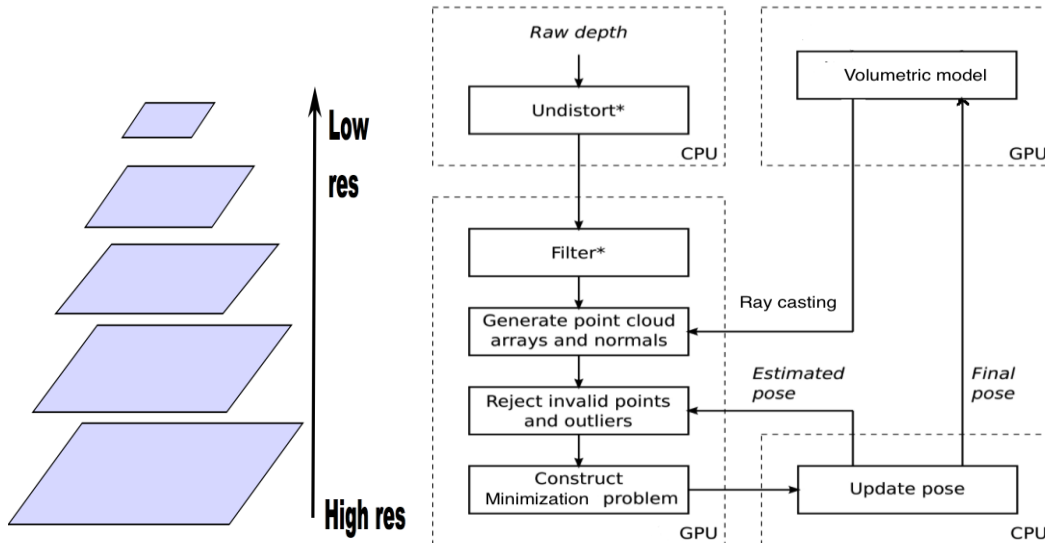


Figure 3.1: Depth pyramid / Camera tracking system

---

**Algorithm 1** Algorithm for Camera Pose estimation

---

**GPU – Find the point correspondence****for** each depth map, pixel  $u \in$  depth map  $D_i$  in **parallel do**    **if**  $D_i(u) > 0$  **then**

$v_{i-1} = T_{i-1}^{-1} v_{i-1}^g$

 $p \leftarrow$  *perspective project vertex*  $v_{i-1}$         **if**  $p \in$  *vertex map*  $V_i$  **then**

$v = T_i V_i(p)$

$n = R_i N_i(p)$

**if**  $\|v - v_{i-1}^g\| <$  *distance threshold* **and**                 $abs(n \cdot n_{i-1}^g) <$  *normal threshold* **then**

point correspondence found

**end if**    **end if****end****CPU – SVD decomposition to update T (OpenCV)****for** the depth map in each pyramid

find the points and correspondence in previous frame

**while** not reaching ICP iteration times

Apply SVD algorithm to update the T matrix

**end****Output T**

---

## 3.4 Volumetric Integration

### 3.4.1 Overview

At this stage, truncated distance function (TSDF) is used to fuse the new depth map with the existing volumetric model in the global space.

A 3D volume of a predefined resolution is used to represent the space we are measuring. This volume is uniformly subdivided into a 3D grid of voxels, to which those global 3D vertices will be integrated using a TSDF value. The value inside each voxel is the distance to closest surface and the distance is signed and truncated. The TSDF values are positive in front of the surface and negative behind the surface, with the zero crossing specifying the surface of the model. To determine the surface of the object, tri-linear interpolation is needed to find the zero crossings between each voxel. The SDF can be calculated as follows, and then truncated:

$$SDF = \| s - c \| - \| v - c \| \quad (6.1)$$

Where  $s$  is the surface vertex,  $v$  is the voxel vertex and  $c$  is the camera position.

### 3.4.2 Implementation

To achieve real-time rates, we basically implement the pseudo code in [2]. Due to the large number of voxels, we only assign threads to each  $(x, y)$  voxel on the  $x, y$  slice of the volume, then iterate all of the  $z$  values. Each voxel is firstly mapped to the vertex in the global space, then projected into the camera space using the camera pose estimation by ICP, and finally projected to the image space to determine the SDF value by looking up the distance map.

For CUDA optimization, we make the distance map built in chapter 3.2 as texture memory for quick look-up. During integration process, KinectFusion gives the following TSDF update for each voxel:

$$TSDF_{avg} = \frac{TSDF_{i-1} * w_{i-1} + TSDF_i * w_i}{w_{i-1} + w_i} \quad (6.2)$$

However, our results were less steady using Equation 6.1. Replacing the equation with 6.2 below solved this, as more weight is given to the existing value.

$$TSDF_{avg} = \frac{TSDF_{i-1} * w_{i-1} + TSDF_i}{w_{i-1} + 1} \quad (6.3)$$

**Algorithm2** illustrates the main steps of our implementation with the (6.2) equation.

---

**Algorithm 2** TSDF integration without color

---

```

for each voxel g in x,y volume slice in parallel do
    while sweeping from front slice to back do
         $\mathbf{v}^g \leftarrow$  convert  $g$  from grid to global 3D
         $\mathbf{v} \leftarrow T_i^{-1} v^g$ 
         $\mathbf{p} \leftarrow$  perspective project vertex  $\mathbf{v}$ 
        if  $\mathbf{v}$  in camera view frustum then
             $\mathbf{sdf}_i \leftarrow D_i(\mathbf{p}) - \|\mathbf{v}\|$ 
            if ( $\mathbf{sdf}_i > -\text{max\_truncation}$ ) then
                 $\mathbf{tsdf}_i \leftarrow \min(1, \mathbf{sdf}_i / \text{max\_truncation})$ 
                 $\mathbf{tsdf}_{i-1}, \mathbf{w}_{i-1} \leftarrow$  fetch TSDF and weight at voxel  $g$ 
                 $\mathbf{w}_i \leftarrow \min(\text{max weight}, \mathbf{w}_{i-1} + 1)$ 
                 $\mathbf{tsdf}^{avg} \leftarrow (\mathbf{tsdf}_{i-1} \mathbf{w}_{i-1} + \mathbf{tsdf}_i) / \mathbf{w}_i$ 
                store  $\mathbf{w}_i$  and  $\mathbf{tsdf}^{avg}$  at voxel  $g$ 
            end if
        end if
    end if
end
end

```

---

### 3.5 Color rendering

At this stage, we want to find the corresponding color for each vertex by camera calibration.

When calibrating the Kinect camera, the calibration matrix between the depth and RGB camera is also obtained, with which we can know how to project the pixel color from the RGB image space to Depth camera space, then to the volume in the global space. Thus the pose of RGB camera can be estimated from the depth sensor.

The only change from non-coloring TSDF integration is that the camera pose  $T$  is now the RGB camera instead of the depth sensor, and TSDF values needs to be updated for R,G,B channels. The algorithm is described in Algorithm 3 as follows.

---

**Algorithm 3** TSDF integration with color

---

```
for each voxel  $g$  in  $x,y$  volume slice in parallel do
  while sweeping from front slice to back do
     $v^g \leftarrow$  convert  $g$  from grid to global 3D
     $v \leftarrow T_i^{-1} v^g$ 
     $p \leftarrow$  perspective project vertex  $v$ 
    if  $v$  in camera view frustum then
       $sdf_i \leftarrow D_i(p) - \|v\|$ 
      if ( $sdf_i > -\max\_truncation$ ) then
         $v \leftarrow$  rgb TSDF volume at voxel  $g$ 
         $w_{i-1} \leftarrow$  weight at voxel  $g$ 
         $w_i \leftarrow \min(\max\_weight, w_{i-1} + 1)$ 
         $v.r \leftarrow (v.r * w_{i-1} + v.r) / w_i \leftarrow$  red
         $v.g \leftarrow (v.g * w_{i-1} + v.g) / w_i \leftarrow$  green
         $v.b \leftarrow (v.b * w_{i-1} + v.b) / w_i \leftarrow$  blue
        store  $w_i$  and  $v$  at voxel  $g$ 
      end if
    end if
  end while
end for
```

---

## 3.6 Ray casting

### 3.6.1 Overview

At this stage, a GPU-based ray casting is implemented to render the surface of current view. Also the result is used as a reference frame for next ICP iteration.

Given the position of the camera to cast ray to the volume of TSDF volume, the GPU threads march along the ray with a pre-defined ray casting step size. At each step, the global position of the ray will be re-projected to a voxel within the volume and the zero crossing of TSDF values will be checked to look for the surface. As discussed in the chapter 3.4, TSDF values are positive in front of the surface but negative behind. So, if a negative TSDF value is found earlier than a positive TSDF, then the implicit surface must have been neglected and such zero crossing is not the true surface. So we must look for the sign change of TSDF values from positive to negative.

Let  $Vox(x, y, z)$  be the voxel that contains the TSDF zero-crossing, we can use the following equations to determine the surface normal:

$$\begin{aligned}n_x &= Tsd f(Vox(x + 1, y, z)) - Tsd f(Vox(x - 1, y, z)) \\n_y &= Tsd f(Vox(x, y + 1, z)) - Tsd f(Vox(x, y - 1, z)) \\n_z &= Tsd f(Vox(x, y, z + 1)) - Tsd f(Vox(x, y, z - 1))\end{aligned}$$

The vector is then normalized.

With the vertex and its normal vector, a lighting equation is calculated for each pixel by GPU thread in the output image, in order to render the current view of the model [7].

### 3.6.2 Implementation

As our TSDF volume is truncated by a pre-defined distance, we can set a ray-casting step-size to accelerate the speed [7]. Once the sign change of TSDF values from positive to negative is detected, a trilinear interpolation is implemented between the two voxels to accelerate the speed to predict the surface position. The algorithm and result is described as follows.

---

**Algorithm 4** Ray casting

---

```
Given the camera pose estimation  $T$ ,  
for each voxel  $\mathbf{g}$  in  $x, y$  volume slice in parallel do  
   $\mathbf{ray}_{org} \leftarrow$  ray casting origin  
   $\mathbf{ray}_{dir} \leftarrow$  ray casting direction  
   $\mathbf{range} \leftarrow$  compute intersection of the ray with volume's six planes  
   $\mathbf{tsdf}_{cur} \leftarrow$  fetch the current tsdf value along the ray  
  while  $\mathbf{cur}$  is within the  $\mathbf{range}$   
     $\mathbf{tsdf}_{next} \leftarrow$  fetch the next tsdf value along the ray  
    if  $\mathbf{tsdf}_{cur} > 0$  and  $\mathbf{tsdf}_{next} < 0$  then  
       $\mathbf{vertex} \leftarrow$  global intersection by trilinear interpolation  
       $\mathbf{normal} \leftarrow$  calculate the normal at  $\mathbf{vertex}$   
      update the normal and depth maps for current rendering  
    end if  
     $\mathbf{cur} \leftarrow$  next  
     $\mathbf{next} \leftarrow$  increment by the raycaster step size  
  end  
end
```

---

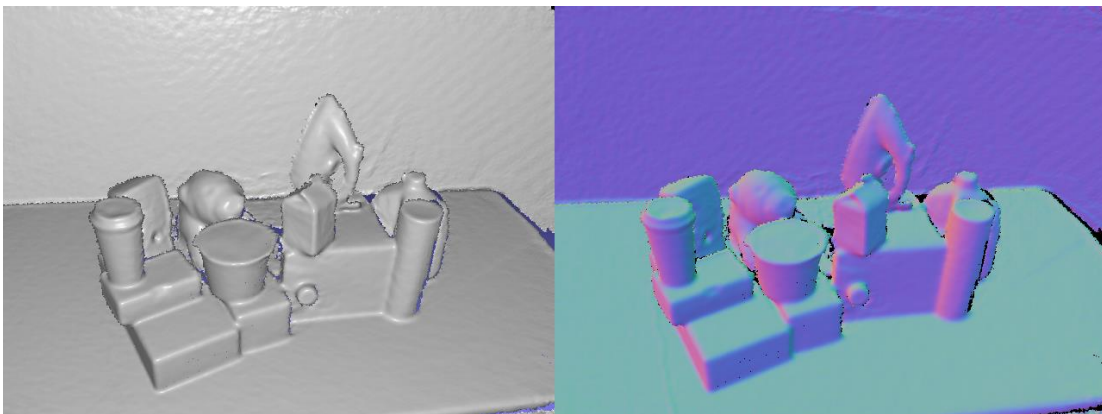


Figure 3.2: Ray casting: Reconstructed surface / Surface with normal

## Chapter 4. Occupancy mapping

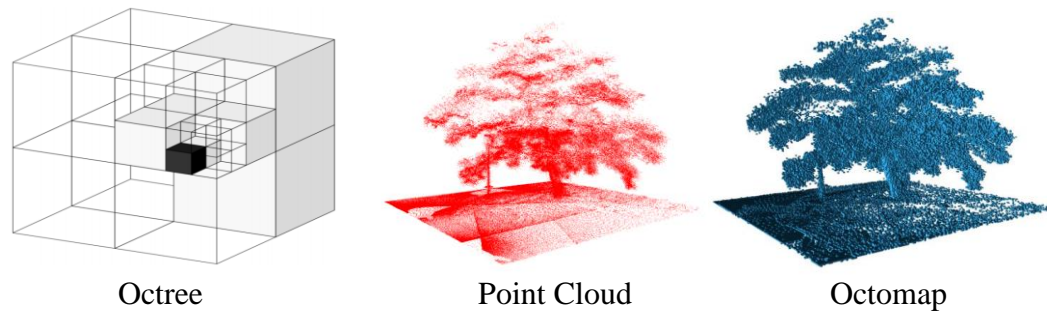


Figure 4.1: Octree/Point Cloud/Octomap. Images from [11].

Octomap is an efficient probabilistic 3D Mapping Framework Based on the Octree. It is a volumetric representation of occupied, free and unknown space based on the tree structure Octree [11]. Octree is used to recursively subdivide the cube into small cells. Compared to the fixed grid or volume, the main advantage of tree structure is memory efficient, as the empty regions do not use memory, and it is not limited to a fixed scene size because the Octree can dynamically expand the depth.

The main reason to implement the Octomap is for the symmetry detection indicated by [1], we need the occupancy mapping to determine a true symmetry or not. So for the symmetry detection, Octomap needs to be implemented in real time on top of the surface reconstruction.

To generate the Octomap, we need the inputs as the depth map and its corresponding camera pose to insert into the Octree. With the current open-source Octomap library, the tree insertion on CPU is slow and cannot be implemented in real time. One insertion of a resolution of 0.005m would cost one second, which is larger than the 30 Fps of the Kinect camera.



One way to speed up the Octomap is to decrease the resolution, which however, cannot guarantee a good symmetry detection, because resolution larger than 0.01m is not good for symmetry detection. Another way to solve this is to down-sample the depth frames and depth map. In other words, we only process down-sampled depth frames, and we also spatially down sample those depth maps. The main steps are illustrated as follows.

---

**Algorithm 5** Octomap

---

Initialize the Counter and Octree by pre-defined resolution  
**for** each depth frame in the video stream  
    **if** Counter is equal to frame down-sampling rate *then*  
        **Depth Map**  $\leftarrow$  downsample the depth map  
        **T**  $\leftarrow$  estimated camera pose by the depth map  
        Insert **Depth Map** and **T** into the Octree  
        Reset the Counter  
    **End**  
    Counter++  
**end**  
**Output** Octree

---

## Chapter 5: System design

In this chapter, CPU-based Octomap is implemented on top of GPU-based KinectFusion by multithreading with image correction thread. The resulting system runs in real time.

### 5.1 Image correction

Another challenge for real-time implementation is to correct the depth and RGB images for lens distortion. As the distortion correction is well studied [12], it is easy to initialize the distortion rectify map both for the depth and RGB image with the OpenCV library. However, this image correction process takes about 0.04s by OpenCV on CPU, which is slower than the 30 Fps of the Kinect camera.

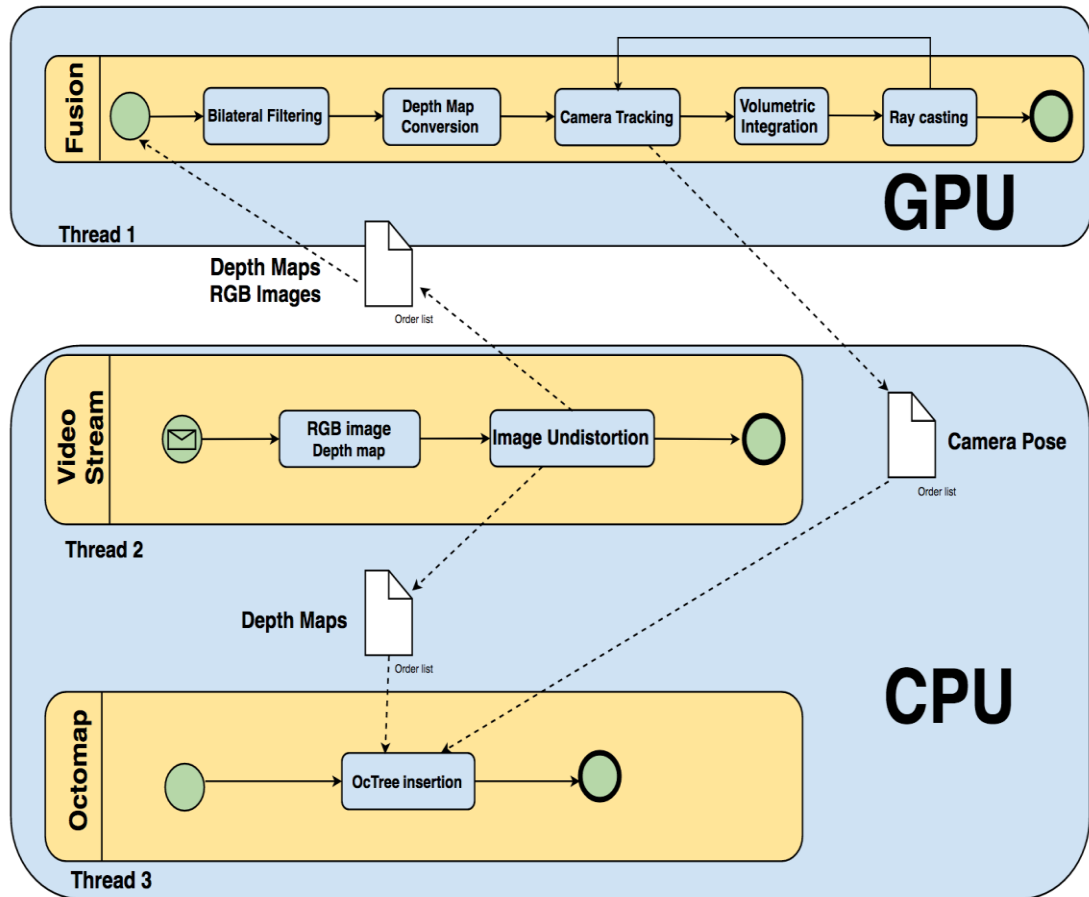
### 5.2 Multithreading

In our work, in order to solve the slow process of image correction, we combine the GPU-based pipeline of KinectFusion with the CPU-based Octomap and image correction by multithreading. The system involves 3 threads, one main thread for image input and correction, another thread for KinectFusion, and the other thread for Octomap.

The main thread inputs and corrects both of the depth and RGB images, and then corrected images are saved in a queue structure. Once the image queue is not empty, KinectFusion keeps fusing the front image of the queue until the queue is empty. Octomap thread detects for both the depth image and its corresponding

camera pose, and uses a counter to down-sample the depth frames. Octomap thread keeps inserting into the Octree on CPU until the depth-camera queue is empty.

The resulting system described by Figure 4.1 runs in real time and can be used as a useful input to the symmetry detection algorithm introduced by [1].



- Thread 1: GPU-based pipelines of KinectFusion
- Thread 2: Input depth and RGB images and correction
- Thread 3: Occupancy mapping algorithm, Octomap

Figure 4.2: System for real-time 3D surface reconstruction

## Chapter 6: Surface Extraction

In this chapter, we provide two GPU-based methods to extract the 3D surface reconstruction of point clouds.

### 6.1 Marching cubes

While in theory every point needs to be rendered, this is not possible in practice. However, the Marching cubes algorithm creates a triangle representation on the surface of a 3D point cloud, thus can make a good approximation to the surface of the point cloud model by triangular mesh [12].

Based on the open source GPU-based Marching cubes implementation by [13], for each 3D vertex generated by the KinectFusion, we need to find those vertices where the surface intersects the voxel. Then we will generate the triangles to represent the surface of the reconstruction object.

For CUDA optimization, the tables for enumerating the Marching Cubes situations can be bind with texture memory for quick look-up. The pseudo code below describes our GPU-based Marching cubes algorithm.

---

**Algorithm 6** Marching cubes

---

```
edgeTable ← 8-bit flag representing which cube vertices are inside
triangleTable ← map same cube vertex index to a list of 5 triangles
numVertsTable ← number of vertices for the triangleTable
for each point v parallel do
    p ← perspective project vertex v
    p[8] ← calculate cell vertex positions
    find the vertices where the surface intersects the voxel by interpolation
    output triangle vertices
end
```

---

## 6.2 Point cloud down sampling

The KinectFusion reconstructs a point cloud to represent the object surfaces in the scene. However, the number of vertex generated by KinectFusion can be up to millions, which significantly slows down the symmetry detection. Thus, in order to decrease the number of point clouds and approximate the surface, down sampling of a point cloud is necessary to be implemented by GPU. Similar with average down-sampling method, by dividing the 3D volume into small voxels, all of the vertices inside the same voxel will be approximated with their centroid, thus down sample the point cloud.

This down sampling method copies the point cloud buffer and runs independently for visualization and exporting.

---

### Algorithm 7 Downsampling a Point Cloud

---

Pre-defined down-sample size as **scale**  
**while** the 3D point  $(x,y,z)$  inside the voxel **parallel do**  
    **for** 3D point  $(x + dx, y + dy, z + dz)$  within down-sample cube  
         $F \leftarrow$  Fetch the current TSDF at  $(x + dx, y + dy, z + dz)$   
         $F_x \leftarrow$  Fetch the next TSDF at  $(x + dx + 1, y + dy, z + dz)$   
        **If**  $F$  and  $F_x$  are different signs,  
            Found surface vertex in x direction  
         $F_y \leftarrow$  Fetch the next TSDF at  $(x + dx, y + dy + 1, z + dz)$   
        **If**  $F$  and  $F_y$  are different signs,  
            Found surface vertex in y direction  
         $F_z \leftarrow$  Fetch the next TSDF at  $(x + dx, y + dy, z + dz + 1)$   
        **If**  $F$  and  $F_z$  are different signs,  
            Found surface vertex in z direction  
    **end**  
    Calculate the **mean** for these vertices  
    Use this centroid to represent the vertex in the down-sample cube  
     $x \leftarrow x + \text{scale}$   
     $y \leftarrow y + \text{scale}$   
     $z \leftarrow z + \text{scale}$   
**end**

---

## Chapter 7: Experiments and results

We simulate our real time surface reconstruction by three 30 fps Kinect video streams as input. Our program is implemented in CUDA and C++. All experiments were run on a machine with one Intel i5-6600K CPU clocked at 3.5GHz. The Octomap runs as a parallel CPU thread, while the KinectFusion is implemented with Nvidia's CUDA framework, which utilizes the computer's dedicated GPU (One Nvidia GeForce GTX Titan X). Additionally we use an extra CPU thread for reading and correcting the input images. Then we use Meshlab to visualize the point cloud and mesh.

### 7.1 Initialization

All of the Kinect and Octomap parameters are initialized as listed in Appendices. The surface reconstruction is done within the 3D volume of length 1.28m and with voxel resolution 0.0025m, and the cursor indicates the camera initial position.

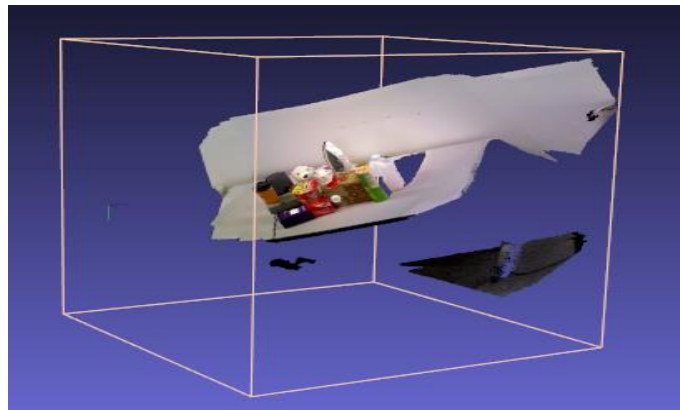
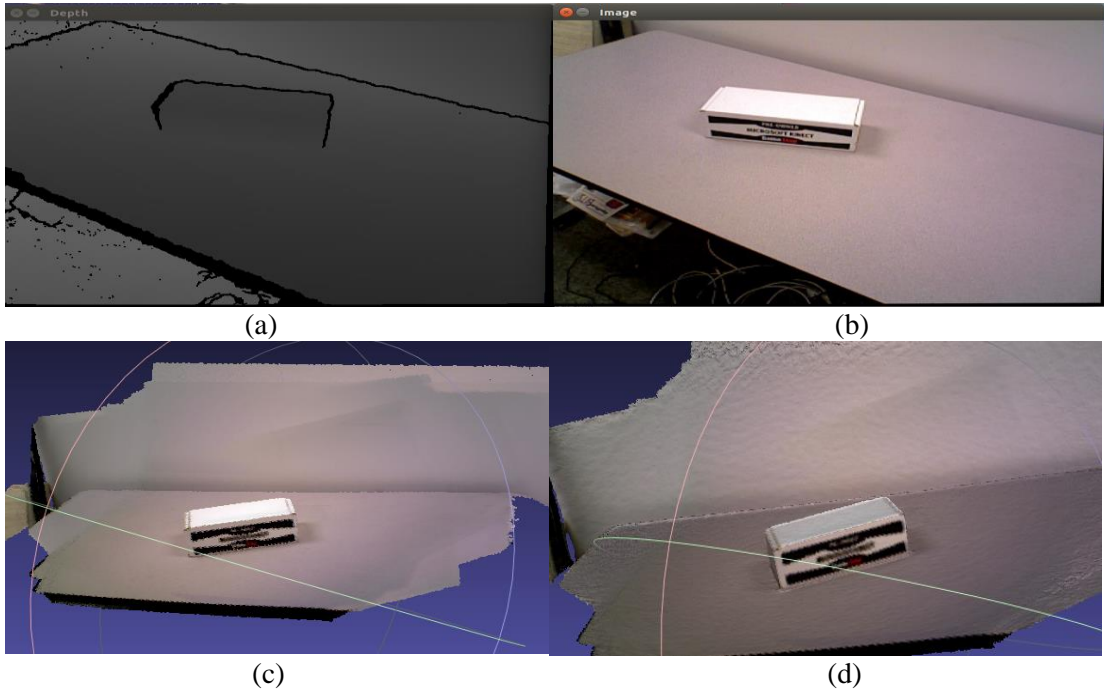


Figure 6.1: Volume and camera initialization

## 7.2 Surface reconstruction



(a) Depth map (b) RGB image (c) Point cloud (d) Mesh  
Figure 6.2: Surface reconstruction from video A

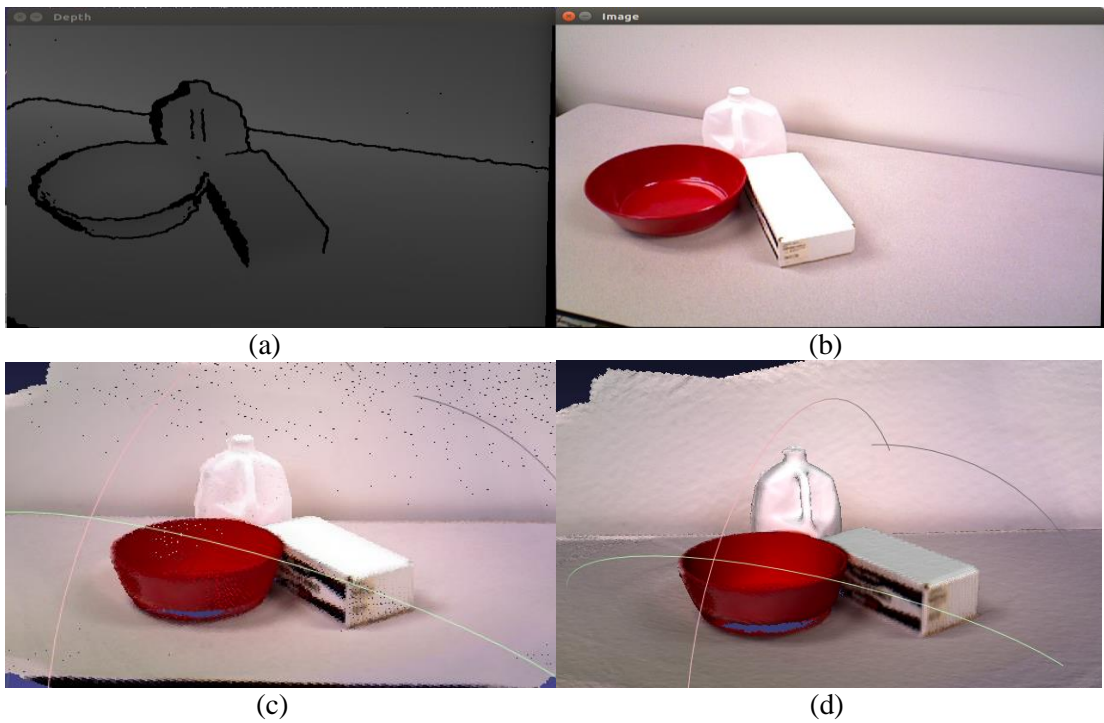
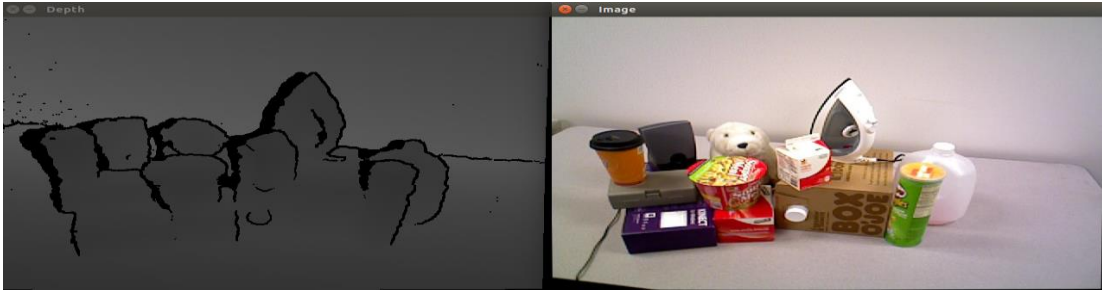


Figure 6.3: Surface reconstruction from video B



(a)

(b)



(c)



(d)

(a) Depth map (b) RGB image (c) Point cloud (d) Mesh  
Figure 6.4: Surface reconstruction from video C

### 7.3 Octomap

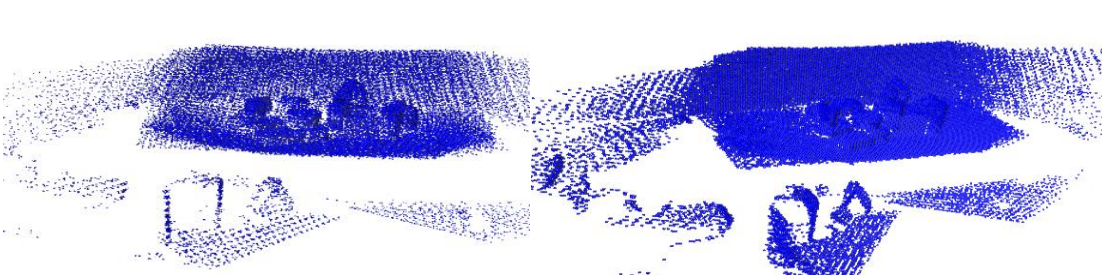


Figure 6.5 Octomap from from video C, resolution 0.006m / 0.01m

### 7.4 Down sampling



Figure 6.6: Point cloud down sampled by 4, 298101 vertices / 92785 vertices



## Conclusion

In this thesis, we present a real time system for 3D surface reconstruction for symmetric objects. This is achieved by adding a CPU-based occupancy mapping step on top of the GPU-based reconstruction algorithm KinectFusion. Moreover, in order to increase the capabilities to process the resulting point clouds, we develop the GPU-based Marching cubes algorithm to take the mesh, and GPU-based down sampling to decrease the point clouds' size, thus speed up the symmetric object detection introduced by [1].

Our experiments demonstrate a fast and accurate colored 3D surface reconstruction by point clouds, mesh and Octomap, which can be a useful input to the symmetric object detection algorithm introduced by [1]. Also our surface reconstruction implementation only depends on CUDA and OpenCV, which simplify the dependency installation and compiler process.

In its current state, our system requires multithreading to implement real-time image correction and Octomap based on CPU. Moreover, only 37 per cent of GPU memory is used even for reconstruction of large point clouds, such as video C. A further idea for the optimization is to implement these two CPU steps on GPU, in order to save the cost of transmitting the data between CPU and GPU and maximize the GPU memory efficiency.

## Appendices

Volume Initialization	Volume size	1.28 meters
	number of voxels	512
Depth Pyramid	Pyramid levels	4
Bilateral Filtering	bilateral kernel size	5
	depth parameter for bilateral filter	0.001 meter
	spatial parameter for bilateral filter	3 pixels
ICP	Camera Position	(-0.64,-0.64,0.2)
	Camera Pose T	Initialized with identity matrix
	ICP iteration time for each pyramid level	{3,3,3,3}
	ICP distance threshold	0.1 meter
	ICP angle threshold	30 degree
TSDF Integration	TSDF truncation distance	0.001 meter
	TSDF max weight (without color)	255
	TSDF max weight (with color)	255
Ray Casting	ray cast step factor	0.75 voxel size
	gradient_delta_factor	0.75 voxel size
Octomap	Frame down-sample rate	15
	Resolution	0.01 meter

Table A: Parameters for KinectFusion and Octomap

## Bibliography

- [1] Ecins, Aleksandrs, Cornelia Fermüller, and Yiannis Aloimonos. "Cluttered scene segmentation using the symmetry constraint." *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016.
- [2] Izadi, Shahram, et al. "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera." *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011.
- [3] <http://www.cse.psu.edu/~rtc12/CSE486/lecture12.pdf>
- [4] Mayer, Nikolaus. "Coupling ICP and Whole Image Alignment for Real-time Camera Tracking." (2014).
- [5] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald, "Kintinuous: Spatially extended KinectFusion," in *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, Sydney, Australia, Jul 2012.
- [6] T. Whelan, H. Johannsson, M. Kaess, J. J. Leonard, and J. McDonald, "Robust real-time visual odometry for dense rgb-d mapping," in *ICRA*, 2013, pp. 5724– 5731.
- [7] N. Schwabsky, V. Cohen. "Open Fusion, Real-time 3D Surface Reconstruction Out of Depth Images." (2013).
- [8] R. A. Newcombe, S. Lovegrove, and A. J. Davison, "Dtam: Dense tracking and mapping in real-time," in *ICCV*, 2011, pp. 2320–2327.
- [9] Y. Chen and G. Medioni. Object modeling by registration of multiple range images. *Image and Vision Computing (IVC)*, 10(3):145–155, 1992.
- [10] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. Technical Report TR04-004, Department of Computer Science, University of North Carolina at Chapel Hill, 2004.
- [11] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems," in *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, vol. 2, 2010.
- [12] Lorensen, William E., and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm." *ACM siggraph computer graphics*. Vol. 21. No. 4. ACM, 1987.
- [13] [https://github.com/Nerei/kinfu\\_remake](https://github.com/Nerei/kinfu_remake)