Project Report

Open Fusion
Real-time 3D Surface Reconstruction Out of Depth Images

Nurit Schwabsky                Vered Cohen

January 2013

Supervisors
Prof. Chaim Gotsman
Mr. Aaron Wetzler

Technion - Israel Institute of Technology
Computer Science Department
Computer Science Department

Geometric Image Processing Lab

Center for Graphics and Geometric Computing
Special thanks to Aaron Wetzler
for his endless help, patience, and support
# Contents

Abstract 1

1 Introduction 3
   1.1 Overview .................................................. 3
   1.2 General Mathematical Background .............................. 5

2 Depth Map Input and Conversion 9
   2.1 Overview .................................................. 9
   2.2 Depth Conversion ............................................ 9
   2.3 Changes from the KinectFusion .............................. 10

3 Camera Tracking 11
   3.1 Overview .................................................. 11
   3.2 Iterative Closest Point ..................................... 11
   3.3 Find Corresponding Points .................................. 12
   3.4 Calculating the Incremental Transform ...................... 13
   3.5 Changes from the KinectFusion .............................. 14

4 Volumetric Integration 15
   4.1 Overview .................................................. 15
   4.2 Volume Integration .......................................... 15
   4.3 Optimizations ............................................... 17
   4.4 Changes from the KinectFusion .............................. 17

5 Raycasting 19
   5.1 Overview .................................................. 19
   5.2 Raycasting ................................................ 19

6 Project Summary 23
   6.1 Problems Along the Way .................................... 23
      6.1.1 Working with Cuda and Matlab ......................... 23
      6.1.2 Difficulty Debugging .................................. 24
      6.1.3 Inaccuracies in the KinectFusion Paper ............... 24
      6.1.4 Modularity and Portability ............................ 24
   6.2 Summary .................................................. 25
   6.3 Future Work ............................................... 27
Abstract

OpenFusion is an implementation of Microsoft’s KinectFusion system. This system enables real-time tracking and reconstruction of a 3D scene using a depth sensor. A stream of depth images is received from the camera and compared to the model built so far using the Iterative Closest Point (ICP) algorithm to track the 6DOF camera position. The camera position is then used to integrate the new depth images into the growing volumetric model, resulting in an accurate and robust reconstruction. The reconstructed model is adapted according to dynamic changes in the scene without losing accuracy. The system is implemented mainly on the GPU, enabling fast parallel computations of vast amounts of data. Microsoft’s KinectFusion was a breakthrough in real-time 3D reconstruction and interaction using a moving depth camera.

Our system supports various depth cameras, including Microsoft Kinect, Asus Xtion-Pro, SoftKinetic and the novel GIP camera that was developed in the GIP lab at the Technion, Israel.

Fig. 0.1: OpenFusion System. A) The rendered model. B) Raw normal map. C) Raw depth map. D) RGB image of scene being scanned.
1 Introduction

1.1 Overview

OpenFusion enables the user to build a 3D model of an indoor scene in real-time using a depth camera. The system receives a constant stream of depth maps from the camera, which it uses to track the 6DOF camera pose and update the model accordingly. Depth data received from the camera is noisy and often full of holes (areas where no depth data is available), thus many different viewpoints are necessary in order to fill in gaps and smooth away noisy signals. As the user moves around the scene, new views of the scanned objects are discovered and integrated, resulting in a complete high-quality model. Though OpenFusion builds a static model, a small amount of dynamic interaction with the scene is possible; over time the system will “fix” the model using the new data received.

Fig. 1.1: A) The model after the first frame (built from one noisy depth map). B) The model after many frames - much smoother with fewer holes. C) Zoom in on the view in B - The voxel composition of the model.

Our system can model scenes within an area of two cubic meters. The model maintained by OpenFusion is stored in a voxel grid as a signed distance function, in which the zero-crossing defines the model’s surface. The voxel composition of the model can be seen in Figure 1.1 in frames B) and C). The volumetric representation of the model can be exported as a mesh using the marching cubes algorithm.

OpenFusion is built as a pipeline of four stages, each of which is computed in parallel on the GPU. The four main stages of the pipeline are:
Chapter 1

Introduction

Fig. 1.2: Gpu Pipeline

- **Depth Map Conversion** – depth data is received from the camera and converted from image coordinates to vertices and normals in the world coordinate system.

- **Camera Tracking** – a rigid 6DOF transformation is computed to align the new depth map with the rendered existing model as seen in the previous frame. This is done using the Iterative Closest Point (ICP) algorithm. The output of the algorithm effectively gives us the relative transformation between two subsequent frames, which is then incrementally applied to a single transformation defining the camera position within the volume.

- **Volumetric Integration** – points from the received depth map are calculated in the world coordinate system using the transformation from the previous stage. These points are then used to update the voxel grid.

- **Raycasting** – the volume is raycast to extract the view of the implicit surface. This view is used both for rendering to the user and for calculating a synthetic depth map which is used as a less noisy reference frame for the next iteration of ICP. Thus camera tracking is based on the refined model built as a result of all previous frames and on the new depth map, instead of using only subsequent noisy depth frames.

The OpenFusion system is based on Microsoft’s KinectFusion algorithm, as described in [1] and [2]. Algorithms used for 3D modeling based on a stream of depth maps have been studied in the past; however existing systems offered only offline reconstructions, real-time but non-interactive rates, or real time camera tracking without real time reconstruction. Using the novel GPU pipeline described above, KinectFusion was the first system to achieve real time rates for both camera tracking and 3D reconstruction. This speed enables KinectFusion to permit dynamic interaction with the reconstructed scene. OpenFusion does not reach the high frame-per-second count of the KinectFusion, and contains some deviations from the original algorithm which will be explained in depth below.
To allow for modularity, OpenFusion can be configured to receive depth images from several different inputs, such as the Asus Xtion Pro, Microsoft’s Kinect, and the GIP Lab camera. A simple interface written into the code enables the user to add a module handling any additional accurate depth sensor they wish to work with. The system has many additional features, such as the ability to record a stream of depth data and to run the reconstruction algorithm on previously recorded data. OpenFusion can also be used to export the reconstructed model as a three-dimensional signed distance function from which a mesh object can be constructed using the marching cubes algorithm. This object can then be used by any 3D modeling program.

Both systems have many diverse applications, which include low-cost object scanning, advanced Augmented Reality, and physics-based interactions. Additionally, it can be used for occlusion handling, segmenting, tracking and reconstructing dynamic users and the background scene simultaneously. “Augmented Reality” refers to interactions between the real world presentation and a 3D virtual world, which is overlaid onto it. In the KinectFusion paper Microsoft demonstrated this ability by simulating physical interaction of virtual objects with a rendered real-world scene.

Amongst the multitude of fields to which this system can be applied are medical systems, robotics, gaming, along with other media, in particular CGI (Computer-Generated Imagery).

### 1.2 General Mathematical Background

This report will use the following definitions:

- **Global Space** – A 3D static world coordinate system, within which the 3D model is maintained. All calculations pertaining to the model within the voxel volume, such as normal calculation and lighting, are computed in Global Space. Global Space coordinates in the OpenFusion system are measured in millimeters, with the voxel volume centered at \((0, 0, 0)\) and extending one meter along each axis.

- **Camera Space** – A 3D coordinate system affiliated with the camera. The camera is placed at \((0, 0, 0)\) within this coordinate system and faces toward the negative Z axis. The camera’s coordinate system moves within the confines of the Global Space. The position of the camera within the Global space is defined by the worldview matrix.

- **Image Space** – A 2D coordinate system which comprises the pixels in the screen. Each pixel holds data pertaining to the point on the model visible through that pixel.

The depth data received from the camera is in Image Space, while the model itself and thus the vertexes of the raycasted depth map are in Global Space. Transformations between the different coordinate systems are often necessary in the process.
of the various GPU pipeline stages in order to compare sets of data from different coordinate systems.

Image to Camera Space: Let $K$ be the projection matrix associated with the camera being used. The projection matrix allows perspective projection from pixels to Camera Space vertexes, using the depth defined for each pixel. The projection matrix is defined by the camera’s view frustum, and thus is a static value associated with the camera.

\[
K = \begin{bmatrix}
-f_x & 0 & c_x \\
0 & -f_y & c_y \\
0 & 0 & 1
\end{bmatrix}
\]  

(1.1)

The $K$ matrix is described in Equation 1.1 with the following definitions:

- $(c_x, c_y)$: the center of the image (in our case $(320, 240)$)
- $f_x$ and $f_y$ are the focal lengths expressed in pixel related units. The focal lengths are calculated using the camera’s field of view (FOV) angles along the $x$ and $y$ axes. These angles can be found within the camera specifications.

\[
f_x = \frac{-\text{ImageWidth}}{2 \cdot \tan\left(\frac{\text{fov}_x}{2}\right) \cdot \frac{\pi}{180}}
\]

(1.2)

\[
f_y = \frac{-\text{ImageHeight}}{2 \cdot \tan\left(\frac{\text{fov}_y}{2}\right) \cdot \frac{\pi}{180}}
\]

Given a pixel $u = (x, y)$ in Image Space with a depth of $D(u)$ the corresponding Camera Space vertex $v$ can be found by: $v(u) = D(u) \cdot K^{-1}[u, 1]$.

Camera to Global Space: Let $T_i$ be the $4 \times 4$ modelview matrix at time $i$. The modelview matrix describes the 6DOF camera pose within the global coordinate
1.2 General Mathematical Background

system. $T_i$ is built of the following:

$$
\begin{bmatrix}
R_i & t_i \\
0 & 1
\end{bmatrix} = \begin{bmatrix}
r_{0,0} & r_{0,1} & r_{0,2} & t_x \\
r_{1,0} & r_{1,1} & r_{1,2} & t_y \\
r_{2,0} & r_{2,1} & r_{2,2} & t_z \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

(1.3)

wherein $R_i$ is a 3x3 rotation matrix and $t_i$ is a vector describing the placement of the camera within the global coordinate system.

Given a vertex $v$ in Camera Space, the vertex $v_g$ in Global Space is found using the following transformation: $v = T_i \cdot v_g$. 
2 Depth Map Input and Conversion

2.1 Overview

In this stage of the pipeline, depth data is received from the camera and converted from image coordinates to vertices and normals in the world coordinate system. We designed our system in a modular manner so it can use any depth camera available today in the market. By implementing a few simple methods to read the stream from the camera, any depth camera can be plugged to our system.

2.2 Depth Conversion

The depth map (Figure 2.1) is retrieved from the camera using a camera-specific interface module. Each depth frame is copied into the GPU where the corresponding Global Space vertex map, $v_{g,i}(x, y)$ is calculated in parallel. A default “bad vertex” is assigned to each pixel $(x, y)$ for which depth data does not exist. Additionally, the “bad vertex” value is assigned to each pixel whose depth is not within the reliable depth values stated in the camera specifications.

Fig. 2.1: Raw depth map. Lighter areas are farther away from the camera and black spots are patches of missing data.
Once the Global coordinates for the new depth map have been found, our system calculates the corresponding normal vectors for each vertex, again in parallel on the GPU. This calculation relies on the assumption that neighboring pixels define neighboring vertices within the scene. While this assumption is often incorrect, it allows us to build a rough approximation of the normal map (Figure 2.2, A) for the new depth, which is later used for comparisons in the camera tracking stage. A more exact normal map (Figure 2.2, B) is later calculated straight from the volumetric representation of the model.

The normal map is calculated using the following equation:

\[
ng,i(x, y) = (vg,i(x + 1, y) - vg,i(x, y)) \times (vg,i(x, y + 1) - vg,i(x, y))
\]

after which the value is normalized. If one of the vertices used in the equation holds the “bad vertex” value the normal is set to (0, 0, 0).

2.3 Changes from the KinectFusion

KinectFusion uses bilateral filtering to smooth away outlying noise signals from the depth map received from the camera. OpenFusion uses the depth data as is, and so relies on data which is less reliable. The result is not as stable as the KinectFusion, and should be fixed in future iterations of the project. Bilateral filtering was not added as a result of time constraints on the project. As with other heavy mathematical calculations in the project, bilateral filtering can be implemented in parallel on the GPU.
3 Camera Tracking

3.1 Overview

In this stage of the pipeline we compute a rigid 6DOF transformation in order to align the new depth map with the rendered existing model as seen in the previous frame. This is done using the Iterative Closest Point (ICP) algorithm. The output of the algorithm effectively gives us the relative transformation between two subsequent frames, which is then incrementally applied to a single transformation defining the camera position within the volume.

3.2 Iterative Closest Point

In order to correctly fuse a stream of consecutive depth maps into a single 3D model we must be able to identify the 6DOF camera pose at each stage during the scanning process. This is calculated using the Iterative Closest Point (ICP) algorithm. ICP is a well-known algorithm for 3D shape alignment. Given source and destination point clouds, ICP decides on a mapping between points in the source cloud to corresponding points in the destination, and estimates the rigid transformation needed to align the source cloud to the destination. This transformation is improved over the course of several iterations, until a close enough match has been found.

Fig. 3.1: Point to plane ICP. Example of two iterations.

In OpenFusion we use a variant of ICP called point-to-plane ICP (Figure 3.1), in
which the objective is to minimize the sum of the squared distances between each source point and the tangent plane at the corresponding destination point:

- Source point \( s_i = (s_{ix}, s_{iy}, s_{iz}) \)
- Destination point \( d_i = (d_{ix}, d_{iy}, d_{iz}) \)
- Surface normal at \( d_i \): \( n_i = (n_{ix}, n_{iy}, n_{iz}) \)

The goal of ICP is to find a transformation \( T \) such that

\[
\sum_i \left( (T \cdot s_i - d_i) \cdot n_i \right)^2
\]

(3.1)

is minimized. The source vertices in our case are the global vertices calculated from the new depth map in the Depth Map Conversion stage of the pipeline, and the destination vertices are the global vertices raycasted from the model for the previous frame. The result is a relative 6DOF transformation matrix that can be incrementally applied to the current transformation to give the new 6DOF pose of the camera. As the destination vertices are extracted each frame from the increasingly stable global model we have a strong base for comparisons between point clouds, allowing more exact and efficient ICP calculations.

### 3.3 Find Corresponding Points

The first step of the ICP algorithm is to find corresponding points between the previous frame (the raycasted points at time \( i - 1 \)) and the new frame (calculated during Depth Map Conversion at time \( i \)). The correspondences are found using projective data association. A possible corresponding point is found for each pixel in parallel on the GPU. The calculations for finding correspondences are based on \( T_i \), the transformation matrix calculated in the previous iteration of ICP. \( T_i \) is initialized as the 6DOF pose of the camera from the previous frame.

For each pixel with valid depth data in the frame we extract the raycasted global vertex \( v_{g,i-1} \) and the raycasted normal \( n_{g,i-1} \) from the raycasted vertex map (Figure 3.2 A). If this vertex is valid (does not contain a “Bad Vertex” value) we convert it to Camera Space and then project it into Image Space in order to find the pixel \( p \) associated with the vertex (Figure 3.2 B).

The next step is to test the global vertex in the new depth map at pixel \( p \) (Figure 3.2 C). The basic assumption is that the camera moves very slightly between frames, and so the corresponding points will be found in similar pixels in consecutive frames. If \( p \) is within the current projected view frustum then the Global Space vertex \( v_{g,i}(p) \) and normal \( n_{g,i}(p) \) associated with the new depth map at pixel \( p \) are extracted (Figure 3.2 D). If the distance between \( v_{g,i} \) and \( v_{g,i-1} \) is below a distance threshold (25mm in our system) and the dot product of \( n_i \) and \( n_{i-1} \) is above a normal threshold (0.65 in our system) then we declare \( v_{g,i} \) and \( v_{g,i-1} \) to be corresponding points.
3.4 Calculating the Incremental Transform

Once all corresponding points have been found we can calculate the resulting rigid transformation matrix, by solving Equation 3.1. The transformation can be written as $T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$, where $R$ is a nonlinear rotation matrix with angles $\alpha$, $\beta$ and $\gamma$, and $t$ is a translation vector $(t_x, t_y, t_z)$. Our objective in this stage is to find these six parameters, which will enable us to build the actual transformation matrix. As shown in [3], we solve a linear least-squares approximation of the nonlinear optimization problem, based on the assumption that movements between frames are minimal. It is well-known that a linear system can be solved more efficiently than a nonlinear one. When an angle $\theta$ is close to zero then $\sin \theta$ is approximately 0 and $\cos \theta$ is approximately 1. Therefore the following linear approximation of $R$ is
obtained:

\[
R(\alpha, \beta, \gamma) \approx \begin{bmatrix}
1 & \alpha \beta - \gamma & \alpha \gamma + \beta & 0 \\
\gamma & \alpha \beta \gamma + 1 & \beta \gamma - \alpha & 0 \\
-\beta & \alpha & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\approx \begin{bmatrix}
1 & -\gamma & \beta & 0 \\
\gamma & 1 & -\alpha & 0 \\
-\beta & \alpha & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

(3.2)

The 6x6 linear least-squares system is computed in parallel on the GPU using a tree reduction. This computation can be optimized by utilizing the shared memory of the GPU. The resulting linear system is then solved on the CPU using the Cholesky decomposition as implemented in the Eigen library.

Finally we incrementally apply the resulting rigid transformation matrix to the current camera transformation. This yields the new position of the camera, to be used in the next stages of the pipeline.

### 3.5 Changes from the KinectFusion

KinectFusion uses a coarse-to-fine approach to finding the new transformation matrix. Three resolutions of the input data are used: the first is a coarse approximation of the input data, the second is of a higher resolution, and the third uses all pixels in the depth map to refine the transformation calculated in the previous stages. This allows a faster convergence to the required transformation matrix and smooths over problems that might otherwise arise as a result of local minimum values. This adds stability to the calculation, especially when there are sudden movements of the camera. Coarse-To-Fine ICP calculation was not added as a result of time constraints on the project, and should be added in future iterations.
4 Volumetric Integration

4.1 Overview

In this stage of the pipeline, points from the received depth map are calculated in the world coordinate system using the transformation from the previous stage. These points are then used to fuse the new data with the existing volumetric model, adding new data and refining the old.

4.2 Volume Integration

The ICP algorithm gives us the global pose of the camera after each frame in the depth video stream. This allows us to integrate consecutive depth maps into a single model in global space. The model is built within a volumetric representation of the Global Space, as described in the introduction, which is subdivided uniformly into a 3D grid of $512^3$ voxels. The surface is represented by a truncated signed distance function. Each voxel in the volume holds a value specifying its distance to the actual surface (the TSDF value) and the weight of the voxel. The TSDF values are positive in front of the surface (outside of the model) and negative behind the surface (inside the model). The zero-crossing defines the surface of the model itself. The value of each voxel is updated as new depth data is received from the camera using a running weighted average which puts more emphasis on the existing value than on the new data received. New depth data is only integrated after a sufficient amount of time, allowing us to determine that the new data is indeed part of the model and not merely noise resulting from the camera’s inaccuracies.

To integrate depth data into the volume our system sweeps the entire volume as follows:

Each $YZ$ slice of the volume is calculated in parallel, sweeping from the front slice to the back. Because of the enormous size of the voxel volume it is not feasible to attach a thread to each voxel. Thus, each $YZ$ value is assigned to a thread, and all of the threads iterate over 512 possible $x$ values. Each voxel is converted to a Global Space vertex (the vertex coinciding with the center of the voxel). The vertex is then transformed to Camera Space using the current transformation matrix as calculated by ICP, and then projected into Image Space to find the pixel $u$ through which a ray from the camera to the vertex must pass.
If the pixel found is in the current camera projected view frustum and contains valid depth data, then the signed distance function is calculated as the difference between the Global Space distance of the voxel from the camera and the distance of the surface vertex seen through \( u \) from the camera:

\[
SDF = \| s - c \| - \| v - c \|
\]  
(4.1)

where \( s \) is the surface vertex, \( v \) is the voxel vertex, and \( c \) is the camera position.

This equation gives us the distance of the voxel from the surface, which is positive if the voxel is in front of the surface and negative if it is behind.

Fig. 4.1: Signed distance function. Values near the surface are closer to zero.

<table>
<thead>
<tr>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>1.6</td>
<td>1.0</td>
<td>0.6</td>
<td>-1.1</td>
</tr>
<tr>
<td>2.0</td>
<td>1.4</td>
<td>0.4</td>
<td>-0.8</td>
<td>-1.7</td>
</tr>
<tr>
<td>1.8</td>
<td>1.3</td>
<td>0.1</td>
<td>-1.2</td>
<td>-2.3</td>
</tr>
<tr>
<td>1.1</td>
<td>0.6</td>
<td>-0.9</td>
<td>-2.1</td>
<td>-3.1</td>
</tr>
</tbody>
</table>

The SDF value is then truncated to a pre-specified truncation distance (in our system 30mm) on either side of the surface, giving us the new candidate for the TSDF value of the voxel. A weighted average of the new TSDF value and the previous TSDF value of the voxel is then calculated, such that the old value has a higher impact than the new. The new TSDF value is given a weight of 1, while the previous TSDF value receives the previous weight value of the voxel (an integer greater than zero), as described in Equation 4.2. This way we ensure that the voxel data is updated only when significant depth data is received, and the value is not swayed by noise received from the camera. Finally, the voxel’s weight is incremented, as shown in Equation 4.3.

\[
TSDF_{avg} = \frac{TSDF_{i-1} \cdot w_{i-1} + TSDF_i \cdot 1}{w_{i-1} + 1}
\]  
(4.2)

\[
w_i = \min(Max\_Weight, w_{i-1} + 1)
\]  
(4.3)
4.3 Optimizations

The voxel volume as a whole takes up about 640MB of GPU memory, far too much to be held in cache. We would like to minimize the number of cache misses as much as possible, as accessing the global memory of the GPU is a very slow process. However, minimizing cache misses is difficult because all threads are run in parallel, and may access different regions of the memory at the same time. Thus, the line of code currently being executed by all threads in the warp must utilize as much of the current cache line as possible. The voxel volume is stored as a 1D array in the global GPU memory. In order to ensure optimal cache utilization we must access continuous segments of the array at the same time, meaning that all threads running in parallel should have similar index values corresponding to the second and third dimensions of the array. For example, in our system we access voxel \((x, y, z)\) by the formula \((x \cdot \text{VOXEL}_\text{SIZE}^2 + y \cdot \text{VOXEL}_\text{SIZE} + z)\). Therefore, the most efficient way to execute a block of threads is by guaranteeing that they have the same \(y\) and \(z\) values, as they are continuous in the memory and will map to the same cache line.

4.4 Changes from the KinectFusion

KinectFusion uses a different calculation for the weight associated with the value in each voxel. Their calculation:

\[
TSDF_{avg} = \frac{TSDF_{i-1} \cdot w_{i-1} + TSDF_i \cdot w_i}{w_{i-1} + w_i}
\] (4.4)

\[
w_i = \min(\text{Max}_\text{Weight}, w_{i-1} + 1)
\] (4.5)

As can be seen from the calculations, KinectFusion gives an equal weight to the previous and the current TSDF value in the voxel. This allows swifter and more accurate responses to dynamic changes within the scene, as the new data will be integrated sooner. The OpenFusion implementation lends far more weight to the existing value. The result of this change is that it takes longer for new changes to be accepted, especially when the camera has been in the same position for a lengthy amount of time, increasing the weight in the viewed voxels. Our results were less steady using Equation 4.4; this was solved by replacing them with Equation 4.2.
5 Raycasting

5.1 Overview
In this stage of the pipeline we raycast the volume to extract the current view of the implicit surface. This view is used both for rendering to the user and for calculating a synthetic depth map which is used as a less noisy reference frame for the next iteration of ICP. The synthetic depth map is based on the refined model built as a result of all previous frames, and so is more accurate than the noisy depth frames received from the camera.

5.2 Raycasting
The model is rendered from the currently calculated 6DOF pose of the camera using volumetric raycasting. Rays are cast in parallel on the GPU for each pixel in the output image. Given the position of the camera and the direction of the ray (defined by the pixel) the GPU threads traverse the volume and discover the position of the implicit surface of the model by looking for zero-crossings amongst the TSDF values, as discussed in chapter 4.

Fig. 5.1: Raycasting.
This stage has two outputs:

- The rendered global vertex and normal maps – when the implicit surface is discovered, each thread calculates the ray’s global intersection point with the surface using trilinear interpolation of neighboring TSDF values. The surface normal at this point is computed as a derivative of the TSDF values in the vicinity.
- The output image – using the vertex and normal data calculated by each thread, a lighting equation is computed for each pixel in the output image, thus rendering the current view of the model.

For each pixel $u$ in the output image the direction of the casted ray is calculated. To do this we first find the global space coordinates $v$ corresponding to pixel $u$ at a depth of 1mm from the camera. The direction of the ray is the direction of the vector between the camera position and $v$ (Figure 5.2). The origin of the ray is the camera position.

![Fig. 5.2: Ray direction calculation.](image)

For each ray that intersects the volume the nearest and farthest intersection points are calculated. We march along the ray starting at the near intersection using steps of a predefined size. At each step the current global position of the ray is translated to a voxel within the volume and the TSDF value is checked. At each step we update a variable holding the previously visited voxel.

As was mentioned in section 4.2, the TSDF values are positive in front of the surface and negative behind. Thus, if a negative TSDF value is found then a zero-crossing must have been passed. Using the previously visited voxel the implicit surface vertex is calculated using trilinear interpolation, as shown in [4]. Note that the previously visited voxel has a positive TSDF value; otherwise we would have stopped before...
reaching the current voxel.

In addition to the implicit surface vertex, we also calculate the normal to the surface at that point. Let \( \text{Vox}(x, y, z) \) be the voxel containing the surface crossing, then the surface normal \( n \) at this point is calculated as follows:

\[
\begin{align*}
    n_x &= TSDF (\text{Vox}(x + 1, y, z)) - TSDF (\text{Vox}(x - 1, y, z)) \\
    n_y &= TSDF (\text{Vox}(x, y + 1, z)) - TSDF (\text{Vox}(x, y - 1, z)) \\
    n_z &= TSDF (\text{Vox}(x, y, z + 1)) - TSDF (\text{Vox}(x, y, z - 1))
\end{align*}
\] (5.1)

The resulting vector is then normalized.
6 Project Summary

6.1 Problems Along the Way

The project was very challenging, with a wide scope covering many aspects of computer graphics and vision. Our first exposure to the KinectFusion project was through the demonstration video released by the Microsoft team. We were fascinated, and though we only had a basic background from related courses such as “Computer Graphics” and “Computational Geometry” we knew we had to take this project. Over the course of the project, through long days, nights and weekends in the lab, we overcame many obstacles, resulting in a fully functional system.

Some of the more difficult obstacles we encountered are described below.

6.1.1 Working with Cuda and Matlab

Before starting the project we had little experience with parallel programming, and no experience with GPU programming. In this project we used an NVidia graphics card which supports the Cuda framework. Though we had the option of implementing an offline system, we wanted to go all the way and deal with real-time tracking. This decision necessitated implementation of massive calculations in parallel on the GPU, so we had to learn from scratch how to utilize the GPU threads. We learned Cuda from books, web and forums. We delved into the GPU architecture and found ways to take advantage of its features, such as shared and global memory, register allocation, execution order, profiling, etc. Additionally, we searched for ways of optimizing the code and managed to accelerate our program by a factor of 16 using only Cuda optimizations.

As the project progressed we found that we needed another way to test our code, preferably through a single-threaded program that allowed sequential debugging. We decided on Matlab, as a mathematically oriented software with wide debugging capabilities. We had one problem - neither of us had much experience with Matlab. So the studying began anew. We implemented the whole project again, this time in Matlab code. This greatly helped our debugging and allowed us to find some major problems we couldn’t have found otherwise.
6.1.2 Difficulty Debugging

Along with the ins and outs of programming a large scale multi-threaded system, we also had to learn how to debug one. Any debugging tool causes the code to execute sequentially, which can change the program’s behavior. Additionally, step-by-step debugging is impossible when each stage of the pipeline launches hundreds of thousands of threads running in parallel. A problem with the configuration of our NVidia graphics card prevented us from using the Nsight debugging software, leaving us with only printf’s. Furthermore, our system handles vast amounts of data and only a small portion could be printed out for debugging purposes.

As a way of dealing with this problem, we decided to write our data to files which could easily be processed and tested using Matlab. This finally led us to implement the project on Matlab, giving us a new strong debugging tool.

6.1.3 Inaccuracies in the KinectFusion Paper

Two major setbacks over the course of the project were caused by inaccuracies we discovered in the KinectFusion paper.

The first one is in the pseudo-code of the algorithm for projective point-plane data association (finding correspondences for ICP). The pseudo-code specifies that the current global camera position should be used when calculating correspondences for each ICP iteration. This prevents convergence of the ICP algorithm, as the same vertices are always chosen as correspondences. In actuality, the transformation matrix calculated in the previous ICP iteration should be used (section 3.3).

The second inaccuracy is in the pseudo-code of the algorithm for projective TSDF integration. When calculating the SDF value (see Equation 4.1), the paper suggests that the depth value at the required pixel should be compared to the distance of the current voxel from the camera. Obviously, a depth value cannot be compared with an actual distance, because the depth value takes into account only the \(z\)-coordinate and disregards the \(x\) and \(y\) coordinates. In actuality, the distance of the surface vertex from the camera should be compared to the distance of the voxel from the camera.

6.1.4 Modularity and Portability

Once we had a working system, we turned our attention towards ensuring our code could be reused and expanded upon in future projects. The first improvement we sought to include was the ability to use OpenFusion to reconstruct data from any depth camera. This entailed a massive redesign of the system with an emphasis on modularity and ease of access for future integrators. Along the way we realized that we had relied on specific versions of Cuda and OpenCV, making portability very
difficult. We exchanged OpenCV for libraries that can be statically linked (CImg for display functions and Eigen for matrix calculations) and rewrote version-specific Cuda functions in order to ease the way for future developers. After putting in many hours of extra work we ended up with a more robust system that can be used and further developed by anyone with a depth camera.

6.2 Summary

After many hours of work we managed to create a working pseudo-real-time 3D reconstruction system based on a very challenging paper. We learned how to work in both Cuda and in Matlab, and implemented a number of advanced mathematical algorithms. Along the way we dealt with difficult software problems such as synchronization of a multi-threaded system and optimization of code for a faster run time. OpenFusion can be run with multiple cameras, and allows easy integration of new depth sensors. The system can be used to export the reconstructed model and to create a 3D mesh which can be opened in 3D editors, and even printed out in a 3D printer.

Some screenshots from our OpenFusion:

![Fig. 6.1: Scan using the Asus camera](image1)

![Fig. 6.2: Scan using Microsoft Kinect](image2)
The GIP camera gives noisier input data, but all in all gives a closer, more highly detailed scan. We reduced most of the noise using Gaussian and average filters.

The system is stable enough to deal with movement within the scene. When the scene being scanned changes, the model is updated over time to fit the new configuration.
The scanned model was exported as a 3D signed distance function, and then run through an offline Matlab implementation of the Marching Cubes algorithm to get a triangle mesh. The resulting mesh was then printed out using the GIP lab’s Makerbot Replicator 2 3D printer.

6.3 Future Work

There are many directions that can still be explored with OpenFusion, such as:

- Further optimization of the code for a true real-time reconstruction process.
- Implementation of algorithms described in the KinectFusion paper which were left out of this iteration
  - Coarse-to-Fine iterations of the ICP algorithm
  - Bilateral filter for smoothing input data.
- Use of RGB data received in some depth cameras (including Microsoft Kinect and Asus Xtion Pro) to add real-world coloration to the reconstructed module.
- Finding a way to remove the size constraint imposed by the volume, allowing whole rooms, or even entire buildings to be scanned.
Bibliography


