FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

3D Reconstruction with the Kinect-Camera

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Klemens Jahrmann
Matrikelnummer 0826080

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass. Dipl.-Ing. Mag.rer.soc.oec. Martin Knecht, Bakk.techn.

Wien, 18.02.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

3D Reconstruction with the Kinect-Camera

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Klemens Jahrmann
Registration Number 0826080

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Dipl.-Ing. Mag.rer.soc.oec. Martin Knecht, Bakk.techn.

Vienna, 18.02.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Klemens Jahrmann
Bukovicsgasse 33, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

The procedure of collecting 3D data via an input device and processing it to a virtual 3D model is called 3D reconstruction. It is a widely used technique in visual computing, since modern applications like games or visualizations tend to be more and more photo-realistic leading to high costs in content creation. By using 3D reconstruction high quality geometry can be generated out of real objects. However to obtain good reconstructions special hardware is needed which is very expensive.

Since Microsoft released the Kinect camera, which has a depth sensor in addition to the RGB-sensor, a quite cheap hardware is available that is able to extract 3D data of its surroundings. KinectFusion also developed by Microsoft is a technique that uses the Kinect camera for 3D reconstruction in real-time. In order to achieve real-time speed the algorithm is executed almost exclusively on the graphics card.

Each frame the algorithm first gathers the information from the Kinect and processes it. After that it measures the camera's position in space and fills a 3D volume with surface data. Finally a raycasting algorithm is used to extract isosurfaces out of the volume.

During the work on the thesis we implemented the KinectFusion algorithm inside the RESHADE framework. The results and the implementation itself are presented as part of the thesis.

Contents

CHAPTER 1

Introduction

1.1 General Information

3D reconstruction is a technique that is needed in many different areas. In industry very accurate models are used for physical simulations or quality tests. In addition computer games or visualizations are going to be more and more photo-realistic so the models have to look like real objects which is easy and quickly done with a good reconstruction tool.

3D reconstruction tools are used for some time, but they need special hardware which is very expensive and most often just viable for companies. Since Microsoft released the Kinect camera which has a depth sensor in addition to the RGB-sensor at the end of 2010 a quite cheap camera is available for everyone. So developers have the opporunity to use geometric input besides just using color pictures that they get from a webcam for example. With this impact new methods are developed for gesture control systems or 3D reconstruction.

1.2 About the Kinect-Camera

The Kinect camera has made a big influence not only concerning the gaming section but concerning the whole IT environment. James L. McQuivey, Vice President, Principal Analyst, Forrester, describes it as *"Kinect is to the next decade what the operating system was to the 1980s, what the mouse was to the 1990s, and what the Internet has been ever since. It is the thing that will change everything"* [11]. Although this statement is non-scientific and wants to predict the future it somehow reflects the excitement for this device.

The Kinect camera was developed by Microsoft in cooperation with PrimeSense. It was first introduced in June 2009 during the Electronic Entertainment Expo (E3) and finally published in November 2010. The Kinect itself consists of three different systems which are working together on PrimeSense's Carmine chip (PS1080) as it is described in [12]:

- It has a standard CMOS color sensor, to retrieve an RGB picture.

Figure 1.1: Kinect IR pattern from [2]

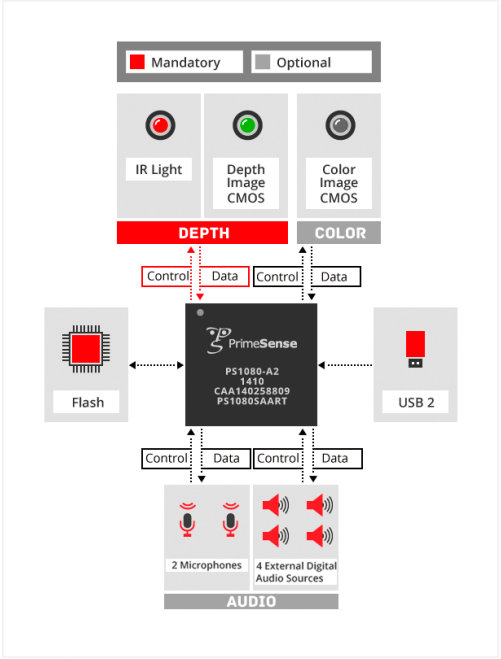

Figure 1.2: PrimeSense Technology [12]

- To gather depth information it includes an infrared laser which shoots IR rays through the whole scene (seen in *Figure 1.1*) and a CMOS sensor that records them. The distance to the camera is then measured by the size and the position of the recorded IR dots. [8]
- In addition the camera has a built in 3D microphone to get audio information.

After all information is retrieved the on chip system produces a depth map with the resolution of 640x480 pixels out of the raw distance values that are sampled at a resolution of 1600x1200. The downsampling is important to fill holes and reduce noise. Then the depth map is aligned to the RGB picture and they are combined to a single RGBD texture that is sent over the USB port. More technical details are descibed in [8].

CHAPTER 2

Related work

In paper [14] the authors descibe the algorithm for bilateral filtering which is discussed in *Chapter 2.1*. In another paper [1] the authors introduce a volumetric method for building models out of range images which is described in *Chapter 2.2*. Both methods are used in the KinectFusion algorithm mentioned in the later sections.

2.1 Bilateral Filtering

The aim of bilateral filtering [14] is to reduce the noise together with preserving hard edges so that the filtered image appears smooth but not blurry. Compared to a low-pass domain filter like the gaussian filter the bilateral filter function does not only consider geometric closeness as a criterion but also photometric similarity like a range filter. This exploits the assumption that high frequent noise on smooth regions has only small variation in the color range whereas edges have high variations.

Before bilateral filtering there only existed iterative methods which were able to satisfy these expectations by solving partial differential equations (e.g. anisotropic diffusion [10]). The disadvantage of iterative methods are that they might become unstable over many iterations and that their efficiency is depending on the computational architecture. On the other hand bilateral filtering is a noniterative and simple method which still preserves edges for both grayscale and color images. Espacially when dealing with color images simple filters like the gaussian filter fail because they can only operate on the three bands of a color image separately. This leads to phantom colors due to the fact that the different bands does not have the same contrast levels. Bilateral filters overcome this problem by working in the CIE-Lab color space in which nearby values correspond to similar perception for the human eye.

Since the bilateral filter acts like both a domain filter and a range filter, it is also computed similar to them. A low-pass domain filter is defined as follows:

$$\mathbf{h}(\mathbf{x}) = k_d^{-1}(\mathbf{x}) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathbf{f}(\xi) c(\xi, \mathbf{x}) d\xi \tag{2.1}$$

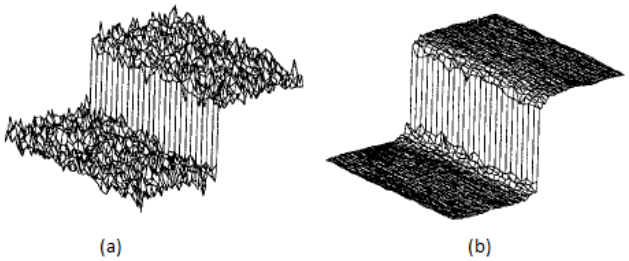

Figure 2.1: Illustration of the result of a bilateral filter from [14]. (a) shows an edge with noise applied to it and (b) shows the bilateral filtered result.

where $\mathbf{f}(\xi)$ is the function being filtered, k_d^{-1} works as a normalisation factor and $c(\xi, \mathbf{x})$ measures the geometric closeness between the neighbourhod center $\mathbf{x}$ and a nearby point ξ. A range filter is defined quite similar:

$$\mathbf{h}(\mathbf{x}) = k_r^{-1}(\mathbf{x}) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathbf{f}(\xi) s(\mathbf{f}(\xi), \mathbf{f}(\mathbf{x})) d\xi \tag{2.2}$$

with the difference that $s(\mathbf{f}(\xi), \mathbf{f}(\mathbf{x}))$ refers to the photometric similarity between the center pixel $\mathbf{x}$ and a nearby pixel ξ.

The bilateral filter is then defined as the combination of the definitions of domain and range filters mentioned above:

$$\mathbf{h}(\mathbf{x}) = k^{-1}(\mathbf{x}) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathbf{f}(\xi) c(\xi, \mathbf{x}) s(\mathbf{f}(\xi), \mathbf{f}(\mathbf{x})) d\xi \tag{2.3}$$

with the normalisation factor $k(\mathbf{x})$ defined as:

$$k(\mathbf{x}) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} c(\xi, \mathbf{x}) s(\mathbf{f}(\xi), \mathbf{f}(\mathbf{x})) d\xi \tag{2.4}$$

For smooth regions where all pixels inside a neighborhood are similar to each other the term $k^{-1} \cdot s$ is close to 1 which makes the filter behave like a common low-pass domain filter. So when a bilateral filter is applied to an image it replaces the pixel's value at each position $\mathbf{x}$ with an average of similar pixels in a neighborhood. A schematic illustration of the algorithm is shown in *Figure 2.1* and an example of a bilateral filtered image can be seen in *Figure 2.2*.

2.2 Volumetric Method for Building Models

The aim of this method [1] is to achieve smooth and detailed models out of range images given by a range scanner. A range image is an image which stores distance values from the camera's point of view instead of colors. The complexity of this topic is that the representation has to be updated incrementally, it has to be directional independent and robust in the presence of outliers. The volumetric method is able to overcome all of these problems. It uses a three dimensional

Figure 2.2: Comparison between a gaussian and a bilateral filter from [6].

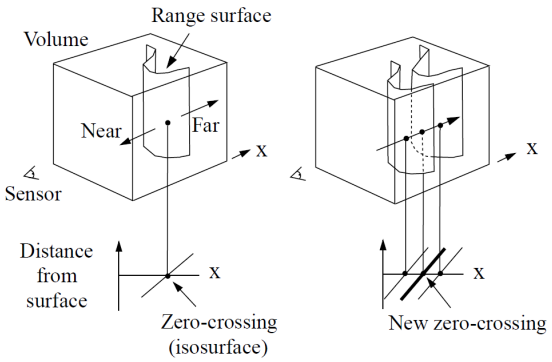

Figure 2.3: Combination of different unweighted distance functions from [1]

voxel grid filled with a signed distance and a weight function for representing the isosurfaces. For multiple range images the algorithm first measures the distance function of each range image before they are added together inside the volume.

The distance function $D(x)$ describes a continous function that is represented by discrete samples inside the voxel grid and linear. A voxel's distance value measures the distance to the nearest surface along the line of sight to the sensor where the sign of the distance value varies depending on whether the voxel lies in front of or behind the surface. If the method only combined unweighted distance functions it would lead to a simple averaging at each step and outliers would deteriorate the results dramatically (shown in *Figure 2.3*).

The weight function $W(x)$ also describes a continous function that is represented by discrete samples inside the voxel grid but it is not linear like the distance function because it tapers of behind range points to prevent surfaces on opposite sides of the object interfere with each other. Although there is a trade-off involved in choosing the best fitting point where the weight function falls off because it should be far enough so that near distance values stay robust, but to the contrary it should also be as narrow as possible to tolerate thin objects. This problem can be solved by aligning the weight fall-off to the sensor's margin of error but this needs a precise

sensor calibration.

When dealing with multiple range images the voxel grid has to be updated for each single image. To measure a voxel's data of the $i + 1$ iteration the two interpolation formulas are described in the following equations:

$$D_{i+1}(x) = \frac{W_i(x)D_i(x) + w_{i+1}(x)d_{i+1}(x)}{W_i(x) + w_{i+1}(x)} \tag{2.5}$$

$$W_{i+1}(x) = W_i(x) + w_{i+1}(x) \tag{2.6}$$

where $D(x)$ and $W(x)$ represent the voxel's data and $d(x)$ and $w(x)$ the measured values from the range image.

After all range images are processed the isosurfaces can be extracted out of the volume by finding the zero crossings of the distance function and a mesh can be formed. This mesh is then tesselated and holes are filled. One result of the method can be seen in *Figure 2.4*.

Figure 2.4: The result of the volumetric method from [1]. (a) shows the original statuette, (b) the result of the method with only 1 range image and (c) the result after processing 48 range images.

CHAPTER 3

Kinect Fusion

KinectFusion is a 3D reconstruction technique that uses Microsoft's Kinect camera as input device to reconstruct arbitrary indoor scenes with subcentimeter accuracy although the final accuracy depends on the maximum dimension of the reconstruction. The larger the space in the real world is defined for the reconstruction, the less detail can be reconstructed due to numerical limits. Three papers [4,5,9] explaining this technique were published in autumn 2011 by the developers Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges and Andrew Fitzgibbon, who are employed either by the Microsoft Research coorporation or by an academic institution. The aim of KinectFusion is to create a 3D reconstruction system which is fast enough and easy to handle to become an enabler for many AR and interaction scenarios, like Microsoft's latest Project called IllumiRoom [7], where the gaming boundaries shall be able to extend the hardware monitor. As stated in [9] the reconstruction method consists of 4 steps:

- In the *surface measurement* step 3D surface positions and normals are calculated (see *Section 3.2*).
- During the sensor *pose estimation* the position and orientation of the camera is estimated (see *Section 3.3*).
- Afterwards the *surface reconstruction update* step fills a 3D volume with surface information (see *Section 3.4*).
- In the end the *surface prediction* step goes through the volume and calculates surface positions that are used for the rendering (see *Section 3.5*).

How those steps interact with each other is illustrated in *Figure 3.1*.

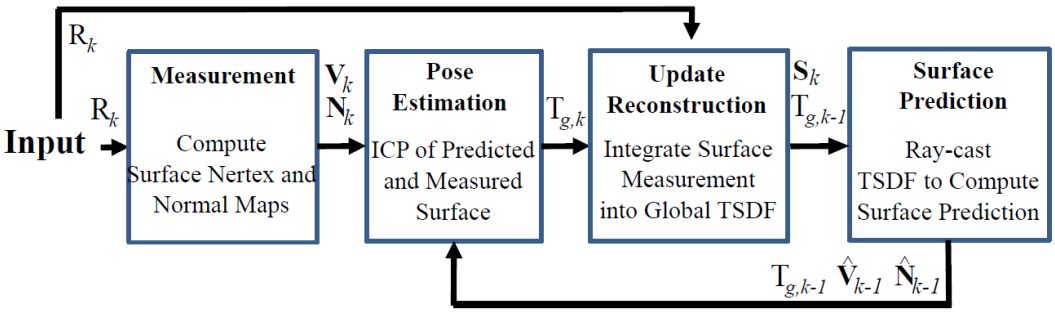

Figure 3.1: KinectFusion system workflow from [9]

3.1 Technical environment

The Kinect camera creates a 10-bit depth map with a resolution of 640x480 at 30Hz and sends it to the application. To attain real-time performance every reconstruction step is executed on the graphics card using highly parallel GPGPU techniques, so that they reach a framerate at least equal to Kinect's frequency.

As mentioned in the previous section the KinectFusion algorithm reconstructs the geometry inside a 3D volume. The volume is filled with truncated signed distance function (TSDF) values and is updated every single frame. The TSDF calculates for each voxel the distance to the nearest surface along the viewing direction. If a voxel is located in front of a surface the TSDF value's sign is positive, otherwise it is negative. Then the distances are clamped by a user defined value μ on both the positive and the negative side.

3.2 Surface measurement

First of all the noisy raw input depth map has to be pre-processed by applying a bilateral filter [14] (see *Section 2.1*) onto it so that the noise is reduced and small holes are closed while the rest of the details stay preserved. After that the depth values are converted to 3D coordinates in camera space to obtain a vertex map. By using the vertex map the corresponding normal map is generated by calculating the cross product between neighbouring vertices. Based on these two maps a multi scale surface measurement pyramid of height $L = 3$ is being built by subsampling both the vertex and normal map using an average smoothing filter with a threshold.

3.3 Pose estimation

The aim of this step is to localise the current camera position in world space by calculating the 6 degrees-of-freedom (6DOF) camera pose. This is done by using all available data from the last and current frame for an "iterative closest point" based pose estimation method [13]. The algorithm takes the surface measurement data consisting of a vertex map and a normal map from the current frame (V_k, N_k) (described in *Section 3.2*) and the result of the surface prediction

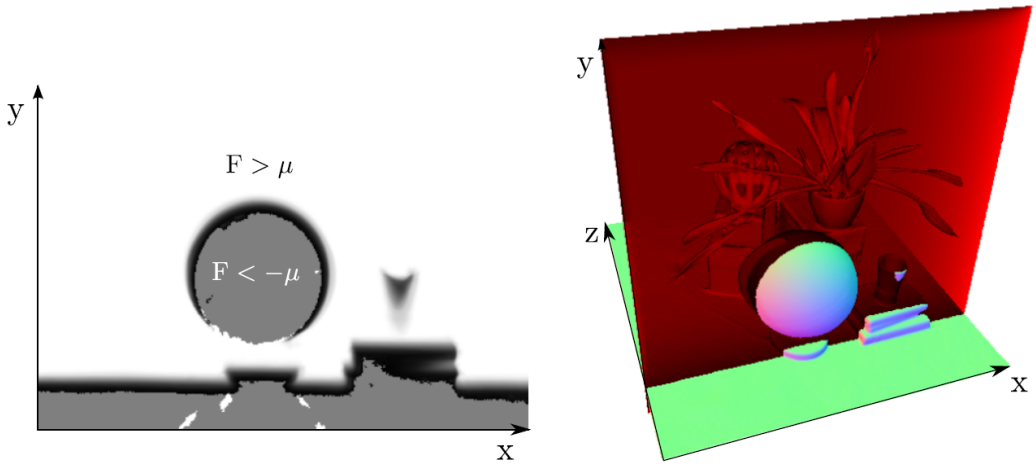

Figure 3.2: One slice of the 3D volume showing the TSDF with a given μ from [9]

from the previous frame $(\hat{V}_{k-1}, \hat{N}_{k-1})$ (described in *Section 3.5*). Assuming only small frame-to-frame motion the two measurements are compared with each other and corresponding points are selected to calculate an estimated frame-to-frame transform which is applied to the previous frame's data. This step is iterated by a linearised solver that tries to minimize the error value formed by the *Equation 3.1*, where $T_{g,k}$ is the estimated transformation and $\mathbf{V}_k(\mathbf{u}), \hat{\mathbf{V}}_{k-1}(\mathbf{u})$ are the corresponding points:

$$\mathbf{E}(T_{g,k}) = \sum_{\mathbf{u}\in U} \left\| \left(T_{g,k}\mathbf{V}_k(\mathbf{u}) - \hat{\mathbf{V}}^g_{k-1}(\hat{\mathbf{u}}) \right)^T \hat{\mathbf{N}}^g_{k-1}(\hat{\mathbf{u}}) \right\|_2 \tag{3.1}$$

As result a six-dimensional feature vector is found which describes the 6 degrees-of-freedom (3 rotation angle values and 3 translation values, see *Equation 3.2*) of the camera motion between the last and the current frame. This feature vector can be added to the previous frame's camera pose to calculate the camera pose of the current frame.

$$\mathbf{x} = (\beta, \gamma, \alpha, t_x, t_y, t_z)^T \in \mathbb{R}^6 \tag{3.2}$$

3.4 Update reconstruction

The reconstruction step uses the raw depth map to fill a 3D volume with surface data. The reason why the raw depth map is used instead of using the bilateral filtered one like in the steps before is that the reconstruction shall also include details with high frequencies which are lost during filtering. At first the depth values are converted to 3D world space positions using the current camera pose which was calculated during the previous step. Then for each voxel of the 3D volume a signed distance is calculated and truncated at μ which forms a truncated signed distance function (TSDF) over the volume so that every voxel shows its distance to the next surface point in view direction of the camera. In addition every voxel saves a weight value

which is accumulated over each frame a voxel is in μ-range of a surface and is used to blend the new TSDF values over the old ones to reduce noise errors.

$$F_k(\mathbf{p}) = \frac{W_{k-1}(\mathbf{p}) \cdot F_{k-1}(\mathbf{p}) + W_{R_k}(\mathbf{p}) \cdot F_{R_k}(\mathbf{p})}{W_{k-1}(\mathbf{p}) + W_{R_k}(\mathbf{p})} \tag{3.3}$$

$$W_k(\mathbf{p}) = \min\left(W_{k-1}(\mathbf{p}) + W_{R_k}(\mathbf{p}), W_{max}\right) \tag{3.4}$$

The formula above *3.3* describes the blending between the old and the new TSDF value where at frame k for each voxel $\mathbf{p}$ a new TSDF value F_k is calculated by averaging between the old value F_{k-1} and the measured one F_{R_k} with their weights W_{k-1} and W_{R_k}. The second formula *3.4* describes how the weights are incremented where at step k for each voxel $\mathbf{p}$ the new weight W_k is calculated by adding a value W_{R_k} to the old W_{k-1} until it reaches some maximum value W_{max}. In practice $W_{R_k} = 1$ leeds to a casual averaging between the values, it is simple and still provides good results.

Every voxel with a distance greater μ is not changed during this step. After each voxel is processed one slice of the volume might look like the example in *Figure 3.2*.

3.5 Surface prediction

In the last step the actual rendering of the scene is executed. From the position of the camera calculated in step 2 (see Section *3.3*) for each pixel of the final image a ray is shot through the TSDF volume which has already been filled in step 3 (see Section *3.4*). The calculation starts at a minimum depth from the camera's point of view to simulate the near clipping plane and ends if a surface is hit or if the ray exits the volume. A surface hit is found when the TSDF performs a zero crossing: A positive to negative zero crossing indicates a front facing surface and a negative to positive zero crossing indicated a back facing surface. In case of leaving the volume or hitting a back face no surface is assumed for the current pixel. The surface normal at a surface point $\mathbf{p}$ can be measured directly by the TSDF values using the numerical derivatives:

$$\nabla F = \left[\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right]^T \tag{3.5}$$

In dealing with ray marching a common issue is how empty space is skipped. One approach might be min/max blocks that define spaces with no information but since the volume is updated all the time this would lead to a pre-processing overhead. Since the volume filling algorithm uses a TSDF instead of a non-truncated signed distance function (SDF) the volume has to be sampled only at a stepsize < $2 \cdot \mu$ because then empty spaces can be passed faster and zero crossings are still found properly. In addition it can be assumed that a TSDF value near $F(\mathbf{p}) = 0$ is also a good enough approximation to speed up the ray calculation even more. For higher quality surface prediction a more accurate zero crossing has to be found. This can be obtained by solving a ray/trilinear cell intersection but that costs too much time to be processed for every ray in real-time. Still a simple approximation of this algorithm is used by measuring two trilinear interpolated TSDF values at either side of the zero crossing F_t^+ and $F_{t+\Delta t}^+$ which are at the ray distances t and $t + \Delta t$ to calculate a distance t^* that indicates the zero crossing point.

$$t^* = t - \frac{\Delta t \cdot F_t^+}{F_{t+\Delta t}^+ - F_t^+} \tag{3.6}$$

The improvement of the *Equation 3.6* over the faster approximation can be seen in *Figure 3.3*.

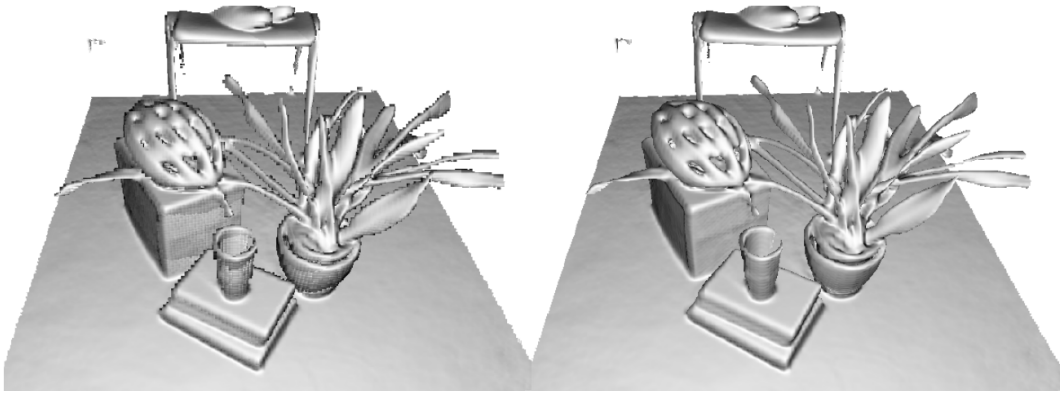

Figure 3.3: A scene reconstructed with a fast approximation (left) and with a more precise approximation (right) for the surface position calculation from [9]

CHAPTER 4

Implementation

The KinectFusion algorithm was integrated into the RESHADE framework, which is a rendering engine for visually plausible mixed reality.

Section 4.1 describes the initialization steps that have to be done before the algorithm is ready to start. The next *Sections 4.2,4.3,4.4* will cover the 3 steps in which the volume is filled, the camera position is calculated and the final image is processed. Afterwards some technical limitations that had to be overcome are also mentioned in *Section 4.5*.

4.1 Setup

In the first step the TSDF volume must be set up. It consists of a Texture2DArray which is initialised using a resolution that is passed to the volume. Together with the resolution the volume gets a dimension value which shall indicate the covered size of the volume in the real world, for example if the dimension is 5.0 then the program can reconstruct everything in 2.5m range along each axis from the starting camera position. The ratio between resolution and dimension determines the accuracy of the algorithm.

In fact not only one volume is needed but two. This is due to the fact that DirectX can not read and write to a single texture simultaneously (see *Section 4.5*).

For retrieving the data from the Kinect camera, I use the KinectCapture class that is built in the framework. Also for rendering purpose I use the built in deferred renderer, for which I set up a texture for the surface positions and one for the surface normals.

4.2 Tracking

At first the camera's position in space is tracked. For this the Studierstube marker tracker [3] is used instead of the ICP tracker mentioned in *Section 3.3*. During the setup some BCH markers (e.g. *Figure 4.1*) of a given dimension are registered together with a pose which they are referring to. In this case only one marker is registered and bound to the camera pose. Then every frame the

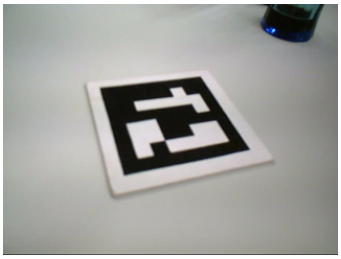

Figure 4.1: Example of a BCH marker that can be tracked by the Studierstube tracker from [3]

tracker receives a monochrome picture from the camera and searches for the registered markers. When a marker is found its corresponding pose can be calculated.

4.3 Volume data filling

In the first step of the scene reconstruction the TSDF volume needs to be filled which is executed completely on the graphics card. In order to do that some data have to be acquired:

- The camera's view matrix is retrieved from the tracking system (described in *Section 4.2*) and with it the view vector is calculated.
- The depth map of the Kinect device is gathered by the KinectCapture class.

Then all the data and textures are transferred onto the graphics card. To fill the volume the shader program has to draw a screen spanning quad for each slice of the volume. In order to reduce drawing calls I use the geometry shader to create additional quads, so that one draw call can fill several slices.

In the shader stage the vertex shader remains passive by just passing through the input position and texture coordinates. The geometry shader determines which slice is currently processed by setting the SV_RENDERTARGETARRAYINDEX value to the index of the current slice. In addition it calculates for each slice the texture lookup coordinates in the Kinect's depth map coordinate space by converting the three dimensional texture coordinates of the volume to world position coordinates and applies then the camera's combined view and projection matrix (see in *Listing 4.1*).

One important thing to note from *Listing 4.1* is that a three dimensional vector is returned by the method rather than just a two dimensional since the depth map has only two dimensions. The third dimension indicates if the voxel lies in front of the camera (in viewing direction) or behind. This is important because I only want to process voxels that are seen by the camera.

In the pixel shader several steps have to be processed. At first the texture lookup coordinates have to be normalised and checked if they are inside the view frustum by identifying if they are in between the [0,1]-interval. If they are positioned outside the frustum or behind the camera this voxel will not be processed anymore and the value of the previous frame is assigned. Otherwise the depth map is sampled and its value is converted into world-space coordinates. After that the voxel's position is also converted into world-space coordinates and the distance between

```
float3 getCameraUVFromSliceUVW(float2 slice_uv, uint slice_w) {
   float3 sliceWP = getVolumeWP(slice_uv, slice_w);
   float4 camUV = mul(float4(sliceWP,1.0f),viewProjectionMatrix);

   return camUV.xyz;
}

float3 getVolumeWP(float2 slice_uv, uint slice_w) {
   float z = ((float) slice_w) / volume_resolution.z;
   float3 worldPosition = float3(slive_uv, z) - 0.5f;
   worldPosition = worldPosition * volume_dimension.xyz;

   return worldPosition;
}
```

Listing 4.1: Converting texture coordinates from volume space to view space.

both positions is calculated. The sign of the distance in viewing direction is retrieved by the dot product of the camera's view vector and the direction vector between the two world space positions. With a given μ it is checked whether the distance is inside the relevant range. If that is the case the weight is incremented and the distance value is blended over the old TSDF value sampled linearly by *Listing 4.2* (this special method is needed since a Texture2DArray is used instead of a Texture3D) using *Equation 3.3* otherwise the previous' frame value stays current. The process of the volume filling is shown in *Algorithm 4.1*.

```
float4 sampleVolume(float3 uvw) {
   float sliceId = floor(uvw.z);
   float factor = frac(uvw.z);

   float4 sample0 = volume.SampleLevel(VolumeSampler, float3(uvw.xy,
       sliceId), 0);
   float4 sample1 = volume.SampleLevel(VolumeSampler, float3(uvw.xy,
       sliceId+1), 0);

   return lerp(sample0, sample1, factor);
}
```

Listing 4.2: Method to retrieve linearly interpolated volume values

4.4 Raycasting pass

In the next step the 3D data needs to be measured, which is also executed exclusively on the graphics card. The shader program for calculating the 3D surface positions needs only the TSDF volume and the inverted view and projection matrix of the camera which are transmitted to the

Data: $uvCamera$; ← inhomogenious texture coordinates for depth map
$uvVolume$; ← 3D texture coordinates for TSDF volume
$viewVector$; ← camera's view vector in world-space
μ; ← given value for truncation of the distance values
max_weight; ← given maximum weight
Result: $TSDFdata$; ← TSDF value and weight of a specific voxel

```
1  depthLookup = uvCamera.xy / uvCamera.z;
2  depthLookup = depthLookup · 0.5 + 0.5;
3  if depthLookup inside [0,1] ∨ uvCamera.z ≤ 0.0 then
4  |  return sampleVolume(uvVolume);
5  else
6  |  depthSample = sampleDepthMap(uvCamera);
7  |  if depthSample == 0.0 then
8  |  |  discard;
9  |  end
10 |  wsPos = calculateWorldSpacePositionFromDepth(depthSample);
11 |  volumeSample = sampleVolume(uvVolume);
12 |  slicePos = calculatePositionInVolume(uvVolume);
13 |  distance = distance(wsPos,slicePos);
14 |  sign = sign(dot(slicePos - wsPos, viewVector));
15 |  if abs(distance > μ) then
16 |  |  return volumeSample;
17 |  else
18 |  |  oldWeight = volumeSample.y;
19 |  |  oldDistance = volumeSample.x;
20 |  |  weightIncrease = 1;
21 |  |  weight = min(oldWeight + weightIncrease,max_weight);
22 |  |  distance = (oldDistance · oldWeight + distance · weightIncrease) /
   |  |  (oldWeight + weightIncrease);
23 |  |  return distance,weight;
24 |  end
25 end
```

Algorithm 4.1: Volume filling step (see *Section 4.3*)

GPU by the application. In the vertex shader step the position is passed through normally and in addition the begin and exit point of the ray is calculated by multiplying the two dimensional near and far plane point with the inverted view and projection matrix (shown in *Listing 4.3*).

At the beginning of the pixel shader step the interpolated ray start and endpoints are normalized by dividing by their homogenous coordinates. The reason why the normalization is executed in the pixel shader rather than in the vertex shader is that the rays obtain perspective correctness if they are interpolated first and then normalized as the other way round. With the two ray points the ray direction can be calculated and the step width between the sample points

```
...
float4 ptNear = float4(input.position.xy, 0.0f, 1.0f);
float4 ptFar  = float4(input.position.xy, 1.0f, 0.0f);

ptNear = mul(ptNear, invViewProjectionMatrix);
ptFar  = mul(ptFar, invViewProjectionMatrix);
...
```

Listing 4.3: Calculation of the near and far plane point of the ray in the vertex shader

is set to a fraction of μ. In the implementation $\mu \cdot 0.9$ is used if the ray is in empty space ($weight < max\_weight \cdot 0.1$) and $\mu \cdot 0.1$ otherwise which led to the best results in performance and accuracy. Another approach is to adjust the step width to the sampled TSDF values. If the weight is higher a specific value (e.g. $weight > max\_weight \cdot 0.9$) the step width is set to $sample \cdot 0.7 + \mu \cdot 0.001$. This leads to finer steps the nearer the ray comes to the surface. This approach gives more precise results but the performance gets worse since more steps are needed. After that the raycasting loop starts where at each iteration the current ray position inside the volume is converted to texture coordinates, the volume is sampled with these coordinates using linear sampling (using *Listing 4.2*) and it is checked whether there was a zero crossing between the old and the new sample points. When the loop is finished and a zero crossing is found two positions with referring samples remain, one before the zero crossing and the other one behind it. To calculate the approximate position of the zero crossing I linearly interpolate between the two positions by their sample's distance to zero. The interpolation factor is calculated as follows:

$$\begin{aligned} d1 > 0 &\leftarrow TSDF\ value\ before\ zero\ crossing \\ d2 < 0 &\leftarrow TSDF\ value\ after\ zero\ crossing \\ fac &= \frac{d1}{d1 + |d2|} \end{aligned} \tag{4.1}$$

If no zero crossing is found until the ray leaves the volume no surface position is measured for this pixel. The structure of this algorithm is shown in *Algorithm 4.2*.

4.5 Limitations

During the implementation some limitations in contrast to the implementation of the original KinectFusion paper have been found. Since DirectX is used instead of a GPGPU technique (like CUDA or OpenCL) a double buffering system for the TSDF volume has to be implemented because it's data is needed for read and write purpose which is not possible with DirectX textures. On the contrary GPGPU techniques work with buffer objects which allow parallel read and write access. As a result twice the memory space is needed for the double buffered volume texture which leads to a bit rougher reconstruction. In addition DirectX is not as parallelized as GPGPU

Data: $rayPosition$; ← current position of the ray
$stepSize$; ← step size to the next iteration
$sample$; ← volume sample at current position
Result: $surfacePosition$; ← found 3D surface position

```
1  oldPosition = rayPosition;
2  oldSample = sampleVolume(rayPosition);
3  rayPosition = rayPosition + stepSize;
4  while rayPosition is inside the volume and no zero crossing is found do
5  |  sample = sampleVolume(rayPosition);
6  |  if zero crossing is found then
7  |  |  break;
8  |  else
9  |  |  oldSample = sample;
10 |  |  oldPosition = rayPosition;
11 |  |  rayPosition = rayPosition + stepSize;
12 |  end
13 end
14 if zero crossing found then
15 |  return lerp(oldPosition, rayPosition, oldSample / (oldSample + abs(sample)));
16 else
17 |  discard;
18 end
```

Algorithm 4.2: Raycasting algorithm (see *Section 4.4*)

techniques since it is mainly optimized for rendering purpose than for calculation purpose like filling a volume.

As TSDF volume a Texture2DArray is used instead of a Texture3D because three dimensional textures were not accessible in the framework the way I needed them. Although a Texture2DArray works similar to a Texture3D there are some limitations following it. First the SV_RENDERTARGETARRAYINDEX has to be passed by the geometry shader which has a limited output size what results in less slices that can be processed each draw call. In addition the linear interpolation between neighbouring slices is not done automatically by the graphics card so the interpolation has to be executed in the shader programm (see *Listing 4.2*).

Another limitation is that a marker tracker is used instead of a SLAM based tracker or the proposed method from KinectFusion using iterative closest point algorithms for pose estimation (see *Section 3.3*). Since there always has to be the marker inside the camera's visible field of view it is hard to reach the back side of objects. The reason why a marker tracker is used though is that they have a good performance and their tracking accuracy is quite high.

CHAPTER 5

Results

In the following sections the results of the implementation are shown. At first some performance evaluations are presented in *Section 5.1* and afterwards several example images are shown in *Section 5.2*.

The implementation was tested on a notebook with a Windows 7 64-Bit system, an Intel Core i7 quad-core CPU running at about 2.7GHz, 8GB of RAM and a NVIDIA GeForce GTX 460M with the latest drivers (18.01.2012) installed. The engine is written in C# with DirectX10 and runs at a resolution of 640x480 pixel.

5.1 Performance

The performance of the implementation is measured by the DirectX10 query object using $Timestamp$ as $QueryType$ to evaluate the amount GPU ticks and $TimestampDisjoint$ to get the frequency (indicates how many ticks can be processed in a second) and to check whether

Volume filling pass			
Elapsed time (s)	Average ticks	Average time (ms)	Full runs per second
5	22,164,571	22.16	45
10	22,457,217	22.46	44
15	22,264,450	22.26	44
20	22,233,696	22.23	44
25	22,570,280	22.57	44
30	22,264,149	22.26	44
35	22,480,539	22.48	44
40	22,089,310	22.09	45

Table 5.1: Performance measurements of the volume filling GPU pass.

Raycasting pass			
Elapsed time (s)	Average ticks	Average time (ms)	Full runs per second
5	4,801,168	4.80	208
10	4,919,667	4.92	203
15	4,754,768	4.75	210
20	6,283,410	6.28	159
25	8,706,719	8.71	114
30	8,271,567	8.27	120
35	6,162,428	6.16	162
40	3,276,874	3.28	305

Table 5.2: Performance measurements of the raycasting GPU pass.

the result is reliable. The following tables show the average measurements of the volume filling GPU pass (*Table 5.1*) and of the raycasting GPU pass (*Table 5.2*).

5.2 Reconstruction examples

In this section some result images are shown. On the left side there is the Kinect's RGB image and on the right side the 3D reconstruction is shown.

Figure 5.1 shows a simple scene with a backpack, a ballpen and an USB flash drive lying on the floor.

Figure 5.2 displays some very small objects each with a height $\leq$ 1cm. For better perception the objects are marked in both images.

Figure 5.3 shows more complex objects (stuffed animals) placed on a work desk. The sunglasses can not be fully reconstructed because their glasses let the infrared light through.

The last *Figure 5.4* pictures a person sitting on a desk chair. The creation of this scene was the hardest because if an object moves it becomes blurry in the 3D reconstruction but if the marker moves just slightly the whole 3D reconstruction gets worse.

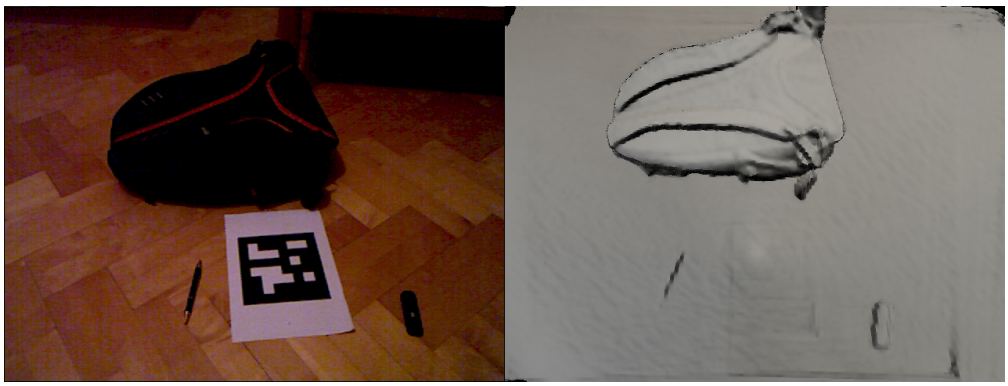

Figure 5.1: Result of the KinectFusion implementation showing a simple scene. On the left side there is the Kinect's RGB image and the right side shows the reconstruction of the scene.

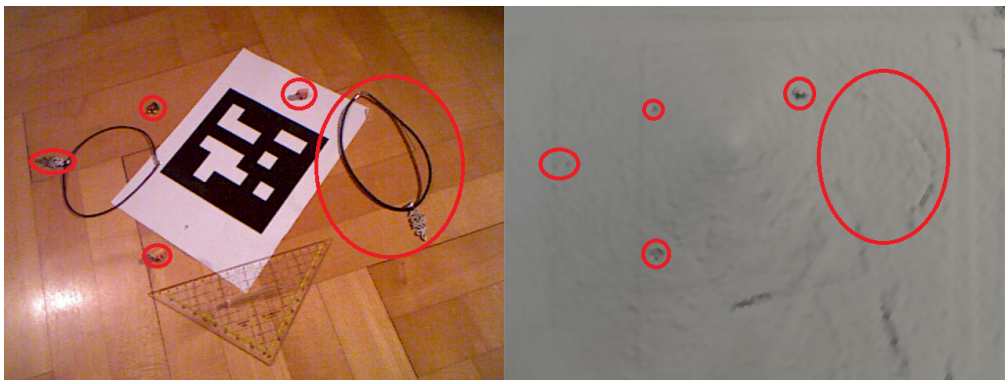

Figure 5.2: Result of the KinectFusion implementation showing very small objects with height $\leq$ 1cm. On the left side there is the Kinect's RGB image and the right side shows the reconstruction of the scene.

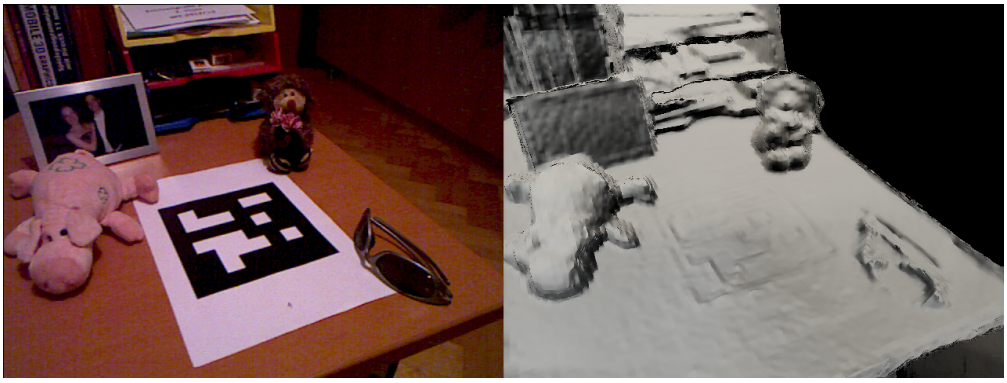

Figure 5.3: Result of the KinectFusion implementation showing more complex objects (stuffed animals) and sunglasses on a work desk. On the left side there is the Kinect's RGB image and the right side shows the reconstruction of the scene.

Figure 5.4: Result of the KinectFusion implementation showing a person sitting on a desk chair. On the left side there is the Kinect's RGB image and the right side shows the reconstruction of the scene.

CHAPTER 6

Conclusion

3D reconstruction is a very useful technique for content creation in a photo-realistic way. Before Microsoft released the Kinect camera only some professional tools existed which needed special and expensive hardware. With the release of the Kinect a quite cheap depth camera is available for everyone. KinectFusion is a technique that uses the Kinect camera to reconstruct complete rooms in real-time although the dimension of the reconstruction is in conflict with the precision of the reconstructed geometry.

During the work on the thesis the KinectFusion method was integrated into the RESHADE framework. Although some limitations had to be overcome the implementation works at reasonable frame rates. In constant lighting the application is very robust and is able to compute a detailed reconstruction within a short time.

Bibliography

[1] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 303–312. ACM, 1996.

[2] Matthew Fisher. http://graphics.stanford.edu/ mdfisher/images/kinectir.png. Accessed: 2012-12-29.

[3] TU Graz. http://handheldar.icg.tugraz.at/stbtracker.php. Accessed: 2013-01-23.

[4] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. 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*, UIST '11, pages 559–568. ACM, 2011.

[5] Shahram Izadi, Richard A. Newcombe, David Kim, Otmar Hilliges, David Molyneaux, Steve Hodges, Pushmeet Kohli, Jamie Shotton, Andrew J. Davison, and Andrew Fitzgibbon. Kinectfusion: real-time dynamic 3d surface reconstruction and interaction. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, pages 23:1–23:1. ACM, 2011.

[6] Mitsubishi Electric Research Laboratories. http://www.merl.com/projects/images/bilateralfilters.jpg. Accessed: 2013-02-10.

[7] Microsoft. http://research.microsoft.com/en-us/projects/illumiroom/. Accessed: 2013-01-24.

[8] Microsoft. http://social.technet.microsoft.com/wiki/contents/articles/6370.aspx. Accessed: 2013-01-24.

[9] 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 *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, pages 127 – 136. IEEE Computer Society, 2011.

[10] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(7):629 –639, 1990.

[11] PrimeSense. http://www.primesense.com/casestudies/kinect/. Accessed: 2012-12-29.

[12] PrimeSense. http://www.primesense.com/solutions/technology/. Accessed: 2012-12-29.

[13] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3d model acquisition. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 438–446. ACM, 2002.

[14] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839 –846. IEEE Computer Society, 1998.