FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Interdisciplinary Project Report

Development of a system that allows registration of segmented point cloud to patient CT data and provide augmentations

Sumit Dugar

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Interdisciplinary Project Report

Development of a system that allows registration of segmented point cloud to patient CT data and provide augmentations

Entwicklung eines Systems, das die Registrierung von segmentierten Punktwolken zu Patienten-CT-Daten ermglicht und bieten augmentationen

Author:	Sumit Dugar
Supervisor:	Prof. Dr. Nassir Navab
Advisor:	Dr. Ulrich Eck, Dr. Federico Tombari
Date:	April 30, 2018

Abstract

The need to improve medical diagnosis and reduce invasive surgery is dependent upon seeing into a living human system. The use of diverse types of medical imaging and endoscopic instruments has provided significant breakthroughs, but not without limiting the surgeon's natural, intuitive and direct 3D perception into the human body. Medical Augmented Reality (AR) systems display medical data such as volumetric images and segmented structures co-located with the patient's anatomy. This as a result effectively produces a view as if we are looking into the patient's anatomy [13]. Such Medial AR applications range from surgery planning to intraoperative guidance systems.

We are trying to develop a system that will augment the volumetric medical images onto a segmented body part. In order to correctly augment medical information onto the patient it is essential to accurately determine the transformation between the current pose of the patient, the viewing point of the observer, and the associated medical image data. The required transformations can be estimated by registering the surface (skin) of the patient with a surface extracted from the associated volumetric scans.

Our proposed system consists of four parts. Each part was handled by a different student. In the proposed system we first obtain a 3D scan of our patient via Kinect Fusion [27]. We then segment the 3D scan into it's constituent body parts and obtain the point cloud data for the part that interests us. Simultaneously we obtain the texture mapping for the 3D scan. In the final step we identify transformation between medical volumetric data and Kinect fusion model. This transformation is required for proper augmentation of the scan data on to the patient's body from a user's view who most probably will be wearing a Head Mounted Display (HMD).

Contents

Abstract v

Contents viii

Outline ix

1. Introduction 1
- 1.1. Goals and Requirements . . . 1
- 1.2. Contributions . . . 3

2. Theoretical Background 5
- 2.1. Coordinate systems . . . 5
- 2.2. Camera Intrinsics and Extrinsics . . . 7
- 2.3. Volumetric CT Data . . . 9
- 2.4. Volume Ray Casting . . . 10
- 2.5. Registration . . . 11
  - 2.5.1. Keypoint Estimation . . . 11
  - 2.5.2. Descriptor Estimation . . . 11
  - 2.5.3. Correspondence Matching . . . 13
  - 2.5.4. Correspondence Clustering / Correspondence Rejection . . . 13
  - 2.5.5. Transformation estimation . . . 13
  - 2.5.6. ICP . . . 13

3. Technical Details 15
- 3.1. Overview of Frameworks and Libraries . . . 15
- 3.2. Data Generation System . . . 18
  - 3.2.1. Makehuman Pipeline : Generate Random Human Models . . . 20
  - 3.2.2. Blender Pipeline : Render RGB and Depth Data . . . 20
  - 3.2.3. Results . . . 23
- 3.3. Registration System . . . 25
  - 3.3.1. Volume Rendering Pipeline . . . 26

3.3.2. Initialization of Shaders and Textures 26
3.3.3. Rendering using OpenGL 28
3.3.4. Post Processing 29
3.3.5. Registration Pipeline 30
3.3.6. Initial Estimate 30
3.3.7. Refinement Using ICP 32
3.3.8. Alignment with the Body 32

4. Future Work 37
4.1. Calculate the transformation ${}^{Kinect}T_{HMD}$ 37
4.2. Compare with other registration algorithms 38
4.3. Augmentation of medical data on patient's body 38

5. Conclusion 39

Appendix 39

Bibliography 43

Outline for the report

1. Introduction

In this chapter, we talk about our motivation for the project. We discuss some of the current applications of Augmented Reality in medicine. We state the goals and requirements of our project. We also define the different parts of the project and the contributions made in each part.

2. Theoretical Background

In this chapter, we present some theoretical background and concepts required for understanding the details of this project.

3. Technical Details

In this chapter, we provide a brief overview of the different libraries and frameworks used for the development of this project. Additionally, we go deeper in the proposed system and share the results.

4. Future Work

In this chapter, we talk about some future work in scope of the current project.

5. Conclusion

Finally, we summarize our work in this chapter and present the major conclusions we derived from it.

Appendix

Here we will share references and sources for the information that we have presented in this report. We also share additional information such as source code.

1. Introduction

The need to improve medical diagnosis and reduce invasive surgery is dependent upon seeing into a living human system. Medical Augmented Reality (AR) systems display medical data such as volumetric images and segmented structures co-located with the patient's anatomy[13]. This, as a result, effectively produces a view as if we are looking into the patient's anatomy. Such Medial AR applications range from surgery planning to intraoperative guidance systems.

We are trying to develop a system that will augment the volumetric images onto a body part. In order to correctly augment medical information onto the patient it is essential to accurately determine the transformation between the current pose of the patient, the viewing point of the observer, and the associated medical image data. The goal of this work was to device a pipeline for determining this transformation with minimum human intervention. The required transformation is estimated by registering the surface (skin) of the patient with a surface extracted from the associated volumetric scans.

1.1. Goals and Requirements

The overall goal for the body project is to create a system that will enable automatic augmentation of medical data onto a patient's body. This will hopefully, provide added assistance for medical diagnosis and other tasks.

Our proposed system consists of four parts. Each part was handled by a different student. A detailed diagram of the proposed system can be seen in Figure 1.1.

For proper augmentation of medical data we need to find a transformation ${}^{CT}T_{HMD}$ that can transform points from CT frame of reference to the HMD frame of reference i.e viewer's reference frame. Since it is not easy to directly obtain this transformation so we will indirectly find this using the transformations ${}^{CT}T_{Kinect}$, ${}^{Kinect}T_{HMD}$.

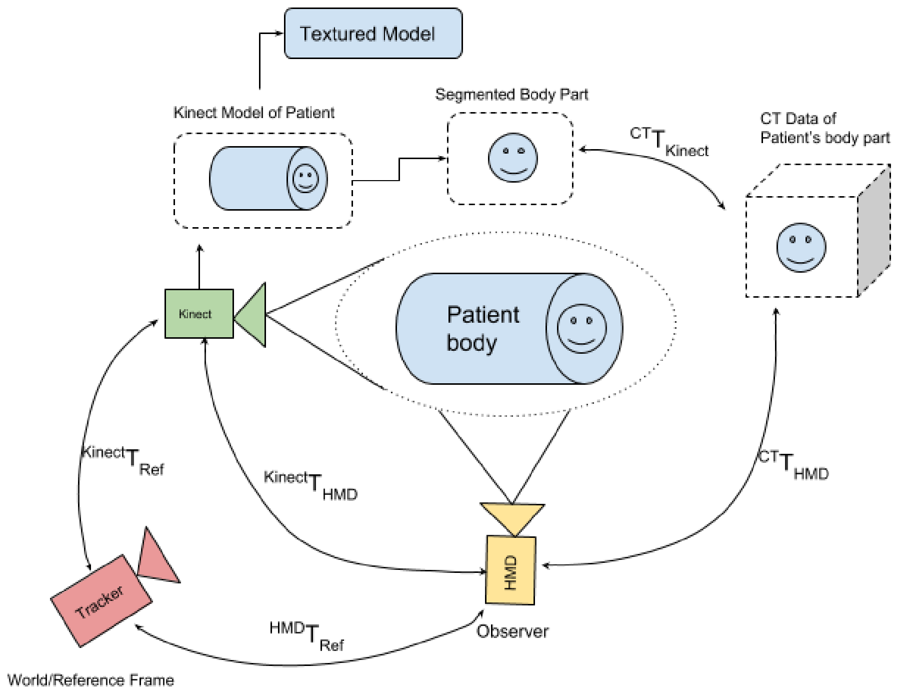


Figure 1.1.: Proposed System - It consists of four parts. We will first obtain a 3D scan of our patient via Kinect Fusion. We will then segment the 3D scan into it's constituent body parts and obtain the point cloud data for the part that interests us. Simultaneously we obtain the texture mapping for the 3D scan. The final step is proper augmentation of this scan data on to the patient's body from a user's view who will be wearing a HMD. To do so we need to know the transformation between the Kinect scan and the medical 3D scan. We obtain this via registering both the point clouds with each other.

Mathematically this can be formulated using the following equations.

$$^{CT}T_{HMD} = {}^{Kinect}T_{HMD} *^{CT} T_{Kinect} \tag{1.1}$$

$$^{Kinect}T_{HMD} = {}^{HMD}T_{Ref}^{-1} *^{Kinect} T_{Ref} \tag{1.2}$$

The first part consisted of creating a three-dimensional body model of the current patient. This is done by utilizing simultaneous localization and mapping with an inexpensive RGB-D camera. We implemented the KinectFusion[27] and Marching Cubes[20] algorithms to create a fully three-dimensional body mesh in real-time from data acquired by the examiner.

In the second step, we try to map the skin texture of the patient to the 3D model with a high resolution so that the lesions can be marked and later be tracked on the 3D body model with exact positions. To tackle this problem, colored images are projected onto the 3D model instead of averaging vertex colors to achieve high resolution.

In the third step, we are trying to segment the 3D body model obtained through Kinect Fusion by separately segmenting the depth maps that are input to the SLAM[9] algorithm. We input each depth map into a Convolution Neural Network Architecture inspired by the paper on MobileNets[18]. This architecture is chosen in order to obtain a fast semantic segmentation of each map. Each segmentation map is then combined through Kinect Fusion in exactly the same way as the RGB maps. After a 3D model is obtained, using color coding of the 3D segmentation model, a body part can be segmented out.

Finally in the fourth part, we first generate a surface point cloud from volumetric medical data such as CT / MRI. We obtain this point cloud by using raycasting[19]. Then, we try to register this point cloud with the point cloud of the segmented body part that was obtained in the third part. This results in a transformation $^{CT}T_{Kinect}$ that helps to transform points from the CT frame of reference to the $Kinect$ frame of reference.

1.2. Contributions

The two major contributions of this work are intended for two separate parts of this project. One of them is intended for the third part i.e segmentation part and other is intended for the fourth part i.e registration part.

The first step of this work is to implement the data generation pipeline for generating training data that can be used by the segmentation network. The main motivation for this is scarcity of existing datasets for human body segmentation from depth images. The data generation pipeline is capable of generating different body types and poses and thus creating a varied and large database of depth and segmented images.

The second step of this work is the implementation of the registration pipeline. As we have already discussed, it's main aim is to register the segmented body part with the volumetric data in order to find the transformation between CT data and Kinect model. The challenging part here is to achieve near perfect registration with minimum human intervention. Iterative closest point algorithm (ICP) [12] or its variants can be used for achieving this if we have a good initial estimate. Which brings us to the question how can we get this good initial estimate. We use correspondence matching strategy [1] for getting the initial estimate and then refining this initial transformation using ICP [12]. In order to be able to apply this registration pipeline we first need to obtain surface point cloud of volumetric medical data. We use raycasting [19]for obtaining this point cloud.

The technical steps included in this project are the following:

- Gather information on the general topic and pre-existing related work.
- Getting familiarized with the modelling tools such as makehuman[22] and blender [14].
- Using the above tools to create a pipeline that automatically generates posed human figure data in the form of rgb images and depth files. This data would be used for training neural network model that would be used for segmentation of body parts in the first part of the project.
- Generation of the point cloud from volumetric medical data.
- Successful registration of the volumetric data with the segmented body part.
- Development of a UI for user interaction and visualization of the final results.
- Creation of a concise and complete documentation.

2. Theoretical Background

Here, we will elaborate on some of the technical terms and concepts needed to understand the next few sections of this report.

2.1. Coordinate systems

There are several coordinate systems in which a vertex can be transformed to. You might be wondering what is the advantage of having so many coordinate systems or reference frames? The answer to this question is that some operations or calculations are more efficient and easier is one coordinate system than in others. There are a total of 5 different coordinate systems that are of importance to us. We can use different transformation matrices in order to switch between these coordinate systems. A complete flow of how to switch from one reference frame to another is shown in Figure 2.1

- **Local space / Object space** - This coordinate system is local to your object. It has it's own local origin. For example while using a modelling software such as Blender [14] for creating models, all the models have same origin irrespective of the position where they are placed in the scene. All the vertices of your model are therefore in object space.

- **World space** - This coordinate space is global in terms of the different objects placed in the scene. It has a common origin with respect to which all the objects are placed in the scene. Therefore world coordinates are relative to a global origin of the world. Without a world origin all the objects would have stacked on top of each other. You can transform the local coordinates of your object into world space coordinates using model matrix. The model matrix is a transformation matrix that translates, scales and/or rotates your object.

- **View space / Eye space / Camera space** - In this space, the coordinates

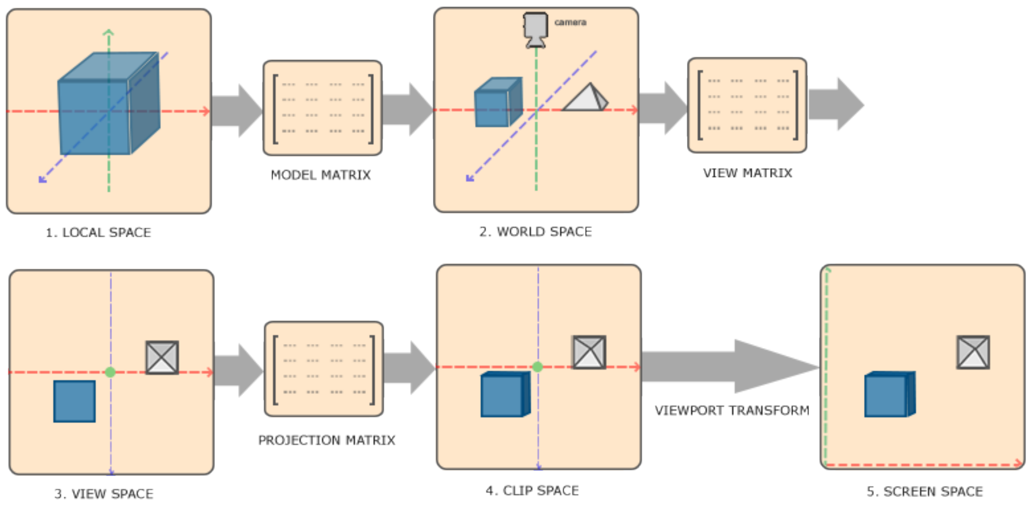


Figure 2.1.: Proposed System - This image shows how a particular coordinate system can be transformed to another using different camera matrices. [28]

are with respect to the camera or viewer's point of view. The position of the camera is considered as the origin of this space and coordinates of all other objects in the scene are transformed with respect to this origin. This transformation from world space to the camera space can be accomplished using translation and rotation of the objects. The combined transformation matrix is know as view matrix.

- **Clip space (this is important for OpenGL)** - This space determines which coordinates will end up on the screen after rendering. It processes the coordinates such that it range between -1 and 1. At the end of each vertex shader run OpenGL [3] expects the coordinates to be within a certain range [-1, 1]. We define a projection matrix to transform view space coordinates to clip space.

- **Screen space** - This is the on screen representation of coordinates. We use camera intrinsics to transform coordinates from view space to screen space.

We have briefly described what each reference frame represents and how we can transform from one frame to another. One of the main reason for using different reference frames is that some operations are easier in certain coordinate systems. For example it is easier to modify objects in local space, while it is more sensible to

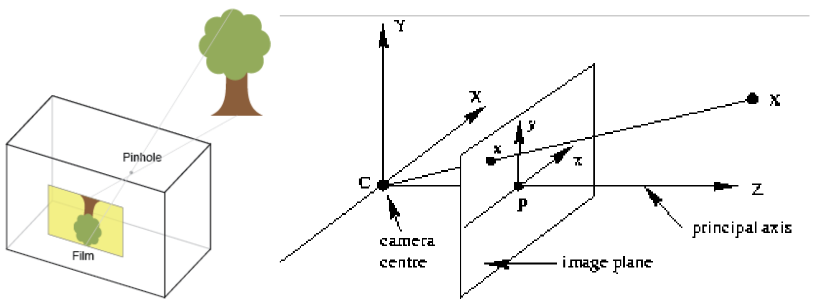


Figure 2.2.: Pinhole camera model [24] [23].

translate objects with respect to other objects in the world space. For more detailed information you can checkout this link [29].

2.2. Camera Intrinsics and Extrinsics

A basic way to model a camera with perspective projection is by using a pinhole camera model as illustrated in Figure 2.2

A 3x4 camera projection matrix is used to define the pinhole model. The camera projection matrix provides a mapping between homogeneous 3D world coordinates and homogeneous 2D image plane coordinates. Mathematically we can write it as x = PX, where x is homogeneous 2D coordinate, X is homogeneous 3D coordinate and P is the camera projection matrix.

The camera projection matrix can be decomposed into intrinsic and extrinsic matrices. Each intrinsic parameter describes a geometric property of the camera like - focal length, principal point, skew and distortion parameters. An intrinsic matrix can be represented as show below.

$$K = \begin{bmatrix} \alpha_x & \gamma & u_0 & 0 \\ 0 & \alpha_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This is what each of the above parameters mean :

- α_x , α_y : Represent focal length in terms of pixels. Here $\alpha_x = f_x.m_x$ and $\alpha_y =$

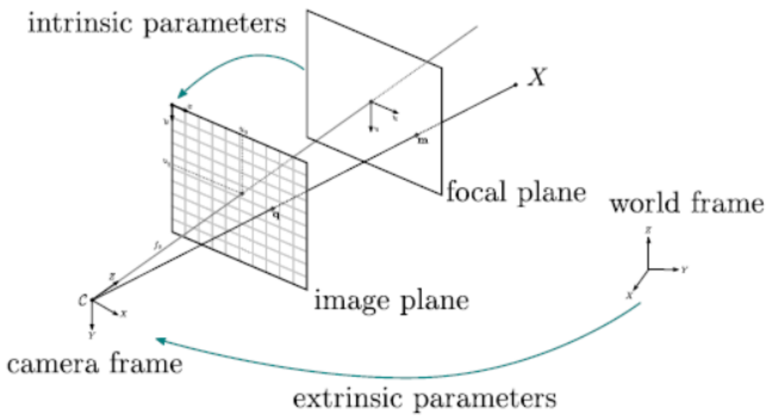


Figure 2.3.: Intrinsic matrix transforms 3D coordinates from the camera view to the 2D coordinates on the image plane. Extrinsic matrix transforms 3D coordinates from the world frame to the camera frame

$f_y.m_y$, where m_x and m_y are scale factors relating pixel to distance. f_x and f_y are focal length in terms of distance.

- γ : Represents the skew coefficient between the x and the y axis. It is often 0.
- u_0 , v_0 : Represents the principal point. Ideally it should be the center of the image.

Intrinsic matrix transforms 3D coordinates from the camera view to the 2D coordinates on the image plane as seen in Figure 2.3.

Extrinsic matrix define camera pose i.e rotation and translation of the camera with respect to the world coordinate system. An extrinsic matrix can be represented as $\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$. Here R represents rotation and t represents translation. Extrinsic matrix transforms 3D coordinates from the world frame to the camera frame as seen in Figure 2.3.

We can generate camera projection matrix by combining intrinsic and extrinsic matrices $P = K.\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$.

Important Note : In a normal camera model a camera looks along the positive z-

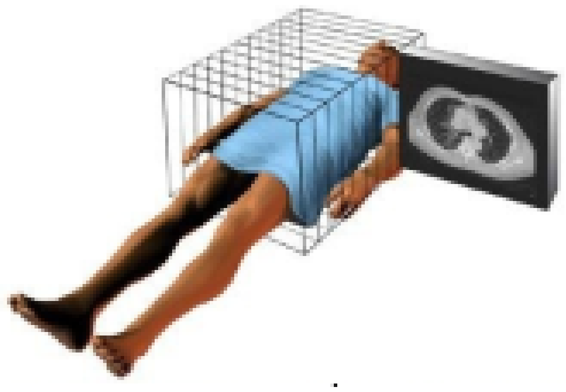


Figure 2.4.: This diagram represents CT scan data of a patient

axis as seen in the second image of figure 2.2. In case of OpenGL, camera by design is fixed at origin, always looking along the -z axis. We had to make appropriate changes in our camera intrinsics for handling this difference.

2.3. Volumetric CT Data

There are different ways to obtain 3D volumetric scans of a patient, for example - MRI, CT, PET etc. In this project we mainly experimented with data obtained from CT scans. Even CT data is available in variety of formats like - RAW, DICOM, MHD etc. For this project we have used RAW, MHD formats.

Generally speaking CT data is only a 3D matrix with some intensity values. CT data consists of multiple equispaced slices of grayscale images. The CT scanner goes through the patient and assign each voxel in each slice certain intensity value, depending on the x-ray attenuation. This x-ray attenuation is material dependent. Since skin, flesh, bones, air etc are made of different materials and have different x-ray attenuation, so each of them get different intensity values. By thresholding on a particular intensity value and using the (x,y,z) coordinates of the voxels in every slice we can define any surface of the volumetric data. In this project we have used raycasting [19] for rendering and generating the surface point cloud from CT data.

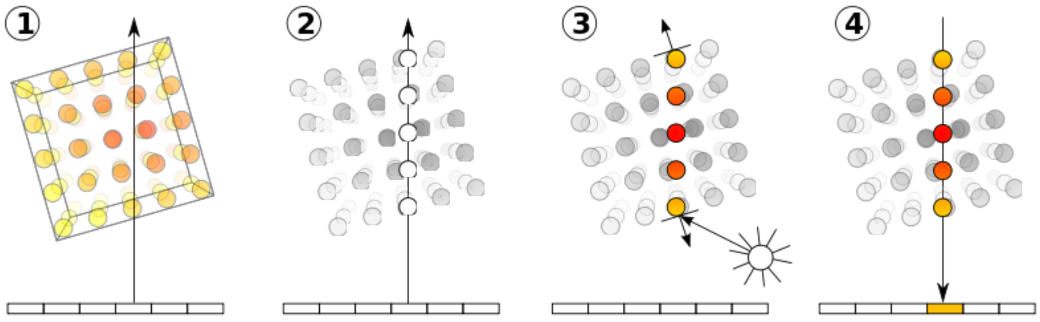


Figure 2.5.: Four basic steps of volume ray casting [8].

2.4. Volume Ray Casting

[19] [8] It is an image based volume rendering technique. It used to render high quality images of volumetric data. The advantage of this method is that it let's you view the 3D volume data from any view point. One of the major application of this method is rendering of medical volumetric data such as Computed Tomography (CT) data, MRI data etc.

There are four basic steps of volume ray casting.

- **Ray casting** - In this step rays are projected from a starting point towards the volume. Inorder to do this efficiently a bounding box or cube is fit to the volume. By rendering the front face and back face of the bounding box separately in different textures we can sample the starting and the ending point of rays.
- **Sampling** - From the starting and ending point of a ray we can compute it's direction. We now sample some points along the ray path by iteratively taking small steps in the direction of the ray.
- **Shading** - With the help of a transfer function, intensity value of each sampled point is mapped to an RGBA value. Different transfer functions help in defining different iso surface of the volume.
- **Composting** - After shading every sample point in the ray path, they are composited resulting in the final colour value for the pixel that is currently

being processed.

For our project requirements we just wanted to render the surface of the volume. Essentially we used these steps only but with slight simplifications and modifications. We will talk about the details when we describe the pipeline.

2.5. Registration

[32] It is the process of aligning point clouds into one consistent coordinate system. It's goal is to either find the relative transformation between two point clouds to align them with each other or to find the relative transformation of point clouds with respect to some global coordinate system. Registration can be rigid or non-rigid. A rigid registration results into rigid transformation consisting of translation and rotation. A non rigid transformation consists of scaling and shear mapping in addition to translation and rotation. This problem of registering point clouds is sometimes referred as pairwise registration. The steps performed in a pairwise registration are shown in Figure 2.6

2.5.1. Keypoint Estimation

From the set of points we need to identify some interest points that best represent the respective point clouds. These point are called keypoints and the process is called keypoint estimation. There are different algorithms that can be used for keypoint estimation such as NARF [10], SIFT [21] and FAST[26]. Alternatively, we can take every point or a subset of points by uniformly sampling from the entire set. We can even apply some sort of voxelization on the entire point cloud and then take centroid of all the voxels as the keypoints.

2.5.2. Descriptor Estimation

In order to define some sort of similarity and dissimilarity measure between the selected key points of a pair of point clouds we need descriptors. These descriptors are generally a vector of features that defines a keypoint. Basically the descriptors helps us to compare keypoints. There are a lot of descriptor estimators like - NARF[10],

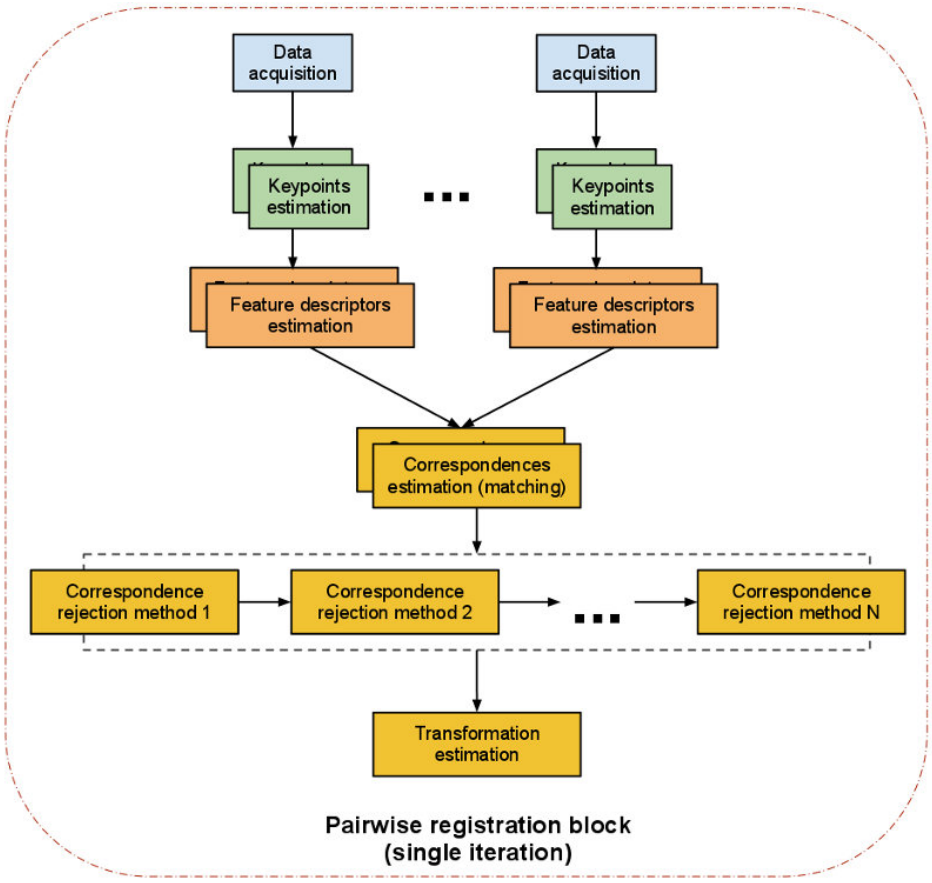

Data acquisition
Data acquisition
Keypoints estimation
...
Keypoints estimation
Feature descriptors estimation
Feature descriptors estimation
Correspondences estimation (matching)
Correspondence rejection method 1
Correspondence rejection method 2
...
Correspondence rejection method N
Transformation estimation
Pairwise registration block (single iteration)

Figure 2.6.: Pairwise registration steps[31].

FPFH[30], BRIEF[25], SIFT[21], SHOT[34].

2.5.3. Correspondence Matching

Using feature descriptors that we defined, we need to find pair of corresponding or similar keypoints in the point clouds. There are different algorithms that can help us in finding the correspondences - brute force matching, kd-tree nearest neighbor search (FLANN)[17] etc. Nearest neighbour is defined based on the euclidean distance of feature vector and not just the euclidean distance of keypoint coordinates.

2.5.4. Correspondence Clustering / Correspondence Rejection

Data is generally noisy and so not all correspondences are valid correspondences, some of them can be false positives. These false positives can negatively affect the estimation of final transformation so we need to remove or reject them before we start calculating the final transformation. For this we can use correspondence clustering or grouping algorithms like - Hough[36], GC[15]. We can also use RANSAC[16].

2.5.5. Transformation estimation

In the final step we actually need to compute the transformation using the filtered correspondences. For this we evaluate some error metric on correspondences. We find an initial estimate of the transformation and minimize the error metric using this estimate in order to find better estimates. We keep repeating these steps until we have reached some kind of convergence criteria. Algorithms such as SVD[37], Levenberg-Marquardt[35] are used for minimizing the error metric.

2.5.6. ICP

Iterative closest Point (ICP)[12] is an algorithm employed to minimize the difference between two point clouds given an initial estimate of the relative transformation. It is generally used as a refinement step in registration for obtaining a better estimate of the final transformation . ICP has several steps and each step may be implemented in various ways which gives rise to a multitude of ICP variants. Following are the generally performed steps : Selection - A set of points are selected from the point clouds. Selecting more points than less usually leads to a more accurate registration but at the cost of execution speed.

- **Matching** - Selected point from the source cloud are transformed via the initially estimated transformation and matched to the points in the target cloud. The aim here is to pair up points in the two point clouds that actually both project to the same point in the real world model.
- **Weighing** - The matched points are accordingly weighted. A higher weight is often assigned to a more undesirable pair of matched points. Alternatively, you can weight all the matched pairs uniformly.
- **Rejection** - Of the matched point, we reject the outliers. We may discard the pairs that are separated by a distance greater than some threshold.
- **Error metric** - We select an error metric for evaluating the matched pairs. For example, a point to point distance metric, or a point to plane distance metric.
- **Optimization** - We must also select an optimization technique for minimizing the error metric for the corresponding pairs from the source and target point clouds. For example we may use SVD[37], Levenberg-Marquardt[35].

You must have observed that the steps for correspondence matching strategy and ICP are very similar to one another. The big difference here is that for ICP we just use a point's 3D coordinate for defining correspondence. Whereas in case of correspondence matching strategy we use feature descriptor vector of a point for defining correspondences. This descriptor may or may not take the 3D coordinate of the point into account. The other difference is that ICP needs a good initial estimate (i.e the point clouds need to be sufficiently aligned) in order to generate good results.

3. Technical Details

In this chapter, we will provide a brief overview of the different libraries and frameworks used for the development of this project. We would further elaborate the system in detail and share the results. We would also talk about the challenges faced by us during the development of the project.

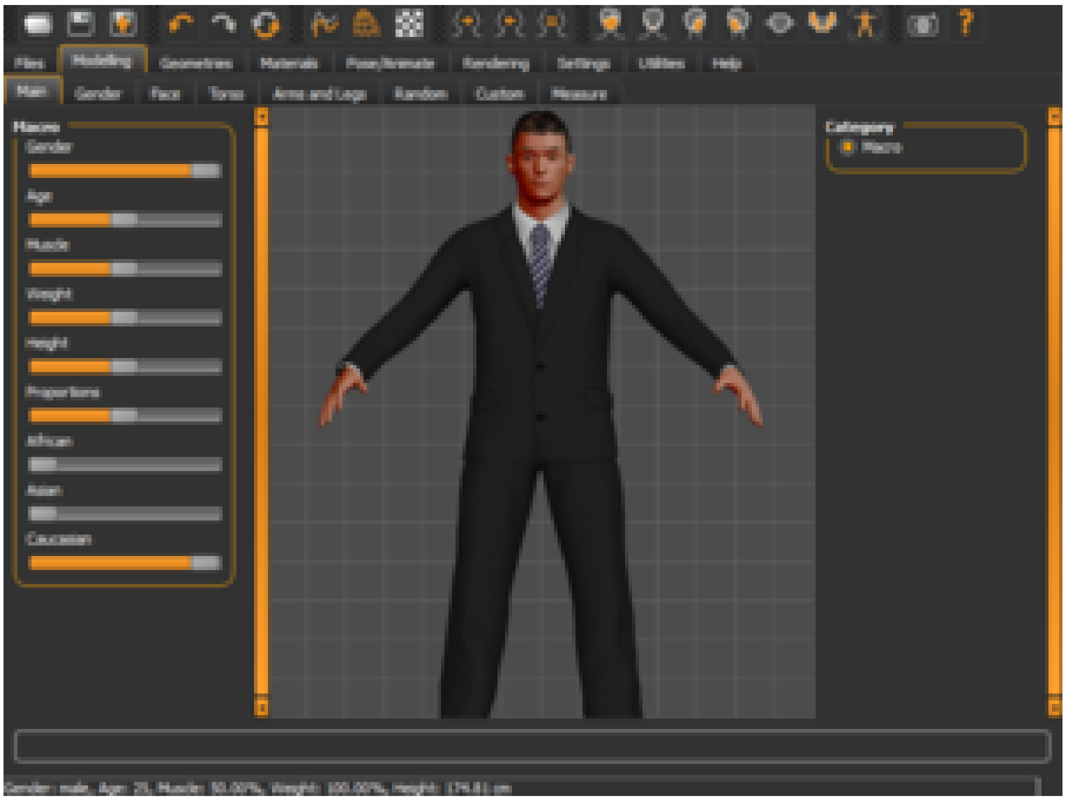
3.1. Overview of Frameworks and Libraries

Following are the libraries and frameworks used for the implementation of the system

- **Makehuman**[22] - It is an open source 3D computer graphics software. It is mainly used for designing realistic looking 3D human models. It has a fairly intuitive GUI. It also provides a python console where you can run your scripts. There's not a large community who is doing scripting on makehuman. API documentation is also not very elaborative. We are using it in our data generation pipeline for generating random variations of some template 3D human model.

- **Blender**[14] - It is one of the most feature rich open source 3D computer graphics software. Its features include 3D modeling, texturing, rigging and skinning, rendering etc. It also features an integrated game engine. It is generally used for creating visual effects, 3D interactive applications, video games etc. You may find the GUI a bit complex at first. With so many features it may feel a bit overwhelming at first. It also provides python console for running scripts. The API documentation is fairly good and elaborate. We are using it for rendering RGB and depth images of a human model from multiple views.

- **OpenGL**[3] - Open Graphics Library is an API for rendering 2D and 3D vector graphics. It is not a platform or a software, it's a library. So it provides



Gender: male, Age: 25, Muscle: 50.00%, Weight: 100.00%, Height: 174.81 cm

Figure 3.1.: Screenshot of Makehuman GUI[22]

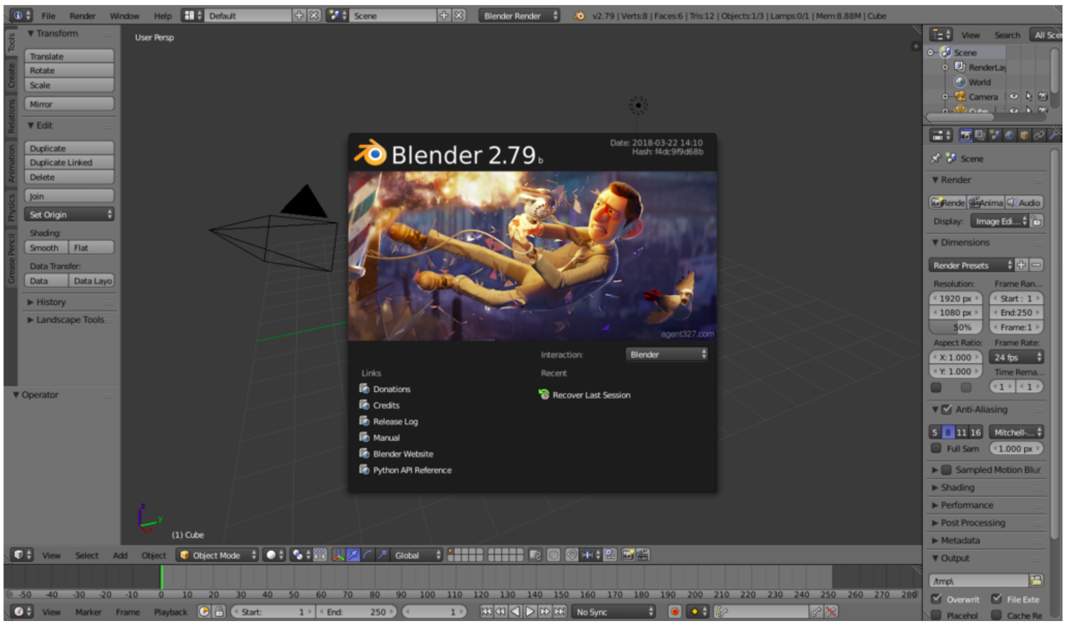


Figure 3.2.: Screenshot of Blender GUI[14]

us with functions which we can use to manipulate images and graphics. The implementation of these APIs are generally provided by graphics card manufacturers. These APIs are available in c++ but now you can also find a python wrapper for it. We are mainly using openGL for rendering of medical volumetric data from different views. A good resource to start learning about openGL is this - https://learnopengl.com/Getting-started/OpenGL

- **GLUT**[6] - We need to create an openGL context and an application window before we can start modifying graphics. These operations are specific per operating system, so OpenGL tries to abstract from these operations. It is for this reason that openGL utility toolkits like GLUT, SDL, SFML and GLFW are used. They help to create an OpenGL context and an application window to draw in. We are using GLUT in our project. You can read more about Glut here - https://www.opengl.org/resources/libraries/glut/

- **GLEW**[4] - OpenGL implementation are generally platform dependent. OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. It's goal is to load the appropriate implementation of openGL functions that an application needs. It provides an efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. You can read more about GLEW here - http://glew.sourceforge.net/

- **PCL**[7] - It is an open source library for the processing of point clouds and images. It has a very strong community base and is widely used. A lot of algorithms for processing point clouds are already pre implement in it. PCL framework contains numerous state-of-the art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. We are using PCL mainly for implementing our registration pipeline. For more information check this link - http://docs.pointclouds.org/trunk/

- **GLUI**[5] - It is an openGL user interface library based on GLUT. The interface is written in c++. It lets you add controls such as buttons, checkboxes, radio buttons, and spinners to OpenGL applications. It is window- and operating system independent, relying on GLUT to handle all system-dependent issues, such as window and mouse management. It doesn't support a lot of GUI related features. It just contains bare minimum features necessary to create some sample GUI.

- **ITK**[2] - It is an open source framework generally used for development of segmentation and image registration programs. In our project we are mainly using ITK for reading CT scan files.

3.2. Data Generation System

The first step of this work is to implement the data generation pipeline for generating training data that can be used by the segmentation network. The main motivation for this is scarcity of existing datasets for human body segmentation from depth images. The data generation pipeline is capable of generating different body types and poses and thus creating a varied and large database of depth and segmented images.

The flow of the pipeline can be seen in Figure 3.3. In order to scale the data creation process we made the pipeline as automatic and flexible as possible. In this pipeline we are using two open source modelling and graphics softwares - Makehuman[22] and Blender[14].

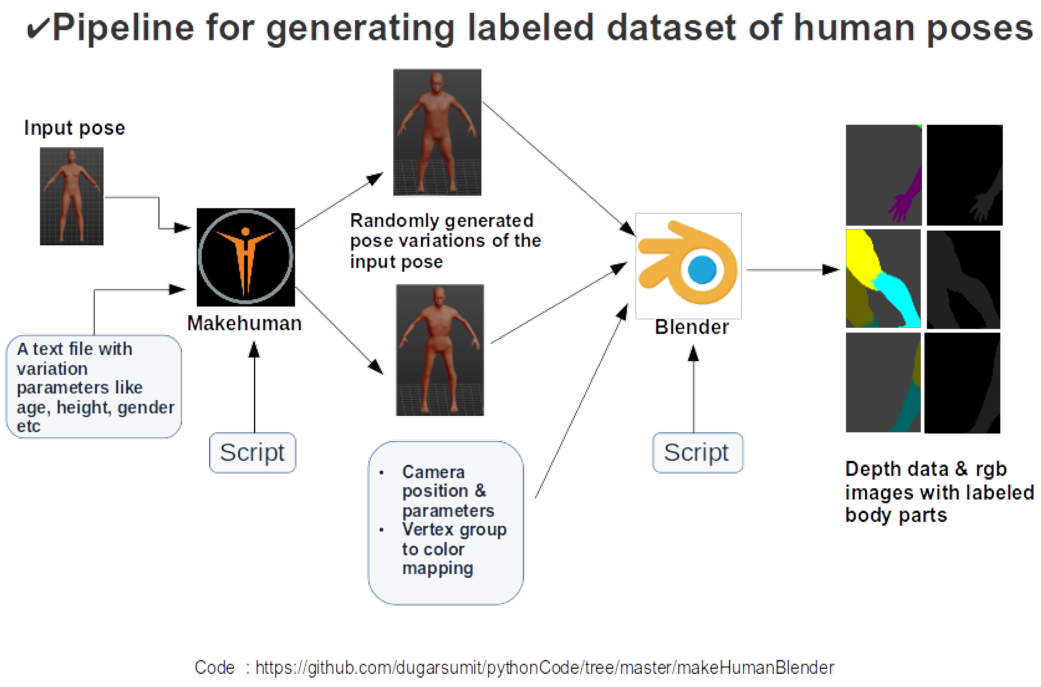

✔Pipeline for generating labeled dataset of human poses
Input pose
Makehuman
A text file with variation parameters like age, height, gender etc
Script
Randomly generated pose variations of the input pose
• Camera position & parameters
• Vertex group to color mapping
Blender
Script
Depth data & rgb images with labeled body parts
Code : https://github.com/dugarsumit/pythonCode/tree/master/makeHumanBlender

Figure 3.3.: Data generation pipeline

3.2.1. Makehuman Pipeline : Generate Random Human Models

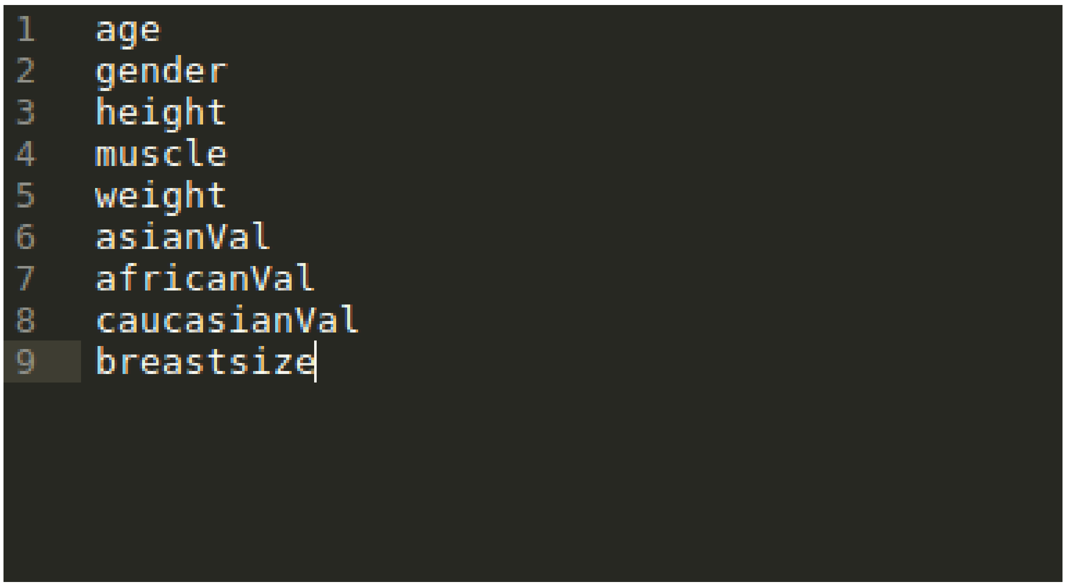
First we manually create a base template human model using Makehuman[22]. We save this model in .mhm format and use a default skeleton for creating this model. It is important to use the default skeleton because other skeleton models are either too detailed or lack enough information. Our script takes this template and some configuration file as input and randomly generates different variations of this template as output. The output variations are generated in .dae format. We are specifically using this format because with other formats we were not able to access skeleton vertex groups in the Blender[14]. We need these vertex groups in order to be able to color body parts differently. The variations are generated by randomly varying certain parameters of the base template model. These parameters include age, gender, height, muscle etc. The configuration file helps to control the variation parameters. If you want your output to be generated based on the variation of a certain parameter then, you need to specify that parameter in the configuration file. The values of these parameters are randomly set by our script. A sample configuration with possible parameters to choose from is shown in Figure 3.4. If you don't want your variations to be dependent on certain parameters then, just remove them from this configuration file and a default value from the base template will be picked for that parameter.

We couldn't find a way to directly call makehuman functions from our script. So in order to make this step work properly you would need to copy the generatePoses.py script and run it inside Makehuman python console. Make sure to edit the input directory and configuration path accordingly before executing the script.

3.2.2. Blender Pipeline : Render RGB and Depth Data

After generating the human models the next step is to render RGB and depth images for each of these models from different views. We use Blender[14] for this step. As before our script for this step takes some configuration files and generated human models as input.

We setup a blender scene consisting of a human model, a camera and a HEMI lamp as the light source. We remove the default cube object and light source from the scene. We recreate this scene for each human model. Now for rendering RGB and depth data we fix the camera pose looking at the human model in the scene



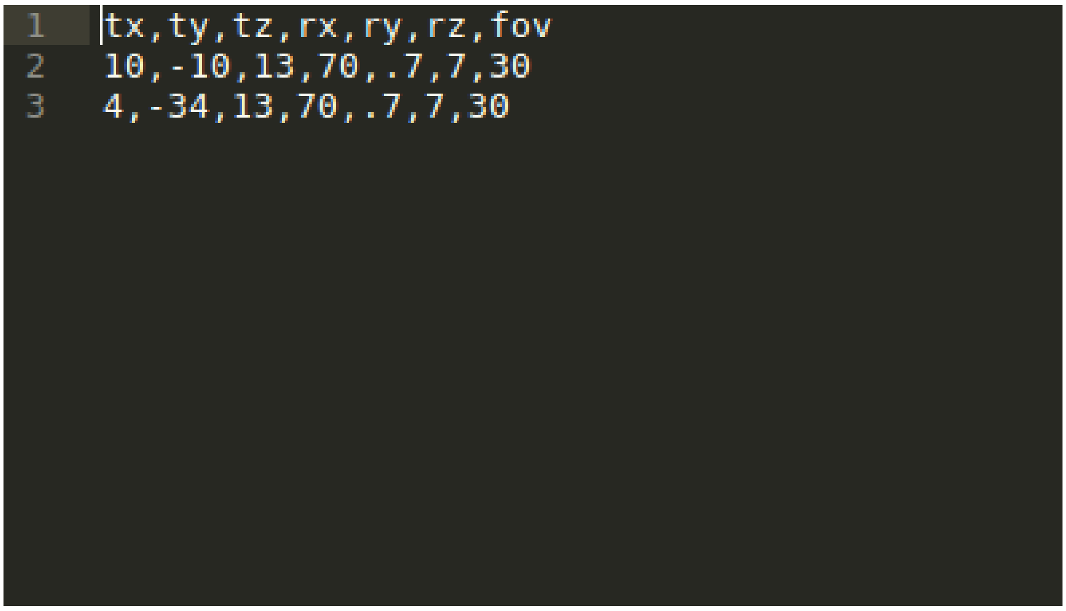
```
age
gender
height
muscle
weight
asianVal
africanVal
caucasianVal
breastsize
```

Figure 3.4.: Pose parameter configuration file. This file is used by makehuman generate pose script

and then revolve the camera around the human model. We revolve with a step of 1 degree and for each step we render images. The position of the camera can be configured with the help of a configuration file. In this file we specify camera pose parameters in the form of 3D coordinates and rotations of camera around the axis. A sample file can be seen in Figure 3.5. Multiple camera positions can be specified in the file. Each of these initial camera orientations are used with each human model for generating different views.

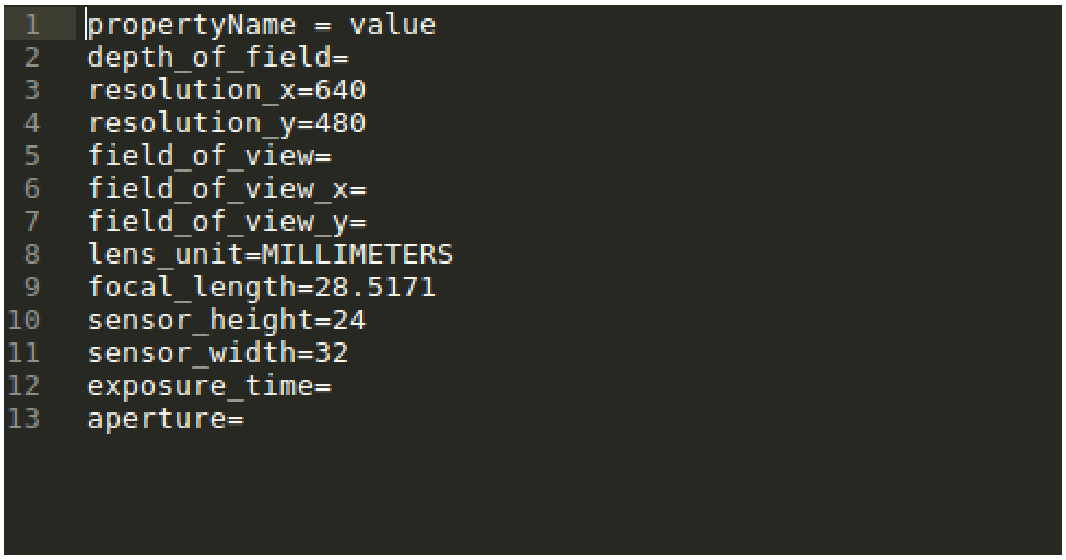
Another configuration file is used for specifying camera properties like focal length, resolution, sensor size etc. This file is only loaded once during the Blender pipeline. A sample camera properties configuration file can be seen in Figure 3.6.

Our aim is to create a labelled data set of human body that can be used for training the segmentation network. We use a coloring scheme for labelling each body part. Before rendering the RGB image each vertex of the human model is colored based on which vertex group it belongs to. There are around 164 vertex groups defined by each bone in the default skeleton that we used while creating the template in Makehuman[22]. We manually mapped each vertex group or bone to a body part (each body part was defined by a different color). There are 10 body



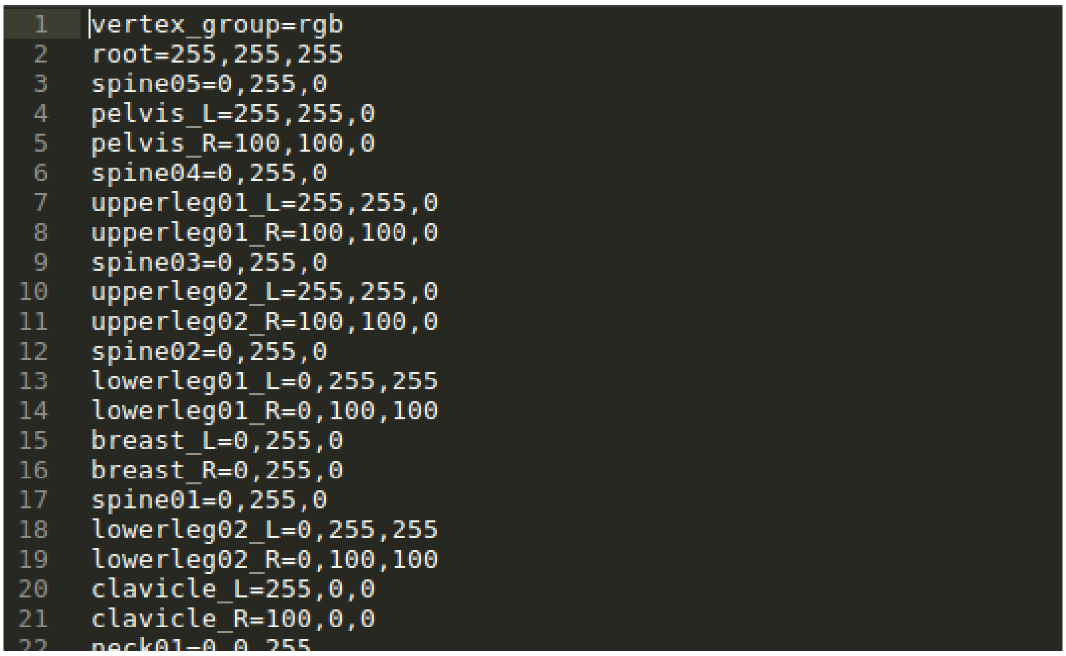
```
tx,ty,tz,rx,ry,rz,fov
10,-10,13,70,.7,7,30
4,-34,13,70,.7,7,30
```

Figure 3.5.: Camera position configuration file. This file is used for specifying the initial camera orientations in the blender scene.



```
propertyName = value
depth_of_field=
resolution_x=640
resolution_y=480
field_of_view=
field_of_view_x=
field_of_view_y=
lens_unit=MILLIMETERS
focal_length=28.5171
sensor_height=24
sensor_width=32
exposure_time=
aperture=
```

Figure 3.6.: Camera properties configuration file. This file is used for specifying camera properties like focal length, resolution, sensor size etc.



```
vertex_group=rgb
root=255,255,255
spine05=0,255,0
pelvis_L=255,255,0
pelvis_R=100,100,0
spine04=0,255,0
upperleg01_L=255,255,0
upperleg01_R=100,100,0
spine03=0,255,0
upperleg02_L=255,255,0
upperleg02_R=100,100,0
spine02=0,255,0
lowerleg01_L=0,255,255
lowerleg01_R=0,100,100
breast_L=0,255,0
breast_R=0,255,0
spine01=0,255,0
lowerleg02_L=0,255,255
lowerleg02_R=0,100,100
clavicle_L=255,0,0
clavicle_R=100,0,0
neck01=0,0,255
```

Figure 3.7.: Vertex groups configuration file. This file specifies a mapping between vertex groups and body parts. Each body part is defined by a different color. There are around 164 vertex groups.

parts that we used for mapping - head, torso, left upper arm, left lower arm, right lower leg etc. This vertex group to color mapping was loaded from a configuration file. A sample configuration file can be seen in Figure 3.7.

In order to execute the Blender[14] pipeline you will need to copy the generateViews.py script and run it from the Blender python console. The depth data is stored in .exr format. The good thing about this format is that it stores the depth values as it is without applying any normalizations.

3.2.3. Results

In Figure 3.8 you can see samples of the generated data.

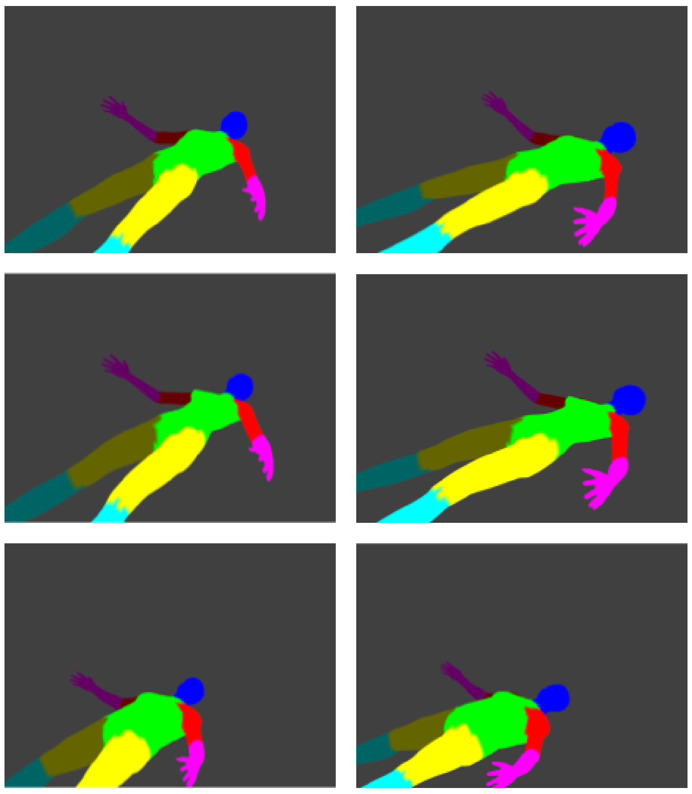


Figure 3.8.: Rendered RGB images of three different body models from two different views.

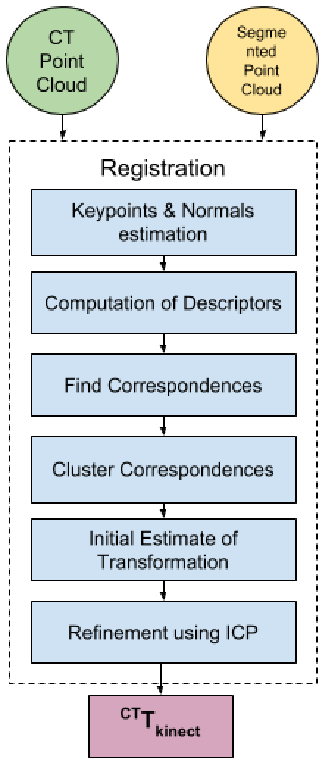


Figure 3.9.: Registration system block diagram.

3.3. Registration System

Registration is the process of aligning point clouds into one consistent coordinate system. The aim of the registration system is to obtain the transformation between CT data and the 3D Model obtained via Kinect Fusion[27]. A block diagram of the registration system is depicted in Figure 3.12. It consists to two pipelines namely - Volume rendering pipeline and Registration pipeline. Overall registration system requires volumetric CT data and segmented point cloud as input. We get the segmented point cloud of a body part from a segmentation system. In the first step we generate point cloud of CT data using volume rendering pipeline. Once we have both the point clouds we then, compute the transformation $^{CT}T_{Kinect}$ between CT data point cloud and segmented point cloud.

3.3.1. Volume Rendering Pipeline

An important part of the project was to develop a pipeline for generating point cloud from the volumetric data. Volumetric data such as CT scan generally consists of multiple equi-spaced slices of grayscale images having some scalar values in each voxels. These values are calculated based on x-ray attenuations. CT scan data can have variety of formats like RAW, DICOM, MHD etc. We will be mainly focusing on MHD data as this is the most commonly used format and many other modalities also support this format. CT data is generally speaking only a 3 dimensional matrix. A simple way of converting this data to point cloud could be by thresholding every voxel in every slice and store the corresponding (x,y,z) coordinates as point coordinates. This algorithm would be computationally expensive as we would need to check every voxel in the volume. A better solution would be to use volume rendering techniques such as ray-casting for rendering only the required surface and then extracting point cloud of this surface. A block diagram of the entire volume rendering pipeline is show in figure 3.10. The input to the pipeline is CT data in MHD / RAW format. We read this data and it's meta information using ITK[2]. Next initialize shaders and textures and render depth images of the volumetric data using OpenGL[3]. Simultaneously we also generate the point cloud of the CT data after applying some post processing on the depths that we get from the OpenGL rendering pass 2.

3.3.2. Initialization of Shaders and Textures

Firstly, we read the meta information of the volumetric CT data using ITK[2]. This meta information will be used later on in the post processing step. For volume rendering and ray casting we are using openGL[3]. We initialize the openGL context and application window with the help of GLUT[6]. Next we initialize shaders and textures that will be used in the rendering passes. We also define buffers for storing some of the intermediate rendering results. Overall we are using six shaders two for each pass. There are three textures that we are using. One of them is a 3D texture which is used for storing volumetric data. Other two are 2D textures, one is used for storing depth information and the other one is used for storing backface rendering of the bounding box.

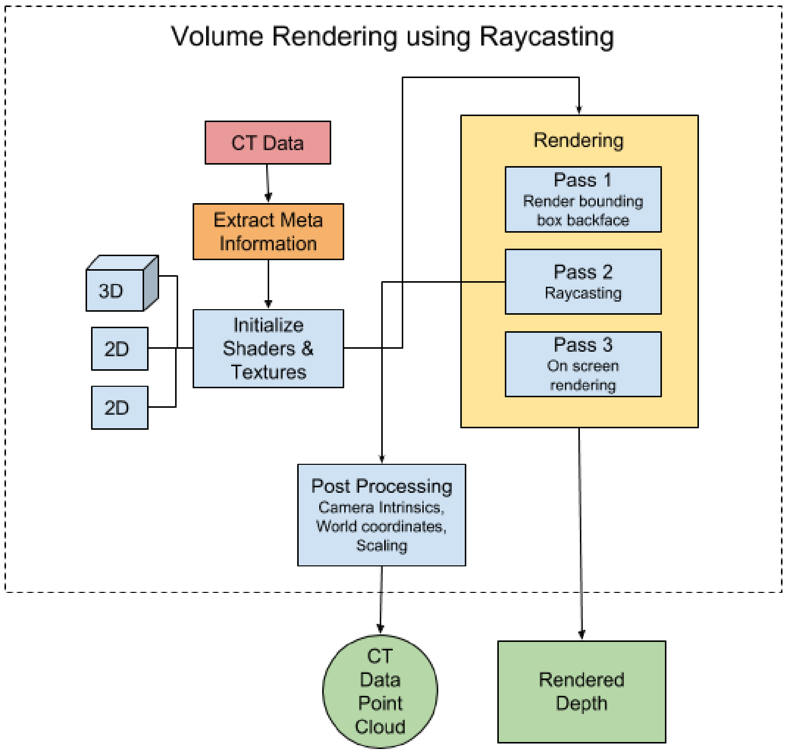
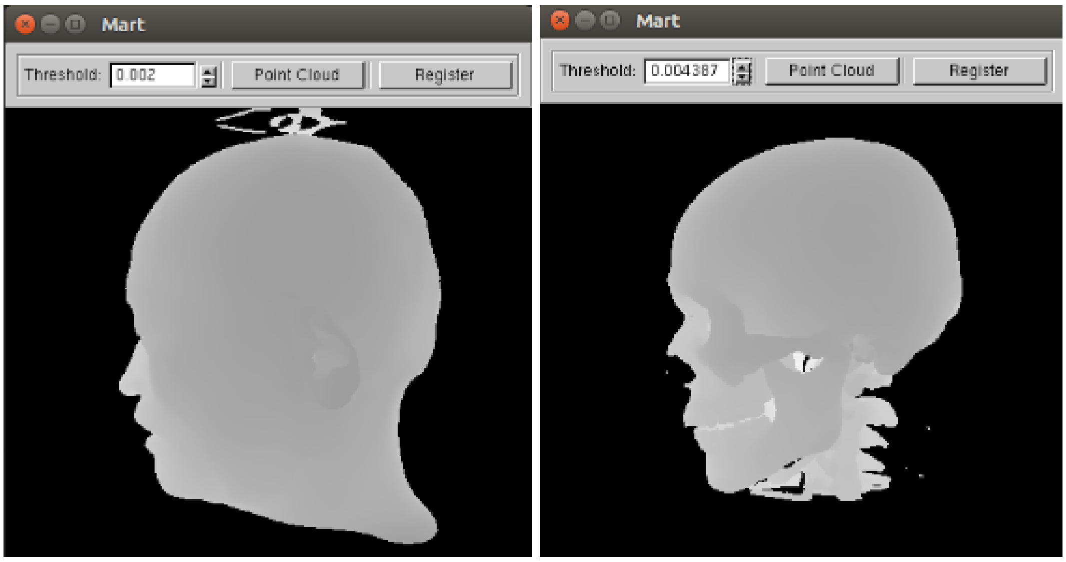


Figure 3.10.: Volume rendering pipeline.

3.3.3. Rendering using OpenGL

We perform the rendering in three passes. In the first pass, we render the backface of the bounding box to an offscreen buffer that has one of the 2D textures that we defined earlier attached to it. This texture will then be used in the second pass. In the second pass we perform the ray casting. The output of this pass is also written onto an offscreen buffer that has a 2D texture attached to it. The depth values of the surface points are saved in this texture. Entry and exit points of the rays for ray casting are sampled from the front face and back face of the bounding box respectively. Back face points were already saved in a 2D texture during the first pass. Using the entry and exit point we can compute the ray marching direction. Iteratively we take small steps in the direction of the ray path. At each step we compare the intensity of the voxel value with some predefined threshold. We get these intensity values from the 3D texture that we defined earlier during the initialization stage. We terminate the marching of the ray if the voxel intensity is greater than the threshold or if the ray has reached it's exit surface. If the ray has been terminated before reaching the exit surface than the terminating coordinate defines the surface of the volumetric data. By varying the threshold we can generate different type of surfaces such as skin, bones etc from the volumetric data. We have also created a GUI where we can vary the threshold and see the resultant surface generated in real time. Figure 3.11 shows a screenshot of this GUI. In this the first image shows a skin surface generated with a threshold of .002 and second image shows a bone surface generated with a threshold of .004. The length marched by a ray defines the depth for that surface point. These depth values are stored in a 2D texture which would be used later on for generating point cloud. In the third pass we just render the depth images in the application window.

In order to perform registration with the generated point cloud it has to be completely 3D. Rendering depth from one view and then using it to generate point cloud generally creates only 2.5D point clouds. So instead of generating point cloud from just one view we repeated this rendering process from 5 different views (left, right, front, back, top) and fused together the point clouds generated from each view. This gave us a completely 3D point cloud.



Threshold: 0.002

Threshold: 0.004387

Figure 3.11.: The first image shows a skin surface generated with a threshold of .002 and second image shows a bone surface generated with a threshold of .004.

3.3.4. Post Processing

In order to generate point cloud from the depth data that we had saved after the second pass we need to apply some post processing steps. The depth data is just like depth images i.e for every pixel values we have corresponding depths saved in the texture. We need to convert this data into 3D coordinates in some reference frame and this set of vertices will be our point cloud. We will use the camera intrinsic parameters to convert depth data from the image plane to the 3D coordinates in camera reference frame. An important thing to note here is that the intrinsic parameters are defined assuming that the camera is looking along the +z direction but in OpenGL camera by definition looks along -z direction. So before applying the intrinsic parameters make sure that you have adjusted the signs in the formula correctly. After modification you will get the following representation of the intrinsic parameters. Here coordinate (x,y,z) are camera frame coordinates for some image plane vertex. In image plane vertex x,y represents pixel coordinates and z represents the corresponding depth value. fx and fy represents focal length. cx and cy represents image plane center.

$$coordinate.z = \quad vertex.z \tag{3.1}$$
$$coordinate.x = \quad -((vertex.x - cx) * vertex.z)/fx \tag{3.2}$$
$$coordinate.y = \quad -((vertex.y - cy) * vertex.z)/fy \tag{3.3}$$

Next we need to convert all the camera frame coordinates to the world frame coordinates so that different point clouds can be fused together. We will use inverse of the view matrix for transforming coordinates from camera frame to the world frame. Finally we will use the meta information of the CT data like - voxel dimension and spacing to bring the generated point cloud upto scale with the actual CT data.

3.3.5. Registration Pipeline

As already stated the aim of this pipeline is to calculate the transformation between the CT data and the 3D scan from KinectFusion. The pipeline expects CT point cloud and segmented point cloud as input and calculates ${}^{CT}T_{Kinect}$ as output. A block diagram of the registration pipeline is show in Figure 3.12.

The entire registration process happens in two steps. We first find a good initial estimate of the transformation. For this, we use a correspondence matching strategy. Then we refine this estimated transformation using ICP[12] in order to obtain the final transformation. This pipeline is implemented in C++ using PCL.

3.3.6. Initial Estimate

We use correspondence matching strategy for calculating the initial estimate. Firstly we downsample the segmented point cloud. CT data point cloud is zero centered whereas segmented point cloud is not. The amount of translation required to align the two point clouds can be approximated with the centroid of the segmented point cloud. So, we will use this centroid while applying the final transformation. Next we need to identify rotation between the two point clouds. For this we first zero center the segmented point cloud so that both the cloud are misaligned only by some rotation. Now we define key points for both the clouds. We use simple key point voxelization technique where centroid of the voxels are considered as key points. We also estimate the normals on these keypoints. Next we compute descriptors

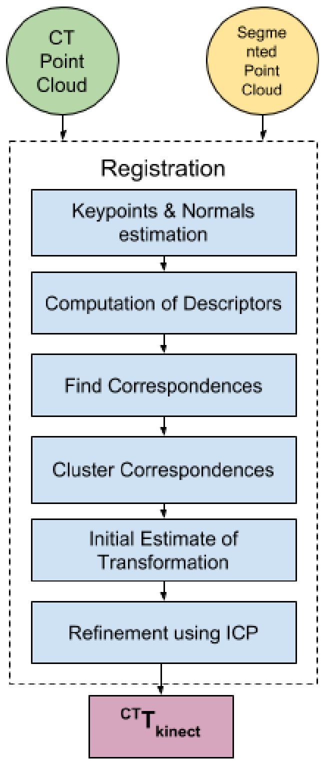


Figure 3.12.: Registration Pipeline.

for these keypoints using SHOT descriptor algorithm[34]. The descriptors helps to define a similarity measure between the selected keypoints. Next step is to find the correspondences and for this we use a distance measure on the keypoint descriptor vector. For every keypoint in one point cloud we find the nearest neighbour in the other point cloud by comparing euclidean distances between the descriptor vectors on the keypoints. If the nearest neighbour is near enough i.e the distance is less than certain threshold then we consider this pair as a correspondence pair. Data is generally noisy and so not all correspondences are valid correspondences, some of them can be false positives. These false positives can negatively affect the estimation of final transformation so we need to remove or reject them before we start calculating the final transformation. For this we use geometric consistency clustering algorithm (GC) [15]. With these filtered correspondences we compute an initial estimate of the required transformation. As a result of grouping we get multiple possible estimates one for each possible group.

3.3.7. Refinement Using ICP

After getting a group of initial estimates for the required transformation we try to refine these estimates using ICP[12]. We run about ten iterations of ICP. In order to register two point clouds using ICP, the two clouds should be close enough. So before actually applying the ICP we transform one of the clouds using the initial estimate of the transformation that we have calculated in the previous step. If we are lucky enough we will get some final transformations that are good enough to algin are two point clouds. The final alignments of the point clouds using the final transformation that we got after applying ICP is show to the user. The best transformation that the user select's we be defined as ${}^{CT}T_{Kinect}$ and will be used for further processing. A simple GUI as shown in figure 3.13 is presented to the user and the user is asked to press the key number corresponding to the best alignment.

3.3.8. Alignment with the Body

The selected transformation is used to align the CT data point cloud with the entire body point cloud and not just the segmented point cloud of a body part. In our project as an example we try to align CT data of a head with the body. If you remember, we described earlier that the final transformation in our case only estimates the rotation that is required to align the two point clouds. This is because

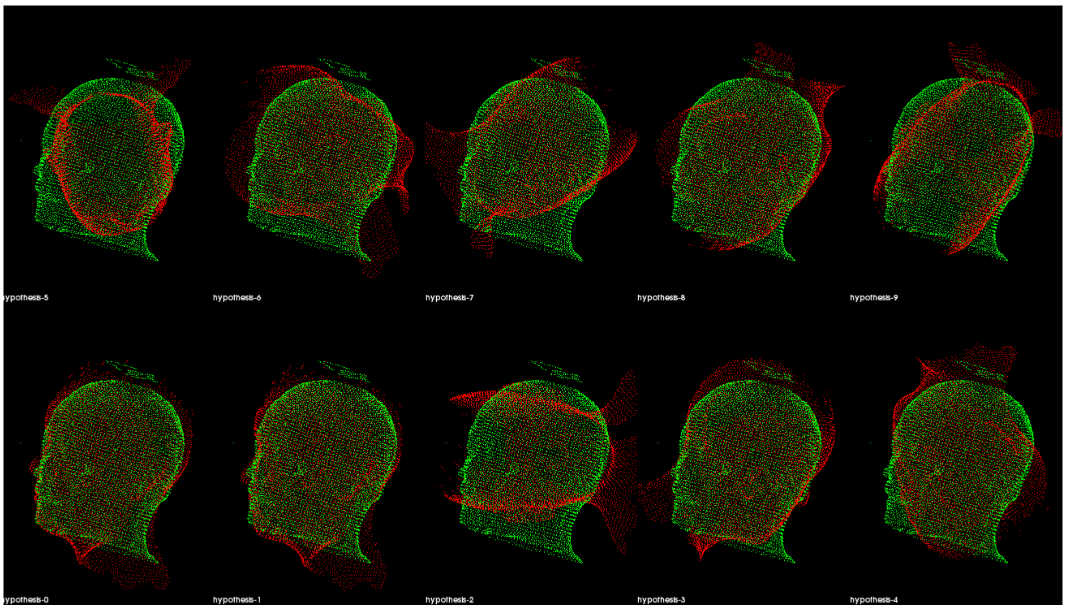


Figure 3.13.: Screenshot of the GUI showing head alignments with different estimated transformations after the ICP has been applied. A user interaction is required here in order to select the best transformation.

we computed this transformation after zero centering both the point clouds. But in order to properly align both the clouds we also need an estimate of the translation. As told earlier, we can compute the required translation using the centroid of the segmented head cloud. Using this translation and rotation we are able to successfully align the body with the head. The results can be seen in figure 3.14. The green colored head in the images is from the CT data whereas the red colored head is part of the body. An important point to note here is that two point clouds belong to two different persons. The point cloud of the body is generated from some synthetic human model whereas CT data is of some real person.

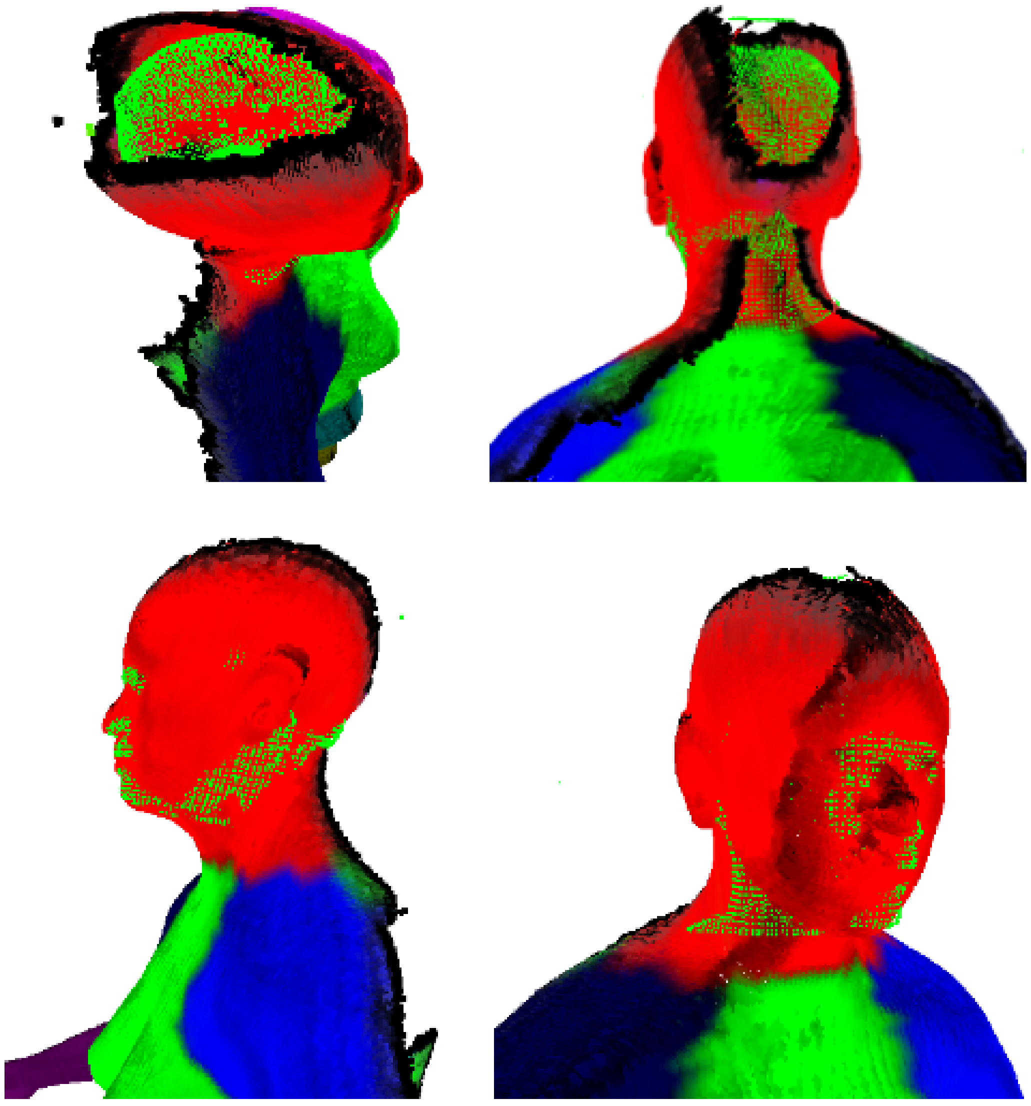


Figure 3.14.: Image showing alignment of entire body's point cloud with the point cloud obtained from CT data. The head in the red color is part of the actual body and the head in the green color is obtained from the CT data.

4. Future Work

4.1. Calculate the transformation ${}^{Kinect}T_{HMD}$

Transformation ${}^{CT}T_{HMD}$ (see Figure 4.1) is what we finally need for augmenting Volumetric CT Scan data. We can find this transformation indirectly using relations 1.1 and 1.2.

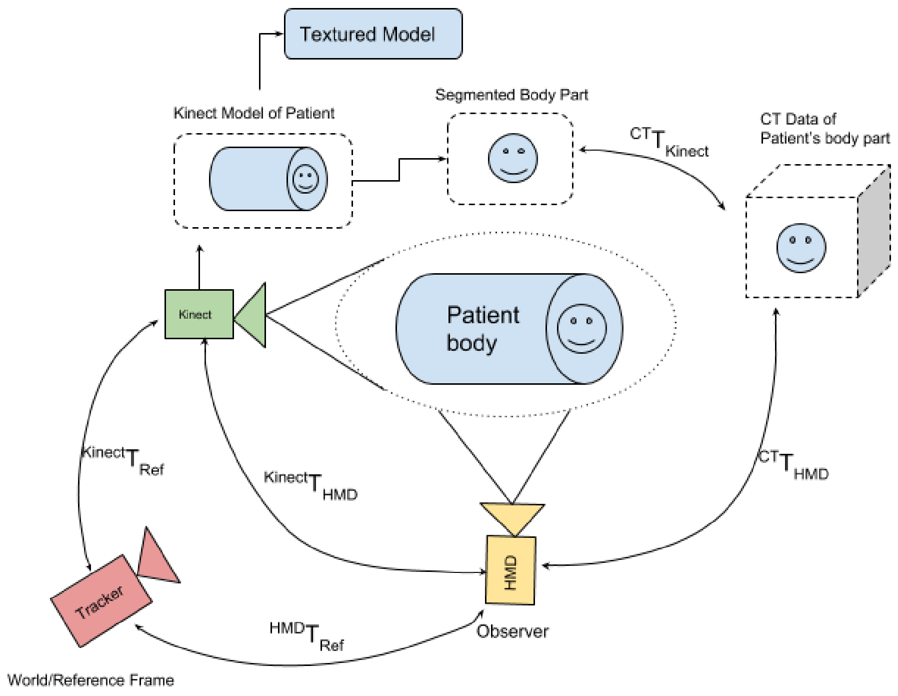


Figure 4.1.: Required Transformations between different frames.

From the help of this project we can find ${}^{CT}T_{Kinect}$, but the transformation ${}^{Kinect}T_{HMD}$ is still missing. We can find this transformation by applying tracking on Kinect and HMD and then using the second relation.

4.2. Compare with other registration algorithms

In our correspondence matching step, we currently choose our keypoints via voxelization which is similar to uniformly sampling points from the point cloud. Some other approaches for keypoint extraction that are worth trying out could be approaches like SIFT[21], SURF[11], ORB[33] etc. Many of these approaches have the added benefit of being scale and shift invariant and thus are ideal for extracting points for correspondence matching.

Also, it could be worth our while to experiment with descriptors other than SHOT[34].

4.3. Augmentation of medical data on patient's body

In the first section of this chapter, we discussed how to get the transformation ${}^{Kinect}T_{HMD}$. Using this transformation, we can actually augment the volumetric data onto the patient. However, augmenting this data on the patient does not simply mean superimposing it on the patient's 3d reconstruction. The actual implementation of the augmentation step will have to be handled keeping in mind several points, for instance, what kind of an HMD we are using, what does the user actually want to focus on, which areas of the scan/patient reconstruction should be shown and what should be occluded.

5. Conclusion

In this work therefore, we have presented the following:

- A configurable and flexible method to generate depth-rgb pairs using the softwares makeHuman and Blender. These data pairs will be used by the Segmentation module of the body project.
- A user friendly and largely automatic method to register a 3D volumetric scan with the 3D point cloud of a patient. This method will require minimal user input for registration.

A. Additional Information

A.1. Source Code

Entire source of this project can be accessed from the following repositories :

- Data Generation pipeline - https://gitlab.lrz.de/ga87yiq/mart/tree/master/makeHumanBlender
- Registration pipeline - https://gitlab.lrz.de/ga87yiq/mart/tree/master/registration

A.2. Docker Image

A docker image with all the installed dependencies is available here :

- https://hub.docker.com/r/dugar/mart/

Bibliography

[1] 3d object recognition based on correspondence grouping. URL: http://pointclouds.org/documentation/tutorials/correspondence_grouping.php#correspondence-grouping.

[2] Insight segmentation and registration toolkit. URL: https://itk.org/.

[3] Open graphics library (opengl)[3][4] is a cross-language, cross-platform application programming interface (api) for rendering 2d and 3d vector graphics. URL: https://en.wikipedia.org/wiki/OpenGL.

[4] The opengl extension wrangler library. URL: http://glew.sourceforge.net/.

[5] Opengl user interface library. URL: https://sourceforge.net/projects/glui/.

[6] The opengl utility toolkit. URL: https://www.opengl.org/resources/libraries/glut/.

[7] Point cloud library. URL: http://pointclouds.org/.

[8] Volume ray casting. URL: https://en.wikipedia.org/wiki/Volume_ray_casting.

[9] Josep Aulinas, Yvan Petillot, Joaquim Salvi, and Xavier Lladó. The slam problem: A survey. In *Proceedings of the 2008 Conference on Artificial Intelligence Research and Development: Proceedings of the 11th International Conference of the Catalan Association for Artificial Intelligence*, pages 363–371, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press. URL: http://dl.acm.org/citation.cfm?id=1566899.1566949.

[10] Kurt Konolige Wolfram Burgard Bastian Steder, Radu Bogdan Rusu. Narf: 3d range image features for object recognition. URL: http://citeseerx.ist.

psu.edu/viewdoc/download?doi=10.1.1.231.1701&rep=rep1&type=pdf.

[11] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008. URL: http://dx.doi.org/10.1016/j.cviu.2007.09.014, doi:10.1016/j.cviu.2007.09.014.

[12] Paul J.; N.D. McKay Besl. A method for registration of 3-d shapes". ieee trans. on pattern analysis and machine intelligence. los alamitos, ca, usa: Ieee computer society. 14 (2): 239–256. doi:10.1109/34.121791, 1992. URL: https://ieeexplore.ieee.org/document/121791/.

[13] Christoph Bichlmeier. Contextual anatomic mimesis hybrid in-situ visualization method for improving multi-sensory depth perception in medical augmented reality, 2007. URL: http://campar.in.tum.de/pub/bichlmeier2007Mimesisismar/bichlmeier2007Mimesisismar.pdf.

[14] Blender. Open-source 3d computer graphics software. URL: https://www.blender.org/.

[15] Hui Chen and Bir Bhanu. 3d free-form object recognition in range images using local surface patches. *Pattern Recognition Letters*, 28(10):1252–1262, 2007.

[16] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981. URL: http://doi.acm.org/10.1145/358669.358692, doi:10.1145/358669.358692.

[17] FLANN. Fast library for approximate nearest neighbors. URL: https://www.cs.ubc.ca/research/flann/.

[18] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL: http://arxiv.org/abs/1704.04861, arXiv:1704.04861.

[19] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, January 1984. URL: http://doi.acm.org/10.1145/964965.808594, doi:10.1145/964965.808594.

[20] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987. URL: `http://doi.acm.org/10.1145/37402.37422`, doi:10.1145/37402.37422.

[21] David G. Lowe. Object recognition from local scale-invariant features, 1992. URL: `https://ieeexplore.ieee.org/document/790410/`.

[22] MakeHuman. Open source 3d computer graphics software. URL: `http://www.makehuman.org/`.

[23] Camera Matrix. Dissecting the camera matrix, part 1: Extrinsic/intrinsic decomposition, 2012. URL: `http://ksimek.github.io/2012/08/14/decompose/`.

[24] Intrinsic Matrix. Dissecting the camera matrix, part 3: The intrinsic matrix, 2013. URL: `http://ksimek.github.io/2013/08/13/intrinsic/`.

[25] Christoph Strecha Michael Calonder, Vincent Lepetit and Pascal Fua. Brief: Binary robust independent elementary features, 2010. URL: `https://www.labri.fr/perso/vlepetit/pubs/calonder_eccv10.pdf`.

[26] Vincent Lepetit Mustafa Özuysal, Pascal Fua. Fast keypoint recognition in ten lines of code, 2007. URL: `https://cvlab.epfl.ch/files/content/sites/cvlab2/files/publications/publications/2007/OzuysalFL07.pdf`.

[27] Richard A. Newcombe. Kinectfusion: Real-time dense surface mapping and tracking, 2011. URL: `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ismar2011.pdf`.

[28] OpenGL. Coordinate systems. URL: `https://learnopengl.com/Getting-started/Coordinate-Systems`.

[29] OpenGL. Extensive tutorial resource for learning modern opengl, 2014. URL: `https://learnopengl.com`.

[30] Michael Beetz Radu Bogdan Rusu, Nico Blodow. Fast point feature histograms (fpfh) for 3d registration. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.332.3710&rep=rep1&type=pdf`.

[31] Registration. The pcl registration api. URL: `http://pointclouds.org/`

documentation/tutorials/registration_api.php.

[32] Registration. Point set registration. URL: https://en.wikipedia.org/wiki/Point_set_registration.

[33] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2564–2571, Washington, DC, USA, 2011. IEEE Computer Society. URL: http://dx.doi.org/10.1109/ICCV.2011.6126544, doi:10.1109/ICCV.2011.6126544.

[34] Samuele Salti, Federico Tombari, and Luigi Di Stefano. Shot: Unique signatures of histograms for surface and texture description. *Computer Vision and Image Understanding*, 125:251–264, 2014.

[35] Ananth Ranganathan Th. The levenberg-marquardt algorithm, 2004.

[36] F. Tombari and L. Di Stefano. "object recognition in 3d scenes with occlusions and clutter by hough voting", 4th pacific-rim symposium on image and video technology, 2010, 2010. URL: https://vision.deis.unibo.it/fede/papers/psivt10.pdf.

[37] Andreas Rechtsteiner Luis M. Rocha Wall, Michael E. "singular value decomposition and principal component analysis". in a practical approach to microarray data analysis. d.p. berrar, w. dubitzky, m. granzow, eds. pp. 91-109, kluwer: Norwell, ma (2003). lanl la-ur-02-4001., 2002. URL: http://public.lanl.gov/mewall/kluwer2002.html.