Bilkent University

Department of Computer Engineering

Senior Design Project

MoveIt, Indoor Manipulation : 3D Semantic Reconstruction, Display and Manipulation System

Final Report

Barış Can

Faruk Oruç

Mert Soydinç

Pınar Ayaz

Ünsal Öztürk

Supervisor: Associate Professor Selim Aksoy, Ph.D.

Jury Members: Assistant Professor Shervin Rahimzadeh Arashloo,Ph.D.

Assistant Professor Hamdi Dibeklioğlu, Ph.D.

Final Report

May 27, 2020

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Contents

Final Report

MoveIt, Indoor Manipulation : 3D Semantic Reconstruction, Display and Manipulation System

1. Introduction

The popularity of smartphones rises daily. While they have many uses, it can be argued that taking and sharing photos are among the most widespread uses of today's incredibly powerful smartphones. Photos capture a moment in nature or an indoor scene, and they all have visual and spatial information of the objects contained within them. It is possible to extract this information, and more information can be extracted as one adds additional angles and positions from which the original scene is viewed. The amount of information one can extract is directly proportional to the extra number of angles and positions, and the quality of these images. On top of this, recently there have been more and more smartphones with special sensors that can extract the depth information from the scene they are looking at. Color and depth information contained in captured images then may be used to model the environment in 3D.

MoveIt: Indoor Manipulation aims to bring our homes into virtual reality by recreating 3D indoor scenes using a smartphone camera. With MoveIt, a quick scan of a room will bring the objects contained within this room into VR where they can be freely moved, interacted with, and manipulated. It is also a helpful tool that enables the user's aesthetic ability by allowing them to redecorate their room with ease while also providing functional help such as object boundary and collision detection. This 3D recreation of the room will be fully visualized in a VR environment that enables the user to freely move, scale, and rotate objects in the 3D reconstruction of the room using a VR Headset. Even if a user does not have a VR headset or does not want to use it, they can still have the same experience from their own PC within the same application.

In this final report, revised requirement details, final architecture and design details, development and implementation details, testing and maintenance details and other project elements will be discussed.

2. Requirements Details

2.1. Functional Requirements

This system has 4 main components, these are mobile phone components, a remote server component, computer vision components and VR Headset together with a personal computer. For smartphone components, we employed a set of Android APIs for the gathering of the required data and sending this data to be processed on a remote, monolithic server. This server component will act as the computation hub for the semantic segmentation of the image, the reconstruction of objects contained within these images into 3D objects, and the storage of the reconstruction of these objects in Firebase cloud storage. Computer vision component reconstructs the objects in the given scene by projecting them on the 3D space. It achieves this by extracting a depth map of the scene, and recognizing objects on the RGB and depth map through semantic segmentation. This semantic segmentation is achieved by a Neural Network model[1] that is already trained for this purpose on the MIT ADE20k dataset. VR headset is utilized for the 3D display of the 3D renderings of the reconstruction of an indoor scene through Unity Game Engine. A personal computer could also be used to run the application.

Mobile Phone Component

For mobile phones, the system's required functionalities include:

- Scan the contents of the indoor area by the use of its depth camera.
- Save the scanned contents into the storage of the mobile device.
- Forward the captured videos to the Computer Vision component.

Cloud Computing

The cloud server provides the following functionalities:

- Provide enough computational power to computer vision algorithms.
- Provide the necessary storage for the already scanned household items.

Computer Vision

The system uses a pre-trained neural network that is specialized on semantic segmentation for labeling the reconstructed 3D environment of the room, which follows these steps:

- Detect surfaces and objects in the indoor setting.
- Create titles, bounding boxes, and masks for the detected objects.
- Clearly separate the boundaries of the room from the boundaries of the interior objects using semantic labels.
- Project each object in the environment and create 3D objects of them.
- Create a detailed 3D map of the indoor environment with distinct objects.

Unity Application

In the final step, this component provides the following functionalities:

- Recreate the room in 3D with fully interactable objects in inventory.
- Place the user in the created room.

- Allow the user to create additional copies of the existing furniture.
- Scale, rotate, freeze, change the places of furniture.
- Apply physics and check for collision for every piece of furniture.

VR Exclusives

The features below are only for a VR headset.

- Allow the user to grab or release objects with their hands in real life.
- Manipulate the grabbed object as if it was a real life object.
- Pull a distant object towards the user's position.
- Perceive and display the real life body and hand movements on the headset’s screen.

2.2. Non-functional Requirements

In the following subsections, usability, reliability, security, smartphone friendliness and availability will be discussed.

Usability

- The MoveIt application has a user-friendly interface that would enable different users from various age and knowledge groups to use the application with ease, and it should not take more than 15 minutes for a user to get accustomed to the interface.
- The application includes a user manual, an instructions page explaining the users the key concepts of the application, such as how to record a room and how to move objects around. These tutorials should not take more than 2 minutes for a user to comprehend.
- Users are able to move around objects with natural controls that are easy to perform and come as natural, and a user should not spend more than 5 seconds on average to perform a given operation on an object.

- Users should not feel nauseous in the VR headset in a usage time of less than 30 minutes.

Reliability

- 3D reconstruction of the objects in a room must be as detailed as possible. Artifacts due to the reconstruction do not yield an error more than 10% in terms of volume of the individual object.
- Different objects from the same scene keep their relative proportions to each other.
- Different components of the application should work together harmoniously. To elaborate, MoveIt has a smartphone application as well as a background side where 3D reconstruction takes place. Since the user interacts with the smartphone application, the background computation time is optimized as much as possible to guarantee responsiveness. However, the reconstruction is very computation intensive and the scene data is very large which increases the required time. In the end, computation for the reconstruction should not take more than an hour.
- When the reconstructed objects are manipulated by the user (e.g. moved around), the previous place of the object as well as the new one should be altered and displayed correctly by the application. The only tolerable error should be the floating point representation rounding-off errors.

Security

- The application should be able to protect the data that the user uploads to the system such as their recordings or their email address since these data could be classified as personal. A secure hashing scheme is used, and this hashing scheme should be considered as “having minor weaknesses” in the worst case [2].

Smartphone Friendliness

- The users are expected to use the smartphone as an interface to interact with the MoveIt application. Therefore, the mobile application should not be very taxing on smartphones and the power usage should be optimized to ensure user satisfaction. The user should be able to use the application for at least two hours before the battery of the smartphone runs out. The video should be recorded at least 24 frames per second.

Scalability

- Since the computations are done on a remote server, it might require too much computational power if several users send their videos at the same time. In order to solve this, the computational power of the server should be increased by hiring a server with double the number of cores and double the amount of memory. Once scaling up costs exceed 16 times the initial server rent, distributed versions of the algorithms to be developed should be introduced, and multiple servers should be used to carry out the computations.

Availability

- The system should be available for the users and function at all times. But since server failures are inevitable, we are aiming for an SLA level of 99.9% (Yearly 8h46m of downtime) [3].

2.3. Pseudo requirements

- Smartphones are not powerful enough to do the processing of the scan by themselves so a remote server is being used.
- The object oriented programming paradigm is followed during the development of the application.

- Semantic segmentation is done using a pretrained-semantic segmentation model [8,29,30] that is trained on the MIT ADE20k dataset specifically for this purpose. The model uses the Pytorch library of Python in order to label the objects correctly and create masks on the original image.
- The visual components of the application such as the movement of the objects are done on Unity Game Engine with C#. [4]
- The mobile application is developed on Android Studio with Java. [5]
- Opencv and Eigen libraries of the C++ language are utilized for the implementation of the 3D reconstruction as the C++ language is the fastest one applicable for this.
- The programming languages used in the project are able to interact with each other.
- A database for the user information, object meshes and their labels are hosted on Google Firebase Cloud Storage.
- MoveIt should run in Android phones (7.0 or above) with at least 2 cameras, Windows 8/10 and optionally a VR device.
- The webpage of the project which contains all necessary information is https://moveitplatform.wixsite.com/website.

3. Final Architecture and Design Details

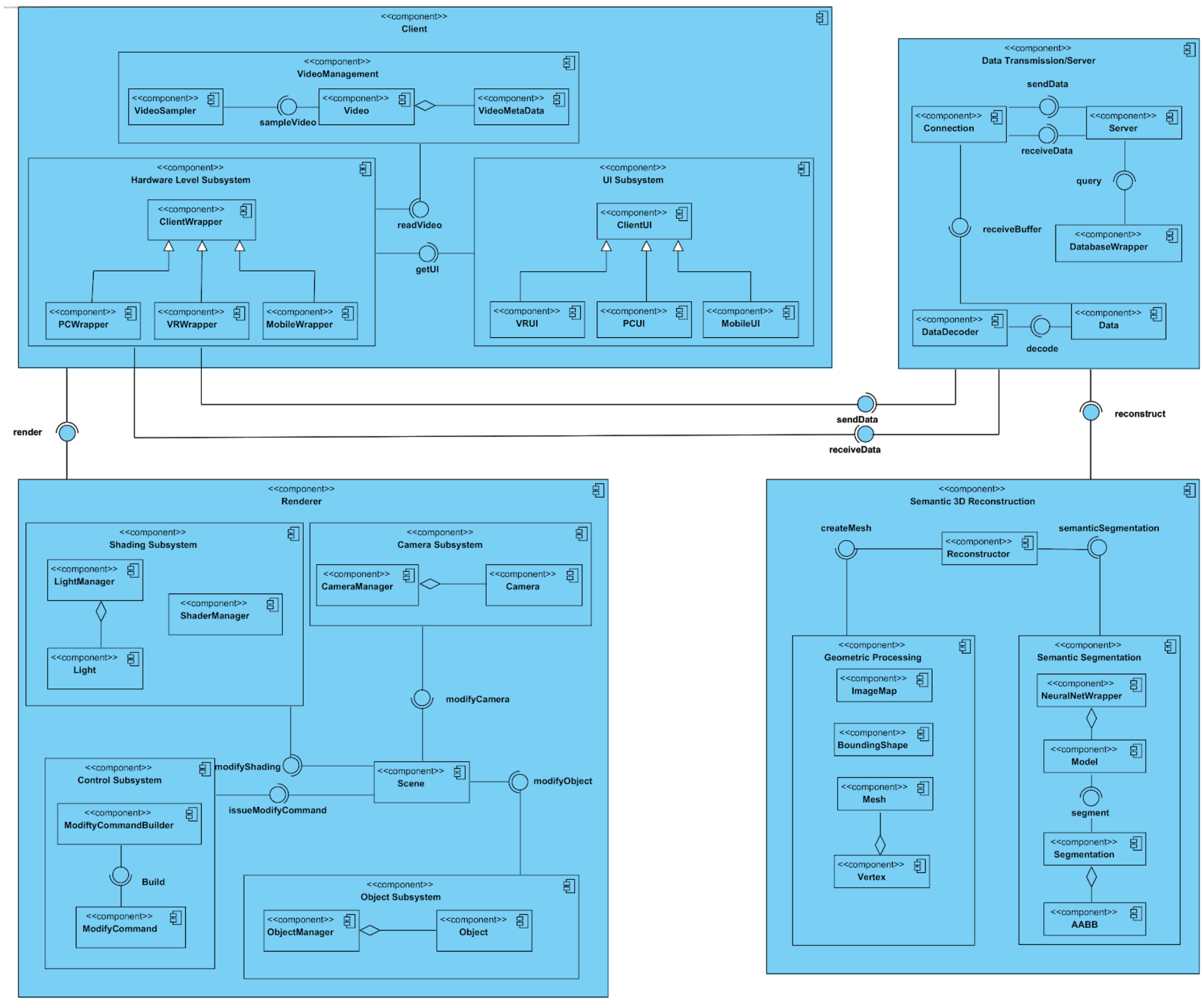


Figure 1: Final Architecture

The MoveIt software architecture is divided into four main subsystems, namely the Client, Server, Renderer, and 3D Semantic Reconstruction subsystems. The Client and Server subsystems are based on Client-Server architecture, and their main purpose is to provide a software backbone for the transmission, storage, and processing of data. These subsystems allow the users to send their data to the server, have it processed and stored, and retrieve a semantic 3D reconstruction of their rooms in a streamlined and efficient manner. The separation between the client and the server incurs an enhancement in privacy and safety measures, as the server manages user connections in a peer oblivious way from the perspective of the client, i.e. the users are semantically unaware of the other users currently connected to the server.

The Server subsystem and its deployment on a separate and a computationally powerful machine, relative to mobile devices and personal computers, also yields an increase in the efficiency of MoveIt, as processing the data on a server capable of performing feature extraction, geometry processing, and parametric mapping of textures and displacements in a parallelized manner. The server also maintains privacy and safety through making the database involved in the storage of meshes and textures to be unavailable to outside users, as a result of the encapsulation present in the design of the subsystems.

The Client and the Server subsystems communicate through a TCP interface, using a MoveIt specific data format encapsulated in a Data object. The data received or sent through a connection is preprocessed according to application specific file/data headers, which is later converted to conventional formats. E.g. the client creates a connection to the server through the services provided by the Server subsystem, and sends the relevant bytes to the server through the connection. The connection blindly passes the bytes to the server along with the application specific header describing the context of the data, and through the services provided by a data decoder, the data is decoded according to the hardcoded headers, and then converted into commercially available formats. As of now, this subsystem is not implemented due to the delay in other subsystems caused by COVID-19. However, this subsystem will be implemented at a later date.

MoveIt also makes use of the MVC pattern for its Renderer subsystem in an event based manner. A thread continuously polls for the inputs arriving to the subsystem, which are then passed to a controller subsystem, which issues modification requests to the relevant subcomponents, which in turn update the abstract representation of 3D models, and in the next render loop, the updated models are rendered, hence changing the view, in accordance to the MVC pattern.

The 3D Semantic Reconstructor does not use any particular design pattern. It provides the logic for geometry processing and has wrapper classes for representing pre-trained neural networks for the purposes of semantic segmentation. The semantic segmentation of successive frames in the video provided by the user is used to estimate semantic tags for the reconstruction. This subsystem combines these two nested subsystems to produce a final 3D semantic reconstruction of the indoor scene. Our C++ executable provides the point clouds for every object in the scene that is created using the Depth information of the images and their object types. The created point cloud is then further processed using the Meshlabserver[6] software. Meshlabserver applies a number of filters to the original point cloud to turn it into a watertight mesh. These filters include computation of normals and smoothing of said normals, poisson reconstruction to create a mesh, hole filling and removal of the disjunct parts from the mesh,laplacian smoothing, transform into a pure triangular mesh and the generation of the texture map. The described software for this subsystem runs exclusively on the server, and the Server subsystem also persists the reconstructions by writing the necessary data to a database. The reconstructed meshes and related parametric texture maps are communicated back to the client in the same manner the client sends the video to the server: the server, through the connection object between the client and the server, sends the data as bytes, and an application specific header for decoding purposes. The data decoder on the client side decodes the data via the header and converts the data into a commercially available format, e.g. .obj files for meshes and .png files for parametric texture/displacement maps.

Client subsystem also provides the UI for the actual usage of the app, along with several OS and file system utilities for the storage and management of scenes and videos of the user. The subsystem provides different wrappers and UI software for different hardware: one for VR, one for PC, and one for Android devices. These wrappers and user interfaces also provide the necessary facilities for rendering and scene manipulation through the services offered by the Renderer subsystem. PC and VR interfaces will be very similar and they will be in the computer application.

4. Development/Implementation Details

4.1. Mobile Application

Android application is implemented using the Java libraries provided by the Android framework. Since the only purpose of the Android application is to record an RGB video and estimate the depth of the scene, no authentication is used for the Android application for the time being.

The Android application uses a camera API for RGB image acquisition and depth map estimation, called SyncCamera, which is implemented by us. The implementation for SyncCamera, along with a sample application using this API is available on the GitHub page of one of our group members [7]. We also provide an implementation for depth map estimation based on feature extraction and constructing disparity maps for devices without ToF cameras. This implementation is not included in the mobile application, however, due to its computational intensity. It is instead included in the software running on the server, and at the current state of our implementation, on one a machine with a graphics card.

SyncCamera is a camera API built on top of Android's camera2 API (package android.hardware.camera2). The main purpose of this API is to provide logical abstractions around the hardware-heavy API exposed by Android's camera2 package. The API also provides utilities to write large buffers containing sensor information on the disk of a mobile device in a parallelized fashion, to track the progress of writing the said buffers on disk, to display processed or raw sensor information on screen, and to access the sensor in a user-defined manner.

SyncCamera implements four camera classes named AbstractCamera, ColorCamera, DepthCamera, and KinectCamera, the latter three of which are subclasses of the first class. AbstractCamera class provides a wrapper around Android's camera2 package classes for accessing a camera, though it does not specify the way in which the camera is accessed, which is left to the subclasses implementing its setCaptureRequestParameters() method. AbstractCamera also allows its subclasses to implement a decodeSensorData(ByteBuffer) method, which allows the users to specify how to transform a given input buffer filled by the hardware to a desired format. Besides these, the AbstractCamera class implements getters and setters for various camera-related fields such as the logical ID of the camera provided by the Android operating system, handlers for camera devices, camera resolution in pixels and millimeters, utilities for building CaptureRequest objects, private and automatically called functions for dispatching CaptureRequests, and camera parameters. AbstractCamera also implements the BitmapProducer interface, which forces its subclasses to implement the broadcastBitmap() method.

ColorCamera is a subclass of AbstractCamera, which implements the acquisition of a color image from an RGB camera registered to the operating system. It requests the intrinsic and extrinsic camera parameters from the OS, along with the memory location to which the physical camera writes its raw sensor information. It provides a choice between raw images, which require user implemented device specific demosaicing, and already demosaiced jpeg-style images. It implements the decodeSensorData(ByteBuffer) function in such a way that the Bitmap provided by the physical sensor is converted into a Bitmap instance representing, visually, a color image. As its CaptureRequest parameters, it forces the camera device to write 30 frames per second to its buffer, requests hardware-level denoising, distortion correction (which attempts to cancel out the predetermined distortion coefficient values, which are hardcoded in a ROM by the manufacturer of the camera device), and asks the device to return a camera device with a focal length closest to the focal length specified by the programmer. After the camera instance is created and the proper BitmapObservers are attached to the camera, the thread containing the instance of the BitmapObserver will be interrupted 30 times per second to let the instance know that a full frame was written by the hardware on the buffer, processed by the handler thread of the camera in the specified way (as per the decodeSensorData(ByteBuffer) implementation), and is ready to be consumed. The class also provides device-universal constants for buffer sizes and aspect-ratios as static constants.

DepthCamera is similar to ColorCamera in terms of its observer pattern implementation. It differs from the ColorCamera implementation in the sense that it decodes the sensor data differently. It provides a single alternative as its decode pattern according to the DEPTH16 format. Sensor-decoding of the depth camera reads the buffer of the sensor two bytes at a time, takes the least significant 13 bits in a big-endian context, and copies the converted values into the same location of another buffer. The first three bits are used to calculate the confidence of the depth estimate of the sensor, and if the confidence is below a certain threshold, then the depth estimate is flagged as unreliable. The depth value is then converted to a 32-bit floating point value, which is enough to fit inside an 32-bit RGBA pixel, and a bit-level hack is used to increase the speedup of the copy.

KinectCamera functions mostly the same as DepthCamera. However, instead of returning the depth estimate as a 32-bit floating point value, it uses the Kinect depth format. It encodes the depth value in two bytes in a big-endian format in the first 13 bits, and stores a “device id” in the last three bits. These two bytes are encoded in a 16 bit pixel, the first byte occupying the alpha and blue channel, and the second byte occupying the green and red channel.

The API also provides a framework for saving color and depth bitmaps, and providing progress information. This is implemented through RGBDCaptureContext, BitmapEncoder, BitmapEncoderTask, BitmapMetadata classes and ProgressObservable and ProgressObserver interfaces.

RGBDCaptureContext exposes a simple API to define a context for the capture of RGB and depth image pairs. It allows the definition of the format of the files (e.g. PNG, JPG), to be saved on the disk, the compression ratio of the Bitmap instances to be saved, and the path and the names of the files. It implements both ProgressObserver and ProgressObservable interfaces. This allows the programmer to attach a ProgressObserver to the RGBDCaptureContext instance to be notified of the progress of the capturing process on the demand of the calling thread. RGBDCaptureContext itself implements the ProgressObserver interface, since this class itself does not capture the sensor information, but rather it asks a static delegator class, BitmapEncoder, to allocate resources for the context to be captured, and checks the progress of capture through BitmapEncoder. In a typical call to RGBDCaptureContext’s startCapture(int, CameraCalibrationMetadata, CameraCalibrationMetadata), the RGBDCaptureContext instance simply piles up the bitmaps broadcasted by BitmapProducer instances to which it is attached in memory. Upon calling stopCapture(), RGBDCaptureContext requests resources and workers for a capture job from BitmapEncoder. The number of threads requested by RGBDCaptureContext for saving the captured bitmaps increases logarithmically with respect to the number of frames to be saved on the disk. After RGBDCaptureContext requests these resources from BitmapEncoder, the BitmapEncoder returns the IDs of the jobs, which RGBDCaptureContext uses to keep track of the progress of the jobs. It keeps a roster of depth and color jobs, and whenever a job is done, RGBDCaptureContext is notified through the ProgressObserver interface with the ID of the job completed. The RGBDCaptureContext instance then marks jobs as finished until all jobs are finished. After all jobs are finished, the context writes a metadata file on the disk regarding the capture, and notifies its observers through the ProgressObservable interface. The object may then be safely discarded.

BitmapEncoder is a static class built using the singleton and builder patterns, which is responsible for scheduling and managing write jobs and resources. To this end, it implements a thread pool with the number of cores as its number of threads, along with a static BlockingQueue to keep track of the threads running on the cores of the mobile device. In order to dispatch a write job, the BitmapEncoder has to be set-up, as it is a finite state machine by implementation. Before a job can be submitted, the programmer has to call setCompressFormat, setCompressionFactor, and setActivity in a synchronous manner. To prevent race conditions, the BitmapEncoder is locked for four consecutive function calls for setting up the encoder and submitting the job. The BitmapEncoder keeps a registry of running jobs, whose progress can be queried through the IDs of the jobs, which are returned to the threads submitting the jobs at the time of submitting, through progress(int), size(int) and bitmapsEncoded(int). A job is considered submitted after encode() is called. A typical call to encode instantiates a BitmapEncodeTask instance and allows this instance to “borrow” a thread from the thread pool to run on. If no threads are available, then the job is queued until a thread becomes available. The queueing is performed in a FCFS fashion.

BitmapEncodeTask represents a portion of a bitmap write job. Each BitmapEncodeTask contains the indices of bitmaps to be encoded and written on the disk. Each of these tasks have a job ID, directory path, compression factor, compression format, the list of bitmaps to be encoded, start and end indices, a reference to the activity from which the job was requested, the ProgressObserver instances observing this job, and its Thread instance, borrowed from the thread pool. It also stores a volatile integer representing the number of bitmaps encoded by the thread to track progress. Each BitmapEncodeTask encodes end_index - start_index + 1 bitmaps on the disk before returning the Thread instance to the thread pool and notifying the ProgressObserver that the job is finished.

BitmapMetadata class allows writing metadata to image files. It is a simple wrapper around Android’s ExifInterface class, which allows for more convenient calls to ExifInterface.

SyncCamera also provides utilities for file tagging and camera calibration. CameraCalibrationMetadata stores intrinsic and extrinsic camera parameters along with distortion coefficients for each camera, and BitmapBroadcastContext and BitmapDataType help programmers identify what kind of bitmaps are being shuffled around memory under what context by providing relevant fields related to the camera from which a bitmap was made available, the ID of the camera, and the type of the bitmap encoding (e.g. regular depth, kinect depth, color).

A newer version of SyncCamera used in our app includes a new class, SensorDataManager, which attempts to perform inertial indoor navigation using gyroscopes and accelerometers present on most Android devices. This class solves a system of linear differential equations using symplectic Euler. It was deemed to be too noisy and an experimental Kalman filter was implemented to reduce noise; however, since the noise produced by the hardware is different for every device, a different Kalman filter is required for each device, and since the hardware noise in some devices can not be estimated using a Kalman filter due to their nature. For the time being, this extra feature is abandoned.

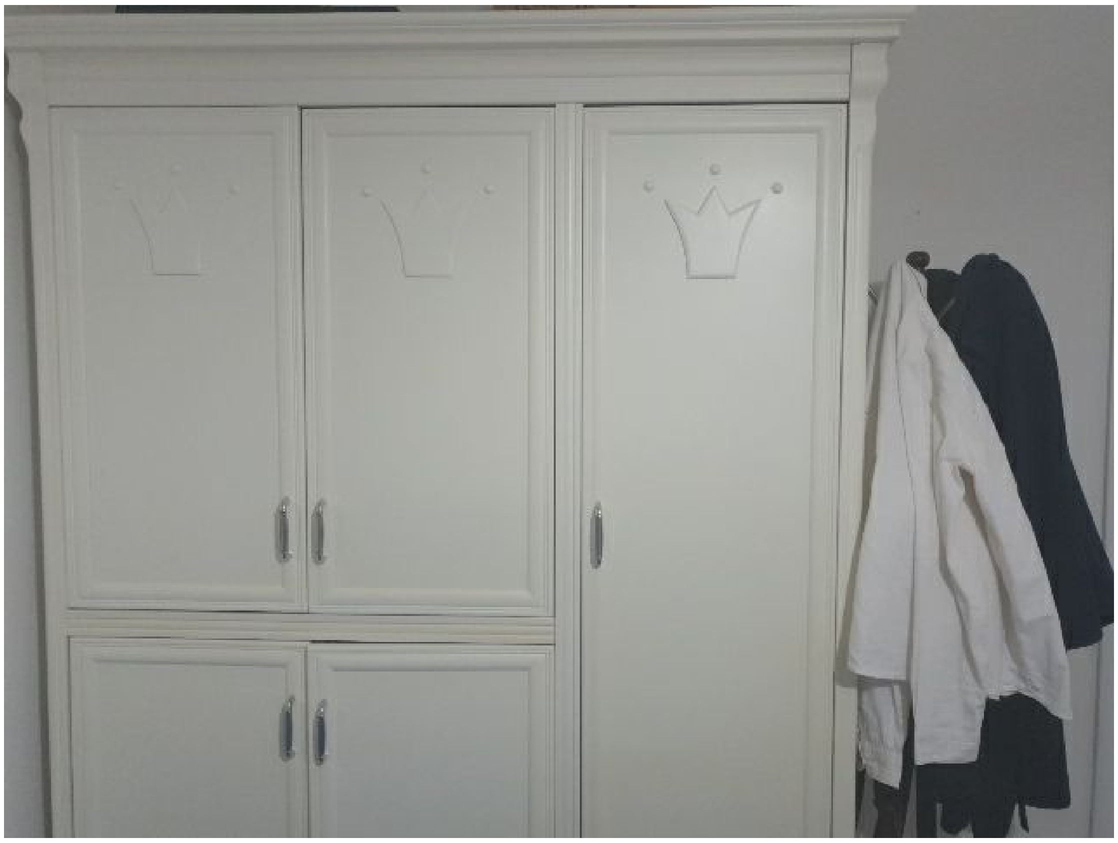


Figure 2: Wardrobe RGB Photo

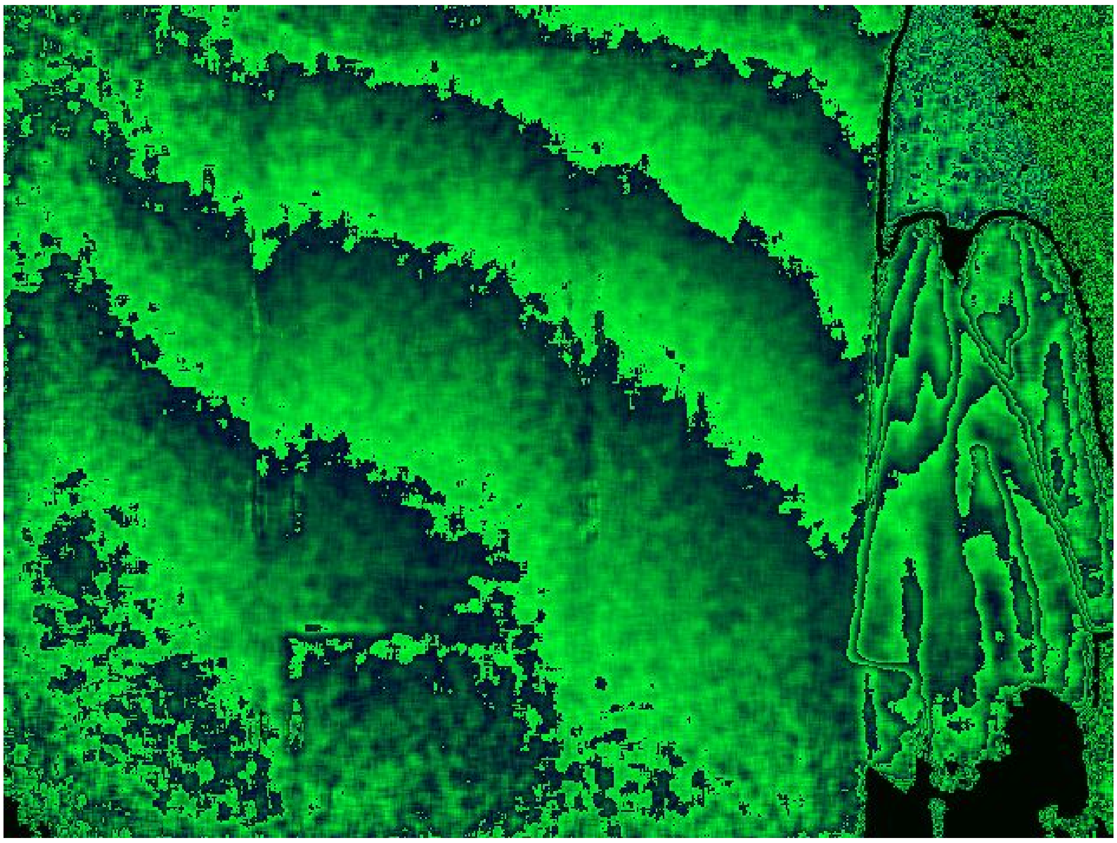


Figure 3: Wardrobe Depth Photo

4.2. 3D Reconstruction and Semantic Segmentation

4.2.1. Semantic Segmentation

In our application we are using a pretrained Deep Neural Network [8,29,30] that is implemented using the PyTorch library. There are many implemented models in the given Python script but we are using ResNet101dilated + PPM_deepsup that is trained on the ADE20k dataset [8] on 150 different classes. The model we are using has a 80.59% Pixel Accuracy which is the most important metric for our purposes. We are supplying our RGB photos into the model and receive an image mask that is overlaid on the original image. We then use this new image to figure out which Vertex belongs to which objects in our Point Cloud Generation.

Figure 4: Segmented Wardrobe Photo

4.2.2. 3D Reconstruction and Point Cloud Generation

We wrote a lightweight, general purpose 3D reconstruction library in C++, available on the GitHub page of one of our group members [9]. The library exposes an API that can be used to read color, depth, and semantic label images, camera parameters, and semantic label metadata, along with several functions that allow 3D semantic point cloud reconstruction through image rectification/alignment and “homogeneous projections” from 2D to 3D through the help of depth information. The library has three dependencies: Eigen for matrix operations and solvers, OpenCV for reading images and extracting features from images, and TinyPly for writing point clouds as files. The library is composed of 12 headers, each header corresponding to a different functionality the library offers.

The first class citizen of the library is Reconstructor, which makes use of the rest of the library to minimize the effort of the programmer. A Reconstructor instance has be configured before it can be used, and the API allows the user to specify the path of the dataset from which the 3D semantic point cloud is to be generated, the indices of the images that are to be loaded in memory or the number of images to be loaded given an offset index, and the sampling rate of the images. The reconstructor assumes that the supplied directory contains four folders named depth, metadata, rgb, and semantic. The directory called "depth" is expected to contain 640x480 depth images in .png format, numbered through 0 to N in SyncCamera KinectCamera encoding format (16 bits per pixel, first 8 bits encoded in alpha and blue channels, latter 8 bits are encoded in green and red channels in a big-endian format). It is expected that "rgb" directory contains 640x480 color images in .png format numbered through 0 to N, and that "semantic" directory contains semantic segmentations of images contained under "rgb" again numbered through 0 to N, labeled with colors specified in CSAIL MIT's color150.mat file [10]. The "metadata" directory is expected to contain three files: color.txt, depth.txt, and labels.csv. The format of these files are documented in the example metadata directory on the GitHub page of the code. "color.txt" and "depth.txt" should contain the camera calibration data, which should be available as output from SyncCamera. If one does not wish to use SyncCamera with this library, one can simply calibrate the camera by hand and supply the parameters manually. After one configures the Reconstructor instance successfully, the Reconstructor instance loads the three types of images through an Image containerization and input class called ImageCollection, which reads rgb, depth, and semantic images from their respective directories. These image collections are then mapped to one another through the ImageToPoseMap. This map can also be used to store estimated camera poses with respect to world coordinates upon configuring the Reconstructor instance. Configuring the reconstructor also loads the camera parameters from "color.txt" and "depth.txt", and instantiates a SemanticLabelManager instance. The SemanticLabelManager contains a color string to semantic name map. After configuring the Reconstructor instance, it is recommended that the programmer calls the project_frame(int, Matrix4, bool) method to obtain a 3D point cloud from a single depth map.

The reconstructor instance provides other utilities for camera pose estimation.
get_feature_match_indices(Image,Image,vector<Coord2>&,vector<Coord2>&) computes feature matches between two images. To compute features for the images, the images are first converted to grayscale using the luminance formula, and then the ORB feature detector (courtesy of OpenCV) detects the features in each image. We chose ORB over other feature detectors such as SIFT, SURF, and KAZE, since ORB is not patented and is easier to compute and match [11]. The features are then matched, and since the two frames provided as inputs to this function are two consecutive frames of a video, the function removes outlier matches by eliminating matches whose L1 norms are than 15 pixels (a value empirically determined) apart. On average, we estimated that this removes 89% of outlying matches. The function then returns the indices of the matched points through the parameter list. These features may then be used to estimate the relative camera pose between two images through either a user defined way, or a way provided by the Reconstructor class.

Another function implemented by the Reconstructor is the estimate_camera_pose(int i, int j) class, which returns a transformation matrix relating the camera poses at frame i and j. It is recommended that i and j should ideally be at most 2 frames apart. The function first extracts and matches features in the same way as get_feature_match_indices, and then uses these matches to estimate the essential matrix relating the two images using the eight-point algorithm and total least squares minimization. The function then estimates the rotation and the translation between the two image planes by performing a singular value decomposition on the essential matrix into $U\Sigma V^T$ and computing a translation and rotation using another matrix, W,

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which is then used to compute $t = UW\Sigma U^T \; and \; R = UW^{-1}V^T$. Note that the rotation might not be accurate, and in this case an alternative matrix Z is

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

used and the rotation matrix is taken as $R = UZU^T$. The rotation matrix to be taken is determined by a chirality check such that the resulting projected depth at any point is not negative. OpenCV implements this, which is also used in our implementation.

The Reconstructor also implements point cloud stitching. The function estimate_transform_between_points(vector<Point4>, vector<Point4>) takes two 3D sets of mapped-matching points in homogeneous coordinates and estimates a projective transform Q with homogeneous scaling of 1 of the form

$$Qp_1 = p_2$$

where $p_1 = [c_1, c_2, ..., c_n]^T$ and $p_2 = [d_1, d_2, ..., d_n]^T$, where c_i and d_i represent the points in point clouds 1 and 2. The projective transform is calculated by converting the system to the form $Ax = b$ as follows:

$$\begin{bmatrix} c_1^T & 0 & 0 & 0 \\ 0 & c_1^T & 0 & 0 \\ 0 & 0 & c_1^T & 0 \\ 0 & 0 & 0 & c_1^T \\ c_2^T & 0 & 0 & 0 \\ 0 & c_2^T & 0 & 0 \\ 0 & 0 & c_2^T & 0 \\ 0 & 0 & 0 & c_2^T \\ \vdots & \vdots & \vdots & \vdots \\ c_n^T & 0 & 0 & 0 \\ 0 & c_n^T & 0 & 0 \\ 0 & 0 & c_n^T & 0 \\ 0 & 0 & 0 & c_n^T \end{bmatrix} \begin{bmatrix} q_{11} \\ q_{12} \\ \vdots \\ q_{44} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$$

where q_{ij} are the entries of Q. Since this system is clearly overdetermined for n more than 4, (A is of size $4n\, x\, 16$), we use linear least squares to estimate the transform, i.e. we estimate the projective transform as $x = (A^T A)^{-1} A^T b$. On properly rectified images, it is usually the case that $q_{41} = q_{42} = q_{43} = 0$ and $q_{44} = 1$, which means that the transformation between the point clouds are affine (also enforced by the homogeneity scaling condition). We use Eigen to handle the matrix multiplications.

The points used by the previous function can be generated by a function called project_project_depth_at_indices(Image, Matrix3, double, vector<Coord2>), which returns a vector<Point4> instance. The function projects the depth map to 3D *only at indices specified by the vector<Coord2> parameter.* This means that for two images with feature match indices, a programmer can call this function on both images with their respective matched feature indices, and obtain an approximate roto-translation-like transformation between two point clouds. This transformation can be extended to the non-feature matches of two images. By multiplying every point in the resulting point cloud obtained by fully projecting a depth map, one can “stitch” two point clouds, the pivot points of which are the feature matches in both images. This can be extended to the entire data set of captured RGB images and estimated depth, with an appropriate sampling rate of images, to produce a photogrammatically realistic point cloud coming from multiple images.

The second class citizen in the Reconstructor class is the project_depth_image(Image, Image, Image, Camera, Camera, Matrix4), which implements the actual projection of points into a point cloud, using a depth image, a color image, a semantic label image, intrinsic and extrinsic camera parameters for RGB and depth cameras, and a transformation matrix that transforms the obtained point cloud to world coordinates. The function first removes the lens distortion in all images. The distortion model assumes that lens distortion “perturbs” the location of the pixels by the following polynomial:

$$r_p = r(\kappa_0 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6)$$

where r_p is the perturbed radius, r is the distance between the pixel and the center of the image and κ_i are distortion coefficients. If used in conjunction with SyncCamera, this step may be safely skipped since distortion is removed at save-time. Then, for every pixel (u,v), the homogeneous “inverse” projection is computed as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R^T \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} z_d u \\ z_d v \\ z_d \end{bmatrix} - R^T t$$

where the left hand side is the corresponding 3D point, the first matrix on the right hand side being the 3x3 rotation matrix obtained from the extrinsic parameters of the depth camera, the second 3x3 matrix being the inverse of the intrinsic matrix of the depth camera, the 3x1 vector being the homogeneous the pixel coordinate with scaling equal to the value of the pixel in the depth map, and t being the translation vector of the camera. To find the color and the label of this point, the point is projected once on the color camera to compute the corresponding pixel coordinates of the point on the color and label image. This projection is where the vector on the left hand side

$$\begin{bmatrix} \lambda u_c \\ \lambda v_c \\ \lambda \end{bmatrix} = R_c K_c \begin{bmatrix} x \\ y \\ z \end{bmatrix} + t_c$$

is the 2D pixel coordinates with a homogeneous scaling of λ and R_c, t_c are the extrinsic parameters of the color camera, and K_c is the intrinsic camera matrix of the color camera. After computing the corresponding pixel coordinates, the 3D point is assigned a color $I(u_c, v_c)$ from the color image. The SemanticLabelManager is consulted for the semantic label of the point. All this information is then wrapped into a Vertex instance, defined in defines.hpp, and is then multiplied by a transform that may be passed through the parameter list.

The Reconstructor class manages the points in the point clouds in a Scene object, which has two cameras representing the color and depth cameras. The Scene object of the Reconstructor class may be written to a file in .ply format. The save_scene(string, bool, bool) function of Reconstructor allows a programmer to save a point cloud on disk. The string specifies the base filename of the .ply file, the first Boolean parameter specifies whether the point cloud should be centered around its centroid and the second Boolean parameter specifies whether the scene should be separated into semantic instances, i.e. whether a point cloud is split up into whatever was detected during semantic segmentation or not. Lastly, the Reconstructor can flush the point cloud of the Scene object, which simply deallocates memory and deletes all the points.

Project specific definitions are located under “recon/defines.hpp”. This header file simplifies the data structures used throughout the library: it provides type definitions as a sequence of typedefs from Eigen and OpenCV libraries. It heavily inherits the dense Matrix algebra aspect of Eigen by redefining double precision vector and matrix types of Eigen (Eigen::MatrixXd) in the project specific namespace under simplified names. This file also provides semantic differentiation between points, vectors, colors, and integer coordinates to provide a programmer-friendly API. This file also contains common data structures used throughout the project, namely SemanticLabel and Vertex. An extra Pixel class is also defined, purely for convenience for later uses of this project (it is not referenced to anywhere in the project).

The implementation also includes a utility header under “recon/fileutils.hpp” to read and write color and depth images, .ply files, and to request information regarding directories. The utility header file is based on TinyPly and the <filesystem> library provided in ISO C++ 2017.

The programmer is also provided the opportunity to use the classes that are used by the Reconstructor. Classes such as Scene, MatchGraph, CameraPose, and CameraParameters are under public packages. These classes provide logical representation of the objects they describe and are equipped with file I/O utilities to automatically read files in specific formats.

4.2.3. Mesh Generation From Point Clouds

At this point we have a Point cloud of a specific object as a “.ply” file. However, we need a pure triangular(watertight if possible) mesh as a “.obj” file and its texture as a “.png” file in order to import them into Unity. We achieve this using a software called Meshlabserver[6] which is the command prompt version of Meshlab (As a side note we choose Meshlab over CGAL because Meshlab provided us a visualization tool which proved very convenient for us). We apply a number of filters on our initial Point cloud using Meshlabserver Scripts that specify the filter, parameters for that filter, input files and output files. These scripts were initially prepared by using the visualization tool of Meshlab. We applied the filters in order on one of our objects then altered the parameters until we had a good result. We then used these good filter parameters in our scripts. These scripts are then called from our c++ project using the “void system(string cmd)” function which executes the cmd command on the command line terminal.

The list of filters and their respective parameters are as follows:

- Compute the normals for the point set using 13 neighbors and 1 smoothing iterations and flip them with respect to the camera(This is necessary for the texture mapping in the later steps).
- Smooth normals using 16 neighbours.
- Surface Reconstruction: Screened Poisson with Reconstruction Depth:8, Adaptive OCtree Depth:5, Conjugate Gradients Depth:0, Scale Factor:1.1, Minimum Number of Samples:1.5, Interpolation Weight:4 and Gauss-Seidel Relaxations:8. This step is crucial as it transforms the point cloud into a proper, semi-watertight mesh.[12]
- Select and Remove edges longer than 1.8947 in order to remove most of the artifact surface created by the earlier reconstruction.
- Smooth Face Normals
- Taubin Smooth with Lambda:0.5, mu:-0.53, steps:10. [31]
- Remove Isolated pieces with respect to diameter with a percentage threshold of 30%
- Close holes in the mesh with the max size of the holes being 300 units.
- Turn the mesh into a pure-Triangular Mesh to avoid Unity mismatch.
  Next two steps create the texture .png file
- Parameterization: Trivial Per-Triangle with a Inner-Triangle border of 0.

- Transfer: Vertex Attributes to Texture

Then we save the file as a .obj file which ends our transformation.
We either used default values on the filters or adjusted them using trial-and-error on one of our objects.

After all the filters are applied we are left with 2 files: an object file and its texture which are then imported into Unity. We are using a special Shader for Unity to render our objects. This Shader is a slightly modified version of the one of the built-in shaders of the Unity Engine which is the Particles Unlit Shader[13]. We disabled the Back Culling of the original shader in order to render the back of the vertices which sometimes becomes transparent in the default shader.

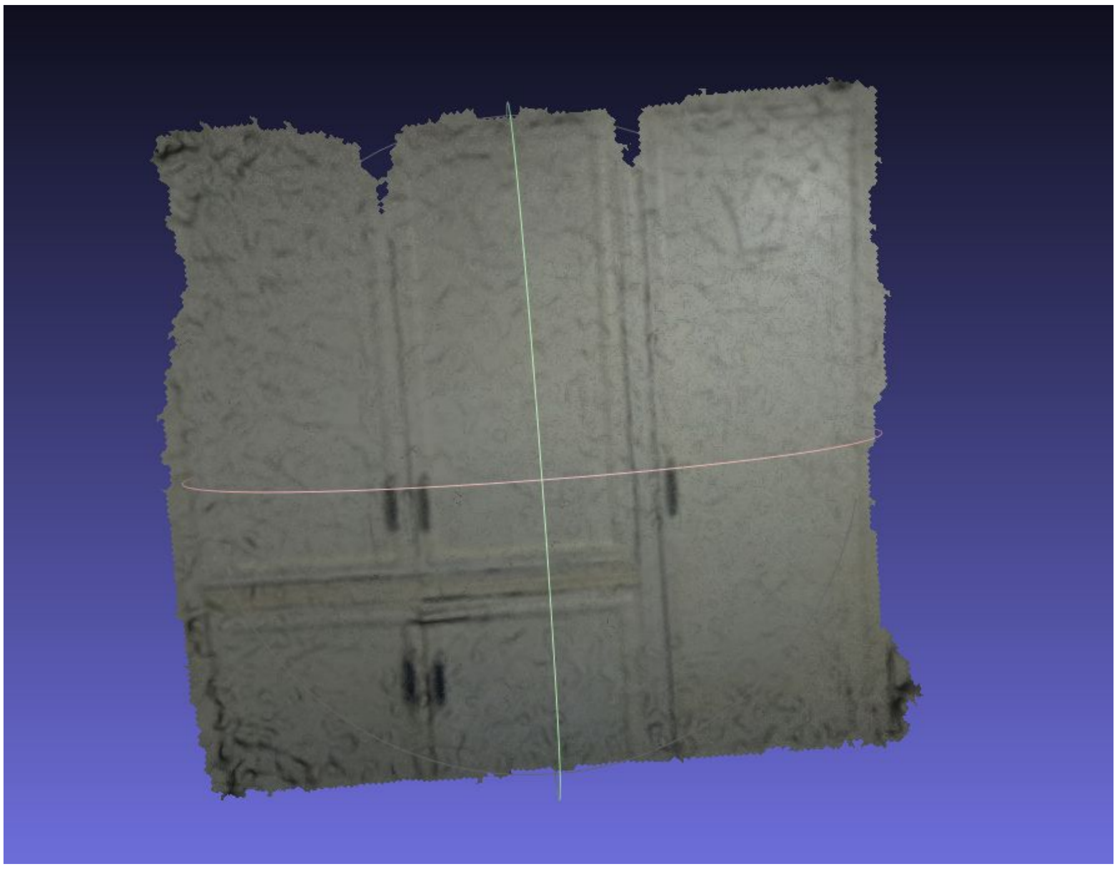


Figure 5: Example 1 - Reconstruction of the previously shown images into a mesh(Front)

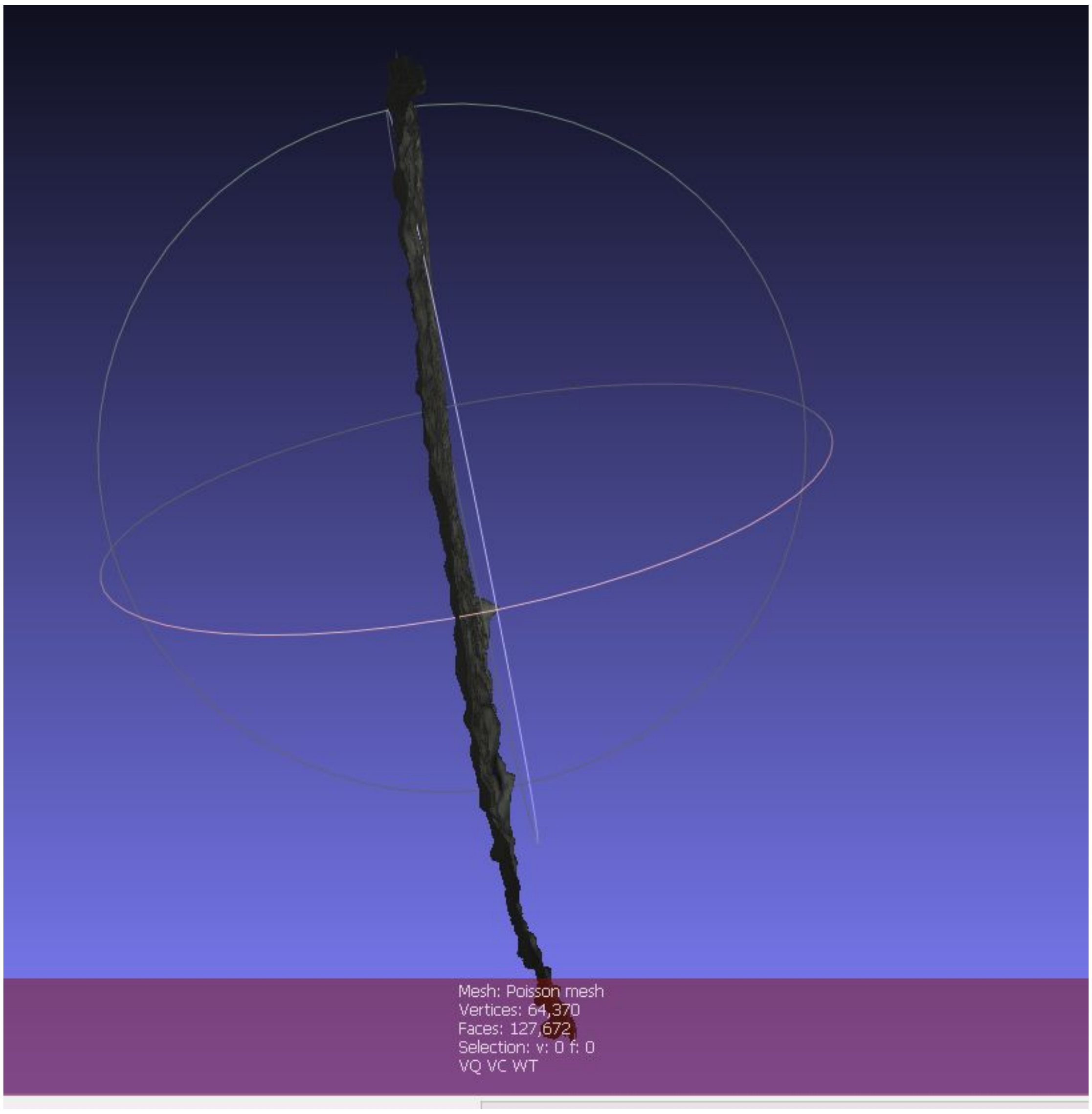


Figure 6: Example 2 - Reconstruction of the previously shown images into a mesh(Sideways)

As you can see from the screenshots above, since we do not have the feature of completion of occluded spaces, even if the front side of the wardrobe is quite good, when looked sideways, it does not have any depth.

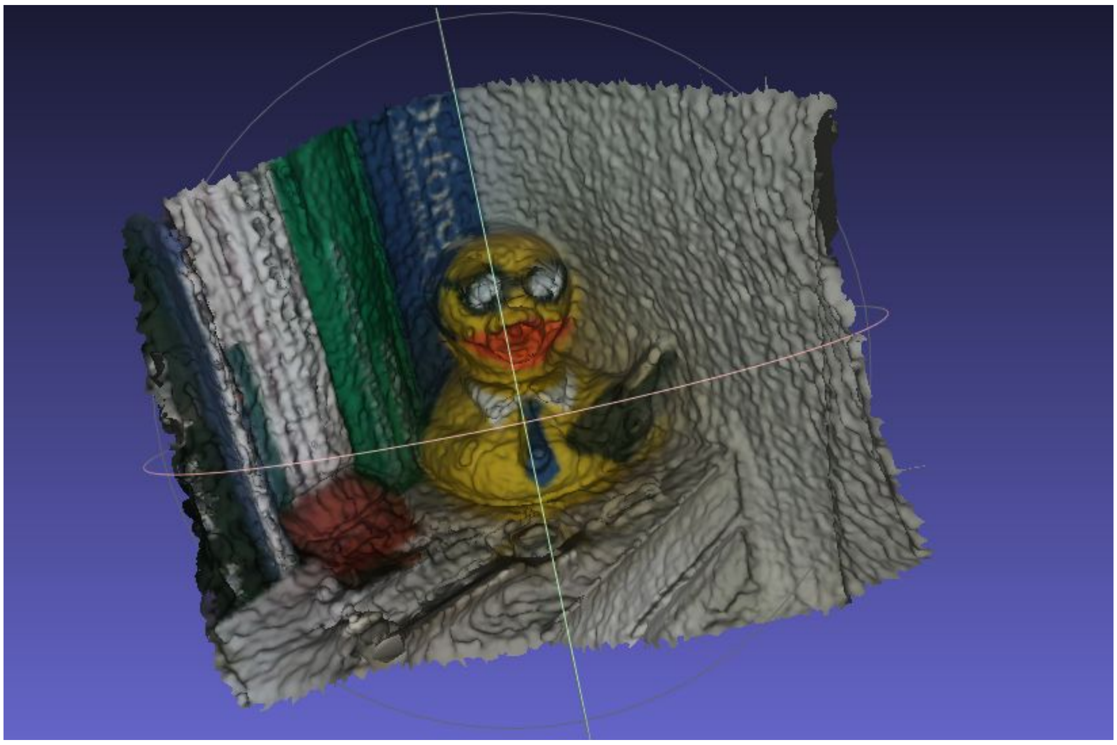


Figure 7: Example 2 - Reconstruction of the Corner of the Room(Without any segmentation)

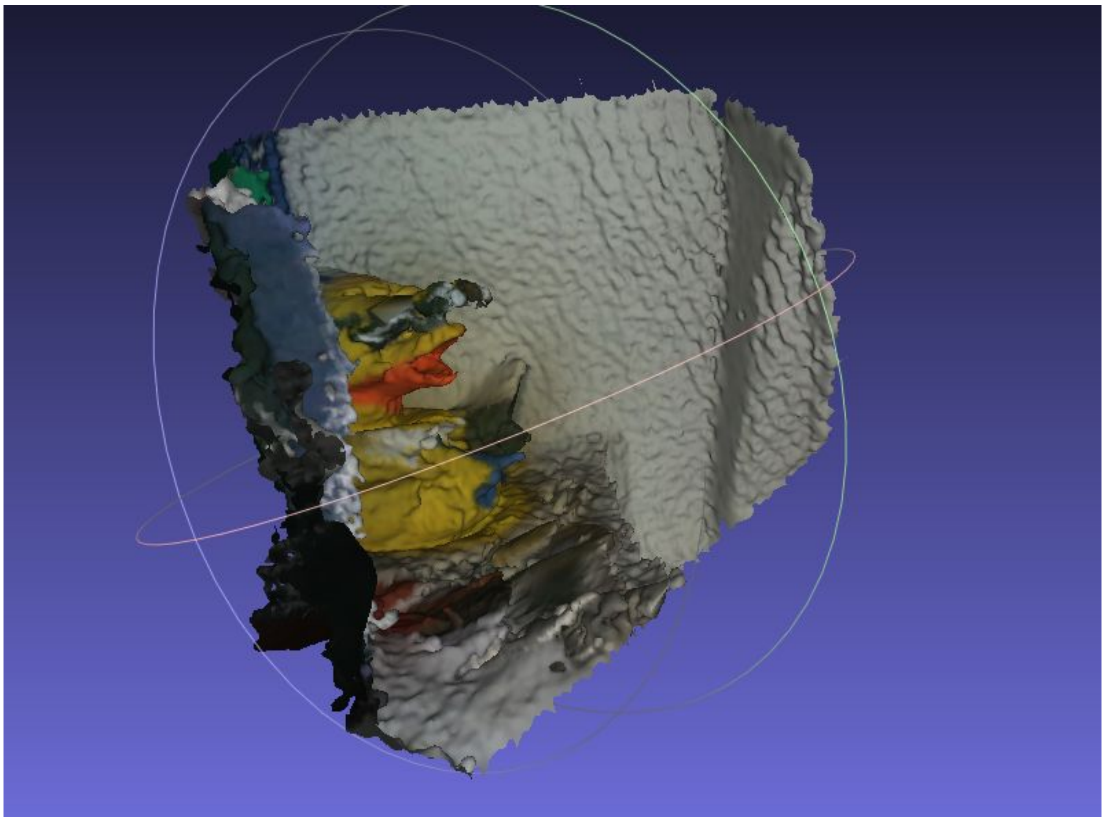


Figure 8: Example 2 - Reconstruction of the Corner of the Room(Without any segmentation)

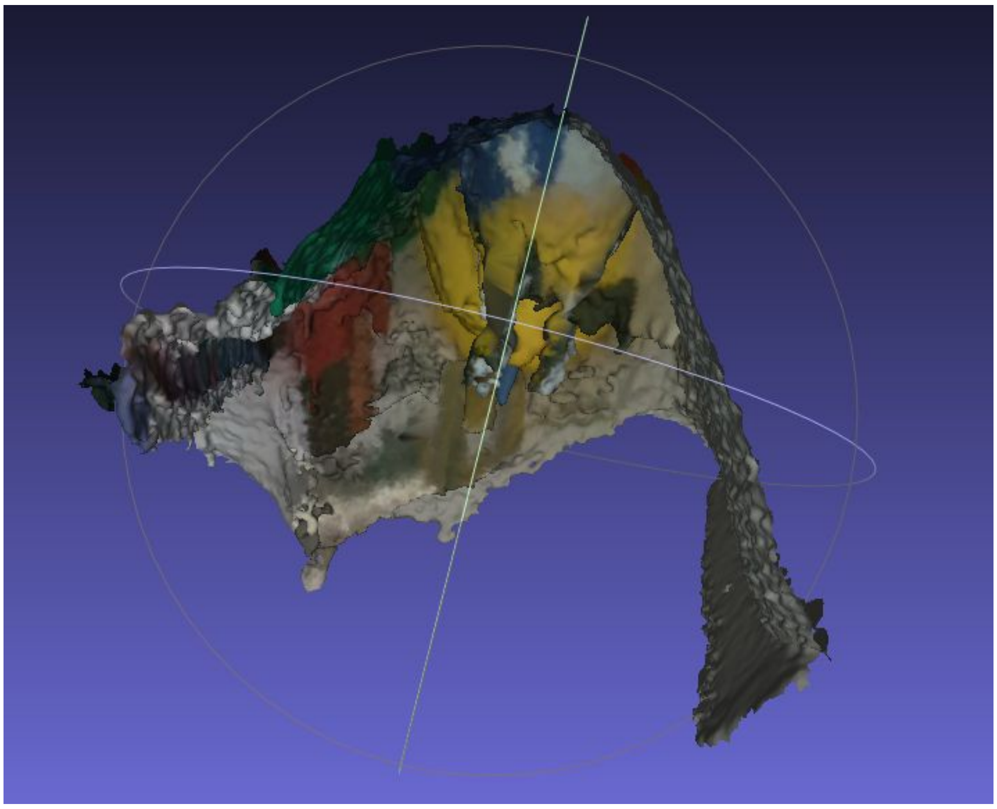


Figure 9: Example 2 - Reconstruction of the Corner of the Room(Without any segmentation)

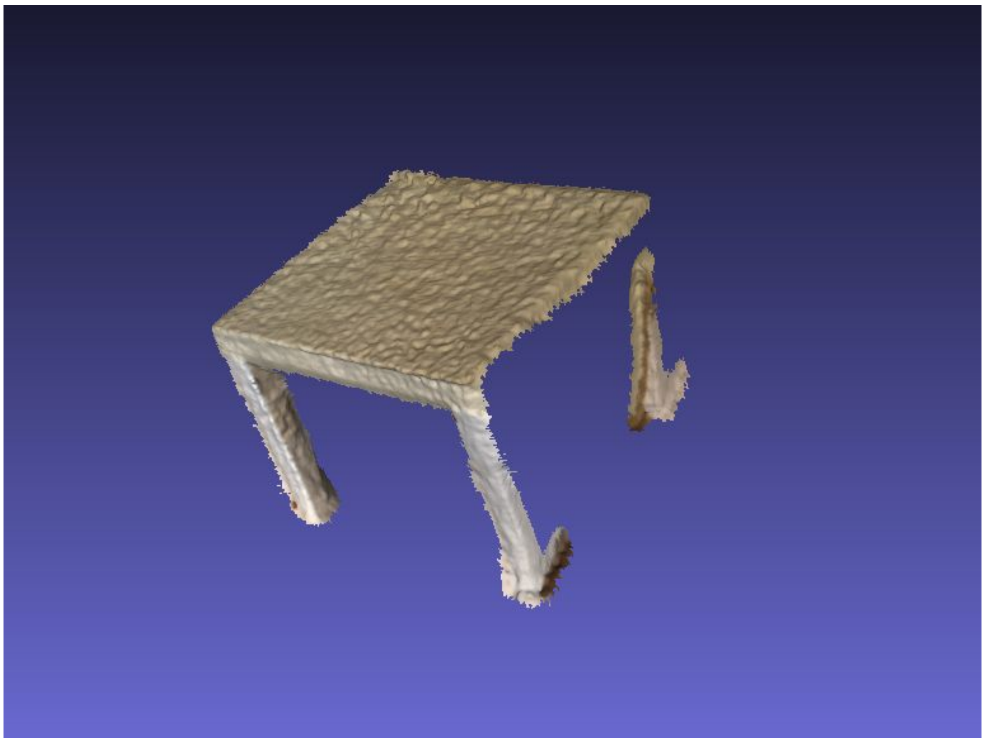


Figure 10: Example 3 - Reconstruction of a table (Original images are taken from this angle)

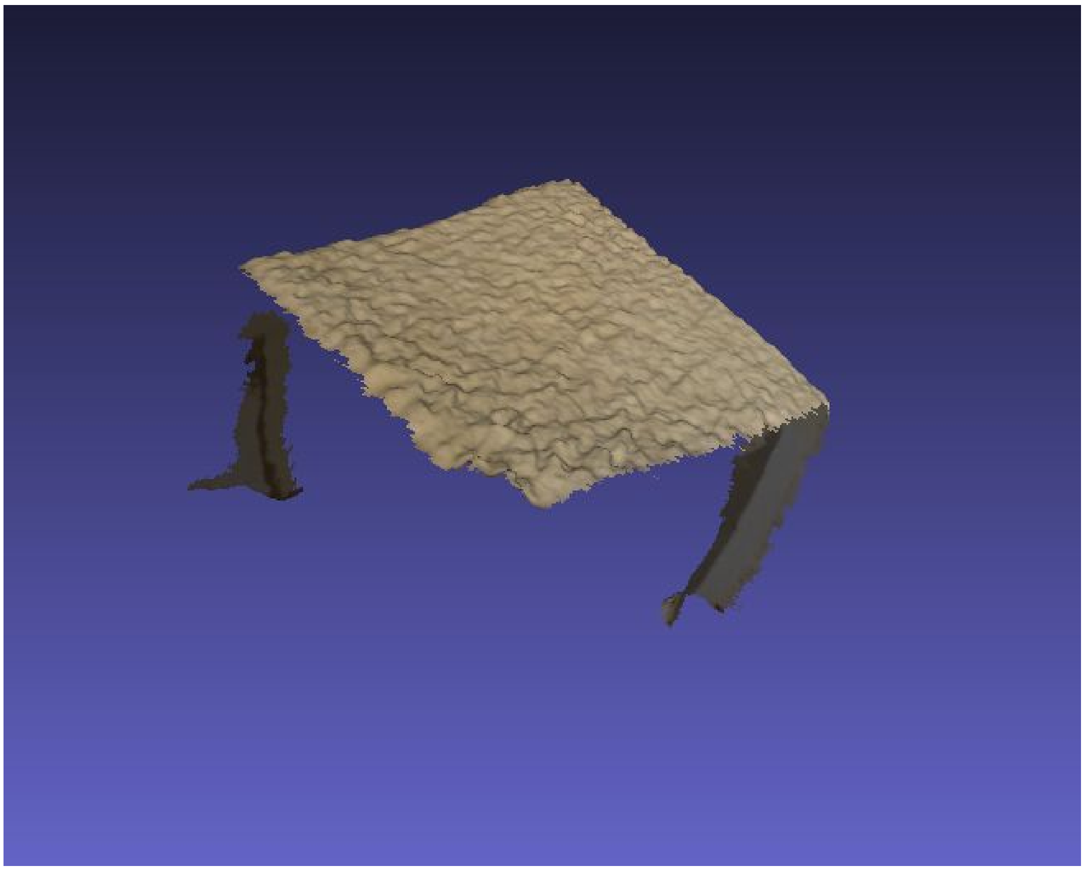


Figure 11: Example 3 - Reconstruction of a table (Backside not completed)

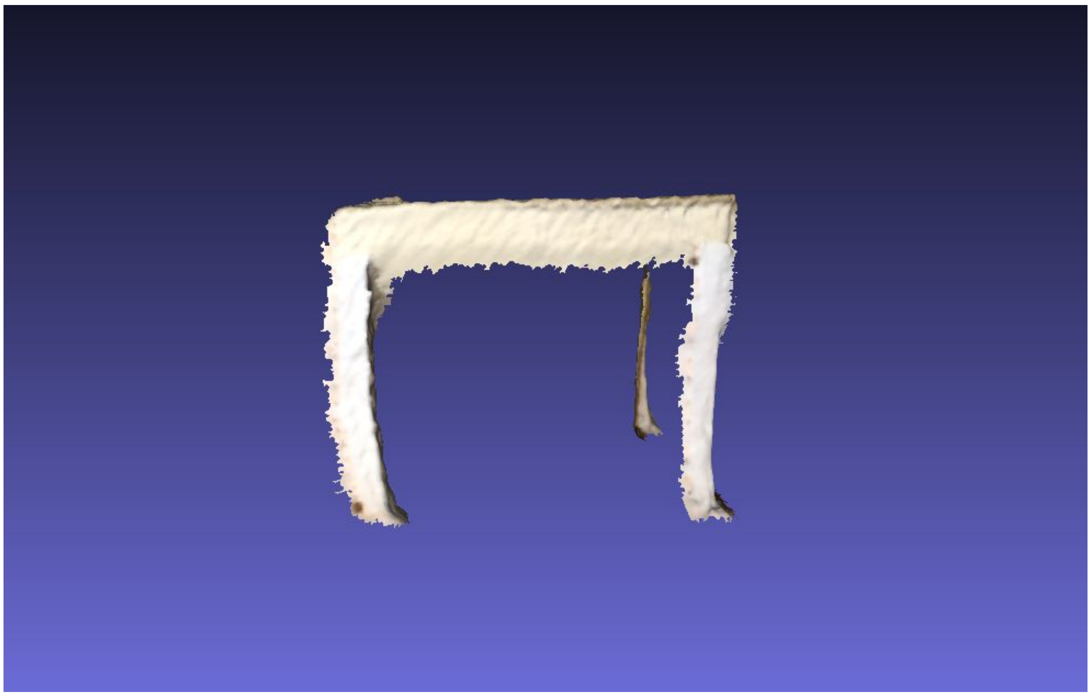


Figure 12: Example 3 - Reconstruction of a table (Front view)

4.3. Unity Application

4.3.1. Database and UI

Two spaces are created in Google Firebase Storage, one is public and for the workshop feature of MoveIt and the other is private and for the user accounts. Each user has their own storage folder with their unique user id where we store their owned objects. The database is used for downloading the desired furniture from the workshop and updating users' database accounts by uploading the furniture to their account. There is also a sign out button in the home page that deletes all the objects from the computer memory for privacy so that multiple accounts can use MoveIt on the same PC. For the workshop, only the icons of the objects will be downloaded to reduce the internet usage and the loading time of the application while signing in. Objects and icons are labeled according to their furniture types and the number in their class, and also a list of objects and icons are used in order to correctly download and upload all the objects since Firebase wants the name of the object that is to be downloaded as an input. If the workshop content is downloaded for one computer, it will not be deleted even if the user is signed out and it will not be downloaded again since it is public and to reduce the internet usage. Also a search bar for the workshop is implemented to search the object according to their labels like "table" or "chair". The objects are placed into different folders in the database according to their labels to keep the database clean and organized. Also the naming convention of the objects is like "table_1" if they are placed by the admins and "recon_table_1" if the object is the output of the reconstruction to change their materials and shaders in a script of the PC application.

Two basic input fields are used for the login page of the PC application. If the email or password of the user is wrong, required error sentences are shown in the screen. After the login page there is a loading screen which downloads all the icons in the workshop and the user account contents. Since Firebase is asynchronous, all the operations are checked via several flags before exiting the loading screen in order to prevent upload function before downloading. For the workshop, all the icons in the database workshop that the user does not have on their account are shown, and after the user downloads them, they disappear from the workshop's UI with the update function of Unity. While designing the workshop, a similar approach with the inventory system in the PC application is followed. An initial button is created, the button is cloned as the number of objects in the workshop folder of the user, the initial button is deleted and the icons of the objects are placed according to their orders in the folder. A hashing function is used to match the objects in the database and computer for this step. Also, the search bar in the workshop looks for an input in every frame, and if there is an input, it updates the frame on the workshop. After downloading the object in the user's folder on their personal storage, the user's folder in the database, their list of objects and icons and the content of the workshop in the PC's storage is also updated. For the scenes menu, a different scene from the PC application, "My Bedroom", is created to show an example room design, however, the objects in that scene cannot be manipulated since all the properties of the objects are initialized at run time. UI scenes of the PC and VR application are the same except their places in the space and designed as different scenes each connected to the database for the authentication.

4.3.2. PC Application

PC application's implementation is done using C#. A simple room layout is created using the interface Unity provides. The walls, floor and the ceiling are all fixed to a place and have a box collider which is used to interpret the collusion of it with different objects. All of these are game objects defined by Unity.

For the lighting of the indoor area, a point light is used with white color in order to ensure the comfort of the user. This light also allows us to create shadows of objects as well as the player model using Unity's own implementation. Similarly, the character controller prefab of Unity is used to implement the movement of the character. This player game object has several scripts attached to it which controls the speed of the movement, camera rotations, opening & closing of the inventory and the visibility of the cursor.

There is also an invisible canvas attached 5 meters from the camera. It is opened by the 'I' button from the keyboard, and it displays every object downloaded from the database as buttons. These buttons have their images set as the icon of the furniture which is also downloaded from the database. Upon clicking any of these objects, the object is added to the current scene in real time. Since Unity game engine does not support adding objects in real time by default, we used an asset to add them in real time. [14] This asset worked with a few problems but tweaking the script of it solved most of the problems. When an object is added to the scene, a script called "Object_Script" is added to it. This script contains every action an object is capable of doing such as rotation, movement, collusion, size etc. It scales all the objects at the same size initially since there will be many objects in the workshop that come with various ratios of sizes. However, the size of the objects can later be changed from the keyboard if the original size of the object is too small or too large. This script also controls whether an object is grabbed or not grabbed by updating a parameter in the player control script. This ensures that only one object can be grabbed at a time.

The “Object_Script” also checks it’s colliders information in real time in order to prevent the user from releasing a grabbed object if it collides with another one. This collusion check is not costly since the usual collusion check is done by Unity Game Engine itself and we are only manually checking one of them when it is grabbed. Finally, this script also updates the texture and the renderer of the object to make it transparent when it is grabbed or make it red when it collides with another object.

4.3.3. VR Application

The overall layout for the VR application is quite similar to the PC application as most of the scripts written for the PC application were reused in this part. Different from the PC application, instead of fixing the inventory canvas to the camera, it is placed on a wall due to health issues which will be further discussed in section 7.3. Since the user cannot have a mouse+keyboard in a VR environment, all of the controls were adjusted to the Oculus Touch Controllers using the asset Oculus provides for the Unity Game Engine. [15] However, some of these scripts provided by Oculus do not work as intended so some of them were adjusted to fit the application. The prefab used for this interaction is named UIHelpers and it contains laser pointers which are used for both object and inventory interactions and an EventSystem which manages its interactions with other objects, which is practically an event manager.

For the control of the player, again, a prefab of Oculus is used with slight manipulations. This prefab is called OVRPlayerController. It contains 3 child objects, 1 by default and 2 modified, which are called ForwardDirection, which keeps the information of head’s forward direction’s position, Grab Manager, which is used for grab operations, and OVRCameraRig, which is a default game object that comes with the prefab. The information kept in ForwardDirection is used to synchronize the head and body rotations with the OVRCameraRig and OVRPlayerController.

The OVRCameraRig has a few scripts attached to it to ensure it’s behaviour and a Physics Raycaster is also attached to allow the laser pointers to interact with the objects in real life. OVRCameraRig has only 1 child which is named as Tracking Space. This is a wrapper object for the LeftEyeAnchor, RightEyeAnchor, CenterEyeAnchor, TrackerAnchor, LeftHandAnchor, RightHandAnchor and the LocalAvatar. The left and right eye anchors keep track of what is displayed from the screens in the VR headset, whereas the CenterEyeAnchor keeps track of what the human eye perceives from these two screens and displays it on the monitor as the third camera. These game objects were used without any changes from the original prefab. The TrackerAnchor keeps track of the real life movements using the gyroscope and the accelerometer in the headset. Again, this game object was not modified but the information is used to update the position of the OVRCameraRig as well as the OVRPlayerController.

The LeftHandAnchor and the RightHandAnchor game objects are again wrappers used to contain the LeftControllerAnchor/RightControllerAnchor as well as any customizations within the hands. In this project, the hands were not heavily customized so only a sphere with a collider is added as hand anchors’ child and a script called OVRGrabber is attached to them to enable grabbing operation for objects. Finally, the local avatar enables the user to see their hands from any camera.

Unlike the PC application, there are 3 ways to manipulate objects in the VR application. The first one is the grabbing operation, which is ensured by adding the OVRGrabbable script to every grabbable object. However, the default version of the OVRGrabbable does not allow us to set multiple grab points in runtime so that script was slightly modified to ensure that it can be set in the runtime. The second way to manipulate them is pulling them towards the user. During the implementation stage, the DistanceGrabber/DistanceGrabbable scripts that were provided by Oculus were tested, however, they did not work as they were not suitable for objects which consist of multiple parts (e.g. a table which consists of 5 parts, 4 legs and 1 top). As a solution, the prefab of the laser pointer was slightly modified to recognize objects and if it collides with an object while the user presses the grab buttons, the object is pulled towards the player.

The last way to manipulate objects is the manipulation mode. In order to enter this mode, the user needs to point towards an object similar to pulling distant objects and press enter manipulation mode button. In this mode, similar to the PC application, the user can adjust the height of the object via a thumbstick and freeze,rotate and scale an object as they see fit. This implementation is done in the "Object_Script_VR" as well.

The VR application is much more free in terms of movement of objects as well as the user due to the health issues caused by the headset. To sum up, every movement the user does in real life must be translated to movement in the application. Otherwise, it would cause the user to feel nauseous since the brain expects to see the result of the movement and it does not happen in the application. [16,17,18,19,20]

5. Testing Details

5.1. Mobile Application

We performed several tests on the Mobile Application. First of all, we checked every function with unit tests and ensured that they are working as intended. After this we conducted experimental tests in our rooms. We overlaid the depth map received from the camera over the actual image and manually checked if the depth values matched the actual results. We saw that while our application works as intended, the depth camera can sometimes output noisy and erroneous reading due to many external factors. These factors include: the color of the object(darker objects tend to observe the rays produced by the camera which stops it from accurately determining its depth), light level of the room, exterior of the objects such as how smooth or bumpy it is, reflectivity of the object(mirrors produce very wrong results) and the motion of the phone during capture(if the user moves slowly then depth readings are more accurate). We tested different video durations and found out that due to the depth camera’s high use of storage space, the data size started to be a problem for phones with smaller storage spaces.

We also tested our mobile application on two different phones which are Samsung Galaxy Note 10+ and Samsung S20 Ultra to ensure that a wide selection of users can use our application.

5.2. Reconstruction

For the reconstruction, boundaries of the segmented objects, types of the objects, the accuracy of the 3D projection process were checked with different videos recorded in different times of the day and in different environments to ensure generalized reconstruction results. We first manually checked the accuracy of our model used for the semantic segmentation and compared it with the advertised pixel accuracy of the ResNet101dilated model which is 80.91%. We found that the indoor segmentations of our pictures worked a bit worse than the advertised general version since the ADE20k dataset also contained many outdoor objects and animals.

We used Matlab scripts to ensure that we have a working camera calibration algorithm and a separate algorithm to align our depth and rgb images on top of each other(Sensors used to capture these images have different field of views and other intrinsic/extrinsic parameters so we need to perform image rectification on them using these parameters) [21].

After this, we created many objects from the previously mentioned videos and performed mesh filters on them to adapt them to our PC application. We manually checked how objects turned out in a software called Meshlab [22]. Depending on the result, we adjusted some filter parameters and got rid of some of the erroneously reconstructed objects. These errors had one main reason, the wrong classification of the seen object which is unavoidable even for highly accurate classification Neural Networks such as ResNet101dilated.

5.3. PC and VR Application

We performed several tests on both PC and VR applications with different scenarios and checked every function with unit tests. We checked movements of the users and the furniture, collision of the furniture both with the other furniture and the user, the actions that the user performs such as opening inventory or placing the furniture, boundaries of the objects and the boundaries of the room. These cases are repeated for PC and VR individually since their implementations are different. There were also VR only scenarios such as pulling an object from a distance or for the view of the user since it is different from the PC implementation and view. We also performed many tests for the database such as downloading and uploading different objects with different sizes or with different types of icons as well as checking the correctness of the authentication. As for testing the overall quality of the VR application, every one of us as well as most of our families tested the application for 30 minutes. This test ensured that there were no health issues regarding the VR headset when used for a long period of time.

6. Maintenance Plan and Details

For the maintenance of MoveIt, we plan to improve the scalability of our system that performs reconstruction since the number of users will increase. Reconstruction time will be decreased because more users will perform reconstruction and segmentation at the same time. In addition, the security of the database will be increased since there will be more users trying to get past the security and also scalability of the database will be increased as well. Since one of the main features of MoveIt is to be able to add other furniture from the workshop, the workshop should be kept clean from non-appropriate content and functioning all the time. If there are not enough people sharing objects for the workshop, we might share some furniture that we found from available datasets in order to encourage people to share. The data flow between the smartphone that records the video, server and PC will be increased since in the current status, the size of the video is very big like 1GB and transferring the video between the environments takes a long time. There are still problems with the library that we used for the reconstruction and segmentation, so if we can find a more accurate library in the future, we might change our current libraries.

7. Other Project Elements

Below, consideration of various factors, ethics and professional responsibilities, judgements and impacts to various contexts, teamwork and peer contribution, project plan observed and objectives met, new knowledge acquired and learning strategies used will be discussed.

7.1. Consideration of Various Factors

MoveIt is an application that touches and enrichens various public issues and global factors. There is no distinct effect of public health issues on the design of MoveIt. Concerning the public safety issue, users can modify their room with the instructions given in an earthquake-safe object placement manual. Before they do this modification, they can use MoveIt to find the best possible furniture arrangement and visualize it with ease. Since the aim of the application is providing its users with a convenient way of visualizing their rooms and changing their furniture arrangements in various ways, MoveIt enables the users to plan ahead without doing the actual work and work smarter instead of harder. This makes MoveIt an excellent application for ensuring public welfare and comfort. In terms of global factors, MoveIt is a convenient medium for furniture stores to promote their merchandise and provide their customers with additional features by using the application. This usage contributes to economic factors and makes the design of MoveIt affected by global factors. With its publicly available dataset in the workshop, MoveIt will enable users from all around the world to share their furniture as well as culture with the rest of the users. People will be able to see different objects that they have not seen in their living spaces and cultures which makes cultural factors play an important role in the design of MoveIt. The users sharing their objects with the rest of the world is also going to help other people to design their rooms which contributes to social factors and bores the need to take social factors into account when designing MoveIt.

	Effect level	Effect
Public health	0	None
Public safety	3	The need to place furniture in a natural disaster-safe manner
Public welfare	8	People can change their room arrangement without actually changing it in real life

Global factors	5	Furniture stores can market their furniture which contributes to economic factors
Cultural factors	4	With publicly available dataset, people can see furniture from other cultures, and they can share their own furniture with the world
Social factors	6	People could share their objects so that they can help other people design their rooms

Table 1: Factors that can affect analysis and design.

7.2. Ethics and Professional Responsibilities

Since MoveIt requires scanned furniture of users' rooms that they want to change the appearance, we store every furniture scanned by every user in the database with their labels. However, these objects are stored anonymously without any of their user information. This way, we allow the use of non-owned furniture in the application while securing the users' privacy. The models and labels will not be shared by other users or third parties without getting permission in the case that companies want to use them for ads, and will just be kept in the database. The furniture will be uploaded to the workshop if and only if the user wants to do so.

Required permissions such as accessing phone's storage and camera are taken from users before using the application for the first time and the models will not be saved, or the camera will not open if they do not give the required permissions. No unnecessary permission is asked. The addition of the scanned models to the dataset will be asked after each scan. This would allow the user not to upload their specified models if they wish to do so. However, they will not be able to use the application if they do not give some permissions such as accessing the camera.

Since all the objects are kept in a database, users will be able to access their data via an account from the MoveIt. No personal information will be asked during the signup phase, and only an email and a password will be required. All the scenes/videos and objects are uploaded to the database according to their user's hashed user id and will be kept private but this can be changed with users' permission. User information will not be shared by third companies.

While constructing the datasets for meshes, copyright and privacy issues will be considered and for all purposes, the National Society of Professional Engineers' Code of Ethics will be followed [23].

7.3. Judgements and Impacts to Various Contexts

Judgement Description:		Designing the Workshop as a platform where every user can contribute and share
	Impact Level	Impact Description
Impact in Global Context	5	With publicly available dataset, people can see furniture from other cultures, and they can share their own furniture with the world
Impact in Economic Context	5	Furniture stores can market their furniture which contributes to economic factors
Impact in Environmental Context	3	Stores can use MoveIt workshop feature to showcase their furniture and they do not need to use paper print catalogs
Impact in Societal Context	6	People could share their objects so that they can help other people design their rooms

Table 2: Judgement and Impacts 1.

Judgement Description:		Implementing PC
	Impact Level	Impact Description
Impact in Global Context	4	People from parts of the world with no access to VR devices can use MoveIt through the PC application
Impact in Economic Context	6	People do not need to purchase VR devices to use MoveIt
Impact in Environmental Context	2	People don't need to buy VR devices to use MoveIt, therefore additional production of VR devices do not happen because of MoveIt
Impact in Societal Context	3	People from parts of the world with no access to VR devices and people with VR devices can use MoveIt and share their furniture with Workshop, which creates a cultural bridge

Table 3: Judgement and Impacts 2.

Judgement Description:		Making MoveIt free to use and open source
	Impact Level	Impact Description
Impact in Global Context	4	People that do not have the means can use MoveIt. Since MoveIt is open source, developers from around the world can contribute to the source code

Impact in Economic Context	6	People do not spend money to buy MoveIt
Impact in Environmental Context	0	-
Impact in Societal Context	3	People from all sorts of financial backgrounds can use MoveIt which creates a bridge between societies with different backgrounds. Developers from all around the world can work on MoveIt source code together.

Table 4: Judgement and Impacts 3.

Judgement Description:		VR Implementation
	Impact Level	Impact Description
Impact in Global Context	0	
Impact in Economic Context	2	People needs to buy a VR headset to use the application
Impact in Environmental Context	0	-
Impact in Societal Context	8	Using the VR application is more entertaining and interactive than the PC application

Table 5: Judgement and Impacts 4.

7.4. Teamwork and Peer Contribution

All team members contributed in all reports and meetings, there were many development phases happening at the same time such as initial mobile application design and initial 3D reconstruction and segmentation research or PC application design and database design. We always worked together before the COVID-19 and used screen-share tools such as Discord after the virus as 2 or 3 person groups. Since we have small groups, if someone knew the implementation environment beforehand, that person acted as a leader for that phase such as Faruk was the leader in the PC application design since he worked on Unity before, Pınar was the leader for database design, Mert acted as the leader for finding utilising a pre-trained neural network, Ünsal was the leader for mobile application design and Barıș was the leader for UI design for Unity. None of us were familiar about the 3D reconstruction before, so that phase was much more complicated and longer than all other parts. Therefore, the leader role was changing between Ünsal and Mert. We also did peer-coding since we did not know the environments well and for example, Barıș and Faruk worked on Unity as peer-coding and after Pınar completed the database for the mobile application, Barıș helped her to understand Unity. In the meantime, Mert and Ünsal worked together on the C++ project for the 3D reconstruction and mesh creation and Faruk completed the VR application. So peer-coding worked well for us. Below, you can find the overall work done for each group member.

Barıș Can: Initial mobile application design, PC application and UI design, initial VR application design, database design for PC, Mobile application design.
Faruk Oruç: Initial 3D reconstruction and segmentation research, PC application, VR application implementation and UI design.

Mert Soydinç: Initial 3D reconstruction and segmentation research, depth camera implementation, 3D reconstruction and segmentation implementation. Point cloud to mesh reconstruction.

Pınar Ayaz: Initial mobile application design, initial 3D reconstruction and segmentation research, database design for Mobile and PC, PC UI design and Mobile application and UI design.

Ünsal Öztürk: Initial 3D reconstruction and segmentation research, depth camera implementation, 3D reconstruction and segmentation implementation. SyncCamera and 3D-Reconstruction library implementation.

7.5. Project Plan Observed and Objectives Met

We completed almost all the work packages we mentioned on our Analysis Report. Namely, 3D Reconstruction and Semantic Segmentation Research, Render and Unity Engine Research, High-Level System Design, Initial Implementation of the Android Application, Semantic Segmentation and 3D Reconstruction from Videos, Low-Level System Design, External Subsystem Design and Implementation, PC Application Implementation, VR Application Implementation, Final Implementation of the Android Application were completed in order, and we are currently on the Final Implementations and Polishing work package. There were some changes on applications caused by the implementation process since at that point, we did not start implementation but did some initial research. Changes such as using a depth camera smartphone, some UI changes from initial mockups and database changes were made. We also tried to apply our initial project plan from the Analysis Report and you can find the completion order above. After the initial research, we started our mobile application implementation since a camera is needed for the 3D reconstruction process and after the camera and initial UI is designed, 3D reconstruction implementation started since it was the most time-consuming process. The camera was changed afterwards according to inputs of the library we used for the reconstruction phase. Then, while the database developed for mobile application, PC application was developed. After PC application completed, VR application and UI design, final mobile application design, website design and the final report were developed at the same time since only one of us had the VR. One main aspect of the project that we could not implement due to time constraints is the prediction and filling of the occluded surfaces of objects(such as the back of a cabinet). We intended to use a Neural Network specifically trained on this task using partially seen objects with labels with supervised-learning. However, such a dataset does not exist which meant that we needed to create it ourselves. Which would have taken a lot of time and effort. In the end, time constraints kept us from implementing this feature.

7.6. New Knowledge Acquired and Learning Strategies Used

In order to recreate the 3D structure of the rooms, we particularly used papers from the Technical University of Munich, namely “Efficient Online Surface Correction for Real-time Large-Scale 3D Reconstruction”, for the 3D reconstruction stage as a reference[24]. We learnt the logic behind the 3D reconstruction through our research and we adapted them to our project. We created some basic 3D point clouds of individual objects using the color and depth information of their pictures. Then we applied several filters on these point clouds in order to transform them into proper semi-watertight meshes. These filters include computation of normals and smoothing of said normals, poisson reconstruction to create a mesh, hole filling and removal of the disjunct parts from the mesh,laplacian smoothing and finally the transformation of the initial mesh into a pure triangular mesh. We learned about what filters to use and their mathematics behind them from the Meshlab Tutorials[25]. We also extracted texture information from these meshes using their vertex colors. To find boundaries for household objects given a series of frames, we studied the paper “Real-Time Dense Geometry from a Handheld Camera” for guidance[26]. The contents of this paper greatly match our own goals as they also aim to create a 3D depth map for images and reconstruct them using this data from a set of images.

For the PC phase of Unity, we watched some tutorials such as how to build an inventory system and tried to apply them on our system. Since some of us never worked on Unity before, and the others did not work on it for a long time, at first, we did not know anything and we generally read Unity forums to solve our bugs and add game mechanics such as movement, grabbing objects. Again, Unity was well documented and we could easily find which methods, variables we should use from Unity documentation. For the VR phase, we had even fewer sources since VR is not popular as much as PC and Unity documentation did not include the VR environment. There were some tutorials and games created by the Oculus Rift developers to tell the users how the VR works, we watched them and tried to understand how we can implement them. We read the codes of the tutorials, games and prepared scenes to see if we can use or modify them and which functions should we change that we already have.

For the mobile and database phases, we knew Firebase, but we have not tried to upload or download objects, icons via C# and Java code. Firebase for Unity was still in beta phase and there was no source on how to implement it. We simply follow the authorization, download and upload tutorials of Firebase from their website and it worked[27,28]. Moreover, some of us also did not know how to work on Android Studio, but we again watched some tutorials, search on the forums and follow Firebase tutorials for the database. Mobile phase also took considerable amounts of time since the access to the depth camera API was very new to us and it did not have widely used examples. However, This Camera2 API for Android Studio is very detailed in its documentation so it was possible for us to learn about its different functions and data types and then use them to achieve our goals. In the end we endep up implementing a joint color-depth image acquisition library which can be used separately from this project in the future.

8. Conclusion and Future Work

By developing MoveIt: Indoor Manipulation as our final project, we aimed to bring our homes into virtual reality by recreating 3D indoor scenes using a smartphone camera. With MoveIt, a quick scan of a room brings the objects contained within this room into VR where they can be freely moved, interacted with, and manipulated. It is also a helpful tool that enables the user’s aesthetic ability by allowing them to redecorate their room with ease while also providing functional help such as object boundary and collision detection. This 3D recreation of the room is visualized in PC and VR environments that enable the user to freely move, scale, and rotate objects in the 3D reconstruction of the room using PC Keyboard or VR Headset.

In this report, we presented the final architecture and design of our system as well as the final status of our project. We discussed our refined analysis and requirements, and the details of our design. We included the details about our implementation and tests, and our maintenance plan for MoveIt. We also talked about how various factors affected our project design and development and we discussed the ethical and professional responsibilities that we recognized, observed and fulfilled during our project. We mentioned which informed judgments we made in our project that consider the impact of our solution in global, economic, environmental, and societal contexts. All team members contributed in all reports and meetings, there were many development phases happening at the same time such as initial mobile application design and initial 3D reconstruction and segmentation research or PC application design and database design.

Considering the future work, we plan to improve the scalability of our system that performs reconstruction by employing a Microsoft Azure remote server. Right now, we are using our own computers to act as our remote server which impacts the scalability of the application. By employing a much more powerful server we will be able to achieve the following. Reconstruction time will be decreased because more users will perform reconstruction and segmentation at the same time. In addition, the security of the database will be increased and also scalability of the database will be increased as well. Since one of the main features of MoveIt is to be able to add other furniture from the workshop, the workshop should be kept clean from non-appropriate content and functioning all the time. The data flow between the smartphone that records the video, server and PC will be increased since in the current status, transferring the video between the environments takes a long time. Moreover, we need a depth camera in the current condition and number of phones that have a depth camera is still low today. Therefore, if we could manage to fetch depth information from a normal camera by collecting data from different angles, or if we could find a library that does not require a depth camera in the future, then we might change those libraries or implementations as well.

In addition to this, we plan to create a partially constructed object dataset using our current applications output and manually fill the obstructed sides of these objects. This dataset then can be used to train a Neural Network which completes the objects by itself. However, one big problem with this approach is that we will need a large number of different data objects in order to train a model of this complexity which can prove to be infeasible.

9. Glossary

9.1. Terminology

DEPTH16 Format: The DEPTH16 format is a depth storage format implemented on mobile devices. It uses 16 bits per value, the first 3 bits being a confidence value, and the least significant 13 bits being the depth estimate in millimeters, all in big-endian fashion. Check the Android documentation for further details.

Kinect Depth Format: Similar to DEPTH16, kinect uses 16 bits to represent a value. The first 13 bits are the depth estimate in millimeters, and the last 3 bits are reserved for a device ID.

SyncCamera: An API serving as a wrapper around Android's camera2 API, implemented by us. Visit the GitHub page in the references section for further information and the source code.

3D-Reconstruction: An API to reconstruct semantic 3D point clouds. Visit the GitHub page in the references section for further information and the source code.

MeshLab: A free Mesh-editing tool which can be configured to be used programmatically through a locally run server, MeshLab server, for point cloud to mesh conversion. See the references for further information on MeshLab.

Tinyply: A C++ single header library to read and write images in .ply format. For further information on the library, check https://github.com/ddiakopoulos/tinyply.

9.2. Software/Hardware System

For the MoveIt, required software/hardware components are:

- Oculus Rift for the VR feature
- An android smartphone with at least two cameras
- A computer which has Windows 10 operating system

Minimum requirements of the computer which will use the PC feature should be:

- At least Intel Core 2 Duo E6320 or equivalent
- At least 2 GB RAM
- At least GeForce 7600 GS or equivalent
- At least 1.5 GB available space

Additionally, minimum requirements of the computer which will use the VR feature should be:

- CPU: Intel Core i3-6100 / AMD Ryzen 3 1200, FX4350 or greater.
- RAM: 8GB.
- GPU: NVIDIA GeForce GTX 1050Ti / Radeon RX 470 or greater.
- GPU RAM: 8GB.
- Ports needed: 1 USB 3.0 , 1 DisplayPort

9.3. User's Manual

9.3.1. Mobile

Since the size of the recorded videos are too big and the server is just a regular computer with regular connection speed, currently it is impossible to upload them into the database, therefore the only functionality of the mobile app is to record videos. However, for this purpose, the depth camera component of MoveIt must be used. There will not be user authentication in this current version since we do not upload the objects or videos from the mobile application to the database. Users that have qualified smartphones should download the MoveIt application, press the record button in the main page, and record the video. Then the video will be saved in the internal storage and they could transfer the video using their USB-C cables and continue the process from the reconstruction phase until the server is implemented.

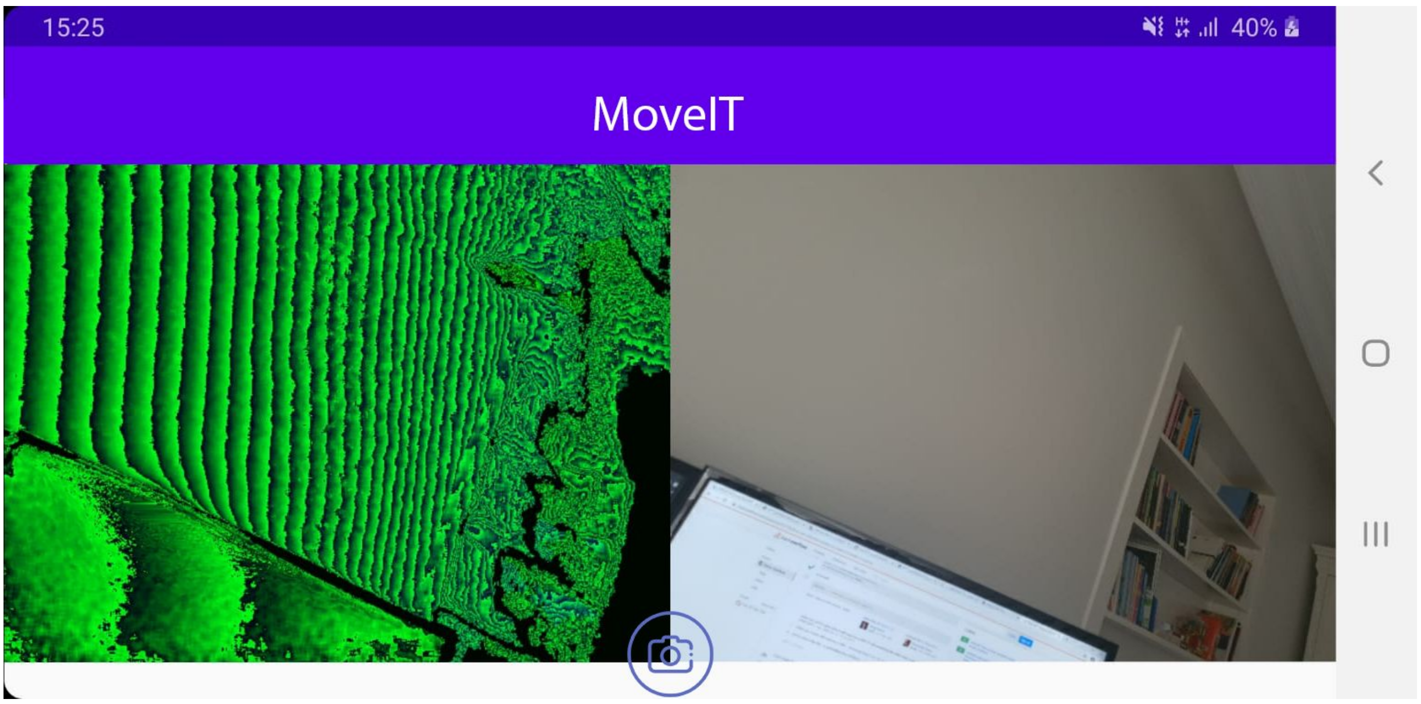


Figure 13: MoveIt Mobile Application

9.3.2. PC

Below, you can find the screenshots and user tutorial for the PC environment.

9.3.2.1. Login Page

Below you can find the login page for MoveIt. Here, users may sign up via their email and password. No additional information is required. After signing up, they can log in with their credentials and start using MoveIt. After logging in, there will be a loading screen where objects and icons that users have and icons that are placed into the workshop will be downloaded. The loading time might change according to the number of objects the user has and the internet speed of the user considering the size of one object is about 2-10 MB. The icons in the workshop will not be downloaded again if they are already on the local PC storage.

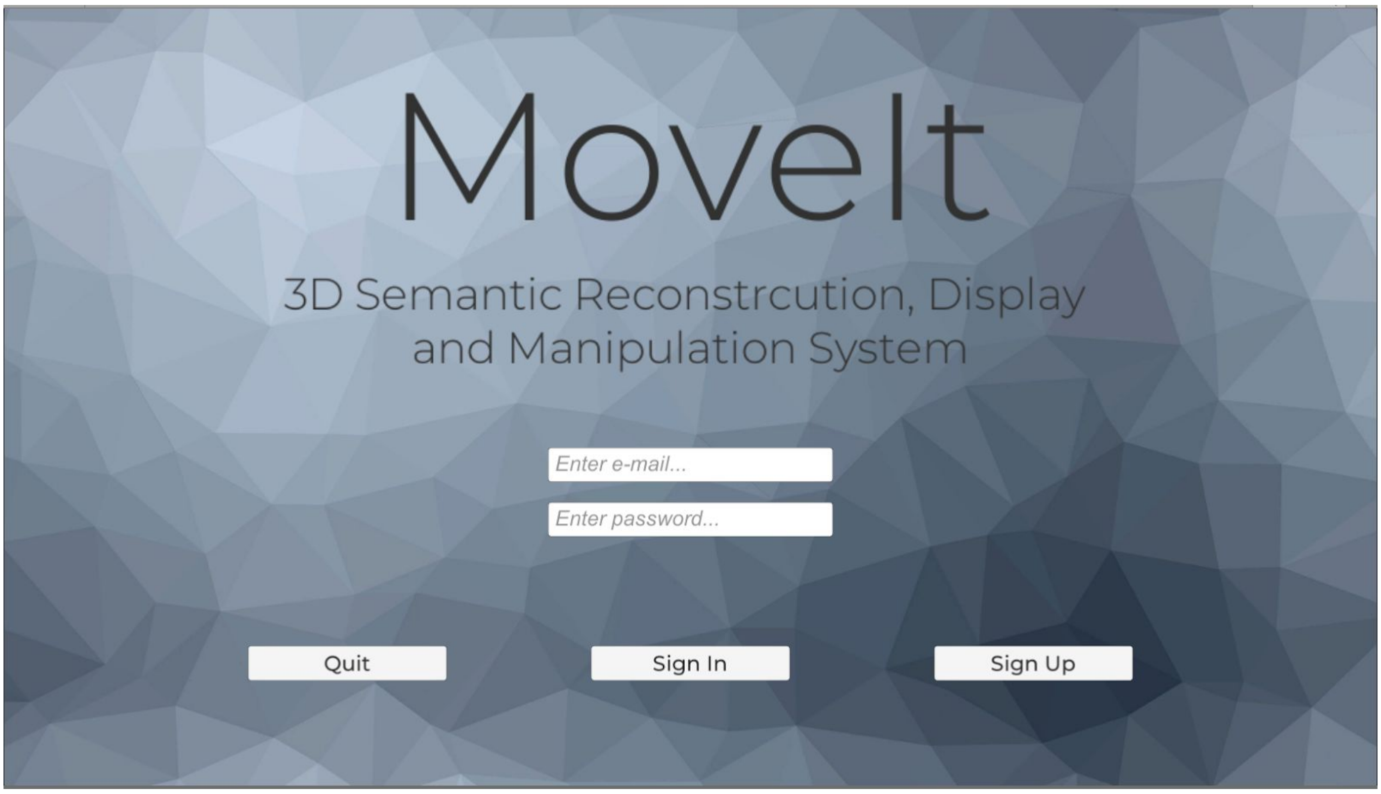


Figure 14: Login Page

9.3.2.2. Home Page

Below you can find the home page for MoveIt. This is the first page that users see after the login. Here, users may choose to play the game, see the workshop which allows them to see the publicly available object and add them to their inventory, and see their scenes. Later, users can see their object from the in-game menu. Users may also sign out which deletes all of their objects from their local storage if they use a public computer.

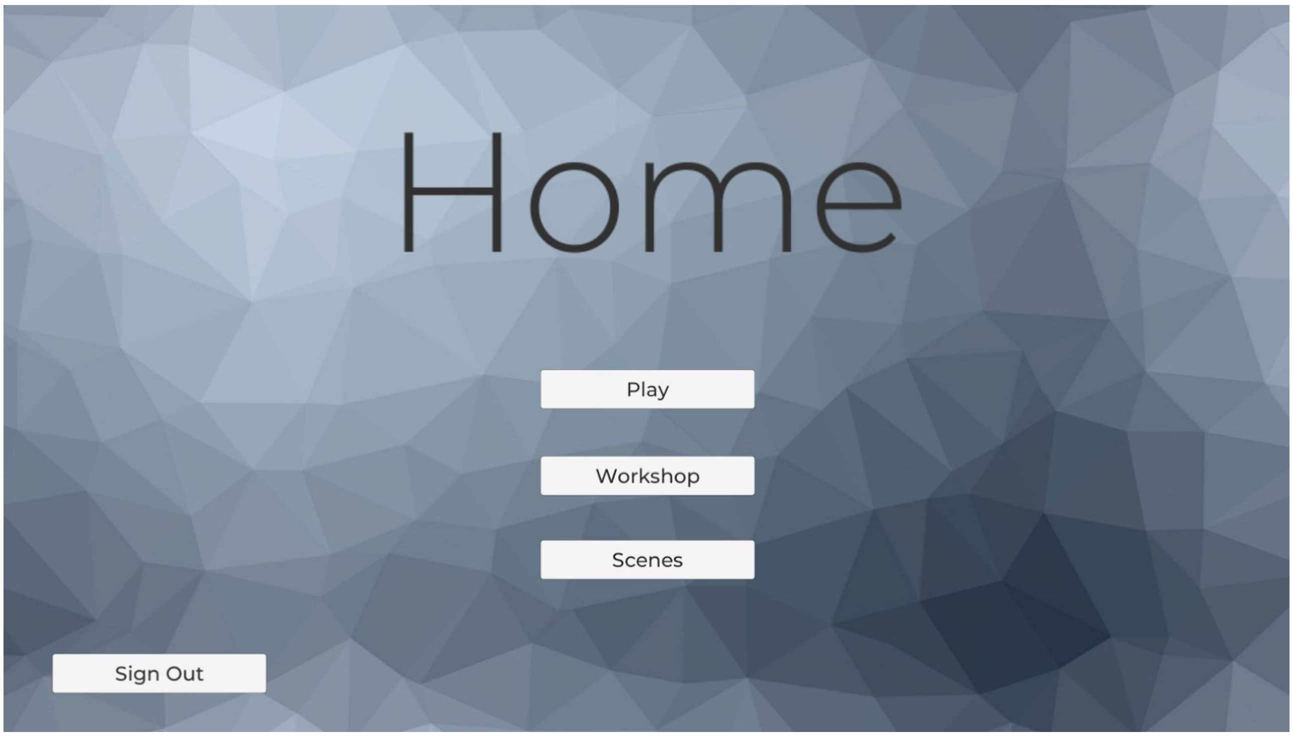


Figure 15: Home Page

9.3.2.3. Workshop

Below you can find the workshop feature of MoveIt. Here, users may see the publicly available objects and their icons. Users may search the categories such as “table” to see specific furniture. If users click on the objects, they will be added to their private accounts and they will be able to use them in the play screen with their scene. The back button will go back to the home page. If the object is already in the user's account, they will not show up in the workshop.

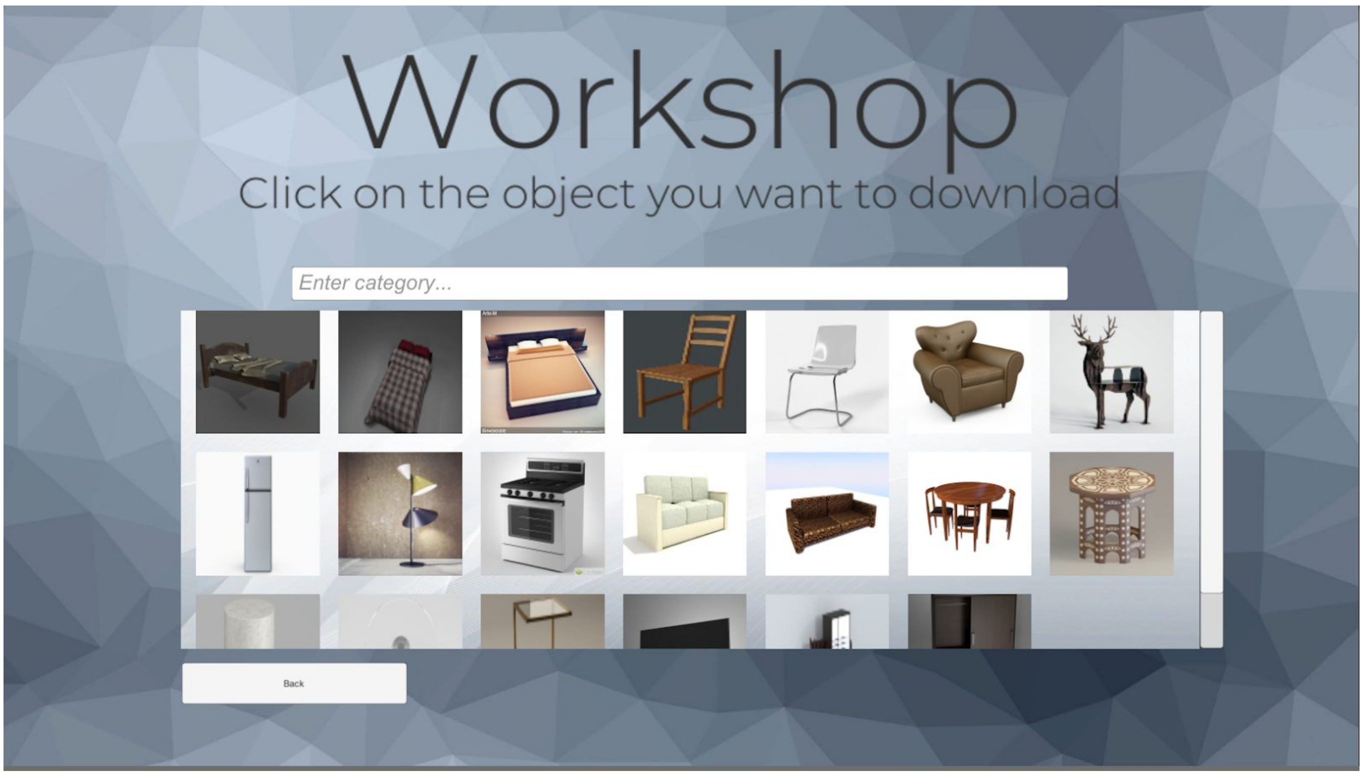


Figure 16: Workshop

9.3.2.4. Scenes Page

Below, you can see the scenes page of MoveIt. They can create custom scenes such as My Bedroom, or they can play with the empty room with the objects they already have on their accounts.

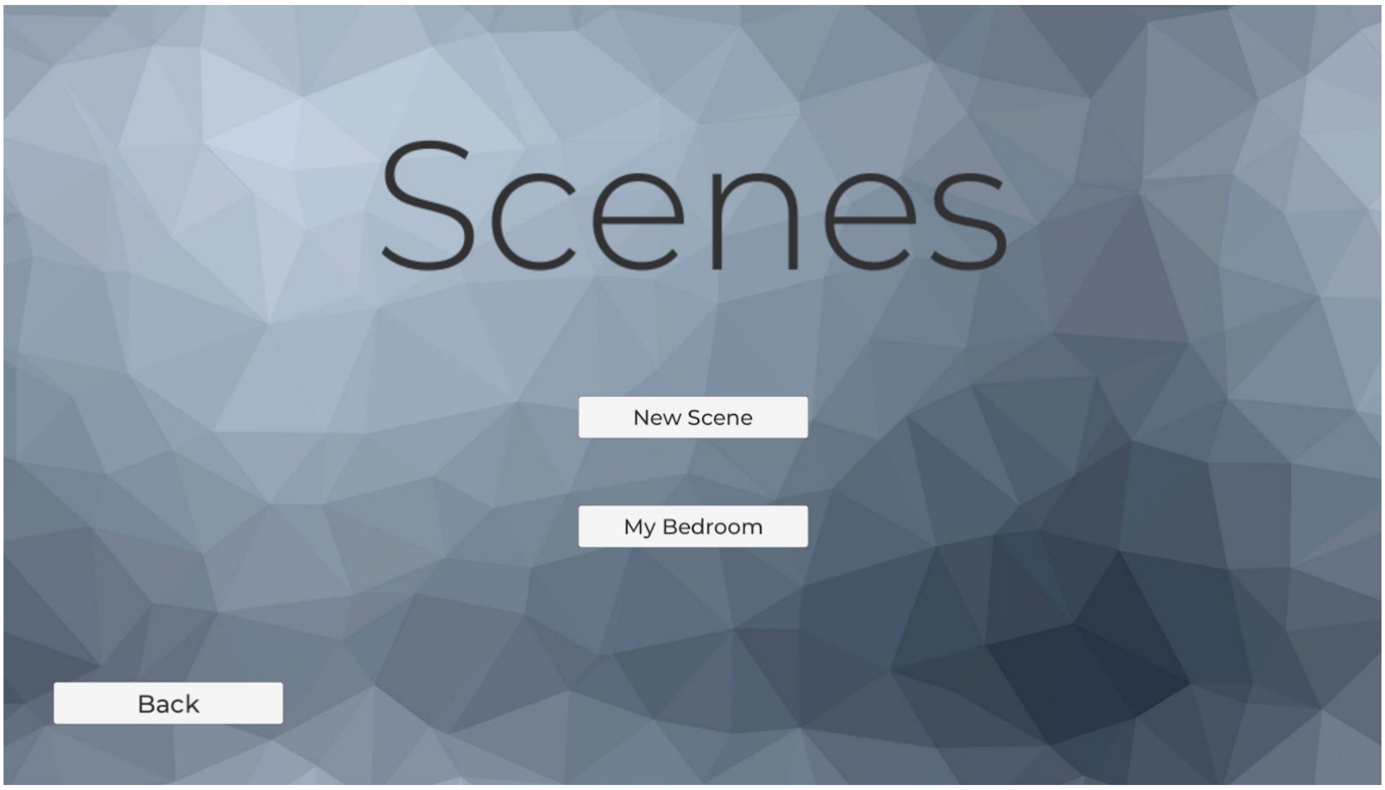


Figure 17: Scenes Page

9.3.2.5. Platform Selection Page

Below you can find the platform selection page of MoveIt. Here, users may choose to use MoveIt on PC or VR environments. If users press the back button, they will go back to the home page. It is also indicating that if the users presses the ESC button while in the inventory of the MoveIt, they will go back to the home page. You can also view the keybindings for relative platform by pressing the button below the platforms' button

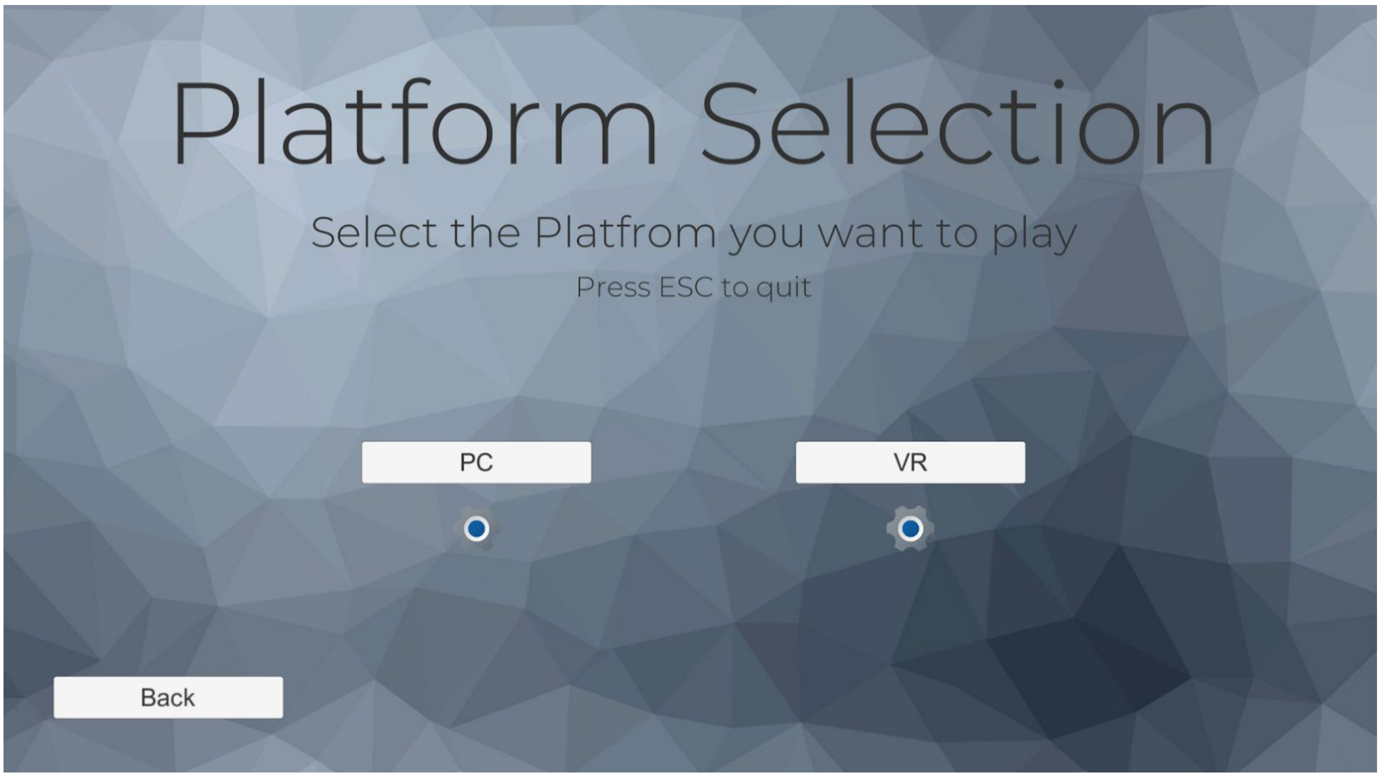
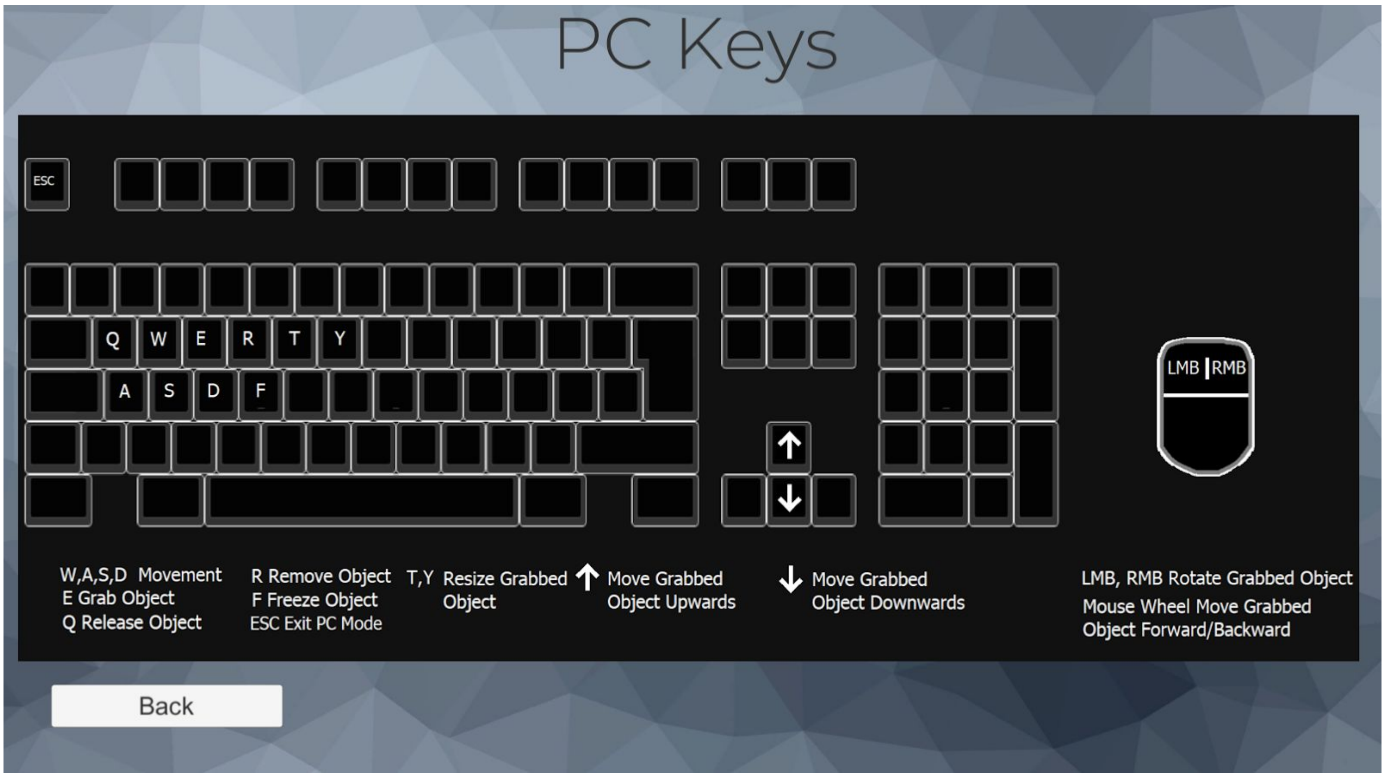


Figure 18: Platform Selection Page

9.3.2.6. PC Key Bindings

The key bindings for the PC application can be seen below which will guide the users.



W,A,S,D Movement
E Grab Object
Q Release Object
R Remove Object
F Freeze Object
ESC Exit PC Mode
T,Y Resize Grabbed Object
↑ Move Grabbed Object Upwards
↓ Move Grabbed Object Downwards
LMB, RMB Rotate Grabbed Object
Mouse Wheel Move Grabbed Object Forward/Backward

Figure 19: PC Key Binding Page

9.3.2.7. Play Page

Below you can find the workshop feature of MoveIt. Here, there are several functionalities. By using the buttons of the keyboard;

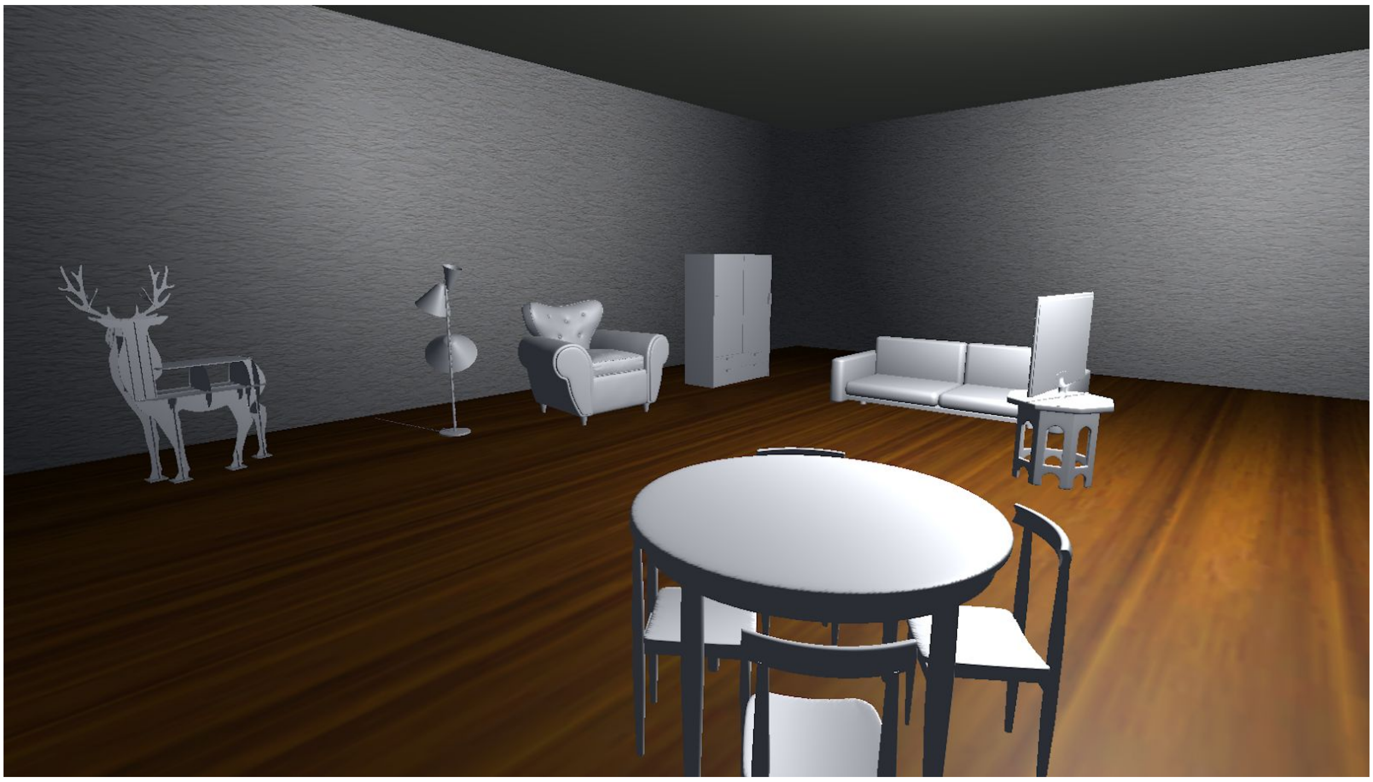


Figure 20: Play Page

- W button will move the user ahead, D button will move the user right, A button will move the user left and S button will move the user to behind.
- Mouse left-click will rotate the object in the Y-axis, right-click will rotate the object in X and Z axis and mouse wheel in the negative direction will decrease the distance between the object and the user and positive direction will increase the distance between the object and the user. While adjusting the distance, objects cannot collide with each other, and also cannot pass through walls. If they collide, the colour of the object will turn into red and it will indicate that the user cannot release

the object at that location. Also, mouse movement will rotate the user perspective.

- The I button on the keyboard will open the inventory. While in the inventory, the user cannot move, and the mouse cursor will be visible to the user to select items from the inventory. After selecting the object, users will be able to select several options.

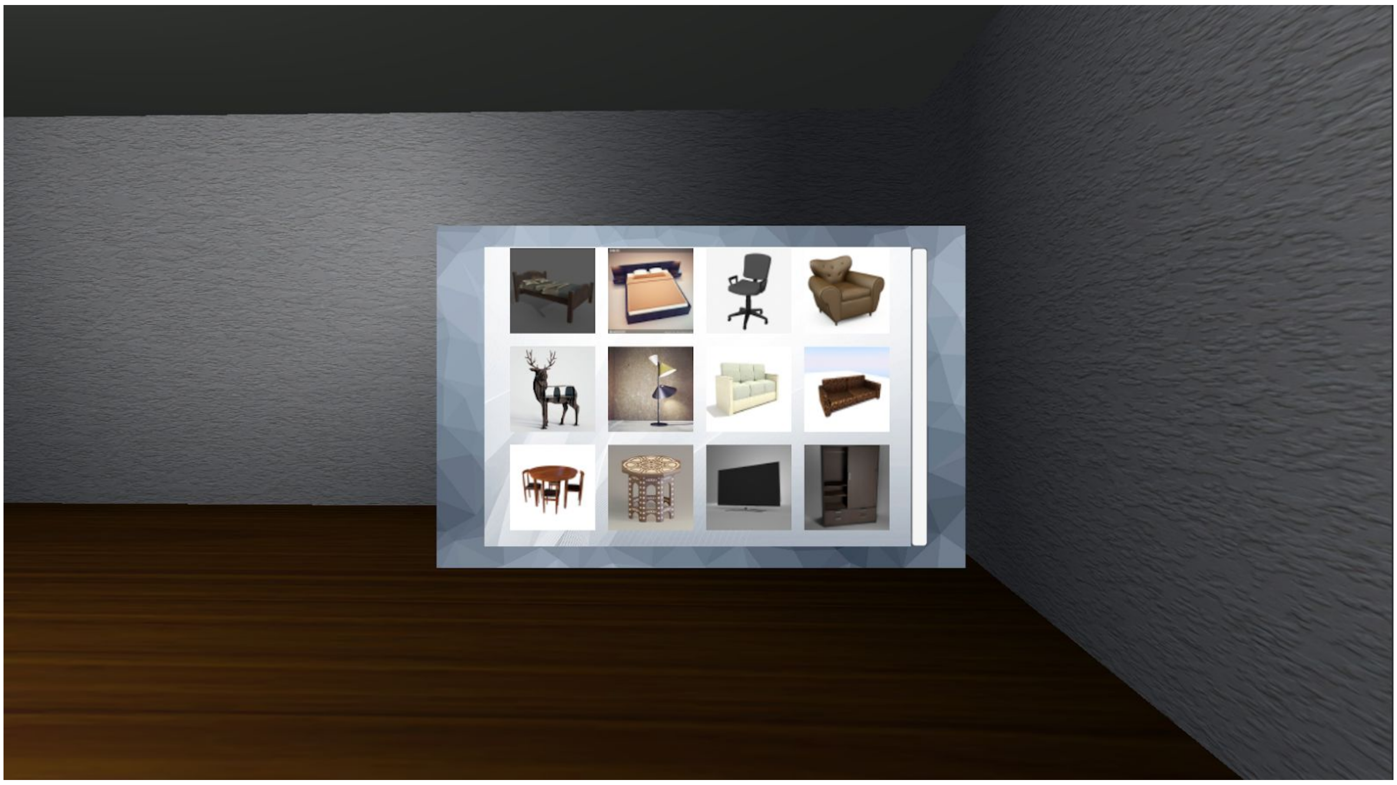


Figure 21: Inventory

- The Q button on the keyboard will release the object at that location. If the object cannot be released, the colour of the object will turn into red and the user will move the object to a different location.

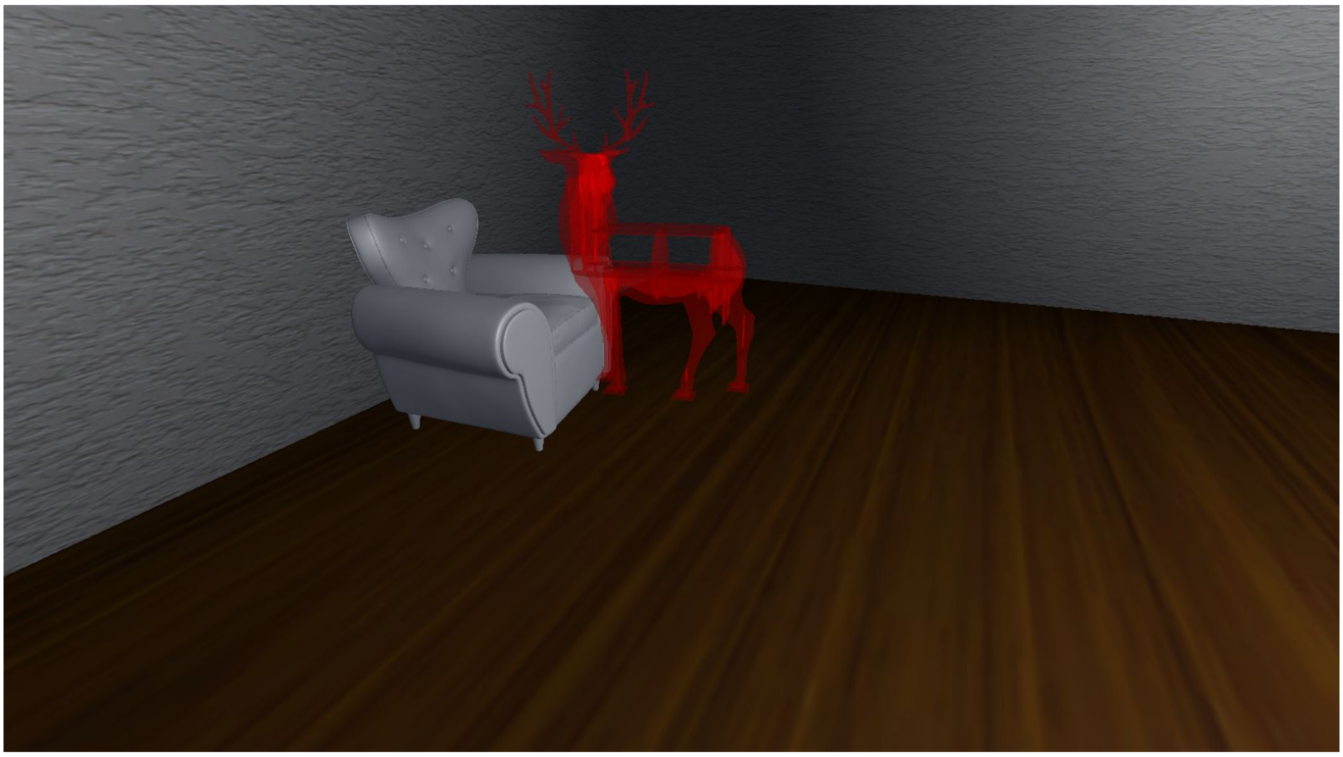


Figure 22: Colliding Objects

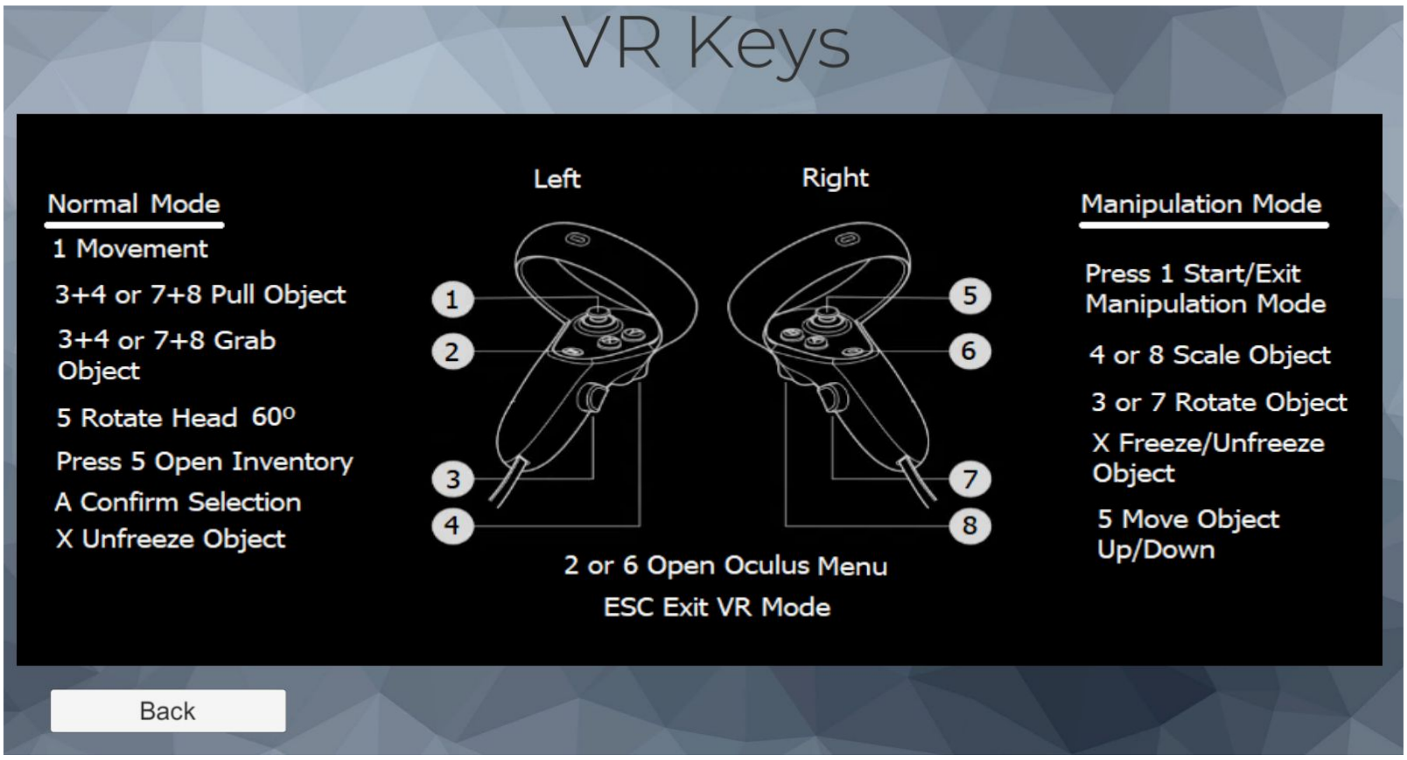
- The E button on the keyboard will grab an object that the user directly looks at. The user will not be able to grab the object if the distance between them is too close. In addition, the user will not be able to grab more than one object.
- The F button on the keyboard will freeze the object at that location. The difference between release and the freeze is that if the user releases the object while in the air, the gravity will be applied and the object will fall down vertically. If the object is frozen, the object will remain in the same position wherever they are.
- The R button on the keyboard will remove the object while in hand. If the object that needs to be removed is placed, the user should first grab the object and then remove it. The user may place the object on a different location by again opening the inventory and selecting it.

- The T Button on the keyboard will make the object bigger and the Y button on the keyboard will make the object smaller. Since the objects are scaled at the start, the first T and Y button will reset the size of the object so users need to press T and Y buttons until they find the desired size.

9.3.3. VR

9.3.3.1. VR Key Binding Page

The key bindings for the PC application can be seen below which will guide the users.



VR Keys

Normal Mode
1 Movement
3+4 or 7+8 Pull Object
3+4 or 7+8 Grab Object
5 Rotate Head 60°
Press 5 Open Inventory
A Confirm Selection
X Unfreeze Object

Left
Right

2 or 6 Open Oculus Menu
ESC Exit VR Mode

Manipulation Mode
Press 1 Start/Exit Manipulation Mode
4 or 8 Scale Object
3 or 7 Rotate Object
X Freeze/Unfreeze Object
5 Move Object Up/Down

Figure 23: VR Key Binding Page

9.3.3.2. VR Play Page

The UI pages for the VR are almost the same as the PC environment since the login, home, workshop, scenes pages are the same. The distinction comes after the 9.2.5 of PC UI, platform selection page. If the user chooses to use the MoveIt on the VR environment, then the controls will be different than the PC environment. However, most functionalities of the features will remain the same. If there are any differences, it will be mentioned below. Additionally, these controls mentioned below are implemented on Oculus Rift, so different VR brands may offer different keys for the inputs. However, even if the names of the buttons are different, the places generally remain the same, so we will add a photo of our controllers to indicate the places of the buttons. The VR application and the general perspective of the application can be seen below.

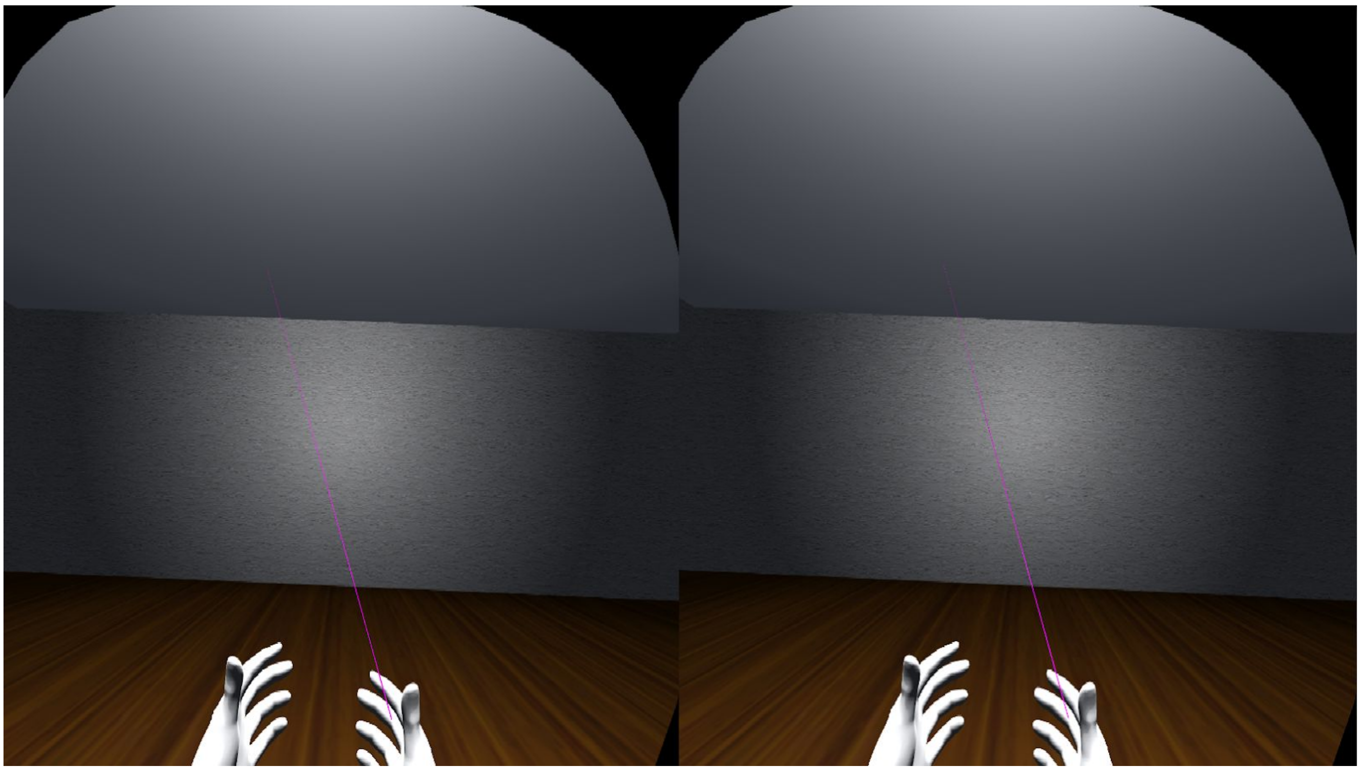


Figure 24: General Appearance of VR

- Left thumbstick (1) will move the user and the right thumbstick (5) will rotate the user camera if pulled to the left or right.
- If the user presses the Right thumbstick (5) button on the right controller, inventory will open or close if it is already open.

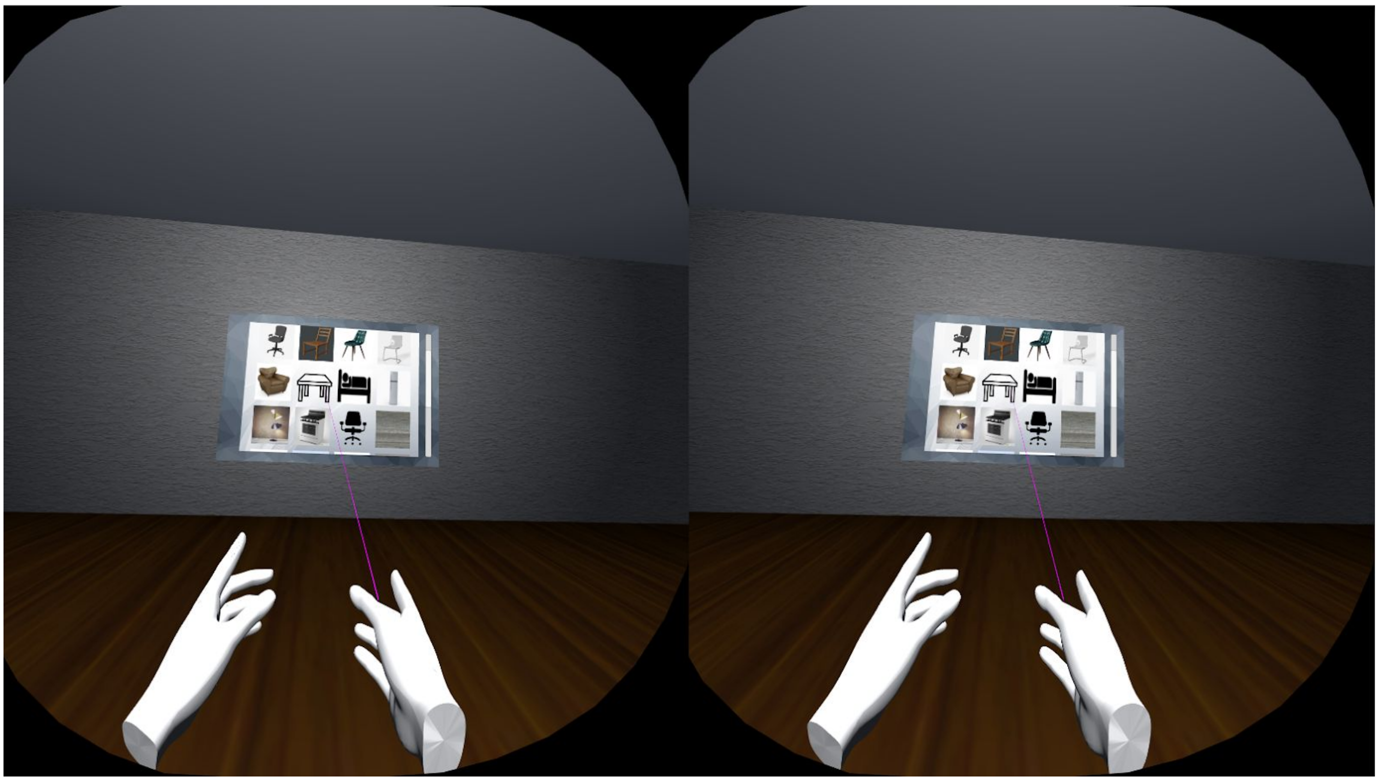


Figure 25: Inventory

- A button in the Left Joystick will confirm the object selection from the inventory.
- Pressing Left Hand Trigger(3) with Left Index Trigger(4) will pull the object to the user if it is far enough and the laser pointer is on the said object. If the object is close enough, pressing these two buttons will grab the object. Releasing these buttons will release the object. The same applies for Right Hand Trigger(7) with Right Index Trigger(8).

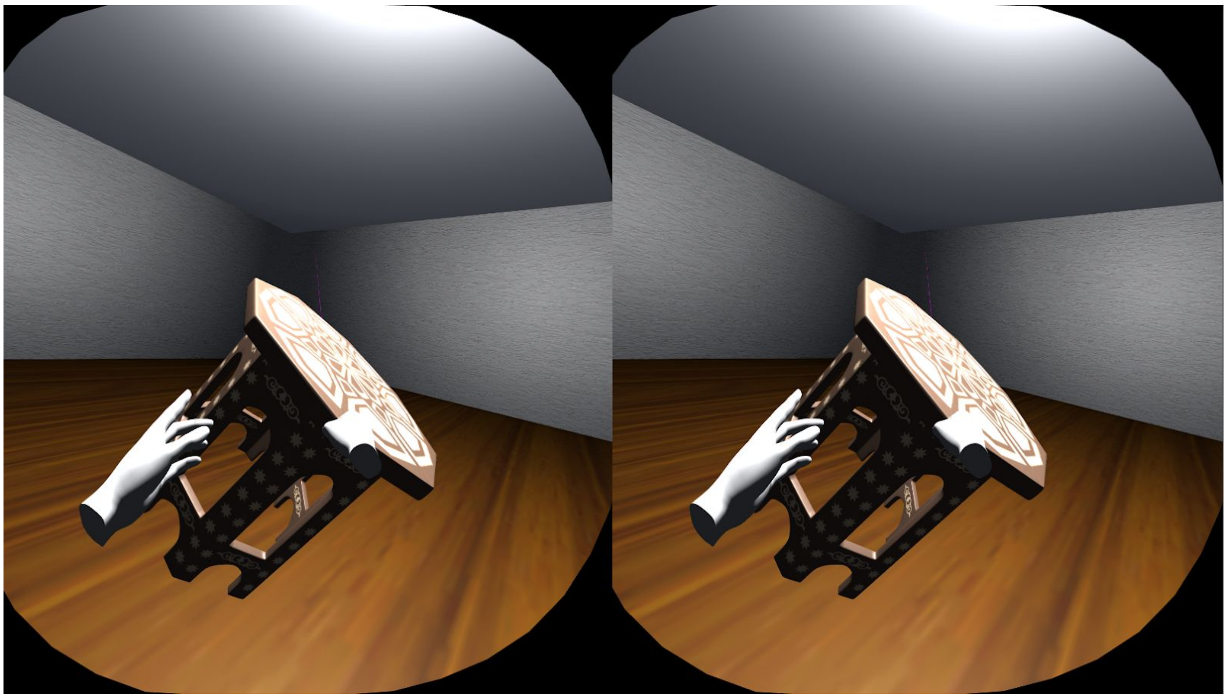


Figure 26: Grabbing Objects

- Pressing Left Index Trigger(4) or Right Index Trigger(8) will change the laser's position to the hand that pressed the button.
- Pressing the Right Thumbstick (1) button while the laser is pointing the object will start the manipulation mode for that object. Pressing it again will release that object from manipulation mode.

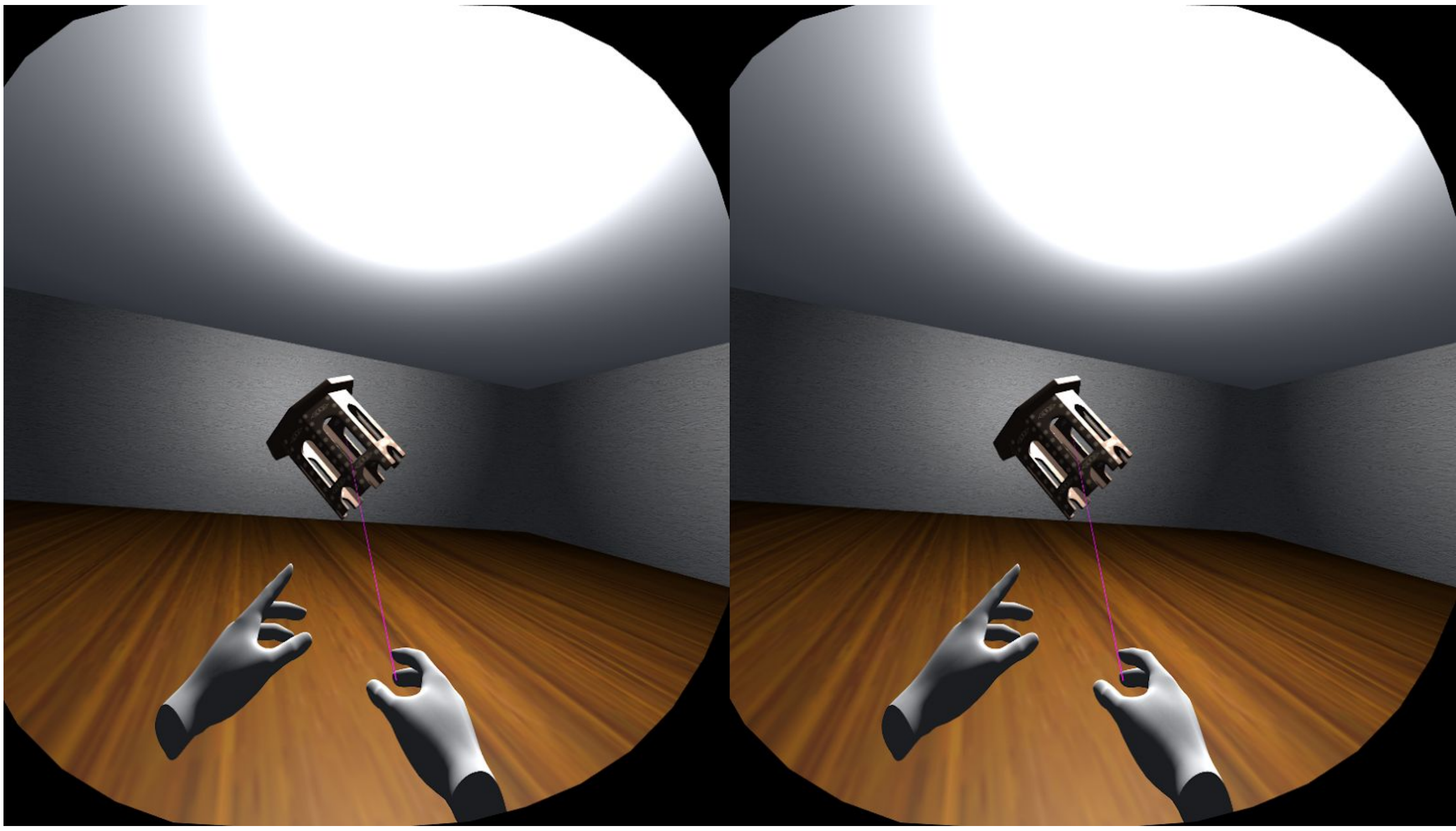


Figure 27: Manipulation Mode

- The Left Hand Trigger(3) and Right Hand Trigger(7) will rotate the selected object 30 degrees in x or y dimension.

- X button in the Right Joystick will freeze the object in manipulation mode or unfreeze the object if the laser is pointing to that object.
- Right thumbstick (5) can be used to manipulate the object's height in manipulation mode.
- Pressing Left Index Trigger(4) or Right Index Trigger(8) will increase/decrease the object's size in manipulation mode.
- Pressing the ESC button from the keyboard will exit the VR Mode
- Pressing buttons 2/6 will open the Oculus Menu of the Oculus Rift S's own software.

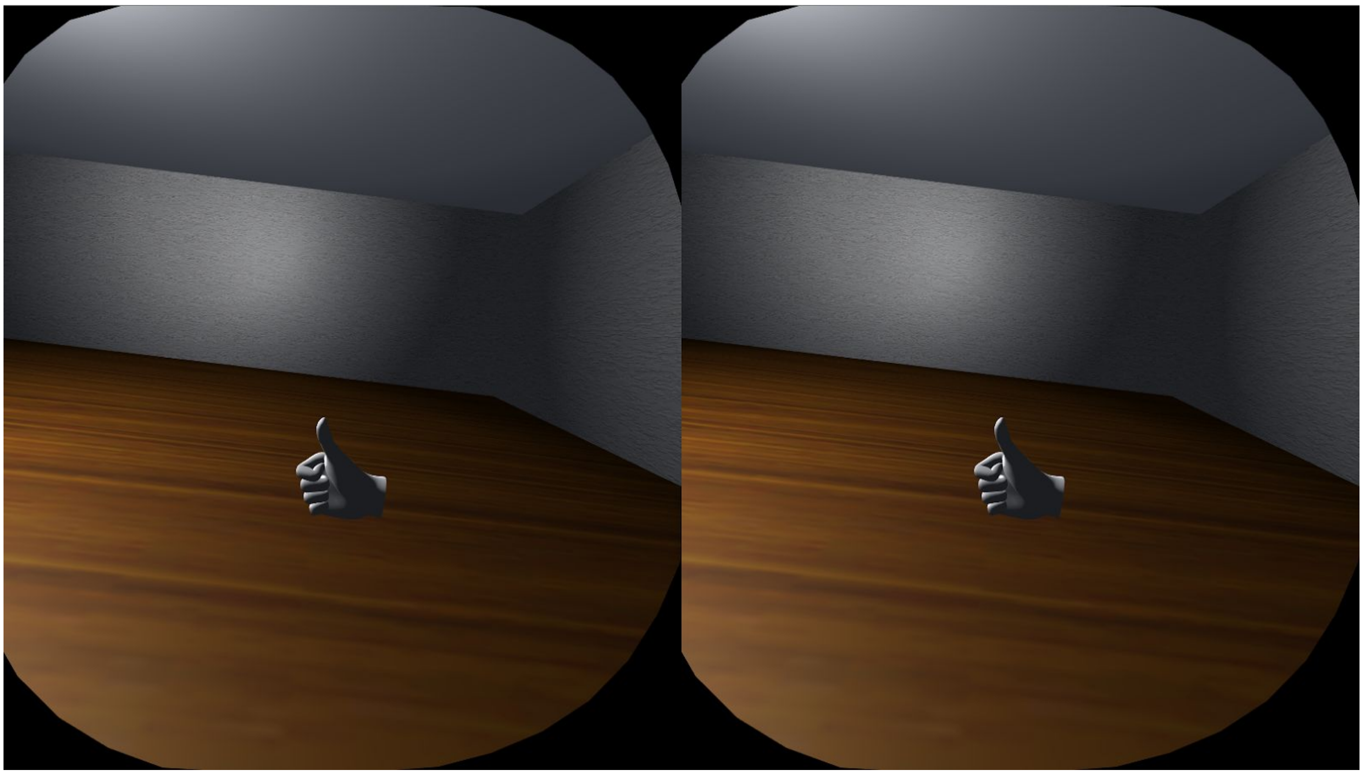


Figure 28: Thumbs Up

References

[1] “ADE20K,” *ADE20K dataset*. [Online]. Available: https://groups.csail.mit.edu/vision/datasets/ADE20K/. [Accessed: 27-May-2020].

[2] Lifetimes of cryptographic hash functions. [Online]. Available: https://valerieaurora.org/hash.html. [Accessed: 09-Nov-2019].

[3] R. Eisele, “SLA Uptime Calculator: How much downtime is 99.9%,” *SLA Uptime Calculator: How much downtime is 99.9% • Open Source is Everything*. [Online]. Available: https://www.xarg.org/tools/sla-uptime-calculator/. [Accessed: 02-Nov-2019].

[4] U. Technologies, “Unity,” Unity. [Online]. Available: https://unity.com/. [Accessed: 27-May-2020].

[5] “Download Android Studio and SDK tools : Android Developers,” Android Developers. [Online]. Available: https://developer.android.com/studio. [Accessed: 27-May-2020].

[6] “MeshLab,” *MeshLab*. [Online]. Available: http://www.meshlab.net/#features. [Accessed: 27-May-2020].

[7] Ü. Öztürk, “Ünsal Öztürk Github,” github.com/uensalo/SyncCamera.

[8] CSAILVision, “CSAILVision/semantic-segmentation-pytorch,” *GitHub*, 30-Apr-2020. [Online]. Available: https://github.com/CSAILVision/semantic-segmentation-pytorch. [Accessed: 27-May-2020].

[9] Ü. Öztürk, “Ünsal Öztürk Github,” github.com/uensalo/3d-reconstruction.

[10] CSAILVision, “CSAILVision/semantic-segmentation-pytorch,” GitHub, 30-Apr-2020. [Online]. Available: https://github.com/CSAILVision/semantic-segmentation-pytorch. [Accessed: 27-May-2020].

[11] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An efficient alternative to SIFT or SURF,” 2011 International Conference on Computer Vision, 2011.

[12] M. Kazhdan and H. Hoppe, “Screened poisson surface reconstruction,” *ACM Transactions on Graphics*, vol. 32, no. 3, pp. 1–13, 2013.

[13] U. Technologies, “Standard Particle Shaders,” *Unity*. [Online]. Available: https://docs.unity3d.com/Manual/shader-StandardParticleShaders.html. [Accessed: 27-May-2020].

[14] “Runtime OBJ Importer: Modeling: Unity Asset Store,” Modeling | Unity Asset Store. [Online]. Available: https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547. [Accessed: 27-May-2020].

[15] “Oculus Integration: Integration: Unity Asset Store,” Integration | Unity Asset Store. [Online]. Available: https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022. [Accessed: 27-May-2020].

[16]N. Statt, “How game designers find ways around VR motion sickness,” The Verge, 13-Oct-2016. [Online]. Available: https://www.theverge.com/2016/10/13/13261342/virtual-reality-oculus-rift-touch-lone-echo-robo-recall. [Accessed: 27-May-2020].

[17]E. M. Kolasinski, “Simulator Sickness in Virtual Environments.,” apps.dtic.mil. [Online]. Available:

https://apps.dtic.mil/docs/citations/ADA295861. [Accessed: 27-May-2020].

[18]J. J. Laviola, “A discussion of cybersickness in virtual environments,” ACM SIGCHI Bulletin, vol. 32, no. 1, pp. 47–56, Jan. 2000.

[19] “Technical Artist Bootcamp: Nasum Virtualis: A Simple Technique for Reducing Simulator Sickness in Head Mounted VR,” GDC Vault. [Online]. Available: https://www.gdcvault.com/play/1022287/Technical-Artist-Bootcamp-Nasum-Virtualis. [Accessed: 27-May-2020].

[20]L. J. Smart, T. A. Stoffregen, and B. G. Bardy, “Visually Induced Motion Sickness Predicted by Postural Instability,” Human Factors: The Journal of the Human Factors and Ergonomics Society, vol. 44, no. 3, pp. 451–465, 2002.

[21] Ü. Öztürk, “Ünsal Öztürk Github,” github.com/uensalo/3d-reconstruction/matlab.

[22] “MeshLab,” MeshLab. [Online]. Available: http://www.meshlab.net/. [Accessed: 27-May-2020].

[23] “Code of Ethics,” *Code of Ethics | National Society of Professional Engineers*. [Online]. Available: https://www.nspe.org/resources/ethics/code-ethics. [Accessed: 10-Oct-2019].

[24] R. Maier, R. Schaller, and D. Cremers, “Efficient Online Surface Correction for Real-time Large-Scale 3D Reconstruction,” *Proceedings of the British Machine Vision Conference 2017*, 2017.

[25] MrPMeshLabTutorials, “Mister P. MeshLab Tutorials,” *YouTube*. [Online]. Available: https://www.youtube.com/user/MrPMeshLabTutorials. [Accessed: 27-May-2020].

[26] J. Stühmer, S. Gumhold, and D. Cremers, “Real-Time Dense Geometry from a Handheld Camera,” *Lecture Notes in Computer Science Pattern Recognition*, pp. 11–20, 2010.

[27] “Add Firebase to your Unity project,” Google. [Online]. Available: https://firebase.google.com/docs/unity/setup#register-app. [Accessed: 27-Apr-2020].

[28] “Download Files with Cloud Storage for Unity | Firebase,” Google. [Online]. Available: https://firebase.google.com/docs/storage/unity/download-files. [Accessed: 27-Apr-2020].

[29] B. Zhou, H. Zhao, X. Puig, T. Xiao, S. Fidler, A. Barriuso, and A. Torralba, “Semantic Understanding of Scenes Through the ADE20K Dataset,” *International Journal of Computer Vision*, vol. 127, no. 3, pp. 302–321, Jul. 2018.

[30] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, “Scene Parsing through ADE20K Dataset,” *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[31] F. Calakli and G. Taubin, “SSD-C: Smooth Signed Distance Colored Surface Reconstruction,” *Expanding the Frontiers of Visual Analytics and Visualization*, pp. 323–338, 2012.

Website

https://moveitplatform.wixsite.com/website

https://github.com/uensalo/3d-reconstruction

https://github.com/uensalo/SyncCamera