Real-time Surface Reconstruction and Interaction for Mixed Reality

by

Eduardo Muñoz Martinez, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2017

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Eduardo Muñoz Martinez

August 30, 2017

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Eduardo Muñoz Martinez

August 30, 2017

Dedicated to the memory of my mother, Carmen Cecilia Martinez de la Garza, who encouraged me to follow my dreams until the very end.

Acknowledgments

The completion of this work is the result of the help, guidance and support of many people.

Firstly, I would like to thank my supervisor Carol O'Sullivan for creating a propitious environment for discussion and guidance, and also for encouraging me to keep going.

I would also thank all my friends and classmates in IET for their help and friendship which made this experience fun and unforgettable. I want to give a special mention to my friends Sarah Fernandes-Pinto-Fachada and Sean Dillon, for their support, their knowledge but most of all for being there by my side.

Lastly, I want to thank my parents for always motivating me to do better and for always believing in me. Thank you, for giving me the strength to follow my dreams even if that meant being apart from you. But above all, thank you for your unconditional love that made all of this possible.

Eduardo Muñoz Martinez

University of Dublin, Trinity College
September 2017

Real-time Surface Reconstruction and Interaction for Mixed Reality

Eduardo Muñoz Martinez
University of Dublin, Trinity College, 2017

Supervisor: Carol O'Sullivan

We proposed a method to reconstruct dynamic surfaces from the real world using a fixed Kinect depth camera. By acquiring a color image video and depth data from the device, the application reconstructs a dynamic point cloud and computes vertex normals to describe the real world scene. Matching the real-time color image with the reconstructed dynamic surfaces enables interaction between real and virtual objects. Considering the virtual objects as rigid bodies, collisions against the dynamic surface are computed by a physics module. Occlusions are generated by manipulating the opacity of virtual object's pixels and comparing depth information between its position and the depth data of the scene. We described techniques to boost real-time performance: pre-processing the background by computing depth segmentation to retrieve bounding boxes for each region and separate collision detection into low-cost phases.

The method and techniques proposed enable the user to interact in a real-time environment with virtual objects, allowing collisions and occlusions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Digital entertainment, especially videogames aims for a more immersive user experience. Different techniques have been implemented in the last couple of years trying to improve this sensation, such as motion controls, stereoscopic cameras, depth cameras, computer vision, among other combination of themselves.

In 2006, Nintendo showed a motion control technology when they released the Nintendo Wii console. The product received generally positive reviews, it brought a new era of gaming control, but still, the player had to hold the motion control at all time in a certain position to be detected by the infrared sensor. In 2010, Microsoft released to the market a depth camera device called Kinect, which tracks the movements of the player's body and use them as signal inputs in Microsoft games without holding any gadget. Kinect provided an augmented virtuality experience, where the movements of the user were imported to the virtual environment of the game.

Augmented reality (AR) in the entertainment field gained a lot of attention in 2016 thanks to Pokemon Go, a smartphone application which uses the built-in camera and computer vision to reproduce characters animation within the video captured. The game was a major success becoming one of the most downloaded smartphones applications of all times, however the game presents some technical issues when recognizing surfaces, which causes the characters to be rendered in wrong surfaces or non-logic sizes.

The same year, Microsoft released the HoloLens, a pair of glasses with the Kinect technology integrated, which is capable to show holograms projected in the lenses to create a Mixed Reality. HoloLens is capable of spatial mapping, gesture recognition and speech

recognition. Although the Augmented Reality provided by the glasses is high technology, there is still development required for the interaction between the player and the virtual objects.

Before AR technology was introduced into the videogames Industry, the immersion of the player was denoted by the physics of the game, how real was the interaction with the character and its environment; the artificial intelligence, how real was the behaviour of the characters and enemies; the consistency in the graphics, the plot of the story, the quality of the gaming controls, among the most important. The intention of this project is to improve the immersion of the virtual objects augmented on real world scenes by taking one of this strategies mentioned before: physics interaction. To achieve our goal we need to construct an environment where virtual and real objects co-exist and interact.

Chapter 2

State-of-the-art

In this chapter we are going to discuss the Augmented and Mixed Reality technology so far, as well as some concepts and methods we are going to use in our application development.

Before going further into the technology advancements in this field it is important that we define what Augmented and Mixed Reality are. For this task we are going to use the taxonomy provided by Paul Milgram [1], where he defines a virtuality continuum that goes from a real world environment to a virtual world environment, and where we can find the augmented reality and augmented virtuality in between, as shown in Figure 2.1. Augmented reality is defined by a real environment where virtual objects are augmented on it; on the other hand Augmented virtuality is a virtual environment where real objects are augmented on it. The virtual environment or virtual reality, is an environment where the observer is immersed in a completely synthetic world, either existing or fictional. Mixed reality can be anywhere between the virtuality continuum, in other words is the technology merging the real and the virtual worlds to produce an environment where real and virtual objects coexist and interact.

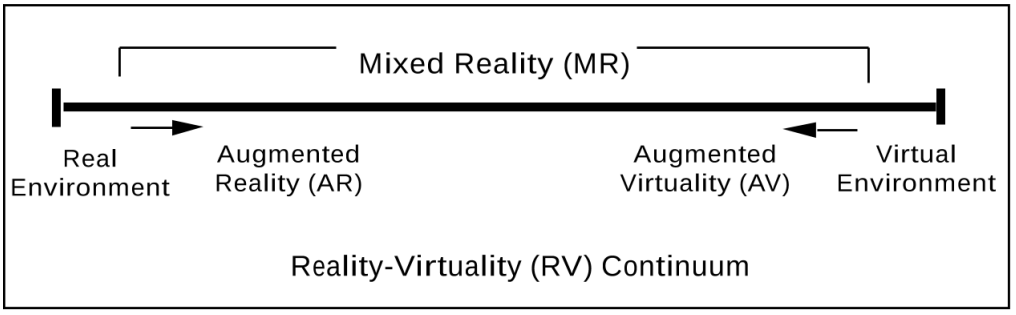


FIGURE 2.1: Virtuality Continuum (VC) going from Real Environment, through Augmented Reality (VR), Augmented Virtuality (AV), to Virtual Environment (VR). Mixed Reality (MR) is the area between the two extremes.

For a long time there has been an interest to merge the virtual and the real world for a variety of applications, not only in the entertainment area. In 2001, R. Azuma et al. provide an article [2] talking about the technological advancements on Augmented reality at that time. From medical applications with the purpose to "tracking and displaying techniques to support laparoscopy surgery" [2], to engineering applications where augmented 3D models of pipelines are created to match factory floor plans [3], R. Azuma presents a vastly quantity of examples where merging both worlds has enabled the development of technology in several areas.

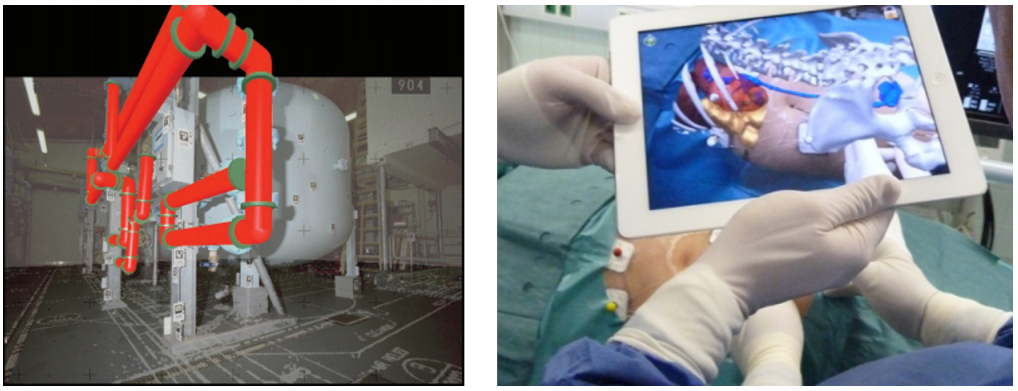


FIGURE 2.2: **Left:** 3D pipeline model augmented in top of industrial planes. **Right:** Medical AR application.

In 1996, David E. Breen et al. [4] presented two methods to make augmented objects interact via collisions and occlusions with real objects in augmented reality: the Model-based and the Depth-based method. Their hardware system consist in a video camera recording the real world environment, a six degrees-of-freedom (6DOF) tracker which provides the position and orientation of the video camera, a workstation that produces the graphical image and a high resolution monitor for display.

The model-based method consists in superposing virtual 3D model on their counterpart real world objects, in other words, if there is a table in the image captured by the video camera, the User must look in a catalog of 3D models, the model of a table similar in geometry to the real one and then by clicking on a number of key points in the image, the virtual table would be aligned or register to the real world image and its perspective transformation will be computed towards the camera view in that moment in time, so if the camera moves, the workstation would receive the new position and orientation of the camera and the virtual table would be transformed to match the position of the real one.

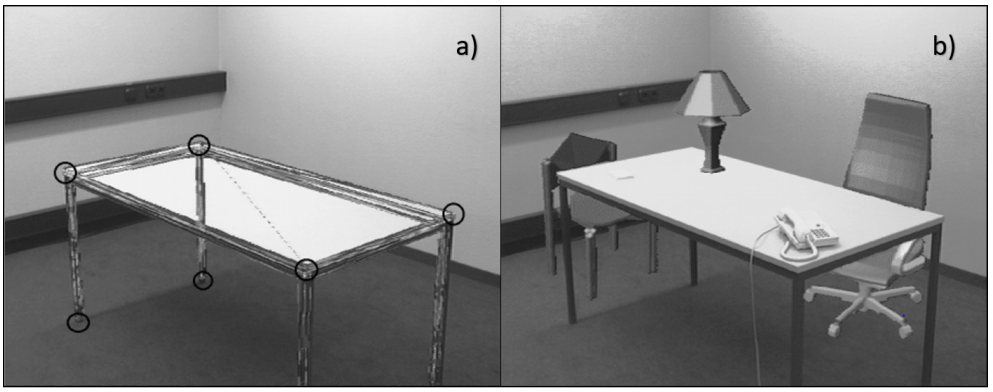


FIGURE 2.3: **a)** Virtual table being placed by the User selecting the corners of the real table on the image. **b)** Augmented chairs being occluded by the table.

After the User completes the alignment between the 3D models and the real objects, the augmented objects are able to collide and be occluded by "the real world objects". By this method, the occlusion and collision happen between virtual objects in a real world environment after a pre-processing stage, but due to registration of models, the image displayed would give the illusion that augmented objects are interacting with real objects.

The depth-based method consists in using depth maps of the real world scene to be compared with the augmented object position. If the pixel from a virtual object has a position further than the z value of the depth map, the pixel won't be drawn. In terms of collision, a virtual camera must be registered to work as the real camera in the virtual framework to reproduce a view-dependant depth map of the real world scene. If one virtual object position get close enough to the depth map view from the virtual camera a collision is made. Then the virtual object just stop moving and it can be manipulated again. The method only checks a bounding box around the virtual object. During the simulations, they made an implementation with computed gravity to test the collisions of virtual object falling on a real table as shown in Figure 2.4, where the objects moves from an arbitrary gravity vector computed, checking in every frame the bounding box against the depth map until a collision is detected. If a collision is detected, a torque calculation is made to perform a rotation around the collision point until the bounding box is aligned with the real table.

FIGURE 2.4: Simulation of augmented objects falling into a real table.

It is notorious that depth-based method has advantages over the model-based one, as the scene can be more complex and no 3D models are needed for the simulation, but at that time generating depth maps in real-time was a complicated task. The depth map used were artificially generated by 3D models of the views of the real world. Figure 2.5 shows a virtual object and the image of its z values to make a depth map of that scene. Having static depth map means that the real camera should be fixed into the same scene and that the scene can not be altered. Both methods got very good results for their time, but none of them were capable to introduce a moving object or a person to interact with the virtual objects into the scene.

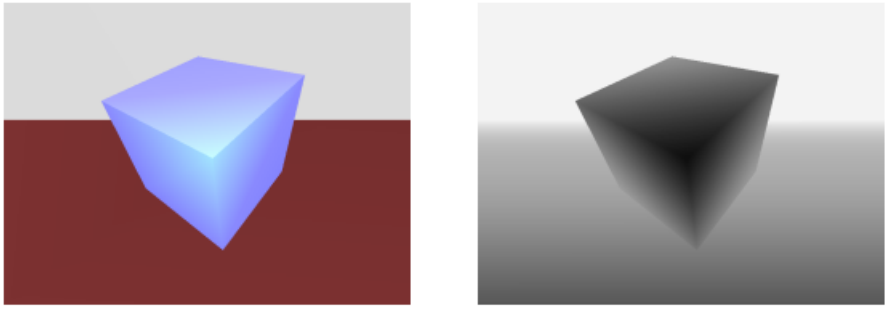


FIGURE 2.5: **Left:** 3D model of a cube. **Right:** Depth map generated from 3D model.

Depth maps were considered to produce augmented reality applications even back then when generating a depth map from a device was not that common as nowadays. With the release of the Kinect from Microsoft in 2010, depth cameras became more accessible to all and new possibilities arrived. Shahram Izadi et al. presented a method of surface reconstruction and interaction in real-time using a moving depth camera, the project was called KinectFusion [5]. The method reconstructs a surface of a real world scene by creating a mesh from the depth map. A single depth map will reconstruct a mesh of the front face of the real object from the camera perspective at that moment in time. The method proposed to move the camera around the real object or a scene to capture the depth map from different perspective views and merged them by applying a 6DOF computational tracker [6]. By moving the camera around the scene in a 360° loop, the

application gets all the faces from the scene and a dense surface mesh is generated as shown in Figure 2.6.

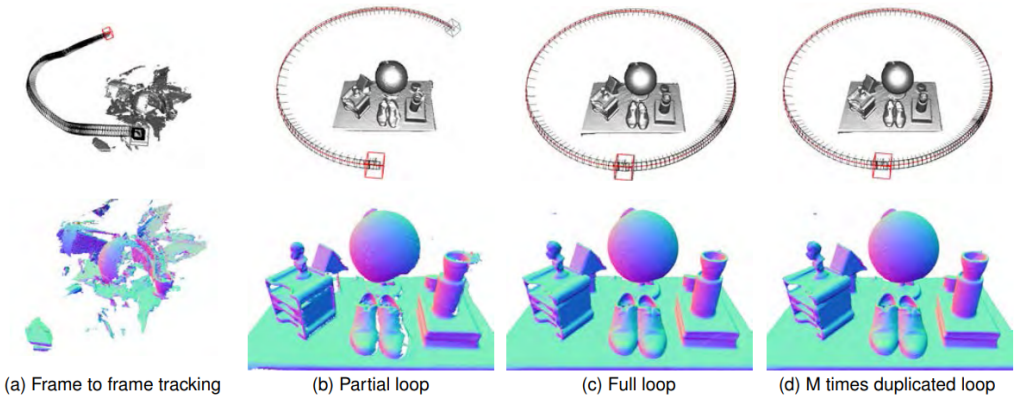


FIGURE 2.6: Moving depth camera looping around a scene to generate a surface mesh.

Figure 2.6(a) shows a poor mesh surface in contrast to Figure 2.6(c) where the loop around the scene has been completed. After the scene is captured, a texture over the surface mesh is applied with the colors picked by the RGB camera of the Kinect. Rendering all together we will have a 3D model of the real scene get from the color and depth camera of the Kinect. In some way this method is very similar to the model-based one from D. Breen but instead of inserting the 3D models into the scene, the 3D model scene is generated by depth maps. Before the scene is rendered, normals are computed for every point of the mesh. Once the scene is completed, virtual objects can be imported into the scene to interact with the environment. Again, we have augmented objects interacting with virtual objects based on real objects. Figure 2.7 shows a 3D reconstructed scene interacting with augmented particles in real-time.

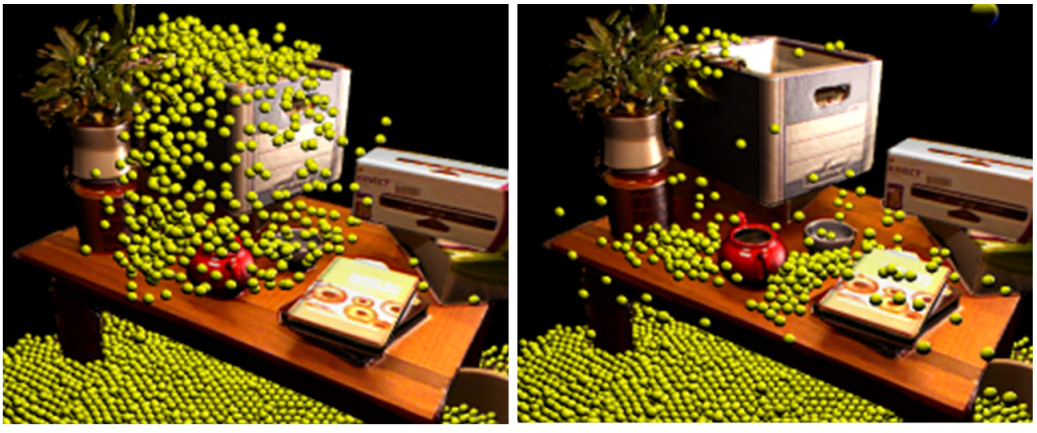


FIGURE 2.7: Augmented particles colliding and being occluded by the surfaces reconstructed of a real world scene using KinectFusion.

This method assumes static scenes to reconstruct while tracking. When there are moving objects into the scene, the model would try to reconstruct the surface with different perspectives from every face resulting in a wrong reconstruction of the real world as shown in Figure 2.8.

FIGURE 2.8: A person moves freely into the scene and the surface are partially generated due to motion.

The solution implemented by the researchers was to lock the tracking when a User is interacting into the scene, separating the User as a foreground from the static scene and only computing the his front face in the real-time interactions. While this solution avoids creating wrong surfaces, the surface reconstructed would depend of a single depth map as Figure 2.7(a) and thus it might contain holes occasionally, losing the immersion experience from time to time.

There is no doubt that depth cameras opened many new opportunities for mixed reality applications to improve the methods proposed in the past or to implement them for the first time, when the methods were only conceptual ideas due to the lack of technology back then.

Chapter 3

Design

The overall goal of the project is to produce a real-time application where the User can interact with the virtual objects in Mixed Reality. To achieve our objective we must follow a complex pipeline and for that same reason every step must be approached computationally inexpensive in order to preserve the real-time experience.

For the implementation of the project a Kinect 2.0 from Microsoft will be used. The hardware is capable to capture depth into a scene of 512 x 424, at 30Hz, FOV: 70 x 60, between 0.5 and 4.5 meters; it is also capable of capturing 1080p color video at 30Hz. Libfreenect2 library will be used to get the raw data from the Kinect 2.0 [7].

FIGURE 3.1: Kinect 2.0 camera from Microsoft.

A general flow diagram of the application architecture is shown in Figure 3.2. The program is going to get the RGB and depth image from the Kinect for every frame. Using the depth information, a cloud point would be generated to reconstruct the surfaces of the real world scene. Virtual objects would be imported to the scene and their dynamics will be computed by a physics module to then be rendered on the screen. Virtual objects are going to interact with the reconstructed surfaces from the real world in real-time.

Unlike the methods of D. Breen et al. [4] and S. Izadi et al. [5] which were focused on a model-based approach, where [5] renders the real world texture in top of the reconstructed mesh models, we are going to work with the real world color image provided by the Kinect and matching it with the surface constructed. The interactions are going to be computed according to the surfaces generated.

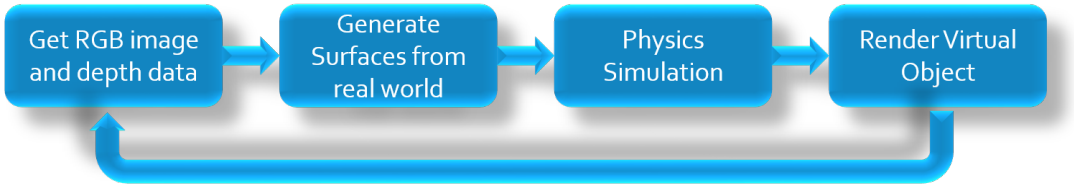


FIGURE 3.2: General process flow design.

The method proposed can be followed by using any device that provides a RGB and depth image of the same scene, even if it is not a Kinect 2.0 camera. OpenGL library is going to be used to render the scene and the virtual objects; while OpenCV library is going to be used for computer vision purposes.

3.1 Calibration

Before we continue to the implementation of the application, we are going to review an important part of the process when using cameras: Calibration.

Geometric camera calibration estimates the parameters (intrinsic, extrinsic and distortion coefficients) of the lens and image sensor of a camera. These parameters help to restore an image from lens distortion or to determine the location of the camera in the scene. From the intrinsic parameters we are interested in the Focal length, which is the distance between the center of the lens and the image plane; and the Principal point, which is the perspective center projected into the image plane. The intrinsic parameters are going to work to transform the pixels to world coordinates later on.

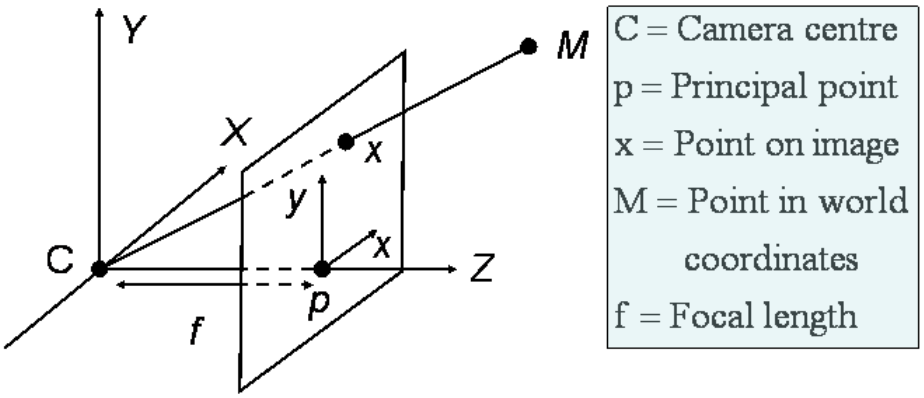


FIGURE 3.3: Camera model showing the focal length and the principal point.

When having more than one camera and a relationship between them is required, the extrinsic parameters should be computed. In our case, we are going to get two images: the RGB and the depth image, from two different points of view, even when they are in the same device. It would be like having two different camera very close to each other capturing the world. The images of the two cameras are going to be slightly different because their point of view are not the same. To look for one point captured in both images, we would need one of the images to be transformed into the other camera's location to obtain its point of view. For this transformation we are going to need the extrinsic parameters: translation T and rotation R. The extrinsic parameters are going to work to match the RGB and the depth image later on.

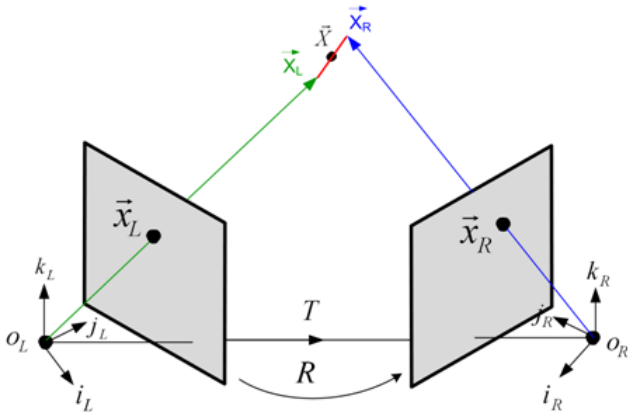


FIGURE 3.4: Stereo camera calibration showing the displacement T and rotation R from the Left camera to the Right.

If the intrinsic parameters of the camera are not available from the manufacturer, Lomonosov Moscow State University released a calibration toolbox [8] that uses the

popular chessboard pattern to compute the intrinsic parameters of the camera. For the depth camera, the process is different as the detection of the pattern is not visible in the depth image. In this case, we use the corners of the chessboard pattern that can be detected due to depth difference and get the four points for the intrinsic calibration as shown in Figure 3.5.

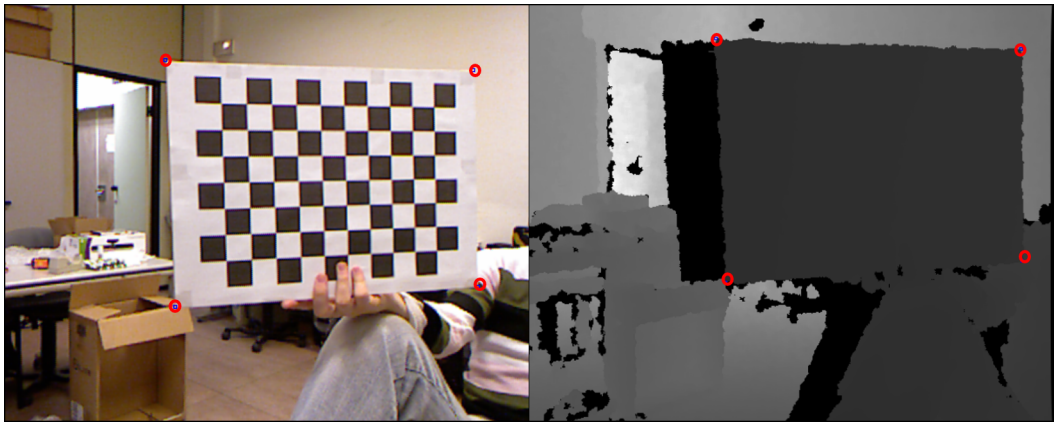


FIGURE 3.5: Stereo calibration performed between a color camera (left) and depth camera (right).

After determining the intrinsic parameters of the camera and having the four corners of the chessboard pattern for n different images, the extrinsic parameters T and R can be obtained by following the procedure proposed by Cha Zhang [9].

Chapter 4

Implementation

This chapter will described thoroughly the implementation of the application following the general stages presented in the design of the previous section.

The implantation is going to explain step by step the method proposed which the reader can follow with any device that can retrieve a color image and a depth image of the same scene in real-time. For simplicity purpose, the implementation is going to be described with the libraries used in this project: OpenGL and OpenCV. If the reader is using different libraries, the function descriptions can be found in the Appendix section.

4.1 Capturing the real world

First, we must capture the real world and make a virtual scene of it so we can bring virtual objects into. From the Kinect 2.0 depth sensor we can get a depth image, and using the open source library libfreenect2 [7], we can generate a point cloud from the depth image using the function *getPointXYZ*. The point cloud retrieved is going to have world unit coordinates in meters.

If the open source library is not used, the point cloud could be generated from the depth image employing the same equations as libfreenect2 [7]: using the intrinsic parameters from the depth camera to undistort the image to real world perspective and the depth value z from the depth image for every pixel n. The equation to transform from a pixel p_i to a 3D point in space P_i is given by:

$$
\begin{aligned}
P_i &= P_i\hat{x} + P_i\hat{y} + P_i\hat{z} \\
P_i\hat{x} &= (u - cx) * z/fx \\
P_i\hat{y} &= (v - cy) * z/fy \\
P_i\hat{z} &= z
\end{aligned}
\tag{4.1}
$$

Where cx and cy are the Principal points of the depth camera and fx and fy the focal length obtained previously through calibration. Pixel $p_i = [u, v, z]$, where (u, v) are the pixel coordinates from the depth image and z the depth value retrieved from the Kinect 2.0 depth sensor. The depth value given from the Kinect is in millimeters so the result should be divided by 1000.

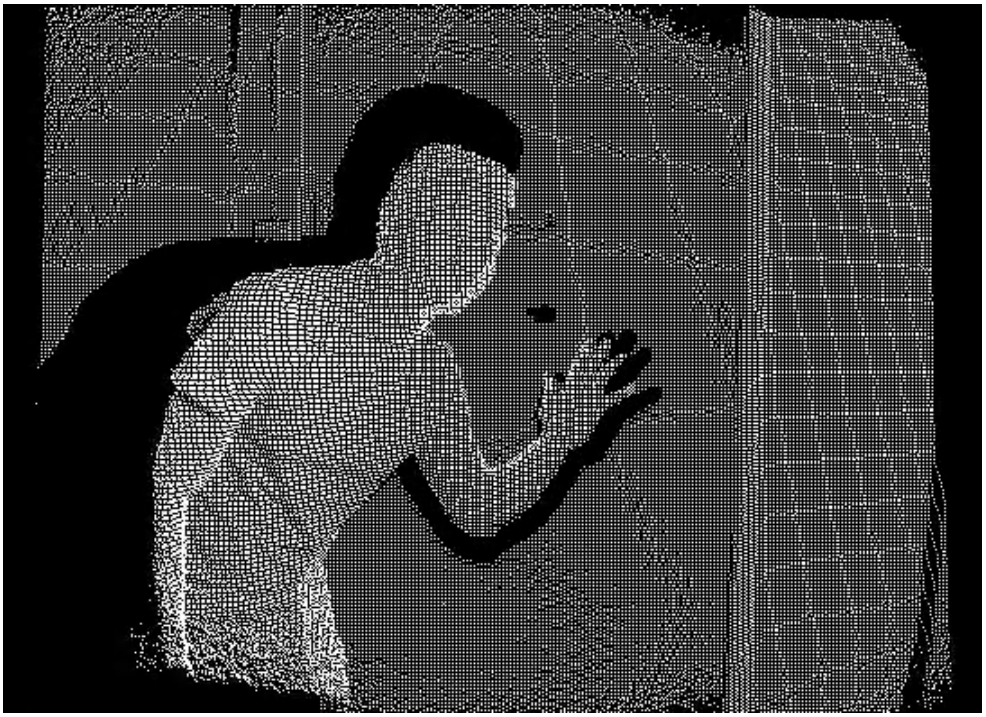


FIGURE 4.1: Point cloud visualization of an image captured with the Kinect 2.0.

4.1.1 Mesh generation

The point cloud, the depth image and the color video image are going to be our main inputs to understand the real world and make a surface reconstruction of it. The depth image retrieved by the Kinect 2.0 has a dimension of 512 x 424, which means that if we compute every pixel for the point cloud we would get 217, 088 3D coordinate points per frame. For our purpose project we want a general idea of the shape of the real world, then we can discard some details in the surface reconstruction by reducing the dimension of the point cloud. At the same time we are going to save memory, computations and

time. In this project we are going to reduce the dimension by a scaling factor $SF = 8$, so instead of taking every pixel, we are going to take every 8 pixel reducing the overall point cloud to 64 x 53, which means that for every frame we would get 3, 392 3D coordinate points.

The point cloud would give us the information of depth of the color image. For this project we are going to assume the Kinect camera is fixed, that means our background e.g. walls, floor, etc. would remain the same over time. Obviously as real-time applications are used to, there could be interaction with the background, for example, a chair being move or a door being open. We are going to store the point cloud as a vector of 3 float coordinates. The size of the vector would be the dimension of the depth map divided by the scaling factor grid. The width value w, and the height h must be updated by the scaling factor selected.

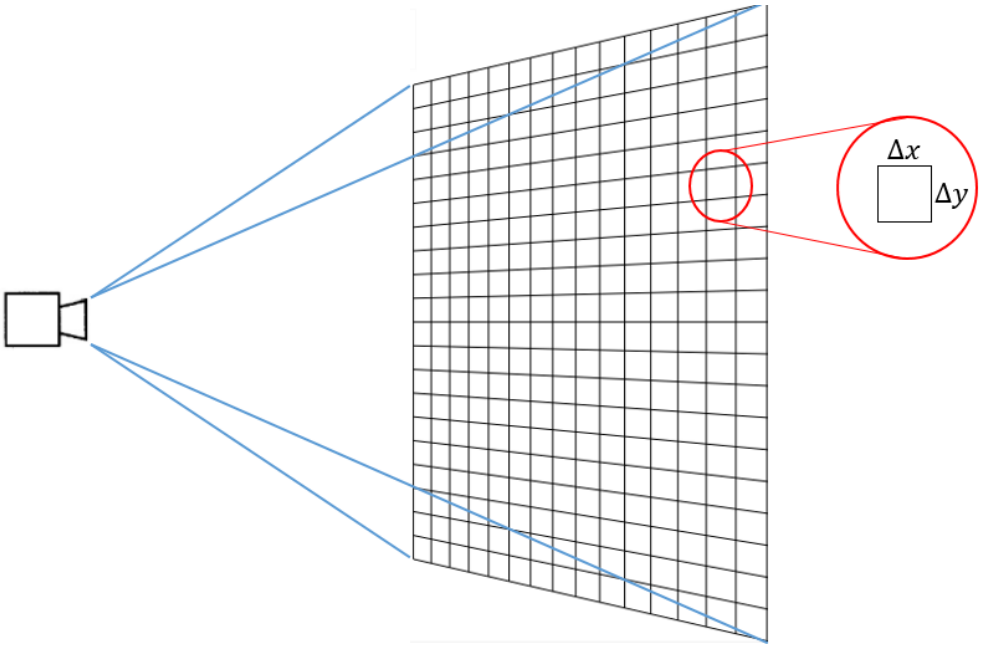


FIGURE 4.2: Grid describing a flat wall in front of a depth camera.

The point cloud describes the physical boundaries of our scene. Joining the dots would give us a mesh detailing the relief of the real world captured, as if we throw a bed sheet on top of the whole scene with a grid drawn on it. As we are using only one camera, the point cloud retrieved is generated by a single point of view: the depth camera location. For this reason the point cloud would describe only one face of the scene and the point cloud could be explained as a grid facing the camera. If the camera is in front of a flat wall, facing it directly, the 3D coordinate points of the grid will have the same depth value in theory, as if we were looking at points in a 2D plane as Figure 4.2. The horizontal and vertical distance of a sample point with its neighbours would be equidistant between

them: Δx and Δy. This feature will help us later when we are calculating the normals of the surface.

In a first instance we are going to create a single mesh covering all the scene as what we have pictured before. In order to create the triangle mesh, we are going to establish a criteria for the triangulation. Taking four adjacent points that form a square from the grid: $\{P_n, P_{n+1}, P_{n+w+1}, P_{n+w}\}$ from Figure 4.3, will contain two triangles; the first one is described by: the top left point, the top right point and the bottom left point $T1 = \{P_n, P_{n+1}, P_{n+w}\}$; the second one is described by: the top right point, the bottom right point and the bottom left point $T2 = \{P_{n+1}, P_{n+w+1}, P_{n+w}\}$. The grid will end up with diagonals between the top right point to the bottom left point of a sample square within the grid. As we have already the 3D coordinate points in a vector, we are going to make an array of indices to specify the graphic program the 3D coordinates that will serve as vertex position of each triangle in the mesh. Taking advantage that the mesh will describe a rectangular grid, we are going to make an algorithm that goes over each row from left to right, from top to bottom, where each point will be the top left vertices of a square in the grid and drawing the two triangles illustrated previously. The algorithm should ignore the last column at the right and the last row at the bottom, due to the incapacity of making a square as not having a vertice next right or next below. A graphic illustration of the array of indices is shown in Figure 4.3. The length of the indices is described by *(w-1)(h-1)*6* and the number of triangles is described by *(w-1)(h-1)*2*.

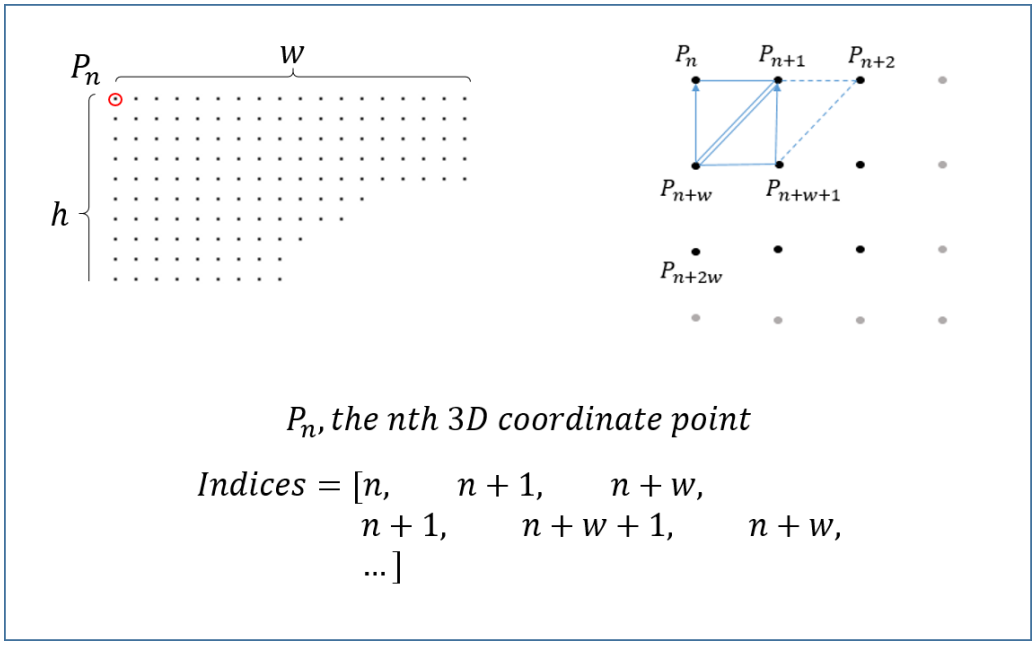


FIGURE 4.3: Sequence illustration of the array of indices to generate a triangle mesh from the point cloud.

The indices instruct the graphical software the location of the point coordinate stored in the point cloud vector to serve as a vertex position to render each triangle and outline the mesh. For this project we are using cross-platform API OpenGL 3.0 and take advantage of the graphics processing unit. The color image would be render as a texture placed in the corner coordinates of the window screen that are always clamp to -1.0 and 1.0 in Height and Width; and then assigning the texture a depth value of $z = 0.0$. Rendering the triangle mesh is a whole different story. We need to transform the 3D coordinate points to a system coordinates that matches OpenGL window screen and the color image displayed. The raw points from the point cloud are in local space delineated by the Manufacturer's device parameters and distance factors, in this case the depth camera of the Kinect. The local space described by the point cloud are in meters, so we need to transform them into screen coordinate points and transform its point of view to match the color image. If the local space units are not provided by the camera's Manufacturer they can be retrieved by doing distance tests with the depth camera and plain objects, to get the relation between real world units and the camera distance units.

Some graphical software expects the coordinate points to be within a range to be rendered. OpenGL has a specific range of -1.0 to 1.0, if one vertex position has a value outside the range it would not be visible and clipped away. We define a projection matrix to transform the vertex position to the clipping space of our graphical software, all coordinates that are within the range are called normalized device coordinates. As we are matching a real world image we need the projection matrix to have a perspective effect. In real life objects that are farther away from our sight appear smaller than those that are closer, the perspective projection matrix tries to replicate this effect. The effect is achieved by dividing all component of the 3D coordinate by a fourth component that OpenGL stored with each point called the perspective division or w value, giving smaller 3D coordinate component value the further the point is from the viewer. OpenGL has a function to compute the perspective projection matrix. The matrix derivation is found also in Appendix A.

While the perspective projection matrix give us graphics as real life, they are restricted by a single point perspective, by default the world center coordinate (0.0, 0.0, 0.0). The view matrix transform the vertex position to a particular point of perspective; i.e. if we want to look the point cloud seen as if we were standing right side the captured scene, then the view location would not be the default anymore, and we could change it with a combination of translation T and rotation R of the scene. For our project purpose we need a point of view that will give us the same view as the color image camera so we can match both images. OpenGL has a function to compute the view matrix according

to the translation T and orientation R obtained by the stereo calibration. The matrix derivation can also be found in Appendix A.

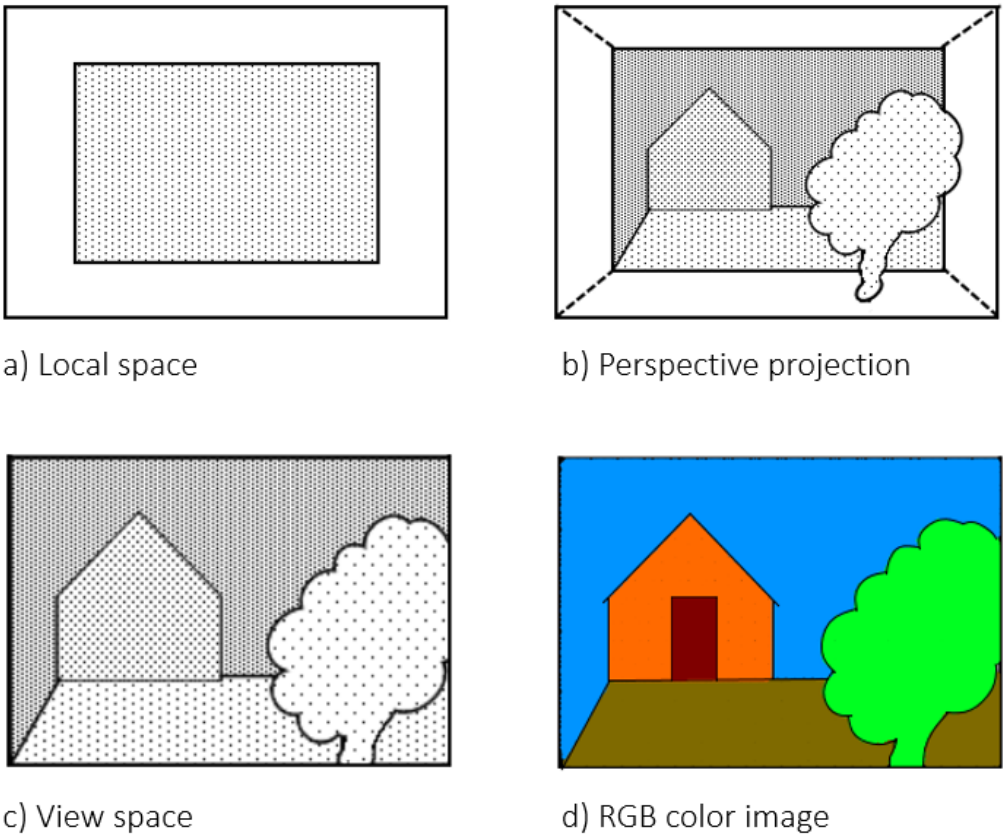


FIGURE 4.4: **a) Local space:** 3D coordinates provided by the device without any transformation. **b) Perspective projection:** Coordinates transformed to screen space applying a perspective effect. **c) View space:** Coordinates transformed to camera point of view to match the RGB camera image. **d) RGB color image:** color image provided by the device.

The matrices transform all our vertex position describing the surface to match the color image received by the Kinect and displayed as a texture in the graphical software. If we render both the color texture, and the now transformed point cloud, we would have our real world scene covered by a triangle mesh describing its reliefs as shown in Figure 4.5.

FIGURE 4.5: Triangle mesh rendered in top of the color image. The triangle mesh match the real world image by the transformation of the **view** and **projection** matrices.

After matching the real world scene with the triangle mesh, we do not need the triangle mesh anymore, and thus neither the computation of indices. Triangle meshes works to create 3D models and adding a texture on top of them. For our project the real world texture is displayed on the corners of the screen and the point cloud is already the surface we need to describe the real world.

4.1.2 Normals computation

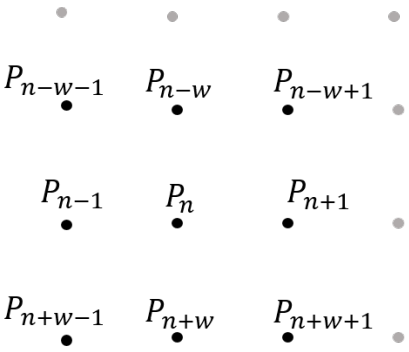
Once we have the surface point of our scene, we can move on to compute the vertex normals of each 3D coordinate. We are going to take a reference from a vertex normal algorithm introduced by Henri Gouraud [10], and evaluated by Shuangshuang Jin et al. [11] referring to it as "The Mean Weighted Equally Algorithm" (MWE) and noted as:

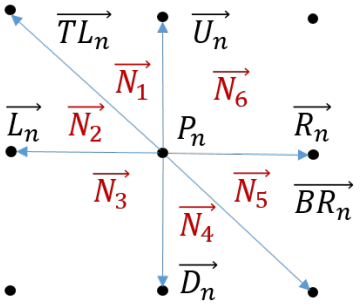
$$N_{MWE} \,||\, \sum_{i=1}^{n} N_i \tag{4.2}$$

Where the normal of a coordinate point is parallel to ("||" , and not equal as the normal must be normalized) the summation of the normals for all n faces incident to

the coordinate point, being N_i the face normal. The algorithm gets its name as each face normal contributes equally to the normal of the coordinate point. Several other algorithms found in literature have a different technique to weigh the contribution of the faces where the coordinate point is present, but as we are working with a grid or a likely organized point cloud we will refer the MWE.

To generate a face normal, let say we have a triangle with 3D coordinates A, B and C, would require to calculate the cross product of the vector $\vec{BA}$ and $\vec{CA}$ to get a perpendicular vector them. The direction of the vector depends on the order of the factors, so in order to get the normal pointing towards the camera we are going to use a clockwise approach. A sample vertex point from the middle of the grid is part of six faces, that is the same as performing six cross product per vertex to retrieve its normal vector. In Figure 4.6 we can visualize a sample of the grid and the equations to calculate normal for point P_n.





$$\overrightarrow{U_n} = P_{n+w} - P_n \qquad \overrightarrow{N_1} = \overrightarrow{U_n} \times \overrightarrow{TL_n}$$
$$\overrightarrow{L_n} = P_{n-1} - P_n \qquad \overrightarrow{N_2} = \overrightarrow{TL_n} \times \overrightarrow{L_n}$$
$$\overrightarrow{D_n} = P_{n+w} - P_n \qquad \overrightarrow{N_3} = \overrightarrow{L_n} \times \overrightarrow{D_n}$$
$$\overrightarrow{R_n} = P_{n+1} - P_n \qquad \overrightarrow{N_4} = \overrightarrow{D_n} \times \overrightarrow{BR_n}$$
$$\overrightarrow{BR_n} = P_{n+w+1} - P_n \qquad \overrightarrow{N_5} = \overrightarrow{BR_n} \times \overrightarrow{R_n}$$
$$\overrightarrow{TL_n} = P_{n-w-1} - P_n \qquad \overrightarrow{N_6} = \overrightarrow{R_n} \times \overrightarrow{U_n}$$

$$\overrightarrow{N_{MWE}} = \sum_{1}^{f} \overrightarrow{N_f}$$

$$\overrightarrow{N_n} = \overrightarrow{N_1} + \overrightarrow{N_2} + \overrightarrow{N_3} + \overrightarrow{N_4} + \overrightarrow{N_5} + \overrightarrow{N_6}$$

FIGURE 4.6: **Top:** Sample of points from the point cloud grid. **Bottom:** Normal computation using MWE Algorithm.

Taking again the example of the flat wall captured by the depth camera Figure 4.2, we are going to assume an equidistant grid and with the objective of reducing computation expenses, we can remove the vectors $\vec{BR}$ and $\vec{TL}$ from the normal calculation to save time in each iteration of points. The normal is going to remain in equilibrium, as we are still taking in consideration opposite vectors from the main axis. An illustration of the grid and the new equations are below in Figure 4.7.

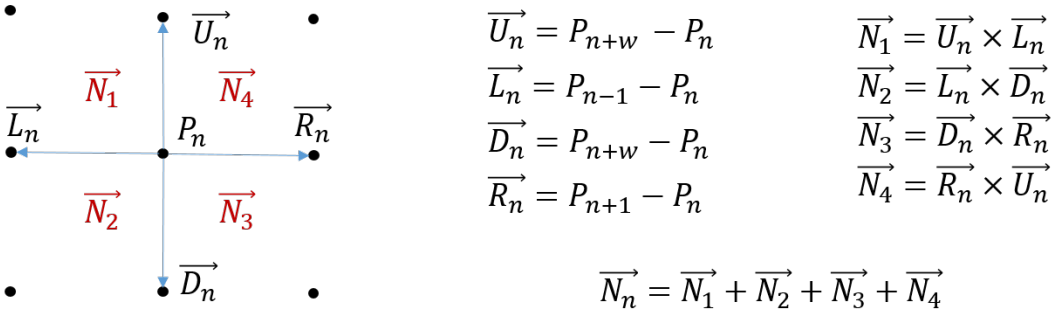


FIGURE 4.7: Normal computation proposed using 4 face normals: Up, Left, Down and Right.

It is true that not always we are going to have a flat background, but if we take a point and its four neighbours top, bottom, left and right, and they are in the same surface the normal calculation will be decently accurate with a fast computation. It is important to note that a point which neighbours are not in the same surface will compute a wrong normal, but for now as we have a single mesh for the whole scene we will ignore that and retake this point on a later chapter. Also there are smoothing techniques to reduce the influence of noise characteristics present in depth sensors when calculating the normals. For details please refer to [12].

Assuming that all points belong to the same surface and that the grid is equidistant, we can calculate the normal in a much simpler and faster way, deriving the computation we just described with arithmetic and matrix operations. For reference the derivation is included in Appendix B. The normal of a 3D coordinate would then be:

$$\vec{N_i} = N_i\hat{x} + N_i\hat{y} + N_i\hat{z} \tag{4.3}$$

$$N_i\hat{x} = \frac{L_i\hat{z} - R_i\hat{z}}{\Delta x} \quad N_i\hat{y} = \frac{D_i\hat{z} - U_i\hat{z}}{\Delta y} \quad N_i\hat{z} = 2$$

$$\Delta x = \frac{R_i\hat{x} - L_i\hat{x}}{2} \quad \Delta y = \frac{U_i\hat{y} - D_i\hat{y}}{2}$$

We are only using this method when the 4 cardinal neighbour points exist for a sample point, if not we would get a wrong and unbalance normal. That means that all the border points of the rectangular grid cannot use this equation, and instead we are going to use the face normals available. If the point belong to the first or the last column or row, with the exception of the corners of the grid, we will have two face normals if we only take in account the cardinal points, which it is acceptable for our calculations, but the corners only have one face normal made by cardinal points, so only for the corners, we are going to use the top right point or bottom left point, to calculate the vertex normal. Please refer to Figure 4.8 for details.

The surface normals should be stored in a vector of 3 floats in the same order as the 3D coordinate position points, in order to use the same index for both position and normal. Regardless the method to compute the vertex normal from the 3D coordinate, the result must be normalized before storing the values.

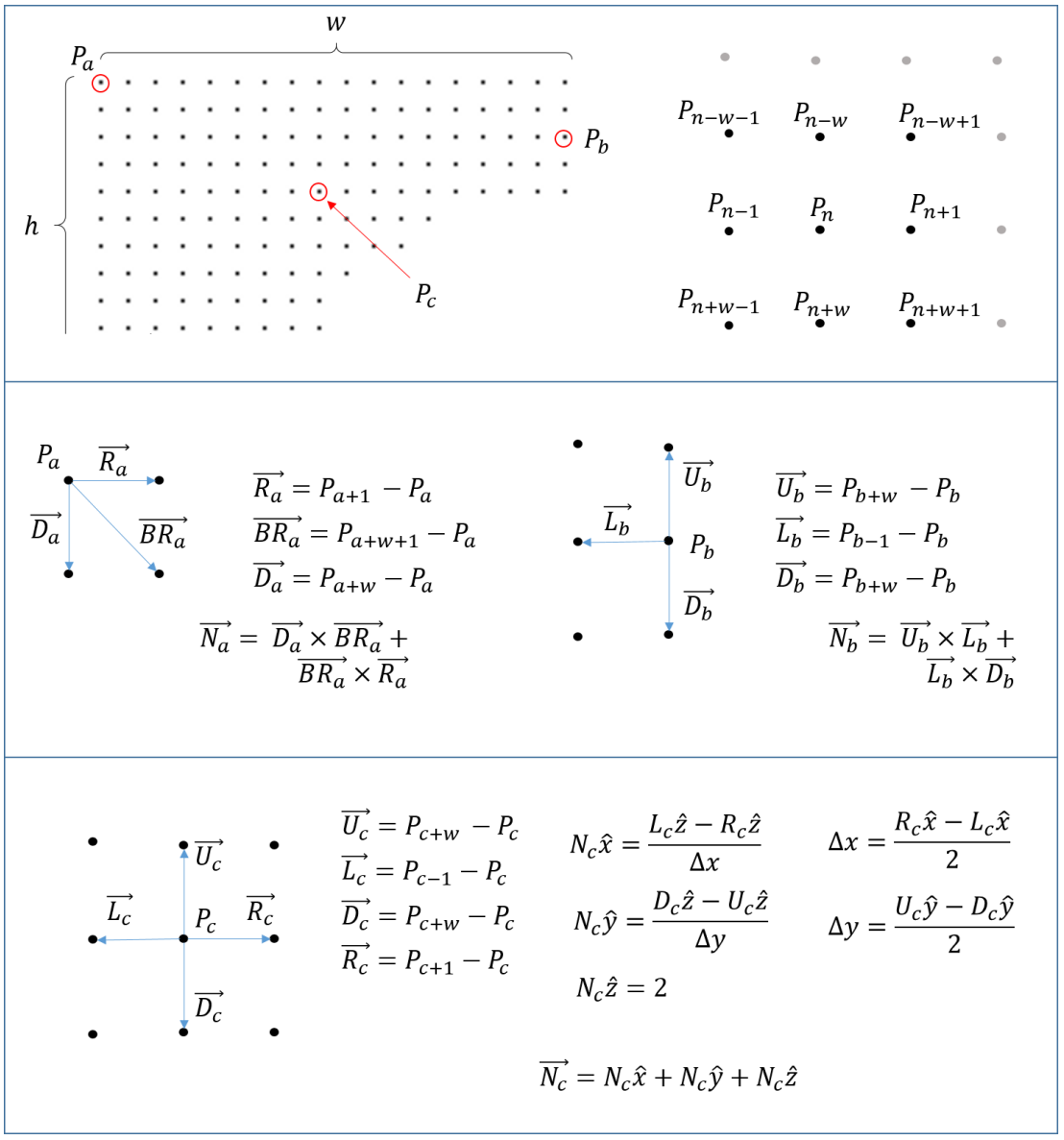


FIGURE 4.8: **Top:** Sample of the point cloud grid. **Center left:** Normal computation of a corner sample point. **Center right:** Normal computation of a border sample point. **Bottom:** Normal computation of a point in the center of the grid using the equation proposed

4.2 Rendering the virtual object

To render a 3D virtual object in OpenGL we need at least its vertex position and a texture. Also we need to set up an environment space where to display the object, which we already have described by the point cloud surface and transformed by the view and perspective projection matrices.

The 3D model object could be imported using a model loader or we can set vertex position to generate a simple model. For this project we are going to work with a 3D cube generated by a set of given coordinates in local space. The cube should be transformed by the same matrices as the point cloud mesh, as they are going to co-exist in the same virtual world.

Rendering the 3D cube in top of the color image texture will create the illusion that the cube is in the real world, getting an Augmented reality experience. The illusion would be destroyed if you try to interact with the cube and the immersion would be lost. To keep the illusion we need to go beyond Augmented reality giving the virtual object real world features. Figure 4.9 shows the limits of Augmented reality as an immersive experience.

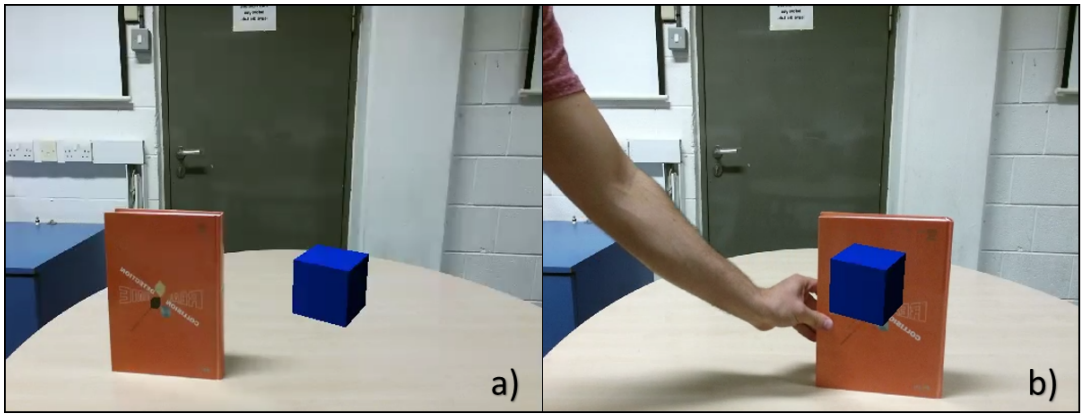


FIGURE 4.9: Virtual object with no occlusions. **a)** A scene with a virtual cube on top of a real table and besides a real book. **b)** Augmented cube is not occluded by the real book.

Referring to Figure 4.9, in order of the application to continue with the illusion of the Mixed reality the cube should be occluded by the book in Figure 4.9(b), as it would happen if a real cube was on the scene. For the rendering program there is only two options, the object is in front of the mesh or behind; the issue now is that we have a single mesh for the whole scene, in other words, by term of the rendering program, the book is part of the table and the wall behind them, instead of being separate things. In that case there is no place in between the book and the wall. To go further, we need to separate the surface in regions by the different objects that appear on the scene, then we could talk about the space between objects and what is behind what.

4.2.1 Depth segmentation

Alexey Abramov et al. present a real-time technique [13] to segment color regions using a Kinect. The technique incorporates color image and depth data to segment color

regions within a real-time video. We are not interested in segment color regions, only depth regions; we do not care if one region has several colors as long as they belong to the same object or depth region, then we are only using some of the concepts described in that paper.

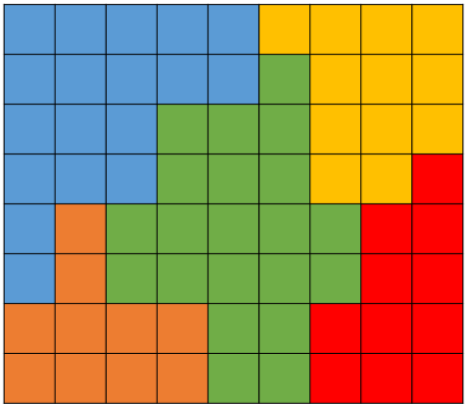
While A. Abramov et al. look for small difference between two pixels in the HSV color space (Hue, Saturation and Value) and using only the depth data as threshold to separate two regions that could be close in color and far in space; we are only looking for small differences between pixels in depth value. If the difference is smaller than a threshold τ a bit will be stored in a binary matrix with the same width and height as the point cloud. As 1 pixel has 8 possible neighbours, [13] used 4 different kernels, looking for difference between pixels in all possible directions: horizontal, vertical, left diagonal and right diagonal. Changing the original method, in the presented paper only horizontal and vertical directions are used, comparing one pixel with the next one, referring to next one as the next right pixel talking for the horizontal direction, and the next below pixel talking for the vertical direction. If the difference is within a threshold τ a bit $B_i = 1$ will be stored. The computation is described by:

$$B_i = \begin{cases} 1 & \text{if } |z_i - z_j| \leq \tau \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Where $< i, j >$ are consecutive points in a row or a column, being j after i, depending the matrix direction in question. After the two matrices have been calculated a bitwise operation AND is performed to find the regions described by the horizontal and vertical limits. The resultant matrix will be a binary image of the regions in the scene. A sample region binary image can be seen in Figure 4.10.

FIGURE 4.10: Binary image of depth-based segmentation.

Since the method is evaluating following a direction (to the right instead of to the left; to the bottom instead of to the top), the bitmap is going to lose some pixels in the regions: the left and top pixels of every border. A sample image depth-based segmented using color is shown in Figure 4.11(a), where the colors represent a region separated by depth from the others, Figure 4.11(b) and (c) are showing the bitmap matrix for horizontal and vertical direction accordingly. Figure 4.11(d) shows the resultant matrix with the regions found in the original image (a). Figure 4.11(e) shows what the matrix look like with the original colors of the regions. Comparing Figure 4.11(a) and (e) the bitmap image has lose details in the borders.



a) Depth-based segmentation using colors.

1	1	1	1	1	0	1	1	1
1	1	1	1	1	0	0	1	1
1	1	1	0	1	1	0	1	1
1	1	1	0	1	1	0	1	0
1	0	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0	1
1	1	1	1	0	1	0	1	1
1	1	1	1	0	1	0	1	1

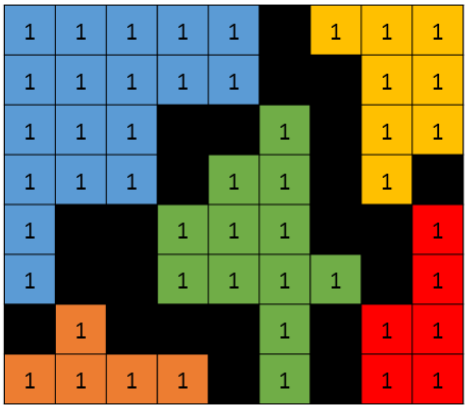
b) Horizontal segmentation matrix

1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	1
1	1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1	0
1	0	0	1	1	1	0	0	1
1	1	1	1	1	1	1	1	1
0	1	0	0	1	1	0	1	1
1	1	1	1	1	1	1	1	1

c) Vertical segmentation matrix

1	1	1	1	1	0	1	1	1
1	1	1	1	1	0	0	1	1
1	1	1	0	0	1	0	1	1
1	1	1	0	1	1	0	1	0
1	0	0	1	1	1	0	0	1
1	0	0	1	1	1	1	0	1
0	1	0	0	0	1	0	1	1
1	1	1	1	0	1	0	1	1

d) Resultant matrix from bitwise AND operation



e) Segments obtained from method

FIGURE 4.11: Segmentation process to capture the depth regions of an image into a binary matrix.

A practical way to reduce the loss is applying a computer vision technique called dilatation by which we expand the high bits in the binary map, in this case by the opposite

direction as we evaluated before. Applying the technique to the previous example the result is going to be as shown in Figure 4.12.

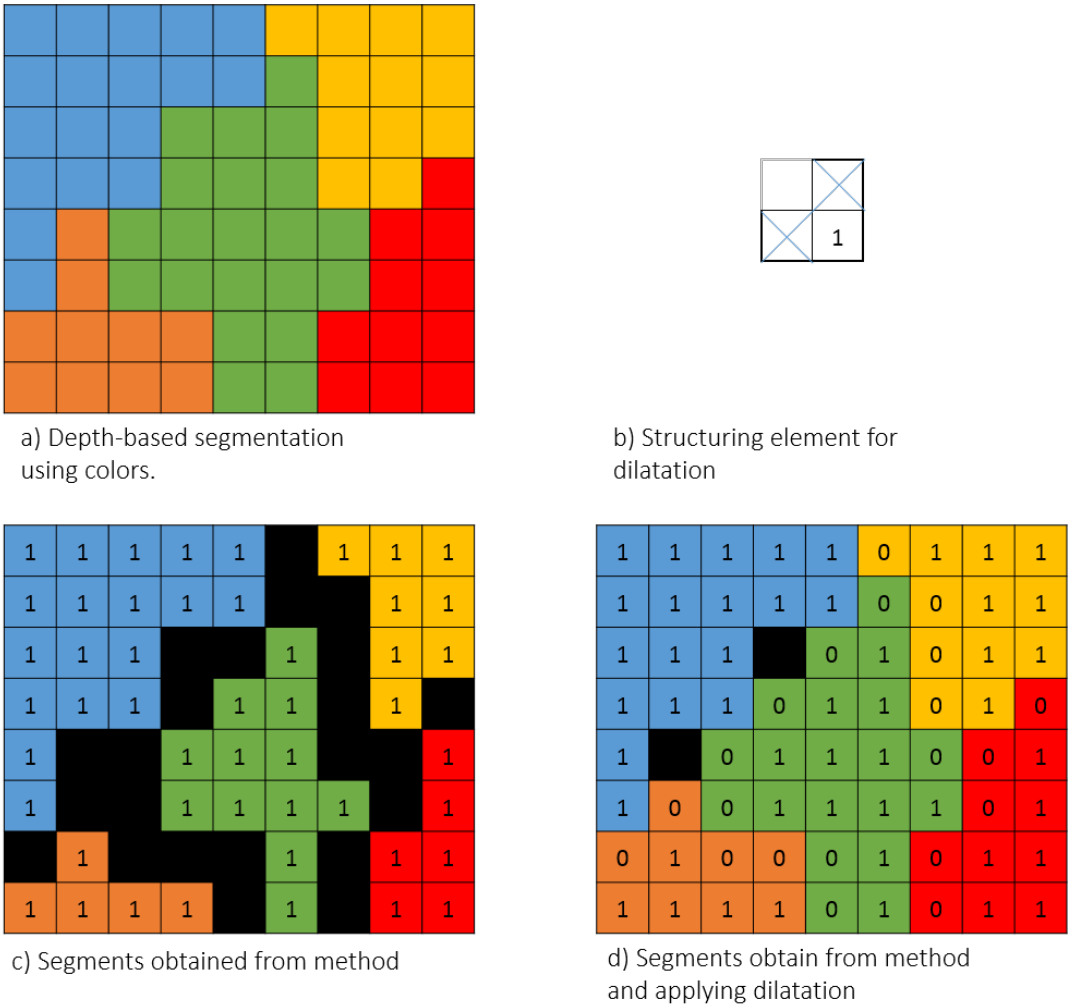


FIGURE 4.12: Applying dilatation to the binary image of the regions.

Before applying dilatation a region detection method must be used, as dilatation is going to make some regions to merge each other. When separating each 3D coordinate to the region it belongs, the dilatation technique could be used to reduce the loss of coordinates in a region.

Taking advantage of the region detection method, the algorithm should be modify to gather important information that the program will use later on. For each region, the application must be capable of storing the smallest values of each axis of the 3D coordinate, as well as the biggest one. These set of points are going to serve as a bounding box around our real object. Also, while storing the 3D coordinates in an array of regions, the program must be capable of storing a binary matrix for each of them, that is going to work as a mask for each region.

4.2.2 Occlusion by depth data

To achieve occlusion the depth of the pixel belonging to a virtual object must be compared to the depth data of the real world; if the virtual object is closer to the camera according to the depth value, that pixel should be rendered, on the other hand, if there is a real object closer to the camera, that pixel should be occluded to make the illusion that the virtual object is behind the real world object.

The operation is going to take place in the GPU thanks to the shaders in OpenGL and saving us computation in the CPU. By now, the real world image is textured as a background in the screen and the virtual object, if any, in top of that. A last layer will be render on top of that where we are going to display the real world image again if the virtual object must be occluded. For this shader, the RGB color image and the depth image are provided, also the position of the virtual object. For each pixel, the z value of the virtual object is compared with the depth data of that same pixel. If the pixel depth is less than the z value of the virtual object, it means a real world object is closer to the camera than the position of the virtual object and this last one must be occluded. When that is the case the RGB color image of the real world must be render in that pixel, now being the reality in top of the virtual world. On the contrary, if the z value is less than the depth information, a pixel with opacity zero must be render or a transparent pixel that let the virtual object in the bottom layer to appear. To render a pixel with opacity feature, the alpha-blending function must be enabled in OpenGL.

To compare both values they should be in the same coordinate space. The depth data is a float number in mm, that means we need to transform it into screen coordinates multiplying the value with the projection and view matrices described before.

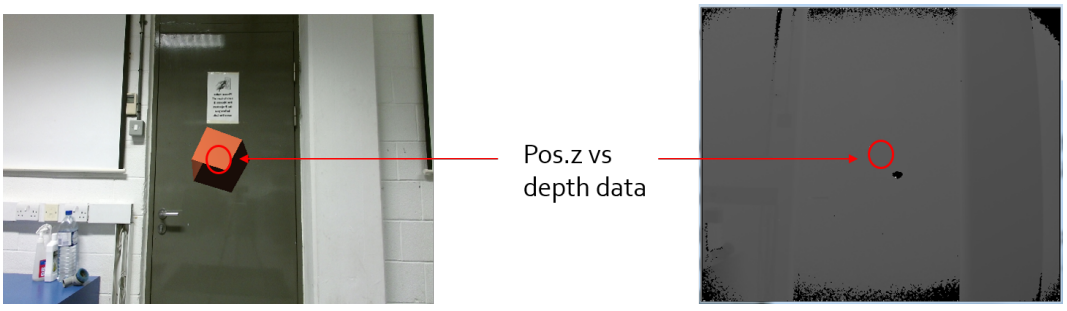


FIGURE 4.13: Checking the position of the virtual object against the depth data.

For a faster computation the Stencil buffer should be used to render the last layer. The stencil buffer allows to discard fragments of the scene that won't be necessary to render, in this case all the scene where there is not a virtual object. We only want to check the depth data where there is a virtual object that might be occluded.

FIGURE 4.14: The depth data is only checked after the stencil buffer. Only where a virtual object is present.

When there are more than one virtual object, the application must be capable of comparing each of their position with the depth image, to occlude only the ones that are beyond a real world object. The method then is a loop iterating between the numbers of virtual objects, enabling depth values between them and resetting the stencil buffer each time. Figure 4.15 shows a scene with two virtual objects at different depths interacting with the real world.

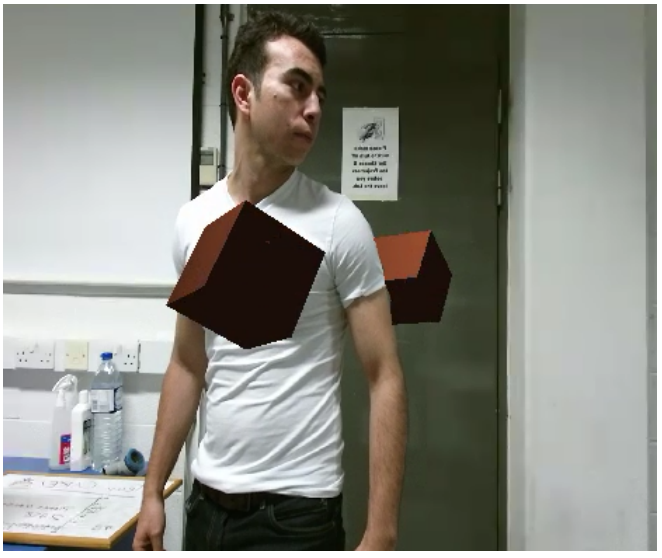


FIGURE 4.15: The user standing between two virtual objects occluding one of them.

4.3 Physics simulation

The final goal to reconstruct dynamics surfaces of the real world is to have a propitious environment where an immersive interactive experience can take place, where rendering an augmented object is only part of. Going further with this task the application is extended to support real world physics simulation between the virtual objects and the dynamic surfaces.

For this project a simple physics module will be implemented based on David Baraff course notes for Rigid Bodies [14]. The physics module can be improved or even a new one can be imported. The reason to implement the module for this project is to test the capacity of the reconstructed dynamics surfaces. The virtual objects used in the current physics module are going to be treated as rigid bodies.

4.3.1 Rigid Body dynamics

To simulate a Rigid body we need two things: its location in the world and its orientation. By updating the position and orientation of the rigid body by each frame, we will end up with a virtual object simulating real world physics properties. The position of the rigid body is described by a 3D vector pointing from the origin to the location of the rigid body center of mass. The orientation is represented by a 4 x 4 matrix, describing the rotation of the rigid body in each coordinate axis. The position and orientation are called "spatial variables" by D. Baraff and we are going to wrap them into a matrix that OpenGL will pass to the virtual object shaders: the Model matrix. Along with the view and projection matrix, the model matrix will transform the vertex position of the virtual object, to be translated and rotated accordingly into the screen from the local space. The equations described in this sections are intended to work with virtual object which center of mass, or the geometric center of the body lies in the origin while in local space. The model matrix it is also capable of scaling the virtual object in any desired direction. To compute the model matrix please refer to Appendix A.

To compute the position $x(t)$ and orientation $R(t)$ of the rigid body, an initial state must be generated, and after that each frame the position and orientation should be updated in function of the linear and angular velocity. The linear velocity is the rate of change of position over time for each axis, being the linear velocity a 3D vector as well. Then the linear velocity will describe the increments or decrements of position for every frame. On the other hand, the angular velocity can be described as the spin of the rigid body noted as a 3D vector. D. Baraff explains: "The direction of $\omega(t)$ gives the direction of the axis about which the body is spinning. The magnitude of $\omega(t)$, $|\omega(t)|$, tells how fast

the body is spinning". Updating the orientation in terms of angular velocity it is not as straight forward as the linear velocity. The relation between angular velocity $\omega(t)$ and orientation $R(t)$ is given by the equation 4.5. For details about this formula, please refer to [14] [15].

$$R(t + \Delta t) = R(t) + \omega(t)^* R(t)\Delta t \tag{4.5}$$

$$\text{where } \omega(t)^* = \begin{bmatrix} 0 & -\omega_z(t) & \omega_y(t) \\ \omega_z(t) & 0 & -\omega_x(t) \\ -\omega_y(t) & \omega_x(t) & 0 \end{bmatrix}$$

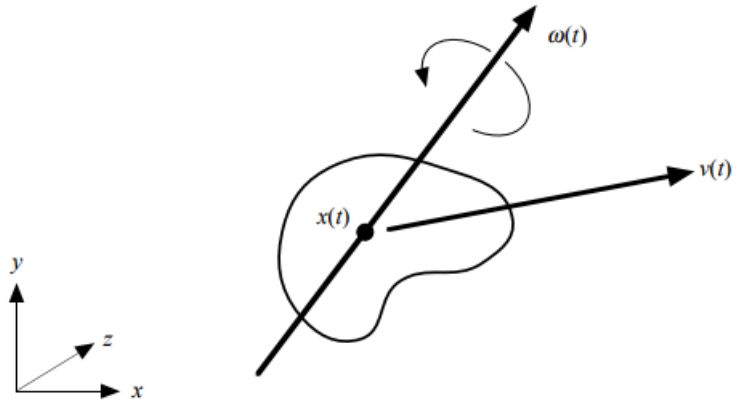


FIGURE 4.16: Linear velocity $v(t)$ and Angular velocity $\omega(t)$ vectors acting on a virtual object.

The incremental update of the matrix $R(t)$ will result in object shearing over time due to accumulation of floating-point error. To avoid this we need to orthogonalize the matrix to get a mutually perpendicular normalized vector axes close to the original matrix. In our project we are using the method proposed by [16]. If C_x, C_y and C_z are the columns from $R(t)$, then the orthogonal matrix can be computed by following this iteration:

$$C_x = \frac{C_x}{|C_x|}$$
$$C_y = C_z \times C_x$$
$$C_y = \frac{C_y}{|C_y|}$$
$$C_z = C_x \times C_y$$
$$C_z = \frac{C_z}{|C_z|}$$

The method is biased, as we are taking an order in which the columns are listed. However this code will give good results for rendering in real-time. For more insight about orthogonalizing matrix please refer to [17]. For fast computation between the Rotation matrix $R(t)$ and the transformation matrices, the first one can be transcribed into a 4 x 4 matrix by adding a last row of zeros with a one at the end, and completing the other 3 rows with zero at the end.

Linear and angular velocity are not always constant, they are also affected by external influences in the form of forces and/or torques (i. e. gravity, contact forces, wind, etc). The forces $F_1(t)$, $F_2(t)$... $F_i(t)$ acting on a rigid body are described by the net force $F(t)$ acting on the object through the center of mass, and it is given by the equation 4.6. The torque $\tau(t)$ on a rigid body is a rotational force that makes the object to spin. The torque depends on the distance from the point where the force is applied to the center of mass, and the net torque acting on the rigid body is given by equation 4.7. Figure 4.17 shows the representation of external forces and torque applied to a rigid body.

$$F(t) = \sum_{i=1}^{i} F_i(t) \tag{4.6}$$

$$\tau(t) = \sum_{i=1}^{i} \tau_i(t) = \sum_{i=1}^{i} (r_i(t) - x(t)) \times F_i(t) \tag{4.7}$$

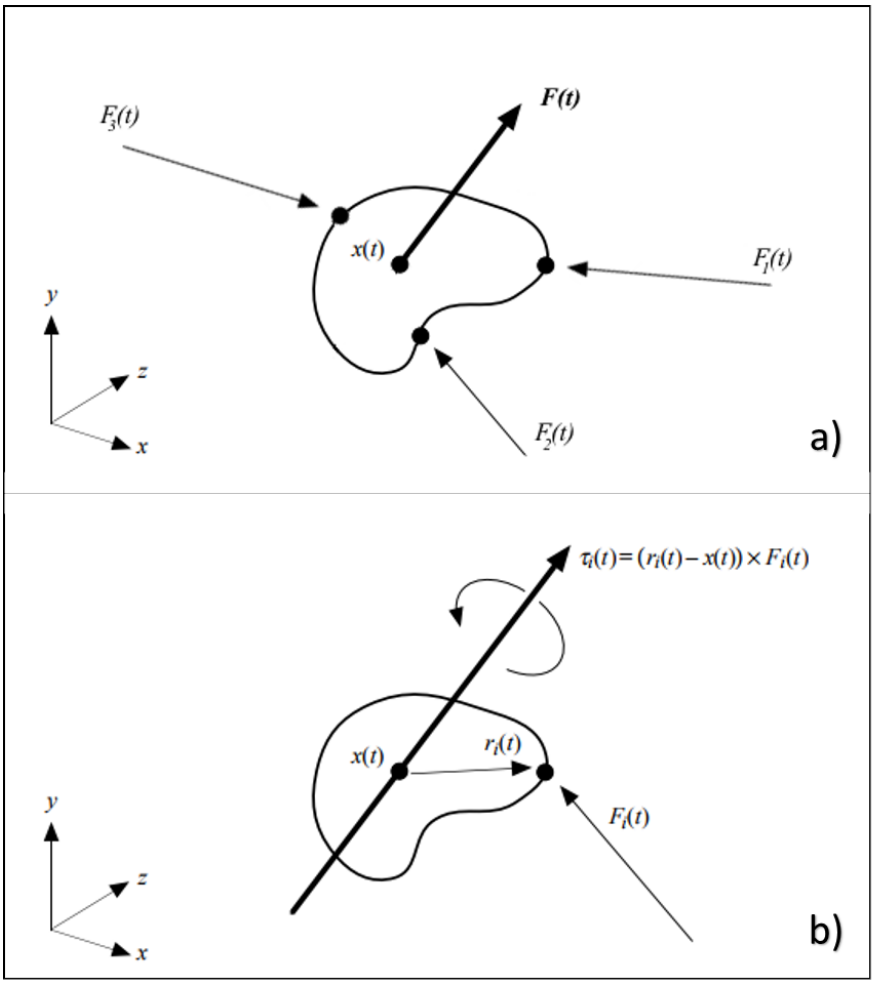


FIGURE 4.17: **a)** A couple of forces and the net force acting on a rigid body. **b)** Torque being generated by the force $F_i(t)$ acting on the rigid body.

To describe how the forces and torques impacts on the linear and angular velocity rate of change, we need to introduce two new concepts: Linear momentum and angular momentum. Linear momentum $P(t)$ is a vector quantity that can be defined as mass in motion and it is represented by $P(t) = mv(t)$. By transforming last equation in terms of linear velocity we have $v(t) = P(t)/m$ and looking for the change in linear velocity we end up with $\Delta v = \Delta P/m$ where $\Delta P = F\Delta t$. For more insight into this equations please refer to [14] [15]. Applying the last equation to the change of linear velocity we get:

$$\Delta v = F\Delta t/m \tag{4.8}$$

Angular momentum $L(t)$ is also a vector quantity that it is represented by the product of the the rigid body rotational inertia and the angular velocity about a particular

axis: $L(t) = \omega(t)I(t)$. $I(t)$ is the moment of inertia or rotational inertia of a rigid body and "describes how the mas in a body is distributed relative to the body's center of mass" [14]. The moment of inertia can be obtained by the following equation $I(t) = R(t)I_{body}R(t)^T$, where I_{body} is a constant of the rigid body and should be precomputed before the simulation. The cube geometry I_{body} calculation can be found in Appendix C. For other types of geometries they could be approximated to the sum of most simple geometries or for more complex bodies, please refer to [14]. The angular velocity and angular momentum are related then by $\omega(t) = I^{-1}(t)L(t)$, and the change of angular velocity by $\Delta\omega = I^{-1}(t)\Delta L$, where $\Delta L = \tau\Delta t$ [14], and thus we have:

$$\Delta\omega = I^{-1}\tau\Delta t \tag{4.9}$$

Now that we have described the computations from external factors to the position and orientation of a rigid body, the program has to compute all this variables for all virtual object per frame. An algorithm example of this computations is shown below:

initialization $x(t)$, $R(t)$, $v(t)$, $\omega(t)$, m;
while $x(t)$ *is within the scene coordinates* **do**
  compute Δt;
  if $F_{1,2...i}(t)$ *are acting on the body* **then**
    compute $F(t)$, $\tau(t)$, Δv, $\Delta\omega$;
    $v(t) = v(t) + \Delta v$;
    $\omega(t) = \omega(t) + \Delta\omega$;
    compute $\omega(t)^*$;
  end
  update $x(t)$, $R(t)$;
  $x(t + \Delta t) = x(t) + v(t)\Delta t$;
  $R(t + \Delta t) = R(t) + \omega(t)^* R(t)\Delta t$;
end

Algorithm 1: Simulation of rigid bodies algorithm.

4.3.2 Collision Detection

The intention of the physics module is to test the physical boundaries recreated by the surface reconstructed detailing the real world. If we throw, for example, a virtual cube object into the scene it should hit somewhere in the reconstructed surface and then collide as a real object would do.

The general principle of a collision detection is to test if any point that compose the virtual object is "close enough" or within a threshold value to any of the vertex position stored in the point cloud of any surface in the scene. By iterating between all possible collisions between all the virtual objects with all other virtual objects or reconstructed surface per frame, would be extremely inefficient and computationally expensive. As an often technique in collision detection, the application is going to broken down this operation in two phases: Broad phase and Narrow phase.

Broad-phase Collision Detection

The aim of the Broad-phase is to make a low cost operation to determine if the virtual object is close to collide with something. If it is not, the method can save the task of checking point by point where is the collision happening.

Our method of surface reconstructions has the ability to obtain the bounding boxes corners of all segments in the scene. In the broad-phase collision detection, the virtual object which has a position vector in a certain frame on time, is going to be checked if it is within a bounding box of all the surfaces in the scene. The position vector points to the center of mass of the virtual object, so the distance between the position vector and the bounding box edges should be greater than a certain threshold that describes the bigger distance between the object origin and its own surface. Figure 4.18 shows an illustrated example of a broad-phase view of a scene.

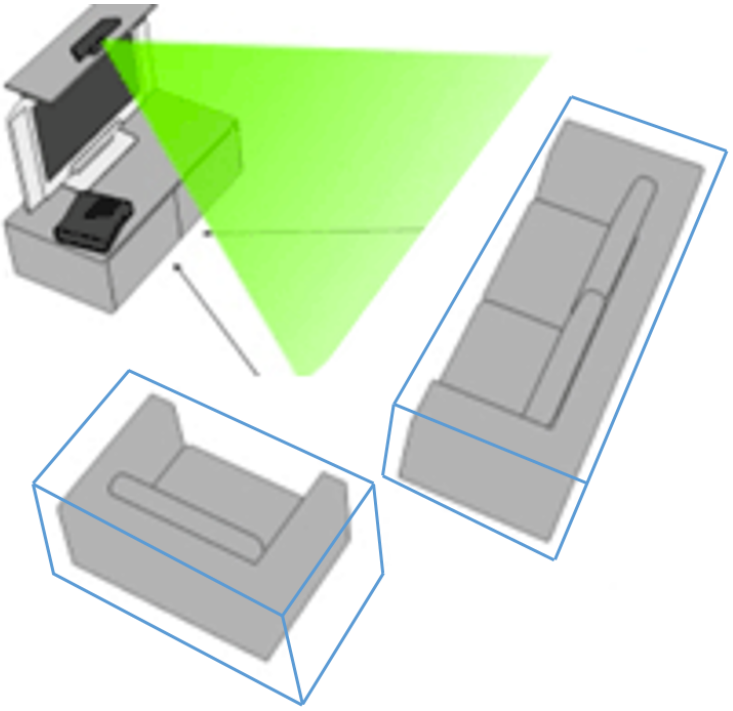


FIGURE 4.18: A scene captured by the depth map and showing the bounding box around the objects in it.

If the virtual object is within the threshold distance with any surface, then we proceed to the narrow phase with that particular region. On the contrary, if no surface was close enough from our virtual object, the application go on with the next virtual object and check or continue to the next frame.

A general algorithm of the broad phase collision detection of a scene with several virtual objects is shown below:

for *Region* $n \in \{1, ..., N\}$ **do**
    get Bounding box B from n;
    if $x(t) \in B$ **then**
        proceed to Narrow-Phase;
    end
end

Algorithm 2: General algorithm for Broad-phase collision detection.

Narrow-phase Collision Detection

The narrow phase collision detection is more intense computationally speaking. This phase aims to obtain which point or points from object A collides with object B. Taking

as a reference the job of Ming C. Lin [18] and Brian Mirtich [19], which focus on a featured-based algorithms using the Voronoi regions to find the smallest distance between two objects by finding first the closest features between them: a vertex V, an edge E or a face F.

For our physics module we are only using the vertex feature, leaving the edge and face testing aside. For improvements on the physics module please refer to [18] [19]. By iterating into the points conformed by the surface and the virtual cube object, to compute the shortest distance between them, the method will obtain which vertex position are close enough to consider them as a collision contact.

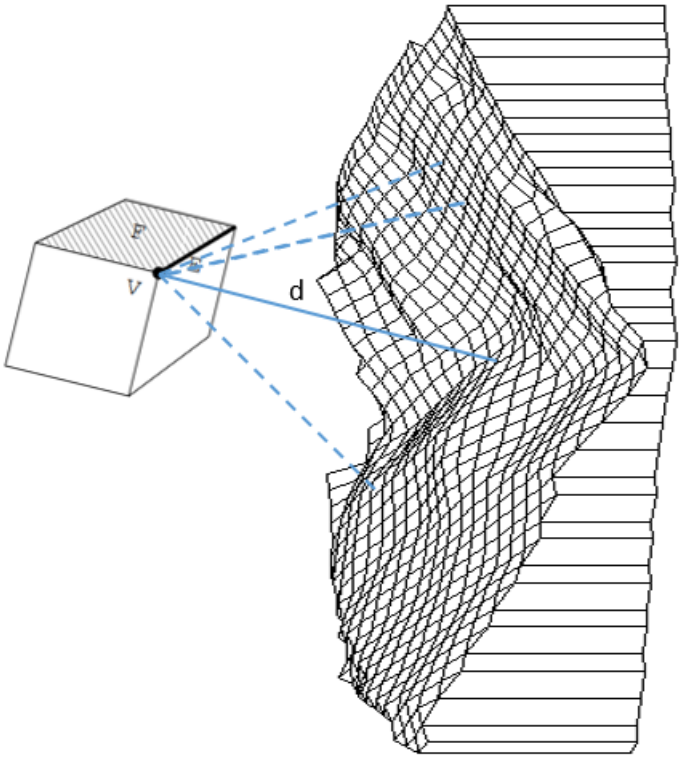


FIGURE 4.19: Representation of the narrow-phase collision detection.

Once the physics module obtains colliding points, then it proceed to compute a collision response. The impulse noted as j is described by the equation derived from D. Baraff [14] calculations:

$$j = \frac{-(1+\epsilon)v_{rel}}{m_A^{-1} + m_B^{-1} + \hat{n}(I_A^{-1}(r_A \times \hat{n}) \times r_A + \hat{n}(I_B^{-1}(r_B \times \hat{n}) \times r_B} \tag{4.10}$$

Where ϵ is the coefficient of restitution, a constant between 0.0 and 1.0; v_{rel} is the relative velocity between the two objects, this value has to be positive meaning that the bodies are moving apart from each other; m_A and m_B are the masses of object A and B; $\hat{n}$ is the normal of the contacting plane; The inverse of the inertial tensor given by $I_n^{-1} = R_n I_{bodyn}^{-1} R_n^T$; and $r_n = p_n - x_n$ is the distance between the contact point p_n in object n to the center of mass x_n from the same object. The relative velocity is given by:

$$v_{rel} = \hat{n}(\dot{p_A} - \dot{p_B}) \tag{4.11}$$

$$\begin{aligned} \text{where,} \qquad \dot{p_A} &= v_A + \omega_A \times (p_A - x_A) \\ \dot{p_B} &= v_B + \omega_B \times (p_B - x_B) \end{aligned}$$

For simplicity, D. Baraff et al. rewrites the equation in the following form:

$$j = \frac{N}{t_1 + t_2 + t_3 + t_4} \tag{4.12}$$

$$\begin{aligned} \text{where,} \quad N &= -(1+\epsilon)v_{rel} \\ t_1 &= m_A^{-1} \\ t_2 &= m_B^{-1} \\ t_3 &= \hat{n}(I_A^{-1}(r_A \times \hat{n}) \times r_A) \\ t_4 &= \hat{n}(I_B^{-1}(r_B \times \hat{n}) \times r_B) \end{aligned}$$

Applying this equation to two virtual objects colliding with each other is straightforward. When one of the objects is a real world surface, then we need to take some considerations. As the surface are not going to move as they are not taking any impulse in real life, we need to consider the surfaces as immovable objects which means infinite mass or $m_n^{-1} = 0$ and Infinite moment of inertia $I_{bodyn}^{-1} = 0$. When the virtual objects collide with a surface we can discard then t_2 and t_4, as they are zero.

To convert the impulse magnitude j to a vector with direction we simply apply the following equation:

$$J = j\hat{n} \tag{4.13}$$

To apply the impulse to the change in linear momentum ΔP and angular momentum ΔL follow this equations [14]:

$$\Delta v = Jm^{-1} \tag{4.14}$$

$$\Delta \omega = I^{-1}(r \times J) \tag{4.15}$$

The complete algorithm for the physics module is shown below:

for *Region* $n \in \{1, ..., N\}$ **do**
  get Bounding box B from n;
  if $x(t) \in B$ **then**
    get Point cloud vertex position PC from n;
    for $i \in \{PC_0, ...PC_I\}$ **do**
      compute distance d from $x(t)$ to PC_i;
      if $d < \tau$ **then**
        collision in Region n at point PC_i;
        compute collision response;
      end
    end
  end
end

Algorithm 3: General algorithm for collision detection.

4.4 Boosting real-time interactions

The application should be capable of reconstructing the surface meshes of the real world scene fast enough, that can be expanded with a robust physics engine to have an immersive experience in real time. Along all the implementations we have made several changes to the methods in literature to save computations, however this section is going to describe new techniques that were implemented after several tests of the application and after doing an analysis about the entire pipeline.

4.4.1 Pre-processing computations

One of the most expensive computations the method applies and that it can be avoid is depth segmentation. Depth segmentation is highly computational: to search depth regions, separate them into segments, compute normals for every frame is not efficient. We can then separate the scene into background and foreground, where the background

is all the user's surroundings that is not likely to change. Then we can avoid to compute the "same static" background in real-time.

To save computation and boost real-time interactions we are proposing to pre-process the scene before the application starts. When the application begins the background will be already stored with all their segments, normals and other features. If we limit the application for a single player, we can even remove depth segmentation for the real-time process, as the only foreground in the application is going to be the User: a single moving segment in the scene.

The new pipeline proposed is a different from what we have. Now there is a pre-processing stage to store the point cloud and compute the normals of the segments that compose the background. It is important to define what is background to our application. By our terms the background would be every segment that is present when the pre-processing is running. Then the User must not be in the scene when the pre-processing stage is running. Our method will run the pre-processing stage for 5 frames, where the point cloud of the background will be averaged and then stored. Afterwards the depth segmentation of the depth will be processed and each region of the background is going to have its point cloud, normals, bounding box and segment mask.

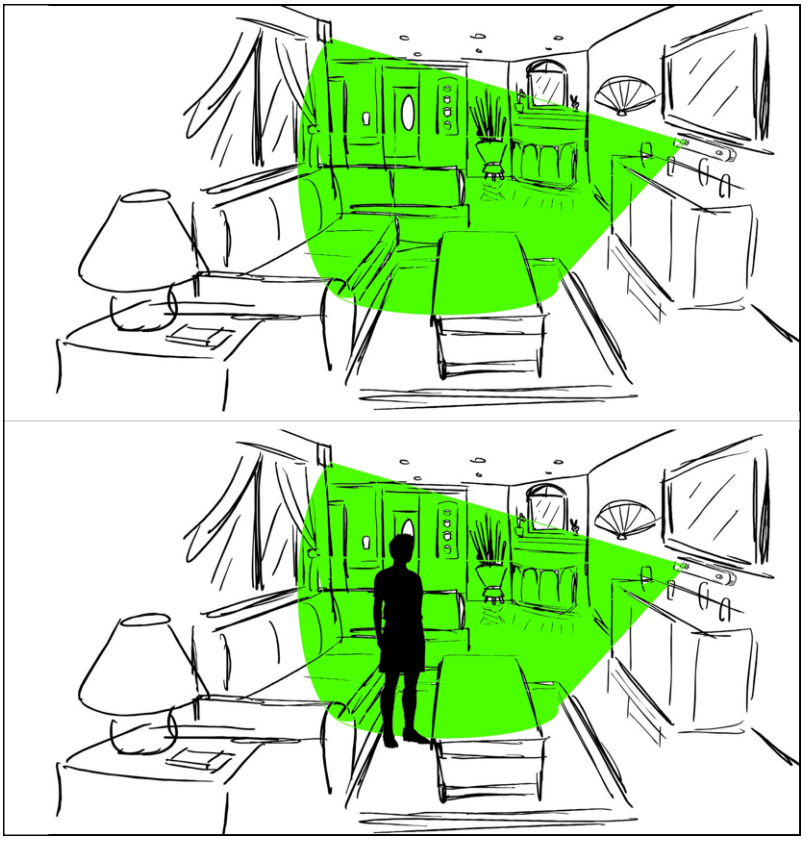


FIGURE 4.20: **Top:** Pre-processing stage of the background surfaces. **Bottom:** Real-time application with the user in the scene.

After the pre-processing stage, the application is going to compare the current point cloud retrieved by the Kinect against the background point cloud. If the difference of depths from the current point cloud and the background of a particular point is greater than a threshold, the point is going to be taken as foreground in this particular frame.

The foreground point cloud is going to be our moving object in the scene. If the application is limited to one person, we can avoid doing depth segmentation in the foreground point cloud. After the comparison is done, the normals should be computed as well as the bounding box and the segmentation mask. The foreground point should be treated as any other region in this particular frame.

Once all the regions are set, the application proceed as before, computing the physics module and rendering the virtual object.

Chapter 5

Results and Discussion

In this section we aim to describe the results of the application implemented and discuss the failure cases found when testing the program.

The results of the application implemented to reconstruct dynamic surfaces and tested with the physics module described before are shown in Figures 5.1, 5.2, 5.3. The real world scene captured consist on a flat wall in front of the camera, a column coming through the wall in the right side and the user. The physics module introduce two virtual cubes at a time: one is thrown from the camera position to the scene and the other is thrown from the right side of the scene to the center of it. In Figure 5.2 and 5.3 a virtual cube is colliding with the user in the third frame and in the next one, the virtual object bounces away from him. In Figure 5.7 **Top** the virtual cube collides with a book held by the user. In Figure 5.1 the user walk between two floating virtual objects and the further one is occluded by him.

FIGURE 5.1: Frame sequence of the user occluding one virtual object in the scene.

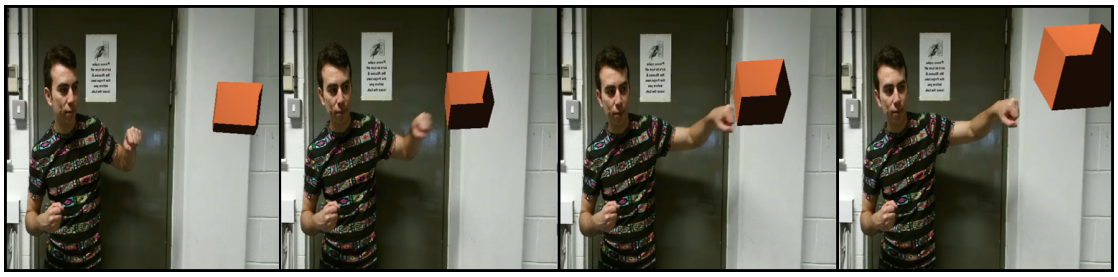


FIGURE 5.2: Frame sequence of the user making a collision with the virtual object with the hand.

FIGURE 5.3: Frame sequence of the user making a collision with the virtual object with his arm.

The application runs an average of 10 frames per second, with two virtual objects on scene and one user, removing segmentation depth in the foreground. The program is run on a 3.40 GHz Xeon PC with a NVIDIA Quadro K2000 graphics card.

5.1 Findings

During the testing of the program there were some cases when the program fails to be consistent with real-world mechanics. In this section we are going to discuss every one of this findings:

- Virtual object going through real objects from behind.
- Virtual object going through a real object that surface is perpendicular to the camera.
- False normals computed.
- Non-static background.

5.1.1 Virtual objects going through real objects

FIGURE 5.4: Frame sequence of a virtual cube colliding against the door in the background and then going through the user from behind.

As the normals are computed from the surface point cloud retrieved from the Kinect, they are dependant of the depth camera view. There are no point cloud to describe the back face of an object facing the Kinect, thus there are no normals on the back of the surface as shown in Figure 5.5. If a virtual object comes from behind a user, it would not collide with the back of the user, as there is no description of his back in the surface constructed. Then the virtual object will go through the user and collide with the front face surface reconstructed from him and the virtual object will speed up as the same direction, as the normal and velocity vector were very similar in direction.

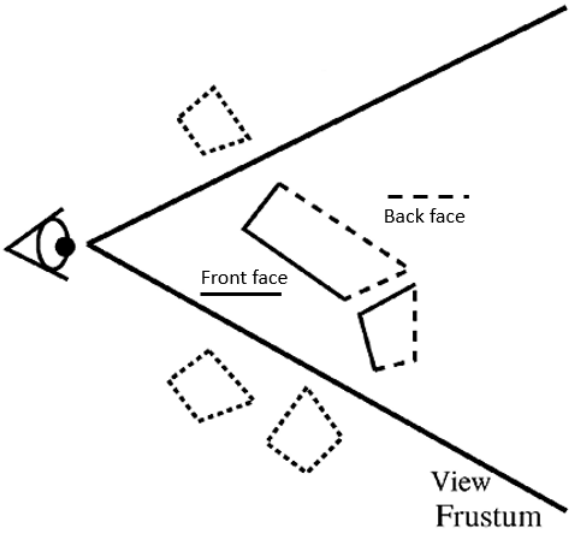


FIGURE 5.5: Visible faces from the camera perspective. **Lines** describe the visible front faces which have a point cloud and normals. **Dashed lines** describe the no visible faces from the camera and which does not have a point cloud nor normals features.

A similar case happens if a surface object is perpendicular o close to be perpendicular from the principal point axis of the depth camera. The point cloud would not be able to describe a surface like that in detail, computing a wrong normal or not computing normal at all. Taking as example Figure 5.6, by analyzing the image we would deduct that the normal N_i from the perpendicular surface should be close to $N_i = [1, 0, 0]$. In order for that to be computed, the point cloud neighbours from point P_i should counter themselves, or in other words there should be at least three points on the same coordinates (x, y) with different z values, which is not possible. Besides, the Normal equation proposed in the implementation was derived from the assumption that the depth z is a function of the camera space $F(x, y) = z$, then that scenario would never be possible.

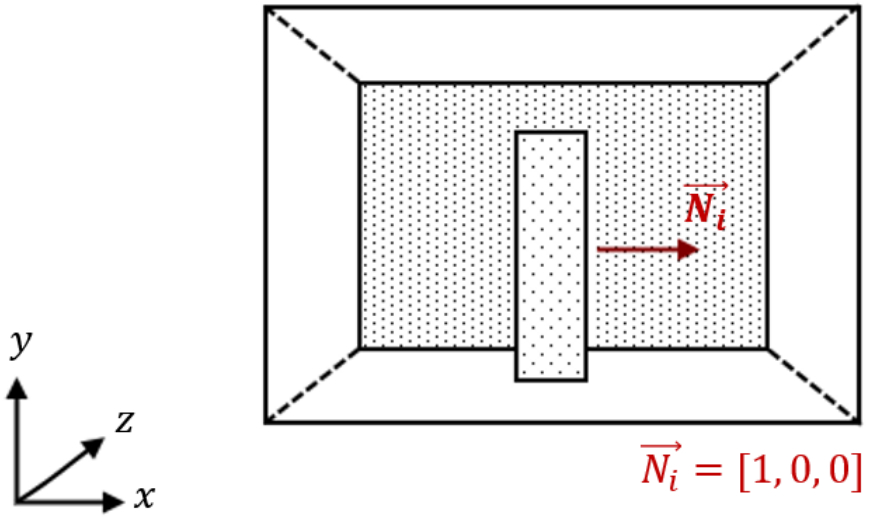


FIGURE 5.6: Normal visualization from a perpendicular surface of a real object facing the camera.

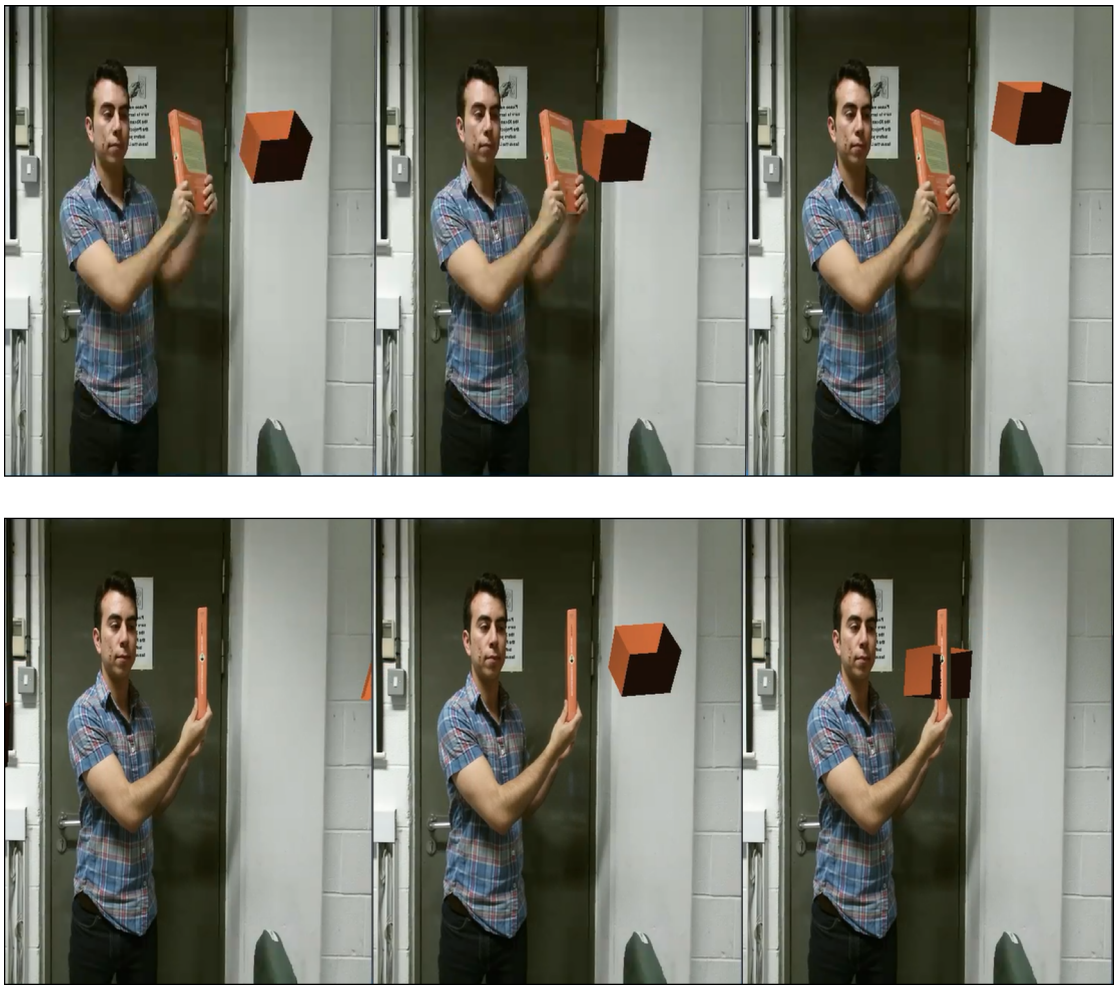


FIGURE 5.7: Collision testing frame sequence. **Top** Frame sequence of a collision of a virtual cube with a book. **Bottom** Frame sequence of a virtual object going through a book, as it is oriented perpendicular to the depth camera.

5.1.2 Unrealistic world mechanics

In some rare cases, the user may detect some anomalies in the virtual objects collisions like bouncing in a direction not logical according to the real world physics. The collisions depends on the normals, and the normals depends on the point cloud. In order to find out what is wrong with the normals, we need to go back to the point cloud. This case is more difficult than the two before. Point clouds and normals are changing every frame, so it is hard to track every normal and look where the problem resides.

Looking closely to the point cloud, there are usually a small number of empty points per frame, some depths that were not picked by the Kinect and the value for that particular position is stored as NaN or Not a number. Having a 3D point coordinate stored with

NaN will generate a wrong vertex normal for the neighbour points for that particular frame.

To measure this error a small script was added to the final version of the application, where for every NaN point stored in the point cloud will be counted and the total number of NaN points can be averaged by n frames. The experiment was ran in four different cases:

a. The average points missing in n frames, checking all the scene when there is no moving objects presents.

b. The average points missing in n frames, checking 77.59% of the scene when there is no moving objects presents.

c. The average points missing in m frames, checking all the scene only when the frame contains a moving object.

d. The average points missing in m frames, checking 77.59% of the scene only when the frame contains a moving object.

We want to make a distinction between the whole scene and the center area of the scene, please refer to Figure 5.8. As we move away from the principal point of the depth camera, the distortion increase and there is more chances that the depth has a minimal error or was not picked at all. With this experiment we want to prove that most of the errors can be found along the borders of the image. For this experiment all the points within the area were checked, no matter the segment region they belong (i.e. wall, column, User).

FIGURE 5.8: **Left** Complete scene retrieved from the Kinect camera. **Right** Center of the scene representing 77.59% of the complete scene, without the shadow border area.

In the table below the results of the experiment are shown. When there is no moving object in the scene, the average missing points are 165.468 in 200 frames. That represents 4.8% of the image, which has a resolution of 64×53. When we focus only in the center part of the scene, the missing points reduce to 15.548 averaged in 200 frames, which is 0.45% of the image. We can observe that the 90.6% of the missing points are found in the borders that represent the 22.41% of the image when there is no moving objects around.

Case	Average missing points	Number of frames	Percentage of the scene registered	Moving objects
(a)	**165.468**	200	100%	No
(b)	**15.548**	200	77.59%	No
(c)	**188.573**	200	100%	Yes
(d)	**57.113**	200	77.59%	Yes

TABLE 5.1: Number of average points missing in the point clouds in n frames per the different cases scenarios.

When there are moving objects the missing points are 188.573 averaged in 200 frames, discarding the frames where no foreground was detected. The percentage of error is higher, being 5.55% of the image. The last experiment dropped a result of 57.113 missing points for the central part of the image when a moving object was on the scene being 1.68% of the whole image. In contrast, we can say that when there are moving objects in the scene there is more chances of errors, and the errors are more likely to present if the object of interest is within the border of the image.

The error of the point cloud is going to affect the normal computation directly, and the error is going to increase. For every NaN point in the center of the image, four normals belonging to the cardinal neighbours are going to be affected. Having an average of 57.113 wrong points in the center of the image, where there is a moving object, we are going to have 228.452 wrong normals in the center of the image. If we focus only in the center of the scene, which covers 77.59% of the image, we are going to end up with 8.67% of wrong normals per frame. From 100 collision that happens in the center of the image, 9 are going to be unrealistic due to a wrong normal computation. That can be approximate to less than 1 every 10.

To notice this unrealistic behaviour the wrong normal computed has to match a collision in that particular point or neighbour point. If there is no collision the missing points won't affect the simulation in any way. In contrast, the method proposed by S. Izadi et al. [5] would notice any point missing from the depth map, as they are reconstructing a

surface for the front face of the user in real-time, so any missing point would be notorious as a hole or shadow on the User model rendered in screen.

Another finding in the current method occurs when the background is not static. By moving an object that is part of the background surfaces when in real-time, the storing background will remain having a surface describing an object that is not more. Collision with an invisible surface might happen when this situation arises. An implementation is being developed to check if part of the foreground has a similar form to the mask of a background surface. If so, an update on the background segments can be done to remove the problem of phantom surfaces.

There is plenty of opportunity to improve the method, specially for the background which is computed in a pre-processing stage. Making estimation about the missing point clouds in function of their neighbours could easily improved the false normal computation. Tracking surface over real-time also could give interesting results and improve the background updating.

Chapter 6

Conclusions and Future work

We have presented a method to create an environment where virtual and real objects can co-exist and interact. The environment is described by a dynamic surface point cloud reconstruction and normal computation of the real world scene by capturing a depth map using a fixed depth camera. Virtual objects are rendered in top of a real world color image texture and they are treated as rigid bodies which dynamics are computed by a physics module. The surface point cloud is matched with the color image, and the virtual objects interact with the surface's reliefs. Occlusions are possible by comparing the depth map of the scene against the position of every virtual object, occluding the pixel that are further away from the z value provided by the depth map. The results of the interaction seems quite promising, with some few exception scenarios where unrealistic world mechanics happens. The collisions seems quite realistic in a little more than 9 of 10 times, the remaining time might not be as realistic but the user must be aware to notice it. The other few scenarios happens when a virtual object comes from behind a real object, going through it, or having a real object perpendicular to the camera and the virtual object would not make a collision. The occlusions looks really well even when there are more than one virtual object in different depths.

While we are quite content with the results of the applications, this work suggest potential areas for future exploration, where we can improve and expand the method. The main interest point for further research are:

- Estimation of back face surface from the regions found in the scene according to the camera perspective.
- Moving camera with 6DOF tracking.
- Hand gesture recognition.

- Migrate the method to see-through moving screen (i.e. HoloLens).

One of the failure cases occurs because there is no information about the back face of the real objects in the scene. Back face's surface can be estimated from the real object location and the camera perspective. Implementing back surface into the application, will improve the physics mechanics illusions of the real world.

Being able to move the camera without affecting or crashing the real-time application would improve greatly the application and we are going to be a step closer to see-through moving devices. However this implementation has many implications for new research problems like dynamic backgrounds and 6DOF tracking with a moving foreground.

Hand gesture recognition is another point of interest for research in this Mixed Reality application. Tracking the hand and giving it special features as: grabbing virtual object and manipulating them, would enhanced the merge of the virtual and real world.

Migrating this technology to a see-through device is the holy grail of the Mixed Reality. Being able to interact with virtual objects as you move freely in the real world is the interest of many and our hope is to enable these experiences of interaction in a new merged world.

Appendix A

Space Transformation

Derivation of the matrices used in the application: Perspective projection, View and Model.

Perspective projection matrix requires the vertical Field of view (FOV) θ in degrees. This parameter depends on the specification of the Depth camera, in the case of the Kinect 2.0 it has a value of 70°. The Aspect ration $A = width/height$ from the screen to display. The near plane $z_n = 0.1$ and far plane $z_f = 100.0$.

$$M_{projection} = \begin{bmatrix} \frac{1}{A*\tan(\theta/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\theta/2)} & 0 & 0 \\ 0 & 0 & \frac{-(z_f+z_n)}{z_f+z_n} & \frac{-(2*z_f*z_n)}{z_f-z_n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{A.1}$$

View matrix requires the translation T and orientation R obtained by the stereo cameras calibration. This values should be really small, so the view matrix could be omitted, but in order to get better result it should be applied.

$$M_{view} = R * \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.2}$$

Model matrix describes the position and orientation of a virtual object transformed into the screen scene. It requires the translation T_n and orientation R_n of virtual object n. Also scaling can be applied to the virtual object as a vector S_n.

$$M_{model} = R_n * \begin{bmatrix} S_{nx} & 0 & 0 & T_{nx} \\ 0 & S_{ny} & 0 & T_{ny} \\ 0 & 0 & S_{nz} & T_{nz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.3}$$

Point cloud transformation PC_i of a point P_i is given by:

$$PC_i = M_{projection} * M_{view} * P_i \tag{A.4}$$

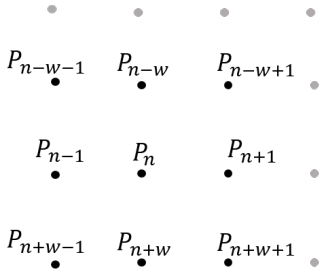
Virtual object transformation vertex position VO_j of a point P_j is given by:

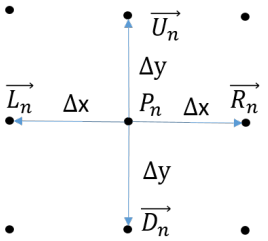
$$VO_j = M_{projection} * M_{view} * M_{model} * P_j \tag{A.5}$$

Appendix B

Normal equation

Derivation of the normal equation from a 3D grid with equidistant points in x and y.





$$P_n(x, y, z_0)$$
$$P_{n-w}(x, y + \Delta y, z_U)$$
$$P_{n-1}(x - \Delta x, y, z_L)$$
$$P_{n+1}(x + \Delta x, y, z_R)$$
$$P_{n+w}(x, y - \Delta y, z_D)$$

$$\overrightarrow{U_n} = P_{n+w} - P_n \qquad \overrightarrow{U_n} = [0, \Delta y, z_U]$$
$$\overrightarrow{L_n} = P_{n-1} - P_n \qquad \overrightarrow{L_n} = [-\Delta x, 0, z_L]$$
$$\overrightarrow{D_n} = P_{n+w} - P_n \qquad \overrightarrow{D_n} = [0, -\Delta y, z_D]$$
$$\overrightarrow{R_n} = P_{n+1} - P_n \qquad \overrightarrow{R_n} = [\Delta x, 0, z_R]$$

$$\Delta x = \frac{R_n\hat{x} - L_n\hat{x}}{2}$$
$$\Delta y = \frac{U_n\hat{y} - D_n\hat{y}}{2}$$

$$\overrightarrow{N_1} = \overrightarrow{U_n} \times \overrightarrow{L_n} \qquad \overrightarrow{N_1} = [z_L\Delta y, -z_U\Delta x, \Delta x\Delta y]$$
$$\overrightarrow{N_2} = \overrightarrow{L_n} \times \overrightarrow{D_n} \qquad \overrightarrow{N_2} = [z_L\Delta y, z_D\Delta x, \Delta x\Delta y]$$
$$\overrightarrow{N_3} = \overrightarrow{D_n} \times \overrightarrow{R_n} \qquad \overrightarrow{N_3} = [-z_R\Delta y, z_D\Delta x, \Delta x\Delta y]$$
$$\overrightarrow{N_4} = \overrightarrow{R_n} \times \overrightarrow{U_n} \qquad \overrightarrow{N_4} = [-z_R\Delta y, -z_U\Delta x, \Delta x\Delta y]$$

$$\overrightarrow{N_n} = \overrightarrow{N_1} + \overrightarrow{N_2} + \overrightarrow{N_3} + \overrightarrow{N_4}$$
$$\overrightarrow{N_n} = [2z_L\Delta y - 2z_R\Delta y, \quad 2z_D\Delta x - 2z_U\Delta x, \quad 4\Delta x\Delta y]$$
$$\overrightarrow{N_n} = [2z_L\Delta y - 2z_R\Delta y, \quad 2z_D\Delta x - 2z_U\Delta x, \quad 4\Delta x\Delta y] * (2\Delta x\Delta y)^{-1}$$
$$\overrightarrow{N_n} = \left[\frac{z_L - z_R}{\Delta x}, \quad \frac{z_D - z_U}{\Delta y}, \quad 2\right]$$
$$\overrightarrow{N_n} = \left[\frac{L_n\hat{z} - R_n\hat{z}}{\Delta x}, \quad \frac{D_n\hat{z} - U_n\hat{z}}{\Delta y}, \quad 2\right]$$

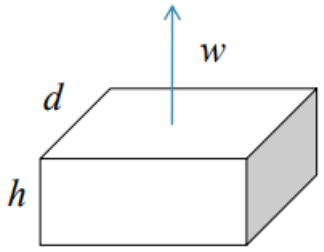
Appendix C

Inertial tensor cube approximation

Here follows the Inertial tensor of the cube used as a virtual object:

$$I_{bodycube} = \begin{bmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(h^2+w^2) \end{bmatrix} \tag{C.1}$$

where m is the total mass of the body.



Bibliography

[1] Paul Milgram and Fumio Kishino. A taxonomy of mixed reality visual displays. vol. E77-D, no. 12:1321–1329, 12 1994.

[2] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *IEEE Computer Graphics and Applications*, 21(6):34–47, Nov 2001.

[3] N. Navab, B. Bascle, M. Appel, and E. Cubillo. Scene augmentation via the fusion of industrial drawings and uncalibrated images with a view to marker-less calibration. In *Augmented Reality, 1999. (IWAR '99) Proceedings. 2nd IEEE and ACM International Workshop on*, pages 125–133, 1999.

[4] David E. Breen, Ross T. Whitaker, Eric Rose, and Mihran Tuceryan. Interactive occlusion and automatic object placement for augmented reality. *Computer Graphics Forum*, 15(3):11–22, 1996.

[5] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 559–568, New York, NY, USA, 2011. ACM.

[6] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, Oct 2011.

[7] J Blake, F Echtler, and C Kerl. libfreenect2 project, 2015.

[8] V. Vezhnevets, A. Velizhev, A. Yakubenko, and N. Chetverikov. Gml c++ camera calibration toolbox. http://graphics.cs.msu.ru/en/node/909, 2013.

[9] Cha Zhang and Zhengyou Zhang. *Calibration between Depth and Color Sensors for Commodity Depth Cameras*, pages 47–64. Springer International Publishing, July 2011.

[10] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–629, June 1971.

[11] Shuangshuang Jin, Robert R. Lewis, and David West. A comparison of algorithms for vertex normal computation. *Vis. Comput.*, 21(1-2):71–82, February 2005.

[12] S. Holzer, R. B. Rusu, M. Dixon, S. Gedikli, and N. Navab. Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2684–2689, Oct 2012.

[13] A. Abramov, K. Pauwels, J. Papon, F. Wörgötter, and B. Dellen. Depth-supported real-time video segmentation with the kinect. In *2012 IEEE Workshop on the Applications of Computer Vision (WACV)*, pages 457–464, Jan 2012.

[14] David Baraff. Physically based modeling: Rigid body simulation. *SIGGRAPH Course Notes, ACM SIGGRAPH*, 2(1):2–1, 2001.

[15] Herbert Goldstein. *Classical mechanics*. Pearson Education India, 2011.

[16] J. Dingliana. Rigid body unconstrained motion. https://www.scss.tcd.ie/John.Dingliana/cs7057/cs7057-1415-03a-RigidBodyMotion.pdf, 2017.

[17] Fletcher Dunn and Ian Parberry. *3D math primer for graphics and game development*. CRC Press, 2015.

[18] Ming Chieh Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, 1993. AAI9430587.

[19] Brian Mirtich. V-clip: Fast and robust polyhedral collision detection. Technical Report TR97-05, MERL - Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, July 1997.