Seamless Integration of Real and Virtual Environments

by

Tom Noonan, B.A.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

September 2014

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.

____
Tom Noonan

September 1, 2014

Permission to Lend and/or Copy

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

Tom Noonan

September 1, 2014

Acknowledgments

I would like to thank my supervisor Dr. John Dingliana for the invaluable advice and guidance offered throughout the length of this dissertation.

Tom Noonan

University of Dublin, Trinity College
September 2014

Seamless Integration of Real and Virtual Environments

Tom Noonan
University of Dublin, Trinity College, 2014

Supervisor: Dr. John Dingliana

Within the field of augmented reality there are a number applications already available which allow for physical interaction between a real scene and certain virtual objects. Often times though this interaction is quite limited and coarse in detail. The objective of this dissertation is to design and create a model which allows for detailed and accurate physical interaction between a real scene and various virtual physical phenomenon. In particular, we wish to be able to expose virtual objects such as rigid bodies, cloth and fluid to a view of a real scene and have them behave like they belong in this scene. To achieve this, the Microsoft Kinect is used to scan in information about the real scene and Kinect Fusion is used to track the motion of the camera and build a fully volumetric 3D representation of the scene. The physically relevant information is extracted from the scene and a unified particle solver is then used to simulate the physical interactions. The results presented far surpass the capabilities of current alternatives

found in games today and real-time performance is maintained by executing nearly everything on the GPU.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **xi**

List of Figures **xii**

Chapter 1 Introduction **1**

1.1 Motivation 2
1.2 Objectives 3
1.3 Dissertation Outline 4

Chapter 2 State of the Art Review **5**

2.1 Input Device 5
2.1.1 Structure Light Scanner 6
2.1.2 Time of Flight Camera 7
2.2 Scene Reconstruction & Camera Tracking 7
2.2.1 Multi-View Stereo (MVS) 8
2.2.2 Simultaneous Localization and Mapping (SLAM) 8
2.2.3 Structure from Motion (SfM) 9
2.2.4 Iterative Closest Point 11
2.2.5 RGB Plus Depth 12
2.3 Volumetric Representation 13
2.3.1 Voxels 13

2.3.2 Signed Distance Function . . . 14
2.3.3 Uniform Spatial Subdivision . . . 14
2.3.4 Hierarchical Structure . . . 15
2.3.5 Spatial Hashing . . . 16
2.4 Surface Extraction & Rendering . . . 18
2.4.1 Raycasting . . . 18
2.4.2 Compositing of Real and Virtual . . . 18
2.5 Kinect Fusion . . . 19
2.6 Extraction of Physically Relevant Information . . . 20
2.6.1 Particles . . . 20
2.6.2 Meshes . . . 21
2.6.3 Primitive Shapes . . . 21
2.6.4 Signed Distance Functions . . . 22
2.7 Physics Calculations . . . 22
2.7.1 Physics Engines . . . 23
2.7.2 Particle Systems . . . 23
2.7.3 Position Based Dynamics (PBD) . . . 24
2.7.4 Collision Detection & Response . . . 24
2.7.5 Rigid Bodies . . . 25
2.7.6 Cloth . . . 26
2.7.7 Fluid . . . 27
2.7.8 Unified Particle Physics . . . 28

Chapter 3 Design **29**
3.1 Key Design Decisions . . . 29
3.2 Method of Input . . . 30
3.3 Representing the Real World Information . . . 31
3.4 Physical Solver . . . 32
3.5 Keeping It All Real-Time . . . 32
3.6 Rendering . . . 33
3.7 Languages, Tools & Libraries . . . 34

Chapter 4 Implementation **36**

4.1 The Kinect . . . 36
4.1.1 Scanning in the Real Scene Information . . . 36
4.1.2 Kinect Fusion . . . 37
4.1.3 Extracting the Physical Information . . . 38
4.2 The Physics . . . 38
4.2.1 The Particle . . . 38
4.2.2 Simulation Loop . . . 40
4.2.3 Timestep . . . 41
4.2.4 Neighbour Detection . . . 42
4.2.5 Collision Response . . . 44
4.2.6 Rigid Bodies . . . 44
4.2.7 Cloth . . . 46
4.2.8 Fluid . . . 48
4.3 Rendering . . . 52
4.3.1 Real World - RGB Image . . . 52
4.3.2 Real World - 3D Mesh . . . 53
4.3.3 Rigid Bodies . . . 54
4.3.4 Cloth . . . 55
4.3.5 Fluid . . . 56

Chapter 5 Results & Evaluation **61**
5.1 Quality of Interaction . . . 61
5.1.1 Fluid . . . 62
5.1.2 Cloth . . . 63
5.1.3 Rigid Bodies . . . 64
5.1.4 Kinect PoV . . . 65
5.2 Performance . . . 66
5.3 Evaluation . . . 68

Chapter 6 Conclusion **70**
6.1 Summary . . . 70
6.2 Limitations & Future Work . . . 70

Appendix A Appendix **73**

A.1 Links to Videos 73
A.2 Source Code 74

Appendices 73

Bibliography 75

List of Tables

5.1 Time taken per frame for the typical physics simulations. 67

List of Figures

2.1 In the image on the left, the infrared line pattern emitted by many structured light cameras. In the image on the right, the dot pattern emitted by the Kinect. 6

2.2 On the left, a sample MVS setup in [1]. In the centre, approximate translations leading to increased error in SLAM[2]. On the right, points of interest detected in a SfM approach[3]. 11

2.3 On the left, visual representation of the shallow hierarchy method in Chen et al.[4]. On the right, visual representation of the spatial hashing method used by Teschner et al.[5]. 17

2.4 On the left, a vertical slice of the volumetrically represented SDF. On the right, ray casting to approximate the surface. 19

2.5 A 2,000 element point cloud (left) represented as 372 basic shapes (right) using [6]. 22

4.1 On the left is the virtual representation of the scene overlaid with the RGB image. On the right is an example of the depth image used for occlusion 53

4.2 Scene rendered as a 3D mesh 54

4.3 Cube visualized just as particles and with final render 54

4.4 Cloth visualized just as particles and with final render 56

4.5 The various stages of the fluid renderer. Images listed in order depict blurred depth, normals, unblurred thickness, reflection, refraction and the final composite 60

5.1 Fluid being dropped on a desk 62

5.2 Fluid being dropped on a chair 62
5.3 Cloth being dropped on a desk 63
5.4 Cloth being dropped on a person 63
5.5 Cubes being dropped on a chair 64
5.6 Kinect POV of fluid being dropped on a desk 65
5.7 Kinect POV of cloth being dropped on some chairs 65
5.8 Time taken per frame by Kinect Fusion with varying amounts of voxels representing a 4metre x 4metre scene 66

Chapter 1

Introduction

When it comes to seamless integration between the real and the virtual, two main areas exist which affect a user's ability to perceive a difference between the two. The first is how the two look together, and the second is how they behave together. Within the field of augmented reality, extensive work has been carried out on the former of these two areas. One approach is to try and make the virtual objects look as realistic as possible and to attempt to match them to the current lighting of the real scene being augmented[7]. And another approach is to abandon the idea of trying to make everything look like it belongs to the real scene, and instead just try to render both the real and virtual with a similar, stylized approach[8]. Not as much attention, however, has been paid to how the real and virtual behave and interact physically with one another. The work of this dissertation will be focused almost entirely on this second area, researching and demonstrating the type of physical interaction possible between the real and virtual in an augmented reality environment.

1.1 Motivation

With the recent release of the next generation of gaming consoles, depth sensing cameras are more and more becoming a feature commonly found in the average household living room. Many applications have been developed specifically for use with these cameras, some of which allow certain amounts of physical interaction between a user and virtual objects within the application. An example of this is Microsoft's Kinect Party [9]. Kinect Party consists of a number of different modes and within most of these modes there is no particular goal. Rather users are just prompted to perform certain actions in front of the Kinect camera as to create humorous results in the form of augmented reality which can then be displayed on the screen connected to the console. Some of these modes allow for users to bat around virtual balloons or watch virtual lava flow in around them. While these features sound intriguing at first, in reality when you observe these modes in action it is quite obvious to see that the actual physical interaction is extremely limited and coarse in detail. As such the the behaviour of the virtual objects displayed on screen are similarly coarse and the results are very rarely what you might describe as realistic.

In most cases these type of interactions do not limit the user experience to a large degree in the context of that particular game/application as the input required is similarly simple. The modes themselves are not meant to have much depth and rather are just for a couple minutes of fun. This does however largely limit what type of gameplay you can achieve as a whole, in future games, if you are forced to keep within the constraints of such simple interaction. A large part of designing games which are meant to contain an amount of depth and keep a user playing for a longer period of time is to maintain some form realism and immersion. At the moment, developers are content with this limited form of interaction. However if detailed physical interactions could be simulated between the real and the virtual it would only serve to improve the user experience and lead to more possibilities and depth in potential gameplay or other applications.

1.2 Objectives

With all this in mind, the main aim of this dissertation is to design and create a model which allows for detailed and accurate physical interaction between a real scene and various virtual physical phenomena. Involved in this is researching the various approaches available for scanning in the real world information, along with the various method of handling the physical interactions. Finding two which work well together in the context of this problem and combining them together to make interactions as realistic as possible. Ultimately the goal is to be able to expose a real scene to virtual rigid bodies, cloth and fluid, and have the virtual objects behave like they belong in the real scene.

Because a model like this would primarily find its use in applications like games, it is also important that the eventual implementation be suitable for convenient use in any scene. This means the system must be easy to set up and capable of working in different environments and capable of handling various lighting conditions. Many existing augmented reality applications require visual prompts such as augmented reality tags or fiducial markers, which we also wish to be able to avoid.

Another important factor which must be considered when designing a system for use in games is performance. The implementation must be able to maintain real-time frame rates. As such the aim is to execute as much the implementation as possible on the GPU.

And finally, very little previous work has been done in the area of trying obtain a high level of physical interaction between the real and the virtual using a simple, consumer grade depth sensing camera. Aside from just provide a specific implementation which achieves these types of interactions, it is also the aim that this dissertation could serve as a point of reference to others who wish to continue research in the area. Be it through expansion or improvement of the implementation developed in this work, or just through using the knowledge gained from the considerations and insight detailed in this text.

1.3 Dissertation Outline

The rest of this dissertation is outlined as follows:

Chapter 2 - State of the Art Review looks at the current state of the art with regard to the two main areas involved in this research, 3D scene reconstruction and physical solvers. A detailed discussion of the pre-existing techniques is provided and some of the advantages and disadvantages of these techniques within the context of the objectives of this dissertation are mentioned.

Chapter 3 - Design describes a high level overview of the design planned for the eventual implementation. Some of the key design decisions which had to be made are highlighted and a discussion behind the various thought processes which lead to the eventual design choice is provided.

Chapter 4 - Implementation gives a detailed step by step description of the implementation used to develop the final application.

Chapter 5 - Results & Evaluation demonstrates some of the results achieved and evaluates these results in the context of realism and performance. Also provides a discussion on the relative success of the dissertation with regards to other alternatives available presently.

Chapter 6 - Conclusion summarizes the work done and main results achieved. Also describes some of the limitations of the final implementation and presents ideas for future work to overcome these limitations.

Chapter 2

State of the Art Review

In this chapter we explore the current state of the art with regards to the areas of research this dissertation will be focusing on. The aim of this is to gain an increased understanding of the current techniques utilized in existing literature, and also to gain a better idea of what pre-existing techniques or areas can further be built upon to develop our own contribution to this area of research. This section will begin with a focus on the area of scanning in real 3D scenes and representing them virtually, and discuss the various methods already developed for this purpose. It will then move on to the topic of physical integration between these real scenes and the purely virtual, and discuss some of methods available for achieving this purpose. Finally, some of the current state of the art in techniques used for physical simulation are presented.

2.1 Input Device

The first barrier in trying to scan in a real world information is what device to use as a method of input. This section briefly describes the very basic theory behind how certain depth sensing cameras work.

2.1.1 Structure Light Scanner

A structured light scanner is a device which uses projected light patterns and a camera to measure the three-dimensional shape of an object. Structured light scanners work by generating a known pattern of light within the device. This pattern is then projected outwards onto the scene to be recorded, where the pattern will be distorted depending on the geometry of the scene and where the light collides with visible objects. A camera, also within the device, will then read in the scene illuminated in the distorted pattern, and compare this with the original known pattern to gain depth information from the scene. The calibration between the projector and the camera must be known and taken into account. Up until relatively recently, structured light scanners were unable to affordably provide high frame rates for full images with a reasonable resolution, but this all changed with the introduction of the Microsoft Kinect. Based off of the range camera technology developed and patented by PrimeSense[10], the Kinect uses a modified structured light technique to build its depth map. Instead of using the time-varying structured light patterns widely applied in previous devices, the Kinect uses a fixed irregular pattern which consists of a very large number of dots. These dots are produced by an infrared laser LED and a diffractive optical element. The Kinect determines the disparities between the emitted light beam and the observed position of the light, where the identity of each dot is determined by utilizing the irregular pattern. Once the identity and position of a dot is known, the depth information can be extracted.

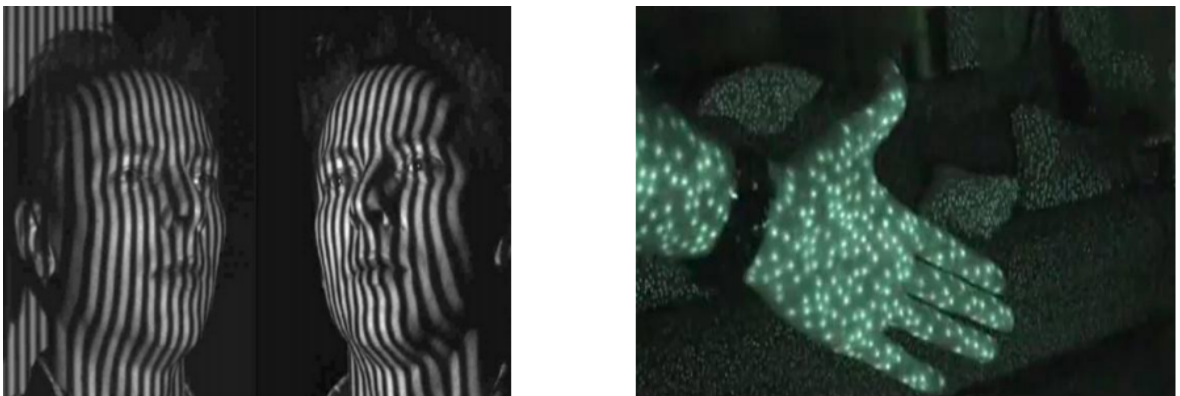

Figure 2.1: In the image on the left, the infrared line pattern emitted by many structured light cameras. In the image on the right, the dot pattern emitted by the Kinect.

2.1.2 Time of Flight Camera

A 3D time of flight(TOF) camera works in a manner very similar to the familiar concepts of radar. The camera emits a specific type of light and essentially uses the time taken for the light to reflect back to its detector to determine the depth of a particular point. More specifically, TOF cameras most commonly work by illuminating the scene to be recorded with a modulated laser light source. The phase shift between the illumination and the reflection can then be measured and translated to distance between the camera and the point of reflectance[11]. Microsofts second generation of Kinect, which was just recently released along with their newest console the XBox One[12], is in fact based off of time of flight technology using range gate imagers. In these devices there is a built in shutter in front of the image sensor that opens and closes at the same rate as the light pulses are sent out. Because part of every returning pulse is blocked by the shutter according to its time of arrival, the amount of light received relates to the distance the pulse has travelled[13]. Despite TOF technology being present in the second generation of Kinect, Microsoft has yet to release it for Windows, and as such cant really be considered for this dissertation. This leaves a lack of consumer grade cameras which will produce a depth image of sufficient resolution and sensitivity.

2.2 Scene Reconstruction & Camera Tracking

With a suitable input device selected, the question which naturally comes next is how to reconstruct the virtual scene from this real world input data. This process is simplified if the position of the camera is fixed and known, however this approach very much limits the level of detail you can acquire. Since you are unable to move the camera there are often holes left in the scene reconstruction where the camera is unable to see. A solution to this is to allow the camera to move and be able to scan the scene from many different angles. Alternatively multiple stationary cameras can be used to obtain the same multi-angle view. Neither of these approaches are new concepts and a large amount of previous research has been done on various ways of achieving a much more comprehensive view of the scene.

2.2.1 Multi-View Stereo (MVS)

Conceptually, the simplest approach to achieve a 3D scene reconstruction from 2D images is multi-view stereo. MVS generally requires a setup of multiple calibrated cameras and additional user input information about the geometric extent of the object or scene being reconstructed[14]. While there are many different approaches to fusing the number of 2D images together to create a single volumetric model, recent advances have been made particularly in the area of multi-view photometric stereo. In Vlasic et al.[1] they use a system of MVPS which combines earlier familiar techniques of silhouette recognition, and combine them with highly detailed normal maps obtained from an extremely elaborate setup of calibrated light sources and cameras. Surface reconstruction algorithms can then be used to process this data and reconstruct high quality, fully volumetric 3D models. The downfall of MVS is the amount of prior calibration and expensive equipment needed to get an adequate setup ready to capture. An example of this is the set up used in Vlasic et al. which is shown in **Figure** 2.2.

2.2.2 Simultaneous Localization and Mapping (SLAM)

SLAM is a broad concept within the robotics society which represents techniques used by robots and autonomous vehicles to build up a map of the environment it is currently in, while simultaneously keeping track of it own location within that environment. The complexity of the technical processes of both locating and mapping under conditions of errors and noise do not allow for a coherent solution of both tasks. SLAM is a concept that binds these processes in a loop and therefore supports the continuity of both aspects in separated processes. Iterative feedback from one process to the other enhances the results of both consecutive steps. SLAM, as a process itself, generally consists of multiple different parts: landmark extraction, data association, state estimation, state update and landmark update. Extensive research have been done on many different techniques for solving each stage.

The seminal work done on SLAM was Smith&Cheeseman[2] in 1986. Here they introduce the idea of approximate transformations (ATs), which consist of an estimated mean relation of one coordinate frame relative to another and a covariance matrix

that expresses the uncertainty of the estimate. In the context of a mobile robot this paper presents the concept of the robot moving from one position in world space to another as a single AT, so long as the kinematics of the robot are known. At the end point of the robots movement, its position within a certain error can be estimated by calculating that AT. As the robot then moves from point to point to point, more ATs are added to the first, with the error associated with the prediction of its new position rising with each new AT. Eventually it will get to a stage where the robots location with respect to the world frame becomes so uncertain that the robot is unlikely to succeed in actions such as going through a doorway, based purely on the information available to it so far. This is where the authors move on to the idea of robot sensing. A mobile robot equipped with sensors would allow it to determine the location of objects to an accuracy determined by the sensor resolution. These sense relationships could then also be represented as ATs in the AT network, along with those due to motion. It is important to note though, that if this kind of sensing is to work, these reference sensors must be established ahead of time, and the robot must always be in range of a reference sensor to receive any contribution to its location estimation. **Figure** 2.2 shows a diagram depicting the accumulating error of multiple ATs.

2.2.3 Structure from Motion (SfM)

All the earliest SLAM techniques developed required a large of amount of prior knowledge about the scene to be explored and mapped, such as sensor positions or the robot kinematics etc. One of the first techniques to attempt to reduce the amount of prior knowledge needed was the structure from motion technique. SfM hinges on the idea of detecting feature points within the scene, and tracking their trajectories over time, as the camera moves[15]. This, in theory, eliminates the need to know the exact kinematics of the robot and instead allows an estimation of location from just the various images alone. This is a step in the right direction however still requires prior knowledge of the scene. In the case of Dellaert et al.[15], the features chosen were hand picked. Also this method still leaves the problem of if the camera becomes orientated so that none or very few of the features chosen are visible, the system immediately falls down.

A class of techniques known as extensible tracking has since been developed to limit

these shortcomings, and further reduce the amount of prior knowledge needed. Extensible tracking in Park et al.[16] works by first calculating the cameras position in the world from a number of reference fiducials(visual points of reference used in camera calibration) within the scene. Natural features around the scene are then detected and calibrated using the same fiducials. Once a natural feature has been detected and calibrated, it can then be used as a point of reference itself to find further natural features as the camera moves around. This allows the user to move the camera so none of the original fiducials are visible and still be able to calculate its position in the world.

So now the the prior knowledge needed for potentially an entire scene has been reduced down to a very small set of fiducials. The logical next step in SfM is to try and eliminate the need for any reference map at all, and that is what they set out to do in Klein et al.[3]. Here they initialize the map purely through a small amount of user interaction, and application of a five point stereo algorithm. The user must first point the camera at a particular set of features to be tracked, press a button, smoothly translate the camera 10cm while maintaining vision on the same set of features, and then press another button. Within this step features are detected and tracked and the disparity between these original two keyframes can be used to calibrate the original set of feature points with regard to the camera. The authors then draw on the basis of the previously mentioned techniques to continue to detect and map further feature points within the scene to be mapped. With the aim of real-time results, the algorithm they developed does not use every frame from the video input. Instead they work off a series of keyframes which can be an arbitrary amount of time apart from one another. The feature points are tracked between keyframes by first searching for 50 of the coarsest-scale features in the new image, and the camera pose is roughly updated with regard to the translation found by this. After this original pose update, a further 1000 points are reprojected and searched for on the image. This allows a much finer and accurate final pose update to be calculated for the camera depending on the matches found. This paper was considered a large stride forward in the field of SLAM, however the application was focused more on scene and shape recognition, rather than reconstruction. As such they were only capable of constructing very sparse maps. **Figure** 2.2 displays an example set of points of interests detected by a SfM technique like this.

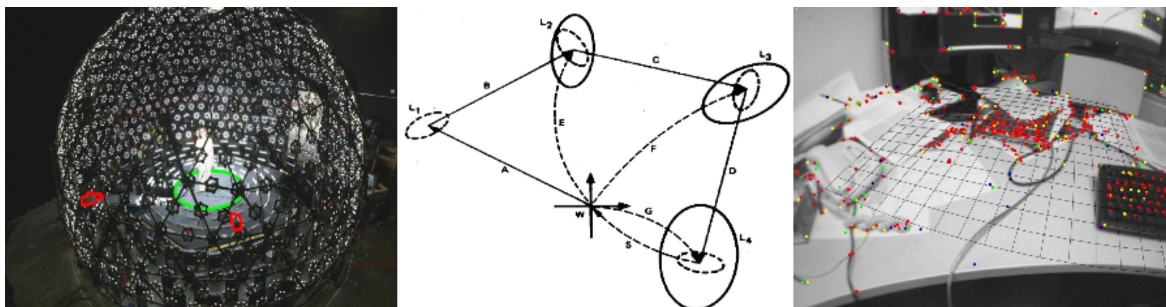

Figure 2.2: On the left, a sample MVS setup in [1]. In the centre, approximate translations leading to increased error in SLAM[2]. On the right, points of interest detected in a SfM approach[3].

2.2.4 Iterative Closest Point

Separate from the ideas of SfM but still a product of SLAM, ICP is another technique which has been adapted to track camera motion within an unknown scene. ICP originally found its use as a way of comparing an ideal 'model' shape and a detected 'data' shape to see if the two match up [17]. Since then though, it has been adapted to track the motion of the camera within a scene[18]. At its core, ICP is an algorithm employed to minimize the difference between two sets of 3D point data. Within the algorithm, one set of data is considered the reference and is kept fixed. The second set is then transformed iteratively using various translations and rotations until it matches the reference set as closely as possible. At the beginning of each simulation frame, each point in the second cloud is matched with the closest point to it in the reference cloud. The transformation need to best align each pair of points is found using a mean square error cost function. The second cloud is transformed by this calculated value and then this entire process is reiterated until the two converge within a certain error.

While it is easy to see why this kind of a method could be used to compare and match shapes, it is also important to note how it can be used to track the motion of a camera. If a transformation that should be executed on the second point cloud to match the two up can be calculated, then the transformation needed to be applied to the virtual camera so that the two appear on top of each other visually can also be calculated. ICP is an attractive solution because it allows the user to map completely arbitrary scenes with purely just the depth information so long as the scene holds a decent amount

of different geometry and is not just a wall, or similar flat surface. One shortcoming however, is since each pair of points is assigned depending on their closest neighbours in the other cloud, very fast or sudden motion will severely skew the results.

Another difficulty with regard to ICP is the handling of a phenomenon known as loop closure. This refers to the process that occurs when the camera is brought on a long sweep around a room which is executed over a larger period of time. The camera carries on being tracked around the scene and the newly scanned geometry is reconstructed within the application, however very small accumulations of error with the ICP can accumulate over time. This means that if the camera then returns to scan the area it began at, the algorithm will now think the camera is in a slightly different position and that all the point data is slightly off. This leads to the morphing and degradation of reconstructions of large areas over time. This phenomenon is mostly irrelevant when only scanning a relatively small area though.

2.2.5 RGB Plus Depth

Up until this point we have only really discusses methods which work solely on 3D point clouds retrieved from depth images. While these point clouds are extremely well suited for frame to frame alignment and for dense 3D reconstruction, they still ignore a large amount of valuable information contained in images. Colour cameras, on the other hand, capture rich visual information which can further be used in the principles of SLAM. RGB-D techniques are based around the combination of both the typical depth information and this usually absent RGB information. The Microsoft Kinect possesses both RGB and depth cameras within a single structure, and the data recorded by the two can be coordinated so long as the exact calibration between the two cameras is known.

In Henry et al.[19] the authors combine techniques of both ICP and feature detection to do their mapping, and they call it RGBD-ICP. In RGBD-ICP, every frame a sparse map of visual features is extracted from the RGB-D image. These features are associated with their corresponding depth values and typical ICP is performed on these points to find the transformation needed. This achieves the best of both worlds getting the increased accuracy from ICP and the more reliable feature detection from RGB-D.

Because the two methods are integrated and work so closely together in this approach, refining each other, this also allows for greater areas to be mapped without skewing the earlier readings with large amounts of accumulating error as the camera is brought on a long sweep, and returns back to its original position, the phenomenon also known as loop closure. An added bonus of the RGB-D approach as well, of course, is that colour can be preserved and present in the final reconstruction. The limitation of this RGB-D technique though is the fact that their RGBD-ICP method only runs at about one second per frame.

2.3 Volumetric Representation

By predicting the global pose of the camera, any depth measurement can be converted from image coordinates into a single consistent global coordinate space. How the data that now makes up this global coordinate space is stored is the next area of interest which must be explored. Primarily the information must be stored in a manner which can be used to visually reconstruct the surface in a later step. Although the raw depth data could just be roughly stitched together and rendered like that, this would leave a very coarse and jagged looking representation. The real challenge lies in storing the information in such a way that curved surfaces etc. can be represented smoothly and realistically. The data structure must also be added to, updated and referenced fast enough to maintain the real-time nature of the application. As such, speed is of paramount importance to the solution chosen. Space efficiency on the other hand, is not of as much importance for these types of applications as long as they are still fast. With that being said though, the lower the amount of memory the data occupies, the larger the scenes that can be represented.

2.3.1 Voxels

Voxels as a concept are somewhat analogous to a 3D pixel. A 3D scene can be split into a grid of equally sized volume elements, with each volume element (or voxel) storing a certain amount of information relevant to the grid position in 3D space it occupies.

Traditionally, voxels had found use solely in representing purely volumetric data, like for instance the kind of information returned by an MRI scan. Recently though, more and more uses are being found for them in other areas of computer science, such as using them as a densely sampled representation of opaque surfaces[20]. Voxels are ideal for use in the kind of fully volumetric representation we are trying to build up of the scene.

2.3.2 Signed Distance Function

In Curless&Levoy[21] the authors introduced the use of cumulative weighted signed distance functions (SDFs) to volumetrically represent the 3D surface of an object. The idea behind SDFs is that each node in the volumetric representation would contain a weighted distance function to the nearest surface. Nodes which lay on the outside of the surface would be given a positive distance, and nodes which lay on the inside would be given a negative distance. This would then allow rays to be casted from the camera or elsewhere, and wherever they ray transitioned from a positive node to a negative node, a surface was detected and could be further approximated using the relative weighted distances of the two crossover nodes. Curless&Levoy also observed that for these kind of applications, the SDF is only meaningful near the surface and that majority of voxels that are a certain distance from any surface can be completely ignored. Therefore they use a truncated SDF (TSDF) region in the vicinity of the observation, which only stores SDF values for voxels close enough to a surface to be relevant. TSDFs are ideal for volumetrically representing the kind of data obtained from the previously discussed imaging techniques and as such are used in nearly all of the state of the art approaches.

2.3.3 Uniform Spatial Subdivision

Both Curless&Levoy[21] and Izadi et al.[18] use the approach of uniform spatial subdivision of a voxel grid to hold their TSDF data. In Izadi et al. the 3D volume of fixed resolution is predefined in a position relative to the starting position of the camera. Once defined, this volume can not be moved and the full 3D voxel grid is allocated

on the GPU as aligned linear memory. They recognize that though this is far from the most space efficient approach (with a 512^3 grid of 32-bit voxels requiring 512MB of memory), they maintain that this does not really matter given that the approach is speed efficient. Of course though this will always limit the overall grid size their algorithm is able to represent, and it will also limit the resolution attainable.

In an effort to combat the limitation of having the scene locked to an initial position as above, Whelan et al.[22] developed a direct extension of Izadi et al. which they named Kinituous. Kintinuous works in a manner very similar to above in that they pre-define their first 3D voxelized volume as a fixed amount of space around the camera. However, when their camera is judged to have moved over a certain threshold of distance away from it original position they shift this volume along the transformation of the camera. This will lead to new empty nodes being made available to the TSDF with fresh input data, and will also lead to certain previous nodes of the TSDF being discarded. So as to preserve the data recorded in a previous TSDF, rather than just discard these nodes as they leave the area being scanned, the surface information is extracted from the voxels to be discarded and this data is streamed from the GPU memory and immediately fed into a greedy mesh triangulation algorithm. The results is a high polygon count surface representation of the previously mapped environment, which can be continuously added to as more of the scene is scanned in and processed by the TSDF.

2.3.4 Hierarchical Structure

In Izadi et al.[18] they discuss potential future work and suggest more memory efficient approaches such as octrees might be needed in order to be able to reconstruct larger scenes. Another direct extension of this paper, Zeng et al.[23] attempts exactly that. Though there are benefits of using an octree to reduce the amount of memory wasted on empty space, it is significantly more difficult to implement an octree based data structure that maintains the same requirements for speed as the simple memory inefficient approach. This is largely down to the fact that it is difficult to maintain the parallelism feature of the GPU due to the sparseness of an octree's nodes. In Zeng et al. they construct their octree structure using arrays of different layers, where one array corresponds to a layer in the octree. In their structure they use three different

types of layers, branch layers, middle layers and data layers. Branch layers hold just children IDs, middle layers hold both children IDs and the node's xyz key and data layers hold both the xyz key and the SDF value and weight. The authors also design new algorithms for the 'reconstruction update' and 'surface prediction' stages to efficiently utilize GPU parallelism and maintain speed. In their conclusion they report using about 10% of the memory [18] uses, and also report about a 2x speed-up on the stages they optimized.

Further work on hierarchical optimization has also been done in Chen et al.[4]. Here they introduce a method of using a shallow hierarchical structure which only goes about 3-4 layers deep as opposed to the 9-10 in the octree based approach. The authors of this paper name Zeng et al. as the work most similar to their own in the area, however they feel that the reliance on an octree imposes significant pointer overhead. In this paper they use regular spatial subdivision to structure their tree, however they exploit the sparseness of the scene to only densely represent areas near a surface, and use a much coarser representation for empty space. To represent their data they decided to use a technique of refinement. The root would consist of a coarse, fully allocated grid which spatially subdivides the entire area. If any of these coarse voxels is found to intersect with a predicted surface, that voxel is chosen for refinement. Refinement continues like this recursively down the tree until the leaf level is reached. Again the authors of this paper had to develop their own algorithm to support the efficient integration of SDF data and ray casting using this newly proposed data structure by exploiting the parallelism of the GPU. Lastly they also extend their algorithm to efficiently be able to stream data out of their voxel grid to support moving volume techniques such as Kintinuous mentioned above. The authors report even better results than the octree based approach and feel that the scalability of their approach provides a very attractive solution.

2.3.5 Spatial Hashing

The approaches of using a hierarchical structure have returned favourable results and effectively combat the massive memory overhead introduced by purely spatial approaches. However, it still remains that due to the fact it is inherently difficult to

parallelize hierarchical data structures, the full speed potential of an efficiently parallel algorithm is not being met. Nie et al.[24] propose to solve this problem using a technique of spatial hashing. They argue their method carries the benefits of other volumetric approaches, but does not require either a memory constraining voxel grid or the computational overheads of a hierarchical data structure. They aim to exploit the underlying sparsity in the TSDF representation by only storing voxels that contain TSDF related information in their sparse and efficient hash table based off of Teschner et al.[5]. In the paper they use the idea of voxel blocks. These voxel blocks are uniformly subdivided into the world and each contain a grid of 8^3 voxels. It is these blocks that are inserted into the hash table using the key of their x,y,z positions. The authors use a GPU accelerated hash table to manage allocation and retrieval of the blocks. In addition to storing a pointer to the voxel block the hash entry also contains its world position and an offset pointer to handle collisions efficiently. This system still efficiently supports the integration of TSDF data and surface extraction by ray casting. They also provide a method for streaming voxel data both from the GPU to the host and the host back to the GPU. This allows for a system like Kintinuous but also has the added functionality of being able to return and reintegrate earlier streamed parts of the scene with the extra host to GPU streaming capability. The paper reports large frame rate increases over Chen et al., which was said to be the current state of the art at the time.

2.4 Surface Extraction & Rendering

The final step in virtually representing the real world scene is to render the surface on screen. This uses the information gained from the camera tracking and combines this with the data structure created in the stage of volumetric representation. Again speed is of great importance in this stage. The rendering system implemented must also be capable of rendering fully virtual 3D meshes within the reconstructed scene also. This means correctly taking into account any occlusion between real and virtual objects.

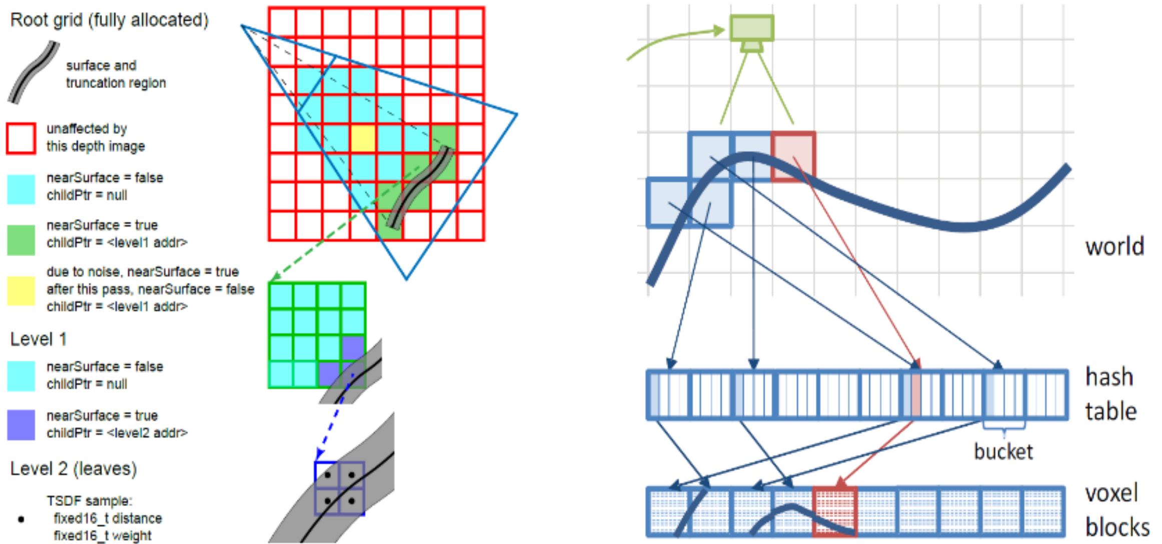

Figure 2.3: On the left, visual representation of the shallow hierarchy method in Chen et al.[4]. On the right, visual representation of the spatial hashing method used by Teschner et al.[5].

2.4.1 Raycasting

Since basically all of the current research in this area is using some form of SDFs to represent their data volumetrically, using a method of raycasting to extract and render the surface is by far the most popular approach. The raycasting works by initiating a ray starting at the virtual cameras position and giving it a certain direction within the current view frustum. This ray will then traverse through whatever data structure is chosen to volumetrically represent the TSDF of the scene. Once the ray reaches a point of zero crossing between adjacent nodes a surface is detected and its exact position can be found using the relative distance functions. Assuming the gradient is orthogonal to the surface interface, the surface normal can be computed directly as the derivative of the TSDF at the zero crossing. Therefore each ray cast can calculate a single interpolated vertex and normal, which can be used to render the surface.

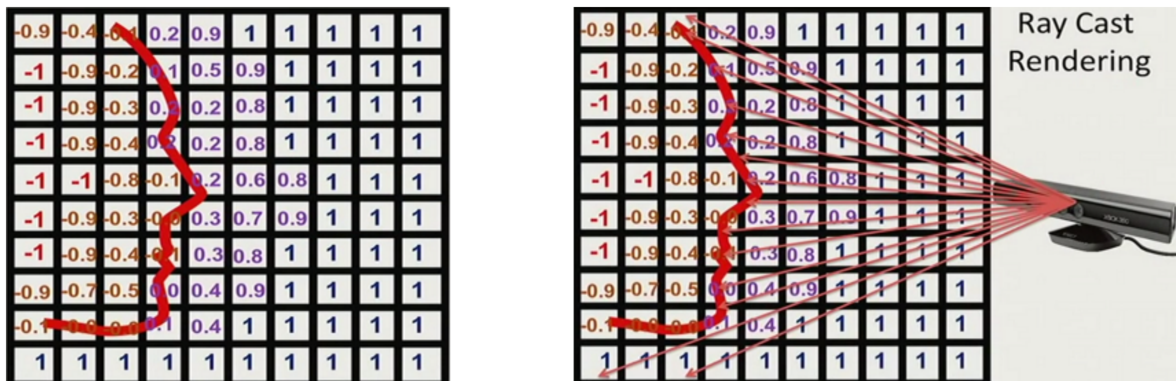

Figure 2.4: On the left, a vertical slice of the volumetrically represented SDF. On the right, ray casting to approximate the surface.

2.4.2 Compositing of Real and Virtual

This method of raycasting provides an effective approach at rendering the virtually reconstructed imitation of the real world scene. However it does not implicitly provide a method for rendering additional purely virtual objects within this reconstruction. In Izadi et al.[18] they introduce a rendering pipeline which allows conventional polygon-based graphics to be composited on the raycasted view. Their method also supports correct occlusion handling. Their pipeline works by first rendering the virtual scene with graphics parameters identical to the parameters being used in the raycast. The vertex buffer, surface normals and colour maps from this are stored in off screen maps and used as input during the ray casting. For each raycast, a distance from the associated mesh vertex from the virtual scene to the camera is first calculated. This is then used as a termination condition when stepping along each ray. If the ray finds a real surface while under this distance the raycast surface is drawn, while if the length of the ray ever exceeds the original virtual vertex distance calculated, the ray is discarded and the virtual object is drawn.

2.5 Kinect Fusion

The Kinect Fusion technique [18][25], is widely considered the basis for the current state of the art in the area of scene reconstruction using a consumer grade camera.

In these papers the authors describe how they developed a GPU based pipeline for reconstructing scene geometry at interactive rates. They use a Kinect camera as their method of input, perform camera tracking using ICP and represent the information volumetrically using a TSDF in a spatially subdivided grid. All of these are concepts mentioned previously. Kinect Fusion is a Microsoft published pipeline and there is a library available for executing some of the techniques mentioned in the papers on MSDN, using the Kinect for Windows SDK[26]. Although the main purpose of Kinect Fusion is more for the detailed reconstruction of surfaces and 3D models, they do also demonstrate some of the physical capabilities of the pipeline. This is demonstrated by performing simple virtual particle collisions with their 3D reconstruction.

2.6 Extraction of Physically Relevant Information

With the real scene fully reconstructed and suitably rendered in the virtual world, the next step is to consider how to represent the scene physically. There are many different types of input data that can be used to calculate rigid body physics within the scene, however this is not necessarily the same kind of information already extracted to render it. The method chosen must yield data that is suitable for calculating the physical interactions between real and virtual objects.

2.6.1 Particles

The easiest approach to this problem is to only use a system of particle physics and work on extracting the relevant particle information from the scene. This fits nicely with the different volumetric representations developed because the various leaf voxels found to be intersecting with a surface can just be used to create a particle of size equal to the voxel itself. This somewhat crude approach provides a very grainy interpretation of surfaces, but for basic simulations of rigid body physics it works visually fine. Since the fixed scene geometry is represented entirely by this particle information, any virtual objects cast into it must also be represented by similar particle data. An advantage of a particle based representation is it is very simple to update the scene in real-time

even if the geometry should ever change.

2.6.2 Meshes

Another type of scene representation suitable for calculation of rigid body physics is as a mesh. In the previously mention paper Kintinuous [22], they provided a method for streaming the volumetric information back to main memory where it can be used into a 3D mesh. While the motives of their research was not to extract this for physically related use, but rather larger scene construction, this method could still be used to generate a mesh for arbitrary purposes. The major disadvantage of this method though is that once the volumetric information has been extracted to a mesh, it is quite difficult to make any dynamic changes to that mesh. This, of course, is not a problem though if you are only concerned with static scenes.

2.6.3 Primitive Shapes

Since we will be dealing with a scene representation that cannot be moved by the physical events triggered in the simulation, it is in theory possible to use a much coarser medium to represent the surface geometry. An ideal medium for this coarse representation would be primitive shapes such as planes, spheres, cylinders etc. Primitive shapes suit well for this kind of application because, while still being able to fit the scanned data quite closely, they would significantly reduce the amount of points that need to be tracked. Primitive shapes also perform much more efficiently in rigid body physics calculations when compared to vast amounts of particle data or relatively fine meshes. The technique of Random Sample Consensus (RANSAC) is an iterative method which estimates parameters of a mathematical model from a set of observed data which contains outliers. In Schnabel et al.[6] they present an efficient RANSAC algorithm for detecting basic shapes in point cloud data. They use RANSAC to extract shapes by randomly drawing minimal sets from the point cloud data and construct corresponding shape primitives. A minimal set, in this case, is the smallest number of points required to uniquely define a given type of geometric primitive. The resulting candidate shapes are tested against all points in the data to determine how many of the points are well

approximated by that primitive, with each being given a score. After a given number of trials, the shape which approximates the most points is extracted and the algorithm continues on the remaining data. This method was developed with the aim of being able to visually represent semi-complex scenes with a reduced data set but this could definitely be extended to our motives. This method was not really developed with interactive rates in mind but then again their focus was on representing complex scenes, perhaps performance could be vastly increased using relatively simple configurations on a work bench.

Figure 2.5: A 2,000 element point cloud (left) represented as 372 basic shapes (right) using [6].

2.6.4 Signed Distance Functions

It would also be possible to just integrate the existing signed distance function with virtual objects and use that to calculate rigid body collisions. In Guendelman et al.[27] they represent geometry using both a triangulated surface mesh and a SDF defined on a grid. They argue this approach has many advantages and use it to simulate interactions between nonconvex rigid bodies. Because the SDF of the scene geometry is already known from the scene reconstruction we would not really need to extract it. The only thing left to do would be to calculate an SDF for any virtual objects that might be thrown into the scene.

2.7 Physics Calculations

With all the physically relevant information extracted the next task would be to decide what methods to use to calculate the various physical interactions. The physical solver eventually chosen depends a lot on the kind of information extracted form the scene as discussed above. Another extremely important factor is that the solver must be quick enough to support being executed at the same time as the scene reconstruction, while maintaining interactive frame rates. Luckily there is an extremely large amount of techniques and algorithms developed in this area which we can draw from.

2.7.1 Physics Engines

Of course the simplest approach would be to just use a pre-existing physics engine. There a few highly optimized and largely robust physics engines such as Bullet[28] or Havok[29] readily available for download and use. The potential difficulty would lie in streaming the relevant information in a suitable format to the engine for use. The kind of scene representation suitable for input into an engine like this would typically be as either a mesh or as an assortment of 3D primitive collision shapes. Mesh generation in every frame can be quite expensive when included with all the other aspects though.

2.7.2 Particle Systems

Extensive research has been carried out in the field of simulating physically based interactions through the use of a particle based representation. Many different particle based approaches are available which use various techniques to simulate the different mediums of rigid bodies, cloth and fluid. The only paper previously discussed in this chapter which simulates physical interactions within their work is Kinect Fusion[18]. As discussed above, they wished to demonstrate the physical capabilities of their implementation without going into too much detail on the topic as a whole. To do this they chose to expose the scene to a system of separate virtual particles and simulate simple collision response using a method outlined in the book GPU Gems 3[30][31]. In GPU Gems they use a discrete element method to simulate their collision response

with a repulsive force being modelled like a linear spring, and a damping force modelled by a dashpot, which dissipates energy between particles. Although they do not go any further than this simple implementation in the Kinect Fusion paper, GPU Gems also describes how this method can be extended to simulate more advanced systems such as rigid bodies, cloth or fluid. Although this implementation provides a suitable framework for the intentions of this dissertation, further work has since been carried out on the area and new state of the art for particle based solvers now exists.

2.7.3 Position Based Dynamics (PBD)

Traditionally the approach taken to simulate the behaviour of dynamic objects has been to work with forces. Internal and external forces are accumulated at the beginning of each time step, and Newton's second law of motion can be used to transform these forces to accelerations. A time integration scheme can then be used to first compute velocities from these accelerations, and then positions from the velocities. Alternatively impulses, which directly alter the velocity of the particular object, can be used to control it, meaning one level of integration can be skipped. In applications like games though, it is often desirable to have direct control over the positions of objects or vertices of a mesh. PBD is a method which affords this control, where the simulation acts directly on the positions[32]. Position based approaches make it possible to control the integration directly, thereby avoiding the overshooting and energy gain problems associated with explicit integration. PBD works by solving different systems of constraints which represent physical interaction. Constraints are conditions which must be satisfied by the solver. So for instance if we take the example of two equally sized spheres which rigidly collide with one another, the constraint that must be satisfied is that the centre positions of the two spheres are always at least a distance of two radii apart. If they were found to be closer than this then a collision response must be calculated and the positions of the two spheres are altered until the constraint is satisfied and they are no longer colliding. Different types of constraints can also be specified for various different types of interaction. Because the PBD approach works solely by altering position, velocity of objects must be inferred from the total change in position from the beginning to the end of a single simulation loop.

2.7.4 Collision Detection & Response

Collision detection and response is one of the most important stages in any physics simulation. Because in particle based systems you are usually dealing with many thousands of particles, distance checks between these particles could potentially become an enormous bottleneck. "CUDA Particles"[33] is a highly efficient method of finding neighbouring particles. It works by spatially subdividing space into a 3D grid and placing particles into specific cell positions within this grid using their position in x,y,z coordinates. A hash value unique to the particular discrete cell position the particle is currently in is calculated and this hash is used to store each particle in their correct cell in a manner which allows you to easily read any cell's contents on demand. Once every particle has been placed in a cell, each particle can then be iterated through and distance checked with all the particles found in the surrounding 26 grid positions, as well as the cell it is currently occupying itself. If any overlaps are found, a neighbour for that particular particle is stored. Depending on the type of medium the two neighbouring particles represent, they could either be designated as collision neighbours, or in the case of a fluid-fluid pairing, they will be used in the fluid neighbour constraint solver. This method is very popular due to the fact that it is highly parallelizable and ideal for use with solvers that carry out a lot of their implementation on the GPU. The actual response triggered by finding collision neighbours with this method is calculated by using the collision constraint described in the paragraph previous.

2.7.5 Rigid Bodies

Although rigid bodies are typically associated with being a single rigid object, it is also possible to represent them as a system of independent particles bound together by a system of constraints[34]. The constraints which govern the motion of these rigid bodies can be calculated by finding a certain translational vector and a certain rotation matrix. As each object is generated, the positional offset of each particle that makes up the object with regard to the its overall centre of mass must be recorded. As the simulation progresses and the object is subject to various collisions etc, the shape of the object is maintained using the knowledge of the offset of each particle at rest. The translational vector basically equates to the difference in the centre of

mass of all the particles at the very beginning, and the current centre of mass. And an ideal rotation matrix is calculated for the object which allows us to constrain the various particles to the shape's current orientation. Originally this paper was written as a method of representing deformable objects. This can be achieved by only applying loose constraints to each particle so that they try to retain their original shape, however deform elastically. This method can easily be adapted to simulate fully rigid bodies though if stiff constraints are used.

2.7.6 Cloth

Traditionally cloth simulations are simulated using a system of springs which replicate the bending and stretching of cloth in real life. In the same paper which for the first time defines PBD as a complete framework [32], they also demonstrate the capabilities of the system using an implementation of cloth. In their approach they replace the traditional spring-like interactions with constraints which imitate these interactions. They treat their cloth as a triangle mesh, with each shared vertex being represented as a single particle. This means that if there are two connected triangles at, for instance, the corner of the cloth, the six vertices associated with the two triangles are actually represented physically as four particles. The stretching constraints are calculated using the distance between particles along the three edges of each triangle. If the distance between a pair of particles along an edge exceeds a certain amount the two particles are drawn together. The bending constraint is calculated by taking pairs of adjacent triangles and comparing the normals between the two. If the triangles are not parallel, the constraint will push them back towards facing in a similar direction.

To produce more realistic interaction, aerodynamics can also be applied to simulate the cloth being somewhat dragged by a certain density of air. In [35] they simulate the effects of aerodynamics by approximating each triangle as a thin airfoil. Lift and drag forces can then be distributed to the cloth's particles. The drag force always acts in a direction opposing the motion of each airfoil and has an impact on velocity depending on the normal of the airfoil. If the face of the triangle is perpendicular to the direction of motion a greater drag force is experienced. The lift force is calculated in a similar manner, but is different in that the lift force always acts in a direction that

is perpendicular to the motion of the airfoil.

2.7.7 Fluid

Early particle based simulations of fluid depended on techniques such as smoothed particle hydrodynamics(SPH) [36]. The motion of particles governed by SPH is defined by a number of smoothing kernels. Different kernels are used to represent the different forces present in a fluid simulation. Kernels work by extending out from each particle a certain distance, if a neighbouring particle is found to fall within the interaction radius, each kernel will apply a weighted force to the particles in question depending on how close they are within the smoothing radius, and depending what force it represents. For instance, if the force brought about by fluctuations in density is being calculated, a nice smooth kernel will be used to gradually ease each particle within system back to be at rest density. These types of kernel produce gradual interactions with no hard limits. However if you wanted to simulate the effects of a force like pressure, spiky kernels are used which give a very firm push apart when particles get too close together. Since the original work done on SPH based fluid simulations a more recent particle based approach has been developed which uses the framework of position based dynamics, "Position Based Fluids"(PBF)[37]. In this approach they still use the idea of various smoothing kernels but no longer use forces and instead calculate constraints to replace the forces. The main governing constraint in PBF is the density constraint. Again a gradual kernel is used to calculate the density of the fluid at each particle location. Once all these densities have been calculated the constraint is formed using the rationale that the current density should ideally be equal to the rest density. If the current density is lower then that particle draws in all neighbouring particles, while if the current density is higher than rest then it pushes away all neighbours. Beside density, PBF also calculates values for vorticity, viscosity and surface tension. Vorticity is a force which acts on particles and is used to replace some of the energy lost through some undesirable damping caused by the PBF technique. Viscosity works by calculating the velocity of a certain particle relative to all the particles surrounding it. Viscosity then serves to act against this relative motion and instead cause the motion of the fluid as a whole to act more coherent.

Although the PBF paper includes a term for surface tension, more recent techniques have achieved better results. Akinci et al.[38] calculates both a term for cohesion between particles and a surface minimization term. The cohesion term is calculated using the distance between neighbouring particles and spline function which acts somewhat like a kernel. If the distance between two particles is below a certain threshold they will repulse each other, and if it is above that threshold but still within the interaction radius, they will attract each other. The surface minimization term is calculated by calculating the gradient of the smoothed colour field at each particle. This is essentially influenced by where a particles neighbours lie around it. If they are all to a single side of the particle this will cause a large gradient, whereas if the particle is nicely surround the gradient will go to zero. The force caused by this term between two neighbouring particles is weighted by the difference in the gradient between the two. Towards the centre of a fluid this term will go to zero, while on the outer edge it will cause more smooth and rounded edges.

2.7.8 Unified Particle Physics

Unified Particle Physics for Real-Time Applications [39] is a paper published by Müller et al. in 2014 which is considered to be the current state of the art in particle based solvers. This paper contributes a unified dynamics framework which combines many of the previous works mentioned in this chapter which use position based dynamics to simulate their physical phenomena. Müller et al. provide a novel simulation loop which can simultaneously simulate rigid bodies, cloth, fluid and gases, and allows for two-way interactions between all of these simulations. Every object within the solver must be composed of equally sized particles, and the various sets of constraints which act on each particle are calculated and solved in a Jacobi fashion. Jacobi iteration works by taking the positions of all particles relevant into a certain constraint solve, and only using this original view of what the particle positions are to calculate the change in position for each particle involved. This is different to Gauss-Seidel iteration, where each particle position is updated as soon as the position change is calculated, and this now modified position is used in future calculations in that particular constraint solve.

Chapter 3

Design

The aim of this chapter is to provide a high level overview of the design process. Key design challenges encountered and decisions made will be highlighted and discussed. Also provided will be an overview of the various programming tools and assets used throughout development.

3.1 Key Design Decisions

From the very beginning of the project it was clear that a few key design decisions must be made with regard to the high level workings of the eventual application, before an implementation can be attempted. These key decisions are:

- What method of input to use
- How to represent the real world information in a manner suitable for simulating physical interaction
- What kind of solver to use to handle the physical interactions
- How to keep this all within the bounds of real-time
- How to represent the interactions and objects visually

3.2 Method of Input

Of course when setting out with the goal of capturing a real scene and having the aim of integrating it with certain virtual aspects, the first question which must be asked is how you wish to record this real world data. Among the many factors that would influence a decision like this, three of the most important factors for an application like this are image quality, how the input data is received, and finally accessibility. Image quality is of particular importance to this paper. In some applications, such as those used for skeletal recognition etc., sufficient information can be inferred from noisy point cloud data. However in order to obtain sufficient information for detailed physical interaction to be possible, a higher level of surface geometry must be obtained. For this reason the input device chosen must be capable of capturing enough detail in the scene to make a detailed reconstruction possible. Also important is the type of data received from a particular input device. If the goals of real-time are to be met, the input data stream must balance on the fine line of enough information to recreate a sufficiently accurate scene, while not overwhelming the application with impractically large or cumbersome data sets. Finally is the issue of accessibility. There are any amount of highly expensive cameras or elaborate multi-camera setups that would be suited to a virtual reconstruction of a real world scene. However, these types of devices or setups are inherently inaccessible and awkward to work with when compared to the more affordable and portable solutions available today.

Because the goal is to design a system that is capable of working in a variety of different scenes without the need for visual prompts such as augmented reality tags or fiducial markers, a depth sensing camera is ideal for scanning in this type of information. As far as what depth sensing camera to use, the Microsoft Kinect was chosen. The Kinect is cheap, widely available and, for the purposes of this dissertation, has performance comparable to even the most expensive of depth cameras. Microsoft also provide a number of useful libraries for processing information scanned in by the Kinect in their 'Kinect for Windows' SDK[40].

3.3 Representing the Real World Information

Traditional depth images are often quite noisy and contain a lot of holes where no depth information has been obtained. Also, once the various pixels which contain depth information have been mapped to a point cloud in 3D space, what you actually get is a very sparse representation of the scene as a whole, with only information corresponding to the surfaces closest to the camera available. Detailed physical interaction requires comprehensive knowledge of the scene and if only the information obtained from the raw depth image was used, virtual objects would constantly be falling down through the holes in the depth information or catching on the sharp edges of the front surface of the scanned in real object. Also if noisy data is present where, for instance, a pixel has no value for depth in a particular frame but then does the following frame, the constant appearance and disappearance of particles which are supposed to cause collisions can lead to instability and an undesirable addition of energy to the system. In order to be able to simulate as accurate physical interaction as possible, a representation of the scene beyond just the front-most surface must be formed. Also desirable is the ability to smooth out the noisy data to a much more stable view of what is present in the scene.

Kinect Fusion, as described in the state of the art, provides an ideal solution for overcoming these two major hurdles. Camera tracking allows us to gain information from various angles and fuse point clouds together to get a much more complete 3D view of the scene. Also as the information is fused it is integrated together to give a much smoother representation of surfaces within the scene over time. This eliminates the problem with noise. Although the state of the art also detailed some improvements on Kinect Fusion's specific implementation which provide better results in both memory efficiency and performance, the fact that Kinect Fusion is a publicly available library was too attractive to ignore for the purposes of this time constrained project.

3.4 Physical Solver

With the method of representing the real world information now known, the next step is to decide on what type of physical solver to use to handle the interactions. Of course the solver chosen must be able to handle the kind of information being exported from the Kinect, so the eventual choice of solver will largely be influenced by the type of information we can conveniently return from Kinect Fusion. While Kinect Fusion is capable of exporting a mesh, this process can actually be quite performance intensive. The volumetric representation of Kinect fusion is stored in a uniform 3D grid that fills the entire space being scanned in. As previously mentioned, discrete voxel representations suit the idea of using particles because any voxel found to be inside a surface can easily be treated as an immovable particle in that discrete position, with diameter equal to the length of the voxel. Also important to keep in mind when choosing a solver is it must also be efficient enough to run real time along with camera tracking and point cloud integration. Particle based solvers are highly parallelizable and it is often possible to carry out most of the calculations on the GPU. For these two main reasons a particle based solver was chosen to be used in this implementation.

The specific particle based solver chosen was the Unified Particle Physics solver described in the state of the art[39]. Collision detection would be handled by CUDA Particles[33] and the different systems of constraints needed to simulate rigid bodies, cloth and fluid would be calculated using the various papers united under the position based dynamics framework[32][34][37]. The order and manner in which all these different steps are executed will be governed by the unified solver's novel simulation loop.

3.5 Keeping It All Real-Time

With the general plan for the structure of the implementation now known, the impact on performance of all these various steps next had to be considered. Kinect Fusion already runs almost entirely on the GPU so that was not so much of a concern. Going forward the plan was to implement all the papers associated with the physical simula-

tion of objects serially on the CPU. This was partly because I had never worked with CUDA before and as such was not hugely comfortable with it. Partly because I didn't know yet if a serial implementation would be too demanding on performance. And partly because I figured that even if I found a GPU based implementation was fully necessary, it would be easier to port a version I had previously got working in serial than diving straight in the deep end and trying to get it to work for the first time in CUDA.

After successfully implementing all these simulations in serial it did become apparent that in order to support larger particle systems for more interesting interactions it would be necessary to port all the implementations to CUDA. This was also necessary in order to be able to execute the physics simulations at the same time as the camera tracking and volumetric integration from the Kinect Fusion stage. Luckily particle based systems are highly parallelizable and the papers do give some guidelines with regards to efficiently calculating the simulations on the GPU.

3.6 Rendering

Although rendering was not originally suppose to be a focus of this dissertation at all, further into the development I realised that I would need to pay at least some attention to the area of giving both the virtual and the real greater visual fidelity. What was implemented so far was just both the real scene and the virtual objects rendered as equally sized particles. If the rendering was left like this it would make it very difficult to tell if the interactions between the two are actually accurate because it is hard to tell exactly what is going on. Also rendering the scene just as particles makes the representation lose any familiarity to it actually being a real scene. This is undesirable because it makes everything just look virtual and no longer seems to be working on the seamless integration of real and virtual.

The first matter was to work on restoring the familiarity of the real scene. The obvious solution to this problem was to just overlay the current image being read in by the RGB component of the Kinect camera on top of the physical details being scanned in. This has the advantage of keeping very strong ties to the real scene but unfortunately

it requires using the rather limited view port of the current Kinect field of view. An alternative solution would be to use Kinect Fusion's export mesh feature which allows us to build a 3D mesh of the scanned in scene which can then be rendered with full colour. The advantage of this approach is it allows us a much more flexible view port and the freedom to move the virtual camera anywhere within the scene and look at it from any different angle. Unfortunately though this method does lose some of the familiarity to it being a real scene and can make the scene look quite static. In the end it was decided to use implementations of both. Both techniques have advantages and disadvantages in showing off particular features of the implementation so being able to compare the two would be a valuable addition.

The final issue is how to render the virtual objects. Although making the virtual and the real appear visually like they belong together was well outside the scope of this project, it was decided that each of rigid bodies, cloth and fluid had to look more like their actual respective visuals, rather than just a system of particles. This means proper rendering for each and the ability to show some simple lighting. While it is possible to generate visuals for rigid bodies and cloth quite easily by using the layout of the particles to calculate triangles and in turn normals, fluid rendering is quite a bit more difficult and an implementation of screen space fluid rendering [41] was needed. A big part of making a fluid look like it belongs in a scene is handling some sort of reflection and refraction. A detailed description of how these problems were tackled is available in the implementation.

3.7 Languages, Tools & Libraries

The application was built using the following combination of languages, tools and libraries:

- C++
- OpenGL
- GLSL
- CUDA

- Thrust
- Kinect Fusion
- GLM
- Eigen

Chapter 4

Implementation

This chapter will discuss the specific details of the implementation used in this dissertation. The implementation itself is broadly separated into three sections: scanning in the scene using the Kinect, performing the physics calculations using the particle based solver, and finally rendering everything. In each section the step by step implementation of the major features will be described. Aside from this some of the problems encountered that hadn't necessarily been considered during the general design phase will be highlighted and the steps take to solve these problems will be discussed.

4.1 The Kinect

4.1.1 Scanning in the Real Scene Information

The first step of every frame is to attempt to scan in the depth information from the Kinect. The Kinect only captures at 30fps so to avoid limiting the possible frame rate of the program to the same cap as the Kinect, a call is made to the depth stream of the camera which checks if a new frame is waiting to be processed. If there is a new frame present, the raw depth image is copied from the depth stream to a pixel buffer to be used later in the process. If the check returns that there is no new frame waiting however, this copy is skipped and no new depth information is passed into the scene

reconstruction. Similarly, the colour image must then also be scanned in as well. Again the program enquires if there is a new colour frame waiting in the colour stream of the Kinect, and if not the colour integration is skipped.

When new images for both depth and colour have been received, the goal is to combine the two together to provide both positional and colour information of each pixel in the respective images. Unfortunately though the Kinect camera has a gap of a few centimetres between the depth camera and the RGB camera. This means that if one image is laid directly on top of the other, they will not match up perfectly. Because of this the pixels of the colour image must first be mapped accordingly onto the depth image. This ensures the correct colour gets associated with the correct depth value.

4.1.2 Kinect Fusion

With the two input images now appropriately mapped onto one another, this information is now suitable to be passed into Kinect Fusion. Kinect Fusion is responsible for taking this information, performing camera tracking using the depth images, updating the camera position based of the tracking, and finally integrating both the depth and colour information into the existing volumetric representation it has constructed. During the initialization of Kinect Fusion it is possible to specify certain parameters such as the size of the area you wish to reconstruct and the resolution of voxels within that area. It is also possible to specify the weight Kinect Fusion has towards either favouring its existing knowledge of what the scene should look like, or favouring what the most recent frames say it should look like. If a heavy weight towards its pre-existing knowledge of the scene is chosen the representation constructed will be very smooth and noise free, however the program will be unable to handle any kind of movement because it will just assume the newest frame is wrong. If a heavy weight towards the most recent frame is chosen, however, movement can easily be detected and accounted for but the overall reconstruction will be much noisier and it becomes much more difficult to build up a stable and comprehensive view of the scene. It is quite difficult to find a balance between the two which allows for both movement and detailed scene representation. As such it was decided to just settle for a mostly static scene and bias fusion towards the smoother, more stable representation.

4.1.3 Extracting the Physical Information

With the scene information being successfully scanned in and integrated, the last stage is to extract the physically relevant information from the Kinect Fusion volumetric representation. This can be achieved by raycasting into the voxel grid and returning points where a surface is hit. This raycast can have a certain stride assigned to it which allows the user to make a point cloud less dense than the actual volumetric representation if such a high resolution of surface data is not needed. The positions of these surface points are then copied into the physical solver for use in the physical simulation.

4.2 The Physics

4.2.1 The Particle

The particle is the fundamental building block of absolutely everything involved in the physics simulation in this dissertation. As such how much information each particle possesses and how this information is stored is an extremely important factor in the possible capabilities and limitations experienced later down the road. In this implementation each particle is responsible for storing a number of key values:

- Position - The position of the particle at the very beginning of the frame
- New Position - The position of the particle after it has been subject to the various constraints
- Delta Position - Because of the Jacobi style of iteration constraints update the delta. Then every 'New Position' is updated at once
- Velocity - The velocity of the particle
- Mass - Its mass
- Phase ID - An ID which specifies whether the particle is part of a rigid body, cloth or fluid etc.

Because we wish to carry out all the simulations on the GPU all this information must be stored in GPU memory. Originally structs which contained each of the components previously listed were being stored. However as I became more accustomed to using CUDA I learned that memory coherency when making calculations can be a major limiting factor for the speed CUDA can achieve. By storing an array of structs like this, a calculation which requires taking the positions of two particles into account is forced to skip along vast chunks of memory. For example, even if an interaction between two particles that are stored adjacent in memory is being calculated, it is still necessary to traverse a step in memory equal to the full size of the particle struct to reference the positions of both particles. This problem is only magnified during an interaction between particles that are thousands away from each other. An alternative approach to this is to use a struct of arrays, rather than the previous array of structs. This essentially means that there would be a single struct which contained information for every particle in the system. This struct would store a number of arrays which held all the information each particle would possess individually. Now each particle just corresponds to a certain index into these arrays, and interactions can be calculated much faster now that similar values are stored in consecutive memory. A calculation which requires referencing two adjacent particle positions now requires a step equal to one 3D vector, as opposed to the capacity of the entire struct.

As described there is now a single struct which contains all the particles in the entire system. This might sound bizarre because there is no real separation between particles that are part of a rigid body cube or particles that are part of a fluid etc. But this is simply the way that unified particle solver works. As far as each particle is concerned it is completely independent from every other particle in the system and is only responsible for holding its own values. The thing that binds particular particles together and make them behave like part of a bigger object are the different sets of constraints which act on them. Each constraint knows the IDs of the particles within the global data structure that it has influence over.

Particle diameter is also quite an important factor that impacts the simulation greatly. The reason behind this is quite closely tied to timestep and will be discussed in section 4.2.3.

4.2.2 Simulation Loop

I will now briefly describe a typical simulation loop executed by the solver each frame, and later go into greater detail on each of the main steps.

Algorithm 1 Simulation Loop

```
1:  read in the array of scene particles
2:  clear spatial hash grid
3:  for all virtual particles do
4:      apply external forces
5:      predict position
6:  end for
7:  for all real and virtual particles do
8:      insert into hash grid
9:  end for
10: for all virtual particles do
11:     find neighbours from hash grid
12:     add neighbours to either collision neighbours or fluid neighbours
13: end for
14: while iter <solverIterations do
15:     for all virtual particles do
16:         solve collision constraints
17:     end for
18:     for each constraint group do
19:         solve all constraints in group
20:     end for
21: end while
22: for all virtual particles do
23:     update velocity
24:     apply internal forces such as aerodynamics and viscosity etc.
25:     update final position
26: end for
```

Each update loop begins by reading in the positions of the scene particle scanned in by the Kinect. The spatial hash grid is cleared as it still contains data from the previous frame. The position of each particle is projected forward using the velocity calculated from the previous frame combined with the application of external forces forces such as gravity. Both scene and virtual particles are placed in their appropriate positions in the spatial hash grid. Neighbouring particles are found using this hash grid and

they are either stored as collision neighbours or fluid neighbours. An iteration loop is started and in each loop it solves the collision constraints between the various collision neighbours, and then solves the respective constraints needed to simulate rigid bodies, cloth and fluid. Because these steps are the most performance intensive to calculate, the iteration count is kept relatively low at a value of two suggested by Müller et al.[39]. The constraint systems themselves have been designed to operate well under low iteration counts like this. After the while loop the velocity of each particle can be calculated by dividing the distance between its 'new' position and its position at the beginning of the frame by the time step. Velocity modifying forces such as cloth aerodynamics or fluid viscosity, vorticity and surface tension are applied. And lastly the actual position of the particle is updated to its calculated new position. This is only a very high level view of the simulation loop described and we will now go into greater detail on the more complex aspects of the loop.

4.2.3 Timestep

In an application which is aimed to be run at 60fps the average timestep to be processed each from would be about 16milliseconds. Obviously the aim for a real time application would be to get the timestep used in the physics simulation as close to 16ms as possible in order to have the behaviour as realistic as possible. There are a few important factors to consider when setting timestep, aside from just this though. Tunnelling is a widely known problem in physics simulations where discrete collision detection is used, and tunnelling is an even bigger problem in particle based simulations. An example of tunnelling is say a dynamic particle of diameter 0.01metres is directly above a static particle with the same diameter. Currently the distance between them is 0.015m and they are not colliding. If the dynamic particle is falling at a speed of 2m/s, using a timestep of 0.016s, the particle's position will be projected forward 0.032metres. The dynamic particle will now be a distance of 0.017m below the static particle, and a collision will never have been detected. At the beginning of every simulation loop in this physical solver there is a position projection which acts like this. Therefore timestep is somewhat limited by the particle diameter chosen. The particle diameter we found best suited this implementation for various reasons such a stability, integrating with

Kinect Fusion and rendering was 0.04metres. The timestep found to be most stable with this particle size was 0.01s.

4.2.4 Neighbour Detection

Collision detection is quite often going to be one of the stages that is the most important and demanding in terms of performance in any physical solver and this one is no different. The implementation in this dissertation all hinges around a spatial hash grid. This hash grid divides a certain set 3D volume of space into cells which have a length equal to the particle diameter. The purpose of this is that only particles found in adjacent cells have any chance of being overlapped. The hash function used for the grid corresponds to the particular cell's index along the x, y and z axis. So if a 10x10x10 grid is defined, the hash for each cell would be

$$hashvalue = xID \times (yID \times 10) \times (zID \times 10 \times 10) \tag{4.1}$$

The single ID for a particular cell is equal to this hash value. Which cell ID a particular particle lies in can easily be determined if the position of the minimum of the volume designated by the grid in world space is known. In the example above the minimum would most likely by (-5.0, -5.0, -5.0) if the cells are set to have a width of 1.0.

Once the method of hashing is known, the next step is to consider how the information is going to be stored and accessed. Each cell must be capable of storing an amount of indexes to particles which are found to be inside that cell. Because the volume of the grid does not change throughout the course of the simulation a large block of memory can be assigned once during the initialization and then just modified frame by frame. Because we are assigning a single block at the very beginning we must assign a maximum amount each cell can contain. If the maximum is set to 12 then the cell with ID 0 'owns' the slots available between position 0 and 11 in this single block, just like cell ID 5 corresponds to positions 60-71. Therefore if you wish to write an index into a particular cell you simply multiply the cellID by the maximum number of particles allowed in each and add however many particles are already stored in that particular cell to find the correct position in the overall block.

$$blockPosition = (cellID \times maxPerCell) + numInCell[cellID] \tag{4.2}$$

This may seem like a very memory inefficient approach but for this implementation we are not worried about memory efficiency as we are not ever memory constrained. Rather we are just concerned with speed and this approach gives a huge speed increase over other implementations, especially in a CUDA based environment.

The next challenge to deal with is how to do this in parallel. As mentioned above the number of particles already stored in a particular cell must be known in order to place the particle index in the correct block position. To achieve this a counter must be incremented each time a particle is stored in a cell and this counter must be stored itself. Incrementing this counter in serial is not a problem and will always work as expected but when you are doing it in parallel errors will occur. This is because if two particles are trying to insert themselves into the same cell, there is a possibility that they could each read the value in the counter at the exact same time before either has had a chance to increment, write their particle into that slot and then increment the counter. This means that one particle will essentially overwrite the other and data will be lost. Simultaneous memory accesses are major problem in parallel computing. To avoid this happening an atomic add is used. Atomic add is a function in CUDA which checks to see if a particular memory address is currently being accessed anywhere else in the program. If it finds that the address is not currently being accessed then it can perform operations such as incrementing the value at that address, while if it detects that there is already somewhere else accessing it, the function will just wait for the other access to be completed before proceeding. Atomic adds are generally frowned upon in parallel computing because if you are trying to write to the same address with atomic adds rather frequently, the waiting essentially brings the program back towards being serial. For the purpose of this dissertation though it is very rare that the atomic add will ever have to wait. This is due to the fact the cells are of equal size to particles and collision response is turned on so you will rarely ever find two particles or more in a single cell.

Once every particle has successfully been placed into the relevant cell, the next step is to detect actual overlaps. This is done by distance checking the current particle

in question with the particles found in the 26 surrounding cell positions, along with the other particles found in the cell it is currently occupying itself. If an overlap is detected each of the two particles will be added to the other's list of collision neighbours. Although if the two particles are detected to be fluid particles, they are instead added to each other's list of fluid neighbours, for use in the fluid constraining process.

4.2.5 Collision Response

With all collision neighbours now detected and known, the first constraint to solve is the collision constraints brought about by these collisions. Collision response constraints for particles simply dictate that no two particles can be overlapping. To solve this constraint we simply just project the positions of the pair of particles away from each other until they no longer overlap. The delta position for one of the two particles is written as

$$delta = \frac{1}{2} \times (dist - particleDiameter) \times \frac{p1 - p2}{dist} \tag{4.3}$$

where p1 and p2 are the positions of the respective particles and dist is the distance between them. It is important to note however that scene particles are treated as immovable so the fraction which divides the displacement by two must removed where the virtual particle is colliding with a scene particle.

4.2.6 Rigid Bodies

As mentioned in the design the paper which we chose to base this implementation off of is [34]. When a rigid body is initialised in this implementation, the centre of mass of the shape is set to be the origin, and the positional offsets of each particle used to represent the shape are recorded. The constraint which governs the motion of these particles later in the simulation is a product of a translation vector and a rotation matrix. These two components are forever trying to maintain the shape of the object as specified in the initialisation step. The translation vector can be calculated by translating the original centre of mass to the current centre of mass. Since the shape

was initialised around the origin anyway the translation simply becomes the position of the current centre of mass. All particles within the object are set to have equal mass so to calculate the centre of mass the simple equation below is used, where n is the number of particles.

$$currentCentre = \frac{\sum_{i}^{n} position_i}{n} \tag{4.4}$$

Calculating the rotational matrix is slightly more complicated. The eventual matrix calculated is based off the difference between the original offset of each particle from its original centre of mass, labelled $\mathbf{q}_i$ and the current offset for each particle from the object's current centre of mass, labelled $\mathbf{p}_i$. A symmetric matrix $\mathbf{A}_{pq}$ is calculated where $\mathbf{A}_{pq} = \sum_i \mathbf{p}_i \mathbf{q}_i^T$. The actual rotation matrix, $\mathbf{R}$, can now be calculated by

$$R = \frac{A_{pq}}{\sqrt{A_{pq}^T A_{pq}}} \tag{4.5}$$

The Eigen mathematics library[42] was used to handle the square root of the denominator matrix. With the translation and rotation components now known, the final delta for each particle can now be calculated.

$$delta_i = (R \times q_i) + currentCentre - position_i \tag{4.6}$$

When we first looked at adapting this to perform in parallel there didn't appear to be a huge amount which could be parallelized because most of the calculations are solved for the rigid body as a whole, rather than just for each particle individually. One calculation it was possible to make a CUDA kernel for is the calculation of the $\mathbf{A}_{pq}$ matrix for each particle, however a lot of the work was still being done on the CPU. Another thing we noticed which could be improved is both the summing of positions(with the goal of getting the centre of mass) and the summing of the individual $\mathbf{A}_{pq}$ matrices calculated for each particle(with the goal of calculating the rotation matrix). Summing in serial works by adding each element consecutively. If a system with 8 elements is

being summed, the first element is added to the second, the third is added to this result, then the fourth and so on. In CUDA there is a concept of 'reductions' which can be used as a more efficient method of achieving this same goal. In a reduction built for the purpose of summing these 8 elements together, 4 simultaneous threads would be executed which added the first element to the second, the third to the fourth and so on until there are now 4 values to be summed rather than 8. In the next stage of the reduction the consecutive pairs would again be added together giving 2 values. These final 2 values can then be added together to give the total sum of the original 8 values. The parallel nature of a reduction means that whereas the original CPU based version would take time equivalent to 7 serial additions, the GPU only takes a time equal to 3 serial additions. The time saved by using reductions is only increased when considering objects which consist of large numbers of particles and also considering the fact they will be used to sum 3D vectors and 3x3 matrices. Thrust[43] is a CUDA based library which has functions to perform certain reductions. Thrust's addition reduction was used to handle these sums in this implementation.

4.2.7 Cloth

The paper we chose to base this implementation of cloth off was the seminal paper in position based dynamics[32]. When this implementation of cloth is initialized the layout of the particles which make up the cloth are structured in such away that is simple to determine the index of the particles directly adjacent to each individual particle in all directions. Once the neighbours can be easily referenced the various constraints can now be executed. The main constraint is the stretching constraint which gives the cloth its bouncy and stretchy behaviour. The particle based representation of the cloth must first be mapped onto a triangle based representation, where 4 adjacent particles in a square could be thought of as 2 connected triangles. The stretching constraint acts along the edges of the the fully mapped triangle representation and consists of a simple spring-like distance constraint. This constraint serves to pull the two particles along the edge in question back towards each other if a certain distance threshold between them is exceeded. Where $\mathbf{p}_1$ and $\mathbf{p}_2$ are the positions of the two particles and $\mathbf{l}_0$ is the original length of the edge, the constraint is calculated as

$$C_{stretch}(p_1, p_2) = |p_1 - p_2| - l_0 \tag{4.7}$$

The actual delta position can be calculated by inserting this constraint into the constraint projection algorithm outlined in the paper which returns

$$\Delta p_1 = -\frac{1}{2}(|p_1 - p_2| - d)\frac{p_1 - p_2}{|p_1 - p_2|} \tag{4.8}$$

where d is equal to the original distance between the two particles along the edge. If we consider that the 4 particles $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ and $\mathbf{p}_4$ represent 2 connected triangles, with $\mathbf{p}_2$ and $\mathbf{p}_3$ being the shared edge. The stretching constraint applies interactions between every particle pairing except between the only 2 particles which don't share an edge, $\mathbf{p}_1$ and $\mathbf{p}_4$. This final interaction is left for the bending constraint to account for. The bending constraint can be calculated by using the equation

$$C_bend(p_1, p_2, p_3, p_4) = acos(\frac{(p_2 - p_1) \times (p_3 - p_1)}{|(p_2 - p_1) \times (p_3 - p_1)|}.\frac{(p_2 - p_1) \times (p_4 - p_1)}{|(p_2 - p_1) \times (p_4 - p_1)|}) - \phi_0 \tag{4.9}$$

where ϕ_0 is the initial dihedral angle between the two triangles. This constraint can then also be inserted into the same constraint projection algorithm which leads to a much more complicated method of finding the delta than it was for the stretching constraint. The full process is listed in the appendix of PBD[32].

Handling self collisions is often another difficult task which must be considered and solved in traditional cloth simulations. While they do suggest a method of handling self collisions in the PBD paper, in the case of this dissertation it is not necessary to deal with these collisions within the cloth calculations. Instead they can simply be dealt with by the unified solver. The particles which make up the cloth can be treated just like any other collision particle in the global system. This means that, as long as the cloth in question has a dense enough representation of particles, self collisions will automatically be handled by the unified solver.

To add further realism to the cloth the effects of aerodynamics were included in the

simulation as well using the method outlined in Keckeisen et al.[35]. As described in the state of the art there are two main forces applied to simulate aerodynamics, the drag force and the lift force. The drag force opposes the motion of each face and points in the opposite direction of the current velocity of the face. The velocity of the face, $\mathbf{v}_i$, is calculated by averaging the velocity of the three associated particles, and the normal, $\mathbf{n}_i$, of the face is calculated using cross products and the positions of the three particles. The drag force applied can then be calculated as

$$F_{drag} = \frac{1}{2} C_D \rho |v_i|^2 A.(n_i.v_i).(-v_i) \tag{4.10}$$

where C_D is the specific air resistance coefficient, ρ is the density of air, and A is the area of the corresponding face.

The lift force is calculated using similar factors however it is always directed perpendicular to the motion of the face. The direction the lift force acts in can be calculated by $\mathbf{u}_i = (\mathbf{n}_i \times \mathbf{v}_i) \times \mathbf{v}_i$. The lift force can then be calculated as

$$F_{lift} = \frac{1}{2} C_L \rho |v_i|^2 A cos\theta.u_i \tag{4.11}$$

where C_L is the specific lift force coefficient.

Conversely to rigid bodies it was possible to approach the calculations necessary for each particle completely independently of any others particles. This made it possible to parallelize pretty much every calculation performed in the cloth simulation and perform them all on the GPU.

4.2.8 Fluid

The paper we chose to base this fluid implementation off of was Position Based Fluids[37]. PBF can be thought of as an extension to the smoothed particle hydrodynamics based approach[36] which instead fits into the position based dynamics framework. The main constraint which governs the motion of this fluid simulation is the density constraint.

$$C_i = \frac{\rho_i}{\rho_0} - 1 \tag{4.12}$$

This constraint essentially means that if the density at a particular particle i is either higher or lower then the rest density of the liquid, then neighbouring particles must either be pushed away or pulled in accordingly. If the rest density is met, the constraint goes to zero. The density for a particular particle can be calculated by using a gradual smoothing kernel, the concept behind which has been explained in section 2.7.7. The gradual smoothing kernel, $W(\mathbf{p}_i - \mathbf{p}_j, h)$, for a particle of position $\mathbf{p}_i$, and its neighbour of position $\mathbf{p}_j$, gives a weighted contribution for density from each neighbour of the particle currently in question. This weight comes from the distance between the two particles with regard to their interaction radius h. The particle accumulates these contributions and the final sum is the particle's current density.

$$\rho_i = \sum_j m_j W(p_i - p_j, h) \tag{4.13}$$

With the density now known and the constraint calculated, the next step is to try use this constraint to find a suitable particle position correction. First a scaling factor, λ_i, must be calculated for each particle. To do that the gradient of the constraint function must be calculated. While a gradual kernel is used for density calculation, a spiky kernel, $\nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$, is used to calculate the gradient.

$$\nabla_{p_k} C_i = \frac{1}{\rho_0} \sum_j \nabla_{p_k} W(p_i - p_j, h) \tag{4.14}$$

where k is both the particle question i, and all its neighbours j. The scaling factor for each particle can now be calculated by

$$\lambda_i = -\frac{C_i}{\sum_k |\nabla_{p_k} C_i|^2 + \varepsilon} \tag{4.15}$$

where ε is a small relaxation constant. The final position update can now be calculated for every particle using

$$\Delta p_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(p_i - p_j, h) \tag{4.16}$$

This is definitely one of the most complicated parts of the implementation as it involves a lot of long calculations and complicated mathematical syntax. Because of this it is quite difficult to describe in a manner that can be easily understood. The algorithm below visualizes a typical fluid constraint solver loop.

Algorithm 2 Fluid Simulation

1: **for each** fluid particle i with neighbours j **do**
2: calculate density at that particle
3: use this density to find the constraint
4: find the gradient of the constraint function at i
5: **for each** neighbouring particle j **do**
6: find the gradient of the constraint function at j
7: **end for**
8: accumulate all these gradients
9: calculate λ_i
10: **end for**
11: **for each** fluid particle i **do**
12: use previously calculated values for λ to calculate Δp_i
13: **end for**

This delta position will give the fluid general fluid-like behaviour, however it will still very visibly lack the kind of cohesion we associate with fluid, and instead it will look more just like a system of interacting particles. Features which must be added to give the system its more fluid-like behaviour are vorticity confinement, viscosity and surface tension. All these effects act directly on the velocity of the particles rather than the positions so they are not included in the constraint solving phase but rather closer to the end of the global simulation loop, during the velocity modifying stage. Vorticity confinement is introduced to return some of the energy that is inadvertently lost during position based methods of simulating. To calculate vorticity confinement we must first calculate the current vorticity at the particle using

$$\omega_i = \sum_j v_{ij} \times \nabla_{p_j} W(p_i - p_j, h) \tag{4.17}$$

Once we have the vorticity we can now calculate a corrective force which can be used to alter the velocity of the particle.

$$\mathbf{f}_i^{vorticity} = \varepsilon(\frac{\nabla|\omega|_i}{|\nabla|\omega|_i|} \times \omega_i) \tag{4.18}$$

Viscosity makes a fluid flow more cohesively by reducing relative velocities between neighbouring particles. The velocity correction due to viscosity can be calculated quite easily using the equation below, where the parameter c is some small viscosity constant.

$$v_i^{new} = v_i + c\sum_j v_{ij}.W(p_i - p_j, h) \tag{4.19}$$

The last velocity modifying effect to apply is surface tension. Although the PBF provides behaviour similar to surface tension through a term dubbed 'tensile instability' in their paper, there has since been a more accurate technique been developed by Akinci et al.[38]. In this paper they achieve their surface tension effect by generating two forces, one to simulate molecular cohesion, and one to simulate surface minimization. The cohesion term can be calculated by

$$\mathbf{f}_i^{cohesion} = -\gamma\sum_j C(r)\frac{p_i - p_j}{|p_i - p_j|} \tag{4.20}$$

where γ is a cohesion constant and *C(r)* is a spline function which pushes particles away if they are too close but pulls them closer together if they are towards the outside of each others' interaction radii. The last thing to calculate is now the surface minimization term. First the gradient of the smoothed colour field for each particle must be found. This essentially detects if there is any imbalance in the distribution of particles directly surrounding the particle in question. Towards the centre of a volume of fluid it can be reasonably assumed that each particle will be uniformly surrounded. However towards the edge of the volume some particles will have particles on one side and air on another. This is where we want our surface minimization force to act. The smoothed colour field can be calculated by

$$n_i = h \sum_j \frac{mass_j}{\rho_j} \nabla W(p_i - p_j, h) \tag{4.21}$$

and the actual force can be calculated by

$$\mathbf{f}_i^{surface} = -\gamma mass_i \sum_j (n_i - n_j) \tag{4.22}$$

Similarly to the vorticity force, these two forces can then be used to calculate a direct modification to the particle's current velocity.

Similarly to cloth, pretty much every calculation in this implementation fluid simulation is isolated per particle. Because of this it was possible to execute every stage in parallel on the GPU.

4.3 Rendering

As stated in the design in section 3.6, at this stage in the implementation it became clear that some form of rendering would have to be carried out on both the real and virtual objects. Accuracy of interaction became hard to judge and the connection between real and virtual became blurred because the whole scene just looked virtual. This section will describe the various rendering techniques used to highlight the physical results.

4.3.1 Real World - RGB Image

The most obvious option was to just overlay the RGB image over the virtual representation of the real scene. In order to be able to align the two properly, this meant that the Kinect's own field of view would have to be used. To achieve this the window size was set to be 640x480 to match the resolution of the Kinect image and we varied the virtual camera's field of view until an angle was found which best lined up the two. This angle was found to be 48°. Of course because the RGB image is coming from the point of view of the Kinect as well, the virtual camera's point of view must also

be matched to this. Kinect Fusion's camera tracking returns a view matrix for where it thinks the real Kinect camera is with regard to the world. This can then be copied straight over to the virtual camera's view matrix.

Although this on its own served the purpose of restoring familiarity to the scene, it was still very difficult to judge interaction and everything looked rather unnatural because no occlusion had been factored into this. What we are essentially dealing with is a flat texture rendered to the screen with no concept of 3D depth within the world. On its own the texture will appear entirely at a single depth, whether it be always at the very front, always at the very back or elsewhere. To gain some idea of the depth for each RGB pixel the depth image was read in simultaneous and the depth pixels again mapped to the colour image. Rather than use the mapped images to integrate into the volumetric representation, this depth is instead passed into the fragment shader along with the RGB information. Here the depth is linearised with respect to the far clipping plane of the virtual camera and the depth of the fragment is manually assigned.

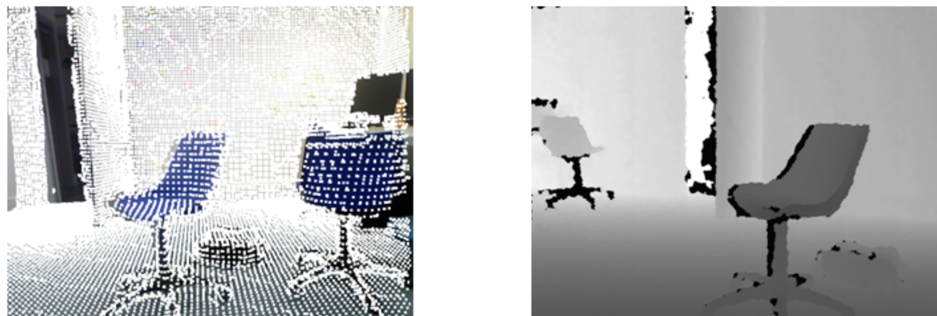

Figure 4.1: On the left is the virtual representation of the scene overlaid with the RGB image. On the right is an example of the depth image used for occlusion

4.3.2 Real World - 3D Mesh

Another option to render the real world information was to use Kinect Fusion's built in export mesh function. This function raycasts into the voxel block and returns a mesh of the surface scanned in which holds values for both position, colour and normal. This information can easily be then passed into shaders and rendered accordingly. Although the normals are provided, no lighting was performed on the mesh. This is because the

colour value read in already has some representation of lighting from the scene, so any more virtual lighting just looks unnatural. It is worth noting though that these normals could be useful in a completely uniform lighting environment, or the information could also be used to create other visual effects such as a toon effect or similar. An advantage of this representation is the virtual camera can be moved around anywhere in the scene by the user, there is no longer any need to match it to the Kinect's PoV. Occlusion also is not an issue using the mesh representation because it is fully 3D information.

Figure 4.2: Scene rendered as a 3D mesh

4.3.3 Rigid Bodies

Basic rendering of the cubes used to demonstrate rigid body interactions was rather trivial. Knowledge of the four particles which correspond to the four corner points of the cube can be used to construct faces. The four particles can also be used to calculate normals for each face. Basic lighting is then carried out in the shader.

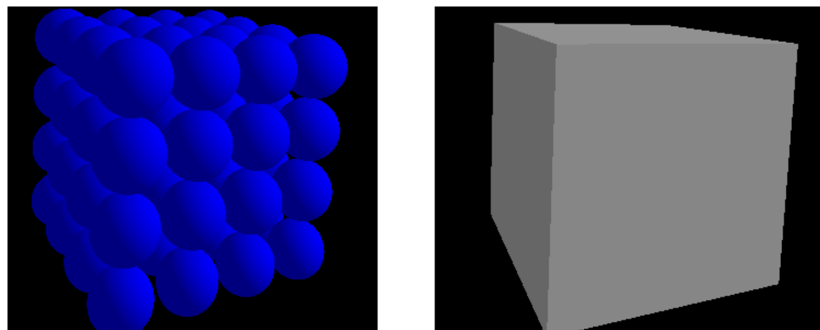

Figure 4.3: Cube visualized just as particles and with final render

4.3.4 Cloth

The implementation used to render cloth was rather similar to the method employed for rigid bodies. If you assume that the cloth has x amount of particles along the x axis and y amount along the y axis, it is possible to use these two numbers to create a network of triangles using the various particle indices. Each frame the positional information for the vertex of each triangle is extracted. If we consider $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ to be the three vertices of a particular triangle, the normal, $\mathbf{n}$ of that triangle can then be calculated by

$$n = normalize((p2 - p1) \times (p3 - p1)) \tag{4.23}$$

Because a single particle can actually be responsible for multiple vertices, all attached to different triangles, if each triangle just had its normal calculated and used like this the visual division between triangles would be extremely apparent. To make the cloth look smooth, the multiple normals associated with each particle must be accumulated and normalized, and this value must then be used for all vertices connected to that particle. This successfully gave the cloth smooth, cloth-like visuals.

Unfortunately though, since only normals facing in one direction were calculated, one side of the cloth will appear nicely lit and rendered correctly, whereas the other one will just look flat and dark. To remedy this, what is essentially just a second cloth is rendered which matches the first in every respect only it faces the other direction. After implementing this though visual artefacts kept appearing due to two things being rendered in exactly the same place, which caused a certain amount of clipping problems. To stop this clipping occurring, rather than render the two clothes using the exact same position(corresponding to the exact centre of all the particles), the vertex positions were instead projected a small amount out from the particle centres along their normals in either direction. This is essentially leaving a small gap between the two cloth renderings which are supposed to represent a single piece, however the gap is not big enough to be detected visually and it does succeed in stopping the artefacts.

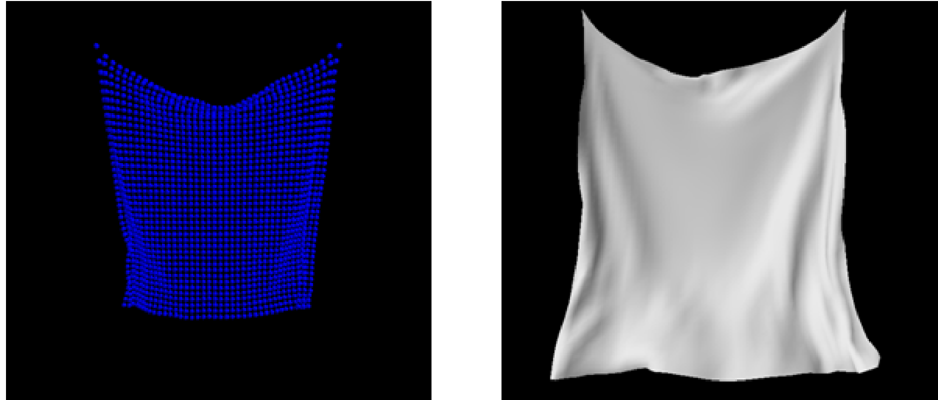

Figure 4.4: Cloth visualized just as particles and with final render

4.3.5 Fluid

Unlike rigid bodies and cloth, rendering fluid can not be achieved based only off of existing knowledge of particle configuration. Instead a rather complex fluid rendering technique named screen space fluid rendering[41] was used. SSFR is a particle based fluid rendering technique which carries out nearly all of its work in screen space. This means that rather than trying to use fully 3D methods, the final result is instead a composite of images generated from the current 2D view of the particle simulation currently on screen. To achieve the final render, a number of steps must first be implemented which will be explained in detail throughout this section.

Before going into depth on the implementation though, it would be worth explaining some terms which will be brought up throughout. The first is a fragment, a fragment can be thought of as the data necessary to shade a single pixel. The data which makes up a fragment is calculated on the GPU in a fragment shader. The next is a frame buffer, a frame buffer can be thought of as a buffer of memory which images can be rendered to, much like how an image is rendered to the screen. The difference between rendering to a frame buffer and rendering to the screen though is the frame buffer is kept off screen and not displayed. Instead, 2D textures can be made from these images rendered to the off screen buffer. Frame buffers are useful in this context because they allow us to render multiple different interpretations of the current scene which can then be written to textures and combined to make a final image, but only the final image is actually rendered to the screen.

Point Sprites

The only information SSFR needs from the actual fluid simulation is just the centre points of all the fluid particles per frame. To turn this scattering of point-like positions in space into a unified representation, the first thing which must be carried out is to approximate some sort of sphere like representation. Instead of doing the costly calculation of trying to approximate actual spherical geometry around every point, the particles are instead represented as quads which always face directly towards the camera and have a width equal to a fluid particle diameter. Once this quad has been passed into the fragment shader, if the distance of the fragment is found to be a distance further than the radius of a particle from the centre of the quad, then the pixel is discarded. This turns the square, screen aligned quad into a round, screen aligned circle. But at this point the circle is still essentially flat so still does not represent any 3D geometry. This geometry can be faked within the same shader by projecting the depth forward depending on where the fragment lies within the circle. Fragments at the very edge of the circle's radius will have a depth equal to whatever the depth of the centre point is. Fragments towards the very middle however will be an additional radius closer to the screen than the centre point, and of course fragments in between can be interpolated. This method is very cheap and succeeds in giving each fluid particle faked spherical geometry, rendered only in 'screen space' because it is essentially 2D. The method of using point sprites to fake 3D geometry can simply be used on its own to render large systems of particle very cheaply with simple lighting. This is, in fact, the method used to render the entire system as particles when that was the only rendering being carried out in this dissertation. In this fluid renderer however, this technique is used to to create a depth map and later to approximate thickness.

Depth Map

As described above the technique of screen facing point sprites can be used to give each particle a spherical appearance with calculated depth. This, in turn, can be used to make a depth map of the current view of the particle system. To make a depth map the artificial depth values for each pixel are all rendered to a floating point texture on the frame buffer.

Bilateral Blur

Because we want the system of particles to look less like a set of individual spheres and more like a single cohesive fluid, applying some kind of blur or splatting or similar is usually necessary. In this case a bilateral blur is performed on the depth map. A bilateral blur works by taking an image(like our depth map), passing it into the blur shader and blurring each pixel of the image a set amount of pixels either side along the x axis. This x blurred view is now rendered to a separate texture on the framebuffer, where it is then passed back into the blur shader and blurred a set amount of pixels either side along the y axis. The final image rendered from this gives a nice smooth version of the depth map where the individual particles are no longer visible. How many pixels you blur along each axis is one of the biggest factors in the smoothness of the final render you achieve. The best number to choose depends both on the particle size and distance of the fluid from the camera. Large blur radii are very performance intensive though and the blur shader will often be the most demanding part of the renderer.

Normals

Now we have a nice smooth depth map, the next step is to calculate a screen space normal map. This can be achieved by passing this blurred depth image into a normals shader and applying an algorithm within that shader which can use the depth of a particular pixel and its x and y coordinates in the screen space image to calculate its 3D coordinates in eye space. The shader then calculates the gradient in eye space position between the pixel in question and the pixels above, below, to the left and to the right of it. This gradient is then the normal. This normal map can then be rendered to an RGB texture but it is worth noting that the texture is incapable of storing negative numbers. Because usually the individual values in normals span from -1.0 to 1.0, this will cause errors and instead the normals must be translated to fit between 0.0 and 1.0.

Thickness

The thickness of a fluid determines the amount of colour attenuated through that fluid and it also impacts the amount of refraction. For this reason we must also build a thickness map. The thickness shader is quite similar to the depth shader in that it uses point sprites to approximate geometry. In the same way the depth shader can fake depth, the thickness shader can fake thickness. An overall thickness map is built up by turning on additive blending and disabling depth testing. This means that every particle in the fluid will be drawn, even if it is hidden behind other particles and all of them will be drawn additively on top of each other. Thickness can be inferred from how coloured a pixel is by the end of the pass. This thickness map is then blurred using the bilateral technique again.

Colour Attenuation

To give a fluid a deeper blue look when looking through a thick chunk of water and a semi transparent look when looking towards the edge of the water, colour attenuation must be factored in. Beer's Law is used to model this attenuation.

Reflection and Refraction

One of the biggest factors in making a fluid look like it actually belongs in a scene is having it accurately reflect and refract its surroundings. Typically, cheap reflection and refraction is carried out using a cube map. A cube map can be considered as a giant cube which is floating around the scene you wish to render, but doesn't necessarily show up on screen. Reflection/refraction at a certain point in the world can be simulated by mapping the visuals to a certain point on the cube map. To carry out some kind of reflection/refraction in this application, some method of creating a cube map from the currently available representation of the real scene must be designed. For the 3D mesh representation of the world a cube map is made by putting a separate virtual camera directly in the centre of the 3D scene being read in. This separate camera takes six snap shots along the different axes and then maps the images to the six faces of the cube map. In the representation where the RGB image is overlaid this method

does not work as we only ever can get one point of view of the scene. Because of this reflection was not really an option but refraction was still possible. Rather than have the cube map stay static in the world it was set to follow around the orientation of the camera. This meant that the current view from the Kinect could be mapped to the negative Z side of the cube and this will give pretty adequate refraction.

Achieving the Final Render

With all these textures with various amounts of screen space information now generated, the last step was to pass them all into one final shader and composite the various images to create the final image.

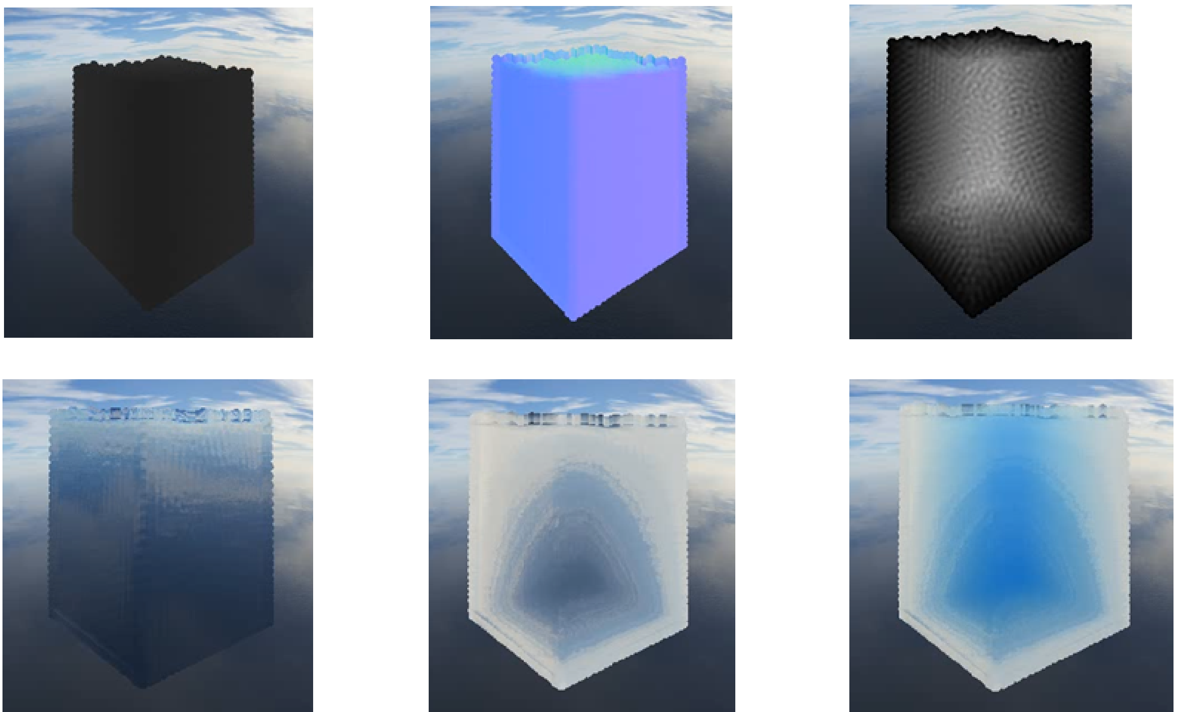

Figure 4.5: The various stages of the fluid renderer. Images listed in order depict blurred depth, normals, unblurred thickness, reflection, refraction and the final composite

Chapter 5

Results & Evaluation

The results which best indicate the degree of success of this implementation can be divided into two categories, quality of interaction and performance. The quality of the interaction is difficult to quantify as the purely visual feedback is somewhat open to interpretation. The performance however provides slightly more tangible results. This chapter will present the results obtained from the implementation described in the previous chapter, provide a discussion on their relative success and also highlight some of their shortcomings. Towards the end of this chapter I will also provide an evaluation of the implementation as a whole.

5.1 Quality of Interaction

Of course the original intention of the dissertation was to design a system capable of carrying out as close to realistic interactions as possible between real and virtual. As such the believability and apparent accuracy of these interactions were always going to be the main metric of the success achieved within this area. Although it is difficult translate these purely visual and dynamic results to paper, screenshots which portray some of the interactions achieved will be presented below. And the link to the actual videos will be provided in the appendix along with digital copies provided on the attached CD.

5.1.1 Fluid

Figure 5.1 and **Figure** 5.2 are taken from two of the videos which show fluid interacting with a scene. As is visibly apparent, the behaviour of the fluid and how it collides with the real scene is actually quite believable and realistic. Fluid particles collide with scene geometry accurately, accumulate in dips and corners correctly, and flow along flat geometry such as the the floor like you would expect. Throughout the simulation the fluid is suitably cohesive and generally behaves quite similarly to how you would expect a real fluid to behave. One disadvantage though is since it is not possible to gain any sort of material information about the scene, the scene is basically treated like glass during the interaction. No feature like soakage or anything like that is possible within the scope of this simulation so the interaction of the fluid with the scene is purely governed by rigid collision and essentially no friction, much like how a real fluid would interact with glass.

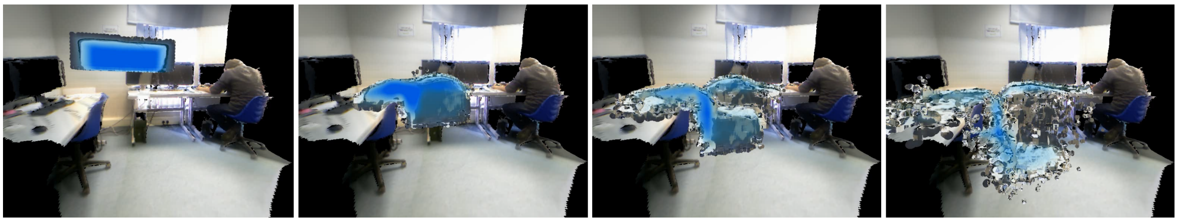

Figure 5.1: Fluid being dropped on a desk

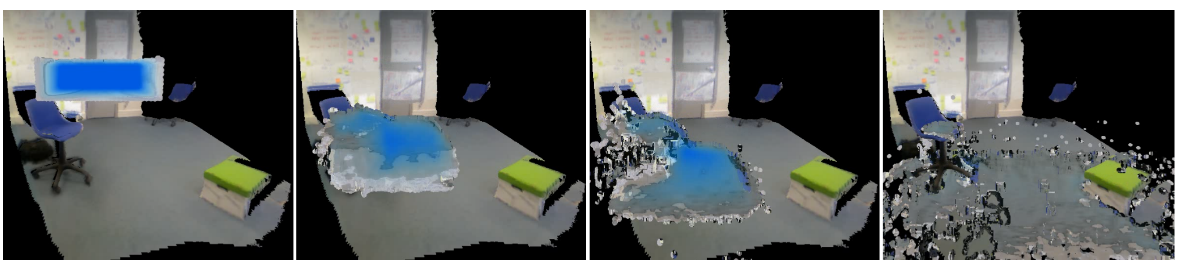

Figure 5.2: Fluid being dropped on a chair

5.1.2 Cloth

Figure 5.3 and **Figure** 5.4 are taken from two of the videos which show cloth interacting with a scene. Again in these examples we can see that the cloth behaves quite believably when exposed to the scene. It collides correctly with the different scene geometry, accurately bends and contours around curved surfaces, and it maintains structure and avoids clipping through itself due to self-collisions being dealt with adequately. One disadvantage of the cloth simulation though is there is currently very limited friction. Due to the 'bumpy' nature of particle based collisions, a certain amount of friction is always present just because the representation of surfaces is already relatively coarse. Unfortunately for cloth though this does not provide sufficient friction and the cloth tends to slide around the scene a little too much. This is not as apparent when cloth is just being dropped on some scene geometry, however becomes more apparent when the cloth falls to rest on the floor but instead continues to slide.

Figure 5.3: Cloth being dropped on a desk

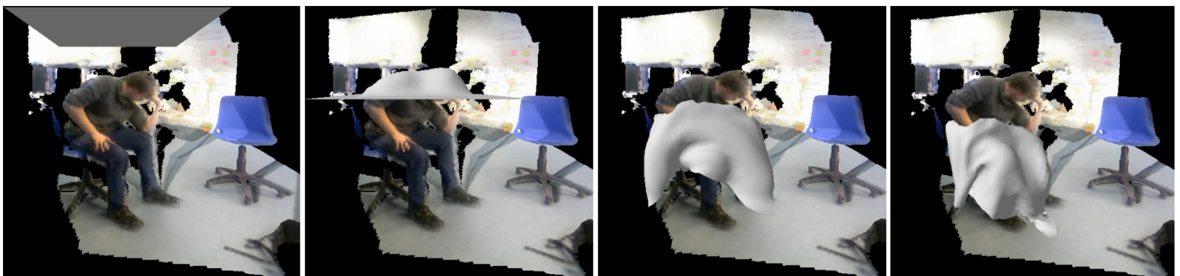

Figure 5.4: Cloth being dropped on a person

5.1.3 Rigid Bodies

Figure 5.5 is also taken from a video, this time showing rigid body interaction with the scene. For the purpose of this demonstration just cubes were used to represent rigid bodies, although any shape could be configured. From these pictures we can see that the cubes collide appropriately with both the real scene geometry and the other virtual cubes around them. They rotate and spin correctly when collisions occur towards the side of the cube geometry and maintain the look of a uniform, structured rigid body despite being made entirely out of completely separate particles. Unfortunately though, since there is no friction with the scene the believability of the collisions suffer somewhat. Out of all the virtual mediums this shortcoming is most apparent with the rigid bodies. Also, since the method is taken from a deformable object implementation, sometimes the rigid bodies can appear very slightly elastic where they collide with something, deform just a small bit and then return to their original rigid form.

The scene representation used to show the rigid bodies in action is rendered just as particles like all scenes were being rendered initially. This is due to the fact that the Eigen math library[42] used to compute the square root of a matrix in the rigid body constraint phase sometimes causes certain conflicts with CUDA. When Eigen is included anywhere in the final implementation it causes errors within CUDA and the project will not compile, so it was necessary to roll back to an earlier version. In some ways presenting the scene rendered in this manner reinforces the decision to spend time rendering it properly in the final implementation.

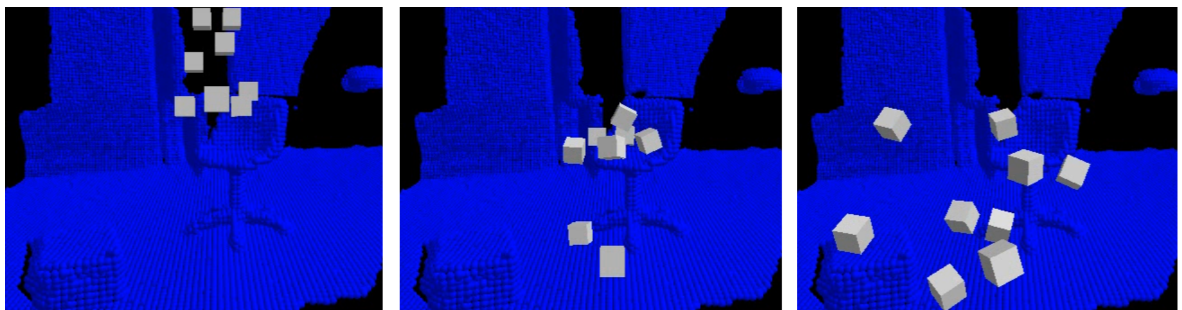

Figure 5.5: Cubes being dropped on a chair

5.1.4 Kinect PoV

So far only the scene being rendered as a 3D mesh with a virtual camera has been presented. An alternative approach implemented was to use the Kinect's point of view as the virtual camera and overlay the Kinect's RGB image on top of the virtual representation of the scene. The advantage of this method is it maintains much greater ties to the real scene. Unfortunately though it requires that the Kinect's much smaller field of view must be used. Because of this the view the user gets of everything is much more close up. Whereas in the previous examples you would be looking at a 4metre cubed area, using the Kinect's FoV would be more like looking at a 1metre cubed area. Due to this fact you get a much closer view of the virtual objects, and each particle will now appear much bigger on the screen. This is not much of a problem when rendering the rigid bodies and cloth as the individual particles are not included in the rendering. However in the fluid rendering particle size is used and if the radius is too big compared to the amount of space available on the screen it will lead to the fluid looking extremely blobby. In an effort to reduce this effect the particle size can be reduced. As discussed in section 4.2.3 a reduced particle size unfortunately means the timestep must also be reduced to avoid tunnelling. Because of this the fluid simulation for the Kinect POV implementation runs at a lower timestep than the others and looks a bit unnatural due to a slower falling speed and velocity applying effects behaving differently.

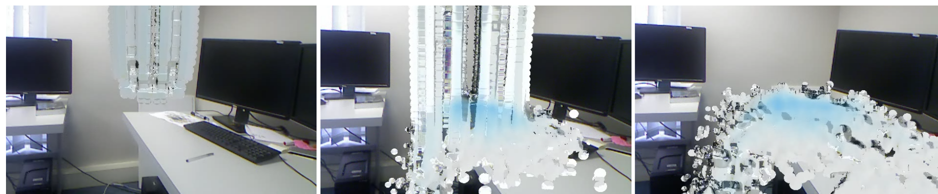

Figure 5.6: Kinect POV of fluid being dropped on a desk

Figure 5.7: Kinect POV of cloth being dropped on some chairs

5.2 Performance

The next relevant area in terms of results is performance. All results were generated using a computer with an Intel Xeon 3.4GHz CPU, 16GB of RAM and an NVIDIA Quadro K2000 4GB GPU. As stated one of the main possible use cases for an application like this is for games, so maintaining an acceptable framerate is of utmost importance. After analysing the performance of the various different parts of this program it was found that by far the biggest bottleneck was Kinect Fusion. The various steps of camera tracking, volumetric integration and then extracting the point data put quite a large drain on performance. The largest defining factor in just how slow or fast these algorithms perform is how many total voxels are included in the volumetric representation of the scene. Within the Kinect Fusion framework it is possible to specify the amount of voxels used per axis, and it is also possible to chose how much area each voxel represents. Figure 5.8 below displays some of the timings with regard to varying voxel amounts.

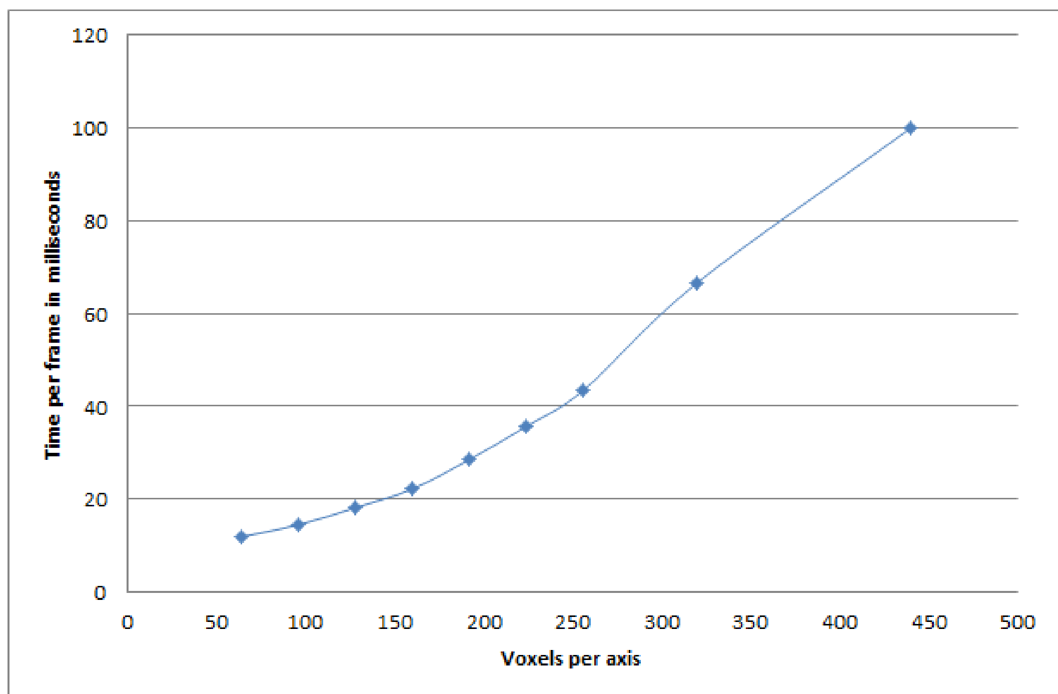

Figure 5.8: Time taken per frame by Kinect Fusion with varying amounts of voxels representing a 4metre x 4metre scene

When choosing the amount of voxels you do wish to use for a particular simulation, certain considerations must be taken into account. The first is the size of the scene you wish to capture. Typically for the scenes presented in the results of this dissertation a 4metre cubed area was needed to sufficiently capture the area we wished to scan in. The next question is what level of detail you need. For a reconstruction which is only needed for the purely physical information, the main defining factor for the level of detail required is that the size of each voxel must at the very least be equal to the relative size of a particle. So if each particle has a diameter of 0.04 metres, in a 4x4x4 representation the resolution of voxels must be at least 100x100x100. This is adequate for giving a physically adequate representation of the scene and is what was used for the RGB overlaid simulation in section 5.1.4. However, when used to generate a mesh of the scene this gives quite a blocky and blurred visual representation. To generate the meshes presented earlier in the results a 256x256x256 voxel block was used instead.

Unfortunately, since the optimization of Kinect Fusion is outside the scope of this project, these timing constraints are unavoidable. For the simulations where we overlay the RGB image and don't have to worry about representing the visuals it was possible to to use a voxel resolution low enough where Kinect Fusion could be performed at the same time as the physics calculations and still achieve suitable framerates. Unfortunately though, where it was necessary to generate the detailed mesh for rendering, it was required that the scene reconstruction be executed in a step previous to simulating the physical interactions.

Although Kinect Fusion is the main bottleneck of the application, timings for some of the typical physics simulations are included in table 5.1. It is important to note that the timings listed are for separate simulations. This means that the time to simulate all of these steps together would be much less than the sum of all three because steps like collision detection etc will only need to be performed once.

	Number of Particles	Time per frame
Fluid	10,000	8.33ms
Cloth	40x40	1.43ms
Rigid Bodies	4 x (4x4x4)	2.27ms

Table 5.1: Time taken per frame for the typical physics simulations.

5.3 Evaluation

As mentioned in the introduction, very little previous work has been carried out trying to obtain this level of physical interaction between the real and virtual using just a consumer grade depth sensing camera. Because there is very little to compare against it is difficult to evaluate the results of this project in the context of this field as a whole. This dissertation has more been treated as an entry into this field where others can refer to this work and try to improve it with their own research or implement their own method if desired. And I feel this has been achieved. A detailed literary review was provided which presented the current state of the art of existing techniques. The key design decisions which must be considered are highlighted and the various thought processes behind every design decision were explained and rationalised. A comprehensive implementation gave an in depth run down of every technique used and provided insight on how to overcome the various problem experienced along the way. And finally results for both quality of interaction and performance were presented in a clear and precise manner, ready for future comparison. I feel that anyone would be able to take this dissertation as a reference and be able to gain a very clear idea on how to go about continuing on this field of research themselves.

As far as how I feel this implementation will compare against future work, I believe that most design decisions chosen were the backed up by logic and were the best choices to make. The decision to base the interactions off particle physics I feel was the right one. The biggest consideration when choosing this is finding a physical solver which matches the kind of information extracted from the scene. I feel that the discrete voxel based representation is the best way of representing the scene and this fits too perfectly with a particle based solver to use anything else. I feel that the interactions this implementation are capable of providing are quite impressive, and with some improvements in future work could achieve some extremely realistic results.

As far as performance is concerned, pretty much every performance intensive task was carried out in parallel on the GPU throughout every stage of the process. Kinect Fusion is all executed on the GPU, the physical solver was all implemented using CUDA and the various techniques for rendering, even the expensive fluid rendering were all carried out on the GPU as well. As mentioned in the state of the art there are existing

techniques out there which have improved on the performance of Kinect Fusion and I feel like this would be the main area to achieve better results in performance.

And finally, one thing that there is no doubt about is the interactions made possible by this implementation far surpass the kind of existing interactions found in games like the previously mentioned Kinect Party. It is difficult to compare the two directly because the results presented in this work were focused on largely static scenes due to some limitations of Kinect Fusion. I do feel though that with this framework in place it would not be a very big task to incorporate it with applications which do allow for more movement, and still be able to reproduce the kind of interactions which will lead to greater depth in gameplay and an overall better user experience.

Chapter 6

Conclusion

6.1 Summary

The stated objective to design a framework capable of handling the detailed and accurate physical interaction between real and virtual has been achieved. Kinect Fusion was used to build a volumetric representation of the real scene and a unified particle solver was used to simulate the various physical interactions. Suitability for extension to games was maintained with real-time frame rates being made possible by executing nearly all of the implementation on the GPU. This dissertation also serves as a point of reference for others who wish to conduct their own research in this area. Whether it be through replicating and improving on the implementation detailed here, or through exploring some of the alternative approaches discussed in the state of the art and design chapters. This implementation is the first of its kind to achieve this level of results, and far surpasses the current alternatives presented in various games.

6.2 Limitations & Future Work

While the potential of this implementation can clearly be seen in the results presented, a few shortcomings hold the system back and prevent it from being ready for integration into an application such as a game just yet. While Kinect Fusion has many many

advantages and was definitely the right choice for this dissertation, it is also the source of a lot of these shortcomings.

The first is the performance limitations. Although real-time framerates were achieved, the system is still quite draining on resources and would not allow for much other computation simultaneously. As mentioned in the state of the art, Kinect Fusion was developed mainly with the aim of using the Kinect to scan in and reconstruct 3D models and being able to maintain a very accurate representation of the surface geometry. As such Kinect Fusion focuses a lot of resources on obtaining a high level of detail. This kind of detail is necessary to maintain the visual fidelity of the scanned in objects, however a much coarser representation would serve just as adequately for physical purposes. Also because it is a library, as a programmer you don't have a whole lot of access to the inner workings, and rather have to go through a layer of abstraction to achieve your goal. It is highly likely that it would be possible to gain a major speed increase if an implementation of camera tracking and volumetric representation tailored exactly to reconstructing scenes in a physically relevant sense was developed.

The second relates to the fact that movement and user input are of course common themes in the usual kind of games developed for the Kinect. Unfortunately though, Kinect Fusion does not deal well with large amounts of movement within the scene it is reconstructing. The Kinect for Windows SDK contains a method for skeleton tracking and it also is capable of matching pixels within a depth scene to a particular skeleton. So for example if the Kinect detects the defining features of a skeleton it will then flag that skeleton, associate it with a certain player number, and then look at the surrounding depth pixels and match the player's silhouette to the skeleton. This means it would in theory be possible to isolate any pixel associated with a player and not include it in the Kinect Fusion scene reconstruction. This would mean user movement would not skew Kinect Fusion's results. It would then be possible to use a much more basic assumption of the players geometry such as assigning collision spheres along the bones. This would allow for a fully 3D representation of the scene which fluid etc can be poured onto and would also include player interaction.

And the final area which I feel would be the next step in future work is improving the fidelity of the physical interactions themselves within the solver. Although the results presented look quite accurate and realistic in a lot of cases, I feel like with a bit of fine

tuning they could be improved even more. A major area of improvement would be to include a friction model, which was omitted from this work due to time constraints.

Appendix A

Appendix

A.1 Links to Videos

Note: Videos also available on CD attached

Fluid

An example of fluid simulation - $https://www.youtube.com/watch?v = LXXsaHa90mI$

Fluid on a desk - $https://www.youtube.com/watch?v = 56wPbTXUZu4$

Fluid on a chair - $https://www.youtube.com/watch?v = V3qc0wJVzNo$

POV fluid on a desk - $https://www.youtube.com/watch?v = QkaG1Qx6bZ4$

Cloth

Cloth on a desk - $https://www.youtube.com/watch?v = S3noKOxPtrk$

Cloth on a person - $https://www.youtube.com/watch?v =_6 IIVzU_DKg$

POV cloth on chairs - $https://www.youtube.com/watch?v = LPYKxeLhS9s$

POV cloth on a desk - $https://www.youtube.com/watch?v = AUZxBQgF_dg$

Rigid Bodies

Rigid bodies on a chair - $https://www.youtube.com/watch?v = 0QO2jsVtqig$

Rigid bodies on a scene - $https://www.youtube.com/watch?v = 5ziVROwHJis$

Miscellaneous

A scene being scanned in - $https://www.youtube.com/watch?v = UUj78PlW748$

Video of fluid render stages - $https://www.youtube.com/watch?v = fdiNAlIqHEo$

Exhibit of fluid refraction - $https://www.youtube.com/watch?v = 4C_2MDPxn2U$

A.2 Source Code

Source can be found on the CD attached.

Bibliography

[1] D. Vlasic, P. Peers, I. Baran, P. Debevec, J. Popović, S. Rusinkiewicz, and W. Matusik, "Dynamic shape capture using multi-view photometric stereo," in *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, (New York, NY, USA), pp. 174:1–174:11, ACM, 2009.

[2] R. C. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainly," *Int. J. Rob. Res.*, vol. 5, pp. 56–68, Dec. 1986.

[3] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces," in *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, ISMAR '07, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2007.

[4] J. Chen, D. Bautembach, and S. Izadi, "Scalable real-time volumetric surface reconstruction," *ACM Trans. Graph.*, vol. 32, pp. 113:1–113:16, July 2013.

[5] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects," in *In Proc.of Vision, Modeling, Visualization*, pp. 47–54, 2003.

[6] R. Schnabel, R. Wahl, and R. Klein, "Efficient ransac for point-cloud shape detection.," *Comput. Graph. Forum*, vol. 26, no. 2, pp. 214–226, 2007.

[7] S. Gibson, "Illumination capture and rendering for augmented reality," 2004.

[8] J. Chen, G. Turk, and B. MacIntyre, "Watercolor inspired non-photorealistic rendering for augmented reality," in *Proceedings of the 2008 ACM Symposium*

on Virtual Reality Software and Technology, VRST '08, (New York, NY, USA), pp. 231–234, ACM, 2008.

[9] "Kinect party," 2012. Microsoft Studios.

[10] B. Freedman, A. Shpunt, M. Machline, and Y. Arieli, "Depth mapping using projected patterns," Oct. 2 2008. US Patent App. 11/899,542.

[11] D. Van Nieuwenhove, W. Van der Tempel, R. Grootjans, and M. Kuijk, "Time-of-flight optical ranging sensor based on a current assisted photonic demodulator," in *Proceedings Symposium IEEE/LEOS Benelux Chapter*, pp. 209–212, 2006.

[12] "Xbox one," 2013. Microsoft. Video game console.

[13] A. Medina, F. Gayá, and F. del Pozo, "Compact laser radar and three-dimensional camera," *J. Opt. Soc. Am. A*, vol. 23, pp. 800–805, Apr 2006.

[14] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," in *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, CVPR '06, (Washington, DC, USA), pp. 519–528, IEEE Computer Society, 2006.

[15] F. Dellaert, S. Seitz, C. Thorpe, and S. Thrun, "Structure from motion without correspondence," in *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, vol. 2, pp. 557–564 vol.2, 2000.

[16] J. Park, S. You, and U. Neumann, "Natural feature tracking for extendible robust augmented realities," in *Proceedings of the International Workshop on Augmented Reality : Placing Artificial Objects in Real Scenes: Placing Artificial Objects in Real Scenes*, IWAR '98, (Natick, MA, USA), pp. 209–217, A. K. Peters, Ltd., 1999.

[17] P. J. Besl and N. D. McKay, "A method for registration of 3-d shapes," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, pp. 239–256, Feb. 1992.

[18] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. 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, (New York, NY, USA), pp. 559–568, ACM, 2011.

[19] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, "Rgb-d mapping: Using kinect-style depth cameras for dense 3d modeling of indoor environments," *Int. J. Rob. Res.*, vol. 31, pp. 647–663, Apr. 2012.

[20] S. Laine and T. Karras, "Efficient sparse voxel octrees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 1048–1059, Aug. 2011.

[21] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, (New York, NY, USA), pp. 303–312, ACM, 1996.

[22] T. Whelan, J. B. McDonald, M. Kaess, M. F. Fallon, H. Johannsson, and J. J. Leonard, "Kintinuous: Spatially extended KinectFusion," in *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, (Sydney, Australia), July 2012.

[23] M. Zeng, F. Zhao, J. Zheng, and X. Liu, "A memory-efficient kinectfusion using octree," in *Proceedings of the First International Conference on Computational Visual Media*, CVM'12, (Berlin, Heidelberg), pp. 234–241, Springer-Verlag, 2012.

[24] M. Niessner, M. Zollhofer, S. Izadi, and M. Stamminger, "Real-time 3d reconstruction at scale using voxel hashing," *ACM Trans. Graph.*, vol. 32, pp. 169:1–169:11, Nov. 2013.

[25] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality*, ISMAR '11, (Washington, DC, USA), pp. 127–136, IEEE Computer Society, 2011.

[26] "Kinect fusion library." http://msdn.microsoft.com/en-us/library/dn188670.aspx.

[27] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex rigid bodies with stacking," *ACM Trans. Graph.*, vol. 22, pp. 871–878, July 2003.

[28] "Bullet physics." http://bulletphysics.org/.

[29] "Havok physics." http://www.havok.com/products/physics.

[30] S. Le Grand, "Broad-phase collision detection with cuda," in *GPU Gems 3*, Addison-Wesley, 2007.

[31] T. Harada, "Real-time rigid body simulation on gpus," in *GPU Gems 3*, Addison-Wesley, 2007.

[32] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, "Position based dynamics," *J. Vis. Comun. Image Represent.*, vol. 18, pp. 109–118, Apr. 2007.

[33] S. Green, "Cuda particles," tech. rep., NVIDIA, 2008.

[34] M. Müller, B. Heidelberger, M. Teschner, and M. Gross, "Meshless deformations based on shape matching," in *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, (New York, NY, USA), pp. 471–478, ACM, 2005.

[35] M. Keckeisen, S. Kimmerle, B. Thomaszewski, and M. Wacker, "Modelling effects of wind fields in cloth animations," 2004.

[36] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, (Aire-la-Ville, Switzerland, Switzerland), pp. 154–159, Eurographics Association, 2003.

[37] M. Macklin and M. Müller, "Position based fluids," *ACM Trans. Graph.*, vol. 32, pp. 104:1–104:12, July 2013.

[38] N. Akinci, G. Akinci, and M. Teschner, "Versatile surface tension and adhesion for sph fluids," *ACM Trans. Graph.*, vol. 32, pp. 182:1–182:8, Nov. 2013.

[39] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, "Unified particle physics for real-time applications," *ACM Trans. Graph.*, vol. 33, pp. 153:1–153:12, July 2014.

[40] "Kinect for windows." http://www.microsoft.com/en-us/kinectforwindows/.

[41] W. J. van der Laan, S. Green, and M. Sainz, "Screen space fluid rendering with curvature flow," in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, (New York, NY, USA), pp. 91–98, ACM, 2009.

[42] "Eigen math library." http://eigen.tuxfamily.org/index.php.

[43] "Thrust - cuda toolkit." http://docs.nvidia.com/cuda/thrust/.