

Studying the effectiveness and algorithms of digital
simulations on an interactive AR crowd simulation
table

Utrecht University



Utrecht University

Noud Savenije 4302907

November 1, 2021

Abstract

We introduce CrowdAR, a realtime interactive tangible *augmented reality* (AR) interface for crowd simulation software. The crowd simulation table's purpose is twofold. It is a tool for researchers and students for studying new interaction techniques in simulated environments. In addition it acts as a tool for teaching museum visitors the complexities of simulation studies. We contribute a spatial AR technique to support dynamic contour detection, allowing users to update the simulation environment in real time, fiducial marker recognition for communicating with the simulation, as well as multi-touch interaction, making the tabletop act as a tablet.

Acknowledgements

I would like to thank my supervisors, Roland Geraerts and Wolfgang Hürst for their insights, support, feedback and hours of meetings and proof reading. Additionally, I would like to thank Nick Brouwer for his help and technical expertise with regards to linking CrowdAR and SimCrowds, Patrick Groot-Koerkamp for getting me up to speed with existing codebase and Unity development in general, as well as the other students who helped develop the prototype: Mark de Jong, Anestis Latos, Niels Mulder, Tiziano Natali, Anca-Mihaela Negulescu, Dimitris Valeris, Floris de Vries, Sietse van der Werf and Clinton Zhuo. In addition I offer my thanks to Michelle Mastenbroek, for her mental support and countless hours of acting as my one and only user test during this pandemic.

Finally, the project was supported by Jet Blokhuis and Aniek Bax from Universiteitsmuseum Utrecht.

Contents

1	Introduction	4
1.1	Project Motivation	4
1.2	Research Question	7
1.3	Contributions	8
1.4	Structure	8
2	Related Works	9
2.1	Interactive augmented reality tabletops	9
2.1.1	Geographical information systems	9
2.1.2	Tangible user interfaces	10
2.1.3	Interactive simulation tables	11
2.1.4	Crowd simulation tabletops	12
2.2	Education	12
2.3	Installation goals	13
3	Preliminaries	18
3.1	Running example	18
3.2	Frames	18
3.3	Coordinate systems	18
3.4	DIRECT algorithm	21
3.5	SimCrowds and uCrowds	22
4	Method	23
4.1	Table installation	23
4.2	Calibration	26
4.2.1	Camera calibration	27
4.2.2	Projector calibration	29
4.2.3	Distortion	31
4.2.4	Invalidation	33

4.3	Contour detection	34
4.3.1	Obstacle types and usability	34
4.3.2	Interactive contour detection	36
4.3.3	Moving objects	38
4.3.4	Intersection filter	38
4.3.5	Movement filter	39
4.4	Multi Touch	40
4.4.1	DIRECT	41
4.4.2	Method	41
4.5	Fiducial marker tracking	43
5	Implementation	45
5.1	System architecture	45
5.1.1	Plugins	45
5.1.2	Data structures	47
5.1.3	Initialization	47
5.1.4	Calibration	48
5.1.5	Operation	48
6	Validation	50
6.1	Validate camera calibration	50
6.1.1	Calibration errors	51
6.2	Validate projector calibration	52
6.2.1	Calibration errors	54
6.3	Validate Contour Detection	54
6.3.1	Operation errors	54
6.4	Validate Multi-touch	55
6.4.1	Operation errors	56
6.5	Validate SimCrowds Integration	57
7	Conclusion	59

Chapter 1

Introduction

Simulation studies are a useful tool for predicting events and behaviors in the real world by using increasingly complex models. Even though these models are as accurate as possible for their intended use, they still have inherent limitations. Conveying these complexities and convincing the public of this, is a hard problem all on itself. *CrowdAR* is a holistic, interactive, *augmented reality* (AR) table for simulating crowds[13]. The goal for the table is twofold. On the one hand it helps teach museum visitors and students the complexities of simulating crowds, and, on the other hand it can be a tool for researchers and students to instantly visualize the immediate effects of changing environments on dynamic crowds. The functionality of CrowdAR is realized by using spatial AR techniques, which are used to project a real-time interactive simulation on a scan of a 3D model that is placed on a flat table. Since the aim of this system is to be exhibited at a museum, it is imperative that the system is intuitive, responsive and robust. The interaction with the system can be classified as intuitive if it can be understood without guidance. It is responsive if the actions taken by users provide clear and immediate feedback. The robustness in this context means the ability to remain operational without supervision as long as there are no changes in external factors, like the table being moved for example.

1.1 Project Motivation

Many interactive AR tabletops have been developed and built as of late. There are even free detailed instructions on how to build your own AR Sandbox made available by UC Davis[1]. These AR sand tables can be engaging pieces in a museum[2] and are intuitive to interact with, because

everyone knows how to play with sandboxes and the one and only degree of freedom in this interaction is manipulating the shapes of the sand. It is important to keep this principle in mind to make the experience educational and satisfying for the many users who will only deal with the table for at most a couple of minutes. There should also be a supervised learning operating mode for the table, where many more specific interactions with the scenario and the simulation need to be available. It is necessary to think through the design of Crowd simulation in an educational and in a museum context. There is a high likelihood the users of the system won't always cooperate when interacting with a museum exhibit, thus they must not be able to ruin the experience of others. Furthermore, people might lose interest if they don't see immediate feedback to their actions. The mission statement of the University Museum Utrecht[5] contains the value they attach to science communication and education, not by transferring knowledge, but by educating their visitors about the scientific process.

In the crowd simulation field, there does not exist a well developed interactive tabletop yet. We believe that having this intuitive and robust simulation table will be a valuable addition for researching crowd behavior and for teaching students about crowd simulation. A prototype of such a table has been developed at Utrecht University[13, 22]. The setup of this installation will be discussed more in depth in the next chapter, but in short it consists of the combination of

a complex, high fidelity crowd simulation[27, 11] with a multi-user, multi-touch tabletop with a *tangible user interface* (TUI). Due to the constantly interactive nature of the table, this has to be a real-time application providing near instant feedback and updates to the simulation. It must remain robust, the simulation needs to run correctly, user interactions need to be



Figure 1.1: *The first version of the CrowdAR installation placed in a museum.*

registered and changes in the environment need to be recognized, all while filtering out noise, like moving arms or reflective objects for example. Maintaining this robustness guarantee and the functionality while computing as little as possible in order to keep running in real-time is the core of this project and we believe this work will benefit the interactive tabletop research field in general.

1.2 Research Question

How can a real-time, interactive AR tabletop simulation be constructed robustly?

This question can be subdivided into its various components: *interactivity*, *real-time operation* and *robustness*.

The *interactivity* aspect spawns the following questions.

- How should the users be able to interface with the system? How should they be able to influence the environment and should they be able to manipulate the simulation settings?
- How should this interactivity work for the different types of users? What is the difference between museum visitors, supervised student and researchers?
- How can interaction be designed such that many people are able to interact with the table simultaneously?

Running a *real-time* interactive simulation also gives rise to a number of questions.

- Since multiple users can interact with the simulation simultaneously, when can the simulation be updated while remaining realtime?
- How can simulation settings be altered in a running simulation?
- How do different users interact when they are all influencing the same simulation?

Then finally there is the question of *robustness*. CrowdAR should be able to run unsupervised for hours on end in a crowded museum setting. This leads to two main questions.

- What is a good, automated design for automatically aligning a sensor and projector for consistent and accurate interaction?
- How can we ensure that this system stays in valid states while operating in a crowded space with possible changing conditions?

1.3 Contributions

This thesis project will have the following main contribution: An interactive tabletop for crowd simulation will be constructed. First and foremost the objective of this thesis is to deliver bounds on the requirements of certain AR, motion tracking and projection algorithms used in the creation of holistic interactive simulation tables. The combination of complex simulations and real-time interactivity has not been achieved in the tabletop world. The design, requirements and optimizations have to be thought through thoroughly, which will hopefully be beneficial for interactive tabletops and tangible user interface development in general.

1.4 Structure

The rest of the report is structured as follows:

- Chapter 2 provides a theoretical background for this project, it puts it in context, discusses current limitations and gives an approach for the rest of the rapport.
- Chapter 3 gives some preliminary definitions of important concepts.
- Chapter 4 discusses the calibration of all systems, gives the method used for interactive real-time object detection algorithm and for the interactive real-time multi-touch detection algorithm.
- Chapter 5 discusses the implementation of the main architecture of the *CrowdAR* system.
- Chapter 6 details the validation of the subsystems.
- Chapter 7 concludes and gives limitations of this study and recommendations for future work.

Chapter 2

Related Works

This section provides an overview of existing *augmented reality* (AR) tables as well as an overview of related techniques that can potentially be used for projection, gesture recognition, object recognition, AR and interaction used in those tables.

2.1 Interactive augmented reality tabletops

Interactive AR tables originated from two separate needs; the need for supporting interaction by dynamically displaying relevant data and the need for displaying a simulation on an often three dimensional model of an environment.

All tables discussed in this section can be found with their specifications in Table 2.1.

2.1.1 Geographical information systems

Geographical Information Systems (GIS) data seems to be a particularly good fit for the displaying information category, and as such, a great many sandtables have been created. The first interactive sandbox was made in 2004 by Ratti et al.[21]. They made two different configurations for their system, where the main difference lies in the 3D sensing technology used. The most accurate and expensive version, called *Illuminating Clay* Fig 2.1b, uses a 3D scanner for its 3D sensing, whereas the other version, *SandScape* Fig 2.1a, senses the geometry of the landscape using infrared light transmitted through the glass beads that make up the surface.

Tateosian et al. made another sandtable which can project real-world data

on physical models called *TanGeoMS*[26]. *TanGeoMS* makes use of two projectors to minimize the shadows cast by steep terrain and interfering arms. It uses a laser camera for scanning the terrain model made from clay or foam boards, as shown in Fig. 2.1c. These specifications make the system prohibitively expensive according to Petrasova[20], so a more affordable table was created as shown in Fig. 2.1d. This physical setup consists of one projector and a Kinect, limiting it somewhat regarding scale or resolution, but lowering cost and thus increasing accessibility. In 2018, Kreyos et al. made an open source table called *AR Sandbox*[1]. The open source setup consists of a formerly commercially available Kinect V2, short-throw projector and consumer grade computer Fig 2.1e. These *GIS* tables are mainly used to process height data in order to project height maps and predict flood planes[20] for example. This means that interaction is usually limited to manipulating the height of the sandbox with differing amounts of granularity. This kind of interaction lends itself well to having multiple people working on it at the same time, but some important considerations do have to be made about update frequencies. Updating the projection too often could give noisy results, because there might be a hand moving over the table or there might still be enduring disturbance in a sandbox after the model has been manipulated. Also the resolution of the scanners can be lower while still providing good results, because the sand or clay models usually form smooth curves, which can be extrapolated from the existing data points.

2.1.2 Tangible user interfaces

Moving away from GIS, there are other applications that use AR for projecting data on a physical model. Another system that uses augmented reality for projecting additional information to aid users in education is the *Tinker Table*[23, 14, 33]. The *Tinker Table* is a system that assists with warehouse design by projecting optimal distance between shelves, reachability and level of stock among other things, (Fig. 2.1f). The *Tinker Table* was built partially to research collaborative learning, so their system has a focus on multi-user collaborative interaction. Keeping the system responsive is helped by using fiducial markers for object tracking. These markers help determining the location and orientation of the objects, while having minimal disruption by noise due to the high contrast of the markers. This table developed by the Swiss EPFL, consists of substantially the same components of the GIS tables, also being made up of a physical model, a camera and a projector.

The main hardware difference between the GIS tables and the *Tinker Table* is that the latter uses an RGB camera. This limits the *Tinker table* to using marker-based AR, with which they had problems due to changing light conditions. There also exists a scaled down version of the *Tinker Table* which uses a multi-touch surface instead of a camera, as shown in Fig 2.1g, which does not have a TUI. The research on the *Tinker Table* resulted in an optimistic view on the effects of TUI's on the users' engagement and collaboration. This view is corroborated by Ma et al.[17], who created physical rings to interact with a multi-touch surface called *MultiTaction*. Their interactive museum exhibit gives users the possibility to explore the type of plankton in different regions of the oceans. They found users kept their interest longer and managed to collaborate better when using the tangible rings when compared to a touch screen.

2.1.3 Interactive simulation tables

The previous tables are all mainly focused on presenting information in a useful way, like being able to project a height map on a sandbox or the distance between shelves. Some tables are more focused on the simulations they show, in combination with the surroundings in which they take place. An example of this is an implementation of the previously discussed *TanGeoMS* which simulates the flood planes of rivers running through the sandbox. Interactive tabletop *simulations* predate computers by quite a while. Kobayashi et al.[15] saw the need to add automated simulation and human computer interaction to an existing analogue tabletop disaster education system. It is a rather primitive example of an interactive simulation table, making use of peripherals and markers in order to facilitate this interaction, which makes the entire experience rather cumbersome according to their user tests. The system in Fig. 2.1h notably does not contain a camera, but touch sensitive paper to facilitate interaction.

The Canadian and American militaries see potential value in the ability of running tabletop simulations. Bortolaso et al. developed *OrMiS*[9] in cooperation with the Canadian forces. Contrary to all previously discussed systems, this table does not make use of augmented reality. The tabletop is made up of a large tablet, thus eliminating the need for camera's and projectors. *ARES*[8], made by the Amburn et al. in the US Army research laboratory, is an AR sandtable, which can potentially be used for dynamically running training simulations. *OrMiS* and *ARES* are interesting for their real-time interaction, but the simulations they run are either slow by nature (*OrMiS*) or light-weight, by following previously specified

paths (*ARES*).

2.1.4 Crowd simulation tabletops

There has been one earlier example of interactive AR crowd simulation by Zheng et al. called ARCrowd[32]. This system focuses on a number of markers in order to facilitate interaction with the simulation. The simulation framework, however, consists mainly of rigid group formation behavior. The system has an RGB camera for scanning the markers, meaning it is not able to sense depth, so it is limited to pre-loaded models. It also opts for a monitor instead of a projector for displaying the results. The *Augmented Reality Table for Interactive Crowd Simulation* that is the basis for this report was first introduced in [12] and it was succeeded by prototype that was developed in [13]. The latter contains an implementation of the interactive crowd simulation application for the HP Sprout [3], which has a wide array of sensors as shown in Table 2.1. The Sprout implementation lends itself well to rapid prototyping, because the camera can scan marker drawings as well as patterns of physical blocks. It is not able to have the physical environment update in real-time, opting instead for user initiated scans, just like *TanGeoMS* did. The interactive table set-up contains a projector for displaying a virtual simulated crowd on a foam block city model. It realizes the interactivity using a Kinect v2[4]. This is used for continuously scanning the city model for changes in its environment and for simultaneously checking for touch input by multiple users. The *CrowdAR* table also implements prototypes for an accompanying app, for an enhanced AR experience and a multi-touch AR user interface for controlling the crowds themselves. These possibilities for interactivity and high fidelity simulation make it difficult are computationally heavy and thus difficult to run in real-time simultaneously.

All in all, interactive tabletops usually do not focus on simulations with environments changing in real-time. The proposed level of interactivity with a complex simulation has not been attempted before. There is a lot to learn about the collaboration aspect of previous works, since this table will run one simulation and everyone influences everyone else.

2.2 Education

The goal of this project is to develop a system that is well suited for education. AR tables are often used as educational tools, but not everyone spends

as much time evaluating if the systems perform well. As discussed, earlier augmented reality seems to be a good fit for helping with GIS education[18], and they conclude that AR supported tabletop interactions are beneficial in this context. Schneider et al.[23] compared the use of a *tangible user interface* (TUI) versus a *multi-touch* interface in the context of teaching students how to design optimized warehouses. Their work is very much influenced by the concept of *Multiple External Representations*, or the scale of abstract to concrete interaction design[7]. They conclude that a tangible user interface akin to the warehouse itself is a intuitive representation for novices, and that the more experienced users are within the domain, the more use they get out of abstraction.

The CrowdAR table is being developed for the University Museum in Utrecht, which is a museum that focuses on scientific exhibitions. The simulation framework behind the CrowdAR system is developed by our research group. The framework, among other thing, handles the creation of the walkable environment and the global path planning, collision avoidance and animation for the agents in the simulation. Affordances are an important part of designing interactive exhibits. People will know how to interact with an AR sandbox [1], even if they have never seen it before. The environment offers user-centric interaction to the user and the AR aids in the education by projecting extra information in the form of heightmaps or floodplains, for example. The target audience for this museum varies significantly – ranging for example from older kids to senior citizens. The CrowdAR table should therefore offer an accessible interactive interface suitable for a museum exhibit. It should be clear from the environment which actions can be used to interact with the simulation. For this reason, the museum wants to make use of both physical and virtual objects, known as mixed reality, on this AR table. Being able to project an area people know, like the area around the museum, a running simulation should help people understand what they are looking at and how they can influence this world. Visitors should thus be able to place physical obstacles on the table surface representing buildings or roadblocks in the simulation, mixing the virtual and physical spaces.

2.3 Installation goals

There are two main goals for this exhibit. Firstly, the visitors should be educated about the complexities of dynamic crowds. They should gain

awareness of the difficulties of quickly altering an entire living space so that everyone can be far enough apart from each other, and also certain design features can act as bottlenecks when trying to evacuate a building or festival area for example. Second, the museum also focuses on engaging the public in the process of science itself, and connecting them to actual research. Thus, the installation should also be used to teach people how simulations are used for scientific research. It should help give the users an intuition for the limits of models and the difficulty of verifying data for example. Due to its educational purpose, the presentation should be interactive and since most visitors will only interact with this installation for a couple of minutes, it should be intuitive and accessible.

For the casual visitor of the museum, the *tangible user interface* with one or a few degrees of freedom is ideal. The changing environment necessitates the simulated people through different paths, allowing the visitors to explore different crowd behaviors. This interaction is also beneficial for the experienced users, since it can be used for the rapid prototyping of simulation environments. The multi-touch interface is beneficial for more advanced users, who want to set up specific scenarios, change simulation settings, or trigger certain events, like an evacuation order. All of these interactions affect the entire simulation, making them unsuitable for unsupervised operation with museum visitors who can not be expected to work together.

The installation itself should be able to deal with crowds, that is, large numbers of visitors as we anticipate larger groups, such as school classes, where each individual wants to explore the system and play with it. This, and the varying age ranges, also pose high requirements on the robustness of the implementation, since we cannot expect kids to always follow certain rules or behavior when interacting with the installation. People will use it in various ways, including things that push its limits and have not been anticipated by the system designers.

First Author	Title	Interface Type	Multi-User	Real-Time (fps)	Simulation	Context	Sensor	Projector	Operating Range (cm)
Ratti (2004) [21]	<i>Tangible User Interfaces (TUIs): a novel paradigm for GIS</i>	TUI: 3D modeling	Possible	0.83fps	None	GIS	320 x 240 laser scanner	Unknown	50 x 50 x 50
Kobayashi (2008) [15]	<i>DIGTable: a tabletop simulation system for disaster education</i>	Digital pen, Sensor puck	collaborative	No	According to preset routes	Disaster Education	Bluetooth table	Unknown	NA
Schneider (2009) [23]	<i>Benefits of a tangible interface for collaborative learning and interaction</i>	TUI: models with markers	Collaborative	Yes (?)	Vehicles	Warehouse Planning	Unspecified RGB	Unknown	Unknown
Tateosian (2010) [26]	<i>TanGeoMS: Tangible geospatial modeling system</i>	TUI: 3D modeling	No	No (manual)	Fluid Dynamics	GIS	Minolta VIVID-910 3D laser scanner	Epson PowerLite 1700/1710c (1024x768)	120-250
Zheng (2011) [32]	<i>ARcrowd-a tangible interface for interactive crowd simulation</i>	Markers	No	Yes (?)	rigid crowd simulation	Crowd Sim	RGB Camera	None	Unknown
Bortolaso (2013) [9]	<i>OrMiS: a tabletop interface for simulation-based training</i>	Multi-touch screen	collaborative	Yes*	According to preset routes	Multi-touch screen	None	NA	
Schneider (2013) [24]	<i>(BrainExplorer) Preparing for Future Learning with a Tangible User Interface: The case for Neuroscience</i>	Infrared pen cutting connections	No	Unknown	Projects what eyes would see based on situation?	Neuroscience	Wiimote (IR)	Short Throw	Unknown
Reed (2014)	<i>Shaping Watersheds Exhibit: An Interactive, Augmented Reality Sandbox for Advancing Earth Science Education</i>	TUI: 3D modeling	Yes	Yes (?)	Water flow	Earth science education	Kinect V1	Unknown	80-400
Amburn (2015) [8]	<i>The Augmented REality Sandtable (ARES)</i>	Sandbox, Mouse/Keyboard, Touch	Possible	Unknown	None (2015)	Military	Kinect V1	Unknown	80-400

Ma (2015) [17]	<i>Using a tangible versus a multi-touch graphical user interface to support data exploration at a museum exhibit</i>	Multi-touch screen OR Tangible Rings	Separate	Yes	None	Museum	Touch-screen	None	NA
Kreylos (2018) [1]	<i>Opensource Sandbox</i>	Sandbox	Possible	Varies	None	GIS	Kinect V2	Varies	50-450
Hürst (2019) [13]	<i>CrowdAR Table-An AR Table for Interactive Crowd Simulation</i>	Tangible, Multi-touch, Pens, hand-held	Yes	Unknown	UUCS Crowd Simulation	Crowd Sim	Kinect V2	Full HD	50-450
Hürst (2019) [12]	<i>(sprout) Augmented and Virtual Reality Interfaces for Crowd Simulation Software – A Position Statement for Research on Use-Case-Dependent Interaction</i>	Multi-touch, TUI, Pens	No	No	UUCS Crowd Sim	Crowd Sim	HP Sprout (Intel Realsense)	HP Sprout (1024x768)	0-10 (from worksurface)
Moore (2020) [18]	<i>(Otago) Comparative usability of an augmented reality sandtable and 3D GIS for education</i>	Sandbox	Possible	Yes (?)	None	GIS	Kinect V1	Unknown	80-400

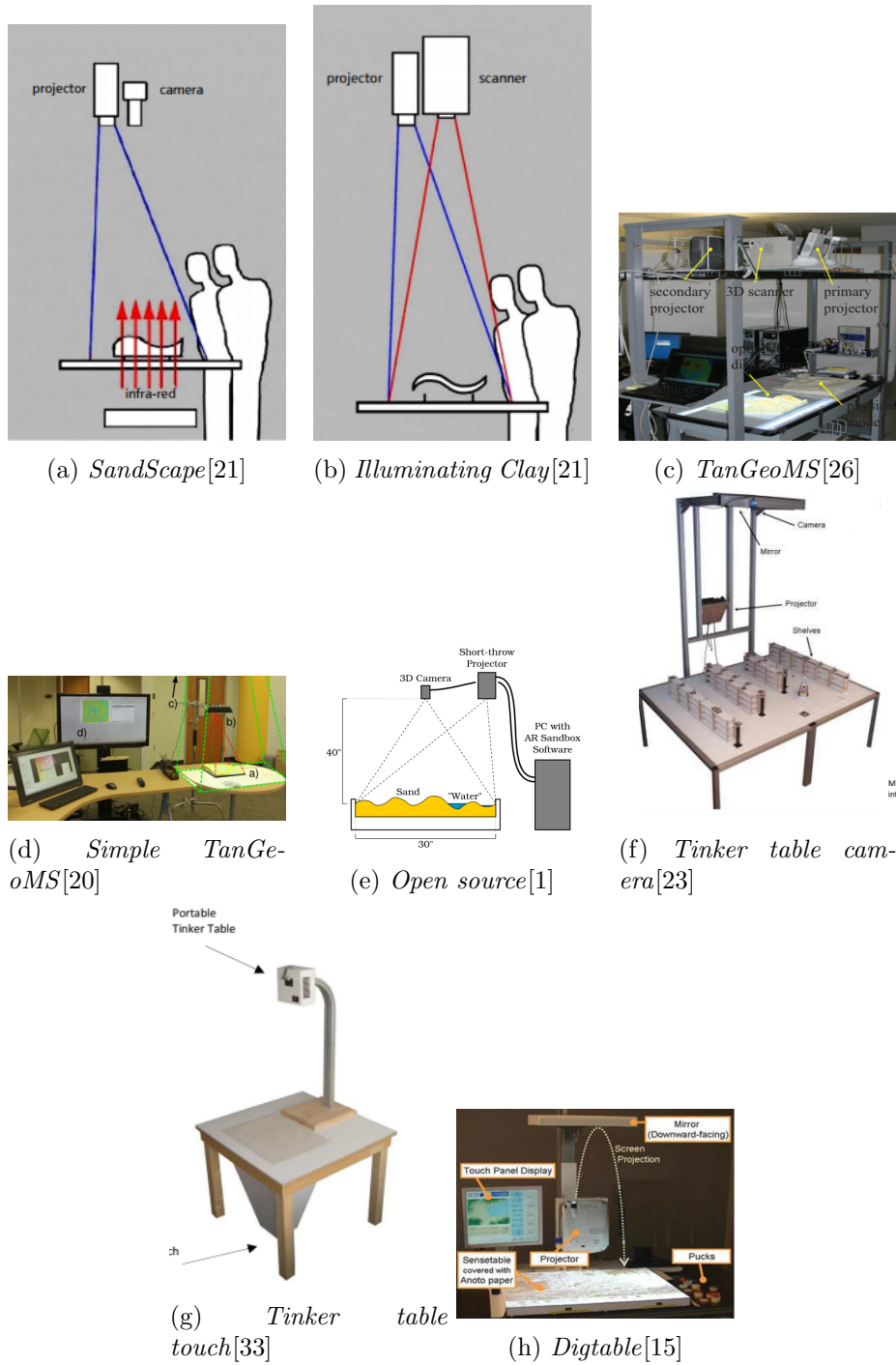


Figure 2.1: *Interactive Tabletops*

Chapter 3

Preliminaries

3.1 Running example

Our tangible user interface consists of four components: A table top, a projector, a color camera, and a depth and infrared camera. Figures 3.1 and 3.2 show a picture and a diagram of the current CrowdAR installation. A computer processes the images according to the requirements and runs the crowd simulation software.

3.2 Frames

Interfacing between the camera and the rest of the program is done using *frames*. A *frame* is a set of matrices that consists of a color, depth and a infrared image. The resolution of these images is dependent on the capturing device.

3.3 Coordinate systems

Between all distinct parts of this installation, a total of five coordinate systems are used. These are the *color camera space*, the *depth* and *infrared camera space*, which handle the input from the camera. The *table space*, the representation of just the table top on which all processing is done. The *game space* that handles all communication with the crowd simulation software. Finally, the *projector space* that is responsible of transforming the image so it can be projected back on the table top. Images can be transformed between these spaces after the calibration is complete.

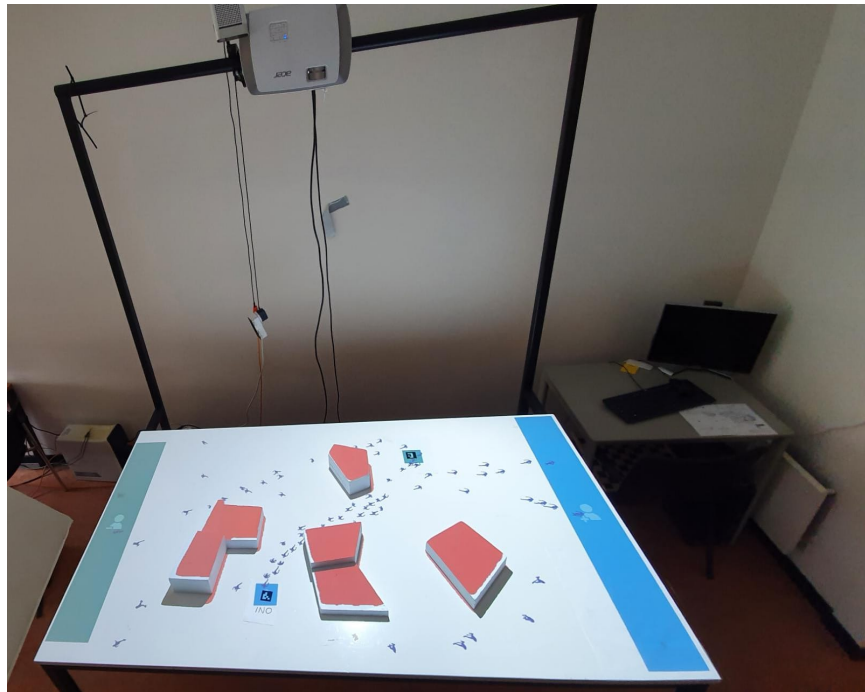


Figure 3.1: *The CrowdAR installation*

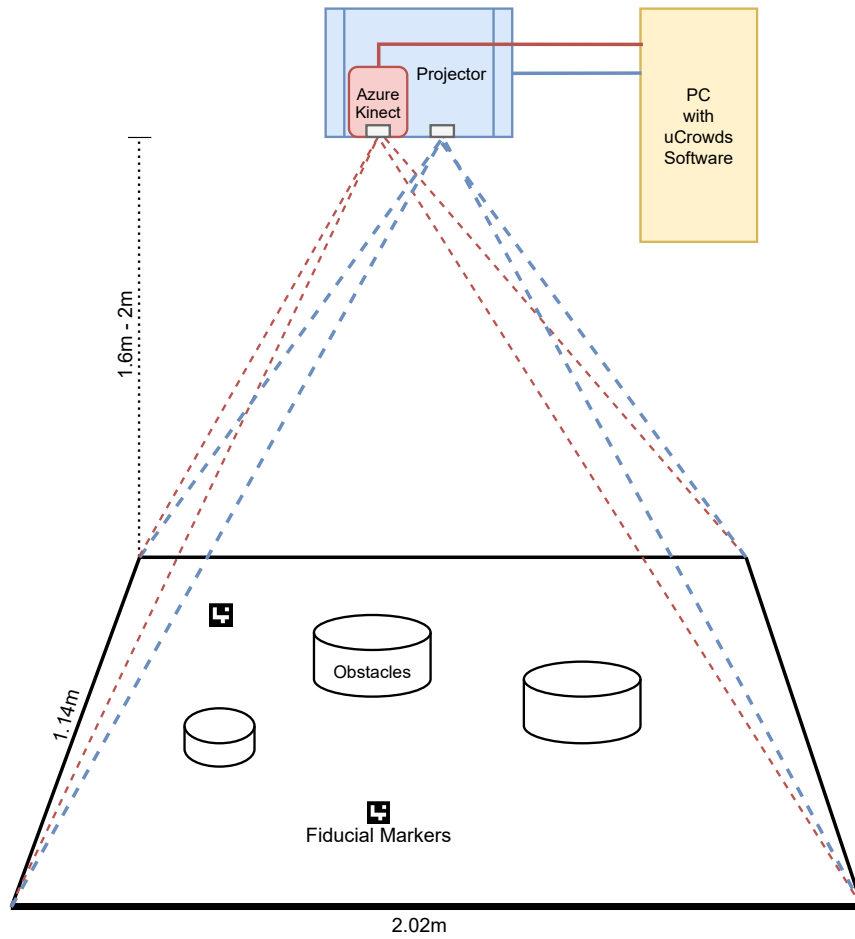


Figure 3.2: *The CrowdAR installation diagram*

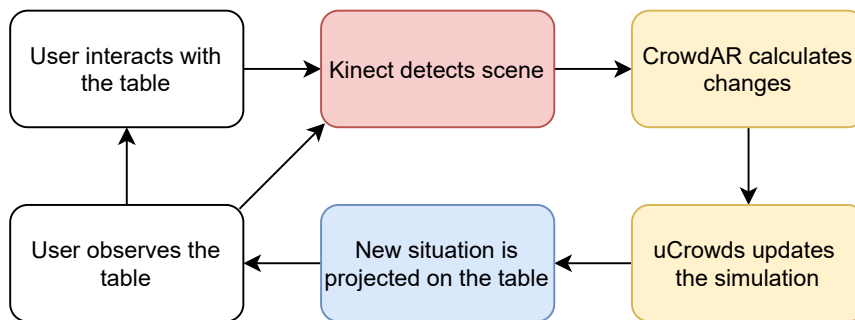


Figure 3.3: *The real-time interaction loop*

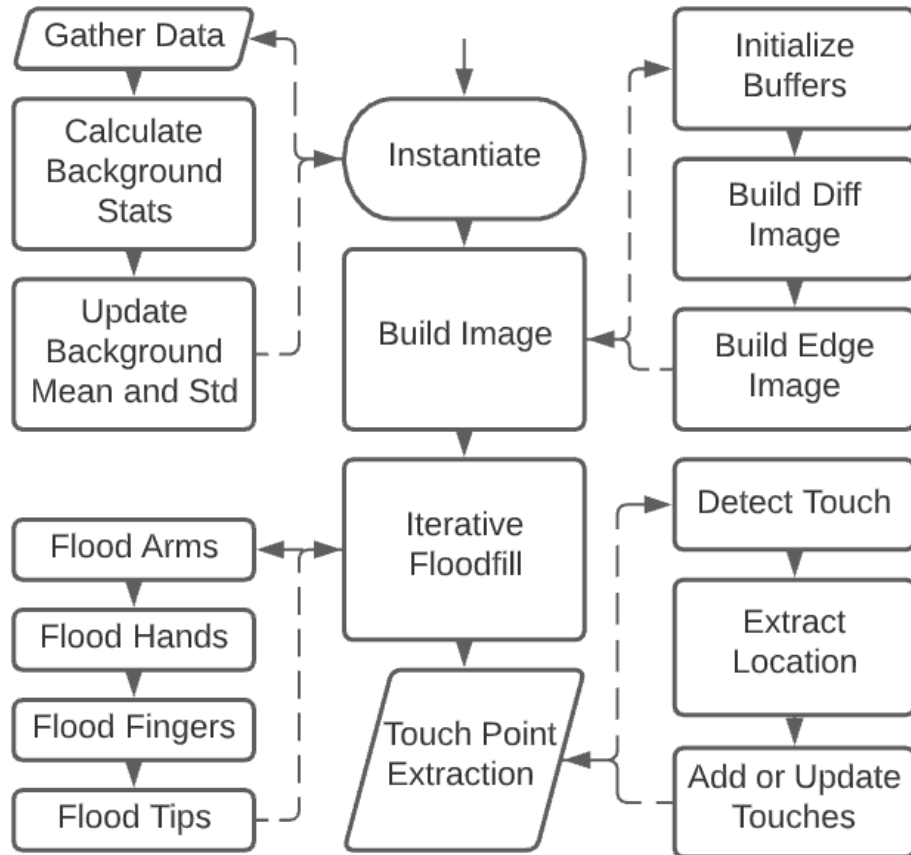


Figure 3.4: An overview of the program flow of the *DIRECT* algorithm [31]

3.4 *DIRECT* algorithm

The multi-touch tracking algorithm we have implemented is called *DIRECT* [31]. An overview of the program flow of this algorithm can be found in Figure 3.4. The source code for the original implementation of the algorithm used in [31] is accessible online at [29]. It is written in *C++*, while our system is based in Unity. In order to keep our implementation as flexible and extendable as possible, we decided to rewrite it in Unity *C#*.

3.5 SimCrowds and uCrowds

The underlying crowd simulation software used for this augmented-reality table is called SimCrowds [11], a real-time, interactive simulation tool. The engine that runs the simulation for SimCrowds is the uCrowds framework [27], which can simulate hundreds of thousands of agents in real-time on consumer computers.

Chapter 4

Method

4.1 Table installation

The setup of the CrowdAR table consists of *four* distinct parts: A projector, a color and depth camera, a computer and a table. The dimensions of our table are 202cm by 114cm, which creates a resolution of approximately 16:9. The *projector* has a full HD resolution, meaning that each pixel projected on the table has a minimum size of $\frac{2020mm}{1920px} \approx \frac{1140mm}{1080px} \approx 1.05mm$ if the projected screen fits the table perfectly. The *depth camera* we use is the Azure Kinect[6], the framework also supports the discontinued *Kinect v2* as well as the *Intel Realsense*. However, since the depth data produced by the Intel Realsense is too noisy for our application and the older Kinects are discontinued, we have only used the Azure Kinect for our tests.

The depth camera of the *Azure Kinect* has two relevant operating modes. The *NFOV* mode has a narrow *Field of View* (FoV), small resolution and large operating range, whereas *WFOV*, the other mode, is the exact opposite. Which mode is best depends on the mounting height of the camera relative to the table size. In our case this is the *NFOV* mode. It has a hexagonally shaped illumination mask, meaning that in order to fit the entire worksurface, the camera needs to be mounted slightly higher. Table 4.1 shows the pixel sizes at the optimal resolution, which could theoretically be realized by putting the camera as near to the table as the field of view allows. The *WFOV* mode uses a circular illumination mask. It captures images through a fish-eye lens, which becomes very distorted when projected to a two dimensional rectangular image, but this can be corrected for. Looking at Table 4.1, it seems counterintuitive to use the low resolution mode, but Figure 4.1 shows that using the *NFOV* mode will result in a larger ef-

Mode	Resolution	FoV	Height(m)	Pixel Size(mm)
NFOV	640x576 (10:9)	75°x65°	1.59	3.51
WFOV	1024x1024 (1:1)	120°x120°	0.58	1.97

Table 4.1: *Depth camera operating mode information and minimal mounting heights for our table. The hexagonal NFOV is taken into account.*

fective resolution on all available mounting heights. The following formula describes the effective depth image resolution for both cameras at all viable heights. Viable heights are those at which the table is within the operating range of the sensor and the entire surface is in view. To get the other value for the resolution, r needs to be divided by $16/9$.

$$r = \frac{wx}{2(\tan(\frac{\alpha}{2}) * h)}$$

r = effective resolution

w = width of table

x = camera resolution

α = Field of View

h = camera height

Mounting the camera as low as possible comes with its own set of problems. The most obvious issue arises when the camera is within reach of users. Should a user move or rotate the camera, then the system needs to recalibrate. The second issue is that objects cast longer shadows with a lower camera. Since we are using only one projector and one camera, there will always be shadows to deal with. Figure 4.2 shows the difference between the shadows cast by a 10cm high obstacle from a camera mounted at 200cm above a table with one mounted 100cm above it. It clearly shows that a low camera will inevitably create longer shadows, or blind spots, behind obstacles on the surface. Mounting the camera at an angle and placing high obstacles at the edge of the table will only worsen this issue, since the angles relative to the objects and the table surface would increase. This is described by $s = \frac{x*y}{h}$, where s is the shadow length, x = object distance from camera center, y = object height and h = camera height. This means, in order to minimize the shadows, that the further from the center an object is put, the lower it should be.

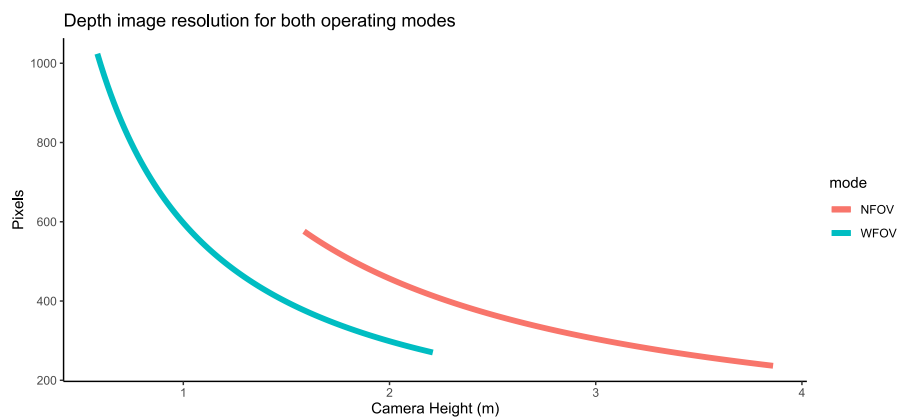


Figure 4.1: *The effective resolution of the depth image is always higher when using the NFOV operating mode.*

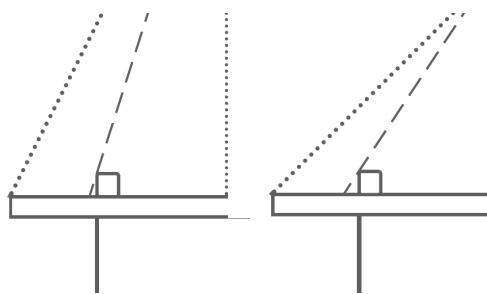


Figure 4.2: *Depth camera shadows increase in size with a lower camera.*

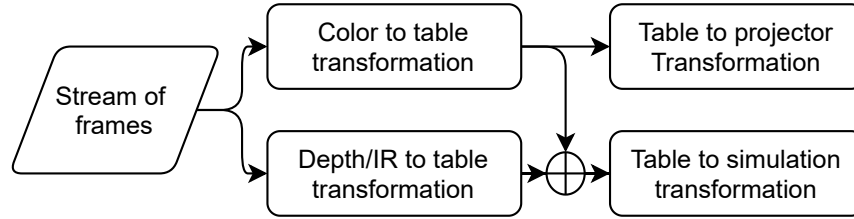


Figure 4.3: *The steps and dependencies for creating the transformations between the coordinate spaces*

4.2 Calibration

Calibration of the system is required to enable communication from the sensors to the table and the simulation and from there to the projector. The infrared and depth cameras are used for the touch interaction and the depth camera is also used for contour detection, which should only be possible on the table top. The interaction with the table relies on the projection of an accurate image of the simulation on the table top. To project such an image, without also projecting on the floor as in Figure 4.6, a perspective transformation has to be applied to the image. These transformations are done using transformation matrices which can be generated when it is known where the corners of the table exist for each system. The sensors, the table, the simulation and the projector all use coordinate systems that need to be aligned, so we define a coordinate space for each. The *table space* for the computer vision, the *simulation space* for the crowd simulation, the *Camera space* for the sensors and the *Projector space* for the projection. The calibration is done in two distinct steps. First, we calibrate the sensors in the camera (color, infra-red and depth), and secondly the simulation and projector calibration are done. This step needs a sensor to already be aligned with the *table space*, hence why they are calibrated last. This is illustrated in figure 4.3.

The calibration algorithms make a number of assumptions:

- The *table* has to be rectangular, because the four corners are determined by drawing the longest lines between two sets of points on the circumference of the table.
- The *table* color has to contrast with the background color, so no bright white table on a bright white floor.
- The *table* has to be opaque and non-reflective.

- The *table* and the background should be on different heights in order for the depth camera to calibrate.
- The entirety of the *table* should be in fit in the view frustum.
- The *Azure Kinect* should be kept in an ambient temperature between $10^{\circ}C$ and $25^{\circ}C$ and an ambient humidity between 8% and 90%.
- The *table* should preferably be white in order to be able to project the largest range of colors.

4.2.1 Camera calibration

For all three sensors, the table corners are extracted from the largest contour in the image by finding the longest diagonals from the convex hull of this contour. It will automatically be assumed that the largest contour in the frame is the work surface. Figure 4.4 shows this system in action.

To make the algorithm more robust, a white image is projected on the table to create a better contrasting image, since contrast is important to differentiate the foreground from the background. Contrast is essential for infrared calibration as well, but flooding the table with infrared light would *invalidate* all relevant pixels when working with the Azure Kinect. This will be discussed further later in this chapter.

Algorithm 1 describes how the color, infrared and depth cameras find their respective table corner points. These table corner points are used for creating a perspective transform in order to fit the table surface precisely in the game space.

Algorithm 1: Camera calibration: *frame, contrast*

```

Result: Matrix transformationMatrix
ProjectWhiteImage();
for n frames do
    image = BoostContrast(frame, contrast);
    edges = Canny(image);
    contours = FindContours(edges);
    contour = GetLargestContour(contours);
    hull = CreateConvexHull(contour);
    cornerPoints.add(FindDiagonals(hull));
end
tableCorners = GetMedian(cornerPoints);
transformationMatrix = GetPerspectiveTransform(tableCorners)

```



Figure 4.4: *Calibration of color camera 1. Largest contour found by using Canny edge detection.*

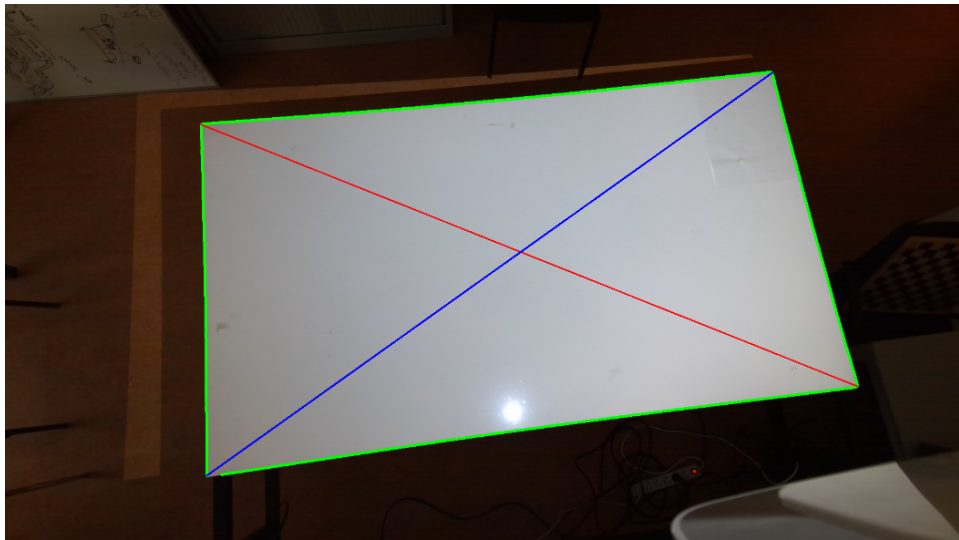


Figure 4.5: *Calibration of color camera 2. Searching corner points by using longest diagonals.*



Figure 4.6: *Calibration of color camera 3. Corner points found by using diagonals.*

4.2.2 Projector calibration

Calibrating the projector space is somewhat more involved. Finding the table like in Algorithm 1 does not work, because we have no way of knowing where the camera is actually projecting, but we can project images to known coordinates relative to the entire image. After calibrating the camera, we do know where the table corners can be detected.

Algorithm 2 describes the process of finding the angle with which a line segment with a chosen origin intersects the known table corner. Figure 4.7 shows the drawing and detecting of these line segments. When the angles of two or more of these lines with different origins are found, the intersection between these lines can be calculated. This coordinate of this intersection represents the corner point of the *Projector Space*. This process is repeated for each of the remaining three corners. The matrix of the four corner coordinates is used to calculate a perspective transform from the projector corners to the table space. The inverse of which is used to generate an image that can be projected on the table without distortion using OpenCV's *warperspective* function.

Aligning the projector to the table surface takes significantly longer than the sensor calibration. The main bottleneck for this operation is that there

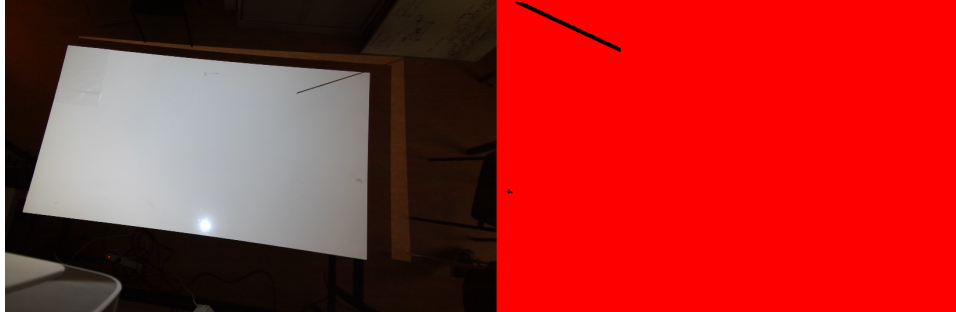


Figure 4.7: *Projector calibration. A line segment with a known origin is projected on the table. It is then detected in the table space, in which the coordinates of the corners are known.*

is a delay between sending a new image to the projector and being able to detect it by the camera again. With current hardware this delay seems to be about $400ms$, so there has to exist a slightly longer delay in code before trying to detect the newly projected line. The table has four corners and each corner is approached from three origins to ensure precision. The number of iterations Algorithm 2 has to do is approximately $i = \log_2(\frac{180}{\epsilon}) \approx 14$, where $\epsilon = 0.01$, because the algorithm uses binary search to find the correct angle from a domain of half a circle to a precision of ϵ . So finding one correct line takes $14 * 500ms = 7$ seconds. In total at least 12 lines have to be identified, meaning that the entire corner detection process takes about one and a half minutes, but since coordinates can be saved, this calibration process only has to be done once every time the camera or work surface change position.

Algorithm 2: Find projection angle: frame, origin

```
Result: double angle
while not foundCorner do
    ProjectLine(origin, angle);
    color = GetWarpedColorFrame(frame) ;    // transform color to
        table space
    DetectLine(frame) ;           // Using Canny and Hough Line Transform
    distance = GetEuclidianDistanceToLine(corner, line);
    if distance ≤  $\epsilon$  OR  $\Delta$ angle ≤  $\epsilon$  then
        | foundCorner = true;
    else
        | if overshotCorner then
            | | angle -=  $\Delta$ angle/2;
        | else
            | | angle +=  $\Delta$ angle;
        | end
    end
end
```

4.2.3 Distortion

By using the wide field of vision mode for the *Azure Kinect*, we can make it possible to move the device closer to the table, but this mode suffers from severe distortion due to the fisheye lens.

The frames from the *NFOV* sensor mode also have to deal with radial distortion, albeit less noticeable. The infrared and depth streams deal with *barrel* distortion. As shown in Figure 4.8, the effect of this barrel distortion is more pronounced the further away from the center of the image. This is a problem for the table corner detector, since it makes the table seem not rectangular, breaking the first assumption and thus causing misalignment errors in the *table space*.

Radial and tangential distortion is corrected by using the Brown-Conrady distortion model[10]:

$$x_u = x_d + (x_d - x_c)(1 + k_1r^2 + k_2r^4 + k_3r^6 + \dots) + P_1(r^2 + 2(x_d - x_c)^2) + 2P_2(x_d - x_c)(y_d - y_c)\dots \quad (4.1)$$

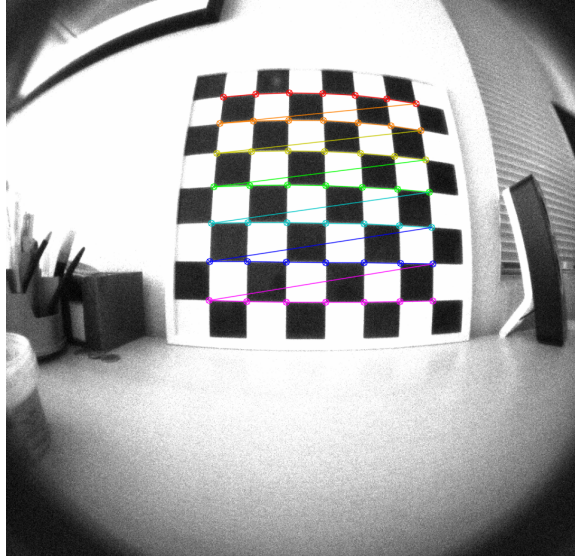


Figure 4.8: *The Barrel Distortion is fixed using a known grid shape.*

$$y_u = y_d + (y_d - y_c)(1 + k_1r^2 + k_2r^4 + k_3r^6 + \dots) + 2P_1(x_d - x_c)(y_d - y_c) + P_2(r^2 + 2(y_d - y_c)^2) \dots \quad (4.2)$$

(x_u, y_u) = undistorted image point

(x_d, y_d) = original, distorted image point

(x_c, y_c) = image center

k_n = n -th radial distortion coefficient

P_n = n -th tangential distortion coefficient

r = Euclidian distance between the (x_d, y_d) and (x_c, y_c)

These series can be expanded for increased accuracy, but in our case this creates an accurate enough picture. The *Distortion coefficients*: $\{k_1, k_2, p_1, p_2, k_3\}$ are calculated by capturing a number of pictures of a known object, in our case the chessboard in Figure 4.8. Once sufficiently many pictures have been collected, we use *OpenCV* to calculate the *distortion coefficients*. By using these coefficients and the camera matrix containing the focal lengths $\{f_x, f_y\}$ and optical centers $\{c_x, c_y\}$, an inverse distortion matrix can be computed.

$$\text{camera matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$



Figure 4.9: *Fixing distortion: The left picture is the corrected image, the right one is the original. Notice the error is larger further from the center.*

$$\begin{aligned}
 \text{Color camera matrix} &= \begin{bmatrix} 1.33\text{e-}3 & 0 & 9.80\text{e-}2 \\ 0 & 1.38\text{e-}3 & 5.58\text{e-}2 \\ 0 & 0 & 1 \end{bmatrix} \\
 \text{Infrared camera matrix} &= \begin{bmatrix} 6.90\text{e-}2 & 0 & 2.90\text{e-}2 \\ 0 & 6.85\text{e-}2 & 2.47\text{e-}2 \\ 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

The distortion coefficients are used early in the pipeline to initialize *OpenCV's* undistort rectify map. Once this has been initialized, the *remap* function is used to perform the inverse distortion map on each relevant frame. The results of this process are shown in Figure 4.9 .

4.2.4 Invalidation

The *Auzre Kinect* has some limitations with regards to its edge cases. When pixels become invalidated, their height is set to 0 and they show up as black pixels in the image. Invalidation happens when the sensor is uncertain about distance of the pixel to the camera. This happens when the pixel are outside of the active infrared (IR) illumination masks, meaning outside of the hexagon for the *NFOV* setting (examples *(A, B, C)* of Figure 4.10 and outside of the circle for the *WFOV* setting (example *D*). Pixels are invalidated when the sensor gets saturated or undersaturated. Saturation happens directly under the camera in our case, the active IR loses its phase information, because the amount of light reflecting is too great. When a signal is ambiguous, because it is on an edge between the foreground and the background, the pixels in question also get invalidated. This happens at the edges of the table and when placing tall objects on the work surface.

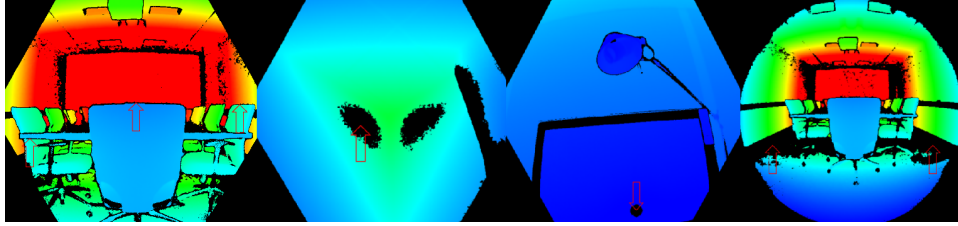


Figure 4.10: *Invalidation from left to right: A: Ambiguous depth due to mixed signal from foreground and background. B: Ambiguous depth due to infrared reflection from walls. C: Phase data lost due to saturation. D: Floor invalidated due to under saturation.*

4.3 Contour detection

Users of the installation can place physical objects on the table surface to act as obstacles for the pedestrians in the simulation. The simulated pedestrians should then respond to this new obstacle in real time by finding a new route to their respective goals. In addition to quickly recognizing the new obstacles, the system should also be able to exclude noise, like moving arms, from being added to the simulation.

The *uCrowds* framework[27] includes functionality to add and remove virtual objects to or from the Explicit Corridor Map[28] in real time, by recalculating the entire navigation mesh. So in order to update the obstacles in the simulation, we would have to make sure there is always a stable, recent version of the objects on the table top available to enable the framework to check whether or not a recalculation has to be performed.

4.3.1 Obstacle types and usability

Most users of this installation will not be domain experts, so giving them clear and near instant feedback of their actions in the form of responsive agents is important to understand the interaction with the system. To achieve this feedback, any object that can be found around the exhibit, backpacks, mugs, spectacle cases or the arms of the users themselves for example, should function as an obstacle in the simulation in a pinch, as this flexibility should enhance the users' ability to experiment and it should ease frustrations when certain actions yield no result or feedback from the system. Consequently, we have opted to use markerless tracking of all obstacles on the surface instead of using specific markers for the obstacles, like infrared markers or QR-codes.

There are still certain **limitations** to what objects can be detected, which are the following:

- Objects must not be too reflective.
- Objects must be opaque.
- Objects must not be too small.
- Objects should not be too detailed.

Material types

The limitations on the reflectance and opacity properties are necessary, as transparent objects can not be detected by *Time of Flight* (ToF) depth cameras[16]. The *Azure Kinect* uses such a sensor, so translucent, transparent and overly reflective obstacles or table surfaces can not be used in this installation. Examples of suitable materials are styrofoam, light wood or other matte painted plastics.

Object size and detail

Objects need to be sufficiently large to be detected by the sensor, as the resolution of the depth camera remains limited. If the camera is mounted at 2m above our table the resolution would result in 456x256 pixels, or 2mm per pixel. To make a stable image, where noise is kept to a minimum, a minimum size can be set for objects to be recognized. Objects that have a minimum length of 25mm in each dimension seem to be sufficiently large.

The simulation uses an *Explicit Corridor Map* (ECM)[28] as a navigation mesh for path planning in the constructed environment. This mesh becomes increasingly dense when objects are more detailed, which slows down both its construction and the agents' pathfinding. Having simple objects is not a strict requirement however, as the simulation can function just fine with a larger, more complex ECM if a decent computer is used. Obstacles can also be simplified to a variable degree in the object detection stage by using the common morphological operations of erosion and dilation. This will smooth jagged edges, making the obstacles less complex, as well as help filtering out noise. Each line segment requires two points to be stored, so simpler shapes aid in keeping the memory requirements down. It is possible to simplify the contours of the obstacles to improve performance very slightly, while decreasing the precision, but this is not necessary on modern computers.

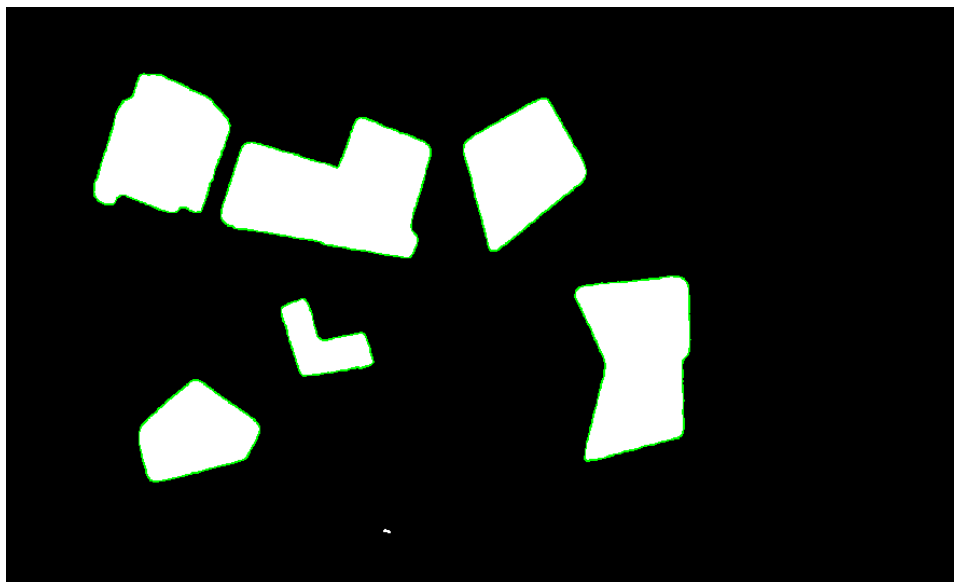


Figure 4.11: *Detecting houses from a model of a city block.*

4.3.2 Interactive contour detection

In order for the objects to be recognized by the system, they need conform to the limitations mentioned above.

The crowd simulation takes place in a single layer environment and it is projected on the flat table surface, so it is sufficient to detect the contours of each object to properly detect the situation at table or simulation level. Note that obstacles with holes and nested obstacles are possible to detect using their hierarchy. The algorithm functions by first trying to detect all contours from each depth camera frame. The image is subdivided in a grid of cells, which are classified as active if they contain minimal movement. These active cells allow partial updates on the table. The contours are then classified as candidate contours. To ensure a stable image, only large enough, stationary objects in a cell with minimal movement are considered. These viable candidates are selected using the following three filters: The first and most simple filter is the size filter. This tests if the detected object is large enough. the second tests if the object itself has been stationary for at least n frames and the third tests if the candidate exists in a region of the table in which there is minimal movement. When a candidate object has passed through all these filters, it can be added to or removed from the simulation. The filtered objects are compared to the objects in the simulation on every

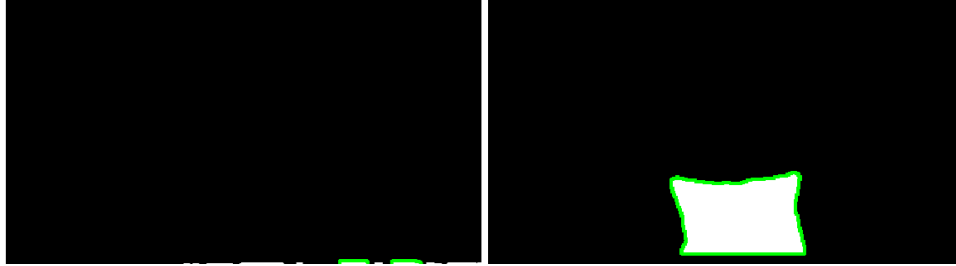


Figure 4.12: *A: Edge noise due to lack of filtering of invalidated pixels. B: Detecting single object with correct filters.*

4.3.3 Moving objects

Moving objects, such as the visitors' arms, should not act as obstacles. There are two reasons for this. The first reason is that the user probably did not intend for their arm to be counted as an obstacle when placing blocks or other obstacles on the table surface, so incorporating these body parts could lead to confusion. The second reason is that it unnecessarily strains the hardware. It requires a lot of computing power for the simulation to recalculate the navigation mesh and the global paths for all affected agents, so it is helpful to update only when it is likely that an object remains stationary.

These moving objects are filtered out using a two-fold solution. Part one is the *Intersection Filter*. The *Interactive Contour Detection* system runs on a certain configurable framerate (*fps*) and every object is only counted when it has been stationary for at least n frames. Meaning that every contour has to have been stationary for $\frac{accTime}{fps} = s$ seconds. Part two is the *Movement Filter*, which is responsible for dividing the table in cells and blocking only the cells in which there is movement, meaning that updates only occur in stable cells, while saved images are used for the noisy cells.

4.3.4 Intersection filter

This first filtering pass is done on a pixel level by testing if an object was detected at that place for at least n frames in a row, as shown in Figure 4.12. Updating more frequently improves the system's responsiveness, but runs a greater risk of showing arms of users or objects that have just stopped moving when this is not intentional. Resulting in *false positives* that cause frequent rerouting of agents, slowing down the simulation. Updating less frequently makes this system less responsive and increases the effects of



Figure 4.13: *A: Purple bordered cells are inactive due to movement. B: Inactive cells use objects from the last time they were active.*

false negatives through noise, because every mistake is carried over through the next full refresh. This noise has a similar effect to erosion, so when using lower frame rates dilating the accumulated image, before detecting the contours, helps mitigate the problem of holes starting to appear within objects.

4.3.5 Movement filter

The second part of noise object filtering is done by dividing the image up in multiple cells, shown in Figure 4.13. For a robust and consistent representation of the objects, they can only be manipulated in the simulation if all cells in which it resides are active. This is necessary to filter objects which are still being touched and stops the merging of objects with pointing fingers, while still being able to update the simulated obstacles frequently and accurately. That is, without leaving parts behind. This comes with its own set of problems, namely the relatively low resolution of the depth camera, objects that cross over the cell borders and an increase in overhead.

The method works by keeping track of separate, learning, background subtractor, which deals with motion. This motion detector can have a lower resolution to the general contour detection, since we do not care about detail or specific placement of objects. The normal contour detector is checked first and provides a list of candidate obstacles. If and only if a candidate obstacle is contained entirely within cells which experience minimal motion is it added to the simulation.

The main drawback of this solution is that a crowded table with populated with large obstacles, can easily lose a great deal of its interactivity. If an object spans many cells, it becomes increasingly likely there is excessive movement within at least one of these cells, perpetually postponing the ad-

dition of the large object to the navigation mesh. This problem is mitigated by not using excessively large objects or by brief cooperation between all users.

Inactive cells can not send new information to the simulation. The last stable configuration of objects remains the configuration used until the movement has died down and the cell becomes active again. This archive only keeps whole contours, so as long as the specific part of the table remains immutable, the simulation keeps the archived versions of all the contours intersecting that cell. This ensures that no partial updates to the simulation can be performed.

4.4 Multi Touch

Touch controls have become ubiquitous in today's world. Devices such as smartphones, tablets and laptops have been using multi-touch for over a decade now and they have rapidly become widely adopted. This familiarity should assist many users with getting to grips with our installation and since most museum visitors will only interact with the table for a couple of minutes, it is vital that the interaction is intuitive to the target demographic.

There is a difference in kind between the two main systems used for multi-touch tracking, which are approaches based on touchscreens and those based on camera's. The former being the largest of these categories by a sizable margin, almost all commercial touch device use touchscreens. One advantage of using touchscreens are that they can be designed very compactly, the front face of contemporary smartphones and tables are almost entirely interactive while managing to remain slim. Another positive attribute is the ubiquity mentioned before. Large multi-touch tables have been commercially available for some time now and they have become very stable and dependable in

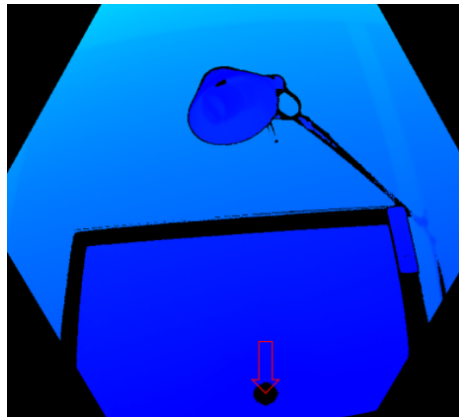


Figure 4.14: *A saturation of infrared light trips up the sensor*

recent years.

Despite these advantages, our system uses a camera-based approach for touch tracking. As our system is a static museum exhibit, its compactness is not relevant, because in order to be able to project on top of the obstacles, a depth camera and projector mounted above the surface are necessary anyway. A saturation of near-infrared light invalidates the affected region of the depth and infrared sensors of the depth camera, shown in Figure 4.14. This saturation makes using a combination of both methods infeasible.

Another problem is that large multi-touch surfaces are expensive and do not scale once purchased, whereas a regular table top is easier to replace and cheaper to resize. To accommodate a larger work surface, the projector and camera can be mounted higher above the table.

4.4.1 DIRECT

Our specific installation requires accurate tracking of touch point positions. Many users need to be able collaboratively interact with the surface from any angle, so the algorithm needs to deal with occlusion well. A recent comprehensive review of touch tracking methods for tabletop projections[19] showed that DIRECT[31] is a good fit for our needs. DIRECT's system setup happens to be similar to our own, as it also consists of a combined depth and infrared sensor and a projector rigidly mounted above an arbitrary table surface it fits all our requirements.

4.4.2 Method

The algorithm works in four distinct steps as shown in figure 3.4. First, it creates a dynamic model for the background, then the edges between the background and the arms are detected and a sequence of flood-fills is performed, starting at the arms and working its way down to the fingertips. After all this it is determined if the fingertip was actually touching the table and if so, the location is extracted, and the touch is registered by the system.

The **dynamic background model** is separate from the background model used by the other systems. There are two essential differences between the systems that forces this to be the case. The first reason is that the system works a lot better when the frustum is as large as possible. In order to not confuse obstacles and touches, the algorithm starts all the way at the arm and works its way down, so having as much space in frame as possible even besides the table, aides the system in detecting the touches even at the

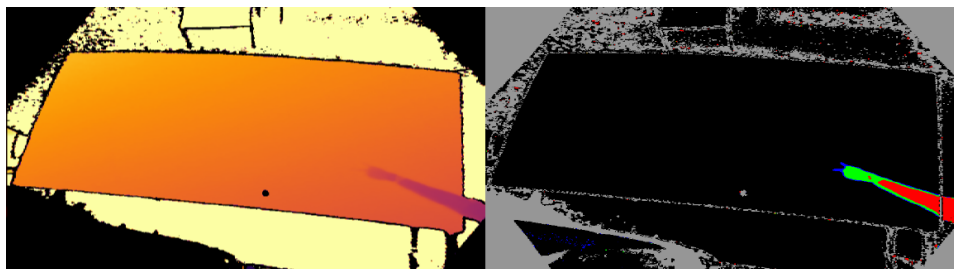


Figure 4.15: *Height zones for each pixel. Black pixels represent the background, the low, medium and high zones are represented by blue, green and red pixels respectively and invalid regions are gray.*

edges of the table. The second reason this background model is different is because of its need to be dynamic. The tabletop is constantly changing by users placing new obstacles and the tops of these obstacles need to act as the new surface. Trying to touch the table in the shadow of an obstacle will not work, as there is no way to detect what is happening there. This is one of the reason why minimizing the length of shadows, as discussed in Section 4.1, is important for smooth interaction.

The background model used in the DIRECT is based on WorldKit[30]. It uses a statistical approach and thus stores the mean distance and a standard deviation for each pixel. This way of modeling the background gives a more stable image, with higher confidence on the table itself. The noise stemming from the edge invalidations discussed in Section 4.2.4 however, breaks this model. An invalidated pixel is set to a distance value of zero and the distance to the table edge is around two meters. This results in mean distance somewhere above the actual surface and a large standard deviation for all pixels in the edge noise region. The *diff image* construction invalidates all pixels that lie lower than $10mm$ beneath their mean and all consistently invalid pixels during the background initialization. In practice, this results in a thin invalid border around the table. This invalidation border is later holefilled both when trying to create the edge map and when an arm is detected near the errors. This hole filling function is rather slow and imperfect when large holes exist however, making touch detection near many invalidated pixels more difficult.

The actual touch detection starts at the arm. The algorithm selects the first pixel from zone high, Figure 4.15B, and looks for all neighbors, eventually forming a continuous blob of all connected pixels from the high and medium zones. Should this blob turn out to be large enough to represent



Figure 4.16: *Only the detected arms are shown. Arms are cyan and everything below that is darker blue.*

an arm, the system starts to look for hands connected to this in the medium and low zones. The size thresholds for these blobs can be configured, but they are set liberally to exclude as few valid touches as possible. Once a hand blob is completed, the individual fingers are flood filled; they are still visible on the depth image. The fingertips lie below the noise threshold, so detecting them can be harder. The main method employs the IR camera. It attempts to extract the fingertip by taking the furthest tip from a connected edge from the finger base, which is the place where it connects to the hand. Should this fail, the fallback option guesses the location of the fingertip by projecting a point forward from the finger base. The result of the flood fill is shown in Figure 4.16. This image clearly shows the fingertips. To judge if the finger is actually touching the table, or just hovering above it, the system checks if there are any pixels higher than a centimeter directly neighboring the touch point.

4.5 Fiducial marker tracking

To enable the user to place entries and exits in the simulation without having to interact with the computer, we have implemented fiducial marker tracking. The *ArUco* library in OpenCV [25] uses binary square fiducial markers. The inner binary codification makes them specially robust, which is important to ensure the flow of agents is not interrupted. Algorithm 4 shows how the markers are detected and coupled to the simulation. The detection is handled by the *ArUco* library and the placement and connection of the markers is handled in the *uCrowds* framework.

Algorithm 4: Marker detection: *frame*

```
Result: List<(int, Point)> markers
// Get the smallest dictionary possible.
dictionary = GetMarkerDictionary(Dict_4x4_50);
while running do
    colorImage = GetWarpedColorFrame(frame);
    DetectMarkers(colorImage, dictionary, out ids, out corners);
    markers = ConvertToCenterPoints(ids, corners);
    for each marker in markers do
        age[marker] = 0;
        if markersOnScreen.Contains(marker) then
            | MoveExistingMarker(marker);
        else
            | markersOnScreen.Add(marker);
            | PlaceNewMarker(marker);
        end
    end
    for each marker in markersOnScreen do
        | if age[marker]++ > maxAge then
        | | RemoveOldMarker(marker);
        | end
    end
    for i = 0; i < markersOnScreen.Count; i+ = 2 do
        | if MarkerDetected(i) && MarkerDetected(i+1) then
        | | ConnectMarkers(marker[i], marker[i+1]);
        | end
    end
end
```

Chapter 5

Implementation

5.1 System architecture

The CrowdAR system has a number of strict requirements that follow from it having to be able to operate unsupervised in a museum setting for hours on end.

To achieve these goals we have to answer the question of how to design a system such that it can maintain robust real-time comprehensive interactivity and how do we ensure that it remains only in valid states? To be able to reason over the program execution, it helps to define a strict program flow. The underlying system of crowdAR can be quite neatly broken up into a number of states. The main sections of the execution consist of **initialization**, **calibration** and **operation**. These sections are further subdivided in a number of processes. The flowchart describing the program execution of CrowdAR is shown in Figure 5.1.

5.1.1 Plugins

CrowdAR is developed by using the *Unity* game engine, therefore it is written using Unity's version of C#. It provides an interactive interface for a museum setting for the uCrowds framework. CrowdAR makes use of a number of plugins to operate. **OpenCV for Unity**[25] is almost a necessity when working with real-time image processing. Moreover, this version of OpenCV is optimized for conversion between data types supported by the Java version and the Unity engine. For further development of CrowdAR within Unity, the DLL for this plugin needs to be installed in the user's system folder. To use the system with the *Intel Realsense*, the **RealSenseSDK**

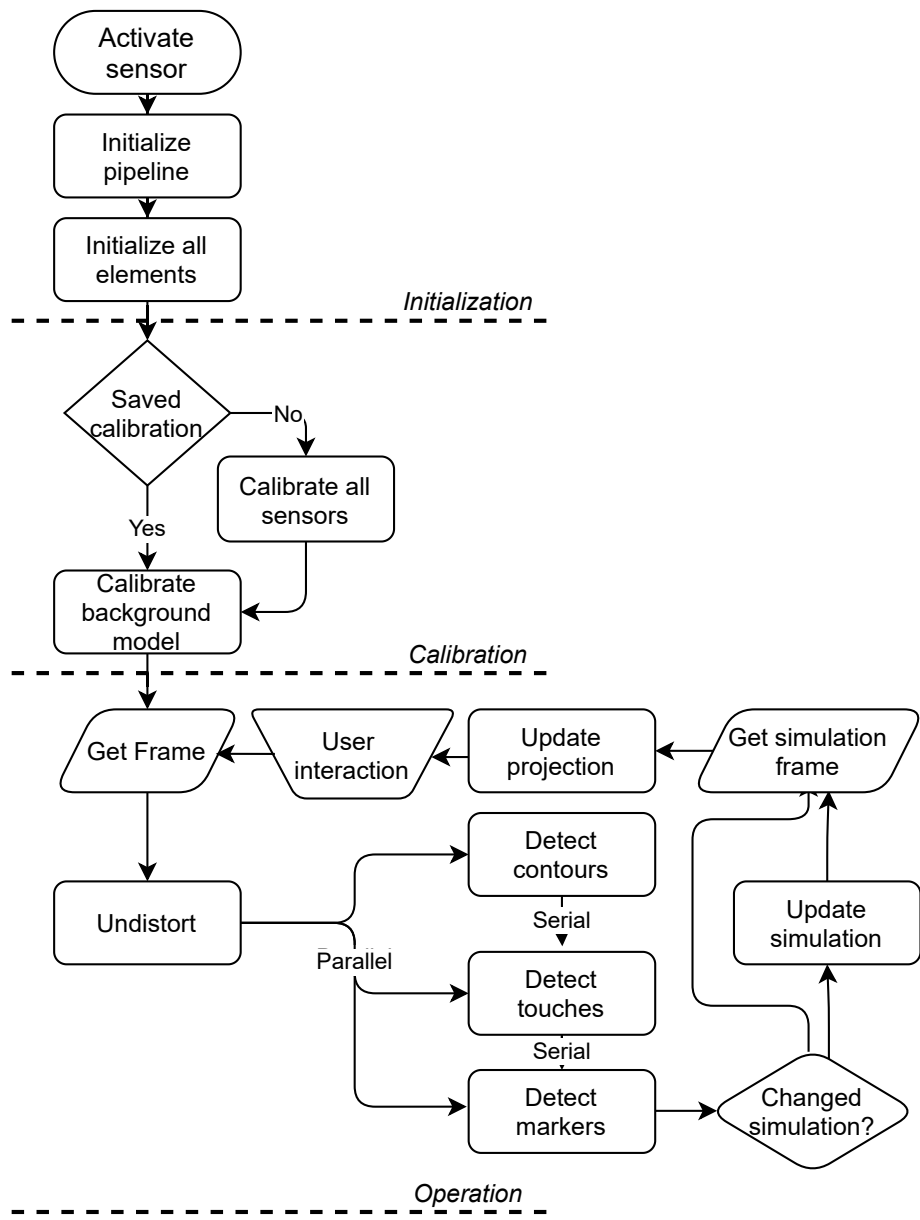


Figure 5.1: Flowchart describing the program execution of CrowdAR

is obligatory and the same goes for the *Azure Kinect*, for which **K4A** is the software development kit. The last plugin we use is called **Serializable Dictionary**. This plugin is not strictly necessary but it aids in constructing clear and structured mutable configuration settings for the sensor in the Unity editor.

5.1.2 Data structures

Efficient and robust handling of *frames* is central to our system. The Azure Kinect can generate these frames, consisting of three images of varying types and resolutions thirty times per second. Aside from processing this data, the computer also requires resources to handle interactivity and to run the crowd simulation itself. It is therefore important to use processing power as efficiently as possible.

The data created by each sensor is held in a sensor specific *FrameData* object. This object contains all specifications for the frame, as well as all its raw and, if necessary, processed data. If the raw data for a specific input channel does not have to be manipulated this frame, it can be converted directly to a Unity texture, skipping the processing pipeline entirely.

The main execution loop of the program is contained within the *Vision Pipeline*. This pipeline is created and updated through the *Vision Manager*. The pipeline and the frame capturing from the sensor run on different threads so they are non-blocking. Modular *Pipeline elements* make up the systems of CrowdAR as shown in Figure 5.1. Calibration, background initialization, contour detection and multi touch are all part of it. These modular generic elements make the program more easily extensible.

5.1.3 Initialization

The initialization steps of the program are mainly contained in the *Vision Manager*. The first step of the flowcharts completes when exactly one sensor exists. Because we can only handle one sensor, all other sensors are removed once one is initialized. The pipeline is built and initialized using preset values, such as the size of the tabletop and the resolution of the projector as well as the camera matrix and distortion coefficients determined beforehand. Once these values have been found, all pipeline elements are initialized. This initialization adds all enabled modules to the pipeline, while loading all required data if there is any.

5.1.4 Calibration

The calibration stage of the program takes place during the preprocessing of the vision pipeline. As the diagram shows, the sensor calibration can be skipped, because the system will try to load saved calibration settings containing the tabletop corner coordinates. Recalibration can be forced if necessary or otherwise the faulty calibration file can be removed to achieve the same effect. The sensor calibration process polls for frames as needed. It also corrects for barrel distortion before determining the corner points of the table. If the projector calibration is done on startup, the background model for the multi-touch interaction has to be rebuilt for touch recognition to be accurate immediately. It is possible to keep a rolling, constantly updating, background model for the touch interaction, making this calibration not strictly necessary. In normal operating conditions, the number of iterations n can be very low, as the only reason for generating more than one set of corner point data is to eliminate outliers. At the end of this process, a confirmation dialog is shown to the user. Should the corner detection have failed, the user can initiate a rerun or manual override, where they can select the corners by hand. This override is hardly ever necessary during normal conditions, as the system runs in about a second and is quite robust.

5.1.5 Operation

As mentioned before, the polling for new frames from the sensor happens on a separate thread from this main operation loop. This makes it possible to display the video feeds from the color, depth and infrared cameras while bypassing the pipeline. Not converting this data to OpenCV's matrices before transforming them to textures speeds up the program. A desired maximum number of frames per second can be specified. The loop will thus only perform an iteration if $t = \frac{1}{fps}$ seconds have passed and if a new frame is available.

Mitigating the barrel distortion is done before the interaction modules. This is only necessary for the infrared and depth frames, as the color does not suffer from it. If the perspective needs to be warped to only include the table surface for the module, that transformation is done in the module itself.

To enable the user to place entries and exits in the simulation without having to interact with the computer, we have implemented fiducial marker tracking. Figure 5.2 shows an example of these markers. When a pair of markers is placed on the table, SimCrowds connects them and starts to sim-

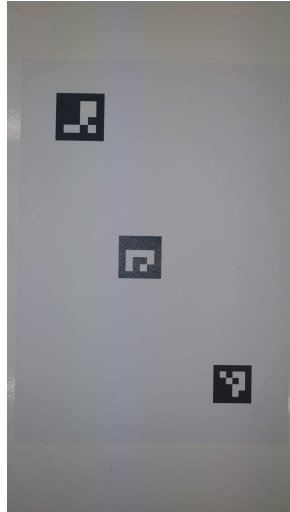


Figure 5.2: *Fiducial Aruco markers using a 4x4 grid.*

ulate a stream of people moving from the entrance to the exit. After the interactivity has been handled, the acquired data is sent to the uCrowds framework if new touches are detected or obstacles have been changed, so it can update its *navigation mesh* for path planning. After which, the new simulation frame is projected back onto the tabletop.

Chapter 6

Validation

To validate the results of the CrowdAR system, we can again split it up into its component parts. This method is useful, since the individual components are more easily validated, than the holistic system. Dependencies should not be an issue when the order of the pipeline is kept in mind, as a pipeline element can only be validated once all its dependencies have been deemed sufficiently accurate. These relationships are visualized in Figure 6.1.

6.1 Validate camera calibration

The two camera calibration steps are validated in the same manner. As described in the calibration chapter, both cameras are calibrated similarly in a two-stage algorithm. The first stage attempts to correct for the barrel distortion caused by the increasing distances of the table to the lens the further to the edge we go. The second stage detects the corners of the table surface by looking for the largest continuous area within the viewing frustum of the camera.

To validate the system's camera calibration, an image is produced conveying the results. This image consists of four vertices, the surface's corners, and four edges, which should lay precisely on the sides of the table surface. The highest precision the system can achieve is determined by the relation between the camera's resolution and its distance to any table surface, which is expressed by the width of the table w over the number of pixels r calculated by the formula in part 4, meaning that the pixel size p is described by:

$$p = \frac{w}{r} = \frac{w}{\frac{wx}{2(\tan(\frac{a}{2}) * h)}} = \frac{2h * \tan(\frac{a}{2})}{x}$$

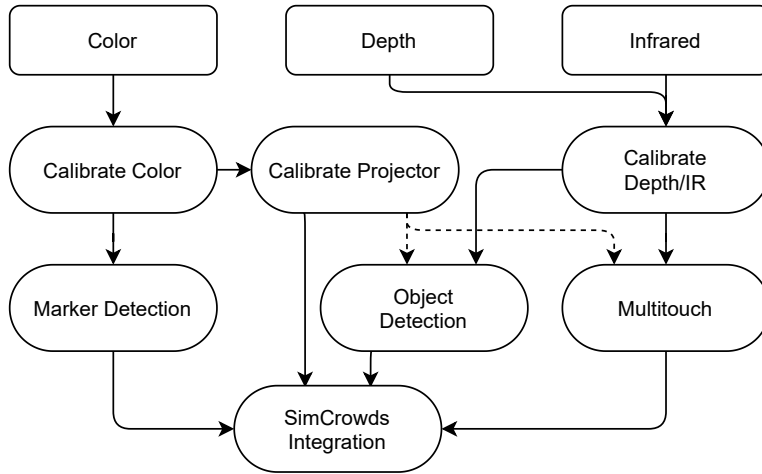


Figure 6.1: *Validation order, each element needs all solid connections to function and dashed connection to be validated.*

Camera Height	Color Camera	NFOV IR
1.6m	1.67mm	3.84mm
1.7m	1.77mm	4.08mm
1.8m	1.88mm	4.32mm
1.9m	1.98mm	4.56mm
2.0m	2.08mm	4.80mm

Table 6.1: *Pixel width at a selection of reasonable camera heights. Notice these sizes increase linearly.*

The pixel sizes p represent the best possible precision and they increase linearly with respect to the camera height and are independent of the size of the table. A sample of the values are shown in Table 6.1.

6.1.1 Calibration errors

The calibration validation image can catch five different types of errors: **curved edges**, **invisible tabletop**, **wrong contour**, **frustum too small** and **misplaced corners**.

Curved edges happen when the first calibration stage, correcting for the radial distortion, has gone wrong. This can have two underlying causes, either the *distortion coefficients* were not calculated accurately or the camera

has moved with regards to the table. The accuracy of the distortion coefficients can be estimated by using the re-projecting error. The closer this error is to zero, the more accurate the parameters are. If the calibration was accurate and precise, but the edges do not line up well, it means that the camera has moved with regards to the surface since the last calibration.

An **invisible tabletop** happens usually to the infrared camera when the image is saturated by the sun or fluorescent lights. Closing the blinds and turning of the lights or making sure the surface of the tabletop is less reflective both solve this issue.

A **wrong contour** can be selected from the image by both the color and infrared cameras. The calibration will always select the largest complete contour it can find. When the system selects a different contour it can either mean that there exists a larger contour within the frustum, or that the table surface does not form a complete contour. When the latter is the case, it can in turn mean two things. Either the view to the table is obstructed or the table lacks contrast with regards to the background.

When the **frustum is too small**, the camera is mounted too closely to the table surface. If it can not be moved further away, the WFOV IR mode can be used. Correcting for the large radial distortion using the WFOV mode is essential.

Misplaced corners can be caused by defects in the table surface itself or more local disturbances due to the background light.

6.2 Validate projector calibration

The maximum achievable precision is determined by the smallest resolution between the color camera and the projector. In the case of CrowdAR, the resolution is limited by the precision of the color camera, shown in Table 6.1, since we are forced to mount the camera at a sufficient height for the IR camera to capture the entire surface. This results in a maximum precision of around $2mm$.

The calibration of the projector relies on a well calibrated color camera, since in order to translate the valid *game space* to a valid *projector space*, the game space needs to exist first.

Figure 6.2 shows the projector validation image. This image is warped and then projected, it should line up perfectly with the table top if the calibration was successful.

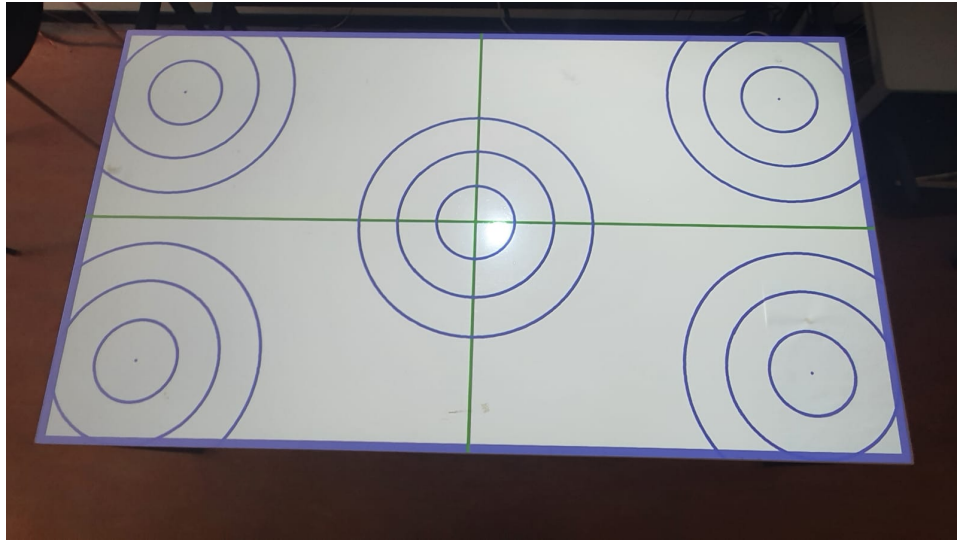


Figure 6.2: *The projection validation image consists of a 2cm thick outer border, five sets of circles and a 1cm thick inner cross.*

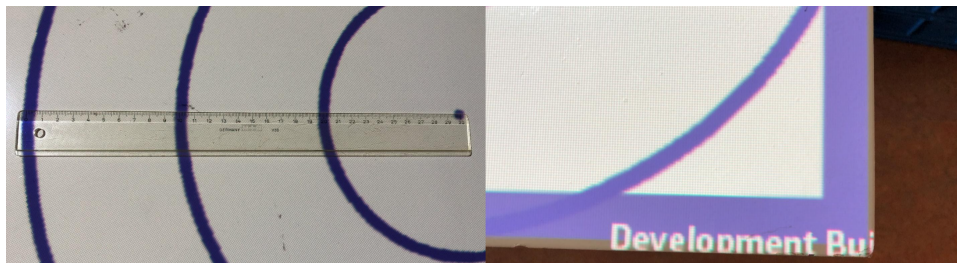


Figure 6.3: *A: (left) The lines should be 10cm apart and 1cm thick, with the error margins of table 6.1. B: (right) A slight defect in the surface trips the algorithm. A smooth and solid colored surface near the table edges is necessary for accurate automatic calibration.*

6.2.1 Calibration errors

The most common error during the calibration process is erratic behavior from the line in the angle finder algorithm. When this line fails increasingly quickly, it means that the delay between the projector, camera and computer is too great for the system to be able to recognize the line from the newly projected image. This problem is mitigated by delaying the code execution before it has to project and find the new line on the table surface. The delays in this step are what take up the most time during the entire calibration, so striking a good balance between rapid execution and robust calibration influences the calibration time by a lot.

Figure 6.3B shows that sometimes the validation image misses a corner and makes a mistake either by drawing it somewhere on the table or by estimating the corner to be on the floor. This indicates that the scene is not lit well enough by the projector, the surface is inconsistent or the contrast variable for the edge detection needs to be changed.

When the round shapes projected by the validation image are not circles, the lines are not straight and the edges are not of a consistent width, the radial distortion has most likely not been corrected for correctly.

6.3 Validate Contour Detection

The interactive contour detection algorithm is validated by projecting the detected contours back at the table. The depth camera calibration is needed for the contour detection itself to work and the projector for validation. If the projected lines fit cleanly on the objects placed on the table, the object detection is correct. Since the contours are slightly simplified for performance reasons, very irregular objects are not supposed to have a perfect fit.

Because of the slight difference in shadows caused by the different mounting positions of the projector and camera, the contours of the objects can have a worse fit depending on distance from the center and object height. An example of this error is shown in Figure 6.4.

6.3.1 Operation errors

Noise can be subdivided in multiple different categories: **Edge Noise**, **Random Noise** and **Structured Noise**. **Edge noise** is caused by the *Azure Kinect* invalidating pixels near edges, generating noise depending on reflection and sharpness of the edge. This noise can be disregarded by not

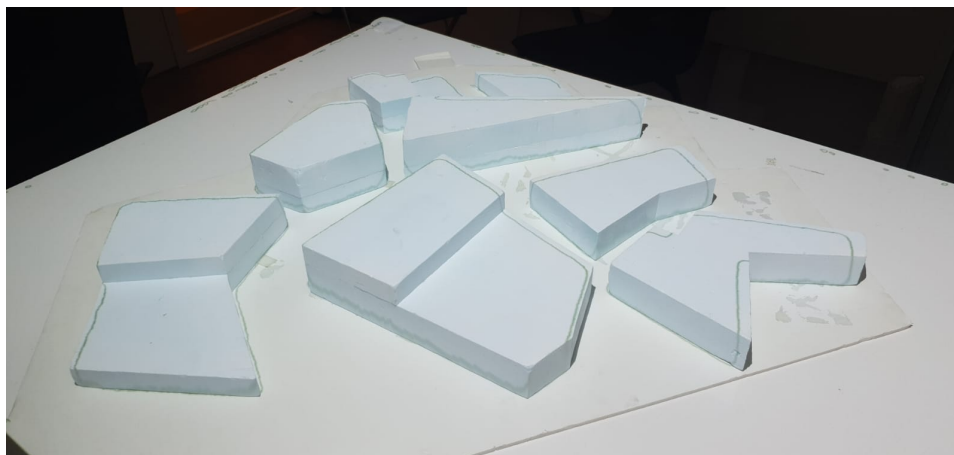


Figure 6.4: *Imperfect alignment and noticeable edge noise.*

taking the outermost pixels into account. Figure 6.4 shows an example where this noise is not filtered out properly. Even though this noise would create only very small disturbances to the pathfinding if not filtered out correctly, it would force the engine to recalculate the navigation mesh every update, freezing the simulation.

Random noise can be caused by anything and is, importantly, irregular. The *intersection filter* and the minimum size requirement filters out this random noise. Taking the intersection of too many frames leads to erosion of the image and chunks of objects that have just stopped moving missing. A minimum size near the size of the smallest object will lead to disregarding that object if only a small part of it is not properly detected, so a balance needs to be found.

Structured Noise, meaning large disturbances that behave predictably, is filtered out by not counting cells that have too much movement in them. This stops people moving or pointing from bringing the simulation to a halt. If this system is too sensitive, the object detection grinds to a halt.

6.4 Validate Multi-touch

The goal of the multi-touch system is to accurately, precisely and quickly determine whether and where a user has touched the surface. The simplest useful validation consists of projecting markers on the surface where active touches have been registered. This method requires the successful calibration of the projector. This technique gives an intuitive sense of the



Figure 6.5: *The multi-touch simple validation method showing a misplaced touch.*

responsiveness and validity of the system. This simple method is only useful when the system is working as intended however, as it can not be used to diagnose faults. To fill this gap, a more complete validation view exists as well. This view shows all invalid pixels and the classification for all pixels which are under consideration. Using this classification, the operation errors can be identified and classified into the categories below.

6.4.1 Operation errors

The simple projection validation method can be used to check if there are **false touches**, **missed touches**, **misplaced touches** and **merged touches**.

False touches are false positives. Random false touches are rare, in fact I have never seen one during at all during development, because *DIRECT* looks for a human arm structure to determine if the surface has been touched. Should random touches occur on the surface, it means that the minimum limb size is set too small. False touches appearing under an arm or hand can mean that *zone height conditions* are set incorrectly. Touches should only appear when the surface is touched or almost touched.

Merged touches happen when the vision algorithm can not distinguish between two or more distinct fingertips. To mitigate this problem, make sure

to spread your fingertips by approximately 2cm . Also keep the angle of the camera relative to your hand in mind. When the camera is perpendicular to the hand, it is better able to distinguish between fingers.

Missed touches can not be diagnosed using the simple validation method, since there is no way of knowing where the issue would lie. However, the validation view could give some insight. If large patches of the image are classified as invalid, the background initialization has gone wrong, it is important to set the maximum distance to include the corners of the table. If the flood-fill stops at a certain point, the size bounds are probably too strict and if the touch itself is not registered, make sure that there is a large enough gap between *Zone Low* and *Zone Noise*.

Misplaced touches are better spotted using the projection method, as shown in Figure 6.5. A small circle should be projected around the touching fingertip, but if the circle is projected further up the hand, it most likely means that the fingertip was detected smaller than the minimum size specified, while the other size and distance minima were generous enough to include places higher up the hand. This only happens when the IR edge detection fails and the depth only fall back method is used. Stricter bounds or increasing the angle of the finger with regards to the camera should help alleviate this problem.

6.5 Validate SimCrowds Integration

The integration of the CrowdAR table with SimCrowds is dependent on the CrowdAR vision pipeline running independently from simulation loop. The subsystems should then function in the same manner as in the isolated environment, which has already been validated before. Figure 6.6 shows the simulation with marker detection for the agents and contour detection for the obstacles. The most important part of this linking is that the absolute minimum number of new contours are sent, as a perpetually updating navigation mesh is lethal to the smoothness of the simulation. To make sure this is the case, a degree of mismatch between the objects on the table and the simulated obstacles is allowed. An object is deemed to be similar enough to its counterpart if their areas are within a certain percentage of each other and their bounding boxes overlap by at least a certain amount. Only if all objects stay between these margins between updates, then the environment does not need to be recomputed. Otherwise the entire mesh and paths are recalculated.

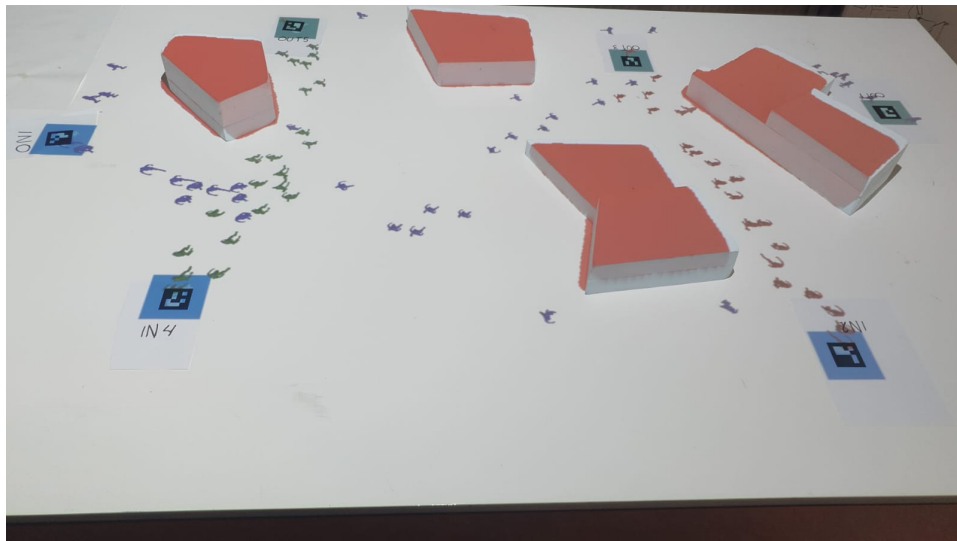


Figure 6.6: *Integration including object and marker detection*

Chapter 7

Conclusion

We have presented CrowdAR, a real-time, interactive, augmented-reality tabletop crowd simulation. The installation consists of a depth, infrared and color camera, a projector, a computer and the table on which the crowd simulation is projected. As well as objects to act as obstacles and binary square fiducial markers to act as entrances and exits for the simulated people. Since this installation is to be an interactive museum exhibit, it is important to make the *tangible user interface* intuitive, as the target demographic does not consist of domain experts. This is done by relying on markerless object detection using spatial AR as the basis of interacting with the simulation. The underlying crowd simulation software used is SimCrowds, a real-time, interactive simulation tool that can dynamically deal with changing environments. Our *vision pipeline* runs through its set of subsystems enabling interaction and displaying the simulation on the table surface. It has two main different states, *calibration* and *operation*. The former handles camera calibration using the computer vision library *OpenCV* [25] as described in Chapter 4.2 and the latter enables contour detection, Chapter 4.3, and multi-touch in Chapter 4.4. The contour detection subsystem lets users place objects on the table, acting as obstacles in the simulation. Updating uCrowds' underlying *navigation mesh* is an expensive operation, so in order to not let the tool stall, the table is subdivided in cells and obstacles are only added to the simulation if they have been substantially changed and once movement in affected cells has died down. The algorithm used for the multi-touch interaction is *DIRECT* [31], which combines depth and infrared images for more precise and robust touch interaction, enabling multi-touch and touch tracking. The infrared images increase precision, while the depth image can always provide a less precise touch point, so the infrared edge

map can gracefully fail.

Validation of the system is done from the bottom up, as the subsystems have stacking dependencies. We have provided bounds within which our system can robustly link the *camera space* of the *Azure Kinect* [6] with the *game space* of the simulation and the *projection space* of the projector. We have created a robust contour detection system and provided best practices for object sizes and materials. The noise mitigation methods enable multiple users to interact with the table simultaneously without negatively influencing the experience for users at the other side of the table. Non-overlapping shadows of the camera and projector hinder the precise validity of the contours, which does not influence the path finding.

In the future, it would be interesting to implement an automatic projection mapping functionality to the system to dynamically project roofs on obstacles and roads or parks on the walkable area for example. This could be achieved by either considering stereo vision, so the system would be less influenced by shadows or extracting the best guess of the object shapes from point clouds. Extending the control users can exert using the markers would enable a more engaging user experience, doing this without adding a disproportional amount of complexity for the user would be an interesting challenge. Adding a display on which users can see the situation on the ground could aid in the immersion of the exhibit. In order for this to work, the obstacles need to be rendered in the full three dimensions instead to the areas we use now.

Bibliography

- [1] Ar sandbox. <https://arsandbox.ucdavis.edu/>. [Online; accessed 15-03-2021].
- [2] Geofort. geofort.nl. [Online; accessed 15-03-2021].
- [3] Hp sprout. <https://www8.hp.com/us/en/sprout/features.html>. [Online; accessed 15-03-2021].
- [4] Kinect for windows. <https://developer.microsoft.com/en-us/windows/kinect/>. Accessed: 2020-02-17.
- [5] Universiteits museum utrecht. <https://umu.nl/over-umu/>. [Online; accessed 07-07-2021].
- [6] Azure kinect dk. <https://azure.microsoft.com/en-us/services/kinect-dk/>, 2020. [Online; accessed 15-03-2021].
- [7] AINSWORTH, S. The functions of multiple representations. *Computers & education* 33, 2-3 (1999), 131–152.
- [8] AMBURN, C. R., VEY, N. L., BOYCE, M. W., AND MIZE, J. R. The augmented reality sandtable (ares). Tech. rep., 2015.
- [9] BORTOLASO, C., GRAHAM, N., SCOTT, S. D., OSKAMP, M., BROWN, D., AND PORTER, L. Design of a multi-touch tabletop for simulation-based training. Tech. rep.
- [10] DE VILLIERS, J. P., LEUSCHNER, F. W., AND GELDENHUYS, R. Centi-pixel accurate real-time inverse distortion correction. In *Optomechatronic Technologies 2008* (2008), vol. 7266, International Society for Optics and Photonics, p. 726611.
- [11] GERAERTS, R. SimCrowds. <https://www.ucrowds.com/>, 2021. [Online; accessed 15-03-2021].

- [12] HÜRST, W., AND GERAERTS, R. Augmented and virtual reality interfaces for crowd simulation software—a position statement for research on use-case-dependent interaction. In *2019 IEEE Virtual Humans and Crowds for Immersive Environments (VHCIE)* (2019), IEEE, pp. 1–3.
- [13] HÜRST, W., GERAERTS, R., AND ZHAO, Y. Crowdar table—an ar table for interactive crowd simulation. In *2019 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)* (2019), IEEE, pp. 302–3023.
- [14] JERMANN, P., ZUFFEREY, G., SCHNEIDER, B., LUCCI, A., LÉPINE, S., AND DILLENBOURG, P. Physical space and division of labor around a tabletop tangible simulation. In *Proceedings of the 9th international conference on Computer supported collaborative learning-Volume 1* (2009), pp. 345–349.
- [15] KOBAYASHI, K., NARITA, A., HIRANO, M., TANAKA, K., KATADA, T., AND KUWASAWA, N. DIGTable: A Tabletop Simulation System for Disaster Education. 4.
- [16] LEE, S., AND SHIM, H. Skewed stereo time-of-flight camera for translucent object imaging. *Image and Vision Computing* 43 (2015), 27–38.
- [17] MA, J., SINDORF, L., LIAO, I., AND FRAZIER, J. Using a tangible versus a multi-touch graphical user interface to support data exploration at a museum exhibit. In *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction* (2015), pp. 33–40.
- [18] MOORE, A., DANIEL, B., LEONARD, G., REGENBRECHT, H., RODDA, J., BAKER, L., RYAN, R., AND MILLS, S. Comparative usability of an augmented reality sandtable and 3d gis for education. *International Journal of Geographical Information Science* 34, 2 (2020), 229–250.
- [19] PERETO, S., AND AGOTAI, D. Review on methods in touch tracking for tabletop projections. In *International Conference on Human-Computer Interaction* (2020), Springer, pp. 285–293.
- [20] PETRASOVA, A., HARMON, B., PETRAS, V., AND MITASOVA, H. Gis-based environmental modeling with tangible interaction and dynamic visualization. In *Proceedings of the 7th International Congress on Environmental Modelling and Software, San Diego, CA, USA* (2014), pp. 15–19.

- [21] RATTI, C., WANG, Y., ISHII, H., PIPER, B., AND FRENCHMAN, D. Tangible user interfaces (tuis): a novel paradigm for gis. *Transactions in GIS* 8, 4 (2004), 407–421.
- [22] SAVENIJE, N., GERAERTS, R., AND HÜRST, W. Crowdar table an ar system for real-time interactive crowd simulation. In *2020 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)* (2020), IEEE, pp. 57–59.
- [23] SCHNEIDER, B., JERMANN, P., ZUFFEREY, G., AND DILLENBOURG, P. Benefits of a tangible interface for collaborative learning and interaction. *IEEE Transactions on Learning Technologies* 4, 3 (2010), 222–232.
- [24] SCHNEIDER, B., WALLACE, J., BLIKSTEIN, P., AND PEA, R. Preparing for future learning with a tangible user interface: the case of neuroscience. *IEEE Transactions on Learning Technologies* 6, 2 (2013), 117–129.
- [25] SOFTWARE, E. Opencv for unity: Integration: Unity asset store, Dec 2020.
- [26] TATEOSIAN, L., MITASOVA, H., HARMON, B., FOGLEMAN, B., WEAVER, K., AND HARMON, R. Tangeoms: Tangible geospatial modeling system. *IEEE transactions on visualization and computer graphics* 16, 6 (2010), 1605–1612.
- [27] VAN TOLL, W., JAKLIN, N., AND GERAERTS, R. Towards believable crowds: A generic multi-level framework for agent navigation.
- [28] VAN TOLL, W. G., COOK, A., VAN KREVELD, M., AND GERAERTS, R. The explicit corridor map: Using the medial axis for real-time path planning and crowd simulation.
- [29] XIAO, R. DIRECT algorithm. <https://github.com/nneonneo/direct-handtracking/>, 2015. [Online; accessed 08-01-2021].
- [30] XIAO, R., HARRISON, C., AND HUDSON, S. E. Worldkit: rapid and easy creation of ad-hoc interactive applications on everyday surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2013), pp. 879–888.

- [31] XIAO, R., HUDSON, S., AND HARRISON, C. Direct: Making touch tracking on ordinary surfaces practical with hybrid depth-infrared sensing. In *Proceedings of the 2016 ACM International Conference on Interactive Surfaces and Spaces* (2016), pp. 85–94.
- [32] ZHENG, F., AND LI, H. Arcrowd-a tangible interface for interactive crowd simulation. In *Proceedings of the 16th international conference on Intelligent user interfaces* (2011), pp. 427–430.
- [33] ZUFFEREY, G., JERMANN, P., LUCCHI, A., AND DILLENBOURG, P. Tinkersheets: using paper forms to control and visualize tangible simulations. In *Proceedings of the 3rd international Conference on Tangible and Embedded interaction* (2009), pp. 377–384.