ProxemicUI: Iterative Design and Evaluation of a Flexible and Generic Framework for Proxemics Aware Applications

By

Mohammed Abdulhamid Alnusayri

Submitted in partial fulfillment of the requirements
For the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia

To my father and mother for their endless love, support and inspiration. To my wife for her love, patient and support, I dedicate this work.

Table of Contents

List of Tables

List of Figures

Abstract

Systems that respond to spatial configurations of people, devices, and objects (sometimes called proxemics-aware systems) have long been explored in HCI research and are now finding commercial applications. While state-of-the-art tracking systems and toolkits have been demonstrated the ability to provide proxemics data, there is still a considerable gap between what existing systems support and what the developers require to define their own custom proxemics events (e.g., to indicate that complex spatial configurations are meaningful) rather than conduct manual tests in the UI layer after low-level events are triggered. In addition, proximity-related events are often intertwined with other events, such as interaction with application interfaces and notifications from smart appliances. While machine learning techniques might detect complex spatial configurations, more research is needed to determine what kinds of input data are required to train an accurate and adaptable classifier. In response to these issues, I developed ProxemicUI, an object-oriented framework that provides a simple but powerful rule-based format for expressing proxemics relationships, an event-driven model for responding to their occurrence, a mechanism for integrating non-proxemics data and events, and extensibility through clearly articulated extension points. I evaluate ProxemicUI through a set of proof-of-concept applications, a comparative code review study with 18 programmers, and the integration of ProxemicUI into an augmented reality storytelling tool, working with three authors and three developers. The results demonstrate that ProxemicUI supports rapid prototyping of systems that can respond to complex proxemics events across various contexts.

List of Abbreviations Used

HCI	Human Computer Interaction
DT-DT	Dal Top-Down Tracker
PSI	Platform for Situated Intelligence
OSC	Open Sound Control
UI	User Interface
API	Application Programming Interface
ML	Machine Learning

Acknowledgements

My unreserved gratitude and praises are for Allah, the Most Compassionate, and the Most Merciful. He blessed me with his bounties, and he has given me the strength and courage to reach my goals during the course of this research.

I would like to thank my supervisor, Dr. Derek Reilly, for his guidance, help and support throughout my research. I also appreciate his patient encouragement and valuable advice over the course of my entire Ph.D. program.

I would also like to thank my committee members Dr. Israat Haque and Dr. Sageev Oore who have given their time and expertise improve my work. Special thanks to Dr. Nicolai Marquardt for accepting to be my external examiner and for his insightful questions and suggestions which improved my thesis.

I would also like to thank Dr. Joseph Malloch for numerous fruitful discussions and useful suggestions for my work. Thank you, Abbey Singh, Shannon Frederick for helping running the Story CreatAR study; thanks also goes the rest of the Story CreatAR project team and to my colleagues at the Graphics and Experiential Media (GEM) Lab.

I would like to thank Jouf University, for providing me a scholarship to complete my studies at the Dalhousie University.

Finally, I would like to express my sincere love and gratitude to my father Abdulhamid and my mother Kawlah. I owe them more than anyone could possibly imagine. I thank them for their love and infinite care, which has made the person I am today. My sincere love and gratitude to my wife Ayat who supported through the course of my Ph.D. program, thank you for being patient and supportive.

Chapter 1: Introduction

Ubiquitous Computing (“ubicomp”) often involves connecting multiple devices such that they are aware of each other and the surrounding environment [38][126]. In other words, digital devices (e.g., smartphones, smartwatches, interactive displays) and their users maintain awareness of each other within an environment. They also might be aware of the affordances and constraints of the environmental context in which they are situated, even to the level of the location, shape, and size of non-digital objects such as sofas and coffee tables. This awareness allows devices and services to provide a better experience to users in the space. The information that is shared to achieve this awareness can include (but is not limited to) identity, position, orientation, status, type, and shape. Some entities in “smart” environments have built-in sensors and can provide some of this data themselves (e.g., through the inertial measurement units (IMUs), dedicated processing chips, and relevant application programmer interfaces (APIs) found in smartphones). Other entities require external tracking or sensing systems to capture this information (e.g., skeletal tracking using Microsoft Kinect [59], positional and activity tracking via a top-down tracker like DT-DT [49], or attaching physical trackers (VICON [120] and VIVE [122]) to non-digital objects). To employ this information effectively, we need first to understand the *environmental context* for the applications that we are developing. Das et al. [27] and Volpentesta [124] explore smart environment contextual factors, including participating entities and their proprieties, relationships between these entities, and the physical layout of the environment. We also need to understand the types of tasks to be performed, the interaction techniques available, the range of possible system responses, and the roles and identities of all participating entities.

When performing tasks using interactive systems, there are preferred or implied relationships between individuals and/or devices we use. In particular, multi-user interactive systems can be characterized into three types based on these relations. First, *collaborative use*, e.g., SIDE [86], a tabletop game for group therapy, Lucero et al.'s work to explore photo collections using multiple mobile phones [68], and teamwork when interacting with CityWall [83], a multi-touch wall display to navigate media. Second, *independent use*, e.g., the museum exploration station [60], the public advertising interactive display [125], and the parallel use of CityWall [83]. Third, *mixed-use*, where users can work independently on a sub-task then contribute to the group's main task (e.g., the RoomPlanner [129], a room furniture layout application and the Proxemic Brainstorming application [70]). By considering the full range of multi-user applications, we can see the presence of three types of *task coupling* [127]. First, *tightly coupled* tasks require working together to complete the task (e.g., tabletop collaborative games, as seen in Futura [8] and designing classrooms together, as seen in [91]). Second, *lightly coupled* tasks involve working independently to complete a higher-level task (e.g., room furniture layout, as seen in RoomPlanner [129] and a mixed-focus collaborative application, as seen in PiVOT [57]). Finally, *uncoupled* tasks involve working independently on tasks that are not directly related (e.g., museum exploration station [60] and MyAppCorner [34]). These coupling types are often reflected in device configurations, and developers could utilize proxemics in useful ways to do so. We might distribute a task between two devices, tightly or loosely coupling subtasks based on device proximity. For example, if we placed a camera in the corner of a room, it will capture the activities in the whole space. When we add a second camera in the other corner, each camera will rotate and zoom in automatically to capture only half the space. For uncoupled tasks, separate devices can be responsible for supporting different tasks. When the proxemic relationships between entities reflect the environmental context of a space, task coupling is facilitated by physical proximity.

With task coupling, multiple users interact with the system simultaneously, where each user's role might change over time. Several abstract roles have been identified in the literature that pertains to an individual's relation toward work being done on the interactive display: direct participant (directly interacts with the system), active observer (actively participate in the task without direct interaction with the system), passive observer (participate in a subset of the activities only), and disengaged bystander (completely disengaged from the task) [60][112]. We can extend these roles to devices as well. For example, we might consider the device that controls all other devices or initiates an interaction in a smart environment, a *direct participant* (e.g., Proximity-Aware control [64]). A device that receives commands, and as a part of its response, communicates with other devices in the environment might be considered an *active observer*. For example, a coffee machine is an active observer that receives a command to start preparing the coffee, and then, it might produce a notification on a phone or TV that the coffee is ready. Finally, a passive observer might be the device that receives a command then generates a response (e.g., show notifications on a wall display similar to [99]) without communicating with other devices. The difference between an active observer and a passive observer in devices is that the active observer contributes to the interaction (by communicating with other devices), similar to a collaborator with a direct user. On the other hand, the passive observer only shows a response (e.g., playing audio on speakers and showing notifications on TV), similar to a person who shows reactions to some activity without direct participation. Applying user's roles to devices shows how smart environments might contain different devices that have different functions, making them play different roles. In addition, we can track non-digital objects (e.g., couch) In the smart environment using different tracking systems (e.g., by attaching VICON markers or VIVE trackers to the objects). Tracking non-digital objects might play important roles in the smart environment. For example, the system might start testing user-to-TV relations only when s/he sits on the couch. A part of the environmental context is the physical layout of the space that include but not limited to identifying the location of digital and non-digital objects in the space.

The awareness of environmental context provides the basic building blocks of proxemics-aware systems. It provides data like identity, positional data, status, and space layout. The ability to access this data allows measuring and detecting proximity relations between participating entities. In addition, the type of task that is performed in the environment defines the role of each entity. For example, by employing both identity and proximity to appliances, the system might prevent guests from changing the temperature in the thermostat, allowing only the host to control it (it is the host and close to the controller). The task here is controlling the thermostat, which is restricted to the host only (the identity and the role). Considering the need for environmental context awareness, the presence of various types of tasks in interactive systems, and the different roles of entities in the environment, how can we build a software infrastructure to support the variety of smart environment applications?

Two theoretical concepts related to the infrastructure layer have been influential in the research literature that we should consider. First, the study of interpersonal proximity (termed *proxemics*) was introduced in 1965 by Edward Hall. He defined proxemics as understanding how the space surrounding a human is being used and how behaviors can be affected by crowded spaces [41]. Hall defines four interaction zones: *intimate* space, *personal* space, *social* space, and *public* space. Each space or zone has minimum and maximum thresholds, which can change based on the environment (e.g., bus vs. dining hall) and can be influenced by cultural factors. Human-Computer Interaction (HCI) researchers have been using the concept of proxemics when designing applications for interactive systems. For example, the work of Vogel and Balakrishnan [123] applied Hall's proximity theory to the spatial relationships between users and a wall display. They established four interaction zones corresponding to Hall's zones. Transitions between zones (changes in the distance relative to the display) triggered changes in content presentation and interaction style. Their system allows multiple users to work independently and simultaneously on different parts of the screen. We can classify users as direct participants (the user's role) who work independently on separate tasks (uncoupled tasks) on a single display. Greenberg et al. [38] applied proxemics theory more generally to interaction with heterogeneous displays integrated into physical work, leisure, home, or other settings. In addition to using the spatial relationships between users and the display, they also incorporated the relative orientation to provide more support (e.g., pause media when turning away from the TV). In their Proxemic Media Player application, users' roles change based on their proximity to the display (e.g., the user standing at the display gains the control). They introduced the five dimensions of proxemics for ubicomp: distance, orientation, movement, identity, and location. Their work also included the development of the Proximity Toolkit to facilitate the development of "proxemics-aware" applications [71]. One of the five proxemics dimensions that Greenberg et al. found useful to consider is orientation. Using orientation, developers can support user interaction by detecting which entity (e.g., person or device) has the user's attention (e.g., sensing attention [104]) and respond accordingly. Combining both proximity and orientation defines the second theoretical concept I want to discuss: F-Formations. F-formation theory, initially developed by Kendon [58], considers the way people arrange themselves when they interact with each other [58][74]. Several canonical formations have been identified, each corresponding to patterns of interpersonal interaction (e.g., circular or face-to-face formation when interacting with a counter assistant and a semi-circular around a wall display). When looking at the existing work in the literature, we can see the presence of the F-formation concept between people only, people and devices, and devices only. For example, in GroupTogether [72], Marquardt et al. used F-formation to determine who is in the group and who is not. Azad et al. [11] explored how single and multiple groups arrange themselves relative to vertical displays. Dynamic F-formations also exist between surgeons, nurses, and imaging technologies, as discussed by Mentis et al. [75]. Zhou et al. [132] used the F-formation to notify the user of a tablet about the presence of onlookers. Other systems used F-formations between devices only (e.g., EasyGroups [54][55]). Considering these systems, we can see that devices take different roles. For example, passive observers are the devices that only show contents with no direct interactions like public kiosks in Azad et al.'s work [11]). The tablet in Zhou et al.'s work [132] plays as an active observer that shows notifications during the interactions. Finally, the Ring technique in EasyGroups [54][55], the role of adding the new devices moves from one device to another (each device touches the device on its right to add it to the group). While the proximity and F-formation theories have been applied by many researchers in HCI, what existing tools employ both theories to support the implementation of proxemic aware applications? And do they provide enough support compared to the developer's needs?

The Proximity Toolkit [71] is well-established in the literature that supports the implementation of rule-based applications that use the relative distance and orientations and pointing relations between entities. Pérez et al. [85] also developed a framework to support implementing mobile proxemic apps for smart environments. Their system also detects the relative distance and orientations between entities based on the collected proximity data from existing smart devices in the environment (e.g., smartphones and wearable devices). However, there remains a considerable gap between what existing systems support and what developers require to detect more complex proximity scenarios. Specifically, when testing multiple proximity attributes for multiple entities and combining proximity events with external data (e.g., system notifications and UI events).

Machine learning researchers also have employed the proximity theory and F-formation to recognize proxemics interactions (e.g., detecting social interaction [1]). Using machine learning requires a lot of processing, including creating and preparing a dataset and training and testing the classifier, making machine learning not suitable for every use case (e.g., when there is a need for a quick setup, frequent changes in the environment). However, even when using machine learning to detect proxemics interactions, the classifier requires data to train the model and input data during prediction. Platforms like Microsoft PSI, "an open, extensible framework that enables the development, fielding, and study of situated, integrative-AI systems" [87], processes one or more input streams through stream operators to generate a single output stream. This output stream then can be forwarded to another component or a machine learning classifier for further processing. Using this output stream, Microsoft PSI can support the machine learning classifier in two ways: generating labeled data for training and feeding the classifier with proximity data for prediction. However, using Microsoft PSI to generate proximity data requires a lot of effort as developers need to write the code to calculate different proximity attributes. This effort will increase when the input stream is low-level proximity data (e.g., position and orientation) from a tracking system (e.g., VIVE trackers) as the low-level proximity data require preprocessing before generating proximity data (e.g., relative distance). Therefore, implementing complex proxemics-aware scenarios using existing toolkits will be very difficult and time-consuming for developers. For example, keeping track of multiple events (e.g., relative distance and orientation) to perform a single response in The Proximity Toolkit requires listening for two different events to draw the response. It also requires creating multiple relations between every two entities in case we have more than two entities. Using Microsoft PSI, collecting training data and providing the machine learning classifier with proximity data (e.g., relative distance) is also difficult and time-consuming. This is because developers need to write the methods to calculate all proximity attributes (e.g., distance and orientation) before passing them to the classifier. These difficulties shift developers' focus from implementing the final product (e.g., system responses) to dealing with low-level data to detect user's behaviors.

To address these limitations, I present ProxemicUI, an object-oriented framework that provides a simple but powerful rule-based format for expressing proxemics relationships, an event-driven model for responding to their occurrence, a mechanism for integrating non-proxemics data and events, and extensibility through clearly articulated extension points. ProxemicUI eases the process of implementing proxemic-aware applications by combining basic proximity tests into a single complex test, where developers keep track of a single object instead of multiple. Similarly, it integrates proximity tests (basic or compound) with an external data source (e.g., direct interactions and system notifications). Besides, ProxemicUI can support the machine learning classifier in three ways. First, developers can test user experience and technical and contextual challenges of classifiers before building them. Second, developers can also use ProxemicUI to train the classifier by generating a labeled dataset (e.g., entities, relative distance, pass/fail). Third, ProxemicUI can feed the proximity data to the classifier as a part of the input data that the classifier requires to complete the prediction. Through its rule-based approach, ProxemicUI also serves as an explainable alternative to machine learning, one that is easy to modify and adapt as circumstances change.

My work makes the following contributions:

1- Defined a set of design requirements for proxemics-aware applications in smart environments that were derived from the literature, my previous experience with the ProximityTable (a museum information kiosk that responds to proximity relations between users and the kiosk), and the outcomes of evaluating ProxemicUI.

2- Fully implemented a robust rule-based object-oriented framework to support proxemics-aware applications, called *ProxemicUI*.

3- Demonstrated how beneficial the ProxemicUI development model is compared to The Proximity Toolkit model (a well-established toolkit in the literature for proxemics-aware development), specifically when detecting complex proxemics interactions through combining basic rules into a single rule (CompoundRule) and integrating proxemics interactions with non-proximity data or events (HybridRule).

4- Demonstrated how ProxemicUI can more readily support a proxemics-aware machine learning classifier for proxemics data, compared to Microsoft PSI, a state-of-the-art ML platform for smart environments, in two ways. First, how ProxemicUI's rules can train a machine learning classifier through labeling proximity data. Second, how to feed the machine learning classifier with data about detected proximity events (e.g., a certain entity's configuration is detected) using ProxemicUI's rules.

5- Demonstrated the process of integrating ProxemicUI into a larger system by integrating ProxemicUI into Story CreatAR, an authoring tool for virtual and augmented reality storytelling.

ProxemicUI was evaluated using four techniques that are commonly used to evaluate toolkits in HCI [65]. These techniques are as follows:

1- *How To* scenario technique is used in chapter 5 where I showed a step-by-step how to use ProxemicUI to implement a smart home scenario.

2- *Replicated examples* technique is used in chapter 6 where I showed how we could use ProxemicUI to reimplement an existing application that was implemented using the Proximity Toolkit.

3- *Comparison* technique is used in chapter 7 where I conducted a code review study to gather external developers' opinion about how each toolkit complete the same task. This comparison includes ProxemicUI vs. the Proximity Toolkit to build proxemic-aware applications and ProxemicUI vs. Microsoft PSI to support a proxemic-aware machine learning classifier.

4- *Case studies* technique is used in chapter 8 where I integrated ProxemicUI into Story CreatAR, a storytelling tool in virtual reality.

The rest of this dissertation is structured as follows. Chapter 2 looks at the related work to this research. Chapter 3 discusses my previous experience implementing the ProximityTable that plays an important role in defining the design requirements. Chapter 4 explains ProxemicUI version 1, including the initial design requirements for proxemic-aware applications. Chapter 5 shows a walkthrough example of how to use ProxemicUI to build a proxemic-aware applications. It also explains the final architecture of ProxemicUI (ProxemicUI version 2). Chapter 6 looks at the proof-of-concept applications of a simulated smart home and how they contribute to evaluate and refine ProxemicUI. Chapter 7 discusses the evaluation of ProxemicUI through a code review study. Chapter 8 shows evaluating ProxemicUI through integrating it into Story CreatAR. Chapter 9 reflects on the ProxemicUI framework in relation to other research. It also summaries the refinements to ProxemicUI as well as the future work. Finally, the conclusion will be in chapter 10.

Chapter 2: Literature Review

This chapter discusses the background for this research and is divided into proximity and interactive systems, entity formation and interactive systems, proximity in virtual environments, proximity in machine learning, and toolkits for smart environments.

Proximity and interactive systems

In ubiquitous computing paradigm, devices are aware of the presence of each other, their environment, and context of use [38][126]. In an effort to achieve such awareness, HCI researchers have applied proxemics theory to interactive systems design.

One of the earliest HCI research projects that applied proxemic theory on interactive displays was Vogel and Balakrishnan's work [123]. Similar to the four zones in Hall's proxemic theory, they established four interaction zones around a wall display: "ambient Display, implicit Interaction, subtle interaction, and personal interaction" [123]. Using the VICON motion tracking system, they tracked the position and orientation of users to detect levels of engagement. The transition between the four interaction zones (changes in levels of engagement) causes changes in interaction style and content presentation. Importantly, in this early work, proxemics was used to adjust the interaction modality and change the content presentation, emphasizing the highly intertwined relationship between proxemics and interactivity.

Proxi-Sketch [7] is an interactive tabletop application that uses Medusa [7], a tabletop interactive system that can track the torso, arms, and hands of users via a set of IR-based proximity sensors. Proxi-Sketch, a prototyping application to create and edit graphical user interfaces, shows a glowing orb on the tabletop display to indicate the detection of users. The clarity of the orb also indicates how far the user is from the display. We can consider the torso, arms, and hands of an individual sub-entities of the main entity (the person). ProxemicUI can sub-class the main entity to define multiple sub-entities (multiple joints). Then, developers can test proximity relations between sub-entities of the same individual (e.g., the hand is touch the face) or sub-entities of two individuals (e.g., hand shaking).

Klinkhammer et al. [60] present a museum exploration station that can track users using IR proximity sensors around the interactive tabletop. Using this data, the system creates a personal workspace for each user, which moves as they move. In addition, when two users come together, the workspace left behind will be removed after 15 seconds. During their study, there were 257 bystanders who did not interact with the tabletop. Forming a group when two people come together and detecting bystanders require multiple tests at the same time. For example, testing a single proximity attribute with multiple thresholds (e.g., the distance between the user and the table, the bystander and the table, and the user and the bystander). Other cases require testing multiple proximity attributes (e.g., the distance between each user and the table, both users, and the relative orientation between each user and the table). ProxemicUI composes multiple basic proximity tests into a complex test using Compound Rules.

Screenfinity [100] is a proxemic-aware system that was developed to allow passers-by to read content on a wall display while walking. The content on the wall display changes according to the position of the users as follows. The content moves, rotates, and changes size according to users' position relative to the display. The system also supports presenting and changing multiple contents on the display for different users. In this work, we can see that the system continuously responses to the user's position relative to the display from multiple users. Using ProxemicUI, this can be achieved by defining a single relative distance rule that includes all users instead of writing a test for each user. Developers then would access the relative distance for each user to respond accordingly.

HCI researchers also employed proximity between multiple users and a single display to control contents on display. Dostal et al.'s work [31] uses a combination of proxemics (distance and position) and multi-view technology so that two people can see different content on the same display at the same time. This shows testing multiple thresholds for a single proximity attribute, which can be achieved using Compound Rules in ProxemicUI. The Proxemic Media Player application [13] and SpiderEyes [30] divide the display between users, where each has different contents. In the Proxemic Media Player application, when a new user stands between an existing user and the display, s/he takes control over the content. This shows changes in user's roles based on their proximity relative to the display. In SpiderEyes, two users can come closer to each other to merge the two spaces into one; then, they can move together (towards/away from) the display to control the content. This also shows testing multiple thresholds for a single proximity attribute. Tafreshi et al. [97] [110] uses the average distance of all users relative to the display to provide a better view for all users. This approach shows calculating multiple distances for all users relative to the display.

Eyes-Free Art [90] is a proxemic audio interface supporting people with visual impairments to explore 2D artwork. Users can interact with the interface by moving between 4 different zones relative to paintings. Each of the four zones will have different auditory experiences: “Background Music, Sonification, Sound Effects, and Verbal Description”[90]. Eyes-Free Art provides direct interactions with paintings (e.g., touching different parts of the painting plays different sounds). It also provides verbal interaction (e.g., repeat instructions for each zone with the voice command “Repeat”). This system is an example of integrating proximity data with external events (direct touch and verbal commands), which ProxemicUI supports through the Hybrid Rule.

The AirPlayer [107] system extends the use of proxemics to cover multiple rooms in the house, using location and movement for interaction with the system. An example of using location is when the system starts to play songs from the library because it knows that there are two people in the living room. When one person goes to the bedroom to take a nap, where they like to listen to a specific singer while falling asleep, they start their preferred list of songs on their phone. The system then detects that they are in the bedroom and plays these songs through the bedroom speakers. Meanwhile, the previous list of songs still plays in the living room. Another example of using movement is when a user starts a list of songs while in the bedroom, then moves to the living room: the system will follow them and play the list on the living room speakers. The AirPlayer checks if an individual is in a specific location within the environment, which ProxemicUI refers to as Absolute Proximity Rules. Playing music when two people in the living room might be more specified. For example, if both people are relaxing on the couch, play music; if one is watching TV and the other is doing some work on the table, then don’t play music. With such a scenario, there is a combination of absolute and relative proximity tests that ProxemicUI supports through Compound Rules. In addition, we can see the presence of HybridRule when the user starts a list of songs, the system would detect the user’s locations to determine on which speakers the songs should be played.

Savannah [14] is an open-field collaborative hunt game. It is a proximity-based game, where players (each player is a lion) have to be in the same location to exchange information about animals. This information is being displayed on handheld devices. To win the game, players need to work collaboratively (hunt together). Hunters keep walking in the field (“virtual Savannah”) and move between different virtual zones; when they enter a zone that “contains an active target”, an “attack” button will be displayed. This game shows different features of ProxemicUI. For example, *RelativeProximityRules* and *CompoundProximityRules* when detecting the relative distance between all players. *AbsoluteProximityRules* can be seen when detecting transition between zones in the tracked field.

Many other systems do not directly employ proxemic interactions, but one can see the presence of proxemics scenarios. For instance, Lucero et al. [68] introduced several prototypes for collected mobile interactions where people use their phones to create a shared workspace. One of their examples is exploring photo collection together, as with traditional photo album sharing. In this example, all users place their phones on a table; as the owner of the photo collection starts to explore photos, all other devices move to the photo that the owner is looking at. A second example is "large-scale sharing" experiences, where multiple phones combine as a single display to present content (by placing phones next to each other), or where several devices combine into a single display while another operates as the joined display controller [67]. Lucero et al. prototypes show an example of how useful to employ proximity on device-to-device interactions. For example, they should know each device's position relative to other devices to determine which part of the photo each device should present.

By considering previous work, one can see the benefits of employing proxemics to support the interactions on interactive systems. For example, using proxemics can allocate personal workspaces [60], change display content [123], and create a joined-scaled display [67].

Entity formation and interactive systems

When people interact with one another, we can see the presence of different spatial patterns as they stand close to each other. These patterns are defined by Kendon [58] as F-formations. Kendon describes several patterns include circular, rectangular, semi-circular, L-arrangement, side-by-side, or linear arrangements, figure 1 left. Paay et al. [81][82] explored how people interact while cooking together and identified four additional F-formation: wide V-shaped, Spooning, Z-shaped, reverse L-shaped. Serna et al. [101] and Tong et al. [116] explored the collaboration in outdoor activities and identified one additional F-formation that is a triangular arrangement. They found that the triangular formation is influenced by the roles of group members who form the triangular for a short period (e.g., one member gives instructions to the group). Solera et al. [106] argue that groups cannot only be detected using the position and orientation of individuals relative to the o-space as proposed by Kendon. For example, in crowded areas, group members tend to reduce the inner distance between group members. They also argue that some group members might not directly connect to the center of the group (o-space) but still a member of that group. The environmental context can influence the type of pattern when people create a formation. For example, two people can create a circular formation when discussing exhibits with the counter assistant at a tourist center; the same pair can create a side-by-side formation when annotating a map on a table [74]. Other temporal environmental contexts might also influence these patterns (e.g., changing the formation when it is crowded around the display [11]). Existing interactive systems create F-formation patterns between people and devices that are similar to the patterns identified by Kendon [58] between people only. For example, groups of two or more tourists created circular formations when interacting with the counter assistant at the museum [74] that are similar to the formations created by visitors who interacted with kiosks or displays discussed by Azad et al. [11]. The difference between the two formations is that the kiosk or the display plays the role of the counter assistant.

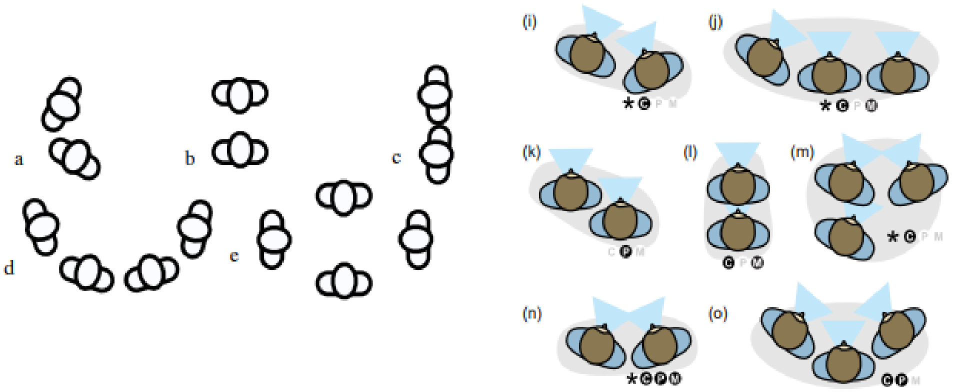


FIGURE 1: LEFT: EXAMPLE OF F-FORMATION CONFIGURATION [56], RIGHT: EXAMPLE OF GROUP FORMATION AROUND PUBLIC DISPLAYS [11]

TouristPlanner [73] is a walk-up-and-use tabletop system that allows understanding group behaviors around interactive displays in public places (e.g., tourist centers). It allows a group of up to four users to independently explore the system to choose places to visit before gathering on one side of the tabletop to discuss the final plan. Marshall et al. found that at most times, group members arrived at the tabletop at different times. While the group was interacting, some members left the tabletop to explore other parts of the tourist center. While some groups used only one side of the tabletop or had less than four members, there were some instances where strangers joined groups. The authors stated that groups chose to work on a single side of the table for two reasons: preferring a shared focus on the contents and not knowing that multiple people can use the display simultaneously. This shows that there is a need to track people around the display and respond to group configurations, for example, by creating openings for new users, notifying groups about new table space, or identifying when groups want to discuss a final plan. This kind of system response requires ongoing consideration of the relative proximity between individuals and the display.

Azad et al. [11] explored how groups reach and use three different vertical displays (cinema ticket kiosks, photo-developing kiosks, and mall directories). They found that groups arrange themselves based on how they reach the display (e.g., one member leading the group vs. all moving together). In addition, since group members' roles (driver, active observer, and passive observer) might change during the interaction, the group formation changes accordingly. Azad et al. also observed a direct effect of the crowd around the display and of arrival time on groups' positioning and formation. This shows how the position and orientation of people play important roles in the flow of interaction (e.g., to establish the owner of a workspace), and how the approach to a device can signify intent. Correspondingly, group formations can be useful to detect even before the interaction begins. They stated that group members change their positions to change their roles (e.g., taking turns at the kiosk when buying movie tickets). Similarly, We might apply these roles on devices to determine the controller of other devices according to device position. For example, during a workshop, we might have multiple presenters who present different tutorials. When the presenter moves to the front of the room, s/he can gain access to participants' devices to fix encountered issues.

Mentis et al. [75] discussed the presence of F-formations in neurosurgery, not only between surgeons and nurses but also involving devices (e.g., imaging technologies). An example of changing F-formation is when a surgeon moves towards an imaging device to direct the nurse to show different views or images. This is another example of how tracking the position and orientation of both people and devices is useful. For instance, we can combine the position of surgeons and their hand gestures to control the level of detail. We can achieve this using hybrid rules in ProxemicUI. Hedayati et al. [45] conducted a user study to explore the differences between participants' F-formation when interacting with physically present vs. virtual moderators. They recruited eight participants to participate in a quiz game. In their setting, the spatial properties (position and orientation) of the moderator (physical or virtual) are static, where participants form a group of up to seven members around the moderator. When a participant loses a round in the quiz game, they leave the group, where the remaining participants will "shuffle" the F-formation before the next round. Shuffling the F-formation and changing the number of group members results in different F-formations that the same group creates in the same setting. All participants participated in both conditions: with physically present and virtual moderators. They found that F-formations with a physically present moderator are wider (with greater distance to the center of the formation) than those with a virtual moderator. This finding supports the need to understand the environmental context to provide better interaction supports.

Luff and Heath [69] defined *micro-mobility* as how we position, orient, and manipulate objects to serve different purposes during collaborations. For example, at travel agencies, the travel agent might print multiple bookings (with different routes and prices), then pass them to the customer. Then, the customer might place them side-by-side to compare total trip time, number of stops, and prices. In this scenario, the paper documents are moving from the agent to the customer and placed on the table in a specific form by the customer to complete the task (e.g., choose a flight).

Marquardt et al. employed the *micro-mobility* concept in their system, GroupTogether [72]. GroupTogether is a system containing a set of cross-device interaction techniques for sharing content between interactive devices. The design of these interaction techniques reflects two different concepts: *micro-mobility* and *F-formation*. They used *micro-mobility* to define how to orient and/or position a handheld device to perform a task (e.g., send a copy of the shared document). They also used the F-formation concept to determine who is in the group and who is not (based on positions of people who hold devices) as well as space where the interaction should take place (the shared space between users). ProxemicUI provides the support for micro-mobility and f-formation through detecting the relative relationships (distance and orientation) between entities (e.g., devices and people), where multiple relative rules can be combined into compound rules.

Lucero et al.'s work [67][68], discussed in the previous section, introduced several collected mobile interaction prototypes to create a shared workspace using mobile phones. Their prototypes show examples of using two, three, and four mobile devices to create a shared display. Based on the number and orientation of devices, the size and presentation of the image change. This also shows the importance of detecting spatial relationships between devices.

Some formations do not indicate creating groups, but there might be a need to detect and respond to their existence. For example, Zhou et al. [132] designed a tablet interface that can detect and notifies users about the presence of onlookers to protect their privacy. They used Polhemus G4 (“a 6 degree-of-freedom electromagnetic motion tracker” [132]) to extract proximity data (position and orientation) of entities (tablet and onlooker). Then, they used this data to calculate the relative distance (< 5 feet) and orientation (within 60º field of view) between the tablet and the onlooker to provide users with notifications about onlookers and/or perform a protection technique of the content on the screen. In addition to the automatic protection, they have “a manual protection mode allowed tablet users to decide when and for how long to use a protection by clicking a button” [132]. This system includes testing multiple relative proximity attributes (distance and orientation) and supporting proximity events with user confirmation (UI touch events), which ProxemicUI can support.

Tracking devices vs. users in some systems, group formations are created by users holding their mobile phones. In these systems, mobile phones play the role of proxies to those users, which are used to track/determine the locations of those users. For example, Jokela and Lucero introduced EasyGroups [54][55], which includes three techniques to create groups of mobile devices for collaborative interaction. Their application interface is represented as a tabletop on which it places devices in-order around it for collaborative work. While employing spatial relationships between devices help in automating the detection and creation of groups, their system creates groups manually as follows. In the Seek technique, one user (which they called “the leader”) creates a group using the “Seek application”. Then, the rest of the users search for that group using the same application while they are in the same network as the leader; users need to enter a password to join the group. If devices' order is different from their physical order, the leader can reorder devices by touch through the application. In the Ring technique, one user creates the group then touches the phone of the next user to his/her right using his/her phone to add that user to the group. The second user follows the same procedure to add the third user (who should be on the right side of the second user) and so on until they complete the group. Following this approach, the application can keep track of users' orders as each new user will be added by the person at their left. In the Host technique, one user creates the group (which they also called "the leader") then keeps touching all devices using his/her phone in counter-clockwise order to keep track of devices order. With the ProxemicUI framework, grouping and ordering can be much easier. For example, when the leader creates a group, anyone within a threshold might get notified about the new group, the leader then can accept or reject the joining request. To determine ordering, ProxemicUI can detect each device's relative position and orientation. The application might also be automatically responsive to f-formations as they occur, removing the need for explicit initialization of join requests from a group leader.

Jokela and Lucero also introduced FlexiGroups [55][56], which expands on the techniques in EasyGroups. In FlexiGroups, one user starts the application and adds other users by touching other users' phones using his/her phone. The first difference here is that other users do not have to start the application where their phones are in an idle state. Then any added user can add new users following the same procedure. In terms of devices' order, all devices will be showing on the tabletop screen (on the phone application) in the same order that they were added. Then, devices can be reordered to match the right physical order by any user in the group, where all screens will be locked while a user starts the reordering process. Here proximity is used to add new devices, which we refer to as relative distance in ProxemicUI. However, they still have the ordering issue that they do through direct interactions, which can be done automatically through relative position and orientation in ProxemicUI. While there exist different scales of f-formations, from micro-mobility to smart home scales, the ability of ProxemicUI to work with these different scales is determined by the ability of sensing technologies to detect proximity at different scales.

In summary, we see that group formation around interactive displays might change over time due to changes in members' roles, group size, and the number of groups. In addition, there are instances of breaking one group formation and creating another one in medical settings. For example, in the operating room, we might have a formation that includes three surgeons around the patient table. During the operation, a surgeon might break this formation and moves towards the imaging display, creating a new formation including the nurse and the display). What is more, group formations can be seen between people only, between people and devices, and between devices only. Therefore, tracking changes within groups as well as between groups might be useful in different ways. For example, content on the interactive display might change according to the group size as well as the number of groups and individuals around the display; and multiple interactive displays might work as a single display; and multiple devices might work individually to process a task. In addition, we might have two different relations in a single formation. For example, if we have two people standing by a tabletop at a museum entrance, one of them faces the tabletop while the other one is facing away from it. In this scenario, we have a group based on their proximity and two sub-groups based on their orientation (user and tabletop face to face, and a bystander and the tabletop side by side/ face to back). The system response of this complex case might be different than other simple cases (e.g., both people are facing the tabletop).

Proximity in virtual environments

Researchers have been studying how proxemic interactions manifest in virtual worlds and if the social norms of physical spaces exist in these virtual environments. Hecht et al. [43] conducted a study where they asked participants to approach an avatar using two different techniques. First, the participant approaches the avatar by physical movements toward the avatar on the screen. Second, the participant controls a second avatar through the keyboard to approach the other avatar. With both tasks, the goal was to reach a comfortable distance to ask a stranger for direction. They found that the average distance in both techniques was 1 metre, which is close to the personal space boundary in the proxemics theory defined by Edward Hall [41].

Bonsch et al. [19] conducted a user study to explore how the personal space of a user can be affected by three factors: the facial expressions of virtual agents, the number of approaching virtual agents, and the direction of their approach. Their setup was a five-sided CAVE, where they asked participants to stand in the middle and only allowed them to turn their heads in order to look around the space. They asked participants to indicate their comfortable and uncomfortable zones when being approached by virtual agents while varying the three factors. Participants indicated their preferences using an ART

Flystick 2. They found that the number of approaching virtual agents and their facial expressions impact personal space preferences. These findings support Hecht et al. claim (discussed above) about the similarities of proxemic interactions in virtual and physical worlds. For example, the o-space (the middle area) size in a circular formation tends to increase as the number of group members increases. Therefore, detecting such formations will be the same in both physical and virtual worlds, which is supported using Compound Rules in ProxemicUI.

Williamson et al. [128] conducted a virtual academic workshop that was built using Mozilla Hubs. Their research studied social interactions in the virtual environments how they can be affected by the size of space they took place in. One of the challenges participants stated is the difficulty of sensing what is behind their avatars. Another issue participants faced is how far they should be from other avatars to start a private conversation or start a private conversation while still participating in the large group. Authors argue that it is important to provide social awareness to participants about different events in the environment (e.g., when an avatar leaves the room). Such awareness might help "organizers to ensure all attendants are stationed in the rooms they are meant to be at any given time" [128].

In summary, proximity in virtual environments is somehow similar to its use in physical spaces where people tend to leave enough distance between their avatars and other avatars (e.g., to have comfortable conversations with others and avoid blocking other's views). Besides, employing proximity in virtual environments will improve the interactions between users. For example, using Compound Rules in ProxemicUI (relative distance and orientation between avatars), the system can allow users to start a private conversation. If avatars already within a distance of each other (e.g., theater settings), the system might use relative orientation only (avatars turn to each other) to start the private conversation. When the private conversation starts, the system might decrease the voice of the main talk to make the conversation event more realistic. In some contexts, proximity awareness is important from a management perspective to control the flow of virtual events. Using the AbsolutePositionRule, ProxemicUI can detect proximity events to know who is in a specific room. ProxemicUI also can create groups (using F-formation Rules) to start a specific event (e.g., start a conversation or talk).

Proximity in machine learning

Researchers in machine learning have also employed proximity and F-formation to detect social interactions in different contexts. This section will discuss some of this work to understand how researchers have used proximity data to train a classifier to detect social interactions.

Aghaei et al. [1] introduced a system that aims to detect social interaction in photo streams captured by a single wearable camera. The social interactions they detect are when the user forms an F-formation with other people. They detect F-formations by estimating the position and facial orientation of other people. Their model consists of two modules as follows. The first module extracts the two features that define the F-formation (distance and orientation) between the user and individuals in the photo streams. The module then prepares these features in the sequence level to pass them to the next module. “The second module analyzes the resulting features from the first module to classify the sequences”[1]. They used a low frame camera to capture user interactions as they stated that high frame cameras might not be suitable to capture interactions throughout the day. They trained a long-short time memory network, “a type of Recurrent Neural Networks that is capable of learning long term dependencies” [1]. The input data to the network are the resulting distance and orientation from the first module (extracting the features). Due to the lack of datasets for their specific settings, they collected approximately 30.000 images by eight users. They used 70% of the dataset for the training. They divided the remaining of the real dataset equally for validation and testing, where they also enlarged the validation set.

Ko et al. [61] proposed a method that detects abnormal human behaviors through surveillance systems. It consists of three models. First, the detection and discrimination module (YOLO network, an object detection algorithm) aims to separate objects in the video to extract human subjects. Second, the posture classification module classifies the abnormal behavior from the extracted human subject frames. This module concatenates from two models. The first model is a convolutional neural network model that is used to recognize postures in the extracted images. The second model is a recurrent neural network model that is used to extract the features for motion dynamics. Third, the abnormal behavior detection module is a long-short term memory network model that is used to detect abnormal behaviors based on the dataset. They trained and tested their models using the “UT-Interaction-Data” dataset. This dataset contains 50 video clips where multiple people perform six activities: handshaking, hugging, kicking, pointing, punching, and pushing. They found that there are difficulties in differentiating between behaviors with similar motions (e.g., punching vs. pointing). They argued that these difficulties are due to the small size of the database and the low resolution of training images.

Cumin et al. [26] employed an environmental context feature (location) to enhance activity recognitions using supervised learning algorithms. Their approach suggests using location information (e.g., living room vs. kitchen) to determine what activities the classifier should look for and from which sensors. To add a new activity during the training phase, they use all classifiers of all locations (e.g., living room vs. kitchen) to recognize this activity. When a decision is being made by every classifier of all locations, all decisions go through a fusion method to finalize the decision. For decision fusion, they used the MultiLayer Perceptron and the Support Vector Machine, two stacking classifiers. They used the *Opportunity* dataset [92], multi-level labeling of activity dataset, where they used only 17 activities and added a *None* activity for no activities or other locations activities. Three classification models were used to validate their approach: MultiLayer Perceptron, the Support Vector Machine, and the Bayesian Network. They found that location-based activity recognitions and decision fusion of classifiers' output increase detection accuracy. Location-based activity recognitions support what ProxemicUI provides with Absolute Proximity Rules, which check spatial relations relative to the environment.

Hedayati et al.’s work [44] detects F-formations using a data-driven approach based on people's spatial data (position and orientation). The first step of their approach is to define a relationship between every two individuals in annotated frames. This relationship includes the distance between the two individuals and how much each individual should rotate to face each other. The second step is to label the relationship data (inside the F-formation or not) and use labeled data to train a binary classifier. The last step is to use the relationship data to form existing F-formations. They trained three ML classifiers: Weighted KNN, Bagged Trees, and Logistic Regression, using the SALSA dataset [2], a large dataset of 18 people interacting in an indoor setting. They used 80% for training and 20% for testing. They compared the result of the three models with their implementation of the Graph-Cuts algorithm [102], a Computer Vision method that detects F-formation from single images, and they found that all three models performed better than the Graph-Cuts algorithm. Their approach processes every two people individually, then combines the results after the classifications. ProxemicUI can provide the same data (relative distance and orientation) for more than two entities simultaneously, making it more suitable to support the ML classifier.

Using machine learning requires a lot of processing, including creating and preparing a dataset and training and testing the classifier. Therefore, the cost of this process will increase as the number and complexity of the required interaction techniques increases, making machine learning not suitable for every context. For example, when collecting training data, the dataset should have a reasonable amount of data for every interaction technique that we need to detect. ProxemicUI provides a robust and straightforward event model to detect interactions that have been explored with machine learning. For example, Aghaei et al. [1] and Hedayati et al. [44] detect F-formation based on the position and orientation of entities. Cumin et al. [25] use a location-based approach to determine what activity to detect to enhance activity recognition. ProxemicUI can accomplish this by combining the absolute position with the interaction technique using HybridRule. In addition, ProxemicUI can ease the process of training the classifier and detecting proxemic interactions by feeding the classifier with high-level proximity data (e.g., relative distance) through its rules.

Toolkits in smart environments

Edwards et al. [32] defined four ways of addressing the infrastructure problem in HCI: *Surface* approach, *Interface* approach, *Intermediate* approach, and *Deep* approach. Without making any changes to the infrastructure itself, the *Surface* approach employs the applications to give users a better product using that infrastructure. The *Interface* approach aims to modify the infrastructures to provide developers with more suitable abstractions of the infrastructures. The goal of the *Intermediate* approach is to ease the application development process by implementing new features that are not supported by existing toolkits. The *Deep* approach aims "to directly influence the architecture of infrastructure itself" [32]. ProxemicUI v1 follows the *Intermediate* approach, where it uses any low-level tracking system (e.g., DT-DT or HTC Vive Lighthouse) as a source of position and orientation data and integrates them with a non-proxemic data source (e.g., Windows event model) to generate higher-level proxemics events (e.g., relative distance between two people).

Toolkits in HCI research can be placed on a continuum between *research toolkits* and *toolkits for research* [131]. Toolkits for research “speed up development by encapsulating common code revealed during research, enabling faster iterations and research participation by more people”[131]. PyMT [42], a toolkit to support the development of post-WIMP interfaces (e.g., multi-touch interface), is an example of an HCI toolkit for research. On the other hand, research toolkits “enable development of interfaces based on entirely new paradigms” [131]. ZOIL Framework [133], a toolkit for post-WIMP zoomable user interfaces, is an example of an HCI research toolkit. It is essential to study existing toolkits to establish the research gap in the literature. ProxemicUI falls in between the two types of toolkits where it enables faster developments and still provides a new paradigm through Compound and Hybrid Rules. This section will discuss some of the existing and relative toolkits to my research.

Schipor et al. introduced two related toolkits called EUPHORIA [98] and SAPIENS [99], where SAPIENS was built based on the architecture of EUPHORIA. The goal of these toolkits is to ease the development and validation process of interactions between different devices in smart environments. Both toolkits classify devices in smart environments as follows. First, *Producers* are input devices (e.g., a microphone, a VICON tracking system). Second, *Consumers* are output devices (e.g., a wall display, a sound system). Third, they consider devices that can be input and output devices simultaneously (e.g., tablets and smartphones), a mix of Producers and Consumers. The SAPIENS design adds a few new features over EUPHORIA, but I will discuss the only two relevant features to ProxemicUI. First, The ATTENTION-DETECTION-MODULE evaluates the received events from Producers (e.g., a change in position event from VICON motion sensors) in order to track the user's attention within the smart environment (e.g., a user is watching TV). Second, the CONTEXT-AWARENESS-MODULE collects spatial data of participating entities (e.g., position and orientation), which can be used to discern which entity has the user's attention and what Consumers should be used to show notifications to users. It is helpful to consider their devices' classifications in smart environments (Producers, Consumers, and both) in relation to the user roles discussed in the introduction: direct participant, active observer, passive observer, and disengaged bystander [11][112]. We can benefit from their device classifications in relation to user's roles to support the interaction by considering the functionality assigned to each device, giving each device a different role. For example, if we have an environment that contains ten devices, five of which are either input/output or only output devices. When the system wants to show notifications, it will only test spatial relationships between the user and those five devices instead of the ten devices. I proposed that we could extend these roles to devices and considered a device that controls other devices in a smart environment, a direct participant, and a device that receives commands and, as a part of its response, communicates with other devices in the environment an active observer. In Schipor et al.'s classification, both roles fall under what they call a mix of Producer and Consumer (i.e., both), although the controller might be considered more purely as a Producer (albeit sometimes as a proxy for the user as in the case with remote controls, for example). I also propose that any device that doesn't communicate with other devices but responds to commands is a passive observer, which falls under the Consumer in Schipor et al.'s classification.

Schipor et al.'s ATTENTION-DETECTION-MODULE and CONTEXT-AWARENESS-MODULE collect proxemics data and evaluate proxemics relationships between entities to support user interactions in the smart environment. Since SAPIENS is not a fully implemented system, Schipor et al. introduced an online simulation application to illustrate how their system works. ProxemicUI is a fully implemented rule-based model that detects proxemics relationships between entities where developers can choose which entities, what property to be tested, and set the threshold for each rule.

Pérez et al. [84][85][88] developed a framework to support implementing mobile proxemic apps for smart environments. Their system gathers proximity data from existing smart devices in the environment (e.g., smartphones and wearable devices). Their framework consists of three components. First, Proxemic Zones: because they consider the four proximity zones, developers would use this component to define the proximity zones by passing maximum thresholds for each zone. Then, developers will connect these zones to one or more entities, which is used to change the interaction between every two entities. The second component is the API: is the interface where developers can access all methods required to create proxemic zones and to test proxemic data. The last component is the DILMO: allows defining combinations of the five proximity dimensions (e.g., change the interaction according to distance and orientation). An example of using the framework is a mobile app that can control the volume of the video based on which of the four proximity zone does the user stand relative to the phone. The first limitation of their system is that they detect orientation based on detected faces that are not suitable for every scenario (e.g., human-to-device interactions, devices with no cameras, and device-to-device interactions). There are also no thresholds for orientation which is not ideal for every use case (e.g., I may not directly facing a tabletop, but I am still engaged with its content). They also do not consider other properties of entities when measuring spatial relationships (e.g., the shape of an entity is important to measure the distance accurately, a person to a tabletop). Besides, they perform one-to-one tests between entities, making the system not scalable with a larger number of entities. XDKinect [80] is a toolkit that employs Microsoft Kinect to support the interaction of cross-device applications. One of its components is the *Proxemic API*, which supports proxemic interactions. It activates/deactivates the interaction mode according to the user distance relative to the display. It can also detect the presence of multiple users around the display based on the number of the detected skeletons using Microsoft Kinect. While ProxemicUI does not limit the number or the type of tracked entities, XDKinect limits the tracked entities to a limited number of human subjects as it uses Microsoft Kinect to detect users.

The Context Toolkit [29][95] was developed to support the implementation of context-aware applications by providing a set of reusable widgets that play as a source of context information to be accessed by the applications. Each widget provides a number of attributes and callback methods that aim to hide the complexity of building context-aware applications. For example, the IdentityPresence widget provides three attributes (location, identity, and timestamp) and two callback methods (PersonArrives and PersonLeaves). Each widget also implements generators which are software components that capture context data from sensing hardware (e.g., voice recognition). One of the most important points discussed when using this toolkit is composing Context Widgets to generate a widget that provide the application context information based on the information gathered from the two widgets. However, each widget “is implemented as a single process” where widgets “uses peer-to-peer communications”. In addition, the Context Toolkit requires creating a widget for every location that it needs to access its context information, which raises the scalability issue. ProxemicUI provides a rule hierarchy that includes a set of rules to combine different types of proxemic and non-proxemic tests in single rules. It also employs the OSC communication protocol to as an interface protocol to capture proximity data from different sensing technologies. In addition, ProxemicUI also addresses the scalability issue by allowing testing multiple entities in a single rule.

One of the most relevant works to ProxemicUI in research toolkits is The Proximity Toolkit [50][71], which is a toolkit that provides developers with a set of proxemics data between entities. According to Marquardt et al. [71], proxemics data includes orientation, distance, motion, identity, and location; and entities include people, interactive displays, digital and non-digital objects. The toolkit provides developers with a 3D visualization window that allows them to examine the relationships between entities. In addition, their “proxemic data model” can employ any tracking system to extract proxemic data. The Proximity Toolkit provides developers with low-level proxemics data (raw proxemics data for each entity, e.g., the position of an entity within the tracked area). It also provides developers with more high-level proxemics data. These data include relationships between two entities (e.g., the relative orientation between two entities) and pointing relationships between two entities (e.g., A is pointing at B). These higher-level proxemics data is represented by events where they only pass two entities as arguments to be tested. On the other hand, ProxemicUI follows the same events approach with more arguments to give developers the flexibility to control the test, where each event has two list of entities and minimum and maximum thresholds.

To understand the use of proximity data that Proximity Toolkit provides and to have a better vision of the differences between Proximity Toolkit and the ProxemicUI framework, we will look at several research prototypes that employed Proximity Toolkit. First, the Proxemics Media Player application [13] uses The Proximity Toolkit to track people, digital devices, and other physical objects, to define the relative proxemics relationships. The system reacts differently based on the received relative proximity data. For example, the system starts showing a set of videos on the wall display as the user enters the room, and the number of videos displayed increases as the user gets closer to the display. After the user selects a video, it is displayed full screen when they sit on the couch. Here we can see that the system used relative distances between the user, the wall display, and the sofa. Another example is, when a second user enters the room, the system starts to show details about the video, where the details will increase as the second user gets closer to the display. Here we also see several relative distances (with multiple thresholds) between four entities, the two users, the wall display, and the sofa.

The Proximity Toolkit implements these by defining multiple relations events where each event represents the relation between two entities. This way, developers need to listen to all events to draw the response. On the other hand, ProxemicUI eases the complexity of testing multiple attributes for multiple entities with multiple thresholds by allowing developers to create basic relative proximity rules and combine them into compound proximity rules. This way, developers listen to a single compound rule to perform the response.

Proximity-Aware control [64] is another system that employs the Proximity toolkit to gain control of appliances based on relative distance and orientation. For example, as a person moves around the room while holding a tablet (the controller for existing devices, e.g., TV and radio), the tablet will show icons for controllable on the screen. Here we can see relative orientation between multiple devices. In addition, the decrease of the distance between that person and a device increases the level of engagement and allows for more control; this uses a single relative distance between a person (the device they are holding) and an appliance.

Proxemic Brainstorming application [70] uses The Proximity Toolkit to transfer information between devices using their relative distance. First, Proxemic Brainstorming shows the presence of other devices by showing small icons at the edges of the personal device, which moves around the edges according to the other device's position. Decreasing the distance between two devices allows increasing the displayed exchangeable content accordingly. Finally, according to the distance between devices, different interaction techniques can be applied.

Proximity Peddler [125] is a public advertising interactive display that uses The Proximity Toolkit and aims to capture the attention of passersby to explore and buy products from online marketplaces(e.g., Amazon). It captures their attention by changing the shown animation speed and presenting more details about products based on a passerby's position and orientation relative to the display. Here, we can also see the use of relative proximity data.

Physio@Home [62] is a system that employs The Proximity Toolkit to retrieve the relative position and orientation of joints (shoulder, elbow, and wrist). This relative proximity data is used to provide visual guidance and feedback for patients to help them do their exercises correctly. In this system, each joint represents an entity which is an interesting aspect where we might derive sub-entities from the base entity (joints are sub-entities from body entity). They measure relative proximity data for three entities (joints): something that is achieved in a straightforward way as a single Compound Rule (as in ProxemicUI).

While a number of toolkits support proxemics awareness in smart environments, there is still a considerable gap between the supported basic proximity triggers and what is required to support complex interactions in smart environments. For example, detecting more complex group formations when not all members are facing the group's center and drawing system responses based on proximity events and non-proxemics data (e.g., system notifications or direct interactions). ProxemicUI fills this gap by allowing developers to compose basic proximity rules into a single compound rule and integrating proximity events with an external data source (e.g., system notifications).

Evaluating HCI toolkits

Ledo et al. [65] analyzed 68 published toolkit papers and defined four different methods that are used to evaluate toolkits in HCI. The first method is "*demonstration*", which shows the toolkit's features and how developers can use them. The second method is "*usage*", which involves external users' participation (programmers) in the evaluation process. The third method is "*technical performance*", which helps to find out how well does the toolkit work. The fourth method is "*Heuristics*", which is "used as a discount method that does not require human participants to gather insight, while still exposing aspects of utility" [65]. Each of these methods has several techniques that can be used to evaluate the toolkit. In this section, I will discuss with examples the four techniques that I used to evaluate ProxemicUI.

Demonstration

I used three of the *demonstration* techniques to evaluate ProxemicUI. First, the "replicated example" shows how the new toolkit can reimplement existing work following broader solutions. For example, SwingStates [10], an extended library from the Java Swing toolkit [109], adds a "state machine" feature to replace the listeners when defining interactions. The authors compared the crossing interaction techniques in their library to the ones in CrossY [9], a drawing application to demonstrate the benefits of the crossing interaction technique for graphical user interfaces. In my evaluation, I am reimplementing a smart home scenario similar to the setup done by Ledo et al. [64].

The second *demonstration* technique that I used to evaluate ProxemicUI is “*case study*”, which demonstrates what the toolkit can do. For example, Prefuse [47] is a toolkit that was implemented to support the visualization of structured and unstructured data. One of the case studies that use Prefues is Vizster [46], a visualization tool to “explore online social network services”. In my evaluation, I am reporting how ProxemicUI was used by Story CreatAR [105], a VR/AR storytelling tool.

The third *demonstration* technique that I used to evaluate ProxemicUI is “*How To*”, which demonstrates in detail how a developer can create a specific application using the toolkit. This demonstration can be done in several ways. For example, RetroFab [89], a non-expert tool to redesign/modify physical interfaces such as a toaster interface, and Pineal [63], a tool for end-users to “prototype interactive smart devices” [63], are both provides detailed steps in how to create an example using the tool. On the other hand, VoodooSketch [18], a system that adds “physical interface palettes” to existing interactive surfaces to control applications, employs a demonstrative scenario to show how users can use the toolkit. Other systems, such as Weave [24], a cross-device framework that uses scripting to support the interaction between wearables and mobile devices, used code pieces to show different features of the system. In my evaluation, I am using a demonstrative example as a context where I show detailed steps on how to use ProxemicUI’s features, including code samples.

Usage

I used one of the *usage* techniques to evaluate ProxemicUI that is "*comparison*", which can be done between the new toolkit and a baseline. The "baselines include not having a toolkit or working with a different toolkit" [65]. For example, the MAUI [48], a toolkit for supporting group awareness in groupware interfaces, compared their implementation with an implementation that does not use a toolkit and the GroupKit [93], a toolkit to build "distributed computer-based conferencing" that support group awareness. While the evaluation of Damask [66], a prototyping tool to support interfaces that work on multiple devices, compared using and not using the tool's features during the implementation of the interfaces. The evaluation of XDStudio [79], a tool that provides two authoring modes to support the development of interactive web interfaces that work on multiple devices, compared using the tool with a single authoring mode (one mode at a time) and both authoring modes at the same time. In my evaluation, I comparing ProxemicUI with two toolkits: The Proximity Toolkit [71] to support building proxemic-aware applications and Microsoft PSI [87] to support proxemic-aware machine learning classifiers.

Summary

This chapter discussed how HCI researchers employed three sociological concepts and theories (Hall's Proxemics, micro-mobility and f-formations) to support the interaction with interactive systems in different context. It also looked at a number of toolkits that were developed to provide application developers with such support. For example, the Proximity Toolkit allows one to track entities and detect their proxemic relationships. While the work we discussed provides supporting evidence of the design requirements I defined to build ProxemicUI, the next chapter (chapter 3) discusses my experience implementing and evaluating the ProximityTable, which also serves as a source of these design requirements.

Chapter 3: ProximityTable

This brief chapter will summarize the implementation and evaluation of ProximityTable, which was completed during my master's degree. Full detail about the work is available in the thesis [4]. I developed ProximityTable before proposing and implementing the ProxemicUI framework: that is, the implementation and evaluation of ProximityTable served as an important source of requirements during the design and development of the ProxemicUI framework.

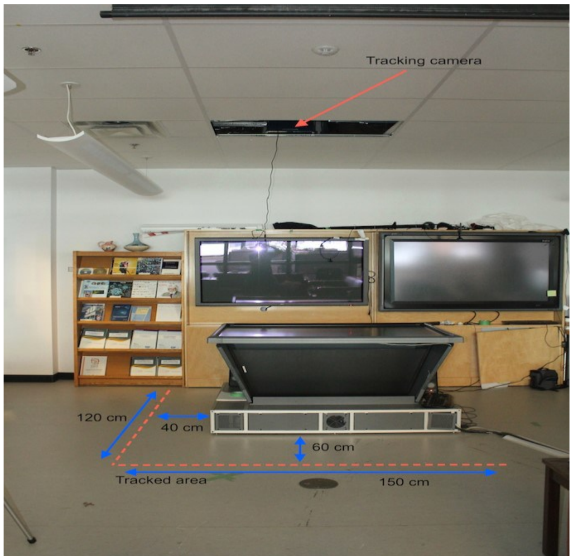


FIGURE 2: SETUP OF PROXIMITYTABLE, KINECT CAMERA INSTALLED IN THE CEILING FOR TOP-DOWN TRACKING, TWO SIDES AROUND THE TABLE ARE BEING TRACKED

Implementation and Evaluation of ProximityTable

ProximityTable is a Windows Presentation Foundation interactive tabletop application that I built using C# and the Microsoft Surface 2.0 SDK. It uses a top-down tracking system to track users around a tabletop display. Based on the tracking data, it generates a set of proxemics-triggered events that allows for changes in the configuration of the display. I evaluated ProximityTable using a focus group study that included completing tasks using ProximityTable and reflecting on the application design.

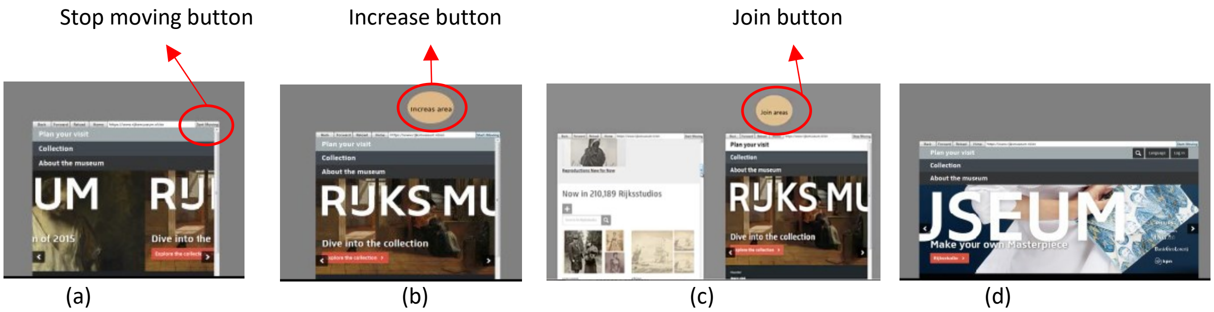


FIGURE 3: SCREEN SHOTS OF THE MUSEUM KIOSK APPLICATION
A) SINGLE USER WORKSPACE WHERE A "STOP MOVING" BUTTON IS ON ITS TOP RIGHT CORNER; B) "INCREASE" TOUCH BUTTON IS SHOWN ABOVE THE WORKSPACE WHEN A BYSTANDER IS GETTING CLOSE TO THE USER; C) "JOIN" BUTTON APPEARS WHEN 2 USERS FORM A GROUP; D) BIGGER WORKSPACE APPEARS WHEN INCREASE OR JOIN BUTTON IS TOUCHED

Implementation

ProximityTable [4] uses the OSC communication protocol to receive low-level tracking data captured through DT-DT [49], a tracking system that uses a Kinect camera with a top-down view to track human activities. DT-DT was used as a source of tracking data because it allows tracking not only users at the tabletop but also users who stand behind them. In addition, DT-DT does not require users to wear any additional equipment. ProxemicUI also employs the OSC communication protocol to receive data about entities in the environment. The data can be sent from a tracking system (e.g., DT-DT) or read from a file (e.g., size of a table that only needs to be configured once). Besides, using the OSC communication protocol allows ProxemicUI to connect with broader tracking systems by writing a communicator code that captures the tracking data and forwards them to ProxemicUI.

Interactive tabletop displays can be used simultaneously by groups and individuals [11][73]. ProximityTable is designed as an information kiosk at the entrance of a museum. The application consists of several windows (private workspaces) that are assigned for groups and individuals, displaying contents taken from the Rijksmuseum (Amsterdam) website (www.rijksmuseum.nl/en). ProximityTable employs two proxemic interaction zones defined by Hall [41]: personal space (45-120 cm) and intimate space (<45 cm). Following the concept of these two zones, I designed five application behaviors, as follows:

1. Creating/removing workspaces based on the presence and absence of users in the tracked region.

2. Moving workspaces based on users' movements around the tabletop display.

3. Recognizing bystanders when they enter the intimate space of existing users while still in the personal space of the tabletop display.

4. Changing the size of the workspace when an existing user enters another user's intimate space. This change allows users to have a bigger workspace or reduce the enlarged workspace's size if they were grouped and broke the group.

5. Switching the orientation of the workspace as users move to a different side of the table.

Implementing these behaviors in ProximityTable made it clear that the development of proxemics-aware applications can quickly become complex. To illustrate, I will explain the process of implementing behavior 4 (increasing the size of a workspace when two existing users create a group). After receiving proxemics data of entities through DT-DT, ProximityTable will perform several calculations to determine if the distance between entities is within a predefined threshold. First, it checks to see if both users have workspaces, so that we have two active users and not a user and a bystander. Second, it calculates the following distances: user1 to user2, user1 to display, and user2 to display. If all distances are within predefined thresholds, the system places a "join areas" button on the display between the workspaces and attaches a touch event listener to the button. If the event is triggered, the size of one workspace (the one with the stable user) will increase, and the other workspace (with the user who moved to the other user) will be completely removed. After enlarging the workspace, when the two users walk away from each other, the system retrieves the original size of the existing workspace and creates a new workspace for the other user after five seconds from moving apart. In this example, the system performs two tests: both users are within a predefined threshold to the tabletop, and the distance between both users is greater than showing the button threshold (grouping threshold). Suppose the two users walk away from each other before the button is clicked (users still have their private workspaces), the system will perform the same test for ungrouping discussed above. If we have a bystander, s/he is in the intimate space of the user while not having a workspace; the system will perform similar tests to check the distance between all three entities (user1 and bystander, user1 and the display, and bystander and the display) to notify the user about the bystander and maybe enlarge the workspace. After enlarging the workspace, when one user leaves the tabletop, the system will check the three distances again to draw the response (retrieve the original size of the workspace for the existing user).

In this example, one can see three Relative Proximity Rules (distance between user1 and user2, user1 and the display, and user2 and the display) and one UI rule (touching event). Existing toolkits only partially support these requirements. For example, both DT-DT and The Proximity Toolkit can provide properties of entities (e.g., ID, position). While DT-DT leaves it to the developer to manually calculate relationships, as shown in this example, the Proximity Toolkit takes these data one further step and processes them to provide different relations between entities (e.g., the direction of a person related to the smartboard). Since Proximity Toolkit can provide developers with relative relationships between two entities, developers will retrieve three relative distances between the two users and the display and test them in the UI layer. If the test passes, the button will be showing, causing the application to listen for a touch event. ProxemicUI goes further by

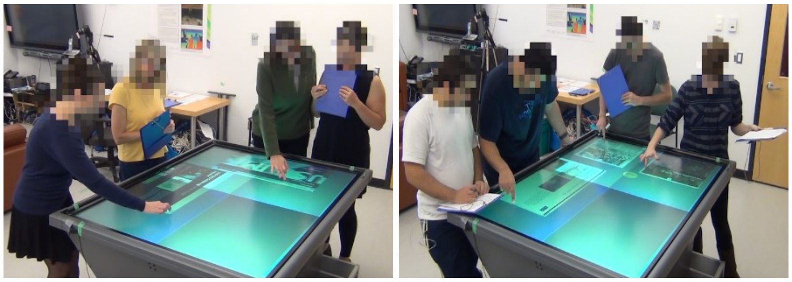


FIGURE 4: PARTICIPANTS WITH DIFFERENT SIZES OF WORKSPACE AT PROXIMITYTABLE

providing an encapsulation mechanism that allows testing these relations between multiple entities and associating a corresponding UI behavior into a single custom rule. For example, instead of removing one workspace in the event of grouping, the system might overlap both workspaces allowing users to switch between them through little taps on top (as suggested by one participant). To break this grouping, one user can touch the tap of his workspace then walk away from the other user. In this scenario, the system detects the touch event then starts testing spatial relationships (distance between user1 and user2, user1 and the display, and user2 and the display). ProxemicUI would implement this with a *compound rule* inside a *hybrid rule*.

ProximityTable could not keep track of the user's identity as it uses DT-DT as a source of tracking data. With DT-DT, every time a user moves out of the tracking area and comes back, DT-DT will assign a new ID. Considering we have a tracking system that can keep track of an entity's identity, each entity in ProxemicUI has an activation status property (active/inactive). This property will change to inactive if the entity is no longer being tracked (e.g., out of the tracking range or in sleep mode). When the entity comes back, the property will change to active.

Evaluation

This section briefly discusses the outcomes from the focus group evaluation. More details can be found in [4][3].

I recruited three groups of four participants: a group of Computer Science students, a group of Biology instructors, and a group was from the Science Atlantic association. The study involved completing information-seeking and planning tasks using the ProximityTable application as though they had arrived at Rijksmuseum, and then taking part in a group critique and design brainstorming session. Tasks were designed to elicit different collaboration styles: tightly coupled coordination in one task and loosely coupled / parallel work in another. The tabletop's proxemic-aware behaviors responded to all participant movements throughout the tasks, in the manner described above.

I now summarize the study findings in relation to how they influenced the design of ProxemicUI.

- Working as a group:

  Although the design of the first task allowed working independently, all pairs chose to work collaboratively on a single workspace. During the first task, P1G3/P2G3 decided to split their workspace to find items independently. P3G3/P4G4 were motivated by P1G3/P2G3 and decided to split their workspace as well. In the second task, four out of six pairs chose to work on a joined workspace throughout the task. This shows the benefit of detecting proxemics relationships between entities to support the user's decision (e.g., detecting distance relationships to determine when to work as a group or independently). ProxemicUI refers to this as relative proximity rules, and it is commonly used in existing proxemic-aware systems.

- Automatic vs. manual control of proxemics-based system responses:

  ProximityTable has a mix of automatic (proxemic-based) and manual (through UI) control. For example, it requires users to touch a button to join workspaces. On the other hand, it waits for 5 seconds before automatic splitting when users break their group. Participants tended to prefer manual control via a confirmation mechanism (e.g., providing a button to join workspaces). Automated control could lead to confusing situations during tasks: for example, when participant P4G3 moved back from the table to make room so her partner P3G3 could move their shared workspace, the system decreased the size of the workspace, and created a brand new workspace for her when she returned, as she had left tracking range and was treated as a new user. This motivated the design of *hybrid rules* (that is, the coupling of proxemics events and other events such as UI interaction) in ProxemicUI to allow developers to manage user-confirmed vs. automated behavior.

- ❖ Classifying new Users:

  As a user of ProximityTable reaches the display, a private workspace will be created for that user if they are not in the intimate zone of an existing user (not classified as a bystander). In practice, two factors play an important role in detecting users, bystanders, and strangers: *distance*, where we detect how far a user is from another user and the display, and *orientation*, which can be used with distance to detect an f-formation, and so who is considered a direct participant in the interaction session. This motivated the design of *compound proximity rules* in ProxemicUI.

- ❖ Moving content:

  Participants appreciated that the workspace moved according to the user's movements but not for every task. For example, you don't want your workspace to follow you while you are moving temporarily to acquire a tool. This also supports our earlier discussion about manual vs. automated control and the need to have hybrid rules. Participants also suggested several usages for the moving content feature. For example, supporting location-based actions (e.g., display a user's documents on a projector when they move to a certain location during a meeting). This motivated the design of *absolute proximity rules* in ProxemicUI, rules triggered based on proximity to a specific location.

Summary

The main outcomes from my experience implementing and evaluating ProximityTable are the need to define two types of tests: compound tests and hybrid tests. Compound testing tests multiple proximity attributes or a single proximity attribute but with multiple thresholds at once. For example, creating a new workspace when the user is within 1m to the table *and* facing the table. A second example is recognizing a bystander standing behind the user; this example includes detecting two distances from the user and the bystander to the table. Hybrid testing combines proximity tests with UI events. For example, when two workspaces are overlapped as a response to grouping event, a user can touch one workspace and start moving to break the group and split the workspaces. Table 1 in Chapter 4 lists existing work that can benefit from such features. Compound and hybrid testing are two key features in ProxemicUI as I will discuss in chapters 4 and 5.

Chapter 4: ProxemicUI Version 1

The remaining chapters detail work completed during my Ph.D. In this chapter, I define four-core design requirements for ProxemicUI, which cover features important for a wide range of proxemics-aware applications, and describe version 1 of ProxemicUI, which I built according to these requirements. The work in this chapter is presented in the following paper:

Mohammed Alnusayri, Gang Hu, Elham Alghamdi, and Derek Reilly. 2016. ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on large displays. In Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16). Association for Computing Machinery, New York, NY, USA, 50–60.

Design Requirements

During the implementation of ProximityTable, it was difficult to keep track of different proximity events giving that I tested two thresholds for four users simultaneously (e.g., create a new workspace and grouping/ungrouping events). These tests involve calculating/checking multiple distances simultaneously. Besides, it was complicated to intertwine distances and UI events to join workspaces. The focus group evaluation emphasized the need to integrate proximity events with user inputs to validate the meaning behind user's behaviors. Considering these difficulties, I identified four design requirements to cover a wide range of proxemic-aware applications: entities, relative proximity rules, compound rules, and hybrid rule. Thereafter, I reviewed some of the existing work in the literature to explore such requirements to support my arguments. This exploration was done by reviewing the proximity interaction techniques (e.g., responding to the relative distance between entities and tracking groups' behaviors) that different systems support and examining how to implement such techniques. Finally, I looked at what some existing toolkits provide to support these requirements to identify the research gap in the literature. To summarize, the four requirements presented here are derived from the literature review of existing work, the development process of ProximityTable, and the findings of the focus group study.

Entities

Entities are the base of any proxemics-aware application, where systems cannot test rules or fire events without knowing about the presence of these objects and their basic properties. Each entity has a set of low-level properties that are used to test any of the association rules. These properties include class (e.g., user and large interactive display), identity, position, orientation, and velocity (if mobile). In addition, I identify two entity “statuses”: mobile and static. This is meant to assist in rule formation when there are a variety of tracked entities, where some normally have static status (e.g., a table), some are mainly mobile (e.g., people and handheld devices), while others have switchable status (e.g., chairs, laptop computers).

One of the fundamental aspects of proxemic-aware applications' development is to measure and understand the spatial relationships between entities. I identified three types of rule to achieve this, as presented in the next sections.

Relative Proximity Rules

Relative Proximity Rules (RPRs) refer to measuring the relationship between two or more entities based on a single property of these entities. For example, creating a workspace for a new user as s/he reaches the tabletop display in ProximityTable. This example measures the relative distance between the user and the tabletop. This type of rule is commonly used in the literature, as seen in table 1.

Absolute Proximity Rules

We note it is also desirable to include Absolute Proximity Rules (APRs) (i.e., rules triggered by proximity to a specific location), as evidenced in some focus group feedback and in prior work. For example, the focus group participants suggested supporting location-based actions (e.g., display a user's documents on a projector when they move to a certain location during a meeting). The AirPlayer [107] also suggests the use of absolute proximity rules. For example, playing favorite songs based on location on the house (e.g., play music on living room speakers if the user is in the living room or on bedroom speakers if the user is in the bedroom). These are supported in a subsequent version of PUI, discussed later.

Compound Rules

Compound Proximity Rules (CRs) measure multiple relationships between *two or more* entities based on *one or more* properties. For example, ProximityTable measures one property (relative distance) between multiple entities (User1, User2, and Table) to support grouping and splitting. This is considered as CR because of the two thresholds (the personal zone between users and the table and the intimate zone between users themselves). Other examples might require measuring multiple properties (e.g., distance and orientation).

While CRs were directly motivated as a way to address the complexity of testing multiple attributes for multiple entities at UI layer during the implementation of ProximityTable, the need for such rules can similarly be seen in much prior work. For example, responding to changes in group formation when arriving to and leaving a display at different times [73], responding to changing roles (e.g., user or observer) [11], and distinguishing users and bystanders [60]. In these examples, we might start with testing the distance between a person and a tabletop display to start the interaction. Later on, when a new user joins an existing user at the display, we might need to test the distance and the orientation between these entities to detect what type of F-formation they have formed. Different F-formations might mean different user roles (e.g., user vs. bystander), which means different system responses. ProxemicUI allows developers to create two basic relative proximity rules (distance and orientation); then combine them into a single compound rule to ease this process. This way, a developer can write a single handler method based on a compound event instead of listening for discrete proxemic events (e.g., a single distance or orientation relation between two entities) and then manually checking to see whether additional criteria are met before responding.

Hybrid Rules

Hybrid Rules (HR) refers to connect a Proximity Rule (RPR or CPR) and UI event. The ability to deeply integrate UI events with proxemic events follows from our focus group participants’ concerns about retaining user agency. This integration can happen in two different ways. In the first approach, the application waits for a Proximity Rule to be triggered and then starts to listen for a UI event. This was implemented manually in ProximityTable, by providing a button once two users get close to each other, and if the button is touched increases the size of the workspace. When following this approach, an application may remove the UI component and/or its event listener once the proximity rule no longer holds true. In the second approach, the application listens for a UI event, and if it is fired, it immediately tests an associated proximity rule, which influences the response to the event. For example, during the evaluation of the ProximityTable, one of the participants of the focus group study suggested that to include tabs in grouped workspaces where each tab would be associated with the state of an individual workspace before merging. Then, the tab could be dragged to separate the individual workspace from the group and restore it. In this case, the system detects the touch interaction (dragging the workspace), then checks the position of the user relative to the display to associate the workspace to that user.

TABLE 1: LIST OF THE DESIGN REQUIREMENTS WITH EACH RELATED WORK THAT CAN MAKE USE OF THE REQUIREMENT

ProxemicUI version		Requirement	Related work that could use
Requirements of ProxemicUI version 2	Requirements of ProxemicUI version 1	Entities	They are the base of proxemic aware applications and are used in existing tool kits such as: Pérez et al. [82][83][86], the Proximity Toolkit [69]
		Relative Proximity Rules	Vogel and Balakrishnan [120], Medusa [7], Screenfinity [97], SpiderEyes [28], Eyes-Free Art [88], Proximity-Aware control [62], Proxemic Brainstorming application [68], Proximity Peddler [122]
		Absolute Proximity Rules	The AirPlayer [104], Savannah [14], Williamson [125], Cumin [25]
		Compound Rules	Klinkhammer et al. [58], Dostal et al. [29], The Proxemic Media Player application [13], Tafreshi et al. [91][107], Savannah [14], TouristPlanner [71], Physio@Home [60]
		Hybrid Rule	Eyes-Free Art [88], the AirPlayer [104]
		Mobility Rule	The AirPlayer [104]
		Test condition	Klinkhammer et al. [58], Savannah [14]
		Activation status	EasyGroups [52][53], FlexiGroups [54]
		F-formation	TouristPlanner [71], Azad et al. [11], Mentis et al. [73], GroupTogether [70], Lucero et al.[65][66], Zhou et al. [129]

Fulfilling the Requirements: ProxemicUI v1

Most tracking toolkits (e.g., DT-DT [49]) provide developers with low-level data about entities in an environment(e.g., identity and position), but leave application developers to determine proximity relationships. Other toolkits use the positional data to generate a set of relative relationships between entities that can then be tested or queried (e.g., the Proximity Toolkit [71]). Proxemic UI further extends toolkit support for proxemic-aware applications by generating proxemic events based on Compound Proximity Rules and Hybrid Rules (Table 2). These two rules types are important when developing proxemic-aware applications as they reduce the complexity of dealing with low-level positional data and many unitary, primitive proxemic relationships (including the need for potentially complicated formulas, repeated tests, and long conditional expressions) by providing developers with a clear mechanic for rule expression, and by letting developers manage test results using a familiar event handling paradigm.

TABLE 2: FULFILLING THE DESIGN REQUIREMENTS

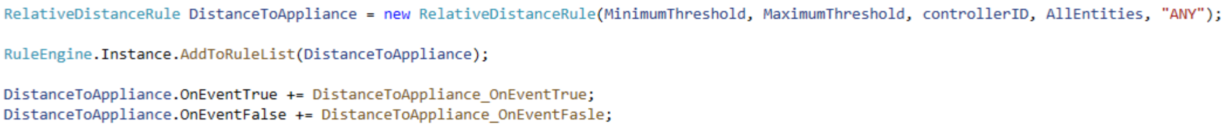
Types of proximity data	DT-DT	The Proximity Toolkit	ProxemicUI
Raw Proximity Data (e.g., position & orientation)	✓	✓	✓
Basic Proximity Data (e.g., relative distance)	-	✓	✓
Compound Test (e.g., two different attributes, or one attribute with two thresholds)	-	-	✓
Hybrid Test (e.g., Basic/Compound test with UI events)	-	-	✓

ProxemicUI Version1 Structure

ProxemicUI v1 follows the *Intermediate* approach, one of the ways to address the infrastructure problem in HCI, discussed by Edwards et al. [32]. The *Intermediate Approach* refers to frameworks or toolkits that “sit atop of a layer of more fundamental infrastructure”, and its goal is to “overcome the limitations of existing infrastructure and ease the construction of novel applications” [32]. ProxemicUI v1 follows this approach, where it uses any low-level tracking system (e.g., DT-DT or HTC Vive Lighthouse) as a source of position and orientation data. It can also integrate the Windows event model as a source of UI events. ProxemicUI v1 then generates higher-level proxemics events using these Low-level tracking data and UI events. ProxemicUI v1 defines a set of rules where it processes the input data (tracking data and UI events) to generate these proxemics events. Application developers can customize these rules according to application requirements. They also can define new rules or extend existing rules to match application requirements. Finally, when the defined rules fire an event, developers then can create application responses accordingly. In this section, I will discuss the architecture of version 1 of ProxemicUI.

ProxemicUI v1 receives two types of input data: proximity data from tracking systems (e.g., DT-DT and HTC Vive Lighthouse) and UI events (e.g., touch events). The proximity data includes information about every participating entity in the environment (e.g., ID, position, and orientation). ProxemicUI v1 implements a communication gateway to receive these tracking data from any tracking system by employing the OSC communication protocol [15][16][103]. When ProxemicUI v1 receives these tracking data, it creates an object for every entity to store its data. Creating objects for entities and storing their data fulfills the first design requirement that is detecting entities and knowing their data, which is the base of any proxemic-aware application.

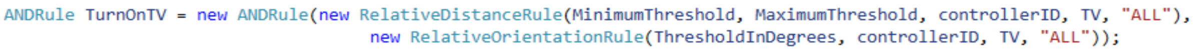
ProxemicUI v1 defines a set of parameterized proximity rules, including Absolute Proximity Rules (e.g., absolute position) and Relative Proximity Rules (e.g., relative distance). When developers initiate a rule, they define its parameters, including what entities to test and the thresholds to use for the tests. At the application layer, developers will add the initiated rule to the Rule Engine, which continuously tests all active rules. Developers will also subscribe to listeners of different events of the initiated rule. Figure 5 shows an example of initializing a proximity rule at the application level. Predefining these proximity rules fulfills the second design requirement that is testing different proximity attributes for participating entities or between them.



```
RelativeDistanceRule DistanceToAppliance = new RelativeDistanceRule(MinimumThreshold, MaximumThreshold, controllerID, AllEntities, "ANY");

RuleEngine.Instance.AddToRuleList(DistanceToAppliance);

DistanceToAppliance.OnEventTrue += DistanceToAppliance_OnEventTrue;
DistanceToAppliance.OnEventFalse += DistanceToAppliance_OnEventFasle;
```

Figure 5: example of initializing a Relative Distance Rule

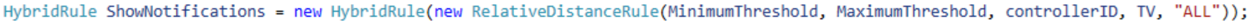
ProxemicUI v1 employs the little language pattern to compose basic proximity rules to more complex rules (Compound Rules) via basic logical operators (AND, OR, XOR, and NOT). When a complex rule is defined, each sub-rule (basic rule) will have its parameters specified. Developers will only need to add the most parent class to the Rule Engine. They also need to subscribe to listeners of the most parent rule only. Defining such complex rules fulfills the third design requirement by allowing developers to define rules that simultaneously involve testing multiple proximity attributes and/or multiple thresholds. Figure 6 shows an example of initializing a compound rule.



```
ANDRule TurnOnTV = new ANDRule(new RelativeDistanceRule(MinimumThreshold, MaximumThreshold, controllerID, TV, "ALL"),
                                    new RelativeOrientationRule(ThresholdInDegrees, controllerID, TV, "ALL"));
```

Figure 6: Example of initializing a Compound Rule (ANDRule)

Finally, ProxemicUI v1 integrates any of the rules discussed above (Relative Proximity Rules, Absolute Proximity Rules, and Compound Rules) with UI events using the Hybrid Rule. A Hybrid Rule contains a single parameterized rule that can be any of the rules discussed above. The Hybrid Rule also has a listener for the UI events to test the proximity rule as soon as the UI event occurs. Developers would subscribe to the Hybrid Rule events only, which will be fired when both proximity events and UI events are fired. Figure 7 shows an example of initializing a Hybrid Rule. The ability to integrate the proximity events with UI events in ProxemicUI v1 fulfills the fourth design requirement.



```
HybridRule ShowNotifications = new HybridRule(new RelativeDistanceRule(MinimumThreshold, MaximumThreshold, controllerID, TV, "ALL"));
```

FIGURE 7: AN EXAMPLE OF INITIALIZING A HYBRID RULE

Summary

This chapter discussed the initial design requirements that I used to build ProxemicUI v1. It also discussed the basic structure of ProxemicUI v1. I evaluated ProxemicUI through three different methods as I will discuss in the next chapters, and each evaluation method produced a set of design improvements. The next chapter (chapter 5) discusses the full architecture of ProxemicUI v2 including all improvements based on the evaluation results. Table 3 shows the key changes made to ProxemicUI v1 to generate ProxemicUI v2.

TABLE 3: KEY CHANGES TO PROXEMICUI V1

Features	Rationale	Implementation
IsFacing Rule	Allows to test relative orientation based on the data of a single entity	Subclassed from the Relative Orientation Rule
Mobility Rule	Allows to check and update the mobility status of each entity	Subclassed form the base Rule class
Generic Hybrid Rule	Allows to combine proximity events with external events beyond UI events	Updated the existing Hybrid Rule class
Activation status	Allows to track existing entities in the space	Added as property to the Proximity Entity
Test Condition	Allows to apply the proximity test to all entities or just a subset of them	Added as an argument to all basic rules

Chapter 5: ProxemicUI Architecture Version2

Chapter 4 discussed the ProxemicUI v1 and the design requirements that I used to build it. I evaluated ProxemicUI through employing ProxemicUI with the proof-of-concept applications, code review study, and integrating ProxemicUI into Story CreatAR, as I will discuss in the next three chapters. This chapter presents ProxemicUI version 2 with consideration to the outcomes of the three evaluation methods I mentioned above. This chapter starts with a "How to" scenario explaining step-by-step how to use ProxemicUI. Then, it will explain in detail the architecture of ProxemicUI v2.

Why ProxemicUI

This section will discuss why I implemented a new proximity tool and not expanded existing tools such as the Proximity Toolkit. I am considering the Proximity Toolkit because it is the closest to ProxemicUI in terms of intended use.

1- *Different programming model*: the programming model for the Proximity Toolkit provides developers with proximity data (e.g., the distance between two entities) where they handle the tests in the handling methods (e.g., check if the distance within thresholds to draw a response). However, while the programming model for ProxemicUI provides information about proximity events to developers (e.g., which entities passed the test), it encapsulates testing the proximity data in the rules. In other words, when the event is triggered, developers do not need to test the proximity data to check if they are within thresholds or not.

2- *Platform support*: the Proximity Toolkit relies on old tracking libraries (e.g., Kinect and VICON) that are not compatible anymore, which will require a significant amount of work to bring it back to work.

3- *Extensibility*: while the Proximity Toolkit provides a set of ready-made features to support the implementation of proxemic-aware applications, it was not designed specifically for extensibility. On the other hand, ProxemicUI provides a set of well-articulated extension points (will be discussed at the end of this chapter), making ProxemicUI more extensible than the Proximity Toolkit.

How to Use ProxemicUI

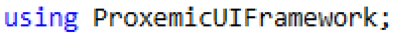
ProxemicUI is a framework that is designed for application developers and not for end-users. Therefore, this section discusses in detail how a developer would use different features in ProxemicUI. This discussion plays two important roles. First, one of the evaluation techniques through demonstration is the “How To” scenario, which is a “step-by-step breakdown of how a user creates a specific application” [65]. This discussion applies this technique for evaluating UI toolkits. It also functions as a tutorial demonstrating how a developer uses ProxemicUI in an actual use case scenario.

To better understand how ProxemicUI works, I will explain step-by-step how to use it in a context example that is a smart home setting. In this example, Alice, who is a homeowner and a developer, has several smart appliances in her home that she wants to interact with according to her proximity. For example, Alice wants the system to open the controller app on her tablet for a specific appliance when she gets closer to it (e.g., open the coffee machine interface when she gets closer to it). She also wants the system to detect more complex interactions as follows. She wants the system to turn on the TV and start the controller app on her tablet only if she is within distance and facing the TV simultaneously. In addition, when Alice starts her coffee machine and goes to watch TV, she wants the system to show notifications on the TV that her coffee is ready. Besides, when Alice places her tablet on the kitchen counter for two minutes, she wants the system to open the recipe app on the tablet and keeps the screen on. Finally, when Alice has dinner guests, she wants the system to detect the presence of F-formations and respond accordingly. For example, when guests arrange themselves around the dinner table, the system turns on the top-down projector, and the dinner table becomes a display. Each of these interactions involves using different rules. The rest of this section will demonstrate the steps that show how to use ProxemicUI, which include the following key steps:

- Adding the required DLL files
- Initializing the OSC server and retrieve the entities
- Creating the rules
- Subscribing to the events and writing the handling methods

Adding the required DLL files

Before I get into deep details about using ProxemicUI, I will start by pointing the required steps to set up the environment. First, adding the required DLL files (ProxemicUIFramework.dll, Bespoke.Common.dll, and Bespoke.Common.Osc.dll) to the project. The ProxemicUIFramework DLL file reference the framework, and the other two Bespoke files for the OSC communication setup inside ProxemicUI. In Visual Studio, Alice would click on “Project” from the menu bar, then navigate to “Add Reference” and click it. A popup window will open, where she needs to click on the “Browse” button, then navigates to where the three DLL files are located; she selects the files, then clicks “Add” to add them to her project. With some other systems like Unity, drag and drop the DLL file somewhere in the Assets folder adds them to the project. Second, after Adding the DLL files, Alice should add the namespace of ProxemicUI to the top of her code to access its features, figure 8. Now, everything is ready to start detecting and responding to Alice’s proximity in her home.

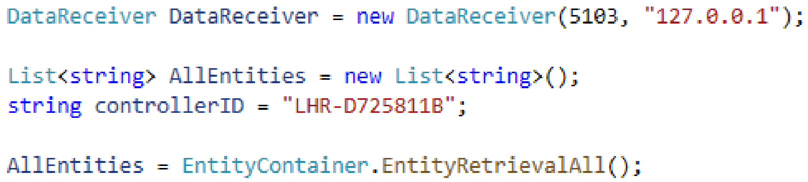


```
using ProxemicUIFramework;
```

FIGURE 8: ADDING THE NAMESPACE OF PROXEMICUI

Initializing the OSC server and retrieving entities

Before creating any rules, Alice would start with initializing the OSC server to receive tracking data from the tracking system, the first line in figure 9. Because Alice wants to control multiple devices through her tablet (controller), she would use one-to-many tests (ProxemicUI allows for one-to-one, one-to-many-, and many-to-many tests). Therefore, she defines two variables: a string for the controller id (her tablet) and a list of strings for the ids of other devices (second and third lines in figure 9). She then would retrieve ids of entities (using one of the methods discussed in the *EntityContainer* in chapter 8) and put them in the list she defined earlier (fourth line in figure 9). Now she is ready to initialize the rules she needs.

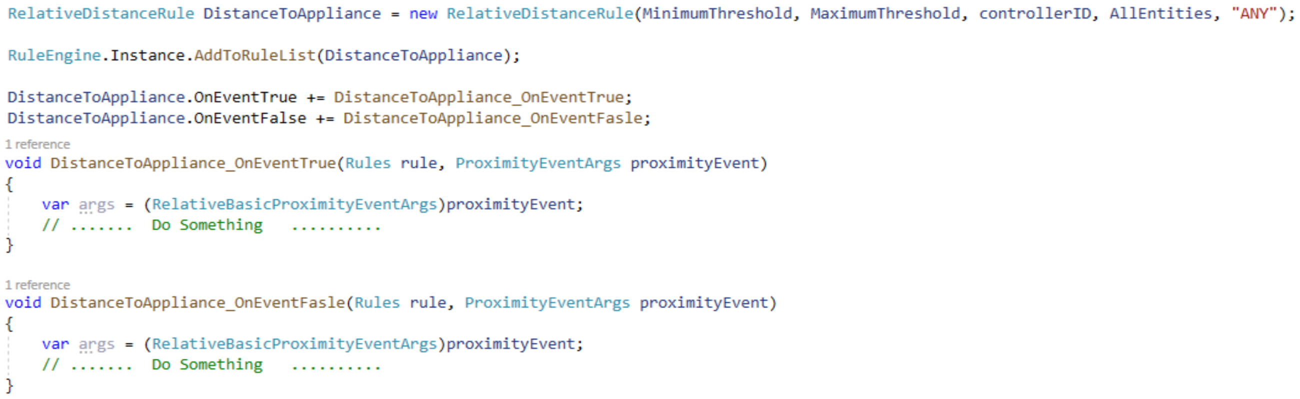


```
DataReceiver DataReceiver = new DataReceiver(5103, "127.0.0.1");

List<string> AllEntities = new List<string>();
string controllerID = "LHR-D725811B";

AllEntities = EntityContainer.EntityRetrievalAll();
```

FIGURE 9: INITIALIZING THE DATARECEIVER SERVER AND RETRIEVING ENTITIES

Creating a RelativeDistanceRule

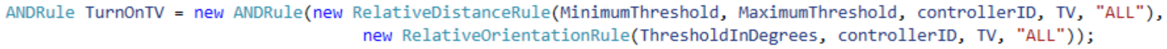
To check the distance between Alice’s tablet and other appliances, she initializes a *RelativeDistanceRule* (first line in figure 10) and passes all required arguments to complete the test. These arguments are minimum and maximum thresholds, the two lists of entities (change depending on the test type, e.g., one-to-many), and the test condition. After completing the initialization, she would add the rule to the *RuleEngine* to be tested (second line figure 10). Then, she would subscribe for *OnEventTrue* and *OnEventFalse* to be notified when the events are fired (third and fourth lines in figure 10). The last step for all rules is to write the handling methods for the events. In the handling methods, she



```
RelativeDistanceRule DistanceToAppliance = new RelativeDistanceRule(MinimumThreshold, MaximumThreshold, controllerID, AllEntities, "ANY");

RuleEngine.Instance.AddToRuleList(DistanceToAppliance);

DistanceToAppliance.OnEventTrue += DistanceToAppliance_OnEventTrue;
DistanceToAppliance.OnEventFalse += DistanceToAppliance_OnEventFasle;
1 reference
void DistanceToAppliance_OnEventTrue(Rules rule, ProximityEventArgs proximityEvent)
{
    var args = (RelativeBasicProximityEventArgs)proximityEvent;
    // .......  Do Something   ..........
}

1 reference
void DistanceToAppliance_OnEventFasle(Rules rule, ProximityEventArgs proximityEvent)
{
    var args = (RelativeBasicProximityEventArgs)proximityEvent;
    // .......  Do Something   ..........
}
```

FIGURE 10: STEPS TO INITIALIZE A RELATIVEDISTANCERULE

would start by retrieving the test results and cast them to the appropriate type based on the initialized rule (the two methods at the bottom of figure 10).



```
ANDRule TurnOnTV = new ANDRule(new RelativeDistanceRule(MinimumThreshold, MaximumThreshold, controllerID, TV, "ALL"),
                                    new RelativeOrientationRule(ThresholdInDegrees, controllerID, TV, "ALL"));
```

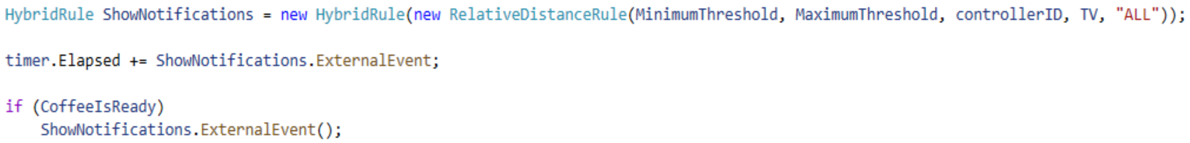
FIGURE 11: INITIALIZING AN ANDRULE

Creating a CompoundRule

To turn on the TV only when Alice is within distance and facing the TV, she would create *ANDRule* (a compound rule) and pass its arguments. These arguments are two sub-rules: *RelativeDistanceRule* and *RelativeOrientationRule*, figure 11. Both basic rules have a one-to-one test (controller to TV), and their test condition is “ALL”. After the initialization, Alice would follow the same steps as the *RelativeDistanceRule* we discussed earlier: adding the *ANDRule* only to the *RuleEngine*, subscribing to the events (*OnEventTrue*/*OnEventFalse*) for the *ANDRule* only, and writing the handling methods for the *ANDRule* only.

Creating a HybridRule

To notify Alice when her coffee is ready on any output device based on her proximity (the TV in this example), she would create a *HybridRule* that takes any rule (ProximityRule or CompoundRule) as an argument (first line in figure 12). *HybridRule* works a bit differently than other rules. It contains the *ExternalEvent* method that can be called as a handler method to any external event (e.g., UI event or system notifications). The *ExternalEvent* method can be called in two ways: when subscribing to an event (second line in figure 12) or directly called when receiving system notifications (third line in figure 12), which is the way to go in our example. After calling the *ExternalEvent* method as a handler method for the external event, she would subscribe to the events for the *HybridRule* (similar to what she did for *RelativeDistanceRule*); she would also write the handling methods for these events. Unlike other rules, the *HybridRule* would not be added to the *RuleEngine* as the test for the proximity rule would take place only once, that is, when the external event has occurred (to make sure they are happening at the same time).

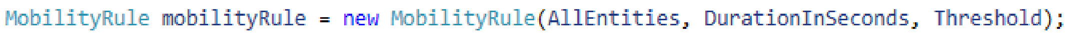


```
HybridRule ShowNotifications = new HybridRule(new RelativeDistanceRule(MinimumThreshold, MaximumThreshold, controllerID, TV, "ALL"));

timer.Elapsed += ShowNotifications.ExternalEvent;

if (CoffeeIsReady)
    ShowNotifications.ExternalEvent();
```

FIGURE 12: INITIALIZING THE HYBRIDRULE

Creating MobilityRule

To open the recipe app on Alice's tablet when she places it on her kitchen counter for two minutes, she would create two rules. First, *AbsolutePositionRule* to check if the table on that specific location in her kitchen. Second, *MobilityRule* to check if the tablet is stable in that position for the duration she needed (in this example, two minutes). The *MobilityRule* checks and updates one or more entities' mobility status. It follows the same approach as all other rules: creating the rule, figure 13, adding it to the *RuleEngine*, subscribing to the events, and writing the handling methods for the events. The duration in this rule is the period between capturing two positions for an entity to check its status. The threshold is the distance between two positions in which the developer considers as movements.

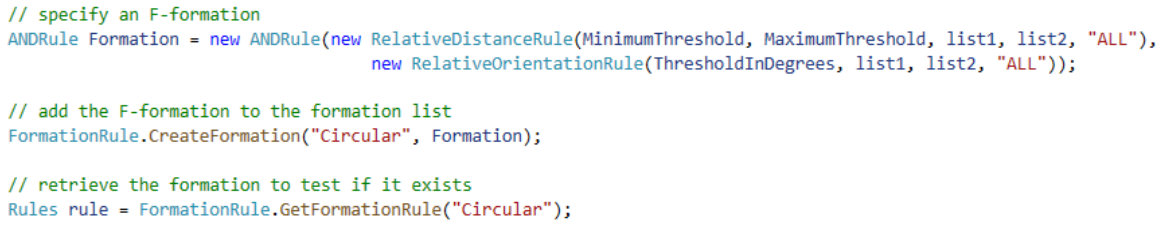


```
MobilityRule mobilityRule = new MobilityRule(AllEntities, DurationInSeconds, Threshold);
```

FIGURE 13: INITIALIZING MOBILITYRULE

Creating F-formation

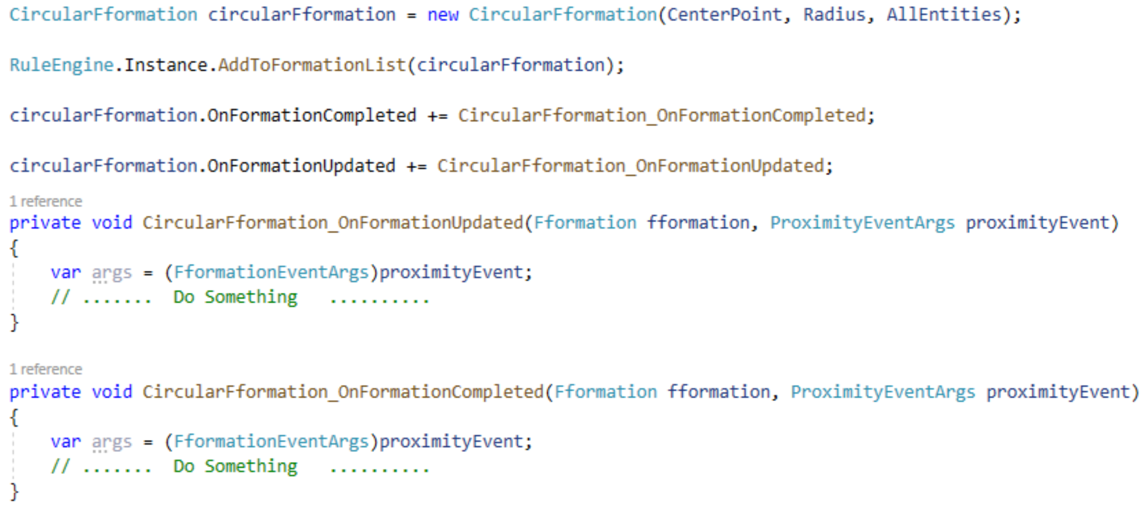
To detect the guests' formation to make the dinner table a display, Alice would start by specifying the required formation. Then she can retrieve this formation to test if it exists in the environment. Figure 14 shows the steps to specify the formation and retrieve it for testing.



```
// specify an F-formation
ANDRule Formation = new ANDRule(new RelativeDistanceRule(MinimumThreshold, MaximumThreshold, list1, list2, "ALL"),
                                  new RelativeOrientationRule(ThresholdInDegrees, list1, list2, "ALL"));

// add the F-formation to the formation list
FormationRule.CreateFormation("Circular", Formation);

// retrieve the formation to test if it exists
Rules rule = FormationRule.GetFormationRule("Circular");
```

Figure 14: Specifying F-formation in ProxemicUI

ProxemicUI can also define the exact placement of entities when an F-formation is created, which can be beneficial in virtual environments. For example, a virtual environment application might require creating conversations between avatars where they are placed on a specific configuration. To use this feature, a developer would initialize the required formation (e.g., CircularFormation), the first line of figure 15. Then,



```
CircularFformation circularFformation = new CircularFformation(CenterPoint, Radius, AllEntities);

RuleEngine.Instance.AddToFormationList(circularFformation);

circularFformation.OnFormationCompleted += CircularFformation_OnFormationCompleted;

circularFformation.OnFormationUpdated += CircularFformation_OnFormationUpdated;

1 reference
private void CircularFformation_OnFormationUpdated(Fformation fformation, ProximityEventArgs proximityEvent)
{
    var args = (FformationEventArgs)proximityEvent;
    // .......  Do Something   ..........
}

1 reference
private void CircularFformation_OnFormationCompleted(Fformation fformation, ProximityEventArgs proximityEvent)
{
    var args = (FformationEventArgs)proximityEvent;
    // .......  Do Something   ..........
}
```

Figure 15: Initializing an F-formation

add the formation to the formation list in the *RuleEngine* (second line in figure 15), which works as the rule list that I discussed above. Each formation has two events: *OnFormationCompleted* and *OnFormationUpdated* (third and fourth lines in figure 15). With these two events, the developer would be able to update the formation if s/he needs to. For example, the developer might create a formation with three avatars, and then, during the gameplay, a new avatar joins the conversation. In such cases, the developer might need to readjust the formation to fit the new avatar in. Therefore, the developer would subscribe to the appropriate event. Lastly, the developer would write the handling method to the event (the two methods at the bottom of figure 15).

System implementation

ProxemicUI consists of seven core classes, where some include several subclasses and helper classes. The main classes are DataReceiver, EntityContainer, ProximityEntity, Geometry, Rules, F-Formation, and RuleEngine, which I will discuss accordingly.

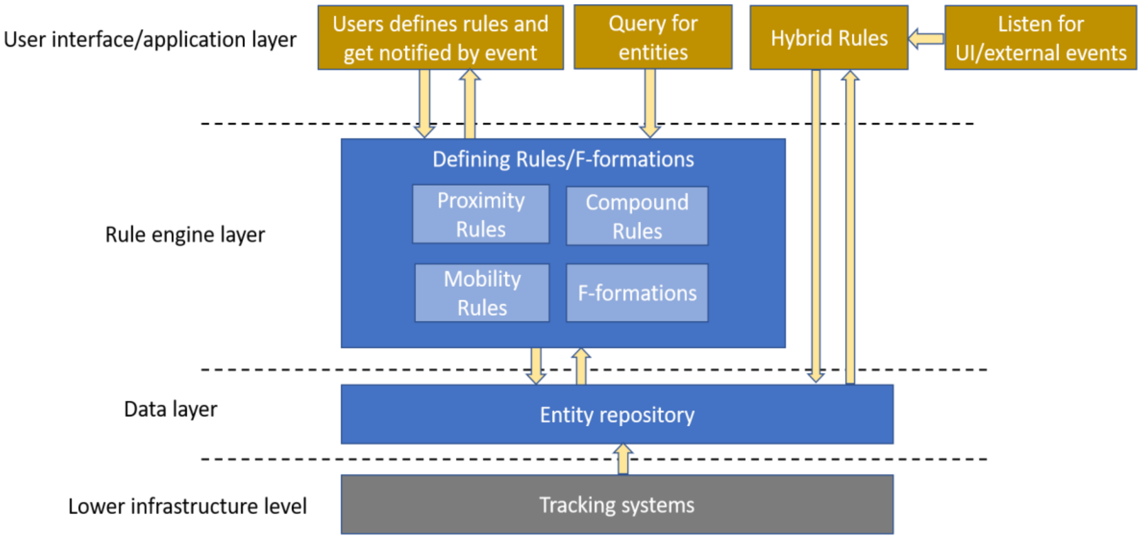

User interface/application layer
Users defines rules and get notified by event
Query for entities
Hybrid Rules
Listen for UI/external events
Rule engine layer
Defining Rules/F-formations
Proximity Rules
Compound Rules
Mobility Rules
F-formations
Data layer
Entity repository
Lower infrastructure level
Tracking systems

FIGURE 16: STRUCTURAL SYSTEM DIAGRAM OF PROXEMICUI FRAMEWORK

DataReceiver

DataReceiver works as a communication gateway between ProxemicUI and tracking systems (e.g., DT-DT and VIVE). It contains a full setup for the OSC communication protocol to receive the proxemics data of tracked entities. Therefore, developers can use any tracking system to capture the proxemics data of entities and forward them to ProxemicUI. To achieve this, developers should write their own OSC sender code to send proxemics data to ProxemicUI from the tracking technology they are using, or use an existing implementation (so far, I have created implementations for DT-DT and HTC Vive Lighthouse). Writing the communicator code is also useful when ProxemicUI receives tracking data from multiple tracking systems. For example, we might receive position from one system and orientation from another; in such a case, ProxemicUI can receive multiple OSC messages for the same entities containing different properties. In the case where redundant data exists (e.g., two tracking systems send position for the same person), developers need to handle this outside ProxemicUI in their OSC communicator code as ProxemicUI makes no assumptions about which data to prefer, nor does it provide built-in sensor fusion capabilities. In addition, the OSC communicator code can be useful to read data from files and send them to ProxemicUI. Appendix 5 shows an example of the OSC communicator code that captures data from the tracking system and read other data from a file, which I used for the smart home setup. The format of their messages should follow the same format as the messages in ProxemicUI. Upon receiving the OSC message, *DataReceiver* breaks it down and passes data of each entity to the EntityContainer class through the *EntityChecker* method.

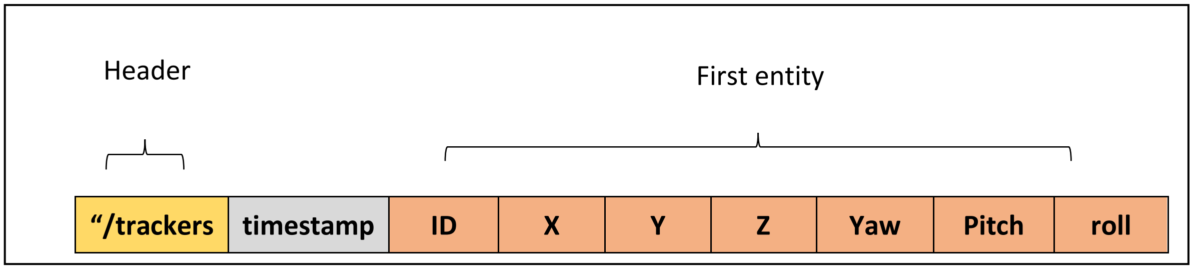


FIGURE 17: BASIC FORMAT FOR RECEIVED MESSAGE

The *DataReceiver* class has one primary message format that contains the most common 3D properties of entities as follows: *timestamp*, *ID*, position data (*X, Y, and Z* axes), and orientation data (*roll* is rotating around the front to back axis*, pitch* is rotating around side-to-side axis*, and yaw* is rotating around the vertical axis), see figure 17. Application developers can send any additional properties in a different message following another format; ProxemicUI then differentiates between messages using the header. The *DataReceiver* class also defines a second message format that was used during the proof-of-concept applications evaluation. This message format includes a different set of properties of entities (in this case, properties are the *width* and *length* of a tabletop), figure 18. Developers can add different formats according to their needs (e.g., to received data about different shapes such as radius for circular entities). When defining a new message format, developers would make changes to two classes. First, in the DataReceiver class, developers will implement their method that breaks and extracts the new message's data (this process will follow the new message format). Then, they will pass these data to the EntityContainer class, where they will implement a method that takes these data and assign them to existing entities or create new entities as required. I chose to follow this approach for two reasons. First, why do we provide application developers with extra data when all they need is the basic proxemics data; this is important when the developers must write the communicator code. Second, some tracking systems might not provide these extra data (e.g., *width* and *length*). For example, initially, I built ProxemicUI based on the use of DT-DT tracking system. But I had to make changes to the received message format and the corresponding part of ProxemicUI code to use the VIVE tracking system.

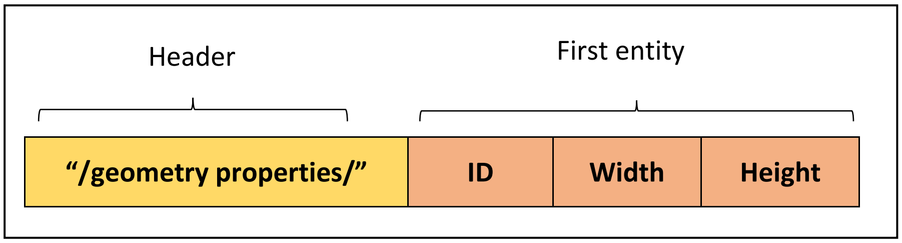


FIGURE 18: ADDITIONAL MESSAGE FORMAT FOR EXTRA DATA

EntityContainer

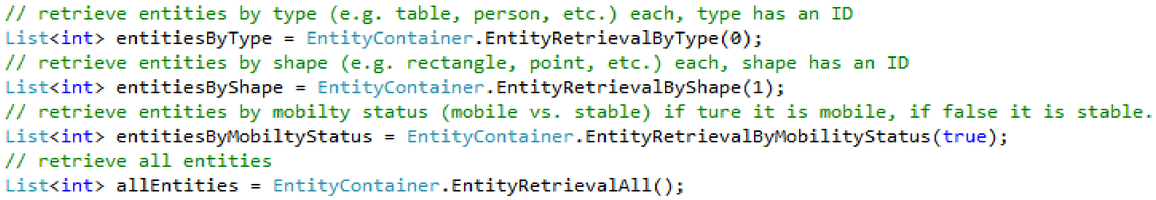
EntityContainer receives the data of entities (e.g., *ID, x, y*, etc.) from *DataReceiver* through a method called *EntityChecker*. Upon receiving proxemics data, *EntityContainer* performs one of two actions. First, if an entity already exists, it calls the *UpdateProperties* method from *ProximityEntity* class to update the entity's data. On the other hand, it creates a new instance of ProximityEntity and assigns its data if it is a new entity. Developers might also create/update entities manually if they wish to. For example, some entities are stable, and their data is static, so ProxemicUI would need to know about these entities only once. For such a case, the developer can create a new instance of ProximityEntity and pass its data manually if they don't want to include the data in the OSC messages.

Querying entities

In addition, *EntityContainer* contains several methods that retrieve references to entities based on different properties (*typeId*, *shapeId*, *isStable*, and *isActive*; more details are in the *ProximityEntity* class description). These methods are listed below, and figure 19 has examples of these methods.

- *EntityRetrievalByType* method retrieves entities by type (e.g., table, person, cellphone, etc.) through testing the received ID against the *typeId* of each entity.
- *EntityRetrievalByShape* method retrieves entities by shape (e.g., rectangle, point, etc.) through testing the received ID against the *shapeId* of each entity.
- *EntityRetrievalByMobilityStatus* method retrieves entities by mobility status (e.g., mobile, stable) through testing the received status against *IsStable* of each entity. If the received status is true, it will return all stable entities; and if the received status is false, it will return all mobile entities.
- *EntityRetrievalByActivationStatus*. It retrieves entities by activation status (e.g., active, inactive) through testing the received status against the status *IsActive* of each entity. If the received status is true, it will return all active entities; and if the received status is false, it will return all inactive entities.
- *EntityRetrievalAll* method retrieves all entities.

- *EntityContainer* has *EntityInactiveEvent* that keeps track of the last truth value of the activation status and notifies listeners of the change. Developers can subscribe to listen to this event through the EntityContainer class.



```
// retrieve entities by type (e.g. table, person, etc.) each, type has an ID
List<int> entitiesByType = EntityContainer.EntityRetrievalByType(0);
// retrieve entities by shape (e.g. rectangle, point, etc.) each, shape has an ID
List<int> entitiesByShape = EntityContainer.EntityRetrievalByShape(1);
// retrieve entities by mobilty status (mobile vs. stable) if ture it is mobile, if false it is stable.
List<int> entitiesByMobiltyStatus = EntityContainer.EntityRetrievalByMobilityStatus(true);
// retrieve all entities
List<int> allEntities = EntityContainer.EntityRetrievalAll();
```

FIGURE 19: EXAMPLES OF ENTITY RETRIEVAL METHODS IN ENTITY CONTAINER CLASS

ProximityEntity

ProximityEntity is the class that represents an entity. To create a new entity, developers need to define a new instance of this class. It consists of a constructor, several methods, and getters/setters to store, retrieve, and update properties of entities. To assign properties of an entity, developers can define new constructors or use setters in *ProximityEntity*. *ProximityEntity* also has the *UpdateProperties* method to be called to update properties. The *UpdateProperties* method can be used to update position and orientation data at once. On the other hand, the setters and getters allow developers to set/update or get each type of data individually. *ProximityEntity* class contains the following properties. First, *entityID* is a string type to support recording different identification numbers from different types of tracking systems. Second, 3D proximity data, including position (*x, y, z*), and orientation (*yaw, pitch, roll*). Third, *timestamp*, which might be useful for calculating other temporal properties (e.g., velocity) or processing data that arrives out of sequence or with a delay. Fourth, *IsActive* (active vs. inactive)*,* which will be true as default when the entity is created. Then, ProxemicUI updates it according to *EntityInactiveEvent* discussed above. Fifth, *IsStable* (stable vs. mobile), which will be false as default when the entity is created. Then, ProxemicUI will update it according to the test of *MobilityRule* that will be discussed later in this chapter. Lastly, *TypeID* allows the system to distinguish between people, devices, furniture, and etcetera. Currently, ProxemicUI defines four types: person, tabletop, tablet, and TV. These can be expanded to include other objects (e.g., couch and coffee table) as required. I am not considering subclassing for these types as they all share the same properties that the *ProximityEntity* provides. The other unique properties that each of these types has and are required for proxemic awareness are related to their shapes, which will be assigned to their shapes. The usages of each of these properties will be discussed later in this chapter. Since each entity has a shape, next, I will discuss the Geometry class and show its relationship with ProximityEntity.

Geometry

Each entity has a unique shape. While in some cases entities can be considered points for the purpose of analysis, for many proxemics applications, the shape of an entity may be highly relevant. For example, to measure the distance between a person and a tabletop display, do I calculate the distance to the person from the center of the table or its edges? Therefore, to accurately measure proxemic relations between entities, the shape and its measures must be known. In addition, some entities might have dynamic shapes (e.g., KirigamiTable [40], TransformTable [111], and Proxemic Transitions [39]). Due to this shape variety of trackable entities, I defined a shape for each entity. To better understand how this works, I looked at few collision detection algorithms for 2D games. These collision algorithms depend on the type of shapes that might collide. This is similar to what I am trying to do in ProxemicUI, where each entity has a bounding area (sometimes 2 bounding areas if the minimum threshold > 0) around it and its edge is the maximum threshold that we are using to test the collision. In addition, this approach is commonly used in existing APIs and frameworks that support collision detection. For example, Unity is a game engine that supports collision detection between objects in the game scene. It defines a set of colliders for different shapes; these colliders are invisible shapes that handle the collisions between objects in the game scene (similar to the bounding area we have in ProxemicUI).

The shape of an entity is derived from the *Geometry* class, which is the base class of all shapes. Geometry defines two subclasses: *Point* and *Rectangle*; it can be expanded to include other shapes as required. In addition, *Geometry* has several setters and getters to set or retrieve properties of shapes. These setters and getters used to be in *ProximityEntity* class, but I shifted them to *Geometry* class. This way, *ProximityEntity* contains only the common properties between all entities (discussed earlier), where other specific properties (e.g., *width*, *height*) are included in subclasses (e.g., shapes). *Geometry* also has several abstract methods that are called from any rule (will be discussed later in rules) to measure proxemic relations between entities. For example, *CheckDistance* is an abstract method that is implemented in all shapes (derived classes), and its arguments are a *Geometry* instance and two thresholds. More details will be discussed later in the rules.

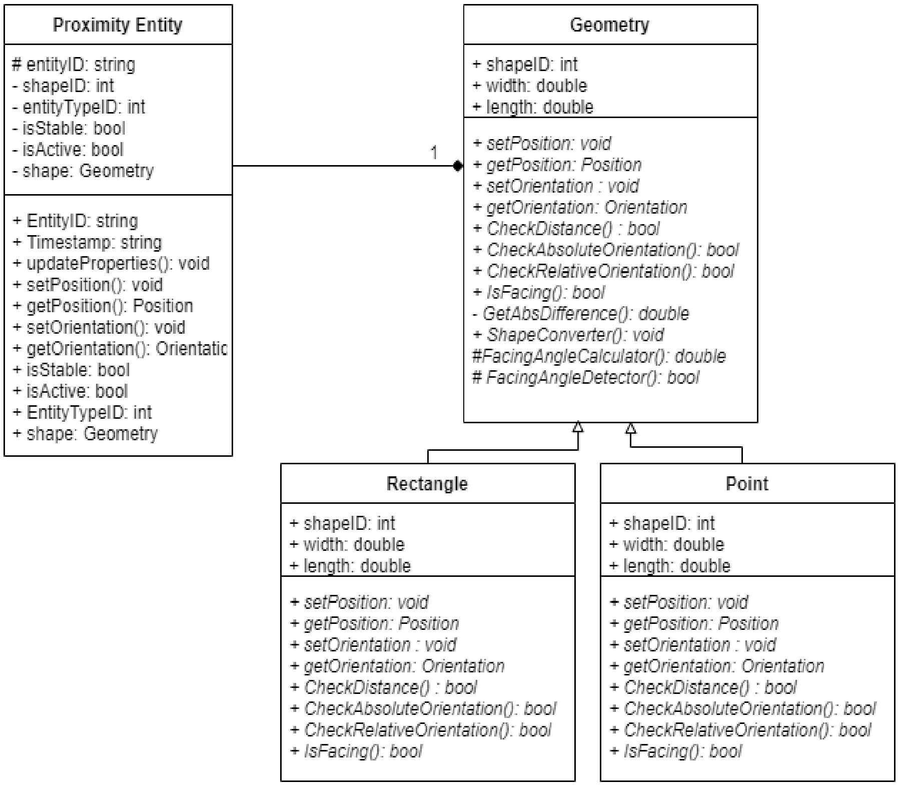
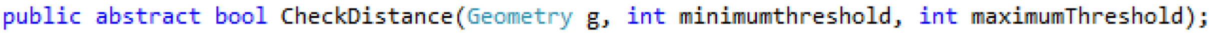


FIGURE 20: CLASS DIAGRAM SHOWN THE RELATIONSHIP BETWEEN A PROXIMITYENTITY AND GEOMETRY

Point

An instance of *Point* class is created in each entity that has a shape of point. It includes several setters and getters to set and retrieve properties of the shape of the entity (*x, y, z, yaw pitch, roll*), figure 20. It used to implement several methods: *CheckDistance*, *CheckAbsoluteOrientation, CheckRelativeOrientation,* and *IsFacing*. These methods are called from different rules, when being tested, through the shape of an entity (will be described later in rules). For example, to check the relative distance between entity A and entity B, the *CheckDistance* method of A's shape will be called, and its arguments will be a *Geometry* instance (B's shape) and the two thresholds (maximum and minimum). The test concept is to perform shape to shape test and return true or false. Therefore, A knows its shape (e.g., *Point*), and it needs to check B's shape. If B's shape is a *Point*, it performs

Point to *Point* check, and if it is a *Rectangle*, it performs *Point* to *Rectangle* check. Then it will return true or false according to the test results. Currently, it has tests between shapes (*Point* and *Rectangle*), and developers can easily add other shapes as required. For example, if a developer wants to add a test for a new shape (e.g., *Circle*), s/he needs to add an "if" statement in the *CheckDistance* method to check what test to perform (e.g., *Point* to *Circle*). This "if" statement contains the formula to test the distance between the *Point* and the edge of the *Circle*, where it returns true if the test passes. The *CheckAbsoluteOrientation* method is overridden where it can check the orientation of a single axis (e.g., yaw within a threshold) or all axes at once (yaw, pitch, and roll). *The CheckRelativeOrientation* method returns true if the relative orientation between two or more entities is within the thresholds. The *IsFacing* method returns true if one or more entities are facing another entity or a point in the space. Finally, the *MobilityCheck* method returns true if one or more entities are mobile.



```
public abstract bool CheckDistance(Geometry g, int minimumthreshold, int maximumThreshold);
```

FIGURE 21: EXAMPLE OF ABSTRACT METHODS IN GEOMETRY CLASS

Rectangle

An instance of the *Rectangle* class is created in each entity that has the shape of a rectangle. It includes several setters and getters to set and retrieve properties of the shape of the entity (x, y, z, yaw pitch, roll). Similar to the *Point* class, the *Rectangle* class implements several methods: *CheckDistance, CheckAbsoluteOrientation, CheckRelativeOrientation, and IsFacing*. ProxemicUI calculates the center point of a rectangular entity because its position represents the top-left corner of the rectangle (e.g., tabletop); the calculation is done before performing the test. *Rectangle* also performs shape-to-shape tests, where it checks the received shape and performs the proper test accordingly. Developers can add other shapes, in the same way I discussed previously in *Point*.

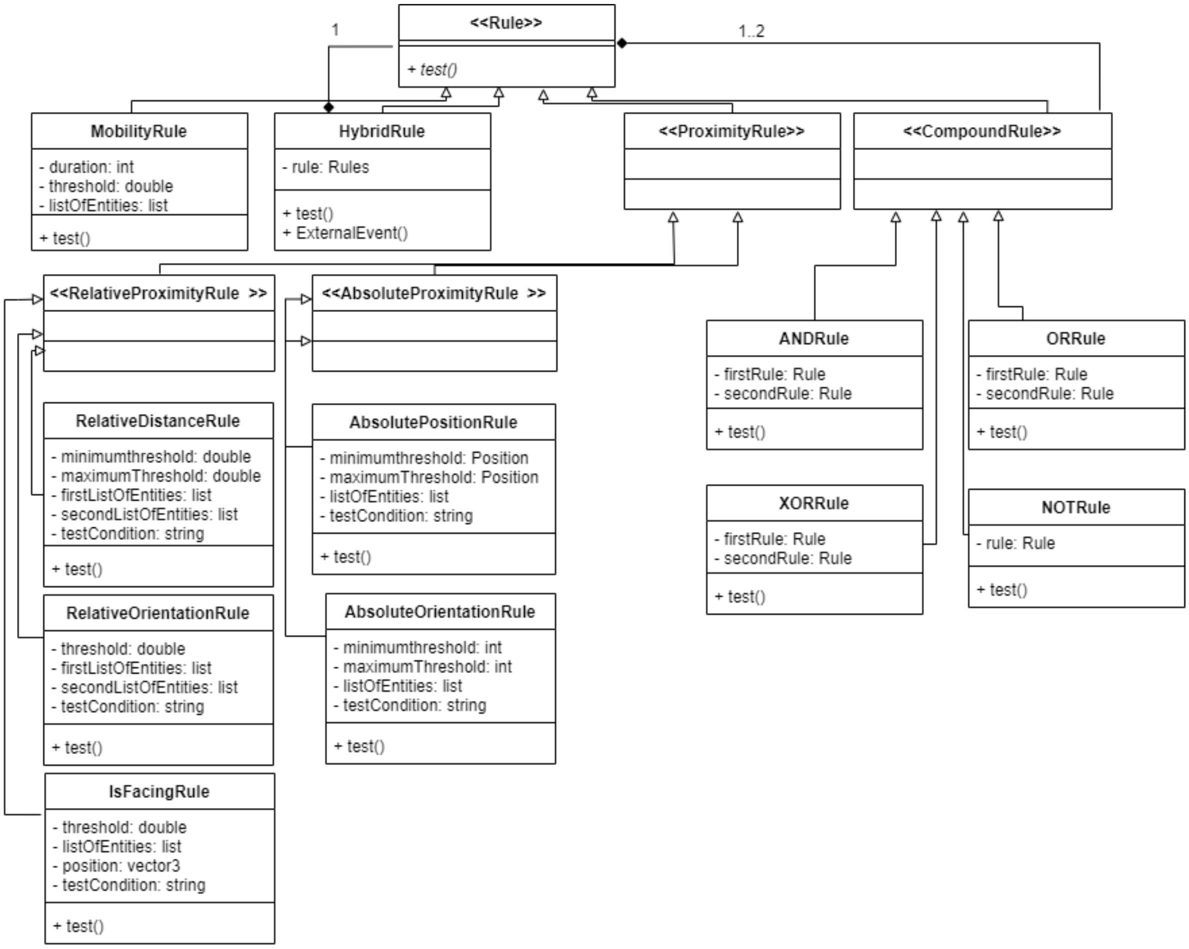


FIGURE 22: CLASS DIAGRAM OF RULE THAT SHOW ALL INHERITED CLASSES AND THEIR RELATIONSHIPS

Rule

Reusability and extensibility are two of the benefits of creating a framework. Reusability is providing interfaces to define “generic components that can be reapplied to create new applications”[35]. Extensibility is allowing developers to expand existing components to add new functionalities to the framework [35][36]. The main goal of the ProxemicUI framework is to provide developers with functionalities that ease the process of developing proxemic-aware applications. The support of these functionalities should allow developers to focus on the final product rather than dealing with low-level programming tasks (e.g., managing the calculation of proximity data). Defining a set of rules where each can handle a specific task and notify developers only when its conditions are met would allow us to achieve this support. Encapsulating these rules into an object-oriented framework allows a reusable set of rules that can be reapplied and expanded according to the application requirements.

One of the behavioral design patterns is the interpreter pattern, which aims to solve a common problem by defining instances of the problem as grammar rules, representing each rule by a class in an object-oriented hierarchy design. With this pattern, the abstract class defines the interpret method; then, all subclasses implement that interpret method to solve the problem instance that the subclass defines [37][51]. ProxemicUI follows this approach as it solves the problem of detecting proximity relationships between entities in the environment. In ProxemicUI, the abstract class *Rule* defines the interpret method *test()*, which is implemented in all subclasses. Each subclass represents an instance of detecting a different type of proximity relationship.

The composite design pattern is one of the structural patterns, which "compose objects into tree structures to represent whole-part hierarchies." [25][37]. Following this pattern, ProxemicUI defines a sub-hierarchy (*CompoundRule*) from the base class (*Rule*) to compose basic objects into a more complex object. The Context Toolkit [29][95] introduced building a new widget (a component that provides attributes and callback methods) based on the data received from multiple widgets. However, the Context Toolkit also requires creating a new widget (a single independent process) for every location that it needs to access its context information, which raises a scalability issue. On the other hand, ProxemicUI can apply a rule to specific types of entities even if they are at different locations in the tracked space, and can support arbitrarily complex compound rules without significantly increasing overhead as all tests operate on a centralized repository of tracked entity data. I chose to follow the composite pattern to allow developers to expand the framework to apply it to different contexts, which is important for prototyping. The *Rule* class inherits all rules, including *HybridRule*, *ProximityRules*, *CompoundRules,* and *MobilityRule*. The base class in the hierarchy is the Rule class and not the ProximityRules class because the other types of rules (HybridRule and CompoundRules) implement the same interpret method, and they are not proximity rules. Figure 22 shows the class diagram of *Rule*, and figure 23 shows the steps of executing each rule.

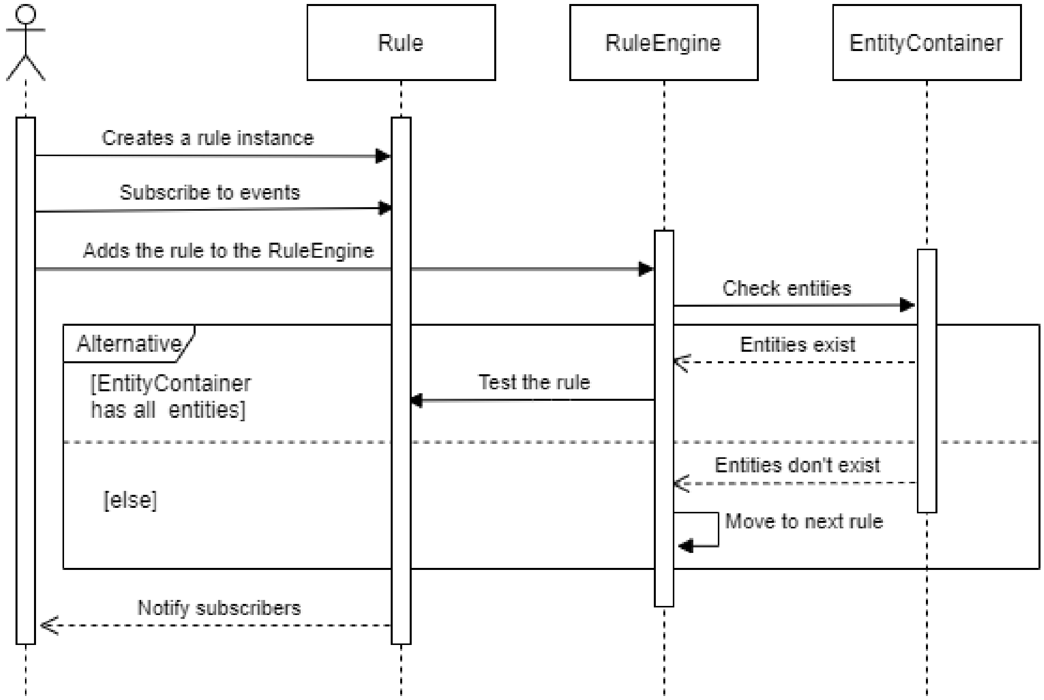


FIGURE 23: THE STEPS OF EXECUTING THE RULES IN PROXEMICUI

ProximityRule

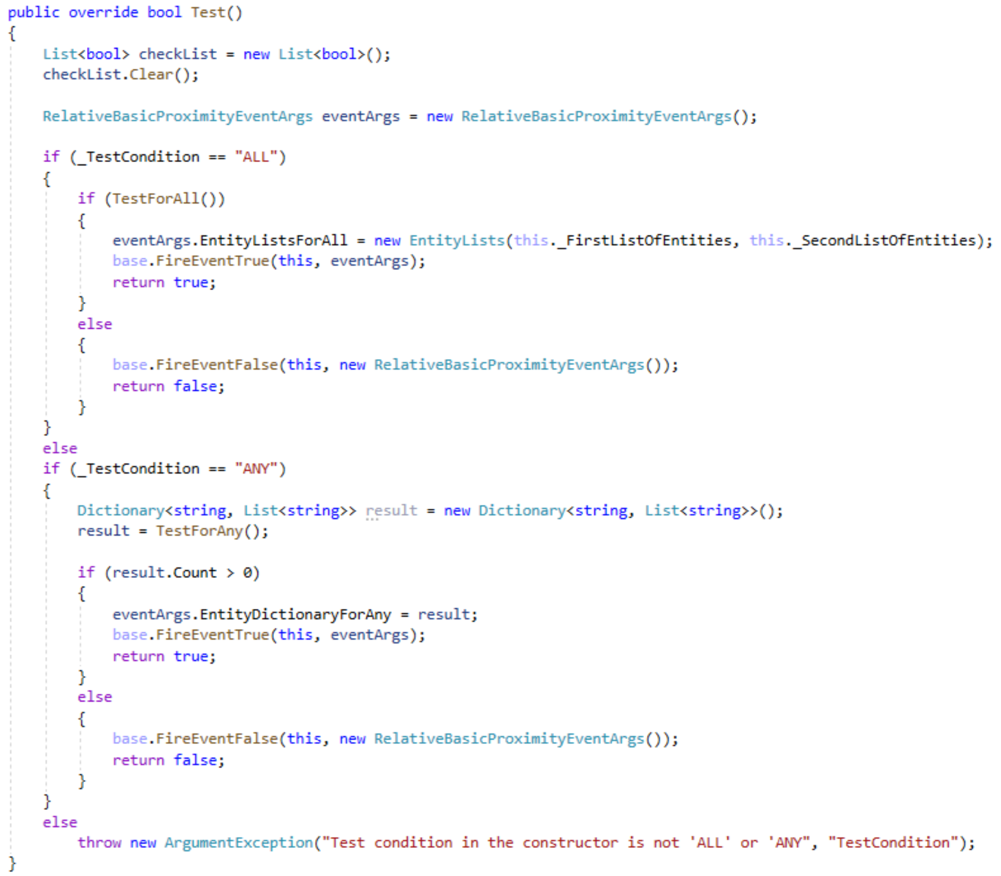
ProximityRule is derived from *Rule*, and it consists of two different subclasses: *RelativeProximityRules*, and *AbsoluteProximityRules*.

RelativeProximityRules

RelativeProximityRules tests a single proximity attribute between entities (e.g., relative distance). This type of rule is similar to the event trigger mechanism in the Proximity Toolkit [71] and is commonly used in existing applications [13][70][125]. *RelativeProximityRules* is the base class for all Relative Proximity Rules, including *RelativeDistanceRule*, *RelativeOrientationRule*, and *IsFacingRule*; other Relative Proximity Rules can be derived from this class as well.

- *RelativeDistanceRule*: is responsible for checking the relative distance between entities and fire *OnEventTrue*, *OnEventFalse*, or *OnEventChanged* according to the test results. It has three constructors to cover the three types of tests (one-to-one, one-to-many, and many-to-many). All constructors consist of five arguments: two doubles representing minimum and maximum thresholds of the distance, lists of entities (two lists, two strings, or one list and one string, depends on the type of test), and one string representing the *TestCondition*. The *TestCondition* can be "ALL" or "ANY," providing developers with two options. First, to fire OnEventTrue when the *TestCondition* is "ALL," the test must be true for all entities. Second, to fire OnEventTrue when the *TestCondition* is "ANY," the test must be true for one or more entities. This means that the "ANY" *TestCondition* can be true when more than one entity passes the test. This meant to support cases where the developers want to know all the entities that pass the test. In addition, *RelativeDistanceRule* overrides the *test* method in *Rule* class that walks through the references for entities and performs the test. As discusses in *Point*, the test method calls the *CheckDistance* method from the shape of one entity, and it passes the shape of the second entity and the two thresholds. *CheckDistance* is of type bool, and the result of each test is stored in *CheckList* of type list in the test method. When all tests are completed, the *test* method will examine the *CheckList*. Then the *test* method will call *FireEventTrue*/*FireEventFalse* from the *Rule* class to fire the event. Figure 24 shows the implementation of the test method.

- *RelativeOrientationRule*: is responsible for checking the relative orientation between entities and fire *OnEventTrue, OnEventFalse,* or *OnEventChanged* according to the test results. Similar to the *RelativeDistanceRule*, it has three constructors to cover three types of tests. The *RelativeOrientationRule* has four arguments: one double representing the threshold in degrees, lists of entities to be tested, and a string for the *TestCondition*. It only has one value to represent the threshold because it checks the difference in degrees from zero to the maximum that the developers specify. *RelativeOrientationRule* follows the same testing



```
public override bool Test()
{
    List<bool> checkList = new List<bool>();
    checkList.Clear();

    RelativeBasicProximityEventArgs eventArgs = new RelativeBasicProximityEventArgs();

    if (_TestCondition == "ALL")
    {
        if (TestForAll())
        {
            eventArgs.EntityListsForAll = new EntityLists(this._FirstListOfEntities, this._SecondListOfEntities);
            base.FireEventTrue(this, eventArgs);
            return true;
        }
        else
        {
            base.FireEventFalse(this, new RelativeBasicProximityEventArgs());
            return false;
        }
    }
    else
    if (_TestCondition == "ANY")
    {
        Dictionary<string, List<string>> result = new Dictionary<string, List<string>>();
        result = TestForAny();

        if (result.Count > 0)
        {
            eventArgs.EntityDictionaryForAny = result;
            base.FireEventTrue(this, eventArgs);
            return true;
        }
        else
        {
            base.FireEventFalse(this, new RelativeBasicProximityEventArgs());
            return false;
        }
    }
    else
        throw new ArgumentException("Test condition in the constructor is not 'ALL' or 'ANY'", "TestCondition");
}
```

FIGURE 24: AN EXAMPLE OF TEST METHOD IMPLEMENTATION IN RELATIVEDISTANCERULE

approach in *RelativeDistanceRule* (discussed above). Instead of calling the *CheckDistance* method, it calls the *CheckOrientation* method from the shape to perform the test. Firing events follow the same approach as *RelativeDistanceRule*.

- *IsFacingRule*: is responsible for checking the relative orientation for one or more entities relative to another entity or a point in space and fire *OnEventTrue, OnEventFalse,* or *OnEventChanged* according to the test results. It has two constructors to cover the two cases: one to one/point in the space or many to one/point in the space. Both constructors have three arguments: double representing the threshold in degrees, string or list for the entities, and vector3 representing the position of an object or a point in the space. *IsFacingRule* follows the same testing approach in *RelativeDistanceRule* (discussed above). Instead of calling the *CheckDistance* method, it calls the *IsFacing* method from the shape to perform the test. Firing events follow the same approach as *RelativeDistanceRule*.

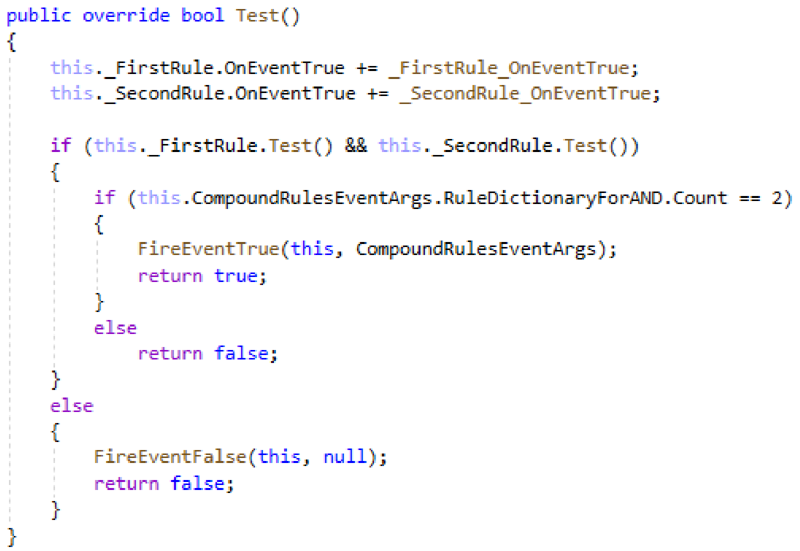
AbsoluteProximityRules

AbsoluteProximityRules tests a single proximity attribute for entities relative to the environment (e.g., check if the user is in a specific room in the house). This type of rule can be seen in some existing applications, such as the Savannah [14] and the AirPlayer [107]. *AbsoluteProximityRules* is the base class for all *AbsoluteProximityRules*, including *AbsolutePositionRule* and *AbsoluteOrientationRule*; other Absolute Proximity Rules can be derived from this class as well. *AbsoluteProximityRules* is responsible for testing if data (e.g., orientation) of an entity is within a threshold according to the whole tracking region. In other words, it provides data similar to DT-DT but in the form of proxemic events.

- *AbsoluteOrientationRule*: is responsible for checking if an entity is facing a certain direction within the tracked region. It allows checking only a single angle (e.g., yaw) or all angles (yaw, pitch, and roll). It also allows checking for one or more entities as well. Therefore, it has four constructors. The two constructors for the single axis test have five arguments: two doubles representing the thresholds, a string or a list for entities, a string for the axis, and a string for the *TestCondition*. The other two constructors all axes test and have four arguments: two lists of double representing threshold (each element represents the threshold for an axis), a string or a list for entities, and a sting for the *TestCondition*. *AbsoluteOrientationRule* follows the same testing approach in *RelativeDistanceRule* (discussed above). Instead of calling the *CheckDistance* method, it calls the *CheckAbsoluteOrientation* method from the shape to perform the test. Firing events follow the same approach as *RelativeDistanceRule*.
- *AbsolutePositionRule*: is responsible for checking if an entity is in a specific location within the tracked region. It has two constructors to test single or multiple entities. These constructors have four arguments: two vector3 representing minimum and maximum thresholds, a string or a list for entities, and a string for the *TestCondition*. *AbsolutePositionRule* follows the same testing approach in *RelativeDistanceRule* (discussed above). Instead of calling the *CheckDistance* method, it calls the *CheckAbsolutePosition* method from the shape to perform the test. Firing events follow the same approach as *RelativeDistanceRule*.

CompoundRules

CompoundRules combines the test of any two rules of type *Rules*: ProximityRule, CompoundRule, HybridRule, and MobilityRule. This type of rule can be seen in some existing applications, such as the Proxemic Media Player [13] and SpiderEyes [30]. *CompoundRules* class is the base class for all Compound Rules, including *ANDRule*, *ORRule*, *XORRule*, and *NOTRule*. Other Compound Rules can be derived from this class as well. When combining two rules, there are three variables that might change: having the same or different entity lists, testing the same or different proximity attributes, and having the same or different thresholds. Currently, it evaluates rules by performing *AND*, *OR*, *XOR*, and *NOT* test for the rules.



```
public override bool Test()
{
    this._FirstRule.OnEventTrue += _FirstRule_OnEventTrue;
    this._SecondRule.OnEventTrue += _SecondRule_OnEventTrue;

    if (this._FirstRule.Test() && this._SecondRule.Test())
    {
        if (this.CompoundRulesEventArgs.RuleDictionaryForAND.Count == 2)
        {
            FireEventTrue(this, CompoundRulesEventArgs);
            return true;
        }
        else
            return false;
    }
    else
    {
        FireEventFalse(this, null);
        return false;
    }
}
```

FIGURE 25: THE IMPLEMENTATION OF TEST METHOD IN ANDRULE

- *ANDRule*: is responsible for testing any two Rule with an AND. Its constructor has two Rule as arguments. It also has a *test* method to test the rule, where both rules must return true for this test to pass. Firing events follow the same approach as *RelativeDistanceRule*; the *test* method will call

FireEventTrue/*FireEventFalse/FireEventChanged* from the *Rule* class to fire the event. Figure 25 shows the implementation of the test method in ANDRule.

- *ORRule*: is responsible for testing any two Rule with an OR. Its constructor has two Rule as arguments. It also has a test method to test the rule, where it will pass if at least one rule returned true. Firing events follow the same approach as *ANDRule*.
- *XORRule*: is responsible for testing any two Rule with XOR. Its constructor has two Rule as arguments. It also has a test method to test the rule, where the test will pass if one and only one rule returned true. Firing events follow the same approach as *ANDRule*.
- *NOTRule*: is responsible for testing a single Rule with NOT. Its constructor has a single Rule as an argument. It also has a test method to test the rule, where the test will pass if the rule returned false. Firing events follow the same approach as *ANDRule*.

HybridRule

HybridRule aims to combine a proximity event with any proxemic external events. The external events can be any type of event that is not based on proximity data. For example, it can be direct interaction with the system (e.g., click, touch, in-air gesture, and voice), receiving system notifications (e.g., timer elapsed), or receiving commands from a different device (e.g., through OSC messages). This type of rule can be seen in existing applications that combine direct interactions with proximity data, such as Eyes-Free Art

[90] and The AirPlayer [107]. As stated in the framework requirements, *HybridRule* can exist in two different ways. First, the system tests the Proximity Rule, and if it's valid, it listens for the external event. With this type of HybridRule, ProxemicUI does not get any notifications about the external event, figure 26 up. Second, the system listens for the external event, and if it is fired, it tests the Proximity Rule. This type of HybridRule requires a mechanism to notify ProxemicUI about the occurrence of external events, figure 26 down. Therefore, ProxemicUI only implements a class for the second type of HybridRule (listening for the external event, then testing the ProximityRule) because the first type can be implemented using a *ProximityRule* (as will be discussed later in this section).

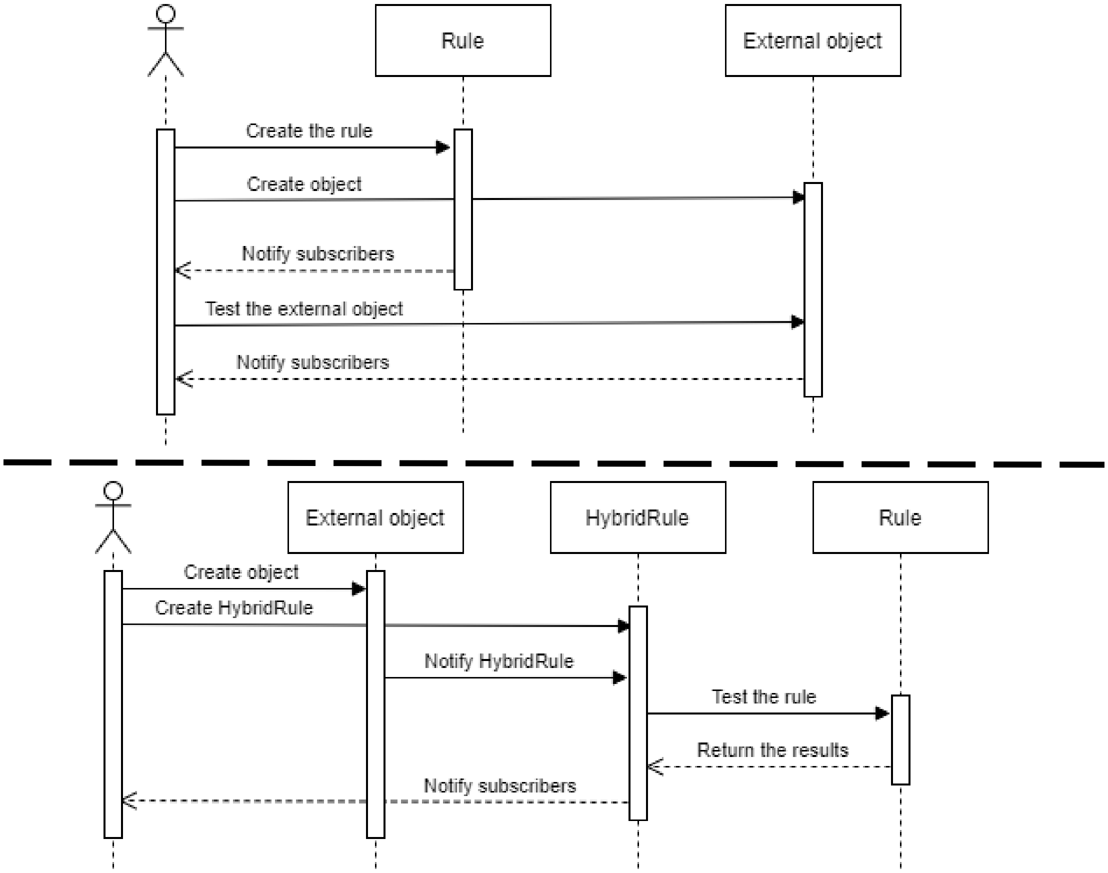
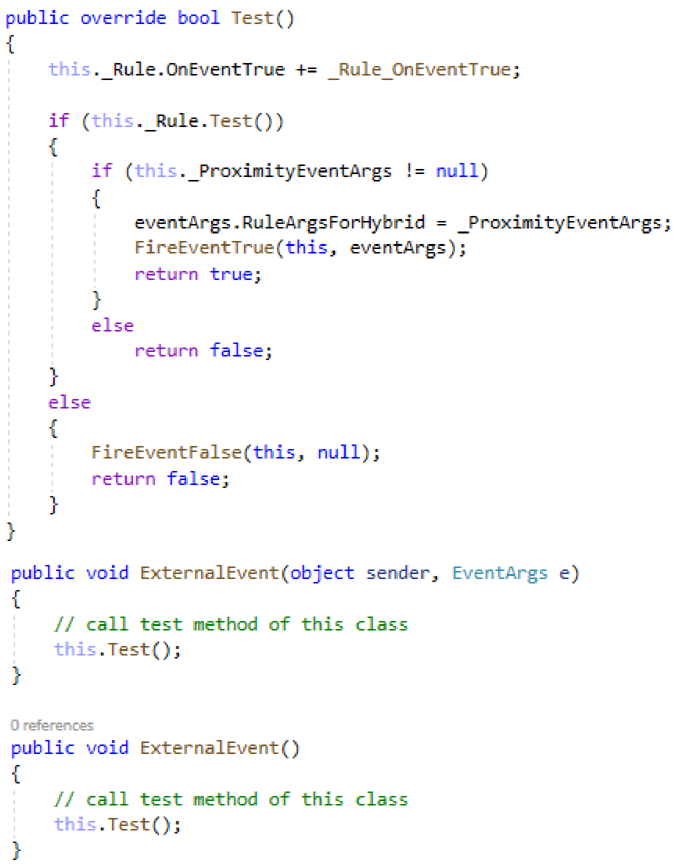


FIGURE 26: UP) THE PROCESS OF HYBRIDRULE THAT DOES NOT COMMUNICATE WITH PROXEMICUI, DOWN) THE PROCESS OF HYBRIDRULE THAT COMMUNICATES WITH PROXEMICUI

HybridRule constructor receives only a *ProximityRule*, which can be of type: *RelativeProximityRules*, *AbsoluteProximityRules*, *CompoundRules*, or *MobilityRule*. It also has two methods: *test* and *ExternalEvent*, and it works as follows. First, application developers call the *ExternalEvent* method when the external event is performed (e.g., a touchdown event), so it will be the handler for the external event (e.g., touch event, timer elapsed). Then, the *external event* will call the test method, which will start testing the *ProximityRule*. Finally, *OnEventTrue*/*OnEventFalse* is fired if the *ProximityRule* is valid. The *ExternalEvent* method is overloaded in the *HybridRule*. The first method has two arguments following the same event model in C#. The first argument is the sender object that created the event. The second argument is of type EventArgs, which is the base class for all classes that contain event data in C# [33]. This method allows developers to subscribe to it when the external event is of any type of the classes representing event data (e.g., touch down or time elapsed). If developers want to support an event that does not have the same types of arguments, they have to overload this method with the arguments they need to support. The second method in the HybridRule has no arguments, so developers can call it directly as the external event occurs. I used this method to notify the HybridRule when an OSC message is received. Figure 27 shows the implementation of *test* and *ExternalEvent* methods in *HybridRule*.



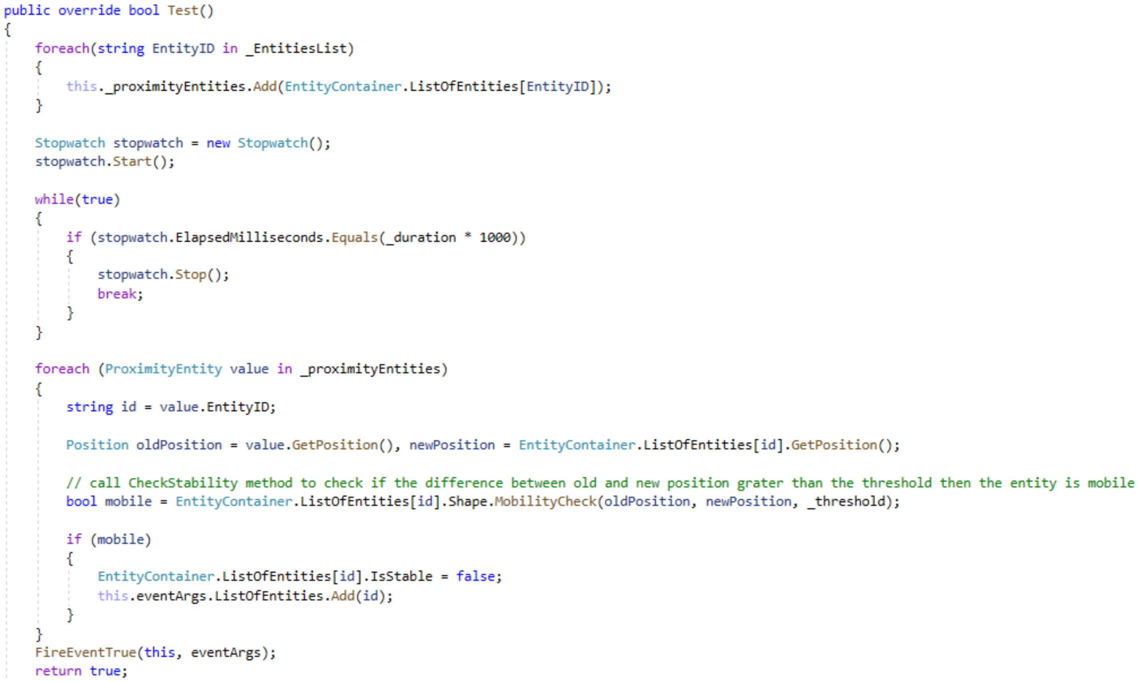
```
public override bool Test()
{
    this._Rule.OnEventTrue += _Rule_OnEventTrue;

    if (this._Rule.Test())
    {
        if (this._ProximityEventArgs != null)
        {
            eventArgs.RuleArgsForHybrid = _ProximityEventArgs;
            FireEventTrue(this, eventArgs);
            return true;
        }
        else
            return false;
    }
    else
    {
        FireEventFalse(this, null);
        return false;
    }
}

public void ExternalEvent(object sender, EventArgs e)
{
    // call test method of this class
    this.Test();
}

0 references
public void ExternalEvent()
{
    // call test method of this class
    this.Test();
}
```

FIGURE 27: THE IMPLEMENTATION OF TEST AND EXTERNALEVENT METHODS IN HYBRIDRULE

ProxemicUI doesn't implement any classes for the first type of HybridRule (testing the Proximity Rule then listening for the external event) because it is a sequence of events, which can be implemented using any ProximityRule. For example, ProximityTable shows a button when two users get close to each other; if the button is touched, it increases the size of the workspace. We can implement this using *RelativeDistanceRule* as follows. First, I create a *RelativeProximityRule* that tests the distance between entities. Second, the handling method of this rule shows the button on display. Finally, the handling method of the button will increase the size of the workspace. We cannot implement this sequence using the current *HybridRule* implementation because *HybridRule* listens for an external event (e.g., UI event) then starts testing the *ProximityRule*.

MobilityRule

MobilityRule checks and updates the mobility status (*IsStable*) of entities. This can be useful when the developer wants to draw a system response based on an entity's location and mobility status. For example, in Alices' smart home scenario, placing her tablet on the kitchen counter for 2 minutes might be an indication that she is using the tablet to read recipes. Therefore, the system might keep the screen on. The mobility status can be updated in two different ways. First, the developer updates *IsStable* field manually. Second, the system automatically updates it through the use of *MobilityRule*. The developer controls this automated process where s/he creates a *MobilityRule* and passes it arguments. The *MobilityRule* has two arguments: a threshold in centimeters between old and new position (to determine its movements) and duration in seconds between capturing the old and new position. Then, it tests the mobility and updated the *IsStable* field of the entity.

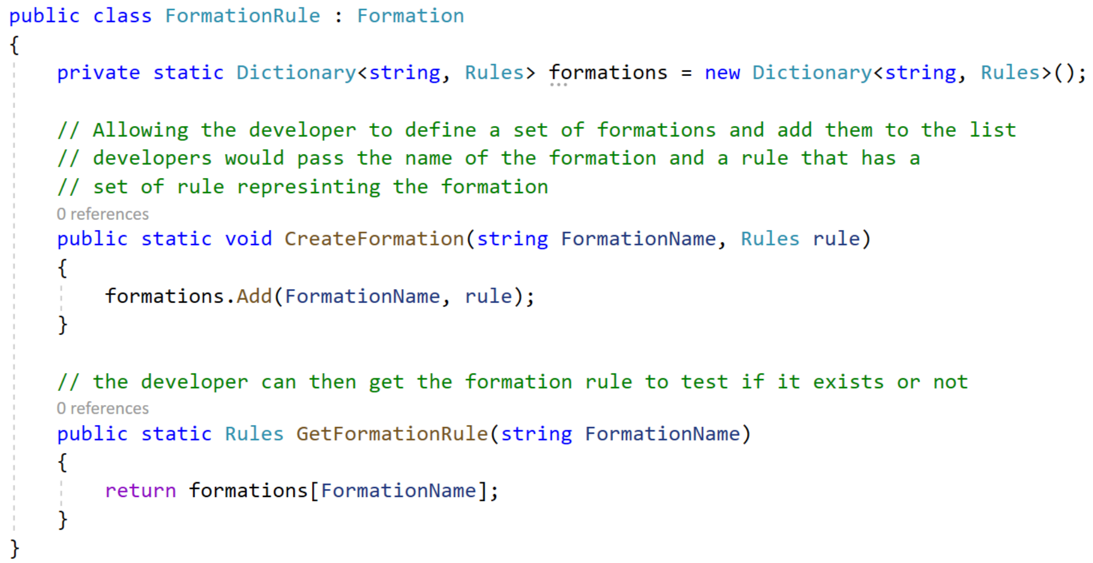


```
public override bool Test()
{
    foreach(string EntityID in _EntitiesList)
    {
        this._proximityEntities.Add(EntityContainer.ListOfEntities[EntityID]);
    }

    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();

    while(true)
    {
        if (stopwatch.ElapsedMilliseconds.Equals(_duration * 1000))
        {
            stopwatch.Stop();
            break;
        }
    }

    foreach (ProximityEntity value in _proximityEntities)
    {
        string id = value.EntityID;

        Position oldPosition = value.GetPosition(), newPosition = EntityContainer.ListOfEntities[id].GetPosition();

        // call CheckStability method to check if the difference between old and new position grater than the threshold then the entity is mobile
        bool mobile = EntityContainer.ListOfEntities[id].Shape.MobilityCheck(oldPosition, newPosition, _threshold);

        if (mobile)
        {
            EntityContainer.ListOfEntities[id].IsStable = false;
            this.eventArgs.ListOfEntities.Add(id);
        }
    }
    FireEventTrue(this, eventArgs);
    return true;
```

FIGURE 28: THE IMPLEMENTATION OF TEST METHOD IN MOBILITYRULE

Formation

F-formations can play an important role in users' experience with interactive systems. For example, detecting F-formations can allow the systems to adjust contents on display according to the formations [11], notify developers about openings on the tabletop [73], or support interactions in medical settings [76]. While developers can use the rules discussed above to detect F-formations, ProxemicUI provides a class hierarchy that allows developers to specify F-formations. The base class of this hierarchy is the *Formation* class tat has two subclasses: *FormationRule* and *FormationPlacement*.



```
public class FormationRule : Formation
{
    private static Dictionary<string, Rules> formations = new Dictionary<string, Rules>();

    // Allowing the developer to define a set of formations and add them to the list
    // developers would pass the name of the formation and a rule that has a
    // set of rule repreasinting the formation
    0 references
    public static void CreateFormation(string FormationName, Rules rule)
    {
        formations.Add(FormationName, rule);
    }

    // the developer can then get the formation rule to test if it exists or not
    0 references
    public static Rules GetFormationRule(string FormationName)
    {
        return formations[FormationName];
    }
}
```

FIGURE 29: IMPLEMENTATION OF THE FORMATIONRULE CLASS SHOWING THE TWO METHODS TO CREATE/ RETRIEVE THE FORMATION RULE

FormationRule

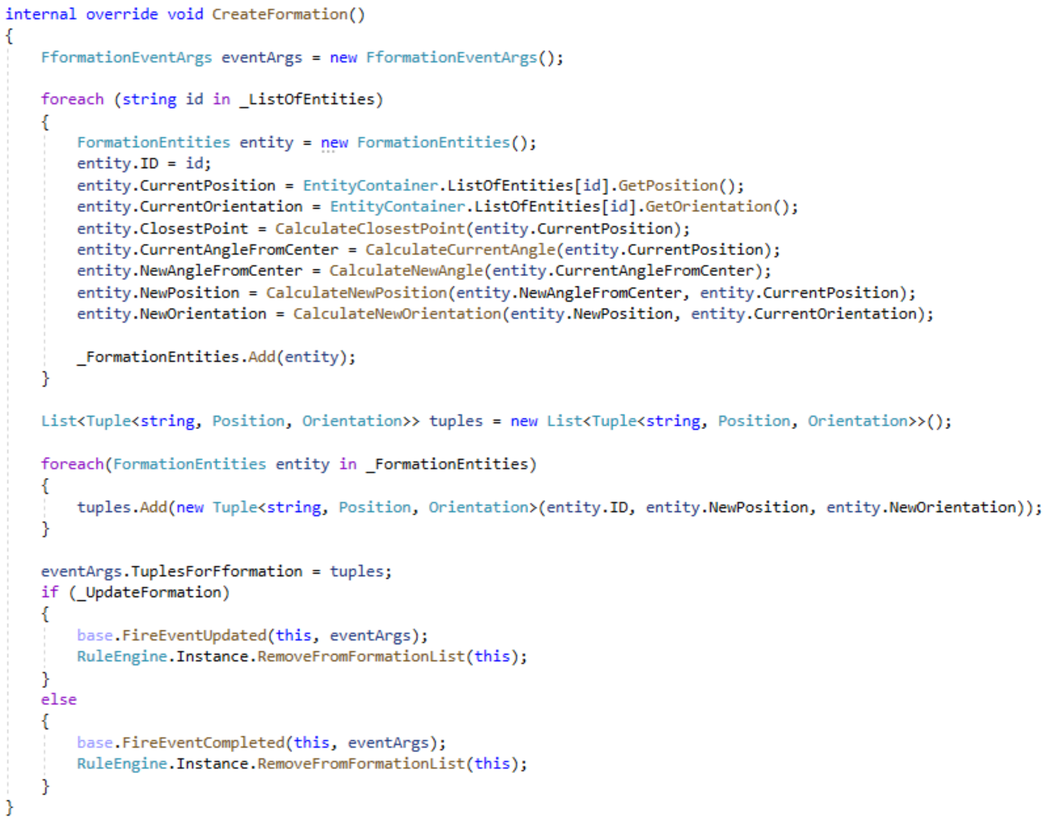
FormationRule allows developers to use the rules we discussed above to define a formation rule (specify F-formations). Then, developers can retrieve a formation rule to check if it exists anywhere in the environment. The *FormationRule* has two methods: one allows developers to create the formation, and the second one to retrieve it. Figure 29 shows the implementation of the FormationRule class.

FormationPlacement

In virtual environments, some scenarios require defining the exact placement of entities when creating an F-formation. For example, in Story CreatAR, there is a need to create different formations for objects in the environment (e.g., creating a conversation where avatars form a circle). To support such requirements, ProxemicUI implements the *FormationPlacement* class, which is the base class of all formation placement rules. Currently, it only has a single sub-class: *CircularFormation* class. Other placement rules can be derived from the *FormationPlacement* base class. The *FormationPlacement* class has two events (*OnFormationCompleted* and *OnFormationUpdated*) and two methods to fire these events (*FireEventCompleted* and *FireEventUpdated*).

CircularFormation

CircularFormation is responsible for calculating and return new positions and orientations for entities participating in the circular formation. It calculates the closest position on the circle for each entity relative to its current position. The constructor of *CircularFormation* has three arguments: vector3 representing the center of the circle, double representing the radius of the circle, and a list containing the participating entities. It also allows updating existing formation by calling the *UpdateFormation*, which takes a double as the new radius (adding new entities increases the size of the circle) and a list for the additional entities. *CircularFormation* also has several methods to calculate different attributes to get to the final results. When calculating the final position and orientation is completed, the *CircularFormation* calls a method (*FireEventCompleted* or *FireEventUpdated*) to fire an event (*OnFormationCompleted* or *OnFormationUpdated*). The event object will contain a list of tuples where each tuple includes a string for the entity’s ID, the entity's position, and the entity's orientation. These data will be used to place entities in the circular formation.

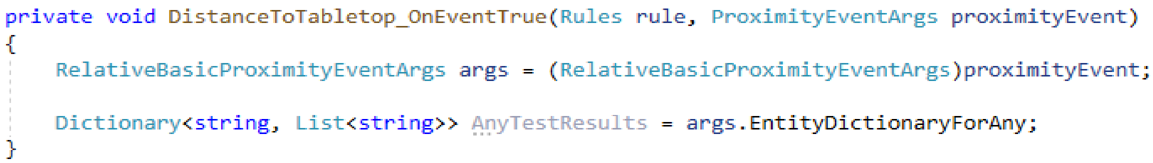


```
internal override void CreateFormation()
{
    FformationEventArgs eventArgs = new FformationEventArgs();

    foreach (string id in _ListOfEntities)
    {
        FormationEntities entity = new FormationEntities();
        entity.ID = id;
        entity.CurrentPosition = EntityContainer.ListOfEntities[id].GetPosition();
        entity.CurrentOrientation = EntityContainer.ListOfEntities[id].GetOrientation();
        entity.ClosestPoint = CalculateClosestPoint(entity.CurrentPosition);
        entity.CurrentAngleFromCenter = CalculateCurrentAngle(entity.CurrentPosition);
        entity.NewAngleFromCenter = CalculateNewAngle(entity.CurrentAngleFromCenter);
        entity.NewPosition = CalculateNewPosition(entity.NewAngleFromCenter, entity.CurrentPosition);
        entity.NewOrientation = CalculateNewOrientation(entity.NewPosition, entity.CurrentOrientation);

        _FormationEntities.Add(entity);
    }

    List<Tuple<string, Position, Orientation>> tuples = new List<Tuple<string, Position, Orientation>>();

    foreach(FormationEntities entity in _FormationEntities)
    {
        tuples.Add(new Tuple<string, Position, Orientation>(entity.ID, entity.NewPosition, entity.NewOrientation));
    }

    eventArgs.TuplesForFformation = tuples;
    if (_UpdateFormation)
    {
        base.FireEventUpdated(this, eventArgs);
        RuleEngine.Instance.RemoveFromFormationList(this);
    }
    else
    {
        base.FireEventCompleted(this, eventArgs);
        RuleEngine.Instance.RemoveFromFormationList(this);
    }
}
```

FIGURE 30: THE IMPLEMENTATION OF CREATEFORMATION METHOD IN CIRCULARFORMATION

RuleEngine

RuleEngine is responsible for continuously checking the developer’s rules, which is done as follows. *RuleEngine* has a list (*listOfRules*) that contains all rules to be tested. There are three tasks that *RuleEngine* can perform using this list. First, using the *AddToRuleList* method, which has a single type of rule argument, the developer can add a rule to the list to be tested. Second, if a developer no longer wants to test a rule, s/he can use the *RemoveFromRuleList* method to remove the rule from the list. *RemoveFromRuleList* has a single argument of type *Rule*. The *RuleEngine* overload *TestRunner* method as follows. The first method is a private method that runs on a separate task to continuously tests all rules in *listOfRules*. The second method is a public method that allows developers to perform the tests at the application layer. This can be useful if a developer does not want to run the test continuously. It also avoids running the test in a separate thread, which I used to overcome the multithreading issue with Unity that I will discuss in chapter 8.



```
private void DistanceToTabletop_OnEventTrue(Rules rule, ProximityEventArgs proximityEvent)
{
    RelativeBasicProximityEventArgs args = (RelativeBasicProximityEventArgs)proximityEvent;

    Dictionary<string, List<string>> AnyTestResults = args.EntityDictionaryForAny;
}
```

FIGURE 31: USING THE PROXIMITYEVENTARGS TO ACCESS THE TEST RESULTS

ProximityEventArgs

ProximityEventArgs class is the base class for subclasses that pass different arguments to the handling methods when events are fired. Because each rule has different arguments, including different *TestCondition* (ALL/ANY), it is required to define different argument types to pass the results to the handling methods. Any additional arguments can be added as a subclass of the *ProximityEventArgs* class. Figure 31 shows an example of using these subclasses. In the first line inside the method, the developer casts the *ProximityEventArgs* object to a specific sub-object (in this example, the sub-object is *RelativeBasicProximityEventArgs*). In the second line, the developer can access a specific data collection type and read its data (in this example, the data collection type is a dictionary). These subclasses are as follows.

RelativeBasicProximityEventArgs

This object will be used for all basic relative proximity rules and includes multiple types for rules and test conditions:

- *EntityListsForAll*: when the test condition for the basic rule is "ALL" this contains the two lists of entities that were passed to the rule.
- *EntityDictionaryForAny*: when the test condition for the basic rule is "ANY" this contains a dictionary where the keys are IDs from the first list, and the value is a list with all IDs of entities that passed the test relative to the key.
- *EntityListForIsFacing*: returns a single list that contains all the entities that are facing the target object.

AbsoluteBasicProximityEventArgs

This object will be used for all basic absolute proximity rules. Currently, it includes only a single type:

- *ListOfEntities*: returns a list of entities that passed the test relative to the environment.

CompoundRulesEventArgs

This object will be used for all basic relative proximity rules and includes multiple types for rules and test conditions:

- *RuleDictionaryForAND*: returns a dictionary where the key is one of the subrules and the value is the results arguments from testing that rule.
- *RuleTupleForORandXOR*: returns a tuple when testing for OR and XOR. This tuple contains the Rule that passes the test and its argument.
- *RuleForNOT*: return a single if it passes the test.

HybridEventArgs

This object will be used for the HybridRule. Currently, it includes only a single type:

- *RuleArgsForHybrid*: returns the proximity event arguments of the proximity rule that was passed to the Hybrid rule.

FformationEventArgs

This object will be used for the FormationPlacement. Currently, it includes only a single type:

- *TuplesForFformation*: returns a list of tuples where the entity's ID and its new position and orientation.

Extension points

Framework extensibility is allowing developers to expand existing components to add new functionalities to the framework [35][36]. This section will discuss the extensibility of ProxemicUI by defining the extension points that developers can use to add more functionality to ProxemicUI to apply it to different contexts.

1. Rule hierarchy: one of the extension points in ProxemicUI is the Rule hierarchy, where a developer can subclass a new rule object in the Rule hierarchy to perform a new task. When a new rule object is added to the hierarchy, developers need to define new arguments in the *ProximityEventArgs* to pass the results of testing the rule in the event object. Developers will follow the same process to a new *CompoundRule* object, where they can use additional logical operators to compose more complex objects to support different contexts. This composition would be extended from the *CompoundRule* class*,* similar to existing rules (e.g., *ANDRule*). A clear example of rule extension is when I implemented the *MobilityRule* and *IsFacingRule*, which are two sub-classes, and each performs a new task.

2. Shape: as stated before, to accurately measure the relative proximity data, ProxemicUI performs a shape-to-shape comparison (e.g., point-to-rectangle). Currently, ProxemicUI implements the test for two shapes: point and rectangle. Developers can subclass a new shape object in the Geometry hierarchy to add additional shapes. When a new shape object is added to the hierarchy, developers only need to implement the Geometry (the subclass) methods, and there are no other changes required to the framework.

3. Entity: Entity is the base of any proxemic-aware system, and it is the third extension point in ProxemicUI. I discussed how to extend rules and shapes and how each extension point would add new functionality to the framework, but why

would we extend an object that holds the properties and how that would be useful? There might be some scenarios where there is a need to subclass the entity to keep track of different interactions. For example, subclassing entities can be beneficial if we have a system that detects in-air gestures. In such cases, the main entity is the user, and the child entities are his/her joints. With this classification, we can detect the relative proximity data within one main entity (e.g., user). To add a child entity, developers need to subclass the Entity class (the base) and add the specific properties for that subclass to it. This process does not require any changes to the rest of the framework.

4. DataReceiver: *DataReceiver* is the communication gateway between ProxemicUI and other tracking systems, which is an important extension point in ProxemicUI. *DataReceiver* implements the OSC messaging server to receive tracking data from external systems, which can be expanded by subclassing the *DataReceiver* in two ways. First, subclassing the *DataReceiver* to define additional OSC messages format that might include different data about the environment. This subclassing includes creating a method that would process the new message to extract the tracking data. Second, subclassing the *DataReceiver* to add a sub-receiver to capture the data from different SDKs (e.g., Microsoft Kinect SDK). This subclassing includes setting up the connection with the tracking system and writing the method that extracts the tracking data. The *DataReceiver* only receives and extracts the proximity data, then passes these data to the *EntityContainer*, which will create new entities and assign their data. Therefore, for both approaches,

developers need to write the method that will process the data in the *EntityContainer* as well.

Summary

This chapter discussed the full architecture of ProxemicUI v2, including a set of improvements derived from the evaluation methods. This chapter also explains how to use ProxemicUI with a walkthrough example that shows all the required steps to use every rule in ProxemicUI. The next chapter (chapter 6) discusses using ProxemicUI in a smart home setup, resulting in a set of additional design requirements that improved ProxemicUI.

Chapter 6: Evaluating ProxemicUI Through Proof-of-Concept Applications

One of the most critical aspects of developing a framework is to evaluate it to validate its concepts. One of the evaluation methods for toolkits in HCI is the *demonstration* method, which shows the toolkit's features and how developers can use them. To evaluate ProxemicUI, I implemented a set of seven proof-of-concept applications representing smart home appliances and integrated them into a smart home setup. In so doing, I apply one of the techniques for HCI toolkit evaluation through *demonstration* identified by Ledo et al. [65] called "*replicated examples*", which show "how the toolkit supports and encapsulates prior ideas into a broader solution space"[65]. Marquardt et al. [71] and Pérez et al. [84][85] employ the use of testing relative proximity attributes (distance and orientation). ProxemicUI uses these data as the base of more complex tests: testing multiple attributes/thresholds for multiple entities simultaneously using Compound Rules and integrating proximity events with external non-proxemic events using Hybrid Rule. In this evaluation, the proof-of-concept applications are similar to Proximity-Aware control [64], which was implemented using the Proximity toolkit. The Proximity-Aware control is a system that grants users control of a specific device based on the spatial relationships between users and devices in a smart environment. This is intended to show how ProxemicUI implements similar examples with less effort (less lines of codes and less tests to keep track of). This chapter starts by discussing these proof-of-concept applications. It will then discuss additional design requirements and several refinements to ProxemicUI that I derived from this integration. Finally, it will discuss the proposed hackathon user study that was canceled due to the restrictions of COVID-19.

Proof-of-concept applications

To evaluate the features that ProxemicUI provides and find out how well it can support the development of interactive applications in smart environments, I developed seven proof of concept applications that simulate smart home appliances. To help contextualize the proof-of-concept applications, I am going to start with the following scenario:

> *Alice has several home appliances, including a toaster, alarm, thermostat, TV, coffee machine, and tabletop. She wants to gain control of all of these appliances through her phone using her proximity data. For example, she wants to open the coffee machine interface on her phone when she gets closer to the coffee machine. After she starts the coffee machine, she wants the system to track her to show her a notification when the coffee is ready based on her location. For example, if she goes to watch TV, the system will show the notification on the TV; but if she goes to work on her tabletop, the notification will be showing there. She also wants to turn on the TV and open its interface on her phone when she is within distance and facing the TV simultaneously. If she is within distance to the TV (e.g., sitting on the couch) but reading a magazine, the TV will not turn on.*

In this scenario, we can see testing multiple relative proximity attributes (position and orientation). For example, we test for the relative distance between Alice and every appliance (e.g., the coffee machine) in her house to give her control of that device. We also test both relative distance and orientation simultaneously to turn on the TV and open its interface on Alice's phone. We can also see combining relative proximity data with external events. The external event data is when Alice interacts with her phone to start the coffee machine (in this case, the external event is the UI event triggered by the phone interactions), and the proximity data is when the system tracks Alice's proximity to other devices (in this case, to the TV or tabletop) to determine where to show notifications.

Following the contextual scenario discussed above, the idea of using these proof-of-concept applications is to replace the button functionalities of the main interface with proximity events using ProxemicUI. To do so, I employed the VIVE trackers to access the proximity data of each appliance. Since all appliances are stale, I record the proximity data of each appliance using a VIVE tracker and only attached the trackers to the tables that would be used as controllers. In the following subsections, I will discuss these proof-of-concept applications.

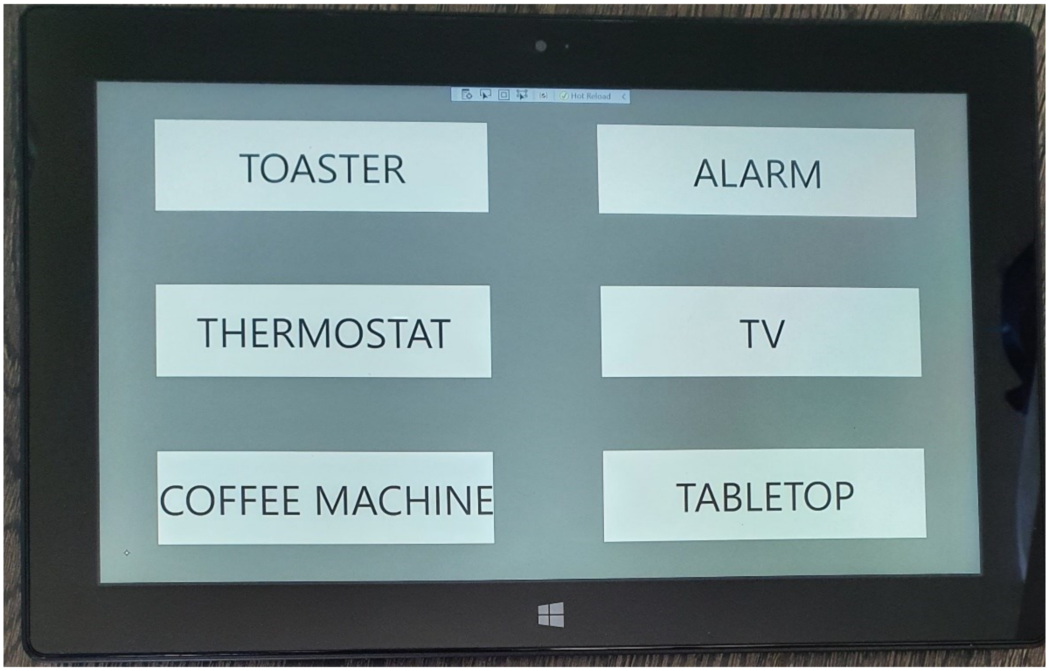


FIGURE 32: SNAPSHOT OF MAIN INTERFACE OF TABLET CONTROLLER APPLICATION

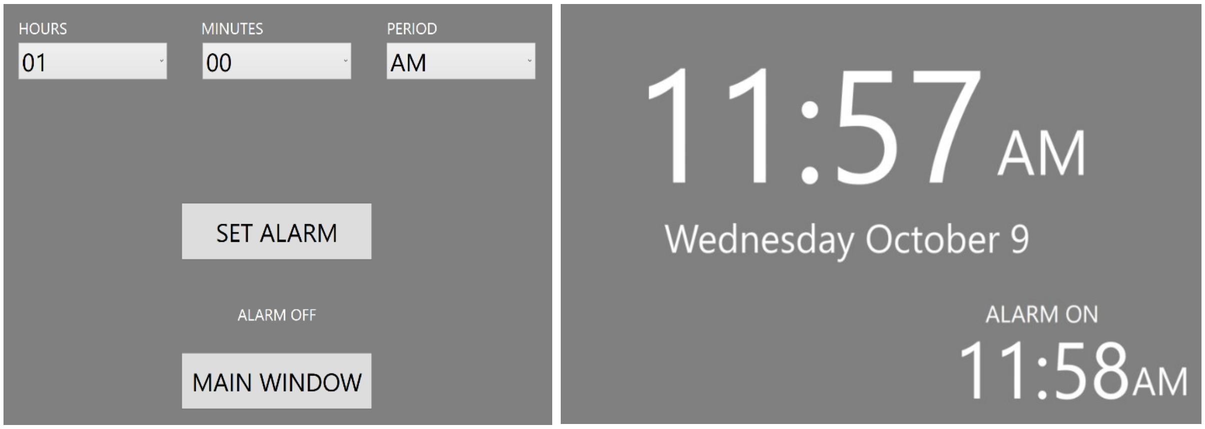
The main application

The main application is the tablet/smartphone application that contains the control interface for all other applications(appliances). The main interface consists of six buttons;

each represents a different application and will show a sub-interface for the selected device. Because I was planning to use these applications for the hackathon user study, I implemented this interface with buttons to have a fully functioning interface. Then, developers are required to replace the touch event with a proxemic event (e.g., relative distance event) using ProxemicUI. Next, I will explain how each application works and communicates with the control interface.

The alarm interface

The alarm interface has a basic setting where users can choose the hour, minutes, and period (am/pm). The alarm interface shows the current time/date, the alarm's status (on/off), and the time of the alarm if it is on.



HOURS 01
MINUTES 00
PERIOD AM

FIGURE 33: LEFT: ALARM INTERFACE IN THE TABLET APPLICATION, RIGHT: ALARM INTERFACE SIMULATION

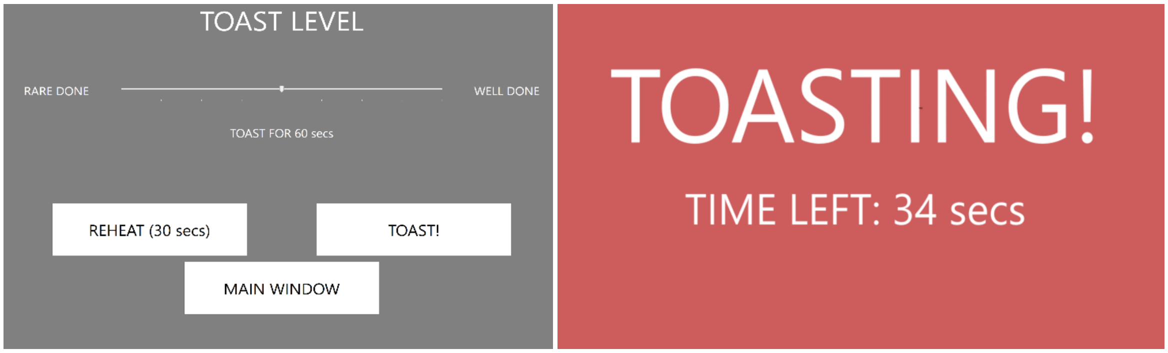


FIGURE 34: LEFT: TOASTER INTERFACE IN THE TABLET APPLICATION, RIGHT: TOASTER INTERFACE SIMULATION

The toaster interface

The toaster interface sets the timer for the toaster in seconds, which has a scale from rare done to well done. It has two different options to start toasting based on the timer setting or reheat the toast for 30 seconds. The toaster application will show the setting (toasting or reheating) and the remaining time to be done.

The thermostat interface

The thermostat interface allows to choose the mode (cool/heat) and set the temperature. The thermostat application shows the current temperature, current setting (cool/heat + temperature), current date/time, and changes in the screen's background based on the setting.

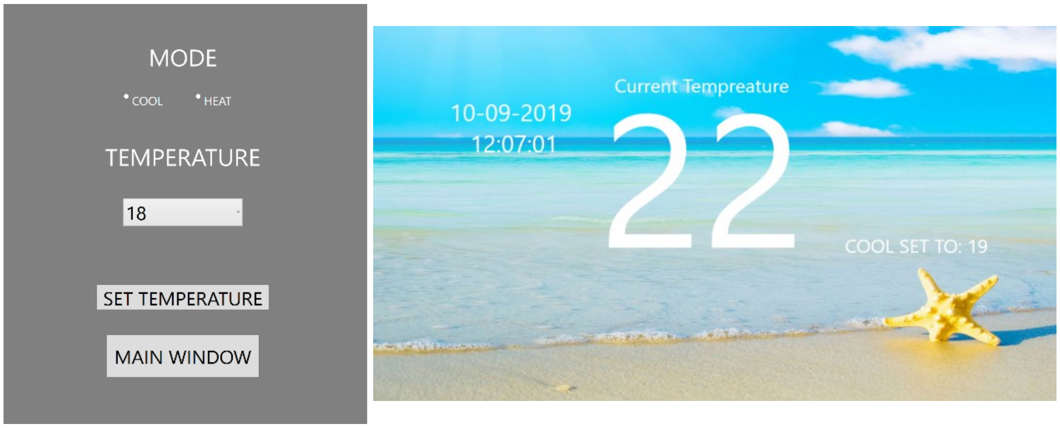


FIGURE 35: LEFT: THERMOSTAT INTERFACE IN THE TABLET APPLICATION, RIGHT: THERMOSTAT INTERFACE SIMULATION

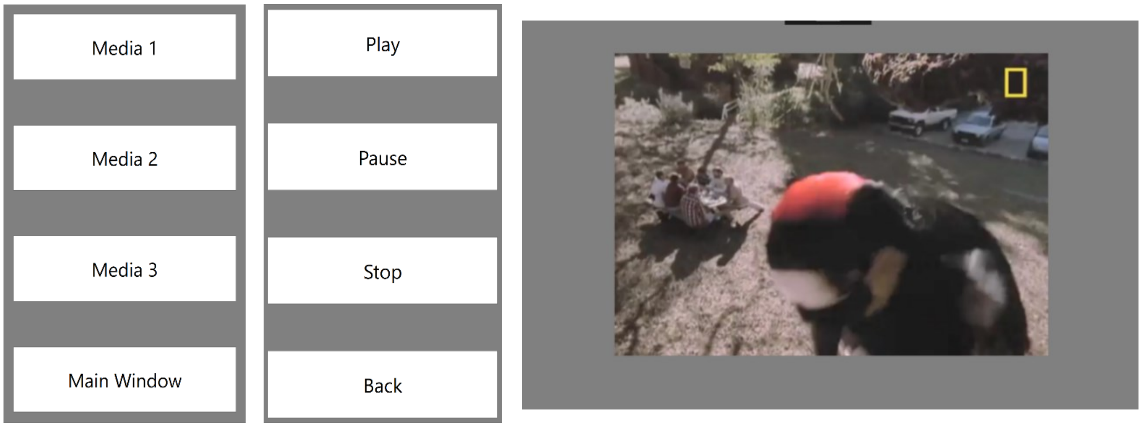


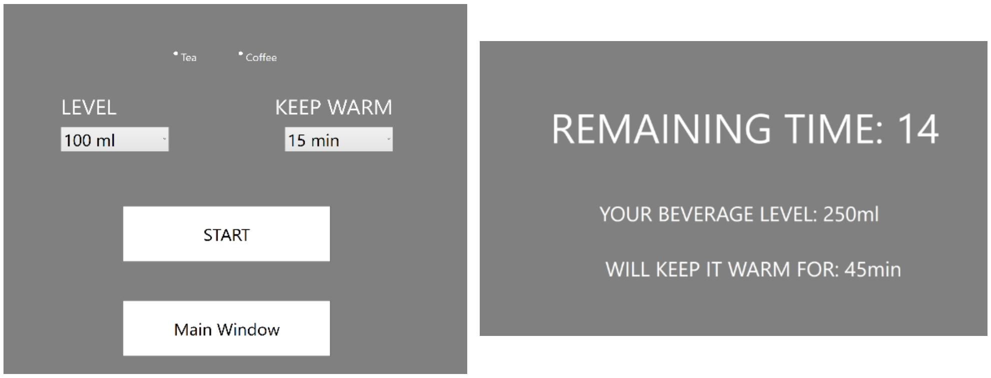
FIGURE 36: LEFT: TWO TV INTERFACES IN THE TABLET APPLICATION, RIGHT: TV INTERFACE

The TV interface

The main TV interface allows the user to choose the media to be played, while the sub-interface allows for controlling current media (e.g., play/pause). The TV application will play the selected media. It can also show a notification to the user on the TV about other events in the smart environment.

The coffee machine interface

The coffee machine interface allows the user to choose the type of beverage (tea/coffee) and its amount in ml. It can also set the duration to keep the beverage warm. The coffee machine application shows the remaining time for the beverage to be ready, the beverage level, and the duration to be kept warm.



LEVEL 100 ml

KEEP WARM 15 min

REMAINING TIME: 14

YOUR BEVERAGE LEVEL: 250ml

WILL KEEP IT WARM FOR: 45min

FIGURE 37: LEFT: COFFEE MACHINE INTERFACE IN THE TABLET APPLICATION, RIGHT: COFFEE MACHINE INTERFACE SIMULATION

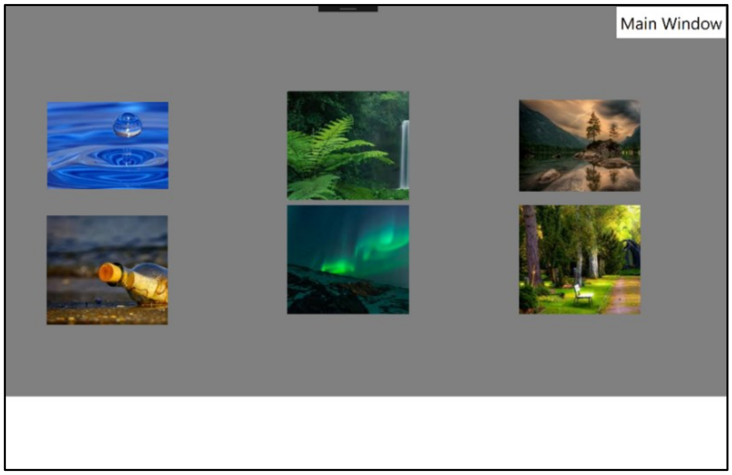


FIGURE 38: TABLETOP INTERFACE IN BOTH TABLET APPLICATION AND TABLETOP

The tabletop interface

Both the tabletop interface in the tablet application and the tabletop application shows several images and a white bar at the bottom of the screen. When the user drags an image to the white bar, it will be transferred to the other application.

All commands from the tablet application are transferred to other applications using OSC messages. Some applications will also respond to these messages or send different messages to other applications (e.g., show notification on TV that the coffee is ready) using OSC messages.

Additional design requirements

As stated earlier, employing ProxemicUI to support a smart home setup using the proof-of-concept applications contributes as an initial evaluation of the framework. This section will discuss what new design requirements I derived from this employment.

A More Generic HybridRule

In v1, Hybrid Rules referred to combining a Proximity Rule (RPR or CPR) and a UI event.2 However, during testing ProxemicUI with the proof of concept applications, I had system notifications instead of UI events while using the HybridRule. For example, when Alice starts the coffee machine and goes to watch TV, the coffee machine will send a notification to the TV telling her that the coffee is ready. In addition, the coffee machine sent the notification to the TV using an OSC message, which is different than subscribing to a button click. Therefore, to make the HybridRule more generic, we require the following two changes. First, changing the name of UIEvent to ExternalEvent to include a broader set of events. Second, add a second ExternalEvent method in the HybridRule, so developers can call it directly instead of subscribing to events. For this second change, ProxemicUI has a method that is called as a handling method when subscribing to any event where it takes arguments following the same approach that Windows event model. When the system receives notifications from a different process/device, we need to add a way to notify the HybridRule that an event has occurred (in this case, I added a second method with no arguments to be called directly).

Extensions to the OSC message format

In v1, ProxemicUI's OSC message format was determined based on the data used from the DT-DT top-down tracker in the ProximityTable application: ID, X, and Y for each entity. However, when employed ProxemicUI v1 to support the proof-of-concept applications, I used the HTC VIVE Lighthouse tracking system to make ProxemicUI's tracker support more robust and generic (e.g., through defining a broader format for the default OSC messages). The HTC VIVE Lighthouse tracking system provides timestamp, ID, X, Y, Z, Yaw, Pitch, and Roll for each tracker. Because ProxemicUI v1 defines a shape for every entity to accurately measure proximity relationships, it needs to know the dimension of each shape. In addition, some properties of the smart environment might only need to be sent once to ProxemicUI and not supported by tracking systems (e.g., the position of stable entities such as a wall display and a thermostat). Since such data might not be provided by tracking systems, developers might send them in separate messages. I added an additional format to other OSC messages that include the shape properties, which can be expanded for other data.

Activation status

When DT-DT can no longer track a person (e.g., when they move outside the tracked region), it will delete all data for that person from the system. So, if this person came back, s/he would be considered a new user. For this reason, ProxemicUI v1 used to delete an entity if it is no longer being tracked. However, this approach is not suitable for a proxemic-aware system, especially if we can detect entities' identities. For example, when a VIVE tracker is not moving for about five minutes (e.g., attached to a tablet that was left on the table), it goes to sleep mode, which doesn’t mean the tracker left the environment. When we turn the tracker back on, the system will know which tracker it is and all of its proximity data. For this reason, I added the activation status (*IsActive* field). The default value of this field is “true” (the entity is active) for all entities. However, if the entity is not stable and ProxemicUI no longer receives its data, ProxemicUI automatically changes its status and notifies the developer about this change. This is important for developers at the application layer to draw the appropriate system response accordingly. In addition, the *EntityContainer* class used to have *EntityRemovedEvent* that would be fired when an entity is deleted (no longer being tracked following DT-Dt's approach). However, replaced the *EntityRemovedEvent* with the *EntityInactiveEvent*, which will be automatically when the entity is no longer being tracked. Besides, I also added an additional querying method based on the activation status that is *EntityRetrievalByActivationStatus.*

MobilityRule

Initially, each entity in ProxemicUI v1 has *IsStable* field, which can be used to check if the entity is stable or mobile. However, some entities might change their stability status over time. For example, a tablet can be stationary for some time on the kitchen island when reading recipes while cooking. It can also be mobile when moving around the smart home to control other devices. To update the field, developers would write a MobilityRule and pass the ids of entities to update their stability status. Then, ProxemicUI uses this rule to update the stability status of these entities and notifies developers when the update is done.

Test condition

Initially, ProxemicUI v1 tests all entities against thresholds in the rule, then fires events if the test passes for all entities. In some scenarios, this approach will be time-consuming and requires a lot of effort to implement. For example, in Alice's scenario we discussed above, we have a controller that controls all smart appliances at home. To implement this using ProxemicUI v1, developers would have to create multiple rules (a rule between the controller and each device). To overcome this issue and allow developers to test multiple entities simultaneously in the same rule, I added the *TestCondition* field to the basic proximity rules. Using the *TestCondition*, developers can specify if the test results are required for all entities or just a subset.

IsFacingRule

Initially, ProxemicUI has only RelativeOrientationRule, which checks if two or more entities are facing each other within a predefined threshold (it checks the relative orientation based on the facing direction of both entities). However, what if we have a scenario where we only care about one or more entities facing another entity. For example, if we want to check if two users are facing a tabletop or not, no matter which side of the table they are at, they might be standing at what we might consider the back/side of the table. IsFacingRule solves this problem.

Small tweaks and refinements

In addition to the previous design requirements that I derived from using ProxemicUI v1 to support the smart home scenario, I completed some refinements to ProxemicUI v1 to: support a wider range of tracking systems, organize the code by moving some fields to where they belong, and reflect the changes that were completed for the new design requirements discussed above. This section summarizes these refinements.

- Changing ID from type integer to string because some tracking systems use a combination of letters and numbers as tracker id.
- Shifting some properties (e.g., width and length) from proximity entity to shape. They are the properties of the shape and left proximity entity with only the most abstract properties (e.g., ID, type of shape, isActive, isStable, etc.).
- Changing the code in *DataReceiver* and *EntityContainer* classes to match the new format of OSC messages. This includes 3D position and orientation data and shape data (e.g., width and length) of entities. They also include defining additional constructors and OSC message formats.
- Replacing *EntityRemovedEvent* with *EntityInactiveEvent* in *EntityContainer*.
- Implementing ProximityEventArgs class that contains all different object types to pass data of different events to developers.
- Changing OnEvent to OnEventTrue and OnEventFalse for all rules.

- Adding multiple class constructors for all rules to cover all types of tests (one-to-one, one-to-many, and many-to-many) to make it easier for developers to pass arguments.

Hackathon User Study

In addition to using the proof-of-concept applications as a tool to draw an initial evaluation of ProxemicUI, I also intended to use these applications in a hackathon user study to gain feedback about ProxemicUI from external developers. I was planning to use these applications to create a smart home environment where I bring in external users to connect these applications based on proximity data using ProxemicUI. Unfortunately, I had to cancel this user study due to the restriction of COVID-19 and evaluate ProxemicUI through two other methods (will be discussed in the next two chapters). This section will briefly discuss the hackathon user study, and the ethics application that contains the full study design and the letter of approval are in appendix 3 and 4.

I chose the hackathon to generate qualitative and experimental data from a relatively small sample in a short period of time. With such a method, I can learn what aspects of the framework developers like and what aspects they get stuck in, leading to suggestions for improvements. In addition, the hackathon participants might find something that I never looked at before; these might be different ways of using the features of the framework or different environments where the framework can be used. I was planning to run the study over the course of 2.5 days (a weekend). During the study, participants would use ProxemicUI to control mock home appliances (the proof-of-concept applications discussed above) based on proxemic data (e.g., when I sit on the couch, turn on the TV and launch the remote control app on my phone). In this evaluation, I am following one of the demonstration techniques discussed by Ledo et al.[65], and that is *replicated examples*, where the design of the proof-of-concept applications is similar to the Proximity-Aware control design [64], which was also implemented using the Proximity toolkit. This is important to show how ProxemicUI implements similar examples in a more fixable way. In addition to asking participants to connect the proof-of-concept applications, I was also planning to allow them to explore different designs/usage of the framework by implementing their scenarios.

The main purpose of this study was to establish that developers with limited exposure to the kinds of applications supported by the framework can rapidly prototype using it. This could be achieved by considering how two key innovations of the framework are used: Hybrid Rules (connecting tracking data with user interface events) and Compound Rules (composing more complex rules from basic building blocks). To accomplish this, I would collect and analyze a variety of data: video recording of the development process, group discussions/reflections, screen capture of a testing computer (that we provide), final demo presentations and critiques, and source code. All videos would be transcribed and annotated using an open coding process to identify themes during development and group discussion. Source code and screen capture will be reviewed and annotated to itemize what elements of the framework were used, what syntactic or semantic errors occurred, and how fluency with the framework emerged over the weekend. The group discussions will also be used to obtain detailed feedback and suggestions about the framework.

In the hackathon user study, I was planning to recruit a minimum of four and a maximum of six groups of three (12-18 participants) who know how to code with C# because the

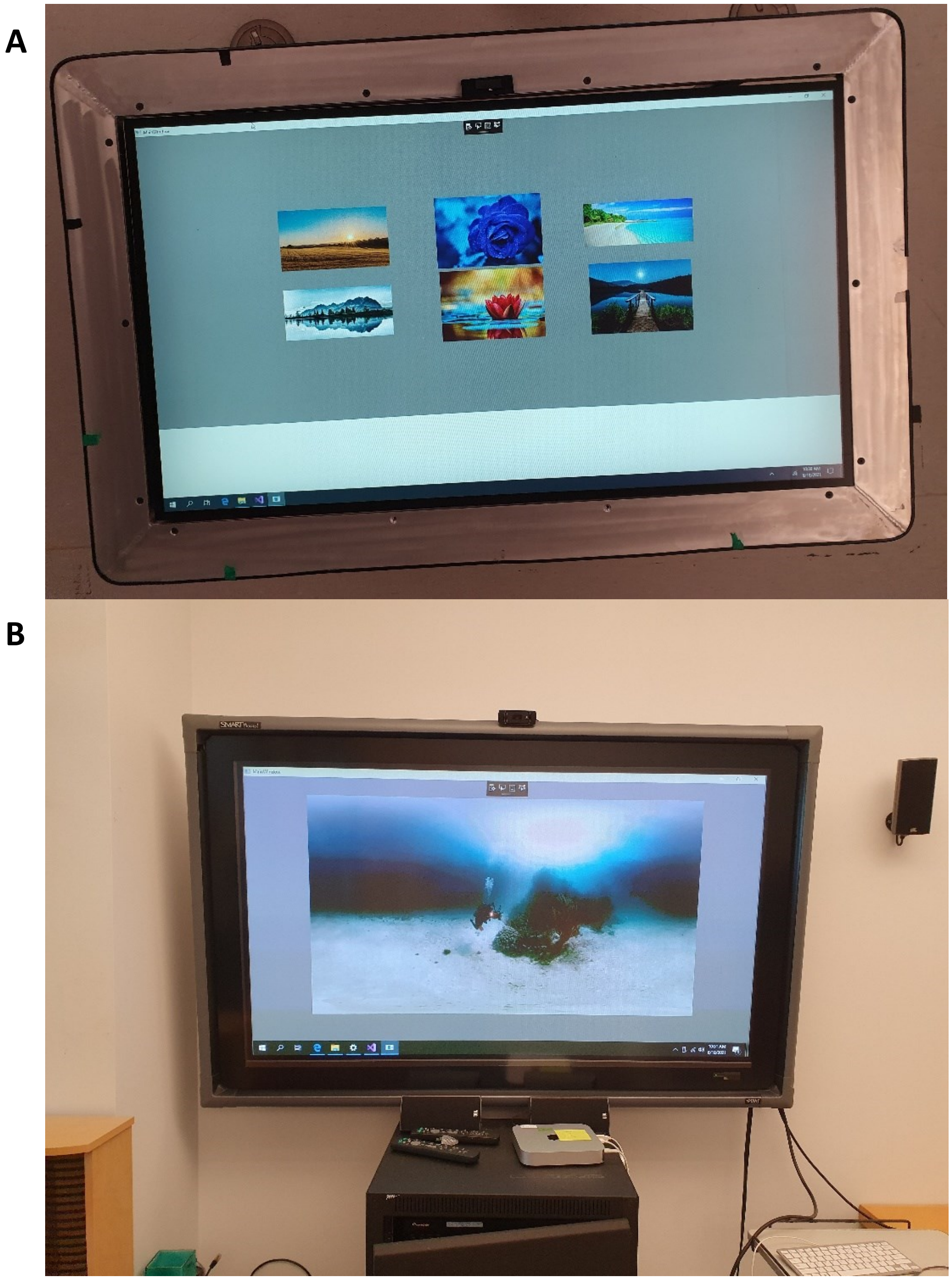


FIGURE 39: A) THE TABLETOP IS RUNNING ITS APPLICATION, B) THE TV IS PLAYING A MEDIA THROUGH ITS APPLICATION

framework was developed using C#. The plan was to run the study over the course of 2.5 days (a weekend: Friday 5:00-8:00 pm and Saturday and Sunday 9:00 am – 5:00 pm). The study would start with a mini workshop where participants would be exposed to some examples of existing work that demonstrates the use of proximity in smart environments. This workshop would end with a hands-on lab session to give the participants a guided experience using the framework. This session would introduce the proof-of-concept applications to participants and use the alarm application as an example to guide participants to try to write the code for different rules that ProxemicUI provides, including ProximityRules, CompoundRules, and HybridRule. Table 4 shows the schedule of the hackathon.

TABLE 4: SCHEDULE OF THE HACKATHON OVER THE WEEKEND

Time Slot	Task
Friday (5:00 – 8:00 pm)	Mini workshop (overview of ProxemicUI + hands-on experience)
Saturday (9:00 am - 4:00 pm)	Using the framework to control the proof-of-concept applications to make them proxemic-aware and responsive
Saturday (4:00 - 5:00 pm)	Discussion session to comment about the pros and cons of the framework and suggestions for improvement
Sunday (9:00 am – 3:00 pm)	Using the framework to imagine and implement their proxemic-aware application
Sunday (3:00 - 4:00 pm)	Discussion session to comment about the pros and cons of the framework and suggestions for improvement
Sunday (4:00 pm – 5:00 pm)	Each group would present their work to a jury. Announce the winning team and runner, then award prizes

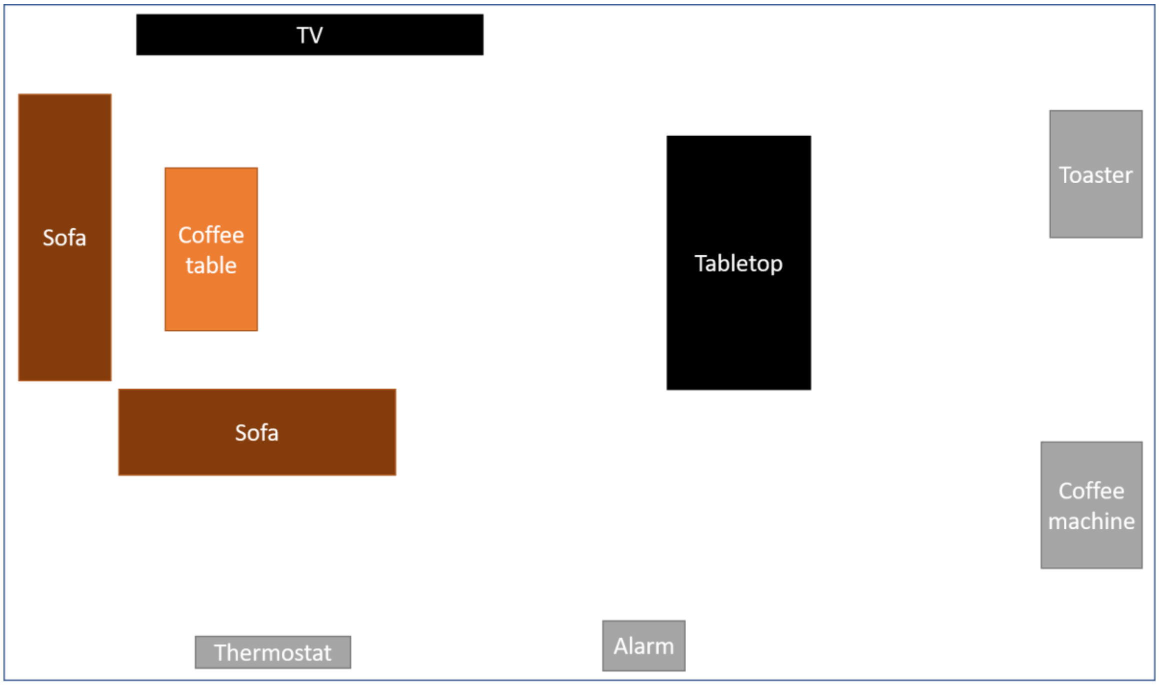


FIGURE 40: FLOOR LAYOUT SHOW HOW TO PLACE DEVICES IN THE LAB FOR THE HACKATHON

Summary

This chapter discussed the proof-of-concept applications I developed to be used in a hackathon user study. It also briefly discussed the study design for the hackathon user study, which I could not conduct due to COVID-19 restrictions. While I could not conduct the hackathon user study, this chapter also discussed how ProxemicUI was improved by applying it to a smart home setup. For example, the HybridRule is used to combine proximity events with UI events. Still, I had system notifications (e.g., the coffee is ready) instead of UI events when applying ProxemicUI to a smart home setup. Therefore, I defined a more generic HybridRule that takes any external events. Other major improvements include defining IsFacingRule and MobilityRule, adding the activation status as an attribute to the proximity entity, adding the test condition to rules, and extending the OSC messages. The next chapter (chapter 7) discusses evaluating ProxemicUI in a code review study.

Chapter 7: Evaluating ProxemicUI Through a Code Review Study

Validating a framework by external users is important to show that the framework can support users' needs. Unfortunately, due to COVID-19, I could not conduct the planned hackathon to gain that feedback. I chose not to run the hackathon online as it requires repeated testing with physical components in the lab (e.g., moving the tablet to the tabletop to exchange files) to make sure the code responds correctly. I followed a different approach to gain developer feedback about the toolkit. One of the evaluation techniques through usage is "*comparison*" between the new toolkit and a baseline. "Baselines include not having a toolkit, or working with a different toolkit" [65]. Comparing ProxemicUI against existing toolkits with the same or similar intended use can show the improvements over the current state-of-the-art, including the benefits of the ProxemicUI programming model. Therefore, I conducted a code review study to compare ProxemicUI to The Proximity toolkit [71] and Microsoft PSI (Platform for Situated Intelligence) [87]. I chose the Proximity Toolkit because, through my literature review of systems and tools for proxemics-awareness, it is the closest toolkit in terms of intended use (an object-oriented toolkit to build proxemic-aware applications) to ProxemicUI and is commonly used and cited by the research literature. Other toolkits provide support for the development of proxemic-aware applications, but I did not consider them: Schipor et al.'s work [99] is not a fully implemented toolkit and only has an online simulation tool to show its functionalities. Pérez et al.'s work [84][85] has a number of limitations, as discussed in the background chapter (e.g., detecting orientation based on detected faces, which is not suitable for every scenario such as human-to-device interactions, devices with no cameras, and device-to-device interactions). I chose Microsoft PSI because it is an emerging standard for data-driven (machine-learning) approaches to smart environment applications. It is also being adopted as a research platform by researchers who conducted the foundational work in proxemics interaction research and developed the Proximity Toolkit. During the code review, participants walked through two separate C# program segments that implemented the same application, each using a different toolkit (ProxemicUI and the Proximity Toolkit). Participants also walked through to the procedures and code required to use ProxemicUI vs. Microsoft PSI alongside a machine learning classifier to support proxemic-aware applications.

Objectives for the code review study

My primary research objective for the code review was to receive external developer feedback regarding different approaches to creating proxemics-aware applications offered by my framework in comparison to the others. This can be formulated broadly as:

- ❖ What benefits and drawbacks exist when using ProxemicUI compared to existing toolkits (specifically Proximity toolkit and Microsoft Psi) to build proxemics-aware applications?

To achieve this objective, I ask the following research questions:

- ➢ Is using basic proximity rules to implement proxemic-aware scenarios easier for developers than approaches offered by existing toolkits? And if so, why?

- Is using Compound Rules to implement complex scenarios easier for developers than approaches offered by existing toolkits? And if so, why?
- Is using Hybrid Rules to combine a proximity rule with an external event easier for developers than approaches offered by existing toolkits? And if so, why?
- Which toolkit would provide better support for a proxemic-aware machine learning classifier in terms of generating the training data and feeding the classifier with proximity data?
- In which ways can the framework be improved?

Study design

This section will discuss the study design, including the study population and recruitment process, the study environment, the study procedure, the data collection and analysis procedure, and the study results.

Participants

I recruited six groups of three, 6 undergrad (from first to the fourth year) and 12 grad (master and Ph.D.) students (3 females and 15 males) from the Faculty of Computer Science at Dalhousie University. To be eligible to participate in the study, all participants must be familiar with coding in C# (as ProxemicUI API was developed using C#) and Java as it is similar to C#, an object-oriented programming language. Familiarity with the Internet of Things and/or mixed/augmented reality is an asset but not strictly required for participation. I chose participants based on their self-declared experience when they responded to the recruitment notice.

Recruitment

I recruited participants by email announcements through the Computer Science mailing list (cs-jobs@kil-lsv-2.its.dal.ca). In the recruitment notice, participants were asked to email their interest to participate to the listed researcher. The participants and researcher communicated to find an appropriate time for the study.

Informed consent

All participants involved in the study signed an informed consent form (Appendix 5.1). When participants indicated their interest in participating in the study, I emailed them a copy of the consent form to ready, sign, and send it back before the study is scheduled. The informed consent outlined the risks and benefits associated with the study, a description of the study, the participant's right to withdraw without consequence (or losing the compensation), and assurances of confidentiality and anonymity of personal data.

Compensation

Each participant received CAD 30 for their participation in the study.

Study environment

Due to the COVID-19 pandemic, all sessions conducted online using Microsoft Teams. During every session, I shared my screen with participants to present study material (presentation slides and source code). I also recorded all sessions, including presentations and discussions, using Microsoft Teams. As Microsoft Teams allows participants to gain control of a machine that shares its screen, while I did not introduce this feature to participants, there was one instance where participants requested control to point at different parts of the presented source codes during the discussion.

Study Procedure

In the code review study, each group had three 1-hour sessions. Two sessions are dedicated to comparing ProxemicUI and the Proximity Toolkit, and one session is dedicated to comparing ProxemicUI and Microsoft PSI. To mitigate learning effects, I flipped the order of exposure to the toolkits. I started each comparison with a short overview of each toolkit so participants will know the toolkits (e.g., an overview of ProxemicUI and the Proximity Toolkit before comparing them). As the code segments were presented, participants were encouraged to ask questions and ask to review implementation details or documentation as the walkthrough is taking place. After introducing each piece of the code segments, I started by giving participants a chance to comment or discuss any thoughts they have about the two code segments. Then, they participated in a discussion where I asked them about the pros and cons of the approach of each toolkit. Every time, I started by asking a different participant to give everyone a chance to express their opinion before it gets affected by other's opinions.

TABLE 5: LIST OF QUESTIONS ASKED DURING THE SESSIONS TO ANSWER EACH RESEARCH QUESTION

Research questions	Related discussion questions
Is using basic proximity rules to implement proxemic-aware scenarios easier for developers than approaches offered by existing toolkits? And if so, why?	- The Proximity Toolkit creates a relation between two entities to detect proxemics events where the system keeps tracking all proxemics attributes looking for a change. On the other hand, ProxemicUI allows developers to choose which attributes to check. Which approach do you think is more sufficient? Explain. - To detect proxemics events using the Proximity Toolkit, developers need to create a relation between two entities; this means that developers will have to create multiple relations if they want to detect the event between multiple entities simultaneously. On the other hand, ProxemicUI allows developers to pass multiple entities in a single rule. Which approach do you think is more sufficient? Explain. - The Proximity Toolkit provides developers with proxemic data, where they have to test these data with their threshold (e.g., The Proximity Toolkit provides the distance to developers, then developers have to test this distance with their threshold). On the other hand, ProxemicUI asks developers to pass their thresholds (minimum and maximum), then only fires the event if the distance within these thresholds. Which approach do you think is more sufficient? Explain. - Can you think of a better way to solve these problems?
Is using Compound Rules to implement complex scenarios easier for developers than approaches offered by existing toolkits? And if so, why?	- When testing multiple proxemics attributes (distance and orientation) using the Proximity Toolkit, developers need to listen for two different events. On the other hand, using the Compound rules in ProxemicUI, developers can combine both tests in a single rule and only listen for a single event. Which approach is more sufficient? Explain. - Can you think of a better way to solve this problem?
Is using Hybrid Rules to combine a proximity rule with an external event easier for developers than approaches offered by existing toolkits? And if so, why?	- When testing an external event (e.g., UI) with a proxemics event using the Proximity Toolkit, developers have to write two handing methods: one is for the external event (where the system will start to test the proxemics event). The second and one is for the proxemics event when it is fired to draw the system response. On the other hand, using Hybrid rules in ProxemicUI, developers need to call the external event method as a handling method for the external event (e.g., the button click) and then write a single handling method for the Hybrid event. Which approach is more sufficient?

Research questions	Related discussion questions
Which toolkit would provide better support for a proxemic-aware machine learning classifier in terms of generating the training data and feeding the classifier with proximity data?	- Which toolkit (Microsoft Psi vs. ProxemicUI) do you think is more suitable to generate training data for machine learning classifier to support proxemics-aware applications? Explain. - If you would integrate one of these toolkits (Microsoft Psi or ProxemicUI) with a machine learning classifier that receives high-level proximity data (e.g., the relative distance), then predict what response to draw, which toolkit do you think will be more suitable for this task? Explain. - If you would integrate one of these toolkits (Microsoft Psi or ProxemicUI) with a machine learning classifier that receives raw proximity data (e.g., position and orientation). The classifier then calculates the high-level proximity data (e.g., the relative distance) to draw a response, which toolkit do you think is more suitable to support (by providing low-level proximity data or calculate high-level proximity data) the classifier with this task? Explain
In which ways can the framework be improved?	- For all discussed points above, I asked this question: Can you think of a better way to solve this problem?
General questions	- Which toolkit do you think minimizes the developer's effort more? - Which toolkit provides straightforward steps for developers to create proxemics applications?

ProxemicUI vs. The Proximity Toolkit

As stated above, both The Proximity Toolkit and ProxemicUI were built with the same intended use, supporting building proxemics-aware applications. Therefore, during the code review sessions, participants were exposed to (via "code walkthrough") two different C# code segments that implement the same applications (solve the same problems). Each code uses a different toolkit (either ProxemicUI or The Proximity Toolkit). I used what I implemented in the proof-of-concept applications (smart home setup discussed in chapter 6) as a context scenario for this comparison so that participants have a better understanding of what the codes do. This comparison focused on four comparisons as follows:

1- Testing a single proximity attribute between two entities: to answer the research question "Is using basic proximity rules to implement proxemic-aware scenarios easier for developers than approaches offered by existing toolkits? And if so, why?"

2- Testing a single proximity attribute between more than two entities: to answer the research question "Is using basic proximity rules to implement proxemic-aware scenarios easier for developers than approaches offered by existing toolkits? And if so, why?"

3- Compound testing: to answer the research question "Is using Compound Rules to implement complex scenarios easier for developers than approaches offered by existing toolkits? And if so, why?"

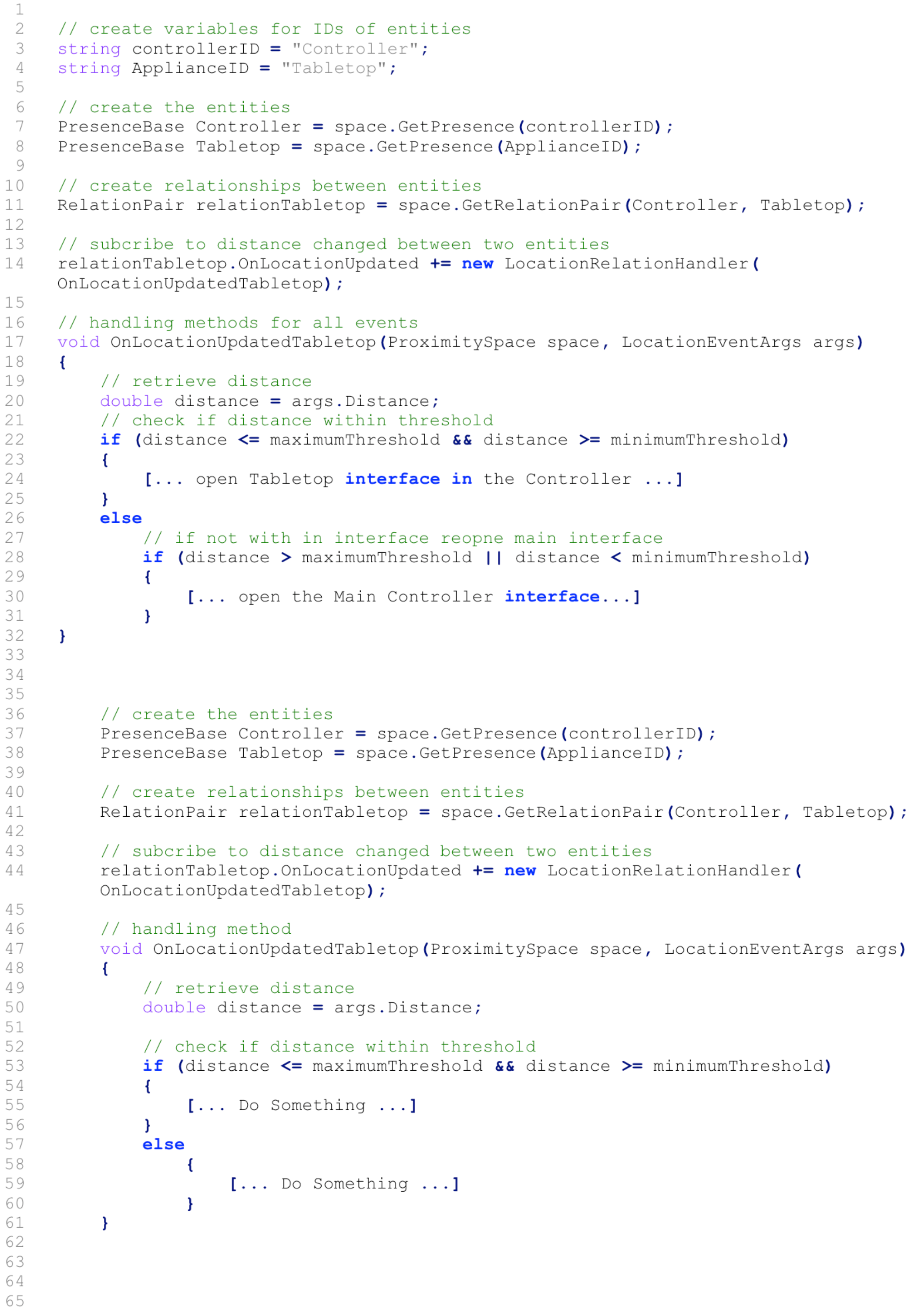
4- Hybrid testing: to answer the research question "Is using Hybrid Rules to combine a proximity rule with an external event easier for developers than approaches offered by existing toolkits? And if so, why?"

Testing a single proximity attribute between two entities

This test checks a single proximity attribute that can be relative distance, relative orientation, relative facing direction, absolute position, and absolute orientation. The code samples show how each toolkit implements the relative distance test between two entities. The scenario for this test is as follows:

When the user gets closer to the tabletop, the system opens the tabletop interface on the tablet.

To do this test, the Proximity Toolkit creates an object for each entity ("PresenceBase"), then pairs those entities by creating a "RelationPair" to access the relative proximity attributes between them. Lastly, through the "RelationPair", developers would subscribe to any event (in this example, "OnLocationUpdate") based on the attribute they want to test; then, write the handling method for that event. The Proximity Toolkit passes the values of the tests to the handling method, where developers would do their checks. On the other hand, ProxemicUI creates entities internally (but can be done manually as well) when receiving the tracking data through OSC messages. Then, developers can choose what rule to create according to the attribute they want to test (in this example, "RelativeDistanceRule"). Developers then add the rule to the "RuleEngine", subscribe to "OnEventTrue" and "OnEventFalse" for that rule, and write the handling methods for those events. Unlike the Proximity Toolkit, ProxemicUI provides developers with the final results by firing one of two events ("OnEventTrue" and "OnEventFalse"). Figure 41 shows performing the test using the Proximity Toolkit, and figure 42 shows performing the test using ProxemicUI.



```

// create variables for IDs of entities
string controllerID = "Controller";
string ApplianceID = "Tabletop";

// create the entities
PresenceBase Controller = space.GetPresence(controllerID);
PresenceBase Tabletop = space.GetPresence(ApplianceID);

// create relationships between entities
RelationPair relationTabletop = space.GetRelationPair(Controller, Tabletop);

// subcribe to distance changed between two entities
relationTabletop.OnLocationUpdated += new LocationRelationHandler(
OnLocationUpdatedTabletop);

// handling methods for all events
void OnLocationUpdatedTabletop(ProximitySpace space, LocationEventArgs args)
{
    // retrieve distance
    double distance = args.Distance;
    // check if distance within threshold
    if (distance <= maximumThreshold && distance >= minimumThreshold)
    {
        [... open Tabletop interface in the Controller ...]
    }
    else
        // if not with in interface reopne main interface
        if (distance > maximumThreshold || distance < minimumThreshold)
        {
            [... open the Main Controller interface...]
        }
}



    // create the entities
    PresenceBase Controller = space.GetPresence(controllerID);
    PresenceBase Tabletop = space.GetPresence(ApplianceID);

    // create relationships between entities
    RelationPair relationTabletop = space.GetRelationPair(Controller, Tabletop);

    // subcribe to distance changed between two entities
    relationTabletop.OnLocationUpdated += new LocationRelationHandler(
    OnLocationUpdatedTabletop);

    // handling method
    void OnLocationUpdatedTabletop(ProximitySpace space, LocationEventArgs args)
    {
        // retrieve distance
        double distance = args.Distance;

        // check if distance within threshold
        if (distance <= maximumThreshold && distance >= minimumThreshold)
        {
            [... Do Something ...]
        }
        else
            {
                [... Do Something ...]
            }
    }
```

FIGURE 41: TESTING RELATIVE DISTANCE BETWEEN TWO ENTITIES USING THE PROXIMITY TOOLKIT

```
// create variables for IDs of entities
string controllerID = "LHR-D725811B";
string ApplianceID = "Tabletop";

//create the rule
RelativeDistanceRule DistanceToTabletop = new RelativeDistanceRule(minimumThreshold, maximumThreshold, controllerID, ApplianceID, "ANY");

// add rule to the rule engine
RuleEngine.Instance.AddToRuleList(DistanceToTabletop);

// subscribe to onEventTrue and onEventFalse
DistanceToTabletop.OnEventTrue += DistanceToTabletop_OnEventTrue;
DistanceToTabletop.OnEventFalse += DistanceToTabletop_OnEventFalse;

// write the handling method for onEventTrue
private void DistanceToTabletop_OnEventTrue(Rules rule, ProximityEventArgs proximityEvent)
{
    [... open Tabletop interface in the Controller ...]
}

// write the handling method for onEventFalse
private void DistanceToTabletop_OnEventFalse(Rules rule, ProximityEventArgs proximityEvent)
{
    [... open the Main Controller interface...]
}
```

FIGURE 42: TESTING RELATIVE DISTANCE BETWEEN TWO ENTITIES USING PROXEMICUI

Testing a single proximity attribute between more than two entities

The code samples in this test show how each toolkit implements the relative distance test between multiple entities. The scenario for this as follows:

> *The user's tablet plays as a controller for multiple appliances (in this example, the TV and the tabletop) in her/his home. When the user gets closer to any appliance, the system opens the interface for that appliance on the tablet.*

The main difference between ProxemicUI and The Proximity toolkit to implement this test is that ProxemicUI allows developers to include multiple entities in a single test (e.g., one-to-many). On the other hand, because The Proximity Toolkit allows for one-to-one relations only, developers need to create multiple "RelationsPairs" for every two entities they need to test. In addition, with ProxemicUI, developers can retrieve entities easily using one of the methods in ProxemicUI (e.g., "EntityRetrievalAll"); wherewith The

Proximity Toolkit, developers would create an object ("PresenceBase") for each entity. Figure 43 shows performing the test using the Proximity Toolkit, and figure 44 shows performing the test using ProxemicUI.

```

// create variables for IDs of entities
string controllerID = "Controller";
string TVID = "TV";
string TabletopID = "Tabletop";

// create the entities
PresenceBase Controller = space.GetPresence(controllerID);
PresenceBase TV = space.GetPresence(TVID);
PresenceBase Tabletop = space.GetPresence(TabletopID);

// create relationships between entities
RelationPair relationTV = space.GetRelationPair(Controller, TV);
RelationPair relationTabletop = space.GetRelationPair(Controller, Tabletop);

// subcribe to distance changed between two entities
relationTV.OnLocationUpdated += new LocationRelationHandler(OnLocationUpdatedHandler);
relationTabletop.OnLocationUpdated += new LocationRelationHandler(OnLocationUpdatedHandler);

// handling methods for all events
void OnLocationUpdatedHandler(ProximitySpace space, LocationEventArgs args)
{
    // retrieve distance
    double distance = args.Distance;

    // check if distance within threshold
    if (distance <= maximumThreshold && distance >= minimumThreshold)
    {
        // retrieve names of entities
        List <string> Names = new List <string>(){ args.PresenceA.Name, args.PresenceB.Name};
        // call the factory method to check which interface to open
        Page page = ApplianceFactory(Names);

        [... open page interface in the Controller ...]
    }
    else
        // if not with in interface reopne main interface
        if (distance > maximumThreshold || distance < minimumThreshold)
        {
            [... open the Main Controller interface...]
        }
}

private Page ApplianceFactory(List<string> Names)
        {
            // walk through list of names
            foreach (string value in Names)
            {
                switch (value)
                {
                    case Tabletop.Name:
                        return new Tabletop();
                    case TV.Name:
                        return new TVInterface();
                }
            }
            return null;
        }

```

FIGURE 43: TESTING RELATIVE DISTANCE BETWEEN MULTIPLE ENTITIES USING THE PROXIMITY TOOLKIT

```
// create lists to add IDs to
public static List<string> AllEntities = new List<string>();

// Add IDs to the lists
string controllerID = "LHR-D725811B";
AllEntities = EntityContainer.EntityRetrievalAll();

//create the rule
RelativeDistanceRule DistanceToAppliance = new RelativeDistanceRule(minimumThreshold, maximumThreshold, controllerID, AllEntities, "ANY");

// add rule to the rule engine
RuleEngine.Instance.AddToRuleList(DistanceToAppliance);

// subscribe to onEventTrue and onEventFalse
DistanceToAppliance.OnEventTrue += DistanceToAppliance_OnEventTrue;
DistanceToAppliance.OnEventFalse += DistanceToAppliance_OnEventFasle;

// write the handling method for onEventTrue
private void DistanceToAppliance_OnEventTrue(Rules rule, ProximityEventArgs proximityEvent)
        {
            // retrieve entities that passed the test
            var args = (RelativeBasicProximityEventArgs)proximityEvent;

            // call the factory method to check which interface to open
            Page page = ApplianceFactory(args.EntityListsForAll.SecondList);

            [... open page interface in the Controller ...]
        }

// write the handling method for onEventFalse
private void DistanceToAppliance_OnEventFasle(Rules rule, ProximityEventArgs proximityEvent)
        {
            [... open the Main Controller interface...]
        }

private Page ApplianceFactory(List<string> IDs)
        {
            // walk through IDs list
            foreach (string value in IDs)
            {
                switch (value)
                {
                    case "Tabletop":
                        return new Tabletop();
                    case "TV":
                        return new TVInterface();
                }
            }
            return null;
        }
```

FIGURE 44: TESTING RELATIVE DISTANCE BETWEEN MULTIPLE ENTITIES USING PROXEMICUI

Compound testing

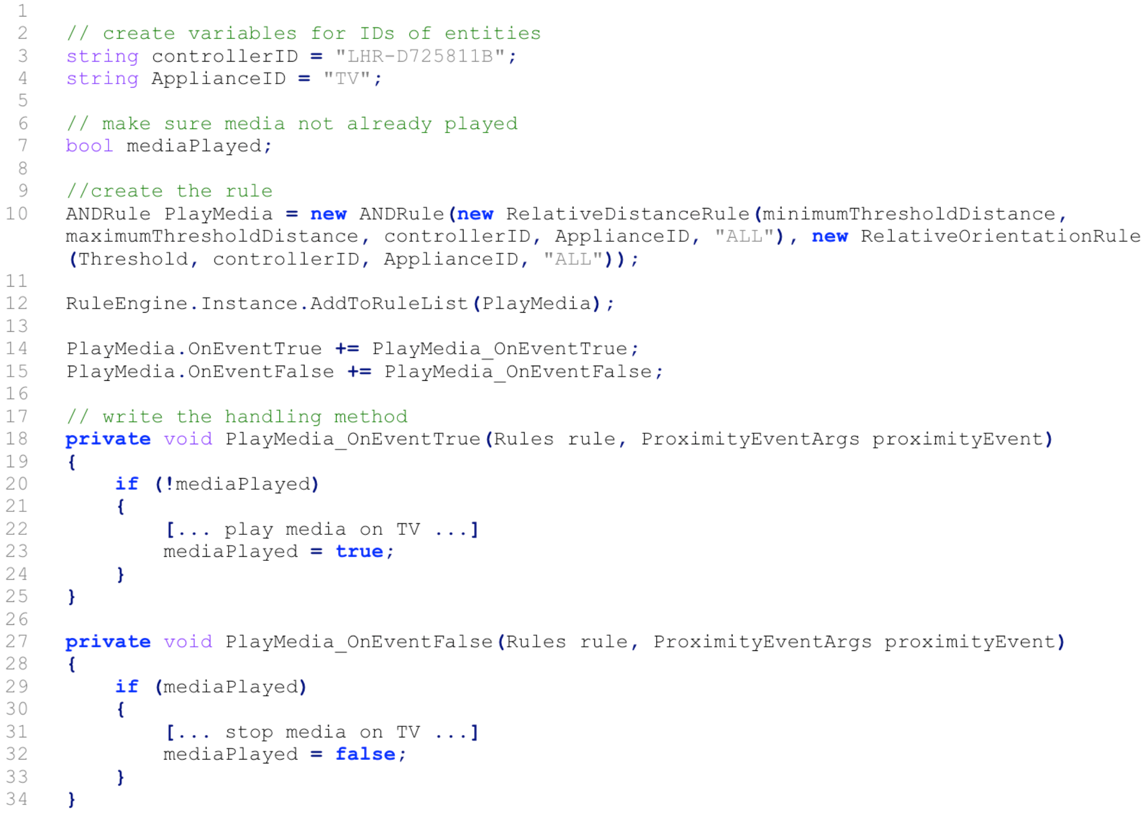
Compound testing can include several scenarios as follow. First, combining two tests for a single proximity attribute (e.g., relative distance), but each test has different thresholds or a different set of entities. Second, combining two tests, each for a different proximity attribute (e.g., one for relative distance and one for relative orientation), where they might share the same set of entities or have different ones. The scenario for this is:

The system should play media on the TV only when the user is close to the TV and facing the TV simultaneously

The main difference between ProxemicUI and The Proximity toolkit to implement this test is that ProxemicUI can combine two rules into a single one, then listen for the events of that rule only (e.g., listen for the events of ANDRule). In contrast, The Proximity Toolkit creates a relation pair between two entities, then listen for two events and check the results to make sure they both passed. Besides, if the test between more than two entities, then The Proximity Toolkit would create multiple relation pairs, then listen for two events of all these relation pairs. Figure 45 shows performing the test using the Proximity Toolkit, and figure 46 shows performing the test using ProxemicUI.

```

// create variables for IDs of entities
string controllerID = "Controller";
string ApplianceID = "TV";

// create the entities
PresenceBase Controller = space.GetPresence(controllerID);
PresenceBase TV = space.GetPresence(ApplianceID);

// to track events changes and to make sure media not already played
bool inDistance = false, isFacing = false, mediaPlayed = false;

// create relationships between entities
RelationPair relationTV = space.GetRelationPair(Controller, TV);

// subcribe to distance and orientation changed events between two entities
relationTV.OnLocationUpdated += new LocationRelationHandler(OnLocationUpdatedTV);
relationTV.OnDirectionUpdated += new DirectionRelationHandler(OnDirectionUpdatedTV);

// to keep checking the events
while (true)
{
    if (inDistance && isFacing && !mediaPlayed)
    {
        [... play media on TV ...]
        mediaPlayed = true;
    }
    else
        if ((!inDistance || !isFacing) && mediaPlayed)
        {
            [... stop media on TV ...]
            mediaPlayed = false;
        }
}

// handling methods for both events
void OnLocationUpdatedTV(ProximitySpace space, LocationEventArgs args)
{
    double distance = args.Distance;
    if (distance <= maximumThreshold && distance >= minimumThreshold)
    {
        inDistance = true;
    }
    else
        if (distance > maximumThreshold || distance < minimumThreshold)
        {
            inDistance = false;
        }
}

void OnDirectionUpdatedTV(ProximitySpace space, DirectionEventArgs args)
{
    if (args.ATowardsB || args.BTowardsA)
    {
        isFacing = true;
    }
    else
    {
        isFacing = false;
    }
}

```

FIGURE 45: COMBINING DISTANCE AND ORIENTATION TESTS USING THE PROXIMITY TOOLKIT



```

// create variables for IDs of entities
string controllerID = "LHR-D725811B";
string ApplianceID = "TV";

// make sure media not already played
bool mediaPlayed;

//create the rule
ANDRule PlayMedia = new ANDRule(new RelativeDistanceRule(minimumThresholdDistance,
maximumThresholdDistance, controllerID, ApplianceID, "ALL"), new RelativeOrientationRule
(Threshold, controllerID, ApplianceID, "ALL"));

RuleEngine.Instance.AddToRuleList(PlayMedia);

PlayMedia.OnEventTrue += PlayMedia_OnEventTrue;
PlayMedia.OnEventFalse += PlayMedia_OnEventFalse;

// write the handling method
private void PlayMedia_OnEventTrue(Rules rule, ProximityEventArgs proximityEvent)
{
    if (!mediaPlayed)
    {
        [... play media on TV ...]
        mediaPlayed = true;
    }
}

private void PlayMedia_OnEventFalse(Rules rule, ProximityEventArgs proximityEvent)
{
    if (mediaPlayed)
    {
        [... stop media on TV ...]
        mediaPlayed = false;
    }
}
```

FIGURE 46: COMBINING DISTANCE AND ORIENTATION TESTS USING PROXEMICUI

Hybrid testing

Hybrid testing integrates any external event (e.g., in this example, the external event is UI event) with a proximity test (basic or compound proximity test). The scenario for this is:

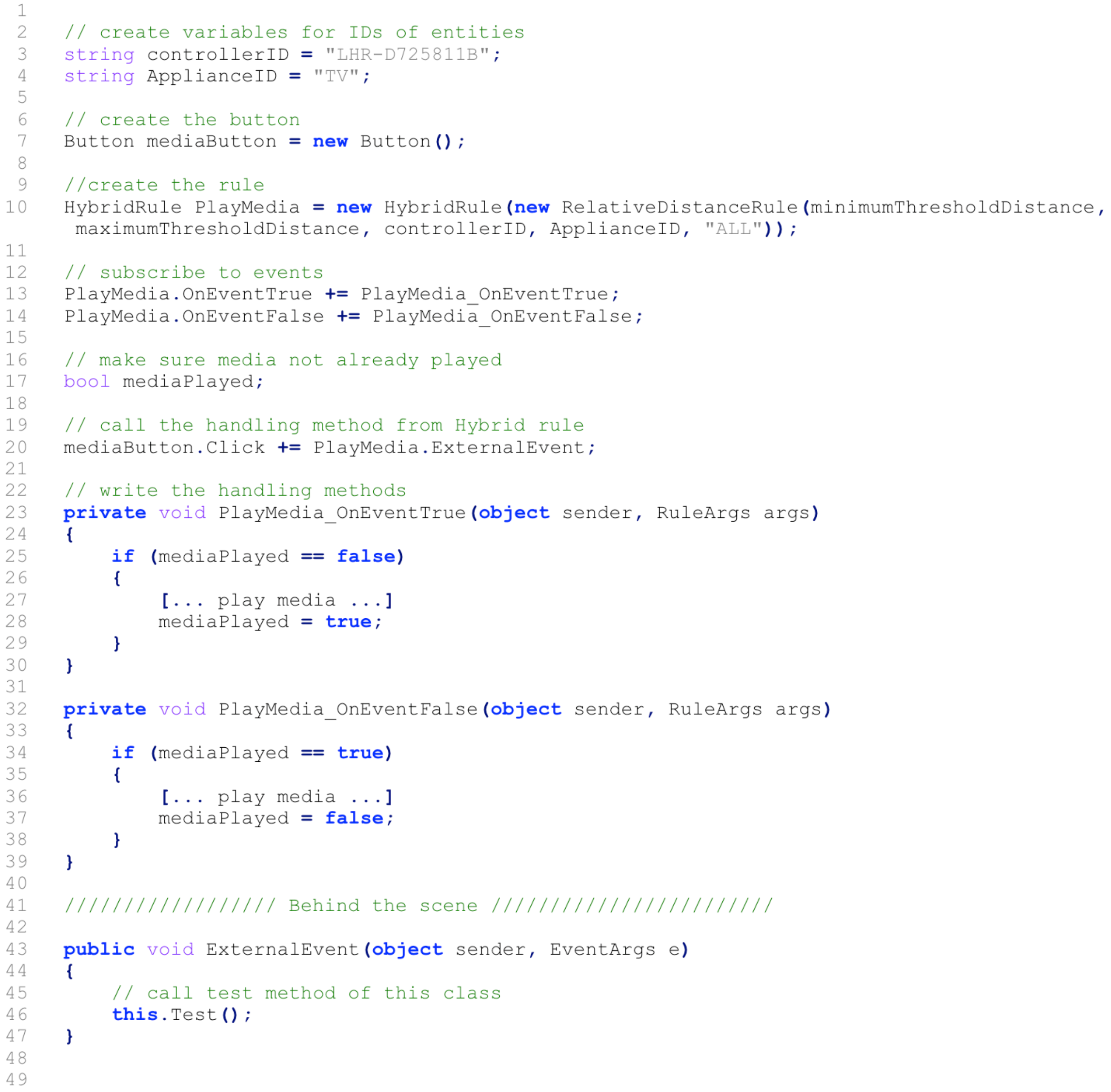
> *When the user touches the play media button on the tablet, the system checks if the user is within the threshold to the TV, it will play the media*

The difference between ProxemicUI and The Proximity Toolkit is that with The Proximity Toolkit, developers would listen to two different events (the proximity event and the external event); then check if both events pass to draw the system's response. This process involves checking the time for each event to make sure both events happened simultaneously. In contrast, ProxemicUI uses the "ExternalEvent" method inside the

“HybridRule” as a handling method for any external event, and developers will only listen for the events from that rule (OnEventTrue or OnEventFalse). With this approach, there is no need to check if both events happened simultaneously as the “ExternalEvent” method will test the proximity rule only when it is called. Figure 47 shows performing the test using the Proximity Toolkit, and figure 48 shows performing the test using ProxemicUI.

```
// create the button
Button MediaButton = new Button();

// subscribe to the event
MediaButton.click += MediaButton_Clicked;

// create variables for IDs of entities
string controllerID = "Controller";
string ApplianceID = "TV";

// create the entities
PresenceBase Controller = space.GetPresence(controllerID);
PresenceBase TV = space.GetPresence(ApplianceID);

// create relationship between entities
RelationPair relationTV = space.GetRelationPair(Controller, TV);

// make sure media not already played
bool mediaPlayed;

// used to make sure both events happne in the same time
DateTime buttonTime;
DateTime proximityTime;

// button clicked handler
private void MediaButton_Clicked(object sender, EventArgs e)
{
    // record time when button clicked
    buttonTime = DateTime.Now;
    // subscribe to location update event
    relationTV.OnLocationUpdated += new LocationRelationHandler(OnLocationUpdatedTV);
}

// location update handler
void OnLocationUpdatedTV(ProximitySpace space, LocationEventArgs args)
{
    // to make sure events occured at the same time
    proximityTime = DateTime.Now;
    TimeSpan duration = proximityTime - buttonTime;
    int seconds = (int)duration.TotalSeconds;

    if (Math.Abs(seconds) <= 5)
    {
        double distance = args.Distance;
        if (distance <= maximumThreshold && distance >= minimumThreshold && mediaPlayed == false)
        {
            [... Play media ...]
            mediaPlayed = true;
        }
    }
    else
        if (distance > maximumThreshold || distance < minimumThreshold && mediaPlayed == true)
        {
            [ ... Stop media ...]
            mediaPlayed = false;
            // unsubscribe if duration between two events is more than 5 seconds
            relationTV.OnLocationUpdated -= new LocationRelationHandler(OnLocationUpdatedTV);
        }
}
```

FIGURE 47: INTEGRATE UI EVENT WITH PROXIMITY EVENT USING THE PROXIMITY TOOLKIT



```
// create variables for IDs of entities
string controllerID = "LHR-D725811B";
string ApplianceID = "TV";

// create the button
Button mediaButton = new Button();

//create the rule
HybridRule PlayMedia = new HybridRule(new RelativeDistanceRule(minimumThresholdDistance,
 maximumThresholdDistance, controllerID, ApplianceID, "ALL"));

// subscribe to events
PlayMedia.OnEventTrue += PlayMedia_OnEventTrue;
PlayMedia.OnEventFalse += PlayMedia_OnEventFalse;

// make sure media not already played
bool mediaPlayed;

// call the handling method from Hybrid rule
mediaButton.Click += PlayMedia.ExternalEvent;

// write the handling methods
private void PlayMedia_OnEventTrue(object sender, RuleArgs args)
{
    if (mediaPlayed == false)
    {
        [... play media ...]
        mediaPlayed = true;
    }
}

private void PlayMedia_OnEventFalse(object sender, RuleArgs args)
{
    if (mediaPlayed == true)
    {
        [... play media ...]
        mediaPlayed = false;
    }
}

///////////////// Behind the scene //////////////////////

public void ExternalEvent(object sender, EventArgs e)
{
    // call test method of this class
    this.Test();
}
```

FIGURE 48: INTEGRATING UI EVENT WITH PROXIMITY EVENT USING PROXEMICUI

ProxemicUI vs. Microsoft PSI

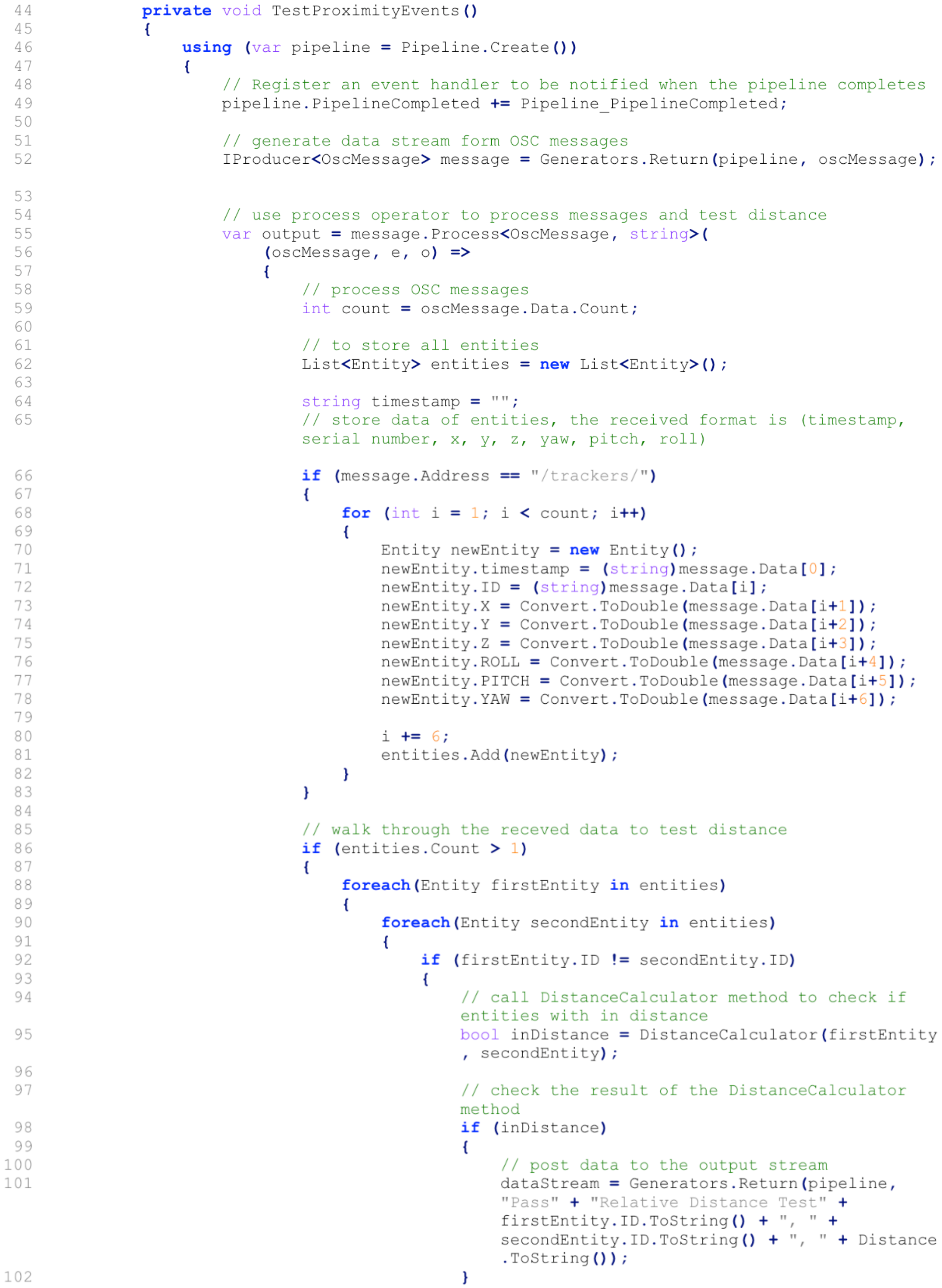
While Microsoft PSI and ProxemicUI are both designed to support situated, adaptive, and responsive smart infrastructure, they support different development paradigms and different ranges of application. ProxemicUI is an event-driven, rule-based object-oriented framework that supports the implementation of proxemics-aware applications, while Microsoft PSI is a data-driven, stream-oriented platform for machine learning applications in the broader domain of smart environments. It is not appropriate to compare them based on proxemics interaction recognition due to these differences, but it is still interesting and important to consider PSI as it could be used to create spatially adaptive systems. Therefore, I compare both toolkits in terms of how they could be used to train a machine learning classifier for proxemics events and as a data source. The main goal of training a machine learning classifier is to have a modal that can answer questions correctly [114]. A machine learning system consists of five steps: defining the problem and proposing a solution, constructing the dataset, transforming the data, training the model, and predictions [28]. I will discuss how constructing the dataset and predictions are related to comparing ProxemicUI and Microsoft PSI later in this section.

In this comparison, participants were exposed to (via "code walkthrough") two different C# code segments; each was implemented with a different toolkit. These code segments s show the steps required for ProxemicUI and Microsoft Psi to generate training data and provide the classifier with low-level proximity data (e.g., position and orientation) and high-level proximity data (e.g., relative distance and orientation). It is important to state that I am only considering generating training data and providing the classifier with low/high-level proximity data to start prediction. Other steps to train the model (e.g., preparing the data samples and training the classifier) were not parts of this study as they are more related to training the classifier and not generating the proximity data. Besides, this comparison looks at how straightforward to generate the data using both tools and not that if the generated data by one tool has a better quality over the other one. In addition, both toolkits provide a mechanism to combine data coming from two different sources (ProxemicUI using Hybrid Rule and Microsoft PSI using "Join" operator). While combining two sources of data is not related to machine learning, it is still worth exploring how each toolkit performs the combination for proxemic use. The C# code segments for this comparison show how each toolkit can integrate external events with proximity events (Hybrid rule). This section will discuss employing both toolkits to support the machine learning classifier and combine external events with proximity events.

Generating training data

Constructing the dataset consists of four steps: "collect the raw data, identify features and label sources, select a sampling strategy, and split the data" [28]. In this point of comparison, I am only focusing on collecting the raw data using both toolkits and identifying the features and label sources. This would involve receiving the proximity data from a tracking system, calculating various proximity attributes, testing if proximity attributes passed or failed, and finally, writing the results into a file to create the dataset. For the sake of comparison, I used only calculating a single proximity attribute just to show the steps that we need to take to generate the data. Figures 49 and 50 show a part of the code to generate training data using Microsoft PSI, and the full code is in appendix

1.1. Figure 51 shows the method that would be added to ProxemicUI to generate the data.



```
private void TestProximityEvents()
{
    using (var pipeline = Pipeline.Create())
    {
        // Register an event handler to be notified when the pipeline completes
        pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

        // generate data stream form OSC messages
        IProducer<OscMessage> message = Generators.Return(pipeline, oscMessage);


        // use process operator to process messages and test distance
        var output = message.Process<OscMessage, string>(
            (oscMessage, e, o) =>
            {
                // process OSC messages
                int count = oscMessage.Data.Count;

                // to store all entities
                List<Entity> entities = new List<Entity>();

                string timestamp = "";
                // store data of entities, the received format is (timestamp,
                serial number, x, y, z, yaw, pitch, roll)

                if (message.Address == "/trackers/")
                {
                    for (int i = 1; i < count; i++)
                    {
                        Entity newEntity = new Entity();
                        newEntity.timestamp = (string)message.Data[0];
                        newEntity.ID = (string)message.Data[i];
                        newEntity.X = Convert.ToDouble(message.Data[i+1]);
                        newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                        newEntity.Z = Convert.ToDouble(message.Data[i+3]);
                        newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                        newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                        newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                        i += 6;
                        entities.Add(newEntity);
                    }
                }

                // walk through the receved data to test distance
                if (entities.Count > 1)
                {
                    foreach(Entity firstEntity in entities)
                    {
                        foreach(Entity secondEntity in entities)
                        {
                            if (firstEntity.ID != secondEntity.ID)
                            {
                                // call DistanceCalculator method to check if
                                entities with in distance
                                bool inDistance = DistanceCalculator(firstEntity
                                , secondEntity);

                                // check the result of the DistanceCalculator
                                method
                                if (inDistance)
                                {
                                    // post data to the output stream
                                    dataStream = Generators.Return(pipeline,
                                    "Pass" + "Relative Distance Test" +
                                    firstEntity.ID.ToString() + ", " +
                                    secondEntity.ID.ToString() + ", " + Distance
                                    .ToString());
                                }
```

FIGURE 49: PART 1 OF GENERATING THE TRAINING DATA USING MICROSOFT PSI

```
                                        else
                                        {
                                            // post data to the output stream
                                            dataStream = Generators.Return(pipeline, "Fail" + "Relative Distance Test" + firstEntity.ID.ToString() + ", " + secondEntity.ID.ToString() + ", " + Distance.ToString());
                                        }
                                        // call DataWriter method to write the data to a file to create the dataset
                                        DataWriter();
                                    }
                                }
                            }
                        }
                    });
                pipeline.Run();
            }
        }

        // create dataset
        private void DataWriter()
        {
            using (var pipeline = Pipeline.Create())
            {
                // create a store to write the data to
                var store = PsiStore.Create(pipeline, "data", "D:\\code review user study\\PSI TEST CODE");

                dataStream.Write("dataSet", store);

                pipeline.Run();
            }
        }

        private bool DistanceCalculator(Entity FirstEntity, Entity SecondEntity)
        {
            // calculate positions difference
            double distance = Math.Sqrt(Math.Pow(FirstEntity.X - SecondEntity.X, 2) + Math.Pow(FirstEntity.Y - SecondEntity.Y, 2));

            if (distance >= minimumThreshold && distance <= maximumThreshold)
                return true;
            else
                return false;
        }
```

FIGURE 51: PART 2 OF GENERATING THE TRAINING DATA USING MICROSOFT PSI

```

// add loger method that can be called from any rle when it is created to loge rule and its detailes
public void RuleLogger(Rules rule, string Result, string FirstEntity, string SecondEntity, double Distance)
    {
        using (StreamWriter writer = new StreamWriter("D:\\Dataset.txt"))
        {
            writer.WriteLine(Result + ", " + rule.ToString() + ", " + FirstEntity + ", " + SecondEntity + ", " + Distance.ToString());
        }
    }

```

FIGURE 50: A METHOD TO BE ADDED TO PROXEMICUI TO GENERATE THE TRAINING DATA

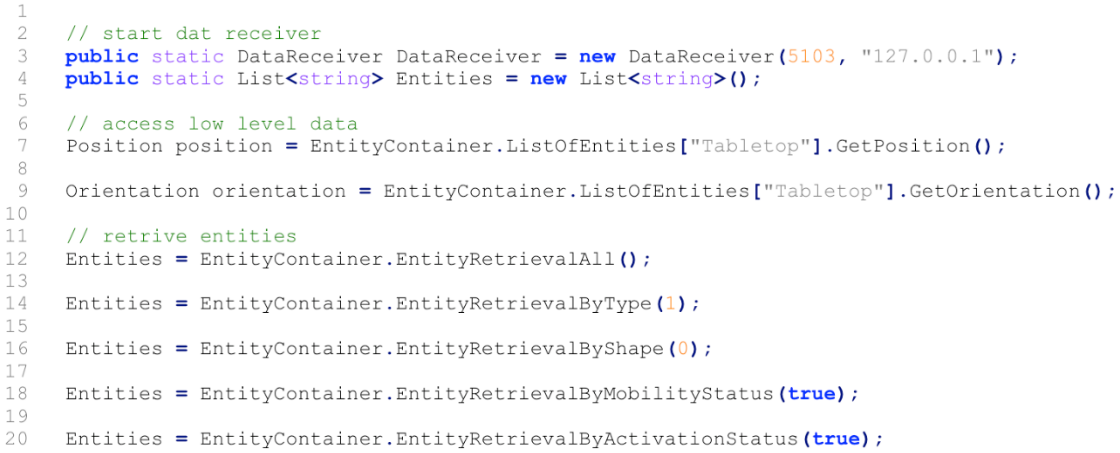
Providing the machine learning classifier with low-level proximity data

For this point of comparison, I assume that the machine learning classifier is already trained, and it is time to use the model for prediction. ProxemicUI, at its base functions, plays as an entity tracker, as it can collect entities' data from one or more tracking systems into a single object for each entity with its data. As part of the discussion, I am considering how an out-of-the-box system like Microsoft PSI can provide the classifier with low-level proximity data (e.g., ID, position, and orientation) compared to the ProxemicUI framework. Figure 52 shows a part of the code that feeds the classifier the low-level

```
        private void oscServer_MessageReceived(object sender,
        OscMessageReceivedEventArgs e)
        {
            using (var pipeline = Pipeline.create())
            {
                // Register an event handler to be notified when the pipeline completes
                pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

                // generate data stream form OSC messages
                IProducer<OscMessage> message = Generators.Return(pipeline, e.Message);

                // use process operator to process messages and test distance
                var output = message.Process<OscMessage, Dictionary<string, List<double
                >>>(
                    (oscMessage, e, o) =>
                    {
                        // process OSC messages
                        int count = oscMessage.Data.Count;

                        List<Entity> entities = new List<Entity>();

                        // store data of entities, the received format is (timestamp,
                        serial number, x, y, z, yaw, pitch, roll)
                        if (message.Address == "/trackers/")
                        {
                            for (int i = 1; i < count; i++)
                            {
                                Entity newEntity = new Entity();
                                newEntity.timestamp = (string)message.Data[0];
                                newEntity.ID = (string)message.Data[i];
                                newEntity.X = Convert.ToDouble(message.Data[i+1]);
                                newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                                newEntity.Z = Convert.ToDouble(message.Data[i+3]);
                                newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                                newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                                newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                                i += 6;
                                entities.Add(newEntity);
                            }
                        }
                        dataStream = Generators.Return(pipeline, entities);
                    }
                pipeline.Run();
            }
        }
```

FIGURE 52: FEEDING THE CLASSIFIER WITH LOW-LEVEL PROXIMITY DATA USING MICROSOFT PSI

proximity data, the full code in appendix 1.2. Figure 53 shows how to access the low-level proximity data using ProxemicUI.

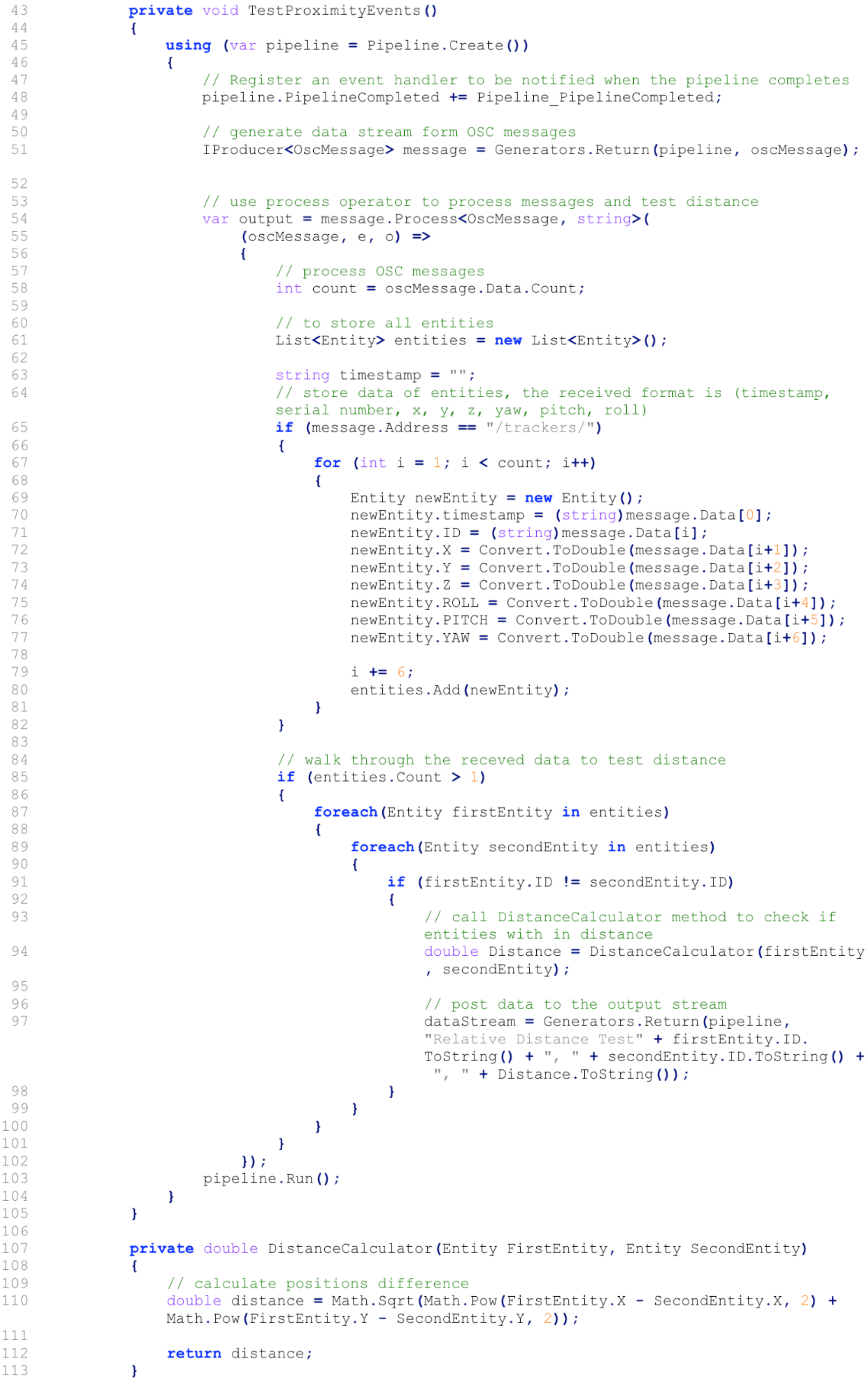


```

// start dat receiver
public static DataReceiver DataReceiver = new DataReceiver(5103, "127.0.0.1");
public static List<string> Entities = new List<string>();

// access low level data
Position position = EntityContainer.ListOfEntities["Tabletop"].GetPosition();

Orientation orientation = EntityContainer.ListOfEntities["Tabletop"].GetOrientation();

// retrive entities
Entities = EntityContainer.EntityRetrievalAll();

Entities = EntityContainer.EntityRetrievalByType(1);

Entities = EntityContainer.EntityRetrievalByShape(0);

Entities = EntityContainer.EntityRetrievalByMobilityStatus(true);

Entities = EntityContainer.EntityRetrievalByActivationStatus(true);
```

FIGURE 53: ACCESSING THE LOW-LEVEL PROXIMITY DATA USING PROXEMICUI

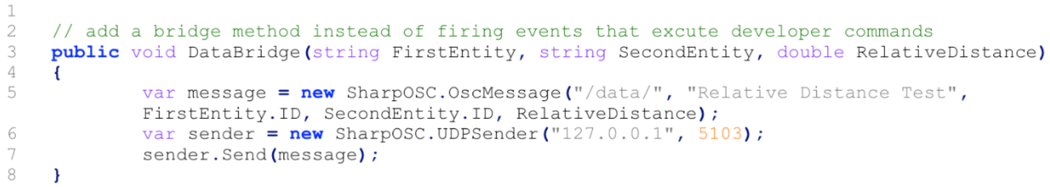
Providing the machine learning classifier with high-level proximity data

For this point of comparison, I also assume that the machine learning classifier is already trained, and it is time to use the model for prediction. I am comparing using ProxemicUI and Microsoft Psi to provide the classifier with high-level proximity data (e.g., the relative distance between two entities). Each toolkit receives the low-level proximity data and calculates the high-level proximity data. Then, the high-level proximity data will be forwarded to the machine learning classifier to start prediction. Figure 54 shows part of the code on how to generate the high-level proximity data forward them to the classifier, and the full code is in appendix 1.3. Figure 55 shows the method to be added to ProxemicUI to forward the data to the classifier.



```
private void TestProximityEvents()
{
    using (var pipeline = Pipeline.Create())
    {
        // Register an event handler to be notified when the pipeline completes
        pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

        // generate data stream form OSC messages
        IProducer<OscMessage> message = Generators.Return(pipeline, oscMessage);

        // use process operator to process messages and test distance
        var output = message.Process<OscMessage, string>(
            (oscMessage, e, o) =>
            {
                // process OSC messages
                int count = oscMessage.Data.Count;

                // to store all entities
                List<Entity> entities = new List<Entity>();

                string timestamp = "";
                // store data of entities, the received format is (timestamp,
                serial number, x, y, z, yaw, pitch, roll)
                if (message.Address == "/trackers/")
                {
                    for (int i = 1; i < count; i++)
                    {
                        Entity newEntity = new Entity();
                        newEntity.timestamp = (string)message.Data[0];
                        newEntity.ID = (string)message.Data[i];
                        newEntity.X = Convert.ToDouble(message.Data[i+1]);
                        newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                        newEntity.Z = Convert.ToDouble(message.Data[i+3]);
                        newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                        newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                        newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                        i += 6;
                        entities.Add(newEntity);
                    }
                }

                // walk through the receved data to test distance
                if (entities.Count > 1)
                {
                    foreach(Entity firstEntity in entities)
                    {
                        foreach(Entity secondEntity in entities)
                        {
                            if (firstEntity.ID != secondEntity.ID)
                            {
                                // call DistanceCalculator method to check if
                                entities with in distance
                                double Distance = DistanceCalculator(firstEntity
                                , secondEntity);

                                // post data to the output stream
                                dataStream = Generators.Return(pipeline,
                                "Relative Distance Test" + firstEntity.ID.
                                ToString() + ", " + secondEntity.ID.ToString() +
                                 ", " + Distance.ToString());
                            }
                        }
                    }
                }
            });
        pipeline.Run();
    }
}

private double DistanceCalculator(Entity FirstEntity, Entity SecondEntity)
{
    // calculate positions difference
    double distance = Math.Sqrt(Math.Pow(FirstEntity.X - SecondEntity.X, 2) +
    Math.Pow(FirstEntity.Y - SecondEntity.Y, 2));

    return distance;
}
```

FIGURE 54: GENERATING THE HIGH-LEVEL PROXIMITY DATA AND FORWARD THEM TO THE CLASSIFIER USING MICROSOFT PSI

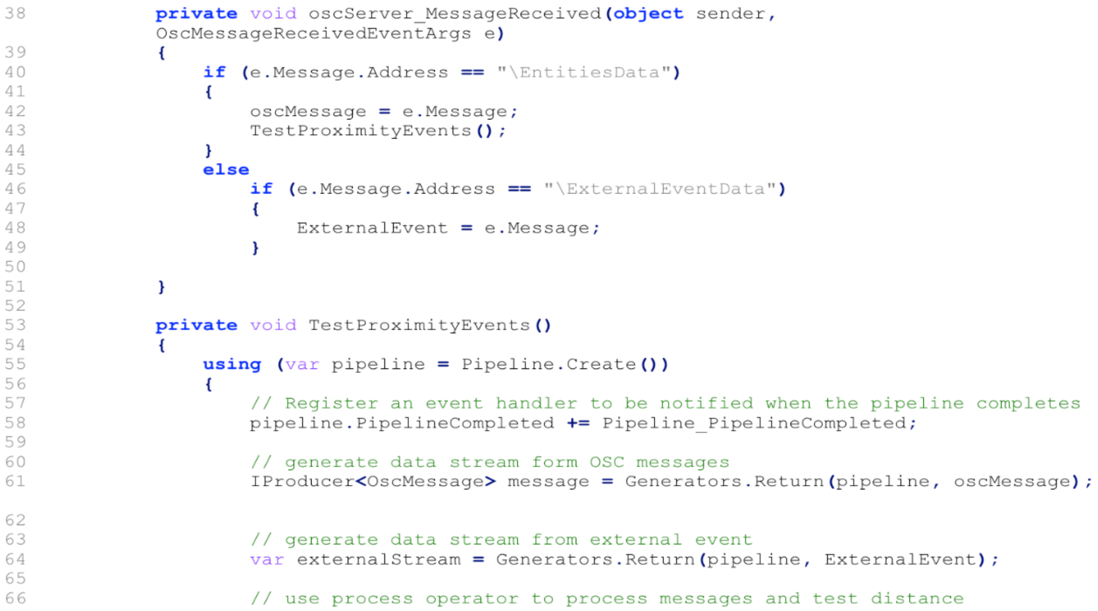


```
// add a bridge method instead of firing events that excute developer commands
public void DataBridge(string FirstEntity, string SecondEntity, double RelativeDistance)
{
        var message = new SharpOSC.OscMessage("/data/", "Relative Distance Test",
        FirstEntity.ID, SecondEntity.ID, RelativeDistance);
        var sender = new SharpOSC.UDPSender("127.0.0.1", 5103);
        sender.Send(message);
}
```

FIGURE 55: A METHOD TO BE ADDED TO PROXEMICUI TO FORWARD THE HIGH-LEVEL PROXIMITY DATA TO THE CLASSIFIER

Hybrid testing

Hybrid testing integrates any external event (e.g., UI events and system notifications) with a proximity test (basic or compound proximity test). While this combination is not related to machine learning, in this point of comparison, I am looking at the two different mechanisms that each toolkit follows to achieve this combination. I am also looking for participants point of view on which toolkit achieve this combination better. Figures 56 and 57 show part of the code on how Microsoft PSI performs the test, the full code in appendix 1.4. Figure 58 shows how ProxemicUI performs the test.

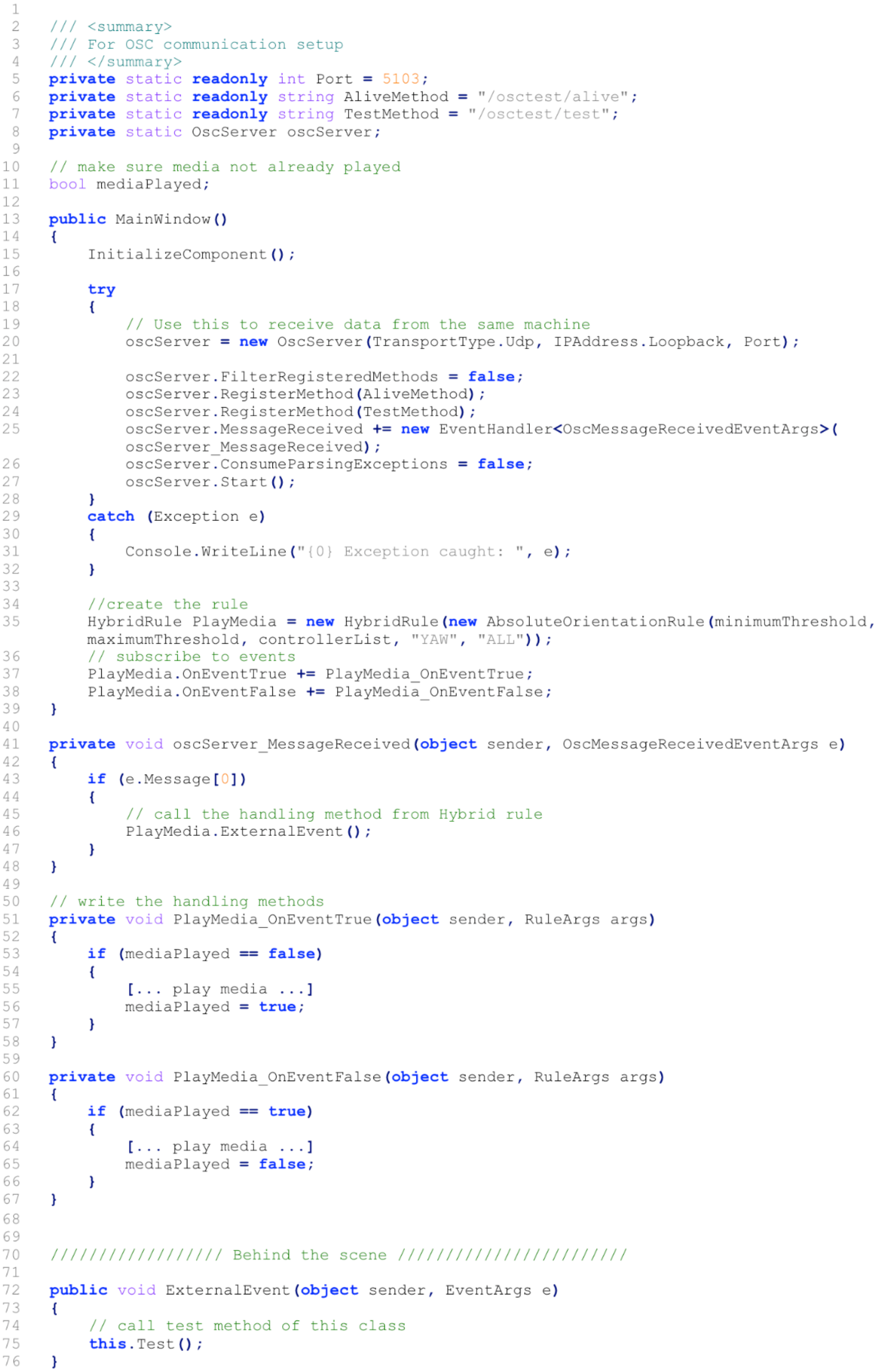


```
private void oscServer_MessageReceived(object sender,
OscMessageReceivedEventArgs e)
{
    if (e.Message.Address == "\EntitiesData")
    {
        oscMessage = e.Message;
        TestProximityEvents();
    }
    else
        if (e.Message.Address == "\ExternalEventData")
        {
            ExternalEvent = e.Message;
        }

}

private void TestProximityEvents()
{
    using (var pipeline = Pipeline.Create())
    {
        // Register an event handler to be notified when the pipeline completes
        pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

        // generate data stream form OSC messages
        IProducer<OscMessage> message = Generators.Return(pipeline, oscMessage);

        // generate data stream from external event
        var externalStream = Generators.Return(pipeline, ExternalEvent);

        // use process operator to process messages and test distance
```

FIGURE 56: PART 1 OF MICROSOFT PSI CODE TO COMBINE TWO DATA SOURCE

```
                var output = message.Process<OscMessage, string>(
                    (oscMessage, e, o) =>
                    {
                        // process OSC messages
                        int count = oscMessage.Data.Count;

                        // to store all entities
                        List<Entity> entities = new List<Entity>();

                        string timestamp = "";
                        // store data of entities, the received format is (timestamp,
                        serial number, x, y, z, yaw, pitch, roll)
                        if (message.Address == "/trackers/")
                        {
                            for (int i = 1; i < count; i++)
                            {
                                Entity newEntity = new Entity();
                                newEntity.timestamp = (string)message.Data[0];
                                newEntity.ID = (string)message.Data[i];
                                newEntity.X = Convert.ToDouble(message.Data[i+1]);
                                newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                                newEntity.Z = Convert.ToDouble(message.Data[i+3]);
                                newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                                newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                                newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                                i += 6;
                                entities.Add(newEntity);
                            }
                        }

                        // walk through the receved data to test distance
                        if (entities.Count > 1)
                        {
                            foreach(Entity firstEntity in entities)
                            {
                                foreach(Entity secondEntity in entities)
                                {
                                    if (firstEntity.ID != secondEntity.ID)
                                    {
                                        // call DistanceCalculator method to check if
                                        entities with in distance
                                        bool inDistance = DistanceCalculator(firstEntity
                                        , secondEntity);

                                        // check if both strems (events) are true
                                        var outputStream = inDistance.Join(
                                        externalStream, Reproducible.Nearest(
                                        RelativeTimeInterval.Past()),
                                        (p, s) => { if (primary && secondary)
                                        DrawResponse(); });
                                    }
                                }
                            }
                        }
                    });
                pipeline.Run();
            }
        }

        private void DrawResponse()
        {
            .... Do Something ....
        }

        private bool DistanceCalculator(Entity FirstEntity, Entity SecondEntity)
        {
            // calculate positions difference
            double distance = Math.Sqrt(Math.Pow(FirstEntity.X - SecondEntity.X, 2) +
             Math.Pow(FirstEntity.Y - SecondEntity.Y, 2));

             if (distance >= minimumThreshold && distance <= maximumThreshold)
                 return true;
             else
                 return false;
         }
```

FIGURE 57: PART 2 OF MICROSOFT PSI CODE TO COMBINE TWO DATA SOURCE



```
/// <summary>
/// For OSC communication setup
/// </summary>
private static readonly int Port = 5103;
private static readonly string AliveMethod = "/osctest/alive";
private static readonly string TestMethod = "/osctest/test";
private static OscServer oscServer;

// make sure media not already played
bool mediaPlayed;

public MainWindow()
{
    InitializeComponent();

    try
    {
        // Use this to receive data from the same machine
        oscServer = new OscServer(TransportType.Udp, IPAddress.Loopback, Port);

        oscServer.FilterRegisteredMethods = false;
        oscServer.RegisterMethod(AliveMethod);
        oscServer.RegisterMethod(TestMethod);
        oscServer.MessageReceived += new EventHandler<OscMessageReceivedEventArgs>(
        oscServer_MessageReceived);
        oscServer.ConsumeParsingExceptions = false;
        oscServer.Start();
    }
    catch (Exception e)
    {
        Console.WriteLine("{0} Exception caught: ", e);
    }

    //create the rule
    HybridRule PlayMedia = new HybridRule(new AbsoluteOrientationRule(minimumThreshold,
    maximumThreshold, controllerList, "YAW", "ALL"));
    // subscribe to events
    PlayMedia.OnEventTrue += PlayMedia_OnEventTrue;
    PlayMedia.OnEventFalse += PlayMedia_OnEventFalse;
}

private void oscServer_MessageReceived(object sender, OscMessageReceivedEventArgs e)
{
    if (e.Message[0])
    {
        // call the handling method from Hybrid rule
        PlayMedia.ExternalEvent();
    }
}

// write the handling methods
private void PlayMedia_OnEventTrue(object sender, RuleArgs args)
{
    if (mediaPlayed == false)
    {
        [... play media ...]
        mediaPlayed = true;
    }
}

private void PlayMedia_OnEventFalse(object sender, RuleArgs args)
{
    if (mediaPlayed == true)
    {
        [... play media ...]
        mediaPlayed = false;
    }
}


///////////////// Behind the scene //////////////////////

public void ExternalEvent(object sender, EventArgs e)
{
    // call test method of this class
    this.Test();
}
```

FIGURE 58: PROXEMICUI CODE TO COMBINE TWO DATA SOURCE

Connecting with tracking systems

ProxemicUI has a setup for OSC communications protocol, making it able to connect with any tracking system. On the other hand, Microsoft PSI has several components to capture various types of data (e.g., Kinect component and speech recognition component). However, to connect with unsupported systems (e.g., VIVE trackers), developers would set up the connection to receive the data. This difference was shown to participants throughout the comparison when I showed the code metrics. Therefore, I asked them which approach they prefer: a generic approach such as the OSC communications protocol setup in ProxemicUI or a built-in component approach such as the Kinect component in Microsoft PSI. During this discussion, I made it clear that developers might still need to write a communicator code with the generic approach to capture the data from the tracking system and pass them to ProxemicUI.

Data collection

During all the code review sessions, screen recordings of explaining the codes and participants' discussions were collected using Microsoft Teams.

Analysis method

Qualitative research can be defined as using participant's textual and verbal input as data to answer the research questions [20]. Thematic analysis is a common method of qualitative research analysis methods [21][22], which Braun and Clarke define as "identifying, analyzing, and reporting patterns (themes) within data"[22]. These themes represent important aspects of the data that helps to answer the research questions.

Identifying themes consists of several steps, including getting familiar with the dataset, identifying initial codes, generating themes based on the initial codes, reviewing and revising the generated themes, finalizing the themes, and writing the report [21][22][77]. Inductive thematic analysis is one approach to defining themes in qualitative research, which is considered a “data-driven” approach. The “data-driven” approach means that there is a direct connection between the identified themes and the questions asked during the data collection phase [21][22]. My analysis follows the inductive thematic approach, where I started by transcribing all video recordings from all sessions. Microsoft Teams, which I used for recording, saved all recordings automatically to Microsoft Stream. Microsoft Stream provides an option to auto-transcribe video, which I used as a starting point for the transcription. Then I walked through the transcription file to compare and correct the results with videos to ensure all data is correct. Then, I identified eight codes according to the discussion questions that I asked during the sessions, and then I divided the transcription accordingly. I used the question as codes because they represent the eight tasks (discussed earlier) that we are trying to accomplish using each toolkit. Upon reviewing the dataset in relation to the eight codes, I identified two themes for ProxemicUI vs. the Proximity Toolkit (flexibility vs. effort) and two themes for ProxemicUI vs. Microsoft PSI (processing the proximity data in the background vs. at the developer’s side). By identifying these themes, I was able to understand participants' opinions about each point of comparison and the reasons behind these opinions. For example, while most participants preferred to use ProxemicUI to test a single proximity attribute between more than two entities over The Proximity Toolkit, each group discussed different reasons, with partial overlap. Finally, I reported the data for each theme including each opinion, how many participants agreed upon it, and listed all the reasons given for the opinion.

Besides the thematic analysis, I produced some code metrics for the code samples that were shown to participants during the code review study, as we discussed the related code sample. Code metrics can be defined as "a set of software measures that provide developers better insight into the code they are developing"[23]. Metrics such as maintainability index and cyclomatic complexity allow developers to improve the quality of their code by identifying parts of the code that may need to be re-designed. The basic code metrics that I generated from the code samples aim to give participants an approximate sense of the amount of work required to perform a specific task using each tool (in terms of lines of code to write, number of objects to instantiate and manage, number of callback methods that need to be created, and number of methods to recall). We acknowledge that such metrics say little about the complexity of the logic within each line of code.

Results

The last phase of conducting a thematic analysis is "producing the report" of the analyzed data [21][22], including reporting the identified themes and the evidence to support these themes. The results consist of two main sections, one for each comparison (ProxemicUI vs. the Proximity Toolkit and ProxemicUI vs. Microsoft PSI). each of these sections has several subsections that represent the tasks we were trying to accomplish in the code samples using each toolkit. Within each subsection, I will discuss the two themes (including the evidence to support each theme) I identified for each comparison, ProxemicUI vs. the Proximity Toolkit (flexibility vs. effort) and ProxemicUI vs. Microsoft PSI (processing the proximity data in the background vs. at developer's side). Throughout the results sections, there are a few incorrect statements that participants reported during the discussion, which I stated as reported. Each incorrect will be underlined with a small number at the end of the statement. Then, in Table 11, I listed all of the incorrect statements and the reasons why they are incorrect.

TABLE 6: NUMBER OF PARTICIPANTS WHO PREFERRED EACH TOOLKIT ACCORDING TO DIFFERENT KEY QUESTIONS

Questions	Proximity Toolkit	ProxemicUI	No answer
Which toolkit is more sufficient to test a single proximity attribute between two entities	5	13	N/A
Which toolkit is more sufficient to test a single proximity attribute between more than two entities	1 one said it is good for beginners	16 one said it is good for experience users	N/A
Which toolkit is more sufficient to test a multiple proximity attributes	N/A	17	1
Which toolkit is more sufficient to combine testing proximity attributes with external events	N/A	18	N/A

ProxemicUI vs. The Proximity Toolkit

Testing a single proximity attribute between two entities

The main point of participants' input can be abstracted to flexibility vs. effort. The majority of the participants (13 out of 18) preferred ProxemicUI and think that it minimizes developers' effort more. First, it has a specific and intuitive approach to handle the test. Second, it does the tests in the background, which minimizes coding errors. Third, it is easier to make changes to a rule (in ProxemicUI) than a method (in The Proximity Toolkit). Two participants think that both toolkits are equal in minimizing the effort. Another participant thinks minimizing the effort is different between developers as each toolkit follows a different approach.

On the other hand, five participants preferred the Proximity Toolkit and think it is more flexible. First, it allows you to access the proximity data (e.g., the value of relative distance), then developers will check if it is within thresholds or not[1]. With the Proximity Toolkit, developers can see what the handling methods are testing, enabling them to make changes if they wish. Two participants think that this flexibility allows them to have multiple tests in the same handling method if they have multiple thresholds. They also think they would need to create multiple rules with ProxemicUI if they have multiple thresholds[2]. Only one participant thinks that flexibility to access the low-level programming tasks (calculating proximity attributes and checking if they are within thresholds) is more important than reducing the effort.

TABLE 7: CODE METRICS TO TEST A SINGLE PROXIMITY ATTRIBUTE BETWEEN TWO ENTITIES

Code Metrics	The Proximity Toolkit	ProxemicUI
Lines of code	19	14
Objects created	3	1
Methods created	1	2
Methods called	1	2

Testing a single proximity attribute between more than two entities

The majority of the participants (16 out of 18) preferred the approach that ProxemicUI follows, and one participant liked the approach that The Proximity Toolkit follows. The last participant thinks that ProxemicUI is better for experienced developers and The Proximity Toolkit is better for beginners. The reason behind this is that she thinks beginners should start with long and detailed code to understand the process they are doing; then, when they get comfortable, they can move to use the abstracted code. In contrast, another participant who preferred ProxemicUI thinks that beginners should start with an abstracted system (e.g., ProxemicUI) because it is easier for them than dealing with low-level codes and detailed tests.

Sixteen out of eighteen participants think that ProxemicUI minimizes developers' effort for many reasons. First, ProxemicUI abstracts most of the stuff that developers need to do into a single rule (e.g., testing multiple entities and having the test condition "ALL" or "ANY"), making it more scalable than The Proximity Toolkit. Participants also think that the complexity of the code will increase as the number of devices increases, making the use of The Proximity Toolkit good with a small number of devices. P2 said with ProxemicUI, "you don't need to think much about scaling, but in The Proximity Toolkit, I would create another class to manage relations" [P2]. P3 said, "ProxemicUI can do all the functions that The Proximity Toolkit does in a simpler way, and it can do more". Two participants think that ProxemicUI limits you to its functions[3], making The Proximity Toolkit more flexible in terms of testing something that is not already in ProxemicUI. One of these participants thinks that The Proximity Toolkit would be suitable for research where you can access all details, where ProxemicUI is good if you know what you need to do and trying to create a final product.

TABLE 8: CODE METRICS TO TEST A SINGLE PROXIMITY ATTRIBUTE BETWEEN MORE THAN TWO ENTITIES

Code Metrics	The Proximity Toolkit	ProxemicUI
Lines of code	40-…	31
Objects created	6-…	3
Methods created	2	3
Methods called	2	3

Compound testing

Seventeen out of eighteen participants preferred the use of ProxemicUI. First, it would reduce/avoid coding errors and is easy to go back and check the code if something went wrong as everything is in one line (e.g., in the ANDRule). P3 said, "ProxemicUI has a more natural way of dealing with these kinds of things by using these rules". Besides, using rules and thresholds in ProxemicUI relieve developers from getting into low-level programming details. Two participants think that the rule approach might end up in problems if a scenario does not have the rule to test it. One participant thinks that combining two compound rules makes ProxemicUI more flexible and sufficient to use. On the other hand, one participant would use The Proximity Toolkit when dealing with complex scenarios because it is more flexible and allowing more control. Even though she believes that The Proximity Toolkit will increase the burden for developers a bit, she would still use it to have more control. However, she also believes that the rule approach is easier to use.

Another participant thinks that the flexibility of The Proximity Toolkit allows doing checks beyond distance and orientation, which ProxemicUI covers with Hybrid Rule.

TABLE 9: CODE METRICS TO TEST MULTIPLE PROXIMITY ATTRIBUTES

Code Metrics	The Proximity Toolkit	ProxemicUI
Lines of code	46	23
Objects created	3	1
Methods created	2	2
Methods called	2	2

Orientation tests

One of the interesting points that we looked at while discussing the compound test above is how each toolkit tests orientation. The Proximity Toolkit provides developers with two types of values regarding the relative orientation (e.g., double for the angle between two entities and true/false if A is facing B); more details are in [71]. In both cases, developers would check these values at their end to draw system responses accordingly. On the other hand, ProxemicUI receives a threshold (in degrees) then checks if entities are facing each other within this threshold or not. Participants appointed the advantages of each approach: the threshold in ProxemicUI makes it more flexible in dealing with, where The Proximity Toolkit might provide more accurate measure if needed (e.g., if A is facing B or angle between two entities). Three participants think that ProxemicUI can incorporate the orientation features in The Proximity Toolkit to implement proxemics-aware applications even easier. One of those participants stated that having the threshold requires thinking about how to implement the function. Another two participants suggest using The Proximity Toolkit with fewer entities and ProxemicUI with more entities.

Hybrid Test

One participant thinks that ProxemicUI is clearer and easier than The Proximity Toolkit, but its logic became more complex. The rest of the participants (17 out of 18) preferred ProxemicUI. First, with ProxemicUI, there is no need to check when both events occurred; the proximity rule will be tested automatically when the external event has occurred, which reduces the complexity of the code. ProxemicUI allows defining extended rules inside the “HybridRule” (e.g., CompoundRule). ProxemicUI abstracts what The Proximity Toolkit does, which provides a straightforward and well-structured code. One participant suggested that we can use The Proximity Toolkit to set up, but when everything is working perfectly, we can use ProxemicUI. She also suggested that The Proximity Toolkit is suitable for beginners to see all the steps, and ProxemicUI is ideal for advanced developers. In contrast, another participant thinks ProxemicUI would be ideal for beginners because it is easy to use and less prone to errors. One participant pointed a disadvantage of the rule approach: developers would need to know what they are doing before they start because they can not figure things out on the way. However, she also thinks that it will be a massive advantage if developers know how the system works. Lastly, one participant suggested that if there is a way to override the “ExternalEvent” method to allow developers to more things when the external event has occurred.

TABLE 10: CODE METRICS TO COMBINE EXTERNAL EVENTS WITH PROXIMITY EVENTS

Code Metrics	The Proximity Toolkit	ProxemicUI
Lines of code	37	24
Objects created	4	2
Methods created	2	2
Methods called	2	2

Processing proximity data internally vs. at the developer's side

Flexibility is one of the main points discussed during the sessions (e.g., getting the value and doing the check with The Proximity Toolkit vs. having the final results with ProxemicUI). Therefore, I asked participants if they prefer to do the check themselves or for the system to do it for them. Seventeen out of eighteen participants preferred the ProxemicUI's approach, where the system does the test and provides the final results. They liked ProxemicUI for many reasons. First, developers will focus on the system response they want to draw instead of wasting their time dealing with low-level details. Second, modifying rules is easier and straightforward than modifying methods. On the other hand, only one participant preferred The Proximity Toolkit's approach with no given reason behind his choice.

TABLE 11: LIST OF INCORRECT STATEMENTS THAT PARTICIPANTS REPORTED AND THE REASON WHEY THEY ARE NOT CORRECT

Note	Statement	Why it is not correct
1	ProxemicUI is not flexible because developer cannot access the values of high-level proximity data	The event object in ProxemicUI can be used to retrieve the values of the high-level proximity data
2	With ProxemicUI, developers would need to create multiple rules with ProxemicUI if they have multiple thresholds	Participants stated this because they were not exposed to the CompoundRules et
3	ProxemicUI limits you to its functions, making The Proximity Toolkit more flexible in terms of testing something that is not already in ProxemicUI	One of the extension points in ProxemicUI is the ability to define/create new set rules, which allows developers to add more functionality to ProxemicUI.

ProxemicUI vs. Microsoft PSI

For this comparison, I considered using the VIVE trackers as a source for proximity data because we can track any entities (humans or objects) by attaching a tracker to them. To use the VIVE trackers, I wrote a communicator code that uses OSC communication protocol as a bridge to transfer proximity data from the tracking system to both Microsoft PSI and ProxemicUI. Because ProxemicUI already has the setup to connect with any tracking system through OSC, there is no need to do the OSC setup at the developer's side. On the other hand, because Microsoft PSI does not support the OSC communication, we need to add the OSC setup at the developer's side. However, when I showed the code segments to participants, we also discussed how much effort to use one of PSI's components (e.g., Microsoft Kinect component) to receive the tracking data. The OSC setup for Microsoft PSI requires defining and starting the OSC server; writing the OSC handling method that the system will call when it receives a message; breaking the messages down to define different attributes before start testing proximity data. I will discuss how this affects the code in different sections below. Before I go into more details, I want to make it clear that I compared both toolkits in the context of proxemic awareness use, and the results might not be applicable in other contexts. For example, Microsoft PSI is a more generic framework that is not limited to support proxemic awareness. Therefore, the data reported in this section is for supporting proxemic awareness that might not be applicable for other domains.

Generating training data

The main difference here is handling proximity data in the background vs. at the developer's side. Generally, all participants preferred using ProxemicUI, but each toolkit has its pros and cons. ProxemicUI performs the same task more easily and with fewer lines of code; as P3 said: "ProxemicUI is right on the point". These tasks are done internally in ProxemicUI, including processing the OSC messages, store the proximity data, calculate high-level proximity data to generate the final results. Wherein Microsoft PSI, developers would do all of these tasks manually. However, we will have fewer tasks (no OSC setup) with Microsoft PSI if we used one of its components (e.g., Microsoft Kinect) to capture the tracking data. Even with using the Kinect, all participants think ProxemicUI would be better as developers would still need to calculate the high-level proximity data. The reason behind this difference is ProxemicUI intended to support the development of proxemic-aware applications, where Microsoft PSI is a more generic framework with no specifications for proxemic awareness support. One participant suggested that ProxemicUI can be a good tool for beginners.

Given that everything is done at the developer's side with Microsoft PSI, four participants think it would make Microsoft PSI more flexible. This flexibility allows developers to

TABLE 12: METRICS TO GENERATE TRAINING DATA

Code Metrics	Microsoft Psi (with OSC)	Microsoft Psi (without OSC)	ProxemicUI
Lines of code	158	59	7
Objects created	8- …	7	2
Methods created	4	3	0
Methods called	4	3	0

customize the data more according to their needs. One participant suggested adding this customization to ProxemicUI by passing the high-level proximity data's value and not only the final pass or fail. This flexibility also allows them to add non-proxemic-related data. They also think Microsoft PSI more complex because it is more open and large scale; as P3 said: “Microsoft PSI seems a bit more robust for streaming data process especially when it gets very complicated”.

Providing the machine learning classifier with low-level proximity data

When it comes to providing low-level proximity data, there is no complex process involved; each system will receive the tracking data, process them, then forward the data according to the required format to the machine learning classifier. One participant thinks they are both equal in providing the low-level data. Sixteen participants preferred to use ProxemicUI for this task. First, there is no need to go into low-level programming tasks to process the data, which is done internally; as P8 and P9 said, “ProxemicUI abstracts a lot of things that I need to do through Microsoft PSI”. Second, the data in ProxemicUI is already processed and can be accessed through different methods based on different attributes (e.g., shape or type). Third, no need to set up the OSC connection as it is built-in; this makes ProxemicUI work with any tracking system and not limited to the component inside Microsoft PSI. However, only one participant preferred to use

TABLE 13: CODE METRICS TO FEED THE CLASSIFIER LOW-LEVEL PROXIMITY DATA

Code Metrics	Microsoft Psi	ProxemicUI
Lines of code	112	5-9
Objects created	7- …	4
Methods created	1	0
Methods called	1	1-5

Microsoft PSI to provide low-level data because it is more flexible to add data other than what is supported by ProxemicUI.

Providing the machine learning classifier with high-level proximity data

For this task, we are not only processing the incoming data but also calculating the high-level proximity data, which will expand as the number of proximity attributes and thresholds test increases. One participant did not provide any input in this regard, but seventeen participants preferred to use ProxemicUI for this task. First, ProxemicUI abstracts a lot of what Microsoft PSI is doing (e.g., calculating the proximity attributes), which is less prone to coding errors. Second, ProxemicUI is specific to what we are looking for (proxemics data), and it is built for that. One participant thinks ProxemicUI is a good tool for beginner and intermediate developers, and Microsoft PSI can be helpful for more advanced developers. However, one participant suggested that Microsoft PSI might be useful in some situations when there is a need to add a final level of tuning to the data. Another participant suggested that calculating the high-level proximity data at the developer side adds more flexibility to a different type of testing (e.g., calculate the distance based on an area, not minimum and maximum thresholds). Two participants

TABLE 14: CODE METRICS TO FEED THE CLASSIFIER HIGH-LEVEL PROXIMITY DATA

Code Metrics	Microsoft Psi (with OSC)	Microsoft Psi (without OSC)	ProxemicUI
Lines of code	138	39	6
Objects created	7- …	6	2
Methods created	3	2	0
Methods called	3	2	0

think that overall, Microsoft PSI is more flexible as developers can access all low-level programming details.

TABLE 15: CODE METRICS TO COMBINE EXTERNAL EVENTS WITH PROXIMITY EVENTS

Code Metrics	Microsoft Psi (with OSC)	Microsoft Psi (without OSC)	ProxemicUI
Lines of code	155	74	49
Objects created	10- …	8	2
Methods created	4	4	3
Methods called	4	4	3

Hybrid testing

This type of testing aims to provide developers with a straightforward feature to simultaneously draw system responses based on proximity events and external events. Fifteen participants preferred using the *HybridRule* in ProxemicUI over the *Join* operator in Microsoft PSI. First, with ProxemicUI, there is no need to initialize the two streams, which might be confusing to deal with, especially when joining them with the *Join* operator. *HybridRule* relieves developers from writing different methods to calculate proximity attributes, making ProxemicUI less prone to errors. In addition, with the rule approach, developers would need to pass the arguments only with no need to worry about processing data streams, making the rules easy, straightforward, and more sufficient in most cases. One participant believes that, with Microsoft PSI, developers would implement more methods as the number of proximity attributes that we need to test increases; therefore, the code size will grow as developers would need to write more formulas, resulting in increasing the execution time. One participant understands that it will be a hassle to calculate different proximity attributes using Microsoft PSI, but he still prefers to use it as it is more readable and understandable. Two participants have no preference as to which toolkit to use.

Connecting with tracking systems

ProxemicUI has a setup for OSC communications protocol, making it able to connect with any tracking system. On the other hand, Microsoft PSI has several components to capture various types of data (e.g., Kinect component and speech recognition component). However, to connect with unsupported systems (e.g., VIVE trackers), developers would set up the connection to receive the data. I discussed this point with participants, and the majority of participants (14 out of 18) preferred the broader approach that ProxemicUI takes, which does not limit developers to use a specific system. However, one participant thinks that it all comes to what type of data the OSC messages support and if developers can transfer their data using OSC. The OSC allows to include any type of data in the messages. Another participant thinks that if we are building a system using Microsoft

TABLE 16: NUMBER OF PARTICIPANTS WHO PREFERRED EACH TOOLKIT ACCORDING TO DIFFERENT KEY QUESTIONS

Questions	Microsoft PSI	ProxemicUI	Equal	No answer
Which toolkit is more suitable to generate training data for a machine learning classifier	N/A	18	N/A	N/A
Which toolkit is more suitable to feed the machine learning classifier low-level proximity data	1	16	1	N/A
Which toolkit is more suitable to feed the machine learning classifier high-level proximity data	N/A	17	N/A	1
Which toolkit is more suitable to combine testing proximity attributes with external events	1	15	2	N/A

environment with Microsoft products, then Microsoft PSI will be ideal. Four other participants did not have comments on this.

Summary of the outcomes

ProxemicUI vs. The Proximity Toolkit

Overall, participants think ProxemicUI would be more suitable to support the implementation of proxemic-aware applications for many reasons. First, ProxemicUI abstracts most of the low-level programming details by performing the proximity tests in the background. Unlike The Proximity toolkit, which allows for creating one-to-one relations, ProxemicUI allows adding multiple entities into a single rule using one-to-many or many-to-many; besides, using the test condition (ALL/ANY) makes ProxemicUI more scalable. ProxemicUI also allows combining two tests (rules) into a single one (e.g., using ANDRule), where developers would check the results of one event (the ANDRule), which is easier than dealing with more details in the handling methods with The Proximity Toolkit. ProxemicUI also allows combining external events with proximity events by using HybridRule. With the HybridRule, developers can define extended rules as well (e.g., CompoundRule). Lastly, modifying rules is easier and straightforward than modifying methods.

On the other hand, The Proximity Toolkit provides some flexibility where developers can have more control over the tests. However, this flexibility would make the code more complex as the number of entities/tests increases. Unlike ProxemicUI, which provides a single threshold for relative orientation, The Proximity Toolkit provides a set of specific attributes regarding the relative orientation (e.g., true/false if A is facing B).

ProxemicUI vs. Microsoft PSI

Because ProxemicUI was built to support proxemic awareness, it is more suitable to generate training data than Microsoft PSI, which does all the tests at the developer’s side. However, few participants think that they can customize the data more with Microsoft PSI as they implement the test at the developer's side compared to the built-in method in ProxemicUI. ProxemicUI can gain this flexibility by passing the values of the tests to the developers and not only the final results (pass or fail). When combining proximity events with external events, HybridRule relieves developers from writing different methods to calculate proximity attributes, making ProxemicUI less prone to errors. In addition, with the rule approach, developers would need to pass the arguments only with no need to worry about processing data streams, making the rules easy, straightforward, and more sufficient in most cases. Finally, have an OSC protocol setup in ProxemicUI makes a broader system that can connect with any tracking system in terms of proxemic awareness use.

Required refinements to the framework

When supporting the machine learning classifier, ProxemicUI provides a built-in function to generate the training data to a file or forward the data to the classifier. However, participants suggested that it would be sufficient to pass the values back to the developers and let them customize the data as they want. Currently, Rules in ProxemicUI notify developers of the final results only (pass/fail). As a refinement to the framework, I will pass the test values to developers along with other arguments.

Limitations of code review

In this section, I will discuss the limitation of setting up and running the code review study. I was planning to generate the code segments for the Proximity Toolkit by implementing the same scenario I used for ProxemicUI (the smart home scenario). Therefore, I downloaded the source code project for the toolkit from the project website [50], built the project to generate the setup installation file, and installed the toolkit. To accomplish this, I followed the available How-To documentation on the project website, including videos and textual documentation. Then, I started the server, where I chose the tracking system (Microsoft Kinect) to load with the server start-up. After this, I am supposed to get the server setup window, but the toolkit froze at this point and did not show any error messages. I reviewed the documentation on the website again but could not find any more data regarding this issue. When I contacted the toolkit authors, they stated that the codebase does not work anymore as it depends on tracking libraries that are no longer compatible. Therefore, I switched my plan to use the source code of an existing system that uses the Proximity toolkit and is similar to the smart home setup I am using for ProxemicUI. I tried to get the source code of the Proxemic-Aware Controls [64] from the same group as a toy example to show how to use the toolkit but did not get any response. Because I was not able to get the Proximity Toolkit running and could not get the source code of an existing application that uses the Proximity Toolkit, I wrote the codes based on my understanding of the toolkit. I generated these Code segments according to the authors' documentation available in their paper [71]. The paper provides a demonstrative scenario that explains step-by-step how to implement the scenario using the toolkit. I followed these steps to generate the code that I presented to my participants.

For Microsoft PSI, I was also trying to look at how existing projects use the toolkit, as their website provides a list of projects that uses the Microsoft PSI. Unfortunately, all projects were ongoing projects, and there was no available documentation on how Microsoft PSI is being used by these projects. Therefore, I wrote the code segments based on my understanding of the available documentation in [87] and sample applications in [96]. The available documentations have steps to show how to use different operators, including code snippets. The sample applications provide how an operator is being used in the context of an application. Therefore, both code segments for Microsoft PSI and The Proximity Toolkit were generated based on my understanding of the tools. The comparison between ProxemicUI and Microsoft PSI was limited to the use of proxemic awareness, which might not be applicable to other contexts as Microsoft PSI is a more generic toolkit. Lastly, to have more accurate measures about how each toolkit performs, all toolkits should be used by external developers. Unfortunately, my participants did not have a hands-on experience due to the restriction of COVID-19, given that I planned a hackathon user study.

Discussion

One of the main goals of developing a framework or a toolkit is to add new functionalities or ease existing ones to support the development of applications in certain domains. These new functionalities should not take developers away from their main task (building the application) to dealing with low-level programing tasks that the toolkit requires to use these functionalities. While the Proximity Toolkit provides developers with some data about proximity relations between two entities where developers would do the check at their side, ProxemicUI encapsulates these tasks in rules. While the flexibility of the Proximity Toolkit is present in the handling method where developers retrieve the proximity data to process them, ProxemicUI provides flexibility in the rules. In these rules, developers can test as many entities as they wish by using different types of tests (one-to-one, one-to-many, many-to-many). Developers also can specify to apply the rule to all entities or just a subset using the test condition (ALL/ANY). Such rule flexibility is important for toolkits, especially if scalability plays an essential role in the new system. In addition, with HybridRule, developers would keep track of a single object (in this case, it is HybridRule) instead of two objects (the proximity rule and the external event). The HybridRule also allows extending rules by defining a CompoundRule inside it, which adds to the rule flexibilities in ProxemicUI.

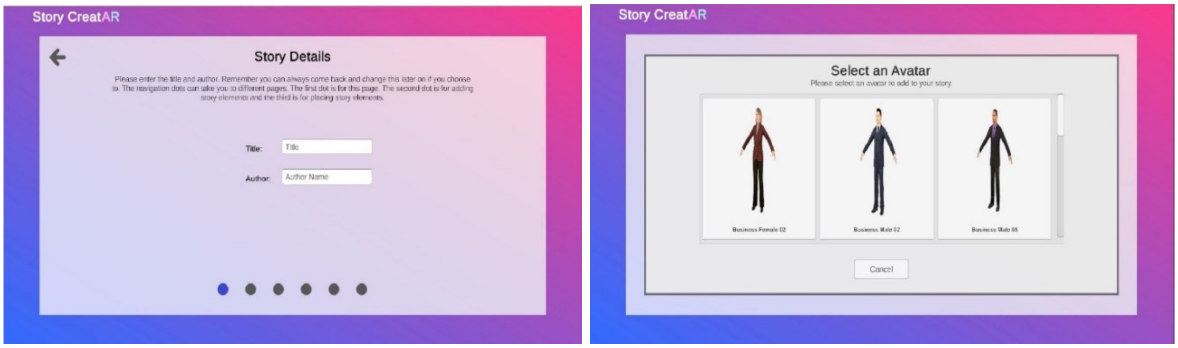
One of the main steps of building a machine learning classifier is to train it, which requires a training dataset. In many cases, there are no available datasets that can be used for training, which forces developers to collect the dataset themselves. Collecting a dataset can be a really hard task, especially if there are no resources to help with the collection. As ProxemicUI was built to support proxemic awareness, it can be more suitable to train a proxemic-aware classifier over other systems that don't support proxemic awareness (e.g., Microsoft PSI). The same concept can also be applied to feeding proximity data to the classifier during the prediction phase. However, when training a proxemic-aware classifier, it is more suitable to provide the proximity data to developers where they can format the dataset as they wish over having built-in functions that write the data. While the current implementation of ProxemicUI provides only the final results (pass/fail), which does not include the proximity data, the event objects can be used to pass these values as well.

Summary

This chapter discussed a code review study that consists of two parts. First we compared ProxemicUI with the Proximity Toolkit, to compare how each tool can support building proxemic-aware applications. Second, we compared ProxemicUI with Microsoft PSI to compare how each tool can support train/feed data to a proxemic-aware classifier. The chapter also discussed the outcomes of the code review study, including a set of refinements to ProxemicUI. For example, most participants preferred using ProxemicUI over the Proximity Toolkit to build proxemic-aware applications as it abstracts most of the low-level programming details by testing in the background and using the rules in ProxemicUI. Besides, while the Proximity toolkit provides some flexibility through passing the high-level proximity data to developers and providing a set of specific attributes regarding the orientation, ProxemicUI's event model and rule hierarchy are extensible and can be used to provide such features. In addition, while most participants preferred using ProxemicUI over Microsoft PSI to support building a proxemic-aware classifier, some participants believe that Microsoft PSI is more customizable when generating training data. However, ProxemicUI can achieve this customization using the event model to pass the proximity data to the developers. The main refinements to ProxemicUI that the code review study generated are to pass the high-level proximity data to developers using the event model and provide additional specific attributes to developers regarding the relative orientation between entities. While the code review study provided insights into how developers feel about different approaches that each tool follows, the next chapter (chapter 8) discusses the process of integrating ProxemicUI into another system.

Chapter 8: Evaluating ProxemicUI through the integration into Story CreatAR

One evaluation technique through demonstration is *Case Studies* that demonstrate what the toolkit can do [65]. This demonstration might involve using the toolkit with different projects similar to the iStuff toolkit [12]. Employing ProxemicUI with other systems allows me to show its range of use and explore additional requirements, especially when it is being used in a different context (e.g., VR/AR). Therefore, I integrated ProxemicUI into Story CreatAR [105], which is an author-facing tool to create VR/AR stories that can be deployed in different environments. This integration will provide evidence that external developers would be able to use ProxemicUI to support the implementation of their system. It will also provide evidence on how end-users might benefit from using ProxemicUI functionalities through interacting with other systems (in my case, the other system is Story CreatAR). This chapter will give a brief overview of Story CreatAR. Then it will discuss the process of evaluating ProxemicUI through its integration into Story CreatAR. This work was done with co-investigators who contributed in building the Story CreatAR, integrating ProxemicUI into Story CreatAR, and running the study with the end-users.



Story CreatAR

Story Details

Please enter the title and author. Remember you can always come back and change this later on if you choose to. The navigation dots can take you to different pages. The first dot is for this page. The second dot is for adding story elements and the third is for placing story elements.

Title: Title

Author: Author Name

Story CreatAR

Select an Avatar

Please select an avatar to add to your story.

FIGURE 59: LEFT: SPECIFYING DIFFERENT DETAILS ABOUT THE STORY, RIGHT: CHOOSING AN OBJECT TO ADD IT TO THE STORY

Story CreatAR

As stated before, Story CreatAR is an authoring tool that allows authors to create stories in AR/VR. To create the story, authors can use an existing map or upload a new map of an environment to Story CreatAR. Authors start creating their story by adding story details (title/author) and story elements, consisting of avatars, 3D audio, and other objects (e.g., table, painting, cat). In addition, there is the option to use Groups that allow authors to group story elements that will be assigned similar spatial placements, or simply to categorize story elements. For example, suppose an author wants all the protagonists to be in an open area. In that case, s/he can create a group called "Protagonists", add all the protagonists to it, and give it the spatial attribute “open area”. Using the tool, story elements and groups can be placed within an environment according to space syntax characteristics [108] and proxemics rules.

Objectives for the integration with Story CreatAR

Our primary research objective for the Story CreatAR study (the ethics application is in appendix 7 and the approval letter is in appendix 8) is to assess how external developers will use/benefit from the ProxemicUI framework when developing proxemics-aware applications. In addition, we want to assess how well ProxemicUI can be integrated into a platform used by end-users (in this case, authors of augmented and virtual reality stories). To achieve this objective, we ask the following research questions about ProxemicUI's integration into Story CreatAR:

1- What kinds of difficulties do developers face when integrating ProxemicUI into Story CreatAR? What was done to overcome these issues?
2- What features of ProxemicUI were deemed to be most useful by the developers for Story CreatAR? What features were less useful in this context?
3- How can the framework be improved to better support integration?

We also pose these research questions about indirect use of ProxemicUI through the Story CreatAR tool and existing proximity events in graphs of stories or their scrips:

1- What types of proxemic rules do the end-users use or express in their stories/graphs?
2- How many rules do the end-users use or express relative to the number of events in the stories?
3- What Type of compound rules exists in end-users' stories and graphs?

Story CreatAR utilizes ProxemicUI for interaction, which triggers actions that progress the story. The triggers can be in the form of relative distance, relative orientation, or a combination. An example of a rule involving relative distance is when the player walks within 2m of an avatar; then the avatar starts talking to the player. Another example involving relative orientation is when the player faces an avatar, then the avatar will turn to look at the player. These examples can be used as independent events or combined as a single event using the Compound Rules that ProxemicUI supports. By using Story CreatAR in this study, I evaluated ProxemicUI in two different ways: *Integrating ProxemicUI into a larger system* and *Indirect use of ProxemicUI (through the Story CreatAR tool)*.

Integration effort of ProxemicUI into Story CreatAR

Integrating ProxemicUI into a larger system is an important validation step for software frameworks that, by definition, are designed to provide a specific feature set or technical capability through adaptation to different application contexts. Therefore, integrating ProxemicUI into Story CreatAR is a good example of this validation step.

Study design

This section will discuss the study design of the integration process of ProxemicUI into Story CreatAR. It will include the study population, the informed consent, the study environment, the study procedure, the data collection, and the analysis method.

Participants

Participants for the integration process of ProxemicUI into Story CreatAR are five students (three grads and two undergrads) from the Faculty of Computer Science at Dalhousie University. There was no recruitment as all participants are involved in the development of Story CreatAR. I also took part in this process as well. I will refer to those five individuals as the “integration team”.

Informed consent

All participants involved in the study signed an informed consent form (Appendix 5.2). I emailed them a copy of the consent form to read, sign, and send it back before starting the study. The informed consent outlined the risks and benefits associated with the study, a description of the study, the participant's right to choose to participate or not, the participant's right to withdraw without consequence (or losing the ability to participate in the Story CreatAR research project), assurances of confidentiality and anonymity of personal data, and assurances of not using their data in the analysis if they chose to withdraw from the study.

Compensation

Participation in this study is voluntary, and there are no compensations.

Study environment

For the integration process, because Story CreatAR was built using Unity (a real-time development platform to build interactive content)[118], the integration team used Unity to integrate ProxemicUI into Story CreatAR. Documentation about the integration process was collected using a shared file in gitbook, accessible by all integration team members. All collaboration and discussion meetings are done through Microsoft Team.

Study Procedure

Systems integration is a common task involving the composition of components and subsystems and their specialization to support a specific context of use. Object-oriented frameworks like ProxemicUI are intentionally designed for integration: they provide basic functionality that requires specialization to be used. The integration effort will include the following aspects:

1- Defining a software build that compiles Story CreatAR and ProxemicUI.

2- Specializing ProxemicUI classes to support Story CreatAR features.

3- Designing user interface elements that expose ProxemicUI capabilities in Story CreatAR.

4- Writing the code that translates user interface element values into proxemics rules.

5- Writing the code that triggers interaction events in VR based on the proxemics rules.

6- Testing, debugging, and refactoring (improving code structure and organization).

In addition, participants proposed and discussed ways to modify and improve ProxemicUI to further benefit the development of Story CreatAR. Some of these improvements were proposed in an unstructured way through the development process and some others during the final interview with developers.

The study started with a one-hour introductory session, where I explained the different features of ProxemicUI. To accomplish this, I used two toy examples that were implemented using Unity. One of the developers implemented the first example, which moves two boxes next to each other while indicating if they are within thresholds by changing the color of a ball in the scene. The second example I implemented rotates four boxes around a center point in the middle and indicates when a box is facing the center by printing on the screen which box is facing the center. Each developer then worked independently where I was available throughout the study for any questions or issues encountered. The introductory session was recorded and used by developers as a reference. They also have access to the source code and examples of how to use the framework. While five developers developed Story CreatAR, only three of them worked with ProxemicUI. One of these three developers did not directly code using ProxemicUI but fixed some bugs of the code that uses ProxemicUI and implemented part of the stories using the interface that uses ProxemicUI as well. Therefore, at the end of the study, I conducted 30-minutes interviews with the two developers who worked with ProxemicUI. During the 30-minutes interviews, I asked the following questions:

1- What positive experiences or outcomes have you had when using the framework? Explain.

2- What negative experiences or outcomes have you had when using the framework? Explain.

3- What specific issues, concerns, or problems have you faced when using the framework?

4- How significant are the issues and concerns you have with the framework?

5- If you could choose a feature to add to the framework, what would it be? Why?

6- Are there any other ways you think the framework can be improved?

7- Can you think of a scenario where we can employ the use of ProxemicUI, other than Story CreatAR?

8- Is there anything else you want to add about the framework?

Data collection

I created a shared document accessible by the integration team and me to keep track of the work required to integrate ProxemicUI into Story CreatAR. This includes decisions that were made and why we chose them, issues encountered, pitfalls faced and how we addressed them, source code, and ideas for modifications. The document followed a journal format, wherein each entry we provide a date and classification of the entry (e.g., who made the note, date, issue, decision). I collected a screen recording of the introductory session when I explained the different features of ProxemicUI and the individual meetings with each developer at the end of the study. I also collected a video demo and a report about how ProxemicUI is being used. Finally, I also collected the source code history during the study and unstructured discussion through slack chats.

Analysis method

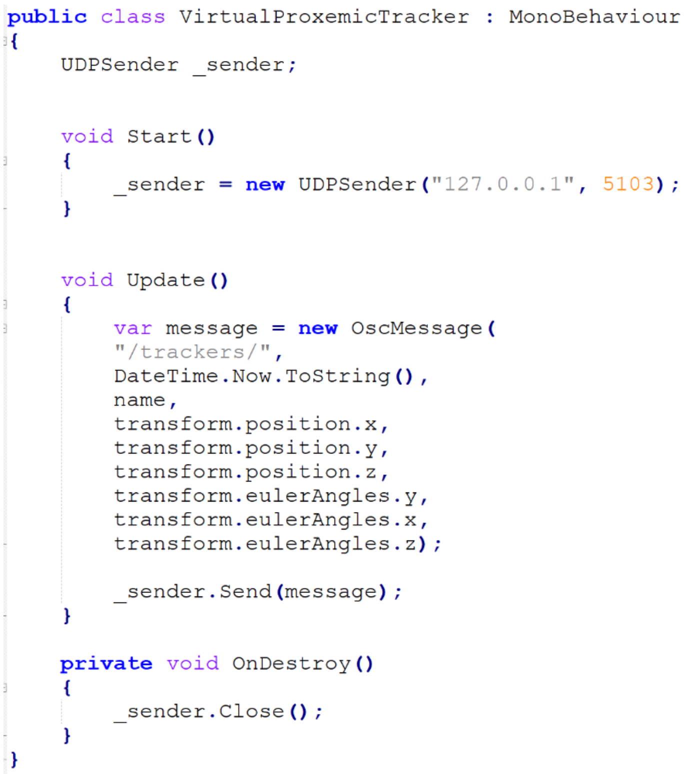
For my analysis, I am following the same approach I discussed in chapter 7 that is thematic analysis [21][22][77]. I started by transcribing video recordings from the interviews, taking notes of coding behaviors from source code history, notes from the interface design, notes from the journal file, and notes from slack chats. I grouped these notes and transcriptions into different groups to create my initial codes. Then I reviewed these groups to create my themes, where I identified six themes. Finally, I reported the results according to these themes, as I will discuss in the next section.

Results

This section discusses the result of the integration effort of ProxemicUI into Story CreatAR. The section is organized according to the six themes: ProxemicUI in use, unexpected use of ProxemicUI, issues while using ProxemicUI, additional features, future expansions, and use case scenarios. In this section, I will refer to developers as D1, D2, and D3.

ProxemicUI in use

- *The OSC communicator code*: the first step developers took was to write the OSC communicator code that captures the tracking data and sends them to ProxemicUI. In this case, Unity was used as a tracking system where the OSC communicator code captures the 3D tracking data of existing objects and sends them to ProxemicUI. D1 wrote the OSC communicator code, and he reported that he likes how generic the OSC messages are and found that it worked well to capture data from trackers to a central place like ProxemicUI. He stated that “I never thought that a Unity object could be used as a tracker, but we send the OSC, and it became a tracker”; “Everything can send its position and orientation through OSC can be used as a tracker”. He also thinks that employing OSC protocol

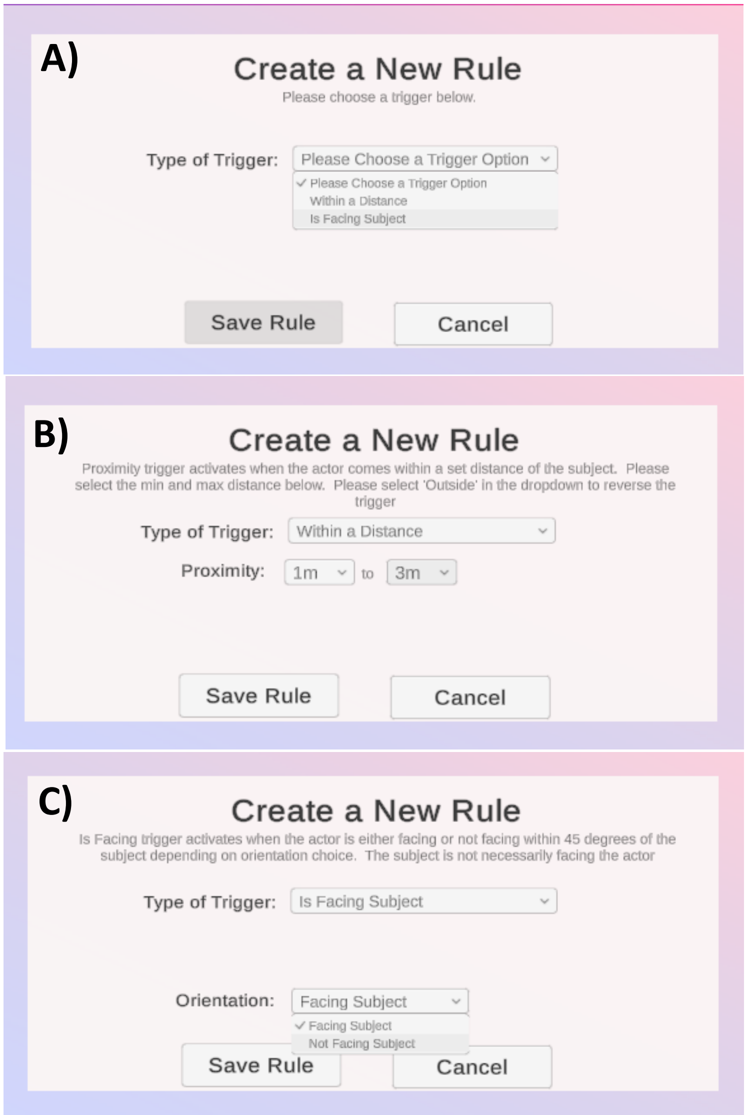


```
public class VirtualProxemicTracker : MonoBehaviour
{
    UDPSender _sender;

    void Start()
    {
        _sender = new UDPSender("127.0.0.1", 5103);
    }


    void Update()
    {
        var message = new OscMessage(
        "/trackers/",
        DateTime.Now.ToString(),
        name,
        transform.position.x,
        transform.position.y,
        transform.position.z,
        transform.eulerAngles.y,
        transform.eulerAngles.x,
        transform.eulerAngles.z);

        _sender.Send(message);
    }

    private void OnDestroy()
    {
        _sender.Close();
    }
}
```

FIGURE 60: DEVELOPERS' OSC COMMUNICATOR CODE

to receive the tracking data demonstrates a real strength of ProxemicUI to connecting with multiple trackers and not only Unity. Figure 60 shows the OSC communicator code that D1 wrote.

- *Implementing the proximity rules*: D2 created an interface that allows end-users to define rules based on their requirements. Four rules were added to the interface: RelativeDistanceRule, IsFacing, AND, and OR rules. The AND and OR



Create a New Rule

Please choose a trigger below.

Type of Trigger: Please Choose a Trigger Option

Create a New Rule

Proximity trigger activates when the actor comes within a set distance of the subject. Please select the min and max distance below. Please select 'Outside' in the dropdown to reverse the trigger

Type of Trigger: Within a Distance

Proximity: 1m to 3m

Create a New Rule

Is Facing trigger activates when the actor is either facing or not facing within 45 degrees of the subject depending on orientation choice. The subject is not necessarily facing the actor

Type of Trigger: Is Facing Subject

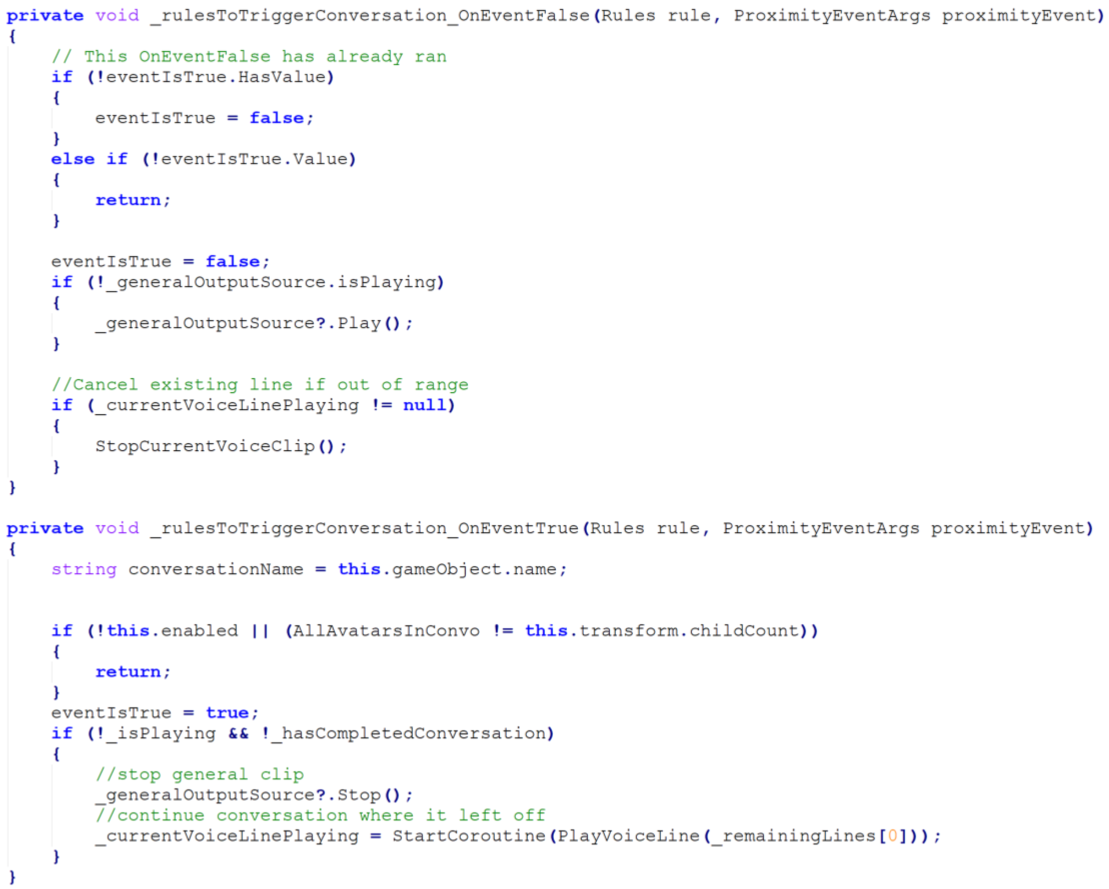
Orientation: Facing Subject

FIGURE 61: A) MAIN INTERFACE TO CREATE A PROXIMITY RULE IN STORY CREATAR, B) CREATING RELATIVEDISTANCE RULE, C) CREATING ISFACING RULE

rules were not used in the end-users approach due to some bugs with Story CreatAR that could not be fixed on time before starting the end-user experience. Therefore, the Rule interface allows end-users to create two types of rules: RelativeDistanceRule, IsFacingRule. Both developers think that the rules in ProxemicUI are very intuitive as they exactly tell what they are going to test. D2 likes how ProxemicUI combines two rules into one using *CompoundRule*, allowing her to test relative distance and orientation at once. Figure 61 shows the interfaces to create RelativeDistanceRule and IsFacingRule.

- *Saving the* rules: when a story is created in Story CreatAR, it will be saved to be experienced at different times. Developers of Story CreatAR store all story settings, including the created proximity rules, in a JSON file (JSON file is a file that uses JavaScript Object Notation to store data). When the story is played again, Story CreatAR retrieves all settings, including the proximity rules to experience the story.

- *Writing the handling methods*: both rules were used to eavesdropping or addressing the player by a non-player character, giving the end-user the option to trigger the eavesdropping when the player is within distance or facing the player the conversation. Figure 62 shows examples of the two handling methods for OnEventTrue and OnEventFalse to start or stop eavesdropping.



```
private void _rulesToTriggerConversation_OnEventFalse(Rules rule, ProximityEventArgs proximityEvent)
{
    // This OnEventFalse has already ran
    if (!eventIsTrue.HasValue)
    {
        eventIsTrue = false;
    }
    else if (!eventIsTrue.Value)
    {
        return;
    }

    eventIsTrue = false;
    if (!_generalOutputSource.isPlaying)
    {
        _generalOutputSource?.Play();
    }

    //Cancel existing line if out of range
    if (_currentVoiceLinePlaying != null)
    {
        StopCurrentVoiceClip();
    }
}

private void _rulesToTriggerConversation_OnEventTrue(Rules rule, ProximityEventArgs proximityEvent)
{
    string conversationName = this.gameObject.name;


    if (!this.enabled || (AllAvatarsInConvo != this.transform.childCount))
    {
        return;
    }
    eventIsTrue = true;
    if (!_isPlaying && !_hasCompletedConversation)
    {
        //stop general clip
        _generalOutputSource?.Stop();
        //continue conversation where it left off
        _currentVoiceLinePlaying = StartCoroutine(PlayVoiceLine(_remainingLines[0]));
    }
}
```

FIGURE 62: ONEVENTTRUE AND ONEVENTFALSE HANDLING METHODS TO START EAVESDROPPING TO A CONVERSATION

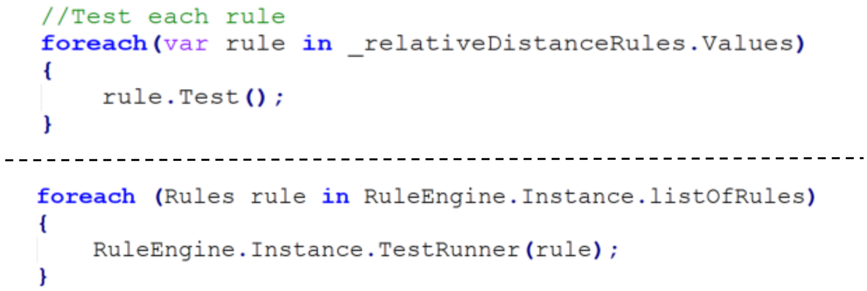
Unexpected use of ProxemicUI

- *Creating multiple rules*: one developer created a rule for each test between the player and every non-player character resulting in subscribing to multiple events and creating multiple handing methods. This can be accomplished by creating a single rule and apply it for all entities using the test condition (ANY/ALL).

- *Using proximity events*: initially, ProxemicUI v1 has only a single event for each rule (OnEvent) that would be fired when the test passes. D2 stated that using a single event was confusing as she was not sure how to check when the event is true or false. To overcome this issue, she created the basic proximity rule then applied the NOTRule to the basic proximity rule to get notified when the rule is not true. I updated ProxemicUI v1, and I added the two events for each rule (OnEventTrue and OnEventFalse) so that the developer would be notified when the event is true and false. D2 worked with both versions and thinks that the new update makes more sense in responding to the rule's status (true/false). However, while she used both OnEventTrue and OnEventFalse for the relative distance rule, she is still following the same approach to detect when the player is not facing a conversation. She created IsFacingRule then applied the NOTRule to it to detect the "not facing condition". This can be achieved by subscribing to OnEventFalse. Therefore, the same IsFacingRule can be created where the OnEventTrue can be used to detect the "facing condition" and OnEventFalse" to detect the "not facing condition. However, the developer is still using the NOTRule to detect the "not facing condition.

Issues while using ProxemicUI

- *Multi-threading in Unity*: As discussed in chapter 5, the *RuleEngine* in ProxemicUI has a separate task [113] that is continuously running in the background to test all rules in the system. D1 reported that there was an incompatibility between ProxemicUI and Unity with testing the rules that is because Unity is not thread-safe. With Unity, developers can create their threads to perform tasks in the background. Still, these threads cannot interact with Unity's main thread [115]. D1 worked around this issue by testing the rule directly inside Unity's *update* method, seen in figure 63 up. He thinks that it is not a major issue, but it is an obstacle that should not be there. However, this approach is inefficient as the *RuleEngine* does a couple of checks (will be discussed in the next point) before performing the test to ensure all arguments are available in the system, which I don't want to involve the developer with to make the implementation straightforward. To overcome this issue, I tried some existing solutions, such as using the Unity job system [117][52] and calling the *RuleEngine* from a different thread [119] but could not get it working with these approaches. Therefore, I added the *TestRunner*, *a* public method in the *RuleEngine*, that needs to be added to Unity's *update* method to perform the test. An example of how a developer would use it is in figure 63 down.



```
//Test each rule
foreach(var rule in _relativeDistanceRules.Values)
{
    rule.Test();
}
```

```
foreach (Rules rule in RuleEngine.Instance.listOfRules)
{
    RuleEngine.Instance.TestRunner(rule);
}
```

FIGURE 63: UP: HOW A DEVELOPER OVERCOME THE MULTITHREADING ISSUE IN UNITY, DOWN: CALLING THE TESTRUNNER METHOD FROM THE RULEENGINE

- *Not existing entities when testing the rules*: in ProxemicUI, each rule has references to the entities (only entity's ID) to be tested in that rule. There was an instance where the entities in the rule do not exist in ProxemicUI. For example, the developer creates the rule and passes the entities as s/he knows what entities to test (no need to retrieve them from ProxemicUI using the retrieval methods discussed in chapter 5). However, ProxemicUI still did not receive the OSC messages that contain the data for these entities, causing the RuleEngine to through an error. To overcome this issue, I added a couple of checks in the RuleEngine to ensure that all entities' data exist before testing the rule.

- *Repeatedly triggering events*: in ProxemicUI, OnEventTrue/OnEventFalse will be continuously triggered as long as testing the rule passes/fails. However, in some cases, developers would need to be notified only once when testing the rule passes/fails. D3 reported that she created a local Boolean to keep track of the rule status. To address this issue, I added the OnEventChange event to allow developers to subscribe to it in such cases.

Additional features

- *F-formation*: ProxemicUI detects existing F-Formation through the use of various rules (RelativeDistanceRule, RelativeOrientationRule, and IsFacingRule). This is suitable when employing ProxemicUI in a physical space context (e.g., smart home or museum). However, when it comes to virtual spaces, there is a need for more than detecting F-Formations. For example, when the integration team started implementing the stories, there was a need to create groups (through the use of F-Formation). This can be seen in creating a conversation that involves multiple avatars facing each other, creating a shape in their formation (e.g., circular formation). Therefore, following the same approach of defining a rule, I added the ability to specify a grouping F-Formation in ProxemicUI.

Future expansions

- *Adding the third* dimension: D1 reported that it would be useful to add calculating the third-dimension proximity data to ProxemicUI. Currently, ProxemicUI can receive 3D data about entities but calculate only 2D high-level proximity data (e.g., relative distance). This is because ProxemicUI was initially built to support proxemic interactions in physical spaces using a top-down tracking system (DT-DT [49]). One of my future works is to expand ProxemicUI to cover 3D to support more scenarios in both physical and virtual spaces. D2 thinks that ProxemicUI is complete for the purpose that she was using it for and stated that "there was not a time when I said I wish that thing existed".

- *Expanding IsFacingRule*: the current implementation of IsFacingRule allows to check if one or more entities are facing a single object (e.g., multiple users are facing a tabletop). In Story CreatAR, the IsFacingRule is used to detect if a player is facing a conversation or not. The conversation is created by creating a conversation node and add all participating avatars to it. Therefore, Story CreatAR is still checking if the player is facing a single object or not (in this case, the object is the conversation node). However, what if we don't have the conversation node and only have a number of avatars? This means we need to check if one entity or more are facing multiple objects. To address this case, I would overload the constructor of the IsFacing method to accept one-to-many and many-to-many. I would also calculate the center point between the objects that I need to check if the entities are facing or not; then, I will use this center point for the IsFacing check.

Use case scenarios

- *Data* Visualization: D1 thinks that ProxemicUI can be useful for data visualization. For example, when having an application that represents a lot of moving points where it highlights or animates specific points when some object within a distance to each other. Such support is already explored in physical settings (e.g., relative proximity between a user and the display [53]). However, it will be interesting to explore how to support visualizations based on proxemic relations between virtual objects.

- *Using HybridRule in* games: D2 suggested using *HybridRule* in video games. In some video games, players are required to perform an action to complete a task then wait for it to be completed. D2 suggests using *HybridRule,* where the game captures the action as an external event; then, instead of waiting, the player can wander around and get notified when the task is completed based on the player's proximity in the virtual world. This is similar to the use of HybridRule in Alice's smart home scenario. First, starting the coffee machine is similar to any action in the game space. Second, going to watch TV or work on the tabletop is similar to wandering around the game space. Finally, checking Alice's proximity to notify her when the coffee is ready is similar to checking the player's proximity in the game space to notify her/him when the task is completed.

Discussion

Integrating ProxemicUI into Story CreatAR provides evidence of the ability of ProxemicUI to support other systems by using the set of rules in ProxemicUI. This integration also provides evidence that ProxemicUI can be expanded to add more features, such as adding the creation of the F-formation feature. With some integration processes, developers might face incompatibility or issues during the integration. While my participants faced an issue with the multi-threading in Unity, they overcame this issue themselves by testing the rules directly. While I provided a solution for developers to test the rules without calling a different thread by calling the test method when they want to test the rule, it is important to provide such functionally to developers especially when they don't want the test to be running all the time. Besides, this integration also confirms that it is not enough to notify the developer when a proximity event has occurred; developers would need to know when the event is true and when it is false. Additionally, developers might want to be notified only once, which confirms the need to notify developers on changes of events using OnEventChanged.

Indirect use of using ProxemicUI

Indirect use of ProxemicUI (through the Story CreatAR tool) allowed us to evaluate the framework in use as part of an application interface. In this evaluation, end-users (authors) interacted with the Story CreatAR (UI interface), thereby indirectly used ProxemicUI by creating proxemics-based interaction triggers.

Study design

This section will discuss the study design of the indirect use of ProxemicUI. It will include the study population, the informed consent, the study environment, the study procedure, the data collection, and the analysis method.

Participants

Participants for the end-user experience of using ProxemicUI are two media studies undergraduates and one media studies faculty member at Dalhousie. I will refer to those three individuals as "authors".

Informed consent

All participants involved in the study signed an informed consent form (Appendix 5.3). I emailed them a copy of the consent form to read, sign, and send it back before starting the study. The informed consent outlined the risks and benefits associated with the study, a description of the study, the participant's right to choose to participate or not, the participant's right to withdraw without consequence (or losing the ability to participate in the Story CreatAR research project), assurances of confidentiality and anonymity of personal data, and assurances of not using their data in the analysis if they chose to withdraw from the study.

Compensation

Participation in this study was voluntary, and there were no compensations.

Study environment

For the indirect use of ProxemicUI, the screen recordings for all interactions were collected using Microsoft teams for both approaches. For the graph-based approach, authors used Miro [5], an online collaborative whiteboard that allows teams to complete different tasks (e.g., brainstorming and design), to create their graphs. For the interface-driven approach, authors interacted with Story CreatAR by gaining access to one of the integration team computers through Microsoft Teams.

Study procedure

Participants used ProxemicUI indirectly through using the Story CreatAR user interface. Participants used Story CreatAR to deploy stories they have written into a VR format in collaboration with the integration team. In this part of the study, participants used two approaches, and each took place at a different stage of the story deployment process:

1- *Graph-based approach*: authors worked with the integration teams to create a story graph. The graph is an intermediate representation that includes details about proximity-based triggers among other event transitions in the story. Each author participated in 5-7 one-hour sessions based on the progress of their graph. In the first session, we started with a quick introduction to the online tool we were using to create the graphs (Miro) and explained with an example how to use it. The rest of the sessions were led by the authors, where the role of the integration team was to take notes and ask questions about steps and decisions the author made during the session. During the sessions, participants were encouraged to think out loud to allow the integration team to understand what they were doing and their reasoning. At the end of this approach, the integration team conducted a 1.5-hour session where each author presented her/his graph to other authors and exchanged their ideas about different methods each has taken. The integration team also conducted a 15-30 minute session for each author, asking them how they can change/improve their graphs after looking at others' graphs. We started the session by having the author's graph that we were interviewing open on the shared display. Then, we referred back to other authors’ graphs through the questions we asked. We asked a lot of questions about the graphs that are related to Story CreatAR. However, below are the questions that are related to ProxemicUI use:

- Are there any events that are triggered by getting close to characters/objects?
- Are there any events that are triggered by facing a certain direction/object?
- Are there any arrangements for avatars while in a conversation (e.g., shape)?

2- *UI-based approach*: authors used the Story CreatAR interface themselves to implement their story, including creating groups, placement, and interaction rules. Each author participated in four 1-hour sessions. In the first session, the integration team started with a quick introduction about using Story CreatAR and showed its features. The rest of the sessions were led by the authors, where the role of the integration team was to support the authors (by answering questions), take notes, ask questions about steps and decisions made during the sessions.

During the sessions, participants were encouraged to think out loud to allow the integration team to understand what they were doing and their reasoning. In the first two sessions, authors were asked to use their graphs to implement their stories. In the last two sessions, authors used their scripts to implement their stories. This was to understand how they can benefit from using their own graphs and scripts, and how each data source can benefit the author. We asked a lot of questions about their use of Story CreatAR. However, below are the questions that are related to ProxemicUI use:

- Do you think there is a need to apply proximity rules (how close/far) to rooms as well? E.g., The three bedrooms have to be next to each other.
- Do you think there is a need for rules that detect when the player is facing a certain direction or towards a certain object/room/avatar? E.g., when the player is facing the door, it opens.
- Do you think you can benefit from different types of formations? If yes, what would they be?

During both approaches and at the end of each week, researchers met as required (weekly/biweekly) to analyze and discuss participants' behaviors during the past sessions and define questions about participants' decisions. A list of questions and/or defined new pointers and/or tasks are summarized in a shared document accessed by researchers only.

Data collection

During all sessions of the indirect use of ProxemicUI, screen recordings of participant usage of Story CreatAR and using Miro to create the graphs were collected. This also includes the authors' feedback during interviews about creating the graphs and using the Story CreatAR interface. Other data also were collected, including copies of the script, story graph, resulting VR output, and summaries of researchers' weekly/biweekly meetings. I also collected the documentation of the integration process and screen recordings of developers' interviews about their experience using ProxemicUI. In addition, I collected the resulting VR output of the developers’ experience implementing different parts of the stories.

Analysis method

For my analysis, I am following the same approach I discussed in chapter 7 that is thematic analysis [21][22][77]. I started by transcribing video recordings from the interviews and interaction sessions, taking notes of final graphs and stories scripts, and notes from the summaries of researchers' weekly/biweekly meetings. I also reviewed the final VR outputs of both the authors' and developers' experiences and took notes about their use of ProxemicUI through using the interface. During the transcription and taking notes phase, I was trying to identify what proximity features each story has and how authors can benefit from ProxemicUI to complete the implementation of their stories. I grouped these notes and transcriptions into different groups to create my initial codes. Then I reviewed these groups to create my themes, where I identified four themes. Finally, I reported the results according to these themes, as I will discuss in the next section.

Summary of the stories

This section briefly gives an overview of each story as a context for the results in the next section.

Spill

The player attends a tea party and tries to find clues and eavesdrop on the NPCs to find the keyword (secret) for each NPC. The player can receive strikes regarding their bad manners (maximum three strikes, then he leaves or is forced to leave the game). The player also can heal strikes by bringing tea to NPCs. The story has four endings based on how the player interacts and behaves during the game.

Tyson’s Peak

The story started when the player got trapped in a cabin during a snowstorm. There are eight friends in the cabin where they try to investigate the murder of one of them who was poisoned. The player can interact with different characters where s/he can eavesdrop or directly interact with members based on proximity data.

Standvillle museum

The story started when the player (who play the role of a detective) and his son visit the museum where the son gets kidnapped. The player has to find clues and solve riddles to find his son. The story has different endings based on the player's decisions during the game.

Results

As stated earlier, the authors participated in two approaches: graph-based approach and UI-based approach. This section will discuss how authors employ proxemics to tell their stories in both approaches, including the parts of their stories that they did not implement in any of the two approaches. It will also discuss some of the similarities and differences between authors and developers of using the interface. This section is organized according to the five themes: using proximity rules, applying proximity rules to game layout, using F-formation, expert user experience with the interface, and system limitations. I will refer to authors: A1, A2, and A3; and to developers D2 and D3.

Using proximity rules

In the Spill story, out of 41 events, there were 18 proximity events. Some of the events can be repeated based on the player interaction, but I only counted them once as there was no specific number on how many times they can be repeated. For example, the player can eavesdrop on NPCs conversation, but there is no specific number on many conversations that can take place during the story. In Tyson's peak story, out of 115 events, there were 52 proximity events. In this story, there is a specific number of conversations that take place during the game. Therefore, I counted every possible event in the story. For the Standville museum story, out of 115 events, there were 42 proximity events. In this story, I also counted all possible events. While this is the total number of events in the three stories, the authors did not implement all of them with both approaches (graph-based and interface-based) because they focused on implementing segments of their stories due to the limited time of the sessions. In addition, the authors faced some difficulties using the interface where they spent some time setting up the scenes, which is required before adding any proximity rules. In the rest of the section, I will discuss what proximity events exist in the stories and how they would be used.

Relative distance rule

All authors used relative distance in their graphs/stories to show how different events occur in the story. In the Spill story, there was a minimum of 10 relative distance rules; in

Tyson's peak story, there were 48 relative distance rules, and in the Standville museum story, there were 20 relative distance rules. For example, there are eavesdropping events in both Tyson's peak and Spill stories when the player is within a specific distance (e.g., 2 metres) to non-player characters. Other events include interacting with avatars when the player is too close (e.g., within 1 meter) to a conversation (e.g., directly addressing the player). Using Story CreatAR, A1 implemented two examples of using relative distance rules where she used the rules for both eavesdropping and directly addressing the player. A1 also liked how the distance rules are presented in Story CreatAR by providing minimum and maximum thresholds following ProxemicUI's approach; "I really like how you set up the proximity triggers, within distance, and you get the range. That is really easy to use" A1. A1 only implemented two examples so she can experience different features of Story CreatAR to implement different parts of the story before the end of the four sessions. In addition, A1 faced two issues with Story CreatAR. First, she could not add more than two proximity rules due to a bug in Story CreatAR. Second, Story CreatAR does not allow editing rules, so she had to delete and recreate the rules. Figure 64 up shows a part of Tyson's peak story that implements the relative distance proximity rule. The Standville museum story used relative distance to interact with objects or playing sounds. For example, when you get within 1 meter of a clue, you can pick the clue; or when you get closer to a room, you can hear sound coming from the room. Using Story CreatAR, A3 implemented a relative distance rule to make the player talk to himself when s/he gets closer to a room. The Spill story used relative distance to interact with avatars (e.g., greeting by the host when entering the front door) or interact with an object (e.g., when s/he gets within a distance of the table, the player talks to him/herself about the object). Using Story CreatAR, A2 implemented a relative distance rule to gather NPCs around the host in the living room. While all authors also used proximity between objects to define how to place game objects in the story (figure 64 down), A2 and A3 reported that they need to define proximity rules to allow the player to interact with objects (e.g., pick clues or teacup). The current implementation of Story CreatAR does not provide this feature. A2 addressed this issue by creating avatars and naming them as objects (e.g., table, clue). Then, he applied proximity rules between the player and objects. A2 also faced the issue of not being able to create more than two proximity events due to a bug in Story CreatAR.

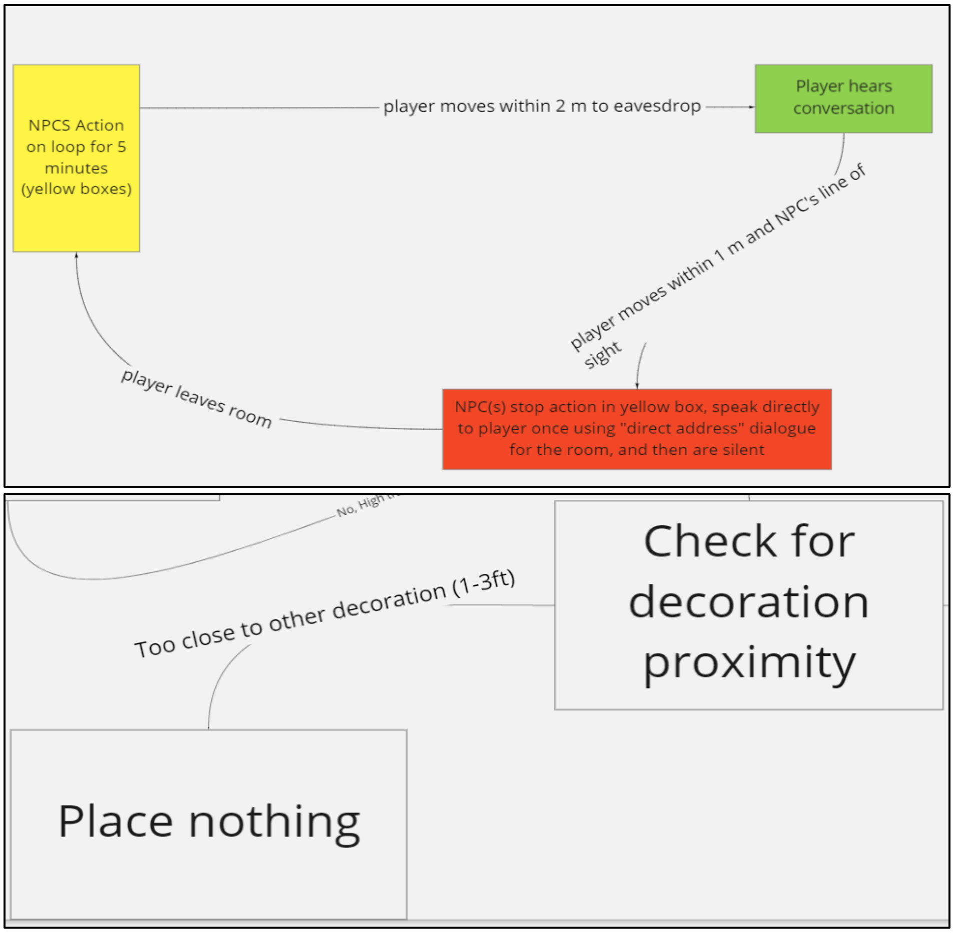

NPCS Action on loop for 5 minutes (yellow boxes)
player moves within 2 m to eavesdrop
Player hears conversation
player moves within 1 m and NPC's line of sight
NPC(s) stop action in yellow box, speak directly to player once using "direct address" dialogue for the room, and then are silent
player leaves room
No, High
Too close to other decoration (1-3ft)
Check for decoration proximity
Place nothing

FIGURE 64: UP: PART OF A1'S STORY SHOWS THE USE OF RELATIVE DISTANCE, DOWN: PROXIMITY BETWEEN OBJECTS IN A2'S STORY

Absolute position rule

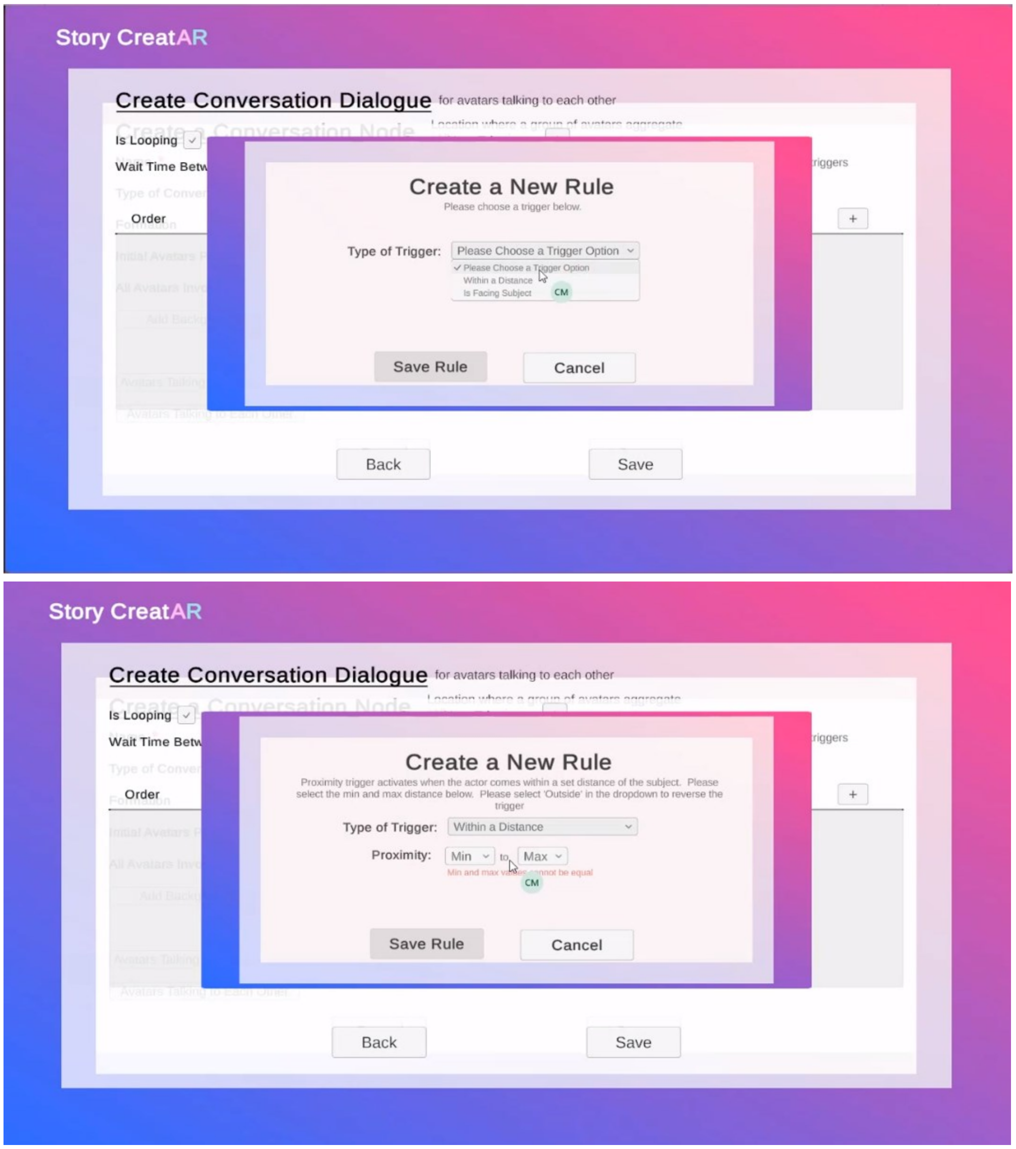
In all of the three stories, there were instances of using the absolute proximity position (the Spill five rules, Tyson's peak four rules, and the Standville museum sixteen rules). It is used to detect if the player or an NPC entered/left a room. For example, in the Standville museum story, the player needs to get to a specific location to find clues. In Tyson's peak story, an NPC will get in the center of the living room to gather all avatars, including the play, around her, so she talks to them. In the Spill story, the host will go to a specific living room location to gather all avatars around. While Story CreatAR does not support the creation of absolute proximity rules, they are suitable to detect such events, as it is relative to a specific location relative to the environment.

IsFacing rule

Only two stories have instances of using the IsFacing rule. For example, in the Spill story, the player must be facing a clue item to find it, and a non-player character should see (face) the player doing inappropriate behaviors to give a strike. Using Story CreatAR, A2 created an IsFacing rule to start addressing the player by the host. A2 also used this rule to allow the player to interact with the clue. In the Standville museum story, the IsFacing rule can be beneficial to ease the flow of the story. For example, A3's original thought is to show different clues after the player spends some time in a room. However, by using the IsFacing rule, he thinks it would be better to let the player wander around the room, and only when s/he faces a specific object, the clue will appear. In addition, when creating the IsFacing rule in Story CreatAR, authors can choose if the player is facing or not facing the subject. Therefore, A3 used the not-facing option to create a conversation where two avatars stand side by side.

Compound rules

Only two stories have instances of using a combination of relative distance and orientation to interact with objects. For example, in the Spill story, the player picks up the teacup by approaching and facing it, or the player must be within distance and facing the


Story CreatAR

Create Conversation Dialogue for avatars talking to each other

Create a New Rule

Please choose a trigger below.

Type of Trigger: Please Choose a Trigger Option

✓ Please Choose a Trigger Option
Within a Distance
Is Facing Subject

Story CreatAR

Create Conversation Dialogue for avatars talking to each other

Create a New Rule

Proximity trigger activates when the actor comes within a set distance of the subject. Please select the min and max distance below. Please select 'Outside' in the dropdown to reverse the trigger

Type of Trigger: Within a Distance

Proximity: Min to Max

Min and max values cannot be equal

FIGURE 65: AN AUTHOR CREATING A RELATIVE DISTANCE RUEL IN STORY CREATAR

NPCs to eavesdrop. Using Story CreatAR, A2 specifies that to see the clue, the player should be within a distance of the table and facing the clue, which is a combination of distance and orientation. In the Standville museum story, the player approaches and faces a game object (e.g., a painting) to get the following instructions. Using Story CreatAR, A3 reported that he would create a combination of distance and orientation to start a conversation if this option was available in the interface.

Applying proximity rules to game layout

In Story CreatAR, authors can add rooms to the scene they are creating, and Story CreatAR picks the room randomly according to the specified size (e.g., small, medium, large room) and the spatial placement rules attached to the entities in the room. In some cases, authors did not like the placement of rooms (e.g., how far they are from other rooms). Therefore, I asked them about applying proximity between rooms as an additional attribute to the size and spatial placement rules of entities when creating the room. All authors agreed that it would be beneficial to apply proximity for room placements. A1 thinks that it would be useful from storytelling and manageability points of view. For example, we can map the story to a full floor and only open part of it, if we need to, using proximity. In addition, in the Standville museum story, there is a couple of rules about applying distance between rooms. For example, the six offices are in close proximity, the security room is far from the bathroom, and the locker room is far from the security room.

Using F-Formation

Each of the three stories has a different type of F-formation that involves avatars only, avatars and objects, and objects only. For example, in the spill story, avatars form a semicircle facing away from the player to prevent him/her from eavesdropping. In the Standville museum story, the player has a conversation with an avatar while standing side by side facing a painting. A1 thinks the formations for her story are more like interacting with objects but not avatars (e.g., lying on bed and sitting on the couch). She also defined formations between objects when placing furniture in a room (e.g., “Couch about 3 m from the fireplace, facing it”).

Expert users experience with Story CreatAR interface

Two members of the development team (D2 and D3) used Story CreatAR to implement different parts of the three stories. This section will discuss the similarity and differences of using the interface by authors and developers. I am only discussing how they used ProxemicUI to complete the story as they have implemented different parts of the stories that do not contain proximity events.

- The Spill story: when A2 was implementing his story's introduction, he focused on creating interaction between the player and objects, which is no available in Story CreatAR. However, D2 implemented the same part of the story and created four distance rules for the player to interact with the host (greeted by the host). On the other hand, both A2 and D2 implemented the same relative distance and IsFacing rules for the same event (start addressing the player by the host). D2 also implemented 27 proximity rules from different parts of the story that A2 was not trying to implement during the sessions.

- Tyson’s Peak story: Both A1 and D3 implemented two distance rules for the same event that implements eavesdropping and addressing the player. A1 skipped implementing the rest of the proximity rules of this part of the story to explore different features of Story CreatAR and to implement different parts of the story before the end of the two sessions. This part of the story has a total of 12 proximity events, which D3 implemented through the interface. In addition, D3 implemented different parts of the story that A1 was not trying to implement during the sessions. These parts have eight proximity events (5 for eavesdropping and 3 for directly addressing the player).
- The Standville Museum story: for this story, there is no part of the story that both developers and authors implemented. However, D3 implemented different parts of the story, and she created eight proximity rules to play different lines where the player talks to him/herself when s/he is close to a room or an object.

System limitations

Story CreatAR

There were three limitations with Story CreatAR that limited authors of implementing some proximity rules:

- While the current implementation of Story CreatAR only implements two basic proximity rules (relative distance and IsFacing), the three stories involve other rules. These rules include the absolute position, relative orientation, and compound rules. For example, D2 reported that if they only used the IsFacing rule, the event will be triggered even if the player is in a different room. Therefore, they have to use a compound rule to trigger the event only when the player is facing and within a distance of the subject. This demonstrates the importance of the compound rules.

- While the current implementation of Story CreatAR allows applying proximity rules between avatars only, there were instances where authors want to create proximity interactions between the player and objects in the game (e.g., relative distance and orientation to a clue).
- Due to a bug in Story CreatAR, authors could not create more than two proximity rules in some instances.

ProxemicUI

There are three limitations of ProxemicUI that need to be added to provide a better to the stories:

- Implement more F-formations to support different events in the stories.
- Add tracing the history of proximity data which would be beneficial in many cases. For example, in the Spill story, the player needs to walk around the mushroom ring to move between the cognitive and real worlds. This can be achieved by tracking the history of the player's position around the mushroom ring, which is not supported currently in ProxemicUI.
- The current implementation of the RelativeOrientationRule takes only a single threshold. This means if the two entities are facing each other within the

threshold, from zero to the threshold (e.g., 45 degrees), then fire the event. However, the eavesdropping event in the Spill story requires the player to face the NPCs but not directly. This means that the threshold starts from more than zero to the maximum (e.g., from 10 to 45 degrees). Therefore, the RelativeOrientationRule should take two thresholds (minimum and maximum) to cover broader cases.

Discussion

Working with end-users confirms their need for different proximity events and F-formations to complete their stories. It also confirms that some experiences in virtual environments not only require detecting F-formations but also creating them (e.g., create a conversation node involving multiple avatars). In addition, tracing the history of proximity data (e.g., position) might play an important role during the interaction. For example, in the Spill story, the player can move between cognitive and real worlds by walking around the mushroom ring three times. This can be achieved by tracing the location history of player movements.

One of the interesting points we discussed with participants is applying proximity rules to the game layout. While all participants agreed this would be beneficial when designing a game, I think it also would be useful in different domains. I can imagine software to create building layouts that would prevent placing a room in a specific location in the layout because it is close/far from another room or an entrance.

Additional design requirements

Virtual environments require more than detecting proximity relationships between virtual objects; that is, to define how to place these objects in the environment. For example, placing avatars in a conversation, what formation they should form, and how far they should be from each other. To provide such support, ProxemicUI should provide developers with the detection and creation of F-Formations. This is a new requirement that ProxemicUI support, as I discussed in the F-Formation section in chapter 5.

Summary

This chapter presents a study that explores the process of integrating ProxemicUI into another system (Story CreatAR). This study consisted of two parts. First, exploring developer experience using ProxemicUI, including what features they used and what issues they encountered. Developers were able to set up ProxemicUI in Unity and implemented the proximity rules and their handling methods. However, while developers considered exposing the ability to define more complex relationships through the interface, they rely on fairly basic proxemic relationships when creating the interface for end-users. This decision needs to be explored more to study the extent to which end-users can define complex relationships through the interface. Second, exploring end-users' experience and how they benefit from using ProxemicUI's features to create their stories. While end-users benefited from ProxemicUI features and implemented few proximity rules during the process of creating some parts of their stories using Story CreatAR, their stories scripts and graphs show evidence of the need to detect different proximity relationships and F-formations between multiple entities. The chapter also discussed the additional design requirements that I derived from integrating ProxemicUI into Story CreatAR. For example, while detecting F-formations plays an important role in a proxemic-aware application, VR application requires calculating objects placements using F-formation (e.g., placing avatars in a semi-circular formation to create a conversation). Therefore, I expanded ProxemicUI to support objects placements using F-formations.

Chapter 9: Discussion

ProxemicUI makes a system contribution by implementing a robust rule-based object-oriented framework that supports proxemics awareness in smart environments.

Relative proximity rules are the most basic rules to detect proxemics interactions and are commonly used in the literature (e.g., using relative distance to control the content on screen [31][60][100]). ProxemicUI extends the support for these types of rules over existing toolkits in two ways. First, it encapsulates the low-level programming details and provides developers with the final results by specifying the minimum and maximum thresholds upon rules creations. Second, it allows developers to test multiple entities in a single rule by covering multiple test cases (one-to-one, one-to-many, and many-to-many) and specifying the test condition (ANY/ALL). These two features can be beneficial when applying the same rule to multiple entities, which reduces the developer's effort by defining a single rule for multiple tests. Besides, this abstraction makes ProxemicUI more scalable compared to The Proximity Toolkit [71]. A clear example is controlling multiple appliances through a tablet/phone based on the proximity to each appliance. In this example, ProxemicUI creates a single rule to test all entities, where the Proximity Toolkit creates a relation pair between every two entities.

In Klinkhammer et al.’s work [60], when two users come close to each other (each with her/his workspace), one of their workspaces is removed. They also stated that there were 257 bystanders out of 968 user sessions. Those bystanders are companions of users or strangers. During using TouristPlanner [73], authors reported that groups are joined by strangers, although the application was built to be used by a single group at a time. Also, members of groups who used TouristPlanner mostly arrived at different times, and some members left the table while others are still interacting, which cause changes in group formation at the display. Other reasons for changing group formation are changing members' roles and the size of other groups around the display [11]. This shows that some cases require the detection of multiple relations (between different users and the display, users, bystanders, and the display, or between groups and the display) to perform a response. ProxemicUI supports the need to test multiple relations by composing rules into a single rule using compound rules (ANDRule, ORRule, XORRule, and NOTRule). This composition of rules relief developers from keeping track of when multiple events have occurred and only listens for the main compound rule. While ProxemicUI allows combining different types of proximity rules, it will be interesting to apply and test the same concept on different types of interactions. For example, in Miners [130], a wall display game, the player would need to use tangible and direct interaction to activate a tool; CollabDraw [78], a collaborative art and photo manipulation application, allows to combine multiple interaction techniques (touch and in-air) from multiple users. It would be interesting to explore how the use of rule composition can be reapplied to combining interaction input.

Other systems integrate direct interactions with proximity data to control systems responses. For example, Medusa [7] shows a blurred half circle as the user reaches the display, which will open interaction options as the half-circle is being touched. Similarly, Vogel and Balakrishnan [123] also track hands to perform some in-air gestures; they also require the user’s direct touch in the personal interaction phase to present detailed information. On the other hand, in the AirPlayer [107], the user starts her/his preferred list of songs, then the system detects her/his location to play the list (e.g., play the list on bedroom speaker vs. living room speaker). Jokela et al. [54][55] introduced a grouping technique where one user starts the group on her/his phone, then touches other users’ phones to add them to the group (phone to phone proximity). Therefore, we can see the presence of the two types of integrating external events with proximity events. First, detecting the proximity event, then listen for the external event. I consider this type of integration a sequence of events that is technically supported, as I discussed earlier. Second, detect the external event, notify the system about it, and then listen for the proximity event. ProxemicUI implements a class to supports the second type of integration through HybridRule, where it provides a mechanism to listen for external events to test the proximity event. HybridRule also allows developers to define extended rules where it can have a CompoundRule as an argument, which itself has multiple rules. While I discussed how to employ the HybridRule in physical and virtual spaces, it will be interesting to explore how we can benefit from the use of HybridRule on presenting contents on the display. For example, Von Zadow et al. [130], Sabri et al. [94], Andrews et al. [6], and Bezerianos et al. [17] discussed the lack of awareness about changes on the display when working on a large wall display with close distance. For example, we might consider a certain change on the display an external event; when this change occurs, the system will check the user proximity or touch proximity (hand position) relative to the display, then show notification about that change.

Proximity awareness plays an important role in supporting social interactions around interactive systems and in virtual environments. For example, detecting F-formations can allow the systems to adjust contents on display according to the formations [11], notify developers about openings on the tabletop [73], or support interactions in medical settings [76]. In virtual environments, Williamson et al. [128] argued that proximity awareness is important from a management perspective to control the flow of virtual events (e.g., to know who is in a specific room). Perhaps, this control expands to cover the placement of people in different rooms and not only detecting their locations. For example, during the COVID-19 pandemic, lots of schools over the world moved to teach through online tools. I can imagine having a virtual world tool where students and teachers log into, and when a class started, all students involved in a specific class would be placed in the same virtual room. Besides, Story CreatAR [105] requires creating F-formations between avatars through creating conversation nodes. Creating F-formation in Story CreatAR means defining the exact placement of each avatar in the formation. ProxemicUI supports specifying F-formations in two ways. First, it allows developers to use existing rules to define a formation between entities (create a formation rule). Then developers can retrieve this formation rule to test if it exists or not. Second, ProxemicUI allows developers to calculate the exact placement of each entity in the formation using the formation placement rules. This rule will return each entity and its new position and orientation.

Using machine learning requires a lot of processing, including creating and preparing a dataset and training and testing the classifier. For example, Aghaei et al. [1] collected their dataset due to the lack of datasets that include the interactions they want to detect. Therefore, the cost of this process will increase as the number and complexity of the required interaction techniques increases, making machine learning not suitable for every context. ProxemicUI provides a robust and straightforward event model to detect complex proxemics interactions, making it a suitable alternative to machine learning in some contexts (e.g., quick setups where time and effort do not permit for machine learning Iterations). However, when building a proxemic-aware machine learning classifier, ProxemicUI can support this process in two ways. First, ProxemicUI can be used to generate/label data to train a machine learning classifier. Second, developers can use ProxemicUI as a source of proximity data to be forwarded to the machine learning classifier. In general, if developers want to use a tool for data collection, this tool should be built for the same specific purpose as the classifier to ease the data collection process. This is demonstrated in the comparison between ProxemicUI and Microsoft PSI. While I explored how to support a proxemic-aware machine learning classifier by generating training data or feeding the high-level proximity data to the classifier, it would be interesting to explore how to use ProxemicUI to validate the classifier's results. For example, Villena-Román [121] introduced an approach for text categorization that validates the results of the machine learning classifier through the use of a rule-based system.

Summary of refinements

As I discussed in previous chapters, I evaluated ProxemicUI through proof of concept applications, code review study, and integration into Story CreatAR. These evaluation methods result in a set of refinements to ProxemicUI. This section summarizes what I added of these refinements to ProxemicUI.

Using ProxemicUI for the smart home scenario

This section will summarize the refinements I made to ProxemicUI to support the interactions in the smart home scenario.

- I defined a default OSC message format in the *DataReceiver* that contains the base 3D proximity data for each entity (ID, timestamp, X, Y, Z, YAW, PTICH, ROLL). This allows ProxemicUI to connect with most tracking systems (e.g., VIVE trackers) as they provide such data. I used this format to connect ProxemicUI to the VIVE trackers when I used ProxemicUI to support the smart home scenario. This new OSC format results in changes for many classes as follows. First, the *EntityContainer* class was modified to match the new format of OSC messages as it is responsible for capturing the data from the *DataReceiver* to create/update entities. Second, the entity's ID changed from type integer to string because some tracking systems (e.g., VIVE trackers) use a combination of letters and numbers as tracker ID. Third, the timestamp field was added to the properties of entities, which can be used to calculate other properties (e.g., velocity). Lastly, the position and orientation data were expanded from 2D to 3D.

- Instead of setting the IP address and port number internally in the *DataReceiver,* I changed it to be specified during the initialization of the *DataReceiver*. With this approach, developers do not need to make changes to ProxemicUI's code every time the IP address and port number change.
- IsActive field was added to the entity properties because some devices might not be active but still exist in the environment (e.g., in sleep mode) or may not be tracked anymore (e.g., not visible to the tracking system). The *EntityContainer* used to have *EntityRemovedEvent*, which is fired when the entity is no longer being tracked. I replaced this event with *EntityInactiveEvent*, which will be fired when the IsActive field changes to false.
- Basic proximity rules used to have a single constructor that takes two lists, which might not be ideal for developers. When creating a rule to perform a one-to-one test, developers would have to create two lists, add an entity to each list, and then pass them to the rule. I added multiple constructors in all basic rules to cover all test cases (one-to-one, one-to-many, and many-to-many) to overcome this issue. In addition, I added the *TestCondition* to the basic proximity rules to allow developers to test for ALL or ANY entities. This means the test must be true for all entities or just a subset of them.
- All rules used to have a single event (OnEvent), which will be fired when the test is true. This approach is not sufficient and increases the developer's effort. Instead of defining two rules: one to test if you are close to an appliance and the other to test if you are far from the appliance, we can create a single rule for both tests

and use two events (OnEventTrue/OnEventFalse). Therefore, I replaced the OnEvent event with the two events (OnEventTrue/OnEventFalse).

- To provide a more generic *HybridRule* that can combine different types of events with proximity events, I replaced the *UIEvent* method with the *ExternalEvent* method. I also overloaded the *ExternalEvent* method: one follows the base event model format in C# and the other one more generic that can be called directly. I used this generic form of the *ExternalEvent* method when the external event was coming through an OSC message.
- Some use cases require checking the mobility status of entities to draw responses accordingly. To support this requirement, I implemented the *MobilityRule*, which allows developers to check and update the mobility status for a set of entities that the developer specifies.
- In some scenarios, developers need to check the relative orientation between two entities based on the direction of one of them only. For example, a developer might want to check if the user is facing the tabletop, but it does not matter at which side of the tabletop the user is standing. This test cannot be fulfilled using the RelativeOrientationRule. Therefore, I implemented the IsFacingRule for this purpose.
- To be able to pass the test results of each rule to the developers, we need to have an object that contains the results and can be included in the event object when it is fired. Therefore, I implemented the *ProximityEventArgs* class that has several

subclasses, one for each type of rule. These subclasses can be used to carry the test results to the developer through the event object.

Integrating ProxemicUI into Story CreatAR

This section will summarize the refinements I made to ProxemicUI during its integration into Story CreatAR.

- I implemented the *Formation* class that allows developers to specify F-formations using the proximity rules in ProxemicUI or calculate the exact placement of entities in an F-formation. Defining the exact placement of entities was a requirement of Story CreatAR, which was used to create conversations between avatars. Currently, the formation placement implements a circular formation only.
- In the *RuleEngine,* I implemented a public version of the *FormationCreator* (this method to create F-formations) and *TestRunner* (this method to test the rules) methods to give developers the option to run the test manually and to overcome the multithreading issue in Unity.
- When using the two events (OnEventTrue/OnEventFalse) in rules, the system will keep notifying the developers as long as the event is true or false. However, in some cases, developers just need to be notified when there is a change in the status of the event (changed from true to false and vice versa). Therefore, I added the OnEventChanged event to all rules.

Future work

In the previous section, I summarized the refinements to ProxemicUI that I completed. In this section, I will list the refinements to ProxemicUI that I left for future work.

- *Expanding F-formation:* I am planning to expand the F-formation support in two ways. First, ProxemicUI implements the placements of entities in a circular formation. This is important to support creating other types of F-formations (e.g., L-shaped, Z-shaped, and triangle formations). Second, ProxemicUI allows developers to create a formation rule using its proximity rules. I am planning to expand this to allow developers to parametrize the formation rule. With this approach, ProxemicUI will have a set of predefined formations (circle, triangle, v-shaped, etc.) where developers can specify the formation parameters. For example, a developer should be able to define a formation rule such as: create a circular formation between this set of entities with a maximum threshold of 2 meters.

- *Expanding the TestCondition*: currently, ProxemicUI provides two modifiers for the *TestCondition*. First, “ALL” is used to apply the rule to all entities, and they must all pass the test in order for the rule to be true. Second, “ANY” is used to apply the rule to all entities, and at least one entity must pass the test in order for the rule to true. This means the “ANY” modifier might return more than one entity. I am planning to expand the *TestCondition* by adding one more modifier (“ONE”), which will fire the event true when one entity passes the test. With this modifier, the rule will be a short circuit that stops testing the entity as soon as one of them passes the test.

- When ProxemicUI tests a rule, it notifies developers if the rule is true/false and passes the IDs that passed the test using the ProximityEventArgs through the event object. I am planning to expand the ProximityEventArgs so that it includes the proximity values of the tests for each entity. This can be beneficial in two ways. First, by providing the developers with the high-level proximity values (the results of the test, e.g., the relative distance between two entities), developers can do further testing at their end if they wish to. Second, when collecting training data for a machine learning classifier, developers can access the proximity data and customize the data as they wish. Both of these points were strongly recommended by the participants of the code review study.

- *Expanding Querying support*: while ProxemicUI allows developers to query entities based on basic attributes (e.g., ID, type, and shape), there might be a need to support querying entities based on their proximity data. Therefore, I am planning to add a set of querying methods as follows. First, retrieve all entities within a specific region, which can be based on 2D or 3D space. Williamson et al. [128] discussed the need for proxemic awareness to control the flow of virtual events. I discussed that this support might be extended to cover the placement of avatars (e.g., place all avatars in the conference room at the start of a lecture). Before we perform such placements, we might need to check who is in the conference room. If all participants are there, we don’t need to perform the placement. Second, retrieve entities within a specific range of another entity, which might be useful to control crowd flow. For example, in a supermarket, we might retrieve all entities that are close to the cashier, where we might open an additional cashier based on the number of customers in line. Third, retrieve all entities spawned within a specific time period, which can be useful for security checks. For example, if a crime occurred at an airport, we might retrieve a list of all entities with the time frame of the crime. Then we might combine this list with another list that we retrieve based on the location of the crime to finalize the suspected subjects list.

- *TracedEntity*: in Story CreatAR, there was a task where the player walks around a mushroom ring three times. To address such tasks, ProxemicUI needs to trace the player's position to make sure that s/he walked three times around the ring. Therefore, I am planning to add *TracedEntity* to each entity to maintain a history of proximity values.
- All comparison methods between entities are currently placed in the Geometry class, which is not suitable, especially if we consider expanding the shapes. With this approach, when developers add a new shape class, they will need to modify all existing shape classes so they can be compared. Therefore, I am planning to move all comparisons to a separate class (e.g., Comparison Container class) to allow developers only to modify this class and not all shapes.

- Currently, ProxemicUI is a centralized system that works on a single machine (e.g., in the smart home scenario, ProxemicUI is used by the controller only, which is the tablet that controls all other appliances). This is because my focus was on the developer model to create more complex proximity rules. However, I am planning to expand the *RuleEngine* to provide a distributed event model. With this distributed event model, multiple devices can create rules and add them to the *RuleEngine*, subscribe to events, get notified when they occur, and inquire about rules created by other devices to subscribe and get notified when their events occur. The communications between the *RuleEngine* will take place over the network.

- While tracking data can be gathered from different hardware components such as depth cameras, physical trackers, and Wi-Fi signals, the current implementation of ProxemicUI provides a single class (DataReceiver) that employs the OSC communication protocol to receive the tracking data from different systems. One extension point in the framework lies in DataReceiver, which can be subclassed to implement different protocols to capture the tracking data or to define a new OSC message format that takes in different types of data. Such changes need to be reflected in a corresponding ProximityEntity subclass if new types of data are being received, and Rule subclasses would need to be written that make use of this data.

- While developers can interact with ProxemicUI through creating rules and subscribing to their events and writing the handling methods, I plan to create a generic interface that allows end-users to interact with ProxemicUI. This interface would allow users to create rules, then show previews of how interaction would take place based on the created rules. This way, end users would build understating of how the rules work such that they can modify them as required for their own needs.

- As discussed in the thesis, the *role* of an entity might be important during an interaction. For example, should we give control of appliances to guests or children? If yes, to what extend should they gain control? While I think such decisions are related to the application layer and the developer should make them, ProxemicUI can support such scenarios. For example, we can add additional attributes that define the role of an entity, and combine the proximity with the roles in Rules to fire events and draw system responses.

Chapter 10: Conclusion

Developers of proxemics-aware applications in smart spaces require tools to support the detection of more complex interaction scenarios. Still, there is a gap between what existing tools provide and what developers require to implement more complex scenarios (e.g., testing multiple proximity attribute for multiple entities simultaneously and combining external events with proximity events). Therefore, I introduced ProxemicUI to fill this gap and support these complex scenarios using CompoundRules and HybridRule. I evaluated ProxemicUI through a set of proof-of-concept applications, a comparative code review study, the integration into an augmented reality storytelling tool. The results demonstrate that ProxemicUI supports rapid prototyping of systems that can respond to complex proxemics events across various contexts, including proxemics awareness in physical and virtual spaces and supporting proxemics awareness through machine learning.

Bibliography

[1] Aghaei, M., Dimiccoli, M. and Radeva, P. 2016. With whom do i interact? Detecting social interactions in egocentric photo-streams. *Proceedings - International Conference on Pattern Recognition* (Jan. 2016), 2959–2964.

[2] Alameda-Pineda, X., Yan, Y., Ricci, E., Lanz, O., Kessler, F.B. and Sebe, N. Analyzing Free-standing Conversational Groups: A Multimodal Approach.

[3] Alnusayri, M., Hu, G., Alghamdi, E. and Reilly, D. 2016. ProxemicUI: Object-oriented middleware and event model for proxemics-aware applications on large displays. *EICS 2016 - 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (New York, New York, USA, 2016), 50–60.

[4] Alnusayri, M.A. 2015. Proximity Table: Exploring Tabletop Interfaces that Respond to Body Position and Motion. March (2015).

[5] An Online Visual Collaboration Platform for Teamwork: 2020. *https://miro.com/*. Accessed: 2021-03-31.

[6] Andrews, C., Endert, A., Yost, B. and North, C. 2011. Information visualization on large,high-resolution displays: Issues, challenges, and opportunities. *Information Visualization*. 10, 4 (2011), 341–355. DOI:https://doi.org/10.1177/1473871611415997.

[7] Annett, M., Grossman, T., Wigdor, D. and Fitzmaurice, G. 2011. Medusa: A proximity-aware multi-touch tabletop. *UIST'11 - Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (New York, New York, USA, 2011), 337–346.

[8] Antle, A.N., Bevans, A., Tanenbaum, J., Seaborn, K. and Wang, S. 2011. Futura: Design for collaborative learning and game play on a multi-touch digital tabletop. *Proceedings of the 5th International Conference on Tangible Embedded and Embodied Interaction, TEI'11* (2011), 93–100.

[9] Apitz, G. and Guimbretière, F. 2005. CrossY: A Crossing-Based Drawing Application. *ACM Transactions on Graphics*. 24, 3 (2005), 930–930. DOI:https://doi.org/10.1145/1073204.1073286.

[10] Appert, C. and Beaudouin-Lafon, M. 2008. SwingStates: Adding state machines to the swing toolkit. *UIST 2006: Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology* (2008), 319–322.

[11] Azad, A., Ruiz, J., Vogel, D., Hancock, M. and Lank, E. 2012. Territoriality and behaviour on and around large vertical publicly-shared displays. *Designing Interactive Systems Conference* (2012), 468–477.

[12] Ballagas, R., Ringel, M., Stone, M. and Borchers, J. 2003. iStuff: A physical user interface toolkit for ubiquitous computing environments. *Conference on Human Factors in Computing Systems - Proceedings* (2003), 537–544.

[13] Ballendat, T., Marquardt, N. and Greenberg, S. 2010. Proxemic Interaction: Designing for a Proximity and Orientation-Aware Environment. *ACM International Conference on Interactive Tabletops and Surfaces - ITS '10* (2010), 121.

[14] Benford, S., Rowland, D., Flintham, M., Drozd, A., Hull, R., Reid, J., Morrison, J. and Facer, K. 2005. Life on the Edge: Supporting Collaboration in Location-Based Experiences. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '05* (2005), 721–730.

[15] Bespoke.osc — Bitbucket: *https://bitbucket.org/pvarcholik/bespoke.osc/src/master/*. Accessed: 2021-05-17.

[16] Bespoke Software » Open Sound Control: *http://www.bespokesoftware.org/wordpress/open-sound-control/*. Accessed: 2021-05-17.

[17] Bezerianos, A. and Isenberg, P. 2012. Perception of visual variables on tiled wall-sized displays for information visualization applications. *IEEE Transactions on Visualization and Computer Graphics*. 18, 12 (2012), 2516–2525. DOI:https://doi.org/10.1109/TVCG.2012.251.

[18] Block, F., Haller, M., Gellersen, H., Gutwin, C. and Billinghurst, M. 2008. VoodooSketch: Extending interactive surfaces with adaptable interface palettes. *TEI'08 - Second International Conference on Tangible and Embedded Interaction - Conference Proceedings* (2008), 55–58.

[19] Bonsch, A., Radke, S., Overath, H., Asche, L.M., Wendt, J., Vierjahn, T., Habel, U. and Kuhlen, T.W. 2018. Social VR: How Personal Space is Affected by Virtual Agents' Emotions. *25th IEEE Conference on Virtual Reality and 3D User Interfaces, VR 2018 - Proceedings*. (2018), 199–206. DOI:https://doi.org/10.1109/VR.2018.8446480.

[20] Braun, V. and Clarke, V. 2013. Successful Qualitative Research: A Practical Guide for Beginners. 400.

[21] Braun, V. and Clarke, V. 2012. Thematic Analysis. *APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological.* 57–71.

[22] Braun, V. and Clarke, V. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology*. 3, 2 (2006), 77–101. DOI:https://doi.org/10.1191/1478088706qp063oa.

[23] Calculate code metrics - Visual Studio (Windows) | Microsoft Docs: *https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019*. Accessed: 2021-08-07.

[24] Chi, P.Y. and Li, Y. 2015. Weave: Scripting cross-device wearable interaction. *Conference on Human Factors in Computing Systems - Proceedings* (2015), 3923–3932.

[25] Composite Design Pattern: 2015. *https://sourcemaking.com/design_patterns/composite*. Accessed: 2021-04-18.

[26] Cumin, J., Lefebvre, G., Ramparany, F. and Crowley, J.L. 2017. Human activity recognition using place-based decision fusion in smart homes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 10257 LNAI, (2017), 137–150. DOI:https://doi.org/10.1007/978-3-319-57837-8_11.

[27] Das, B., Seelye, A.M., Thomas, B.L., Cook, D.J., Holder, L.B. and Schmitter-Edgecombe, M. 2012. Using smart phones for context-aware prompting in smart environments. *2012 IEEE Consumer Communications and Networking Conference, CCNC'2012*. (2012), 399–403. DOI:https://doi.org/10.1109/CCNC.2012.6181023.

[28] Data Preparation and Feature Engineering in ML: *https://developers.google.com/machine-learning/data-prep*. Accessed: 2021-02-27.

[29] Dey, A.K., Abowd, G.D. and Salber, D. 2001. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction*. 16, 2–4 (2001), 97–166. DOI:https://doi.org/10.1207/S15327051HCI16234_02.

[30] Dostal, J., Hinrichs, U., Kristensson, P.O. and Quigley, A. 2014. SpiderEyes: Designing attention- and proximity-aware collaborative interfaces for wall-sized displays. *International Conference on Intelligent User Interfaces, Proceedings IUI* (2014), 143–152.

[31] Dostal, J., Kristensson, P.O. and Quigley, A. 2013. Multi-view proxemics: Distance and position sensitive interaction. *2nd ACM International Symposium on Pervasive Displays (PerDis'13)*. (2013), 1–6. DOI:https://doi.org/10.1145/2491568.2491570.

[32] Edwards, W.K., Newman, M.W. and Poole, E.S. 2010. The infrastructure problem in HCI. *SIGCHI Conference on Human Factors in Computing Systems (CHI'10)* (2010), 423–432.

[33] EventArgs Class (System) | Microsoft Docs: *https://docs.microsoft.com/en-us/dotnet/api/system.eventargs?view=netframework-4.8*. Accessed: 2021-06-12.

[34] Fabregat, M.B., Leyva, F.D.M., Saina, J. and Sarangi, A. 2016. Demonstration of MyAppCorner: Creating personal interaction areas on a public screen. *AcademicMindtrek 2016 - Proceedings of the 20th International Academic Mindtrek Conference* (2016), 445–448.

[35] Fayad, M.E. and Schmidt, D.C. 1997. Object-oriented application frameworks. *Communications of the ACM*. 40, 10 (1997), 32–38. DOI:https://doi.org/10.1145/262793.262798.

[36] Foote, B. and Johnson, R.E. 1988. Designing reusable classes. *Journal of Object-Oriented Programming*. 2, 1 (1988), 22–35.

[37] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company.

[38] Greenberg, S., Marquardt, N., Ballendat, T., Diaz-Marino, R. and Wang, M. 2011. Proxemic Interactions: The New Ubicomp? *Interactions*. 18, 1 (2011), 42. DOI:https://doi.org/10.1145/1897239.1897250.

[39] Grønbæk, J.E., Korsgaard, H., Petersen, M.G., Birk, M.H. and Krogh, P.G. 2017. Proxemic Transitions: Designing Shape-Changing Furniture for Informal Meetings.

[40] Grønbæk, J.E., Rasmussen, M.K., Halskov, K. and Petersen, M.G. 2020. KirigamiTable: Designing for Proxemic Transitions with a Shape-Changing Tabletop. *Conference on Human Factors in Computing Systems - Proceedings* (2020), 1–15.

[41] Hall, E.T. (Edward T. 1966. *The hidden dimension*. Doubleday.

[42] Hansen, T.E., Hourcade, J.P., Virbel, M., Patali, S. and Serra, T. 2009. PyMT: A Post-WIMP Multi-Touch User Interface Toolkit. (2009).

[43] Hecht, H., Welsch, R., Viehoff, J. and Longo, M.R. 2019. The shape of personal space. *Acta Psychologica*. 193, (2019), 113–122. DOI:https://doi.org/10.1016/j.actpsy.2018.12.009.

[44] Hedayati, H., Szafir, D. and Andrist, S. 2019. Recognizing F-Formations in the Open World. *ACM/IEEE International Conference on Human-Robot Interaction* (2019), 558–559.

[45] Hedayati, H., Szafir, D. and Kennedy, J. 2020. Comparing F-formations between humans and on-screen agents. *Conference on Human Factors in Computing Systems - Proceedings* (2020), 1–9.

[46] Heer, J. and Boyd, D. 2005. Vizster: Visualizing online social networks. *Proceedings - IEEE Symposium on Information Visualization, INFO VIS* (2005), 33–40.

[47] Heer, J., Card, S.K. and Landay, J.A. 2005. Prefuse: A toolkit for interactive information visualization. *CHI 2005: Technology, Safety, Community: Conference Proceedings - Conference on Human Factors in Computing Systems* (2005), 421–430.

[48] Hill, J. and Gutwin, C. 2004. The MAUI toolkit: Groupware widgets for group awareness. *Computer Supported Cooperative Work*. 13, 5–6 (2004), 539–571. DOI:https://doi.org/10.1007/s10606-004-5063-7.

[49] Hu, G., Reilly, D., Alnusayri, M., Swinden, B. and Gao, Q. 2014. DT-DT: Top-down Human Activity Analysis for Interactive Surface Applications. *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces - ITS '14* (2014), 167–176.

[50] iLab Cookbook - The Proximity Toolkit: *http://grouplab.cpsc.ucalgary.ca/cookbook/index.php/Toolkits/ProximityToolkit*. Accessed: 2021-04-13.

[51] Interpreter Design Pattern: *https://sourcemaking.com/design_patterns/interpreter*. Accessed: 2021-06-10.

[52] JacksonDunstan.com | Job System Tutorial: *https://www.jacksondunstan.com/articles/4796*. Accessed: 2021-06-11.

[53] Jakobsen, M.R., Sahlemariam Haile, Y., Knudsen, S. and Hornbaek, K. 2013. Information visualization and proxemics: Design opportunities and empirical findings. *IEEE Transactions on Visualization and Computer Graphics*. 19, 12 (2013), 2386–2395. DOI:https://doi.org/10.1109/TVCG.2013.166.

[54] Jokela, T., Chong, M.K., Lucero, A. and Gellersen, H. 2015. Connecting devices for collaborative interactions. *Interactions*. 22, 4 (Jun. 2015), 39–43. DOI:https://doi.org/10.1145/2776887.

[55] Jokela, T. and Lucero, A. 2013. A comparative evaluation of touch-based methods to bind mobile devices for collaborative interactions. *Conference on Human Factors in Computing Systems - Proceedings* (New York, New York, USA, 2013), 3355–3364.

[56] Jokela, T. and Lucero, A. 2014. FlexiGroups: Binding mobile devices for collaborative interactions in medium-sized groups with device touch. *MobileHCI 2014 - Proceedings of the 16th ACM International Conference on Human-Computer Interaction with Mobile Devices and Services* (New York, New York, USA, 2014), 369–378.

[57] Karnik, A., Plasencia, D.M., Mayol-Cuevas, W. and Subramanian, S. 2012. PiVOT: Personalized view-overlays for tabletops. *UIST'12 - Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (2012), 271–280.

[58] Kendon, A. 1990. *Conducting interaction. Patterns of behaviour in focused encounters*.

[59] Kinect - Windows app development: 2020. *https://developer.microsoft.com/en-us/windows/kinect/*. Accessed: 2021-04-29.

[60] Klinkhammer, D., Nitsche, M., Specht, M. and Reiterer, H. 2011. Adaptive Personal Territories for Co-Located Tabletop Interaction in a Museum Setting. *Proceedings of the International Conference on Interactive Tabletops and Surfaces (ITS'11)*. (2011), 107–110. DOI:https://doi.org/10.1145/2076354.2076375.

[61] Ko, K.E. and Sim, K.B. 2018. Deep convolutional framework for abnormal behavior detection in a smart surveillance system. *Engineering Applications of Artificial Intelligence*. 67, October 2017 (2018), 226–234. DOI:https://doi.org/10.1016/j.engappai.2017.10.001.

[62] Langner, R., Brosz, J., Dachselt, R. and Carpendale, S. 2010. PhysicsBox: Playful educational tabletop games. *ACM International Conference on Interactive Tabletops and Surfaces, ITS 2010* (New York, New York, USA, 2010), 273–274.

[63] Ledo, D., Anderson, F., Schmidt, R., Oehlberg, L., Greenberg, S. and Grossman, T. 2017. Pineal: Bringing passive objects to life with embedded mobile devices. *Conference on Human Factors in Computing Systems - Proceedings* (2017), 2583–2593.

[64] Ledo, D., Greenberg, S., Marquardt, N. and Boring, S. 2015. Proxemic-Aware Controls. *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services - MobileHCI '15* (2015), 187–198.

[65] Ledo, D., Houben, S., Vermeulen, J., Marquardt, N., Oehlberg, L. and Greenberg, S. 2018. Evaluation strategies for HCI Toolkit research. *Conference on Human Factors in Computing Systems - Proceedings* (New York, New York, USA, 2018), 1–17.

[66] Lin, J., Jose, S. and Landay, J.A. 2008. Employing Patterns and Layers for Early-Stage Design and Prototyping of Cross-Device User Interfaces. (2008), 1313–1322.

[67] Lucero, A., Holopainen, J. and Jokela, T. 2011. Pass-them-around: Collaborative use of mobile phones for photo sharing. *Conference on Human Factors in Computing Systems - Proceedings* (New York, New York, USA, 2011), 1787–1796.

[68] Lucero, A., Jones, M., Jokela, T. and Robinson, S. 2013. Mobile collocated interactions: Taking an offline break together. *Interactions*. 20, 2 (Mar. 2013), 26–32. DOI:https://doi.org/10.1145/2427076.2427083.

[69] Luff, P. and Heath, C. 1998. Mobility in collaboration. *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. (1998), 305–314. DOI:https://doi.org/10.1145/289444.289505.

[70] Marquardt, N., Ballendat, T., Boring, S., Greenberg, S. and Hinckley, K. 2012. Gradual engagement: Facilitating information exchange between digital devices as a function of proximity. *ITS 2012 - Proceedings of the ACM Conference on Interactive Tabletops and Surfaces* (2012), 31–40.

[71] Marquardt, N., Diaz-Marino, R., Boring, S. and Greenberg, S. 2011. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. *Proceedings of the 24th Annual Symposium on User Interface Software and Technology (UIST'11)* (2011), 315–325.

[72] Marquardt, N., Hinckley, K. and Greenberg, S. 2012. Cross-device interaction via micro-mobility and f-formations. *Proceedings of the 25th annual ACM symposium on User interface software and technology - UIST '12* (New York, New York, USA, 2012), 13.

[73] Marshall, P., Morris, R., Rogers, Y., Kreitmayer, S. and Davies, M. 2011. Rethinking "Multi-User" - An In-The-Wild Study of How Groups Approach a Walk-Up-and-Use Tabletop Interface. *Proceedings of the International Conference on Human Factors in Computing Systems (CHI'11)*. (2011), 3033–3042. DOI:https://doi.org/10.1145/1978942.1979392.

[74] Marshall, P., Rogers, Y. and Pantidi, N. 2011. Using F-formations to analyse spatial patterns of interaction in physical environments. *Cscw 2011*. Cscw (2011), 3033–3042. DOI:https://doi.org/10.1145/1958824.1958893.

[75] Mentis, H., O'Hara, K., Sellen, A. and Rikin Trivedi, R. 2012. Interaction proxemics and image use in neurosurgery. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12).* (2012), 927–936. DOI:https://doi.org/http://dx.doi.org/10.1145/2207676.2208536.

[76] Mentis, H., O'Hara, K., Sellen, A. and Rikin Trivedi, R. 2012. Interaction proxemics and image use in neurosurgery. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12).* (2012), 927–936. DOI:https://doi.org/http://dx.doi.org/10.1145/2207676.2208536.

[77] Miles, M., Huberman, A., Saldana, J. (Arizona S.U. 2014. *Qualitative data analysis: a methods sourcebook*. SAGE Publications, Inc.

[78] Morris, M.R., Huang, A., Paepcke, A. and Winograd, T. 2006. Cooperative gestures: multi-user gestural interactions for co-located groupware. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)* (2006), 1201–1210.

[79] Nebeling, M., Mintsi, T., Husmann, M. and Norrie, M.C. 2014. Interactive development of cross-device user interfaces. *Conference on Human Factors in Computing Systems - Proceedings* (2014), 2793–2802.

[80] Nebeling, M., Teunissen, E., Husmann, M. and Norrie, M.C. 2014. XDKinect: Development framework for cross-device interaction using kinect. *EICS 2014 - Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (2014), 65–74.

[81] Paay, J., Kjeldskov, J. and Skov, M.B. 2015. Connecting in the kitchen: An empirical study of physical interactions while cooking together at home. *CSCW 2015 - Proceedings of the 2015 ACM International Conference on Computer-Supported Cooperative Work and Social Computing* (2015), 276–287.

[82] Paay, J., Kjeldskov, J., Skov, M.B. and O'Hara, K. 2013. F-formations in cooking together: A digital ethnography using YouTube. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 8120 LNCS, PART 4 (2013), 37–54. DOI:https://doi.org/10.1007/978-3-642-40498-6_3.

[83] Peltonen, P., Kurvinen, E., Salovaara, A., Jacucci, G., Ilmonen, T., Evans, J., Oulasvirta, A. and Saarikko, P. 2008. "It's mine, don't touch!": Interactions at a large multi-touch display in a city centre. *Conference on Human Factors in Computing Systems - Proceedings* (2008), 1285–1294.

[84] Perez, P., Roose, P., Cardinale, Y., Dalmau, M., Masson, D. and Couture, N. 2020. A Framework for Developing Proxemic Mobile Applications. *Proceedings of the 2020 Federated Conference on Computer Science and Information Systems, FedCSIS 2020* (2020), 653–656.

[85] Pérez, P., Roose, P., Cardinale, Y., Dalmau, M., Masson, D. and Couture, N. 2020. Mobile proxemic application development for smart environments. *ACM International Conference Proceeding Series* (New York, NY, USA, Nov. 2020), 94–103.

[86] Piper, A.M., O'Brien, E., Morris, M.R. and Winograd, T. 2006. SIDES: A cooperative tabletop computer game for social skills development. *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW* (2006), 1–10.

[87] Platform for Situated Intelligence: *https://github.com/microsoft/psi*. Accessed: 2019-10-30.

[88] Proxemic Interaction: *https://www.iutbayonne.univ-pau.fr/~ppdaza/*. Accessed: 2021-05-09.

[89] Ramakers, R., Anderson, F., Grossman, T. and Fitzmaurice, G. 2016. RetroFab: A design tool for retrofitting physical interfaces using actuators, sensors and 3D printing. *Conference on Human Factors in Computing Systems - Proceedings* (2016), 409–419.

[90] Rector, K., Salmon, K., Thornton, D., Joshi, N. and Morris, M.R. 2017. Eyes-Free Art: Exploring Proxemic Audio Interfaces For Blind and Low Vision Art Engagement. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*. 1, 3 (2017), 1–21. DOI:https://doi.org/10.1145/3130958.

[91] Rick, J., Harris, A., Marshall, P., Fleck, R., Yuill, N. and Rogers, Y. 2009. Children Designing Together on a Multi-Touch Tabletop: An Analysis of Spatial Orientation and User Interactions. *8th International Conference on Interaction Design and Children - IDC '09* (2009), 106–114.

[92] Roggen, D. et al. 2010. Collecting complex activity datasets in highly rich networked sensor environments. *INSS 2010 - 7th International Conference on Networked Sensing Systems*. (2010), 233–240. DOI:https://doi.org/10.1109/INSS.2010.5573462.

[93] Roseman, M. and Greenberg, S. 1996. Building real-time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*. 3, 1 (1996), 66–106. DOI:https://doi.org/10.1145/226159.226162.

[94] Sabri, A.J., Ball, R.G., Fabian, A., Bhatia, S. and North, C. 2007. High-resolution gaming: Interfaces, notifications, and the user experience. *Interacting with Computers*. 19, 2 (2007), 151–166. DOI:https://doi.org/10.1016/j.intcom.2006.08.002.

[95] Salber, D., Dey, A.K. and Abowd, G.D. 1999. The context toolkit: Aiding the development of context-enabled applications. *Conference on Human Factors in Computing Systems - Proceedings* (1999), 434–441.

[96] Samples · microsoft/psi Wiki · GitHub: *https://github.com/microsoft/psi/wiki/Samples*. Accessed: 2021-04-13.

[97] Sarabadani Tafreshi, A.E., Marbach, K. and Norrie, M.C. 2017. Proximity-based adaptation of web content on public displays. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2017), 282–301.

[98] Schipor, O.A., Vatavu, R.D. and Vanderdonckt, J. 2019. Euphoria: A Scalable, event-driven architecture for designing interactions across heterogeneous devices in smart environments. *Information and Software Technology*. 109, (May 2019), 43–59. DOI:https://doi.org/10.1016/j.infsof.2019.01.006.

[99] Schipor, O.A., Vatavu, R.D. and Wu, W. 2019. SAPIENS: Towards software architecture to support peripheral interaction in smart environments. *Proceedings of the ACM on Human-Computer Interaction*. 3, EICS (Jun. 2019), 1–24. DOI:https://doi.org/10.1145/3331153.

[100] Schmidt, C., Müller, J. and Bailly, G. 2013. Screenfinity: Extending the Perception Area of Content on Very Large Public Displays. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*. (2013), 1719. DOI:https://doi.org/10.1145/2470654.2466227.

[101] Serna, A., Pageaud, S., Tong, L., George, S. and Tabard, A. 2016. F-formations and collaboration dynamics study for designing mobile collocation. *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct, MobileHCI 2016* (2016), 1138–1141.

[102] Setti, F., Russell, C., Bassetti, C. and Cristani, M. 2015. F-formation detection: Individuating free-standing conversational groups in images. *PLoS ONE*. 10, 5 (2015), 1–26. DOI:https://doi.org/10.1371/journal.pone.0123783.

[103] SharpOSC: Full implementation of Open Sound Control in C# .NET 3.5: *https://github.com/ValdemarOrn/SharpOSC*. Accessed: 2021-05-17.

[104] Shell, J.S., Selker, T. and Vertegaal, R. 2003. Interacting with groups of computers. *Communications of the ACM*. ACM PUB27 New York, NY, USA.

[105] Singh, A., Kaur, R., Haltner, P., Peachey, M., Gonzalez-Franco, M., Malloch, J. and Reilly, D. 2021. Story CreatAR: a Toolkit for Spatially-Adaptive Augmented Reality Storytelling. *Proceedings of IEEE VR 2021* (2021).

[106] Solera, F., Calderara, S. and Cucchiara, R. 2016. Socially Constrained Structural Learning for Groups Detection in Crowd. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 38, 5 (2016), 995–1008. DOI:https://doi.org/10.1109/TPAMI.2015.2470658.

[107] Sørensen, H., Kristensen, M.G., Kjeldskov, J. and Skov, M.B. 2013. Proxemic interaction in a multi-room music system. *Proceedings of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration, OzCHI 2013* (New York, New York, USA, 2013), 153–162.

[108] Space Syntax Network: 2005. *https://www.spacesyntax.net/*. Accessed: 2021-03-30.

[109] Swing GUI Toolkit Group: *https://openjdk.java.net/groups/swing/*. Accessed: 2021-05-21.

[110] Tafreshi, A.E.S., Marbach, K. and Troster, G. 2018. Proximity Based Adaptation of Content to Groups of Viewers of Public Displays. *International Journal of UbiComp*. 9, 1/2 (2018), 01–09. DOI:https://doi.org/10.5121/iju.2018.9201.

[111] Takashima, K., Aida, N., Yokoyama, H. and Kitamura, Y. 2013. TransformTable: A Self-Actuated Shape-Changing Digital Table. (2013), 179–188.

[112] Tang, A., Tory, M., Po, B., Neumann, P. and Carpendale, S. 2006. Collaborative coupling over tabletop displays. *Proceedings of the SIGCHI conference on Human Factors in computing systems - CHI '06*. (2006), 1181. DOI:https://doi.org/10.1145/1124772.1124950.

[113] Task Class (System.Threading.Tasks) | Microsoft Docs: *https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-5.0*. Accessed: 2021-04-14.

[114] The 7 Steps of Machine Learning: 2017. *https://www.youtube.com/watch?v=nKW8Ndu7Mjw&list=PLIivdWyY5sqJxnwJhe3etaK7utrBiPBQ2&index=2*. Accessed: 2021-02-28.

[115] Threading in Unity - Unity Answers: 2013. *https://answers.unity.com/questions/180243/threading-in-unity.html?_ga=2.22523643.1163458672.1618373117-1748809650.1517507927*. Accessed: 2021-04-14.

[116] Tong, L., Serna, A., Pageaud, S., George, S. and Tabard, A. 2016. It's not how you stand, it's how you move: F-formations and collaboration dynamics in a mobile learning game. *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services, MobileHCI 2016* (2016), 318–329.

[117] Unity Job System: Safe and Easy Multithreading in Unity • Infallible Code: *http://infalliblecode.com/unity-job-system/*. Accessed: 2021-06-11.

[118] Unity Real-Time Development Platform | 3D, 2D VR & AR Engine: 2021. *https://unity.com/*. Accessed: 2021-06-02.

[119] Using threads in Unity3D | universityofgames.net: *https://www.universityofgames.net/using-threads-in-unity-engine/*. Accessed: 2021-06-11.

[120] Vicon | Award Winning Motion Capture Systems: 2019. *https://www.vicon.com/*. Accessed: 2021-04-29.

[121] Villena-Román, J., Collada-Pérez, S., Lana-Serrano, S. and González-Cristóbal, J.C. 2011. Hybrid approach combining machine learning and a rule-based expert system for text categorization. *Proceedings of the 24th International Florida Artificial Intelligence Research Society, FLAIRS - 24* (2011), 323–328.

[122] VIVE^{TM} Canada | Discover Virtual Reality Beyond Imagination: *https://www.vive.com/ca/*. Accessed: 2021-04-29.

[123] Vogel, D. and Balakrishnan, R. 2004. Interactive public ambient displays: Transitioning from implicit to explicit, public to personal, interaction with multiple users. *UIST: Proceedings of the Annual ACM Symposium on User Interface Softaware and Technology* (2004), 137–146.

[124] Volpentesta, A.P. 2015. A framework for human interaction with mobiquitous services in a smart environment. *Computers in Human Behavior*. 50, (2015), 177–185. DOI:https://doi.org/10.1016/j.chb.2015.04.003.

[125] Wang, M., Boring, S. and Greenberg, S. 2012. Proxemic Peddler: A Public Advertising Display that Captures and Preserves the Attention of a Passerby. *Proceedings of the International Symposium on Pervasive Displays (PerDis '12)* (2012), 3–9.

[126] Weiser, M. 1991. The Computer for the 21 st Century. *Scientific American*. 265, September (1991), 94–105.

[127] Wigdor, D. and Wixon, D. 2011. *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture*.

[128] Williamson, J., Li, J., Vinayagamoorthy, V., Shamma, D.A. and Cesar, P. 2021. Proxemics and Social Interactions in an Instrumented Virtual Reality Workshop. (2021), 1–20. DOI:https://doi.org/10.1145/3411764.3445729.

[129] Wu, M. and Balakrishnan, R. 2003. Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays. *Proceedings of the 16th annual ACM symposium on User interface software and technology - UIST '03* (2003), 193–202.

[130] Von Zadow, U., Bösel, D., Dam, D.D., Lehmann, A., Reipschläger, P. and Dachselt, R. 2016. Miners: Communication and awareness in collaborative gaming at an interactive display wall. *Proceedings of the 2016 ACM International Conference on Interactive Surfaces and Spaces: Nature Meets Interactive Surfaces, ISS 2016* (2016), 235–240.

[131] Von Zadow, U. and Dachselt, R. Research on HCI Toolkits and Toolkits for HCI Research: A Comparison.

[132] Zhou, H., Tearo, K., Waje, A., Alghamdi, E., Alves, T., Ferreira, V., Hawkey, K. and Reilly, D. 2016. Enhancing mobile content privacy with proxemics aware notifications and protection. *Conference on Human Factors in Computing Systems - Proceedings* (May 2016), 1362–1373.

[133] Zöllner, M., Jetter, H. and Reiterer, H. 2011. ZOIL: A Design Paradigm and Software Framework for Post-WIMP Distributed User Interfaces. *Distributed User Interfaces: Designing Interfaces for the Distributed Ecosystem*. (2011), 87–94. DOI:https://doi.org/10.1007/978-1-4471-2271-5.

Appendices

Appendix 1: ProxemicUI vs. Microsoft PSI codes

Appendix 1.1: Generating training data using Microsoft PSI

```

/// <summary>
/// For OSC communication setup
/// </summary>
private static readonly int Port = 5103;
private static readonly string AliveMethod = "/osctest/alive";
private static readonly string TestMethod = "/osctest/test";
private static OscServer oscServer;

// -------------------
private OscMessage oscMessage;
private IProducer<string> dataStream;
double minimumThreshold = 0, maximumThreshold = 1.0;
// -------------------

public MainWindow()
{
    InitializeComponent();

    try
    {
        // Use this to receive data from the same machine
        oscServer = new OscServer(TransportType.Udp, IPAddress.Loopback, Port);

        oscServer.FilterRegisteredMethods = false;
        oscServer.RegisterMethod(AliveMethod);
        oscServer.RegisterMethod(TestMethod);
        oscServer.MessageReceived += new EventHandler<OscMessageReceivedEventArgs>(oscServer_MessageReceived);
        oscServer.ConsumeParsingExceptions = false;
        oscServer.Start();
    }
    catch (Exception e)
    {
        Console.WriteLine("{0} Exception caught: ", e);
    }
}

private void oscServer_MessageReceived(object sender, OscMessageReceivedEventArgs e)
{
    oscMessage = e.Message;
    TestProximityEvents();
}

private void TestProximityEvents()
{
    using (var pipeline = Pipeline.Create())
    {
        // Register an event handler to be notified when the pipeline completes
        pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

        // generate data stream form OSC messages
        IProducer<OscMessage> message = Generators.Return(pipeline, oscMessage);

        // use process operator to process messages and test distance
        var output = message.Process<OscMessage, string>(
            (oscMessage, e, o) =>
            {
                // process OSC messages
                int count = oscMessage.Data.Count;

                // to store all entities
                List<Entity> entities = new List<Entity>();

                string timestamp = "";
                // store data of entities, the received format is (timestamp, serial number, x, y, z, yaw, pitch, roll)
```

```
                        if (message.Address == "/trackers/")
                        {
                            for (int i = 1; i < count; i++)
                            {
                                Entity newEntity = new Entity();
                                newEntity.timestamp = (string)message.Data[0];
                                newEntity.ID = (string)message.Data[i];
                                newEntity.X = Convert.ToDouble(message.Data[i+1]);
                                newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                                newEntity.Z = Convert.ToDouble(message.Data[i+3]);
                                newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                                newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                                newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                                i += 6;
                                entities.Add(newEntity);
                            }
                        }

                        // walk through the receved data to test distance
                        if (entities.Count > 1)
                        {
                            foreach(Entity firstEntity in entities)
                            {
                                foreach(Entity secondEntity in entities)
                                {
                                    if (firstEntity.ID != secondEntity.ID)
                                    {
                                        // call DistanceCalculator method to check if entities with in distance
                                        bool inDistance = DistanceCalculator(firstEntity, secondEntity);

                                        // check the result of the DistanceCalculator method
                                        if (inDistance)
                                        {
                                            // post data to the output stream
                                            dataStream = Generators.Return(pipeline, "Pass" + "Relative Distance Test" + firstEntity.ID.ToString() + ", " + secondEntity.ID.ToString() + ", " + Distance.ToString());
                                        }
                                        else
                                        {
                                            // post data to the output stream
                                            dataStream = Generators.Return(pipeline, "Fail" + "Relative Distance Test" + firstEntity.ID.ToString() + ", " + secondEntity.ID.ToString() + ", " + Distance.ToString());
                                        }
                                        // call DataWriter method to write the data to a file to create the dataset
                                        DataWriter();
                                    }
                                }
                            }
                        }
                    });
                pipeline.Run();
            }
        }

        // create dataset
        private void DataWriter()
        {
            using (var pipeline = Pipeline.Create())
            {
                // create a store to write the data to
                var store = PsiStore.Create(pipeline, "data", "D:\\code review user study\\PSI TEST CODE");

                dataStream.Write("dataSet", store);

                pipeline.Run();
            }
        }

        private bool DistanceCalculator(Entity FirstEntity, Entity SecondEntity)
        {
            // calculate positions difference
            double distance = Math.Sqrt(Math.Pow(FirstEntity.X - SecondEntity.X, 2) + Math.Pow(FirstEntity.Y - SecondEntity.Y, 2));

            if (distance >= minimumThreshold && distance <= maximumThreshold)
                return true;
            else
                return false;
        }

        // struct to be used to store data
        public struct Entity
        {
            string id, timestamp;
            double x, y, z, roll, pitch, yaw;
            public string TIMESTAMP
            {
                get { return timestamp; }
                set { timestamp = value; }
            }
            public string ID
            {
                get { return id; }
                set { id = value; }
            }
            public double X
            {
                get { return x; }
                set { x = value; }
            }
            public double Y
            {
                get { return y; }
                set { y = value; }
            }
            public double Z
            {
                get { return z; }
                set { z = value; }
            }
            public double ROLL
            {
                get { return roll; }
                set { roll = value; }
            }
            public double PITCH
            {
                get { return pitch; }
                set { pitch = value; }
            }
            public double YAW
            {
                get { return yaw; }
                set { yaw = value; }
            }
            public Entity(string timestamp, string id, double x, double y, double z,
            double roll, double pitch, double yaw)
            {
                this.timestamp = timestamp;
                this.id = id;
                this.x = x;
                this.y = y;
                this.z = z;
                this.roll = roll;
                this.pitch = pitch;
                this.yaw = yaw;
            }
        }
```

Appendix 1.2: Provide the classifier with low-level proximity data using Microsoft PSI

```

/// <summary>
/// For OSC communication setup
/// </summary>
private static readonly int Port = 5103;
private static readonly string AliveMethod = "/osctest/alive";
private static readonly string TestMethod = "/osctest/test";
private static OscServer oscServer;

// -------------------
private IProducer<List<Entity>> dataStream;
// -------------------

public Main()
{
    InitializeComponent();

    try
    {
        // Use this to receive data from the same machine
        oscServer = new OscServer(TransportType.Udp, IPAddress.Loopback, Port);

        oscServer.FilterRegisteredMethods = false;
        oscServer.RegisterMethod(AliveMethod);
        oscServer.RegisterMethod(TestMethod);
        oscServer.MessageReceived += new EventHandler<
        OscMessageReceivedEventArgs>(oscServer_MessageReceived);
        oscServer.ConsumeParsingExceptions = false;
        oscServer.Start();
    }
    catch (Exception e)
    {
        Console.WriteLine("{0} Exception caught: ", e);
    }
}

private void oscServer_MessageReceived(object sender,
OscMessageReceivedEventArgs e)
{
    using (var pipeline = Pipeline.create())
    {
        // Register an event handler to be notified when the pipeline completes
        pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

        // generate data stream form OSC messages
        IProducer<OscMessage> message = Generators.Return(pipeline, e.Message);

        // use process operator to process messages and test distance
        var output = message.Process<OscMessage, Dictionary<string, List<double
        >>>(
            (oscMessage, e, o) =>
            {
                // process OSC messages
                int count = oscMessage.Data.Count;

                List<Entity> entities = new List<Entity>();

                // store data of entities, the received format is (timestamp,
                serial number, x, y, z, yaw, pitch, roll)
                if (message.Address == "/trackers/")
                {
                    for (int i = 1; i < count; i++)
                    {
                        Entity newEntity = new Entity();
                        newEntity.timestamp = (string)message.Data[0];
                        newEntity.ID = (string)message.Data[i];
                        newEntity.X = Convert.ToDouble(message.Data[i+1]);
                        newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                        newEntity.Z = Convert.ToDouble(message.Data[i+3]);
```

```
                            newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                            newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                            newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                            i += 6;
                            entities.Add(newEntity);
                        }
                    }
                    dataStream = Generators.Return(pipeline, entities);
                }
            pipeline.Run();
        }
    }



    // struct to be used to store data
    public struct Entity
    {
        string id, timestamp;
        double x, y, z, roll, pitch, yaw;
        public string TIMESTAMP
        {
            get { return timestamp; }
            set { timestamp = value; }
        }
        public string ID
        {
            get { return id; }
            set { id = value; }
        }
        public double X
        {
            get { return x; }
            set { x = value; }
        }
        public double Y
        {
            get { return y; }
            set { y = value; }
        }
        public double Z
        {
            get { return z; }
            set { z = value; }
        }
        public double ROLL
        {
            get { return roll; }
            set { roll = value; }
        }
        public double PITCH
        {
            get { return pitch; }
            set { pitch = value; }
        }
        public double YAW
        {
            get { return yaw; }
            set { yaw = value; }
        }
        public Entity(string timestamp, string id, double x, double y, double z, double roll, double pitch, double yaw)
        {
            this.timestamp = timestamp;
            this.id = id;
            this.x = x;
            this.y = y;
            this.z = z;
```

```
                this.roll = roll;
                this.pitch = pitch;
                this.yaw = yaw;
            }
        }
```

Appendix 1.3: Providing the classifier with high-level proximity data using Microsoft PSI

```
        /// <summary>
        /// For OSC communication setup
        /// </summary>
        private static readonly int Port = 5103;
        private static readonly string AliveMethod = "/osctest/alive";
        private static readonly string TestMethod = "/osctest/test";
        private static OscServer oscServer;

        // -------------------
        private OscMessage oscMessage;
        private IProducer<string> dataStream;
        double minimumThreshold = 0, maximumThreshold = 1.0;
        // -------------------

        public MainWindow()
        {
            InitializeComponent();

            try
            {
                // Use this to receive data from the same machine
                oscServer = new OscServer(TransportType.Udp, IPAddress.Loopback, Port);

                oscServer.FilterRegisteredMethods = false;
                oscServer.RegisterMethod(AliveMethod);
                oscServer.RegisterMethod(TestMethod);
                oscServer.MessageReceived += new EventHandler<OscMessageReceivedEventArgs>(oscServer_MessageReceived);
                oscServer.ConsumeParsingExceptions = false;
                oscServer.Start();
            }
            catch (Exception e)
            {
                Console.WriteLine("{0} Exception caught: ", e);
            }
        }

        private void oscServer_MessageReceived(object sender, OscMessageReceivedEventArgs e)
        {
            oscMessage = e.Message;
            TestProximityEvents();
        }

        private void TestProximityEvents()
        {
            using (var pipeline = Pipeline.Create())
            {
                // Register an event handler to be notified when the pipeline completes
                pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

                // generate data stream form OSC messages
                IProducer<OscMessage> message = Generators.Return(pipeline, oscMessage);


                // use process operator to process messages and test distance
                var output = message.Process<OscMessage, string>(
                    (oscMessage, e, o) =>
                    {
                        // process OSC messages
                        int count = oscMessage.Data.Count;

                        // to store all entities
                        List<Entity> entities = new List<Entity>();

                        string timestamp = "";
                        // store data of entities, the received format is (timestamp, serial number, x, y, z, yaw, pitch, roll)
                        if (message.Address == "/trackers/")
                        {
                            for (int i = 1; i < count; i++)
                            {
                                Entity newEntity = new Entity();
                                newEntity.timestamp = (string)message.Data[0];
                                newEntity.ID = (string)message.Data[i];
                                newEntity.X = Convert.ToDouble(message.Data[i+1]);
                                newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                                newEntity.Z = Convert.ToDouble(message.Data[i+3]);
                                newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                                newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                                newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                                i += 6;
                                entities.Add(newEntity);
                            }
                        }

                        // walk through the receved data to test distance
                        if (entities.Count > 1)
                        {
                            foreach(Entity firstEntity in entities)
                            {
                                foreach(Entity secondEntity in entities)
                                {
                                    if (firstEntity.ID != secondEntity.ID)
                                    {
                                        // call DistanceCalculator method to check if entities with in distance
                                        double Distance = DistanceCalculator(firstEntity, secondEntity);

                                        // post data to the output stream
                                        dataStream = Generators.Return(pipeline, "Relative Distance Test" + firstEntity.ID.ToString() + ", " + secondEntity.ID.ToString() + ", " + Distance.ToString());
                                    }
                                }
                            }
                        }
                    });
                pipeline.Run();
            }
        }

        private double DistanceCalculator(Entity FirstEntity, Entity SecondEntity)
        {
            // calculate positions difference
            double distance = Math.Sqrt(Math.Pow(FirstEntity.X - SecondEntity.X, 2) + Math.Pow(FirstEntity.Y - SecondEntity.Y, 2));

            return distance;
        }

        // struct to be used to store data
        public struct Entity
        {
            string id, timestamp;
            double x, y, z, roll, pitch, yaw;
            public string TIMESTAMP
            {
                get { return timestamp; }
                set { timestamp = value; }
            }
            public string ID
            {
                get { return id; }
                set { id = value; }
```

```csharp
        }
        public double X
        {
            get { return x; }
            set { x = value; }
        }
        public double Y
        {
            get { return y; }
            set { y = value; }
        }
        public double Z
        {
            get { return z; }
            set { z = value; }
        }
        public double ROLL
        {
            get { return roll; }
            set { roll = value; }
        }
        public double PITCH
        {
            get { return pitch; }
            set { pitch = value; }
        }
        public double YAW
        {
            get { return yaw; }
            set { yaw = value; }
        }
        public Entity(string timestamp, string id, double x, double y, double z, double roll, double pitch, double yaw)
        {
            this.timestamp = timestamp;
            this.id = id;
            this.x = x;
            this.y = y;
            this.z = z;
            this.roll = roll;
            this.pitch = pitch;
            this.yaw = yaw;
        }
    }
```

Appendix 1.4: Hybrid test using Microsoft PSI

```

        /// <summary>
        /// For OSC communication setup
        /// </summary>
        private static readonly int Port = 5103;
        private static readonly string AliveMethod = "/osctest/alive";
        private static readonly string TestMethod = "/osctest/test";
        private static OscServer oscServer;

        // -------------------
        private OscMessage oscMessage, ExternalEvent;
        private IProducer<string> dataStream;
        double minimumThreshold = 0, maximumThreshold = 1.0;
        // -------------------

        public MainWindow()
        {
            InitializeComponent();

            try
            {
                // Use this to receive data from the same machine
                oscServer = new OscServer(TransportType.Udp, IPAddress.Loopback, Port);

                oscServer.FilterRegisteredMethods = false;
                oscServer.RegisterMethod(AliveMethod);
                oscServer.RegisterMethod(TestMethod);
                oscServer.MessageReceived += new EventHandler<
                OscMessageReceivedEventArgs>(oscServer_MessageReceived);
                oscServer.ConsumeParsingExceptions = false;
                oscServer.Start();
            }
            catch (Exception e)
            {
                Console.WriteLine("{0} Exception caught: ", e);
            }
        }

        private void oscServer_MessageReceived(object sender,
        OscMessageReceivedEventArgs e)
        {
            if (e.Message.Address == "\EntitiesData")
            {
                oscMessage = e.Message;
                TestProximityEvents();
            }
            else
                if (e.Message.Address == "\ExternalEventData")
                {
                    ExternalEvent = e.Message;
                }

        }

        private void TestProximityEvents()
        {
            using (var pipeline = Pipeline.Create())
            {
                // Register an event handler to be notified when the pipeline completes
                pipeline.PipelineCompleted += Pipeline_PipelineCompleted;

                // generate data stream form OSC messages
                IProducer<OscMessage> message = Generators.Return(pipeline, oscMessage);


                // generate data stream from external event
                var externalStream = Generators.Return(pipeline, ExternalEvent);

                // use process operator to process messages and test distance
                var output = message.Process<OscMessage, string>(
                    (oscMessage, e, o) =>
                    {
                        // process OSC messages
                        int count = oscMessage.Data.Count;

                        // to store all entities
                        List<Entity> entities = new List<Entity>();

                        string timestamp = "";
                        // store data of entities, the received format is (timestamp,
                        serial number, x, y, z, yaw, pitch, roll)
                        if (message.Address == "/trackers/")
                        {
                            for (int i = 1; i < count; i++)
                            {
                                Entity newEntity = new Entity();
                                newEntity.timestamp = (string)message.Data[0];
                                newEntity.ID = (string)message.Data[i];
                                newEntity.X = Convert.ToDouble(message.Data[i+1]);
                                newEntity.Y = Convert.ToDouble(message.Data[i+2]);
                                newEntity.Z = Convert.ToDouble(message.Data[i+3]);
                                newEntity.ROLL = Convert.ToDouble(message.Data[i+4]);
                                newEntity.PITCH = Convert.ToDouble(message.Data[i+5]);
                                newEntity.YAW = Convert.ToDouble(message.Data[i+6]);

                                i += 6;
                                entities.Add(newEntity);
                            }
                        }

                        // walk through the receved data to test distance
                        if (entities.Count > 1)
                        {
                            foreach(Entity firstEntity in entities)
                            {
                                foreach(Entity secondEntity in entities)
                                {
                                    if (firstEntity.ID != secondEntity.ID)
                                    {
                                        // call DistanceCalculator method to check if
                                        entities with in distance
                                        bool inDistance = DistanceCalculator(firstEntity
                                        , secondEntity);

                                        // check if both strems (events) are true
                                        var outputStream = inDistance.Join(
                                        externalStream, Reproducible.Nearest(
                                        RelativeTimeInterval.Past()),
                                        (p, s) => { if (primary && secondary)
                                        DrawResponse(); });
                                    }
                                }
                            }
                        }
                    });
                pipeline.Run();
            }
        }

        private void DrawResponse()
        {
            .... Do Something ....
        }

        private bool DistanceCalculator(Entity FirstEntity, Entity SecondEntity)
        {
            // calculate positions difference
            double distance = Math.Sqrt(Math.Pow(FirstEntity.X - SecondEntity.X, 2) +
                Math.Pow(FirstEntity.Y - SecondEntity.Y, 2));

            if (distance >= minimumThreshold && distance <= maximumThreshold)
                return true;
            else
                return false;
        }

        // struct to be used to store data
        public struct Entity
        {
            string id, timestamp;
            double x, y, z, roll, pitch, yaw;
            public string TIMESTAMP
            {
                get { return timestamp; }
                set { timestamp = value; }
            }
            public string ID
            {
                get { return id; }
                set { id = value; }
            }
            public double X
            {
                get { return x; }
                set { x = value; }
            }
            public double Y
            {
                get { return y; }
                set { y = value; }
            }
            public double Z
            {
                get { return z; }
                set { z = value; }
            }
            public double ROLL
            {
                get { return roll; }
                set { roll = value; }
            }
            public double PITCH
            {
                get { return pitch; }
                set { pitch = value; }
            }
            public double YAW
            {
                get { return yaw; }
                set { yaw = value; }
            }
            public Entity(string timestamp, string id, double x, double y, double z,
            double roll, double pitch, double yaw)
            {
                this.timestamp = timestamp;
                this.id = id;
                this.x = x;
                this.y = y;
                this.z = z;
                this.roll = roll;
                this.pitch = pitch;
                this.yaw = yaw;
            }
        }
```

Appendix 2: Example of OSC Communicator Code in Python

```
import triad_openvr
import time
import sys
from datetime import datetime
from pythonosc import osc_message_builder
from pythonosc import udp_client
from pythonosc import osc_message
from pythonosc import dispatcher
from pythonosc import osc_server

oscClient = udp_client.SimpleUDPClient("localhost", 9090)


v = triad_openvr.triad_openvr()
v.print_discovered_objects()

dataSent = False

trackers = v.get_discoverd_trackers()
print (trackers)

stablefile = open("stableEntities.txt", "r")

while (True):
        interval = 1/250
        v.update_devices()
        numberOfTrackers = v.get_number_of_trackers()

        if numberOfTrackers > 0:
            if interval:
                start = time.time()
                trackers = v.get_discoverd_trackers()

                now = datetime.now()
                ts = now.strftime("%H:%M:%S")

                msg = osc_message_builder.OscMessageBuilder(address="/trackers/")
                msg.add_arg(ts)
                for key, value in trackers.items():
                    msg.add_arg(key)
                    msg.add_arg(value)
                    for line in stablefile:
                        for word in line.split():
                            msg.add_arg(word)

                oscClient.send(msg.build())
                msg.build()

                if dataSent == False:
                    file = open("data.txt", "r")
                    for line in file:

                        for word in line.split():
                            massage = osc_message_builder.OscMessageBuilder(address =
                            "/geometry properties/")
                            massage.add_arg(word)

                    oscClient.send(massage.build())
                    massage.build()
                    dataSent = True

                sleep_time = interval-(time.time()-start)
                if sleep_time>0:
                    time.sleep(sleep_time)


```

Appendix 3: Ethics Application for Hackathon User Study

RESEARCH ETHICS BOARDS

APPLICATION FORM

DALHOUSIE UNIVERSITY

Prospective Research

This form should only be used if new data will be collected. For research involving only secondary use of existing information (such as health records, student records, survey data or biological materials), use the *REB Application Form – Secondary Use of Information for Research.*

This form should be completed using the *Guidance for Submitting an Application for Research Ethics Review* available on the Research Ethics website (application instructions).

SECTION 1. ADMINISTRATIVE INFORMATION

[File No: office only]

Indicate the preferred Research Ethics Board to review this research:

[ ] Health Sciences OR [✓] Social Sciences and Humanities

Project Title: **ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays**

1.1 Research team information			
Dalhousie researcher name	**Mohammed Alnusayri**		
Banner #	B00562801	Department	Faculty of Computer Science
Email (@dal)	mh625076@dal.ca	Phone	902-999-5052
Study start date		Study end date	

Co-investigator names and affiliations	Derek Reilly, FCS Joseph Malloch, FCS Hubert Hu, FCS Chinenye Ndulue, FCS Robert Macgregor, FCS			
Contact person for this submission (if not lead researcher)	Name			
	Email		Phone	

1.2 For student submissions:

Degree program	PhD of Computer Science		
Supervisor name and department	Derek Reilly, Faculty of Computer Science		
Supervisor Email (@dal)	reilly@cs.dal.ca	Phone	902-494-4057

Department/unit ethics review (if applicable). **Undergraduate minimal risk research only**.

Attestation: [ ] I am responsible for the unit-level research ethics review of this project and it has been approved.

Authorizing name:

Date:

1.3 Other reviews:

Other ethics reviews (if any)		Where	Status
Funding, if any (list on consent form)	Agency		
	Award Number		
Peer review (if any)			

1.4 Attestation(s). The appropriate boxes *must* be checked for the submission to be accepted by the REB)

[√] I am the **lead researcher**. I agree to conduct this research following the principles of the Tri-Council Policy Statement *Ethical Conduct for Research Involving Humans* (TCPS) and consistent with the University *Policy on the Ethical Conduct of Research Involving Humans*.

I have completed the TCPS Course on Research Ethics (CORE) online tutorial.

[√] Yes [ ] No

For Supervisors (of student / learner research projects):

[√] I am the **supervisor** for this research named in section 1.2. I have reviewed this submission, including the scholarly merit of the research, and believe it is sound and appropriate. I take responsibility for ensuring this research is conducted following the principles of the TCPS and University Policy.

I have completed the TCPS Course on Research Ethics (CORE) online tutorial.

[√] Yes [ ] No

SECTION 2. PROJECT DESCRIPTION

2.1 Lay summary

2.1.1 In lay language, describe the rationale, purpose, study population and methods. Include the background information or literature to contextualize the study. Mention what new knowledge is anticipated, and whether this is a pilot project or fully developed study. [500 words]

Interactive display applications can be designed based on three types of territories: a single territory for all users (for collaborative use), personal territories for each user (for independent use), or a mix of personal and public territories (to support a mix of collaborative and independent use). These different designs are chosen based on the type of task coupling that being performed using the interactive display as well as different user roles, both of which can impact group formation and dynamics. Tightly coupled tasks require working together to complete the task, lightly coupled tasks involve working independently to complete a higher level task, while uncoupled tasks involve working independently on tasks that are not directly related. A number of abstract roles have been identified in the literature that pertain to an individual's relation toward work being done on the interactive display: direct participant, active observer, passive observer, and disengaged bystander. Therefore, large interactive displays might be used by individuals and groups simultaneously, and the roles of users, a group's size and formation around an interactive display might change over time. In addition, there are a number of interactive display systems that employed the use of relative proximity of individuals to interactive displays and to others. Proximity can be defined as how a person uses the surrounding area, which is divided into: intimate space, personal space, social space, and public space. Considering these concepts, we developed and evaluated a kiosk application (ProximityTable [1]) that can detect and respond to groups and individuals behaviors (combine personal work spaces into a single large workspace as a response for creating a group). Based on the literature review and implementation and evaluation of ProximityTable we defined four core requirements for proxemics-aware interactive display applications. These requirements are: entities, which is the base of a proxemics-aware application, association rules between entities to measure and understand proxemic relationships between entities; these rules include: Relative Proximity Rules (e.g. test relative distance), Compound Proximity Rules (e.g. test distance and orientation), and Hybrid Rules (combine proxemic-rules with UI events). By looking into existing toolkits (e.g. Proximity toolkit [2]), we can see such toolkits provides only basic events based on single proxemic rule (e.g. distance, orientation) and between two proximity entities. Therefore, based on the four requirements defined above, we developed ProxemicUI, a proximity-based event model and software framework to support the implementation of interactive display applications that respond to the position and orientation of individuals relative to the display and each other.

In this study, we want to validate our concept through exploring the design space of how ProxemicUI can be used in real settings. The study will take place over the course of 2.5 days (a weekend: Friday 5:00-8:00 pm and Saturday and Sunday 9:00 am – 5:00 pm) and it will be conducted in the GEM lab, Mona Campbell, Dalhousie University. The study will be organized in the form of a "hackathon", a block of time over which teams move from idea through to implementation, testing, and demonstration. We are planning to recruit a minimum of four and maximum of six groups of three (12-18 participants), who know how to code with C#. Recruitment will take place through ShiftKey Labs channels (email and webpage for the event) and posting the hackathon ad on ad boards on the Dalhousie campus.

Developers will use ProxemicUI to control a number of home appliances (e.g. lamp, thermostat) and interactive displays (e.g. tabletop, tablet) based on proxemic data (e.g. turn on lamp when entering a room). The study will start with a mini-workshop to provide an overview of the framework to participants. This includes a training session where participants will be given a guided experience of using the framework. Participants then will start using the framework to create a smart home application. At the end of each development session, participants will participate in a group discussion to discuss the pros and cons of the framework and ways to improve it. At the end of the study, two winning teams will be announced, and prizes will be awarded (the prizes are Amazon Echo Dots, a voice-controlled smart-home device, for each member in the first and second place teams); it is common to give prizes for the best work at the end of hackathon events.

During the study, development sessions, discussion sessions, and final presentations will be videotaped. Source code directories will be copied and a recording of each team's development process will be screen captured as well.

References:

[1] Gang Hu, Derek Reilly, Mohammed Alnusayri, Ben Swinden, and Qigang Gao. 2014. DT-DT: Top-down Human Activity Analysis for Interactive Surface Applications. In Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS '14), 167-176. DOI=10.1145/2669485.2669501 http://doi.acm.org/10.1145/2669485.2669501

[2] Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. 2011. The proximity toolkit: prototyping proxemic interactions in ubiquitous computing ecologies. In Proceedings of the 24th annual ACM symposium on User interface software and technology (UIST '11), 315-326. DOI=10.1145/2047196.2047238 http://doi.acm.org/10.1145/2047196.2047238

2.1.2 If a phased review is being requested, describe why this is appropriate for this study, and which phase(s) are included for approval in this application.

[√] Not applicable

2.2 Research question

State the hypotheses, the research questions or research objectives.

Our study addresses the following research objective:

1- How well can the framework support the implementation of different proxemic-aware interactive applications?

In addition, the study explores the following research questions:

1- What type and amount of proxemic rules do the developers use?
2- With what type of interactions are Hybrid Rules used? Were there any unexpected ways of using these?
3- Under what circumstances are Compound Rules used? Were there any unexpected ways of using these? How complex are the composed rules?
4- What difficulties do developers face when using ProxemicUI?
5- In which ways can the framework be improved?

2.3 Recruitment

2.3.1 Identify the study population. Describe how many participants are needed and how this was determined.

The study population will be a minimum of 12 developers (four groups of three), and no more than 18 developers (six groups of three), who will be recruited through ShiftKey Labs, Computer Science, Dalhousie University. ShiftKey Labs was chosen because we can contact a large number of developers during the recruitment process, and our study will be conducted as a hackathon, which Shiftkey has experience promoting.

2.3.2 Describe recruitment plans and append recruitment instruments. Describe who will be doing the recruitment and what actions they will take, including any screening procedures. Describe and justify any inclusion / exclusion criteria.

The main investigator Mohammed Alnusayri will do the recruitment through ShiftKey Labs channels. A copy of the recruitment notice is included in appendix A. In the recruitment notice, it is clearly states that to be eligible to participate in the study, participants must have programming skills and know how to code with C# (C# was chosen because the framework was built with it). If anyone wishes to participate, they will indicate so on a Google Form (through ShiftKey Labs website). The main investigator then contacts them through email to explain the study in detail and verify that they are willing to participate. Participants will give informed consent at the outset of the event by signing an informed consent form (detailed below).

2.3.3 Describe any community or organizational permissions needed to recruit your participants (attach support letters). Describe any other community consent or support needed to conduct this research. (If the research involves Aboriginal participants, please complete section 2.10).

[] Not applicable

Participants will be recruited through the ShiftKey Labs channels. Permissions from the labs manager is in Appendix C.

2.4 Informed consent process

2.4.1 Describe the informed consent process, including any plans for ongoing consent (how and when the research will be described to prospective participants, by whom, how the researcher will ensure prospective participants are fully informed). If non-written consent is proposed, describe the process. Address how any third party consent (with or without assent) will be managed. Append copies of all consent/assent documents, including oral consent scripts.

If participants are interested in the hackathon, they will indicate so through the ShiftKey Labs website (on a Google Form). The advertisement for the hackathon will clearly indicate that it is being conducted for research purposes. The main researcher Mohammed Alnusayri will contact them through email to explain the study in detail and confirm their willingness to participate. At the outset of the study participants will sign the consent form (Appendix D). The informed consent outlines the risks and benefits associated with the study, a description of the study, the participant's right to withdraw without consequence, and assurances of confidentiality and anonymity of personal data.

2.4.2 Discuss how participants will be given the opportunity to withdraw (their participation and/or their data) and any limitations on this.

[ ] Not applicable

Participants will be informed that they can choose to withdraw their participation at any point of the study.

2.4.3 If an exception to the requirement to seek prior informed consent is sought, address the criteria in TCPS article 3.7A.

[✓] Not applicable

2.5 Methods and analysis

2.5.1 Describe the study design, where the research will be conducted, what participants will be asked to do and the time commitment, what data will be recorded using what research instruments (append copies).

The study will take place over the course of 2.5 days (a weekend), where participants will use the framework (ProxemicUI) to control a number of home appliances (e.g. lamp, stereo, thermostat) and interactive displays (tabletop, wall display, tablets, phones) based on proxemic data (e.g. when I sit on a living room couch or chair, turn on the TV and stereo and launch the remote control app on my phone). The study environment will be the GEM lab, which will have one large area set up like a living space, and separate breakaway areas for groups to code in. We also will have the DT-DT tracking system ready for participants' use.

The study procedure will be as follows:

- Friday evening (5:00 – 8:00 pm): mini-workshop to provide an overview of the framework and it features. The researcher will explain to participants how to use the framework and inform them about its features (e.g. examining existing rules and demonstrating how to add new rules). The framework includes a set of different proxemic-events using different proxemic rules and UI events. The researcher also will show some examples of proxemic-aware applications to give users an idea about how the framework can be useful. These examples extracted from the main investigator previous work as well as the literature. The workshop will end with a hands-on lab session to give the participants guided experience using the framework.

- Saturday and Sunday (9:00 am – 5:00 pm): participants will be working with the framework to create a smart home application by controlling several appliances and making them proxemic-aware and responsive. Development sessions will be videotaped, and all coding will be captured with screen recording software. At 4pm on Saturday and 3pm on Sunday groups will participate in a discussion session to comment on pros and cons of the framework as well as to offer suggestions for improvement. Starting at 4pm on Sunday, each group will present their work to a jury consisting of Computer Science faculty members working in the area of Human-Computer Interaction and Internet of Things. The jury will be free to ask questions during this time. The discussions and presentations will be videotaped. After this, the winning team and runner up will be announced, and prizes will be awarded. Questions for the group discussions are in appendix G.

To make sure we recruit groups of three, the recruitment notice makes it clear that we are looking for groups of three, but individuals are welcome to participate. In case we have individuals, we will match them into groups of three. If a participant decides to withdraw from the study, his/her group will be given the option to continue the study or withdraw as well. In either case, we will not use any collected data up to the point where the participant(s) decided to withdraw from the study.

The main investigator Mohammed Alnusayri (the developer of the framework) will be available through the weekend to answer questions and support the groups while using the framework. Dr. Derek Reilly will provide oversight during the event. Investigators Dr. Joseph Malloch, Hubert Hu, Chinenye Ndulue, and Robert Macgregor will help during the study to guide participants and to control the study flow.

Discussion and hacking sessions will be video recorded, source code will be collected, and screen capture software will capture the process of the groups. We will review them to understand developers' behaviors when using the framework. Video recording of hacking session will be transcribed, and the prototypes designed and built over the weekend will be annotated. Source code will be reviewed and annotated as well to understand different coding behaviors and build common themes between groups.

This research will take place in the GEM lab, fourth floor, Mona Campbell building, Dalhousie University.

[ ] This is a clinical trial (physical or mental health intervention) – ensure section 2.11 is completed

2.5.2 Describe plans for data analyses.

The main purpose of this study is to establish the systems contribution of the ProxemicUI framework: that is, to establish that developers with limited exposure to the kinds of applications supported by the framework are nonetheless able to rapidly prototype using it.

We will be looking carefully at how two key innovations of the framework are used: Hybrid Rules (connecting tracking data with user interface events), and Compound Rules (composing more complex rules from basic building blocks).

To accomplish this we will be collecting and analyzing a variety of data: video recording of the development process, group discussions/reflections, screen capture of a testing computer (that we provide), final demo presentations and critiques, and source code.

All video will be transcribed and annotated using an open coding process to identify themes during development and in group discussion. Source code and screen capture will be reviewed and annotated to itemize what elements of the framework were used, what syntactical or semantic errors occurred, and how fluency with the framework emerged over the course of the weekend. The group discussions will also be used to obtain detailed feedback and suggestions about the framework.

The resulting applications produced by the groups will also be annotated and described to provide evidence of the framework's utility.

2.5.3 Describe any compensation that will be given to participants and how this will be handled for participants who do not complete the study. Discuss any expenses participants are likely to incur and whether/how these will be reimbursed.

Since we are running a hackathon, at the end of the study we will announce first and second place teams and award prizes of Echo Dots for each member of the winning teams. The winners will be determined by a panel of experts, who will provide critiques of each application. The judging session and the critiques will be collected and used in analysis. The panel members will be made aware of this beforehand and will provide informed consent (see Appendix F).

2.5.4 Describe and justify any use of deception or nondisclosure and explain how participants will be debriefed.

[✓] Not applicable

2.5.5 Describe the role and duties of local researchers (including students and supervisors) in relation to the overall study. Identify any special qualifications represented on the team relevant to the proposed study (e.g. professional or clinical expertise, research methods, experience with the study population, statistics expertise, etc.).

Dr. Derek Reilly is a faculty member in Computer Science, who is the PhD advisor for the main investigator Mohammed Alnusayri. Mohammed has developed the framework and the study design under the direction of Dr. Reilly. Investigators Dr. Joseph Malloch, Hubert Hu, Chinenye Ndulue, and Robert Macgregor will help during the study to guide participants and to control the study flow. They will help in data analysis as well.

2.6 Privacy & confidentiality

2.6.1 Describe any provisions for ensuring privacy and confidentiality (or anonymity). Describe who will have access to data and why, how data will be stored and handled in a secure manner, how long data will be retained and where. Discuss any plans for data destruction and/or de-identification.

[ ] This research involves personal health records (ensure section 2.12 is completed)

All audio recording and source codes will be stored, accessed, and processed on a secure computer (password protected) accessible only to the researchers associated with the project. After analysis is completed, all data will be kept for three years in a secure location as a future reference for publications, then it will be deleted.

2.6.2 Describe how participant confidentiality will be protected when research results are shared. Discuss whether participants will be identified (by name or indirectly). If participants will be quoted address consent for this, including whether quotes will be identifiable or attributed.

To ensure anonymity of all participants, all data collected will be treated anonymously by using pseudonyms. When a participant agrees to do the study, they will be assigned a participant identification number (e.g., P1, P2, etc.). With the exception of the consent form, any time the participant is referred to in any document (e.g., study logs, reports, papers, etc.) only the participant ID will be used. Consent forms will be kept separate from the ID numbers.

2.6.3 Address any limits on confidentiality, such as a duty to disclose abuse or neglect of a child or adult in need of protection, and how these will be handled. Detail any such limits in consent documents.

[√] Not applicable

2.6.4 Will any information that may reasonably be expected to identify an individual (alone or in combination with other available information) be accessible outside Canada? This includes sharing information with team members, collecting data outside Canada, use of survey companies, use of software.

[√] No

[ ] Yes. If yes, describe how you comply with the University *Policy for the Protection of Personal Information from Access Outside Canada*, such as securing participant consent and/or securing approval from the Vice President Research.

2.7 Provision of results to participants

2.7.1 The TCPS encourages researchers to share study results with participants in appropriate formats. If you plan to share study results with participants, discuss the process and format.

[ ] Not applicable

If a participant requests (e.g., on the consent form) to see the results, we will provide publication details once complete. Participants will be able to see aggregate results in the publications. Publications will provide context to the study and results (e.g., the motivation of the study and why the topic is important).

2.7.2 If applicable, describe how participants will be informed of any incidental findings – unanticipated results (of screening or data collection) that have implications for participant welfare (health, psychological or social).

[√] Not applicable

2.8 Risk & benefit analysis

2.8.1 Discuss what risks or discomforts are anticipated for participants, how likely risks are and how risks will be mitigated. Address any particular ethical vulnerability of your study population. If applicable, address third party or community risk. Risks to privacy from use of identifying information should be addressed.

There are very low risks associated with this study. There is a low risk that some users may feel uncomfortable if they experience some difficulty using the framework during the study but there is a researcher will always be available during the study to answer any questions.

2.8.2 Identify any direct benefits of participation to participants (other than compensation), and any indirect benefits of the study (e.g. contribution to new knowledge)

There are a number of direct benefits for participants taking part in this research project: they will experience creating a smart home environment, having fun in the competition, using novel framework, and finally possibility of winning the prizes. An indirect benefit is the opportunity to advance research knowledge and potentially benefit others. In addition, participants will be able to keep a copy of their source code; IP over what the teams develop using the framework will be shared equally between the team and the designers and implementers of the software framework (Mohammed Alnusayri and Derek Reilly). This is made explicit in the informed consent form.

2.9 Conflict of interest

Describe whether any dual role or conflict of interest exists for any member of the research team in relation to potential study participants (e.g. TA, fellow student, teaching or clinical relationship), and/or study sponsors, and how this will be handled.

[√] Not applicable

2.10 Research with Aboriginal peoples

[√] Not applicable – go to 2.11

2.10.1 If the proposed research involves Aboriginal peoples, describe the plan for community engagement (per TCPS Articles 9.1 and 9.2). Attach supporting letters, research agreements and other relevant documents, if available. If community engagement is not sought, explain why the research does not require it, referencing article 9.2.

2.10.2 State whether ethical approval has been or will be sought from Mi'kmaw Ethics Watch or other Indigenous ethics review group(s), and if not, why the research does not fall under their purview.

2.10.3 Describe any plans for returning results to the community and any intellectual property rights agreements negotiated with the community, with regard to data ownership. If there are specific risks to the community involved, ensure these have been addressed in section 2.8.1.

2.11 Clinical trials

[√] Not applicable – go to 2.12

2.11.1 Does the proposed research require clinical trial registration, in keeping with national and international regulations?

[ ] No. Please explain why not.

[ ] Yes. Please indicate where it was registered and provide the registration number.

2.11.2 If a novel intervention or treatment is being examined, describe standard treatment or intervention, to indicate a situation of clinical equipoise exists (TCPS Chapter 11). If placebo is used with a control group rather than standard treatment, please justify.

2.11.3 Clearly identify the known effects of any product or device under investigation, approved uses, safety information and possible contraindications. Indicate how the proposed study use differs from approved uses.

[ ] Not applicable

2.11.4 Discuss any plans for blinding/randomization.

2.11.5 What plans are in place for safety monitoring and reporting of new information to participants, the REB, other team members, sponsors, and the clinical trial registry? These should address plans for removing participants for safety reasons, and early stopping/unblinding/amendment of the trial. What risks may arise for participants through early trial closure, and how will these be addressed? Are there any options for continued access to interventions shown to be beneficial?

2.12 Use of personal health information

[√] Not applicable

2.12.1 Describe the personal health information required and the information sources, and explain why the research cannot reasonably be accomplished without the use of that information. Describe how the personal health information will be used, and in the most de-identified form possible.

2.12.2 Will personal health information be combined with information from other sources to form a composite record (data linkage)? Will the research create individually identifying health information by combining information from two or more databases without the consent of the individuals who are the subjects of the information (data matching)?

[ ] No.

[ ] Yes. Describe the other information and how linkage will be conducted, and/or why data matching is required.

2.12.3 Describe reasonably foreseeable risks to privacy and how these will be mitigated.

SECTION 3. APPENDICES

3.1 Appendices Checklist. Append all relevant material to this application. This may include:

[✓] Recruitment documents (posters, oral scripts, online postings, invitations to participate, etc.)

[ ] Screening documents

[✓] Consent/assent documents or scripts

[✓] Research instruments (questionnaires, interview or focus group questions, etc.)

[ ] Debriefing forms

[ ] Permission letters (Aboriginal Band Council, School Board, Director of a long-term care facility)

[✓] Support letters

3.2 Consent Form

Sample consent forms are provided on the Research Ethics website and may be used in conjunction with the information in the *Guidance* document to help you develop your consent form.

Appendix A – Recruitment Notice

We are recruiting participants to take part in a research study to evaluate our framework (ProxemicUI) that is a proximity-based event model and software framework to support the implementation of interactive display applications that respond to the position and orientation of individuals relative to the display and each other. We are recruiting groups of three who have programming skills, know how to code with C#. We encourage you to join our study with your friends or colleagues as a group of three, but individuals are welcome as well (we will match individual users into groups of three).

If you agree to participate, use the following link (google form link) to sign up for the study. When sign up, the main researcher Mohammed Alnusayri will contact you to explain the study in detail and to give consent to do the study.

The study will take place over a weekend (Friday 5:00 - 8:00pm, Saturday and Sunday 9:00 am - 5:00pm) in the GEM lab, Mona Campbell Building at Dalhousie University. You will be using our framework (ProxemicUI) to control home appliances apps and make them proxemic-aware; and then reflect on the experience in a group discussion. During the study, development sessions, discussion sessions, and final presentations will be videotaped. Source codes will be collected and screen recording of developing process will be screen captured as well. Your source code will be collected as well to help us understand your behaviors when using the framework. You are going to work with a minimum of four different teams and by the end of the study we will pick first and second place teams. an Echo Dot for each member of both first and second places. participants will be able to keep a copy of their source code; and IP over what participants developed will be shared between Mohammed Alnusayri, Derek Reilly, and the team.

Appendix B – Hackathon Ad Copy

It's the one-and-only Smart Home Track-a-thon !

Have you dreamed of making your toaster alive and interactive? We've all *heard* about smart homes, now here's your chance to *build* one!

The Graphics and Experiential Medial Lab and ShiftKey Labs in the Faculty of Computer Science have teamed up to offer a unique hackathon event where teams will use state-of-the-art software that tracks people and connects devices, allowing you to develop innovative smart home applications.

Here are just a few of the appliances that you can work with and some of the possible intelligence you can add to them:

Lamp: e.g. turn the lamp on when you enter the room.

TV: e.g. automatically launch a remote control app on your phone when you sit on the couch.

Dining room table: e.g. select a decorative pattern based on who is seated at the table

… and of course, many opportunities to make the **Toaster** intelligent!

Event schedule:

Friday (5:00 – 8:00 pm): mini-workshop and tutorial.

Saturday (9:00 am – 4:00 pm): development session

Saturday (4:00 pm – 5:00 pm): group discussion

Sunday (9:00 am – 3:00 pm): development session

Sunday (3:00 pm – 4:00 pm): group discussion

Sunday (4:00 pm – 5:00 pm): group presentations and announcing winners.

Lunch will be served on Saturday and Sunday.

The event is open to all students enrolled in a postsecondary program, who have programming skills and know how to code with C#.

You will form teams of three (you can either register as a team or ask to be placed in one). Your final creations will be judged by an expert panel at the end of the weekend. Each member of first and second place teams will win an Echo Dot! All participants will be keep a copy of their source code, and IP over what you build with the framework will be shared between your team and the software framework developers.

Why should you participate?

1. For the fun! Work with your peers to build cool applications using cool technology.
2. For the experience! Hackathons look great on your resume.
3. For the prizes! Need we say more?
4. For science! The GEM Lab will collect and review your experiences during the hackathon to understand how to better support rapid prototyping for smart home applications.

where: GEM lab, 4th floor, Mona Campbell building, Dalhousie University

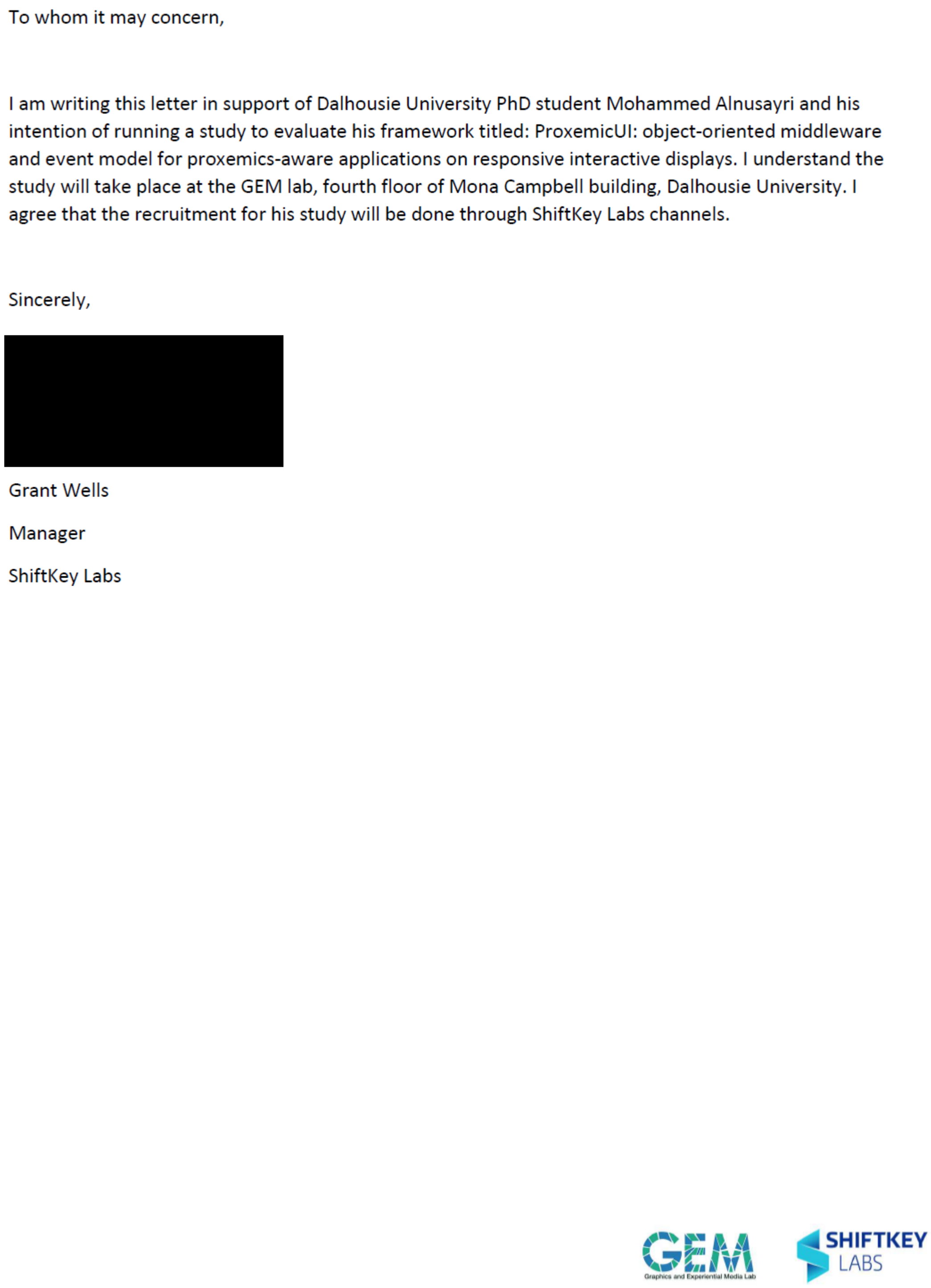
when: date

Hosted by:

GEM Lab (gem.cs.dal.ca)

Shiftkey Labs (shiftkeylabs.ca)

Appendix C – Supporting Letter



To whom it may concern,

I am writing this letter in support of Dalhousie University PhD student Mohammed Alnusayri and his intention of running a study to evaluate his framework titled: ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays. I understand the study will take place at the GEM lab, fourth floor of Mona Campbell building, Dalhousie University. I agree that the recruitment for his study will be done through ShiftKey Labs channels.

Sincerely,

[redacted]

Grant Wells

Manager

ShiftKey Labs

GEM
Graphics and Experiential Media Lab

SHIFTKEY
LABS

Appendix D – Informed Consent for participants

ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

Principal Investigators: Mohammed Alnusayri, Faculty of Computer Science

Dr. Derek Reilly, Faculty of Computer Science

Joseph Malloch, Faculty of Computer Science

Hubert Hu, Faculty of Computer Science

Chinenye Ndulue, Faculty of Computer Science

Robert Macgregor, Faculty of Computer Science

Contact Person: Mohammed Alnusayri, Faculty of Computer Science, mh625076@cs.dal.ca

We invite you to take part in a research study being conducted by Mohammed Alnusayri at Dalhousie University. Your participation in this study is voluntary and you may withdraw from the study at any time. Your academic (or employment) performance evaluation will not be affected by whether or not you participate. To be eligible to participate in the study, you must have programming skills, know how to code with C# and be 18 or older. We encourage you to join our study with your friends or colleagues as a group of three, but individuals are welcome as well (we will match individuals into groups of three). The study is described below. There are a number of direct benefits for participants taking part in this research project: you will experience creating a smart home environment, have fun in the competition, use a novel software framework, and possibly win a prize. An indirect benefit is the opportunity to advance research knowledge and potentially benefit others. In addition, you will be able to keep a copy of your source code and IP over what is developed will be shared between your teammates and the software framework developers.

The purpose of the study is to evaluate our framework (ProxemicUI) and to understand how developers can benefit from it. You will be asked to participate in 2.5 days (a weekend: Friday 5:00-8:00 pm and Saturday and Sunday 9:00am–5:00pm) study where you will use the framework to control home appliances apps and make them proxemic-aware; and then reflect on the experience in a group discussion. You and two other participants will be working as a team throughout the study and group discussion will be audio recorded.

Your team will be competing with other three teams and at the end of the weekend we will pick two winning teams: the prize for each member of first and second place teams is an Echo Dot. You can withdraw from the study at any time without consequence.

When you sign up for the study, you will meet with an investigator (in the Mona Campbell building). At this initial meeting you will be asked to give consent to do the study. You will be given a general description of the study procedure. At the beginning of the study, you will participate in a mini-workshop where you will learn how to use the framework. The rest of the weekend will be spent on developments to create smart home environment. After each developing session, you will participate in a group discussion about you experience using the framework. At the end of the study, you will present your work to a jury consists of Computer Science faculty members. The jury then will announce two winners of the hackathon.

During the study, development sessions, discussion sessions, and final presentations will be videotaped. Source codes will be collected and screen recording of developing process will be screen captured as well. All personal and identifying data will be kept confidential. Anonymity of textual data will be preserved by using pseudonyms (e.g., an ID number) to ensure your confidentiality. The informed consent form and all research data will be kept in a secure location under confidentiality in accordance to University policy for three years post publication.

In the event that you have any difficulties with, or wish to voice concern about, any aspect of your participation in this study, you may contact Catherine Connors, Director, Office of Research Ethics Administration at Dalhousie University's Office of Human Research Ethics for assistance: phone: (902) 494-1462, email: Catherine.connors@dal.ca.

"I have read the explanation about this study. I have been given the opportunity to discuss it and my questions have been answered to my satisfaction. ***I understand that being video taped is necessary to participate in the study.*** *I hereby consent to take part in the study. However, I understand that my participation is voluntary and that I am free to withdraw from the study at any time."*

Participant Researcher

Name: ________________________________ Name: ________________________________
Signature: _____________________________ Signature: _____________________________
Date: _________________________________ Date:
____

Please select **one** of the options below:

> *"I agree to let you directly quote any comments or statements made in any written reports without viewing the quotes prior to their use and I understand that the anonymity of textual data will be preserved by using pseudonyms."*
>
> Participant Researcher
>
> Name: ____________________________ Name:
> ____________________________
>
> Signature: ____________________________
> Signature:____________________________
> ***Date:*** ____________________________
> ***Date:*** ____________________________
>
> ***<u>Or</u>***
>
> *"I want to read direct quotes prior to their use in reports and I understand that the anonymity of textual data will be preserved by using pseudonyms."*

[if this option is chosen, please include a contact email address: ______________________]

Participant   Researcher

Name: ________________________________   Name: __________________________

Signature: _____________________________   Signature:________________________

Date:* ________________________________**   ***Date:

____

If you are interested in seeing the results of this study, please check below and provide your email address. We will contact you with publication details that describe the results.

"I would like to be notified by email when results are available via a publication."

[if this option is chosen, please include a contact email address: ______________________]

Appendix E – Informed Consent for jury

ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

Principal Investigators: Mohammed Alnusayri, Faculty of Computer Science

Dr. Derek Reilly, Faculty of Computer Science

Joseph Malloch, Faculty of Computer Science

Hubert Hu, Faculty of Computer Science

Chinenye Ndulue, Faculty of Computer Science

Robert Macgregor, Faculty of Computer Science

Contact Person: Mohammed Alnusayri, Faculty of Computer Science, mh625076@cs.dal.ca

We invite you to be a member of the jury that will evaluate developers' performance in a research study conducted by Mohammed Alnusayri at Dalhousie University. Your participation in this study is voluntary and you may withdraw from the study at any time. Your academic (or employment) performance evaluation will not be affected by whether or not you participate. Participating in the study might not benefit you, but we might learn things that will benefit others. You should discuss any questions you have about this study with Mohammed Alnusayri.

The purpose of the study is to evaluate our framework (ProxemicUI) and to understand how developers can benefit from it. We are going to run a hackathon where we have a number of developer use the framework to create a smart home environment by controlling home appliances apps and make them proxemic-aware. The study will take place over the weekend of (date) and continue through the weekend (2.5 days) in the GEM lab, Mona Campbell building, Dalhousie University. Starting at 4pm on Sunday, each group will present their work to the jury members, where you and the rest of the jury members are free to ask questions during this time. The discussions and presentations will be videotaped. After this, the winning team and runner up will be announced, and prizes will be awarded.

In the event that you have any difficulties with, or wish to voice concern about, any aspect of your participation in this study, you may contact Catherine Connors, Director, Office of Research Ethics Administration at Dalhousie University's Office of Human Research Ethics for assistance: phone: (902) 494-1462, email: Catherine.connors@dal.ca.

"I have read the explanation about this study. I have been given the opportunity to discuss it and my questions have been answered to my satisfaction. ***I understand that being video taped is necessary to participate in the study.*** *I hereby consent to take part in the study. However, I understand that my participation is voluntary and that I am free to withdraw from the study at any time."*

Participant **Researcher**

Name: ________________________________ Name: ________________________________
Signature: _____________________________ Signature: _____________________________
Date: _________________________________ Date:
____

Please select **one** of the options below:

"I agree to let you directly quote any comments or statements made in any written reports without viewing the quotes prior to their use and I understand that the anonymity of textual data will be preserved by using pseudonyms."

Participant Researcher

Name: _____________________________ Name:
____

Signature: __________________________
Signature:_____________________________
Date: _____________________________
Date:______________________________

Or

"I want to read direct quotes prior to their use in reports and I understand that the anonymity of textual data will be preserved by using pseudonyms."

[if this option is chosen, please include a contact email address:
__________________________]

Participant Researcher

Name: ______________________________ Name: _________________________

Signature: ___________________________ Signature:_______________________

Date: _____________________________ ***Date:***
____

If you are interested in seeing the results of this study, please check below and provide your email address. We will contact you with publication details that describe the results.

"I would like to be notified by email when results are available via a publication."

[if this option is chosen, please include a contact email address:____________________]

Appendix F – invitation for jury

Dear Dr. (faculty member name)

My name is Mohammed Alnusayri, a PhD student in Computer Science at Dalhousie University and under the supervision of Dr. Derek Reilly. We have developed a framework (ProxemicUI) that is a proximity-based event model and software framework to support the implementation of interactive display applications that respond to the position and orientation of individuals relative to the display and each other. We are going to evaluate the framework through running a hackathon where we have a number of developer use the framework to create a smart home environment by controlling home appliances apps and make them proxemic-aware. The study will take place over the weekend of (date) and continue through the weekend (2.5 days) in the GEM lab, Mona Campbell building, Dalhousie University. Starting at 4pm on Sunday, each group will present their work to a jury consisting of Computer Science faculty members working in the area of Human-Computer Interaction and Internet of Things. The jury will be free to ask questions during this time. The discussions and presentations will be videotaped. After this, the winning team and runner up will be announced, and prizes will be awarded.

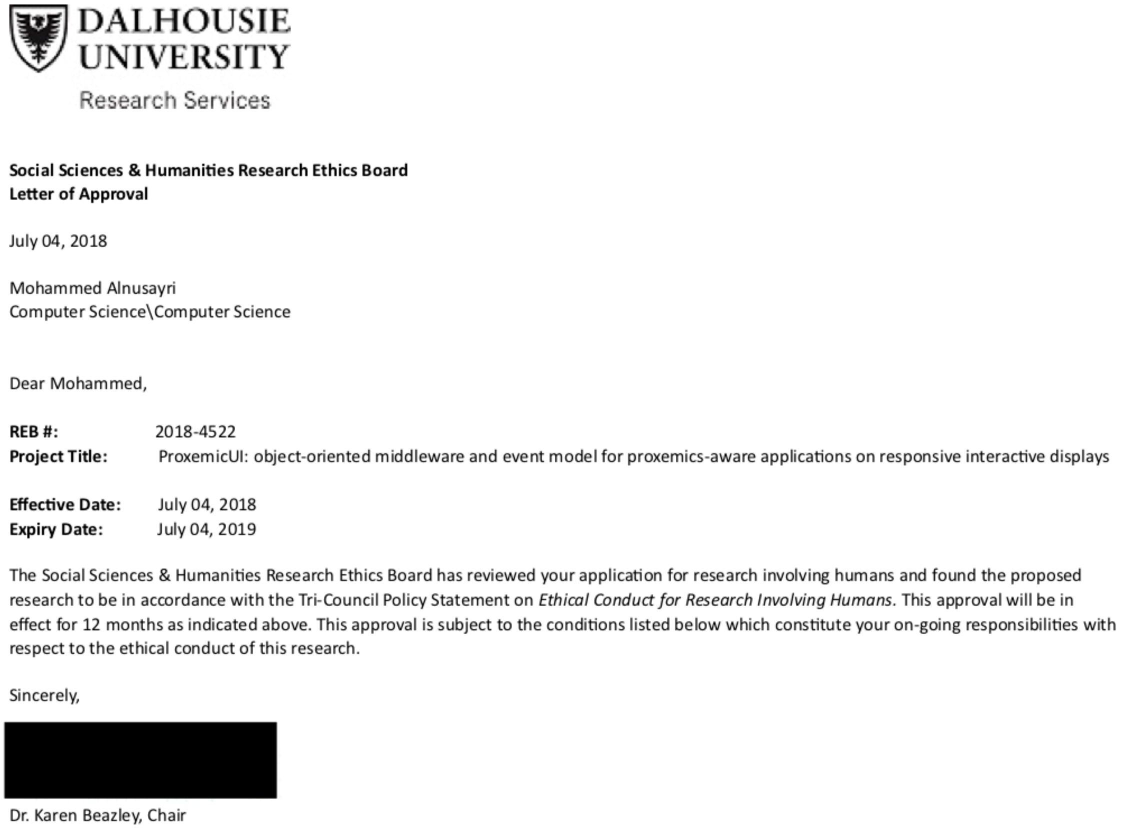
We are happy to invite you to be a member of the jury. We hope you would accept our invitation, if you are please reply to this email letting me know that the date and time suits

Sincerely,

Appendix G – group discussion questions

1- List three things that you liked about the framework, explain each.
2- List three things that you disliked about the framework, explain each.
3- What type of interactions/applications/scenarios require the use of Compound Rules?
4- What type of interactions/applications/scenarios require the use of Hybrid Rules?
5- What limitations have you faced when using the framework?
6- How do you think we can overcome these limitations?
7- In which ways do you think the framework can be improved?

Appendix 4: Letter of Approval for Hackathon User Study



DALHOUSIE
UNIVERSITY
Research Services

Social Sciences & Humanities Research Ethics Board
Letter of Approval

July 04, 2018

Mohammed Alnusayri
Computer Science\Computer Science

Dear Mohammed,

REB #: 2018-4522
Project Title: ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

Effective Date: July 04, 2018
Expiry Date: July 04, 2019

The Social Sciences & Humanities Research Ethics Board has reviewed your application for research involving humans and found the proposed research to be in accordance with the Tri-Council Policy Statement on *Ethical Conduct for Research Involving Humans*. This approval will be in effect for 12 months as indicated above. This approval is subject to the conditions listed below which constitute your on-going responsibilities with respect to the ethical conduct of this research.

Sincerely,

[redacted]

Dr. Karen Beazley, Chair

Appendix 5: Informed Consent for Code Review and Story CreatAR studies

Appendix 5.1: Informed Consent for the Analytical Comparison Sessions

ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

Principal Investigators: Mohammed Alnusayri, Faculty of Computer Science

Dr. Derek Reilly, Faculty of Computer Science

Dr. Joseph Malloch, Faculty of Computer Science

Contact Person: Mohammed Alnusayri, Faculty of Computer Science, mh625076@cs.dal.ca

We invite you to take part in a research study being conducted by Mohammed Alnusayri at Dalhousie University. Your participation in this study is voluntary and you may withdraw from the study at any time. Your academic (or employment) performance evaluation will not be affected by whether or not you participate. To be eligible to participate in the study, you must have programming skills, know how to code with C#, have experience developing applications in one or more of the following domains: Internet of Things, smart spaces, mixed/augmented reality, public digital installations, ubiquitous and mobile computing, and be 18 or older. The study is described below. There are a number of direct benefits for participants taking part in this research project: you will learn how to use three different toolkits and how each was built and what it can support. This might also increase your knowledge in how to build a toolkit. An indirect benefit is the opportunity to advance research knowledge and potentially benefit others.

The purpose of the study is to evaluate our framework (ProxemicUI) and to understand how developers can benefit from it. The study consists of three 45 minutes sessions. During these sessions, you will be exposed to two different C# codes that implement the same application (solve same problems). Each code implements the same application using different toolkit (ProxemicUI and The Proximity Toolkit). You will also be exposed to a set of steps in how to integrate two different toolkits (ProxemicUI and Microsoft Psi) with a machine learning classifier. Then you will be asked about the pros and cons of each toolkit. You and two other individuals will remotely participate (e.g. through Google hangout) in these sessions with the main investigator Mohammed and these sessions will be screen recorded (only the moderator's device).

Your participation is voluntary and there is no compensation. You can withdraw from the study at any time without consequence.

When you sign up for the study, you will be emailed an informed consent form to give consent to do the study.

During these sessions, screen recording of the moderator's device will be captured. All personal and identifying data will be kept confidential. Anonymity of textual data will be preserved by using pseudonyms (e.g., an ID number) to ensure your confidentiality. The informed consent form and all research data will be kept in a secure location under confidentiality in accordance to University policy for three years post publication.

In the event that you have any difficulties with, or wish to voice concern about, any aspect of your participation in this study, you may contact Catherine Connors, Director, Office of Research Ethics Administration at Dalhousie University's Office of Human Research Ethics for assistance: phone: (902) 494-1462, email: Catherine.connors@dal.ca.

"I have read the explanation about this study. I have been given the opportunity to discuss it and my questions have been answered to my satisfaction. ***I understand that being video recorded is necessary to participate in the study.*** *I hereby consent to take part in the study. However, I understand that my participation is voluntary and that I am free to withdraw from the study at any time."*

Participant **Researcher**

Name: ________________________________ Name: ________________________________
Signature: _____________________________ Signature: _____________________________
Date: _________________________________ Date:
____

Please select **one** of the options below:

"I agree to let you directly quote any comments or statements made in any written reports without viewing the quotes prior to their use and I understand that the anonymity of textual data will be preserved by using pseudonyms."

Participant Researcher

Name: ____________________________ Name:
____

Signature: ____________________________
Signature:____________________________
Date: ____________________________
Date:____________________________

<u>Or</u>

"I want to read direct quotes prior to their use in reports and I understand that the anonymity of textual data will be preserved by using pseudonyms."

[if this option is chosen, please include a contact email address: ______________________]

Participant Researcher

Name: ______________________________ Name: _________________________

Signature: ___________________________ Signature:_______________________

Date:* ____** ***Date:

____

If you are interested in seeing the results of this study, please check below and provide your email address. We will contact you with publication details that describe the results.

"I would like to be notified by email when results are available via a publication."

[if this option is chosen, please include a contact email address: ____________________________]

Appendix 5.2: Informed Consent for participants integrating ProxemicUI into Story CreatAR

ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

Principal Investigators: Mohammed Alnusayri, Faculty of Computer Science

Dr. Derek Reilly, Faculty of Computer Science

Dr. Joseph Malloch, Faculty of Computer Science

Contact Person: Mohammed Alnusayri, Faculty of Computer Science, mh625076@cs.dal.ca

We invite you to take part in a research study being conducted by Mohammed Alnusayri at Dalhousie University. Your participation in this study is voluntary and you may withdraw from the study at any time. Your academic (or employment) performance evaluation will not be affected by whether or not you participate. To be eligible to participate in the study, you must be a researcher in the Story CreatAR project in the Graphics and Experiential Media Lab. The study is described below. There are a number of direct benefits for participants taking part in this research project: by integrating ProxemicUI into Story CreatAR you will add useful features to the application and gain experience with the object-oriented framework integration process that will be more generally useful in other software development projects. An indirect benefit is the opportunity to advance research knowledge and potentially benefit others

The purpose of the study is to evaluate our framework (ProxemicUI) and to understand how developers can benefit from it. You will work with the main investigator and the rest of the Story CreatAR team during the period of integration process. You will expose the features of ProxemicUI in Story CreatAR to enhance stories. You will be involved in the regular activities of the Story CreatAR research project. This includes documenting the integration process in a collaborative journal, participating in group meetings, writing and annotating source code, and running tests. You are under no obligation to participate in this study, and your choice has no impact on your ability to participate in the Story CreatAR research project. You can withdraw from the study at any time without consequence. If you choose not to participate or to withdraw participation, your entries in the collaborative journal, your source code annotations, and your participation in group meetings will not be used in analysis.

The main investigator will have emailed you this informed consent form to review, sign, and email back. During the integration process the main investigator will work directly with you, and will be available to answer questions and help overcome issues with the framework as you use it.

During the study, we will document the process of integrating ProxemicUI into Story CreatAR in a collaborative journal, and annotate code changes in version control. The journal, annotations, and source code will be collected for study. We will also collect screen capture of meetings, including collaborative design and coding sessions. All personal and identifying data will be kept confidential. Anonymity of textual data will be preserved by using pseudonyms to ensure your confidentiality. Video capture of meetings will not be used in publication. The informed consent form and all research data will be kept in a secure location under confidentiality in accordance to University policy for three years post publication.

In the event that you have any difficulties with, or wish to voice concern about, any aspect of your participation in this study, you may contact Catherine Connors, Director, Office of Research Ethics Administration at Dalhousie University's Office of Human Research Ethics for assistance: phone: (902) 494-1462, email: Catherine.connors@dal.ca.

"I have read the explanation about this study. I have been given the opportunity to discuss it and my questions have been answered to my satisfaction. ***I understand that being video recorded is necessary to participate in the study.*** *I hereby consent to take part in the study. However, I understand that my participation is voluntary and that I am free to withdraw from the study at any time."*

Participant **Researcher**

Name: ___________________________________ Name: ___________________________________
Signature: ________________________________ Signature: _________________________________
Date: ____________________________________ Date:
____

Please select **one** of the options below:

"I agree to let you directly quote any comments or statements made in any written reports without viewing the quotes prior to their use and I understand that the anonymity of textual data will be preserved by using pseudonyms."

Participant Researcher

Name: _______________________________ Name:
____

Signature: ____________________________
Signature:_______________________________
Date: _______________________________
Date:_________________________________

Or

"I want to read direct quotes prior to their use in reports and I understand that the anonymity of textual data will be preserved by using pseudonyms."

[if this option is chosen, please include a contact email address: ______________________]

Participant Researcher

Name: ________________________________ Name: ___________________________

Signature: _____________________________ Signature:_________________________

Date: _________________________________ ***Date:***

____

If you are interested in seeing the results of this study, please check below and provide your email address. We will contact you with publication details that describe the results.

"I would like to be notified by email when results are available via a publication."

[if this option is chosen, please include a contact email address: _____________________]

Appendix 5.3: Informed Consent for Author Participants in Story CreatAR

DALHOUSIE
UNIVERSITY

ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

Investigators: Mohammed Alnusayri, Faculty of Computer Science

Dr. Derek Reilly, Faculty of Computer Science

Dr. Joseph Malloch, Faculty of Computer Science

Peter Haltner, Faculty of Computer Science

Abbey Singh, Faculty of Computer Science

Ramanpreet Kaur, Faculty of Computer Science

Matt Peachey, Faculty of Computer Science

Contact Person: Mohammed Alnusayri, Faculty of Computer Science, mh625076@cs.dal.ca

We invite you to take part in a research study being conducted by Mohammed Alnusayri at Dalhousie University. Your participation in this study is voluntary and you may withdraw from the study at any time. Your academic (or employment) performance evaluation will not be affected by whether or not you participate. To be eligible to participate in the study be 18 or older. The study is described below. You will directly benefit from participating in this research project by working with investigators to produce a version of a story you have authored as an interactive virtual reality experience, using an authoring toolkit called Story CreatAR. An indirect benefit is the opportunity to advance research knowledge and potentially benefit others.

The purpose of the study is to evaluate the benefit to authors of features integrated into the Story CreatAR toolkit. You will work on creating your stories using Story CreatAR in collaboration with the study investigators, who will provide authoring assistance and support as needed. You can withdraw from the study at any time without consequence.

The main investigator will have emailed you this informed consent form, which you should carefully review, sign, and email back.

During the study, we will collect data of about your usage of Story CreatAR and also ask for your feedback about the process of using the tool and its features. Collected data includes screen capture of collaborative authoring sessions, screen recordings of post-session interviews, design documents created during the authoring process, and copies of the script and resulting VR content. You will retain copies of all creative output. All personal and identifying data will be kept confidential in publication. Anonymity of textual data will be preserved by using pseudonyms to ensure your confidentiality. Audio and video capture will not be used in publication. Inclusion of creative materials (script elements, VR story) in publications will not occur without your explicit verbal consent. The informed consent form and all research data will be kept in a secure location under confidentiality in accordance to University policy for three years post publication.

In the event that you have any difficulties with, or wish to voice concern about, any aspect of your participation in this study, you may contact Catherine Connors, Director, Office of Research Ethics Administration at Dalhousie University's Office of Human Research Ethics for assistance: phone: (902) 494-1462, email: Catherine.connors@dal.ca.

"I have read the explanation about this study. I have been given the opportunity to discuss it and my questions have been answered to my satisfaction. ***I understand that being video recorded is necessary to participate in the study.*** *I hereby consent to take part in the study. However, I understand that my participation is voluntary and that I am free to withdraw from the study at any time."*

Participant Researcher

Name: ___________________________________ Name: ___________________________________
Signature: ________________________________ Signature: ________________________________
Date: ____________________________________ Date:
____

Please select **one** of the options below:

"I agree to let you directly quote any comments or statements made in any written reports without viewing the quotes prior to their use and I understand that the anonymity of textual data will be preserved by using pseudonyms."

Participant Researcher

Name: ______________________________ Name:
____

Signature: ____________________________
Signature:______________________________
Date: **______________________________**
Date: **________________________________**

Or

"I want to read direct quotes prior to their use in reports and I understand that the anonymity of textual data will be preserved by using pseudonyms."

[if this option is chosen, please include a contact email address:
_____________________________]

Participant Researcher

Name: _______________________________ Name: __________________________

Signature: ____________________________ Signature:________________________

Date: **_______________________________** ***Date:***

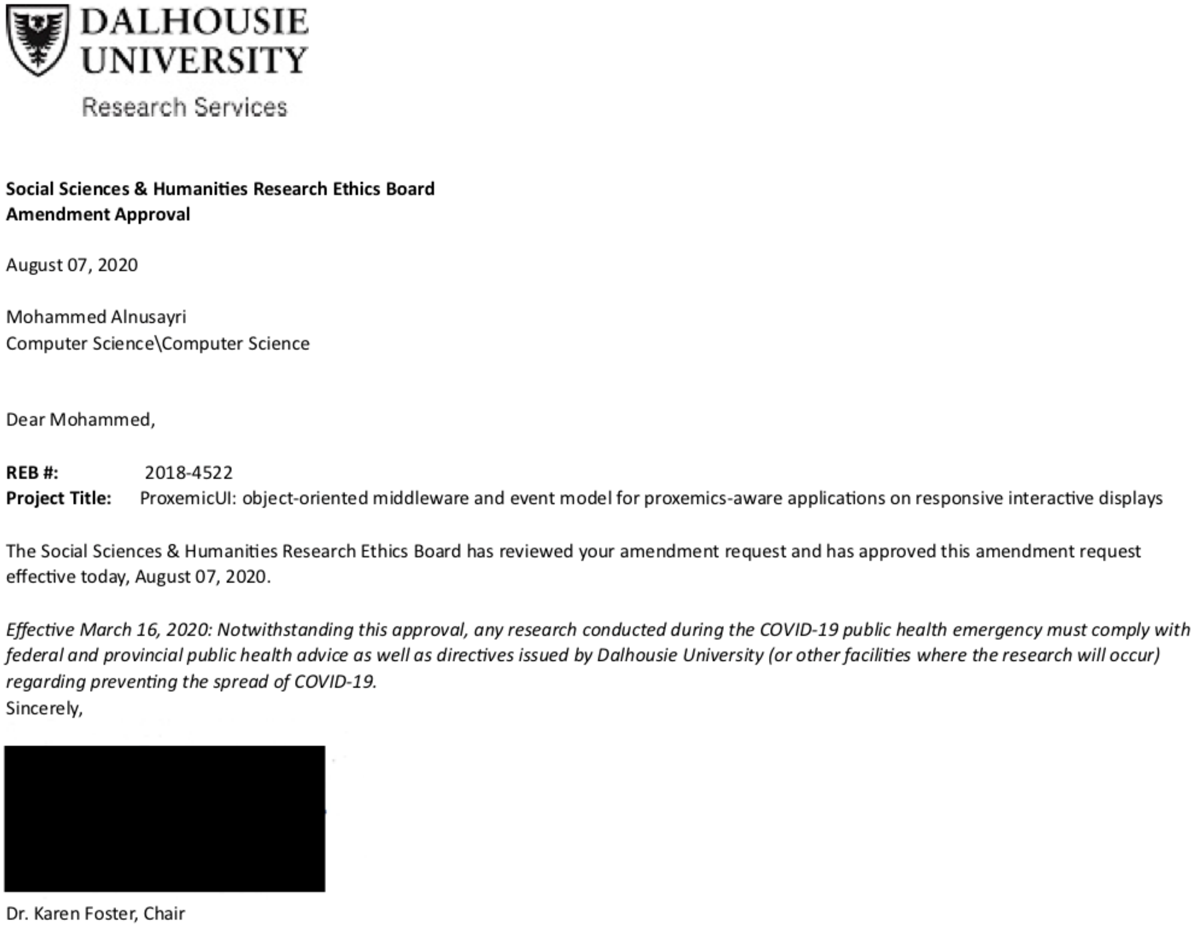
____

If you are interested in seeing the results of this study, please check below and provide your email address. We will contact you with publication details that describe the results.

"I would like to be notified by email when results are available via a publication."

[if this option is chosen, please include a contact email address: __________________________]

Appendix 6: Amendment Approval Letters for Code Review and Story CreatAR studies



DALHOUSIE UNIVERSITY

Research Services

Social Sciences & Humanities Research Ethics Board
Amendment Approval

August 07, 2020

Mohammed Alnusayri
Computer Science\Computer Science

Dear Mohammed,

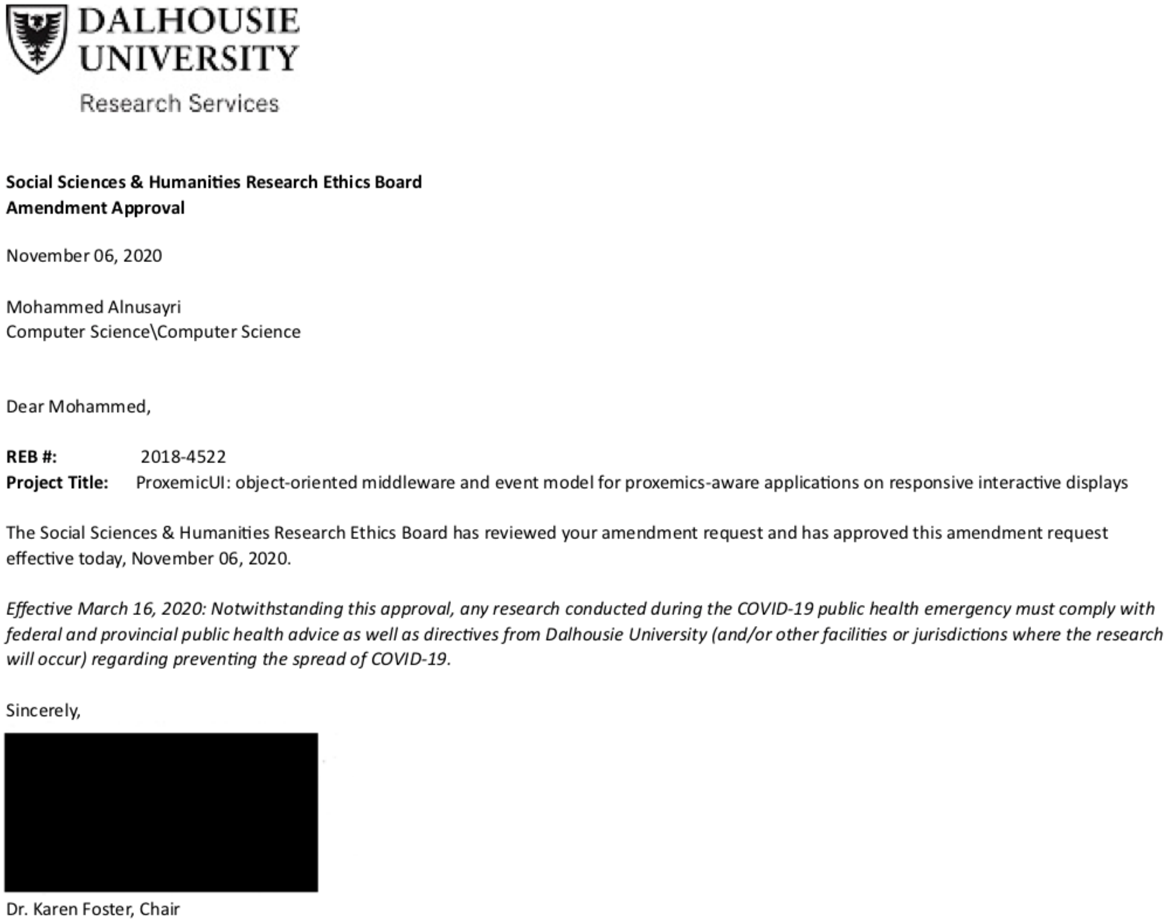
REB #: 2018-4522
Project Title: ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

The Social Sciences & Humanities Research Ethics Board has reviewed your amendment request and has approved this amendment request effective today, August 07, 2020.

Effective March 16, 2020: Notwithstanding this approval, any research conducted during the COVID-19 public health emergency must comply with federal and provincial public health advice as well as directives issued by Dalhousie University (or other facilities where the research will occur) regarding preventing the spread of COVID-19.
Sincerely,

[redacted]

Dr. Karen Foster, Chair



DALHOUSIE UNIVERSITY

Research Services

Social Sciences & Humanities Research Ethics Board
Amendment Approval

November 06, 2020

Mohammed Alnusayri
Computer Science\Computer Science

Dear Mohammed,

REB #: 2018-4522
Project Title: ProxemicUI: object-oriented middleware and event model for proxemics-aware applications on responsive interactive displays

The Social Sciences & Humanities Research Ethics Board has reviewed your amendment request and has approved this amendment request effective today, November 06, 2020.

Effective March 16, 2020: Notwithstanding this approval, any research conducted during the COVID-19 public health emergency must comply with federal and provincial public health advice as well as directives from Dalhousie University (and/or other facilities or jurisdictions where the research will occur) regarding preventing the spread of COVID-19.

Sincerely,

[redacted]

Dr. Karen Foster, Chair