The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies

Nicolai Marquardt[1], Robert Diaz-Marino[2], Sebastian Boring[1], Saul Greenberg[1]

[1] Department of Computer Science
University of Calgary, 2500 University Drive NW
Calgary, AB, T2N 1N4, Canada
[nicolai.marquardt, sebastian.boring, saul.greenberg]@ucalgary.ca

[2] SMART Technologies
3636 Research Road NW
Calgary, AB, T2L 1Y1, Canada
robdiaz-marino@smarttech.com

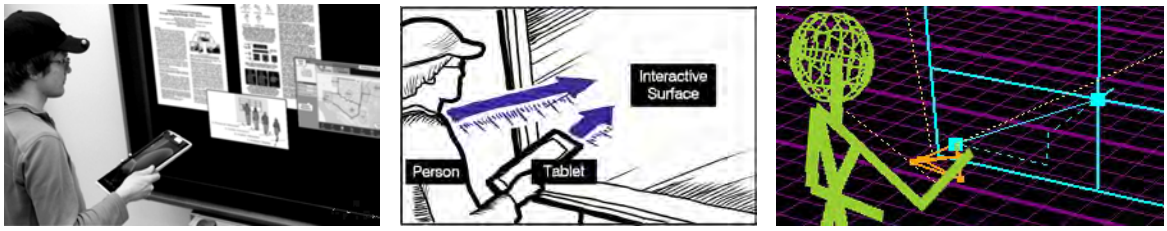


Figure 1. Left: three entities – person, tablet and vertical surface; **Center**: proxemic relationships between entities, e.g., orientation, distance, pointing rays; **Right**: visualizing these relationships in the *Proximity Toolkit* visual monitoring tool.

ABSTRACT

People naturally understand and use proxemic relationships in everyday situations. However, only few ubiquitous computing (ubicomp) systems interpret such *proxemic relationships* to mediate interaction (*proxemic interaction*). A technical problem is that developers find it challenging and tedious to access proxemic information from sensors. Our *Proximity Toolkit* solves this problem. It simplifies the exploration of interaction techniques by supplying fine-grained proxemic information between people, portable devices, large interactive surfaces, and other non-digital objects in a room-sized environment. The toolkit offers three key features. 1) It facilitates rapid prototyping of proxemic-aware systems by supplying developers with the orientation, distance, motion, identity, and location information between entities. 2) It includes various tools, such as a visual monitoring tool, that allows developers to visually observe, record and explore proxemic relationships in a 3D space. (3) Its flexible architecture separates sensing hardware from the proxemic data model derived from these sensors, which means that a variety of sensing technologies can be substituted or combined to derive proxemic information. We illustrate the versatility of the toolkit with a set of proxemic-aware systems built by students.

ACM Classification: H5.2 [Information interfaces]: User Interfaces – input devices and strategies, prototyping. D.2.2 [Software Engineering]: Design Tools and Techniques

General terms: Design, Human Factors

Keywords: Proximity, proxemics, proxemic interactions, toolkit, development, ubiquitous computing, prototyping.

Cite as:
Marquardt, N., Diaz-Marino, R., Boring, S., Greenberg, S. (2011) The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies.
Research Report 2011-1001-13, Department of Computer Science, University of Calgary, Calgary, AB, Canada T2N 1N4, April.

INTRODUCTION

Ubicomp ecologies are now common, where people's access to digital information increasingly involves near-simultaneous interaction with multiple nearby digital devices of varying size, e.g., personal mobile phones, tablet and desktop computers, information appliances, and large interactive surfaces (Figure 1). This is why a major theme in ubiquitous computing is to explore novel forms of interaction not just between a person and a device, but between a person and their set of devices [25]. *Proxemic interaction* is one strategy to mediate people's interaction in a room-sized ubicomp ecology [2,7]. It is inspired by Hall's Proxemic theory [8] about people's understanding and use of interpersonal distances to mediate their interactions with others. In proxemic interaction, the belief is that we can design systems that will let people exploit a similar understanding of their proxemic relations with their nearby digital devices, thus facilitating more seamless and natural interactions.

A handful of researchers have already explored such proxemic-aware interactive systems. These range from spatially aware mobile devices [14], office whiteboards [12], home media players [2], to large public ambient displays [24]. All developed novel interaction techniques as a function of people's and devices' proxemic relationships.

The problem is that building proxemic-aware systems is difficult. Even if the sensing hardware is available, translating low-level sensing information into proxemic information is hard (e.g., calibration, managing noise, calculations such as 3D math). This introduces a high threshold for those wishing to develop proxemic interaction systems. As a result, most do not bother. Of the few that do, they spend most of their time with low-level implementation details to actually access and process proxemic information *vs.* refining the interaction concepts and techniques of interest.

To alleviate this problem, we built the *Proximity Toolkit*. Our goal was to facilitate rapid exploration of proxemic interaction techniques. To meet this goal, the Proximity Toolkit transforms raw tracking data gathered from various hardware sensors into rich high-level proxemic information accessed via an event-driven object-oriented API. The toolkit includes a *visual monitoring tool* that displays the physical environment as a live 3D scene and shows the proxemic relationships between entities within that scene. It also provides other tools: one to record events generated by entities for later playback during testing; another to rapidly calibrate hardware and software. Thus our work offers three contributions:

1. The design of a toolkit architecture, which fundamentally simplifies access to proxemic information.
2. Interpretation and representations of higher level proxemic concepts (e.g., relationships, fixed/semi-fixed features) from low level information.
3. The design of complementing visual tools that allow developers to explore proxemic relationships between entities in space without coding.

The remainder of the paper is structured as follows. First, we recap the concepts behind proxemic interaction and derive challenges for developers. Next, we introduce the design of our toolkit; we include a running example, which we use to illustrate all steps involved in prototyping a proxemic interaction system. Third, we introduce our visual monitor and other tools. Fourth, we explain the toolkit's API. Fifth, we discuss the flexible toolkit architecture and implementation. This is followed by an overview of applications built by others using our toolkit. Finally, we discuss related toolkit work in HCI.

BACKGROUND: PROXEMIC INTERACTION

Proxemics – as introduced by anthropologist Edward Hall in 1966 [8] – is a theory about people's understanding and use of interpersonal distances to mediate their interactions with other people. Hall's theory correlates people's physical distance to social distance. He noticed zones that suggest certain types of interaction: from intimate (6-18"), to private (1.5-4'), social (4-12'), and public (12-25'). The theory further describes how the spatial layout of rooms and immovable objects (*fixed features*) and movable objects such as chairs (*semi-fixed features*) influence people's perception and use of personal space when they interact [8].

Research in the field of *proxemic interaction* [2,7,24] introduces concepts of how to apply this theory to ubicomp interaction within a small area such as a room. In particular, such ubicomp ecologies mediate interaction by exploiting fine-grained proxemic relationships between people, objects, and digital devices. The design intent is to leverage people's natural understanding of their proxemic relationships to the entities that surround them.

Proxemic theories suggest that a variety of physical, social, and cultural factors influence and regulate interpersonal interaction. Not all can be (or needs to be) directly applied to a proxemic ubicomp ecology. Thus the question is: what information is critical for ubicomp proxemics? Greenberg et al. [7] identified and operationalized five essential dimensions.

1. *Orientation*: the relative angles between entities; such as if two people are facing towards one another.
2. *Distance*: the distance between people, objects, and digital devices; such as the distance between a person and a large interactive wall display.
3. *Motion*: changes of distance and orientation over time; such as a person approaching a large digital surface to interact with it directly.
4. *Identity*: knowledge about the identity of a person, or a particular device.
5. *Location*: the setup of environmental features; such as the fixed-feature location of walls and doors, and the semi-fixed features including movable furniture.

Previous researchers have used a subset of these five dimensions to build proxemic-aware interfaces that react more naturally and seamlessly to people's expectations of proxemics. *Hello Wall* [23] introduced the notion of 'distance-dependent semantics', where the distance of a person to the display defined the possible interactions and the information shown on the display. Similarly, Vogel's public ambient display [24] relates people's presence in four discrete zones around the display to how they can interact with the digital content. Ju [12] explored transitions between implicit and explicit interaction with a proxemic-aware office whiteboard: interaction from afar is public and implicit, but becomes more explicit and private when closer. Ballendat et al. [2] developed a variety of proxemic-aware interaction techniques, illustrated through the example of a home media player application. The system exploits almost all of the 5 dimensions: it activates when the first person enters, reveals more content when approaching and looking at the screen, switches to full screen view when a person sits down, and pauses the video when the person is distracted (e.g., receiving a phone call). If a second person enters, the way that the information displays is altered to account for two viewers in the room.

DERIVED CHALLENGES FOR DEVELOPERS

This previous research in proxemic interaction opened up a promising direction of how to mediate people's interaction with ubicomp technology based on proxemic relationships. Building each of these individual systems is, however, a difficult and tedious task; mostly because of the serious technical challenges that developers face when integrating proxemic information into their application designs. Several challenges are listed below.

1. *Exploring and observing proxemic properties between entities in the ecology*. Developers need to do this to help them decide which properties are important in their given situation.
2. *Accessing proxemic measurements* from within software that is developed to control the ubicomp system. Developers currently do this through very low-level

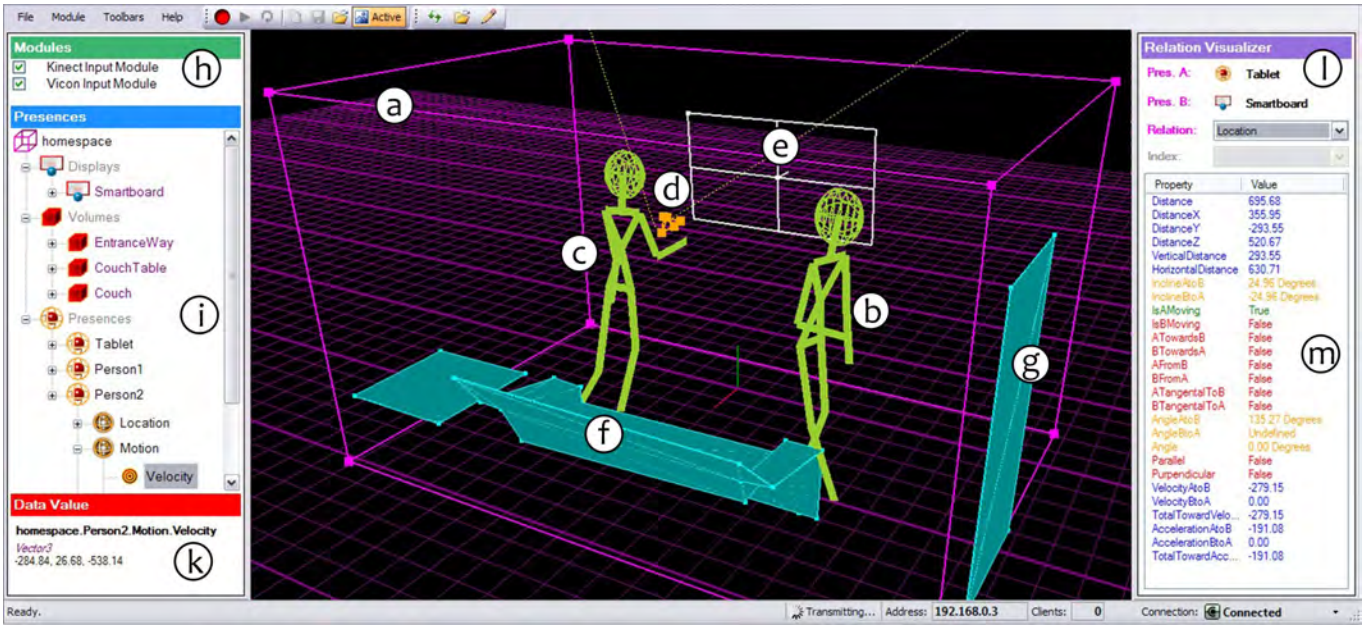


Figure 2. Proximity toolkit monitoring tool: the tracked ubicomp environment (a), the visual representation of tracked entities in space (b-g), list of available input modules (h), list of all tracked entities (i,k), and relation visualizer (l,m)

programming against a particular tracking technology, requiring complex 3D transformations and calculation, and often resulting in brittleness.

3. *Support for proxemic concepts* is created by developers from scratch, e.g., when considering distance of spatial zones or the properties of the fixed and semi-fixed features (e.g., the spatial arrangement) in applications.
4. *Debugging and testing* of such systems is difficult due to a lack of matching monitoring tools.

THE PROXIMITY TOOLKIT

The Proximity Toolkit directly addresses these challenges. It facilitates programmers' access to proxemic information between people, objects, and devices in a small space ubicomp environment (such as the room shown in Figure 3). It contains four main components.

a) **Proximity Toolkit server** is the central component in the distributed client-server architecture, allowing multiple client devices to access the captured proxemic information.
b) **Tracking plug-in modules** connect different tracking / sensing systems with the toolkit and stream the raw input data of tracked entities to the server.
c) **Visual monitoring tool** visualizes tracked entities and their proxemic relationships.
d) **Application programming interface (API)** is an event-driven programming library used to easily access all the available proxemic information from within developed ubicomp applications.

We explain each of these components in more detail below, including how each lowers the threshold for rapidly prototyping proxemic-aware systems.

However, we first introduce a scenario of a developer creating a proxemic interaction system. Through this scenario, we will illustrate how the Proximity Toolkit is used in a real programming task to create a prototype of a proxemic-aware ubicomp application. The example is deliberately trivial, as we see it akin to a *Hello World* illustrating basic programming of proxemic interaction. Still, it shares many similarities with more comprehensive systems built for explorations in earlier research, e.g., [2], [12], or [24].

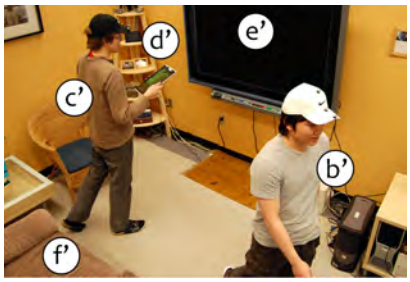


Figure 3. The Proximity Toolkit captures proxemic relationships between: people (b' and c'), devices (d' and e'), and fixed- and semi-fixed features (f').

Scenario. Developer Steve is prototyping an interactive announcement board for the lounge of his company. In particular, Steve envisions a system where employees passing by the display are attracted to important announcements as large visuals from afar, see and read more content as they move closer, and post their own announcements (typed into their mobile phones) by touching the phone against the screen. To create a seamless experience for interacting with the large ambient display, Steve plans to recognize nearby people and their mobile devices. Steve builds his prototype to match the room shown in Figure 3.

Proximity Toolkit Server

The Proximity Toolkit Server is the central component managing proxemic information. It maintains a hierarchical data model of all fixed features (e.g., walls), semi-fixed features (e.g., furniture, large displays), and mobile entities (e.g., people or portable devices). This model contains basic information including identification, position in 3D

coordinates, and orientation. The server component then performs all necessary 3D calculations on this data required for modeling information about higher level proxemic relationships between entities.

The server is designed to obtain raw data from various attached tracking systems. For flexibility, each of the tracking systems is connected through a separate plugin module loaded during the server's start-up. These plugins access the captured raw input data and transfer it to the server's data model. The current version of our toolkit contains two plugins: the marker-based VICON motion capturing system which allows for sub-millimeter tracking accuracy [www.vicon.com], and the KINECT sensor, which allows tracking of skeletal bodies [www.kinect.com]. (A later section discusses the implementation, integration, and combination of these tracking technologies, and how to setup the server to match the environment.) Importantly, the server's unified data model is the basis for a *distributed Model-View-Controller* architecture, which in turn is used by the toolkit client API, the monitoring tool, and to calculate proxemic relationships between entities.

Scenario. Developer Steve begins by starting the server. The server automatically loads all present tracking plugins. Based on the information gathered from these plugins, it populates and updates the unified data model in real-time. By default, our toolkit already includes a large preconfigured set of tracked entities with attached markers (such as hats, gloves, portable devices) and definitions of fixed and semi-fixed features (large interactive surface, surrounding furniture). To add a new tracked object, Steve attaches markers to it and registers the marker configuration as a new tracked entity. This process takes minutes.

Visual Monitoring Tool: Tracked Entities

The visual monitoring tool helps the developer see and understand what entities are being tracked and how the data model represents their individual properties. Figure 2 is a screen snapshot of this tool, where the visualized entities in Figure 2 b-f corresponds to the real-world entities captured in Figure 3b'-f' .

Specifically, the visual monitoring tool connects to the server (through TCP) and presents a 3D visualization of the data model (Figure 2 centre). This view is updated in real-time and always shows:

- the approximate volume of the tracked space as a rectangular outline box (Fig. 2a)
- position and orientation of people (Fig. 2bc)
- portable digital devices, such as a tablet pc (Fig. 2d)
- digital surfaces, such as the large wall display (Fig. 2e)
- fixed and semi-fixed features, such as a table, couch (Fig. 2f), and entranceway (Fig. 2g).

The left side of the monitoring window shows a list of the activated input tracking plugins (Figure 2h) and another list with an overview of all currently tracked entities (Figure 2i). Clicking on any of the items in this list opens a hierarchical list of properties showing the item's current status (e.g., its location, or orientation). When Steve selects any of these properties, the monitoring window shows the corresponding value (e.g., the current position as a 3D Vector, or the velocity; Fig 2k). Part A of Table 1 shows an overview of the most important available properties.

Scenario. Before Steve starts to program, he explores all available proxemic information through the visual monitoring tool. He inspects the currently tracked entities (Figure 2 left, also displayed in the center), as well as what entity prop-

		Property name	Description	Data type	Distance	Orientation	Movement	Identity	Location
A Individual entity	I1	**Name**	Identifier of the tracked entity	string				■	
	I2	**IsVisible**	True if entity is visible to the tracking system	bool				■	
	I3	**Location**	Position in world coordinates	Point3D					■
	I4	**Velocity**	Current velocity of the entity's movement	double			■		
	I5	**Acceleration**	Acceleration	double			■		
	I6	**RotationAngle**	Orientation in the horizontal plane (parallel to the ground) of the space	double		■			
	I7	**[Roll/Azimuth/Incline]Angle**	The orientation angles (roll, azimuth, incline)	double		■			
	I8	**Pointers**	Access to all pointing rays (e.g., forward, backward)	Array [ ]		■			
	I9	**Markers/Joints**	Access individual tracked markers or joints	Array [ ]					■
B Relationships between two entities A and B	R1	**Distance**	Distance between entities A and B	double	■				
	R2	**ATowardsB, BTowardsA**	Whether entity A is facing B, or B is facing A	bool		■			
	R3	**Angle, HorizontalAngle, ...**	Angle between front normal vectors (or angle between horizontal planes)	double		■			
	R4	**Parallel, ATangentalToB, ...**	Geometric relationships between entities A and B	bool		■			
	R5	**[Incline/Azimuth/Roll]Difference**	Difference in incline, azimuth, or roll of A and B	double		■			
	R6	**VelocityDifference**	Difference of A's and B's velocity	double			■		
	R7	**AccelerationDifference**	Difference of A's and B's acceleration	double			■		
	R8	**[X/Y/Z]VelocityAgrees**	True if X/Y/Z velocity is similar between A and B	bool			■		
	R9	**[X/Y/Z]AccelerationAgrees**	True if X/Y/Z acceleration is similar	bool			■		
	R10	**Collides, Contains**	True if the two volumes collide, or if volume A contains volume of B	bool	■				■
	R11	**Nearest**	The nearest point of A's volume relative to B	Point3D	■				■
C Pointing Relationships between A and B	P1	**PointsAt**	Pointing ray of A intersects with volume of B	bool		■			
	P2	**PointsToward**	A points in the direction of B (w/ or w/o intersection)	bool		■			
	P3	**IntersectionDegree**	Angle between ray and front facing surface of B	double		■			
	P4	**DisplayPoint**	Intersection point in screen/pixel coordinates	Point2D	■	■			
	P5	**Intersection**	Intersection point in world coordinates	Point3D	■				■
	P6	**Distance**	Length of the pointing ray	double	■				
	P7	**IsTouching**	A is touching B (pointing ray length ~ 0)	bool	■				

Table 1. Accessible proxemic information in the Proximity Toolkit: individual entities, relationships between two entities, and pointing relationships. This information is accessible through the toolkit API and the toolkit monitor visualization.

erties are available for him to use. Steve finds this visual overview particularly important to his initial design, as he is still investigating the possible mappings of proxemic relationship to system behaviour. In later stages, he will also use this monitoring tool to test and debug his program.

Visual Monitoring Tool: Relationships

Another major feature of the visual monitoring tool is to let people set and observe particular *proxemic relationships* between entities, where developers will use these relationships to define particular proxemic interaction behaviours. Specifically, the *Relation Visualizer* panel (Fig. 2, l-m) allows a developer to select a type of relationship between entities, and then to observe the values of all related properties. The complete list of proxemic relationships that are available to observe are summarized in part B/C of Table 1.

Scenario. Steve wants to observe a relationship between *Person1* (representing the first person entering the space) and the *Smartboard* display. Steve drags the two entries from the list of tracked entities (Fig. 2i) to the top of the *Relation Visualizer* panel (Fig. 2l). Next, Steve selects one of the following relationship categories from a drop down menu.

- **Orientation** (e.g., angles between entities)
- **Location** (e.g., changes in distance between the person and the smartboard)
- **Direction** (e.g., if the front of the person's body faces towards the screen)
- **Movement** (e.g., acceleration or velocity)
- **Pointing** (e.g., the display intersection point of the right arm pointer of the person)
- **Collision** (e.g., if the volumes of two tracked entities are so close that they collide)

Steve can now observe how those entities relate to each other. The panel in Fig. 2m shows the numeric values of any properties belonging to this category. The categories plus the properties within them operationalize the 5 essential elements of proximity mentioned previously.

With his public announcement application in mind, Steve is interested in knowing when a person is in close distance to the display. He selects the *Location* category, and views the values of the *Distance* property, which in this case measures the distance of the person's body to the board (Fig. 2m). Next, he wants to know when the person is facing towards the screen. He selects the *Direction* category from the menu, and immediately sees the related proxemic properties with their current values and their graphical appearance in the visualization. He is particularly interested in the *ATowardsB* property (is *true* if the person [A] is facing towards the smartboard [B]). He decides to use the information about direction and distance to adapt the content shown on the announcement board.

Steve continues exploring other proxemic relationships categories and makes note of the types of relationships that he will integrate into his application. As he selects these other categories (Fig. 2l), the 3D visual representation changes accordingly. Figure 4 illustrates three other visualizations of proxemic relationships that Steve explored: the distance between the person and the display (Fig. 4a), the forward pointer of the left arm and its intersection point with the smartboard (Fig. 4b), and the collision volumes (Fig. 4c).

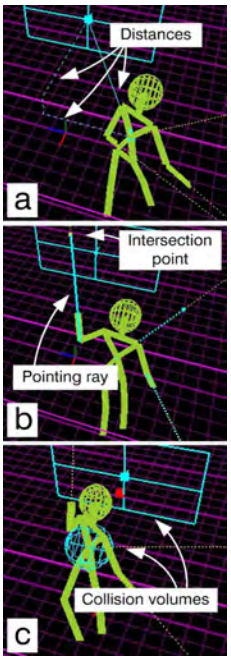


Figure 4. Visualizing proxemic relationships: distance (a), pointing (b), and collision (c).

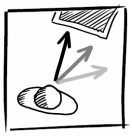
SIMPLIFIED API ACCESS TO PROXEMIC INFORMATION

We now take a closer look at the development API, offered via an *object-oriented C# .NET development library.* We designed it to be fairly easy to learn and use by taking care of and hiding low-level infrastructure details and by using a conventional object-oriented and event-driven programming pattern. Essentially, the API lets a developer programmatically access the proxemic data previously observed in the monitoring tool. We explain how this works by continuing our scenario.

Scenario. Steve adds the Proximity Toolkit API DLL to his own PC-based software project. The only criteria is that his PC needs network access to the proximity server. Steve begins by initializing his software. To set up his software to use the server, he adds three lines of code (lines 1-3 in Figure 5). First, he creates a new client connection object, then starts the connection to the server (at the given IP address and port), and finally creates a `ProximitySpace` object which provides a high-level framework for monitoring the interaction of tracked presences, such as people and objects. The `ProximitySpace` object maintains a list of all available tracked entities, and is used to create instances of entities or for initializing event handlers to monitor relationships. Next, Steve initializes three of the entities he is interested in lines 4-6: the `person` representing the first person entering the space, the `smartboard`, and a `tablet` (`PresenceBase` is a special object that represents individual tracked or static objects).

The following describes how Steve then monitors the relationships *between* these entities. We go through each of the five proxemic dimensions introduced earlier (albeit in a slightly different order), explain how Steve writes his application to monitor changes in each of these dimensions, and how he uses that information to mediate interaction with his interactive announcement board.

1. Orientation

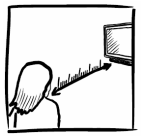


Monitoring orientation changes allows (1) accessing the exact angle of orientation between two entities or (2) determining whether two entities are facing each other. Steve is

mostly interested in the relationship between a person and the smartboard display. He adds line 7, which creates a relationship between these two as indicated by their parameters. The system is now tracking both entities relative to each other. Steve is also interested in knowing when the orientation and location between these two changes. For orientation, he initializes an event handler to receive updates of the *Direction* relationship between the person and the smartboard (line 8). The `OnDirectionUpdated` method is invoked when the system recognizes any changes in orientation between the person and the smartboard (line 10). While Steve could access each entity's precise orientation values (e.g., angles of orientation), he is only really interested in knowing whether a person is facing towards the smartboard. Consequently, he writes the event handler callback method (lines 10-12) to access the `ATowardsB` property in the event arguments: it is `true` if the person is facing the smartboard (line 11).

Entries R2-R5 and P1-P3 in Table 1 give an overview of further orientation relationships that can be monitored. As well, the programmer can access the absolute orientation of an individual entity at any time (see entries I6 – I7 in Table 1). For example, the following property returns the current yaw angle of the tablet: `tablet.Orientation.Yaw;`

2. Distance, including Location, Pointing and Touching

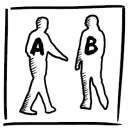


Similarly, Steve can monitor changes of distance between entities. We illustrate how Steve can receive updates about distance changes by adding another event callback for `OnLocationUpdated` events (line 9). This callback method (line 13-15) is invoked whenever the location of at least one of the two entities changes. In line 14 Steve accesses the current distance between the person and the smartboard, and uses this distance value to make the visual content on the announcement board vary as a function of the distance between the person and the display. The closer the person, the more content is revealed.

Other available properties relate to distance. First, the actual location property of each entity, i.e, their position within the space, is accessible at any time. For example Steve can access the current coordinates of the person by accessing `this.person.Location`. Second, *pointing relationships* monitor orientation *and* distance simultaneously. Pointing is similar to ray-casting. Each entity can have one or multiple pointers. Each pointer has a pointing direction, and the callback returns the intersection of that direction with the other entity. It also returns the length of the pointing ray between entities, which may not be exactly the same as distance. To illustrate, Steve tracks not only the close distance of a tablet computer to the smartboard, but where that tablet raycasts onto the smartboard. He initializes a second `RelationPair` between the `tablet` and the `smartboard` (line 16). He subscribes for `OnPointingUpdated` events that are triggered whenever any of the pointers of the tablet changes relative to the board (line 17). In the event callback method (lines 18 to 22) Steve first checks if the tablet's forward pointer faces the display (`PointsTowards`) and if the ray length between tablet and board is smaller than 50 cm (line 19). If this is the case, he shows an icon on the ray's intersection point (line 20) on the smartboard to let the person know they can touch the surface to initiate a transfer.

Third, Steve checks if the tablet is touching the surface - (`IsTouching`, line 21) – a distance of ~0. If so, he initiates transfer of the content on the tablet to the large display. By using the intersection point of the tablet with the screen Steve can show the transferred content at the exact position where the tablet touches the board.

3. Identity



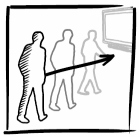
The toolkit allows access to the identity information of all tracked entities. The `Name` property provides the identifier string of each entity, and `IsVisible` is true if the entity is currently tracked by the system. A developer can subscribe to events notifying about any new tracked entities that enter the ubicomp space through the `space.OnPresenceFound` event. In the associated event callback method, the event arguments give information about the type and name of the detected entity. For example, Steve could have his system track and greet a previously unseen person with a splash

```
01  ProximityClientConnection client = new ProximityClientConnection();                                   Setup
02  client.Start("192.168.0.11", 888);
03  ProximitySpace space    = client.GetSpace();
04  PresenceBase person     = space.GetPresence("Person1");
05  PresenceBase smartboard = space.GetDisplay("SmartBoard");
06  PresenceBase tablet     = space.GetDisplay("Tablet");
07  RelationPair relation   = space.GetRelationPair(person, smartboard);                                  Events
08  relation.OnDirectionUpdated += new DirectionRelationHandler(OnDirectionUpdated);
09  relation.OnLocationUpdated  += new LocationRelationHandler(OnLocationUpdated);
10  void OnDirectionUpdated(ProximitySpace space, DirectionEventArgs args) {                             Callbacks
11      if (args.ATowardsB) { [... person is facing the display, show content ...] } else { [...hide...] }
12  }
13  void OnLocationUpdated(ProximitySpace space, LocationEventArgs args) {
14      double distance = args.Distance; [... change visual content as a function of distance ...]
15  }
16  RelationPair relationTablet = space.GetRelationPair(tablet, smartboard);                              Event
17  relationTablet.OnPointingUpdated += new PointingRelationHandler(OnPointingUpdated);
18  void OnPointingUpdated(ProximitySpace space, PointingEventArgs args) {                               Callback
19      if (args["forward"].PointsToward && (args["forward"].Distance < 500.0)) {
20          Point intersection = args["forward"].DisplayPoint;
            [... show awareness icon on smartboard display ...]
21          if (args["forward"].IsTouching) {
                [... transfer content from the tablet to the large display ...]
22  }}}
```

Figure 5. Partial source code for the proxemic-aware announcement board application.

screen on first appearance, and dynamically initialize any necessary event callbacks monitoring that person to other entities in a scene.

4. Motion



Motion events describe the changes of distance and orientation over time. For example, it is possible to receive updates of changes in acceleration and velocity of any entity. For example, Steve can have his application ignore people moving quickly by the display, as he thinks they may be annoyed by any attempts to attract their attention. To receive such velocity updates, Steve would add an event handler (similar to lines 8 and 9) through `OnMotionUpdated` and then simply access the value of the `args.Velocity` property. Based on that value, he would activate the display only if the velocity was less than a certain threshold. Of course, Steve could have determined a reasonable threshold value by observing the velocity value of a person rushing by the display in the visual monitoring tool.

5. Location: Setup of Environment

Using location, the toolkit lets one track the relationships of people and devices to the semi-fixed and fixed features in the physical environment. For example, the model may contain the fixed-feature position of the entranceway to a room, allowing one to know if someone has crossed that threshold and entered the room. It may also contain the location of semi-fixed features, such as the chairs and table seen in Figure 3. Monitoring event handlers for fixed and semi-fixed features can be initialized similarly to the ones we defined earlier.

Steve sets up several fixed feature entities – the smartboard and the entrance-way – through several initial configuration steps. This only has to be done once. Using a physical pointer (the stick in Figure 6a), he defines each entity's volume by physically outlining them in space. Under the covers, the toolkit tracks the 3D tip location of this stick and builds a 3D model of that entity. Each location point of the model is confirmed by pressing a button (e.g., of a wirelessly connected mouse). Figure 6 illustrates how Steve defines the smartboard. After placing the pointer in the four corners of the display plane (Fig. 6a), the coordinates appear in the visualization (6b), and a control panel allows fine adjustments. He saves this to the Proximity Toolkit server as a model. Similarly, Steve defines the entrance-way by outlining the door (Fig. 2g), and the couch by outlining its shape (Fig. 2f). Steve can now monitor proxemic relationships between all moving entities and these new defined features. For example, he can create an event handler to receive notifications when a person passes through the entrance-way (by using the `OnCollisionUpdated` event) and when a person sits on the couch (using the `Distance` property of the `OnLocationUpdated`).

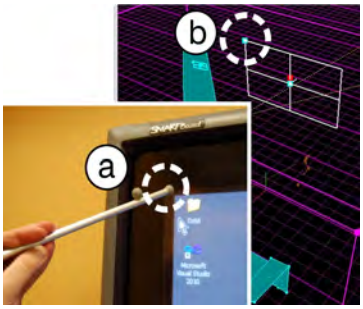


Figure 6. Defining new fixed and semi-fixed features (e.g., display) using a tracked physical pointer (a) and visual feedback (b).

Semi-fixed features differ. While they are part of the environment, they are also movable. As with fixed features, a developer would model a shape by outlining it with the stick. Unlike fixed features, he would also add markers to that entity. The toolkit tracks those markers, and repositions the entity accordingly. For example, Steve could have modeled a chair, tracked where it is in the room, and adjusted the presentation if a person was sitting on it.

We should also mention that we believe location should also include further contextual information about this particular environment, e.g., the meaning of that place. Such contextual information is not yet included in the toolkit, but could be easily added as metadata.

Additional Tools Facilitating Prototyping Process

The toolkit is more than an API, as it offers additional tools to lower the threshold for developing proxemic-aware systems. The already-discussed visual monitoring tool is one of these. Several others are described below.

Recording and playback of proxemic sequences. To test applications, developers would need actors to perform the proxemic movements between entities every time. This is problematic for many reasons. First, it is tedious. Second, it may involve multiple people and multiple devices moving at the same time, which may be both hard to gather logistically and/or to choreograph. Third, the sensing equipment may not be available, e.g., if a developer works at their desk. Fourth, it is difficult to repeat particular test sequences. To alleviate this, the toolkit provides a record/playback tool within the visual monitoring tool. With the click of a button, developers can record events generated by entities moving in the environment. They can later play back these sequences for testing. Under the covers, each individual sequence is recorded as an XML file, where the toolkit uses that record to recreate all events. In turn, this drives the application as if these events were actually happening in real time. Because the tracking hardware is not necessary during playback, testing can be done anywhere, e.g., a desktop workstation located elsewhere. For example, Steve could have recorded test sequences such as: a person passing by the screen, a person approaching the display, or a device pointing towards the display. He would then replay these sequences while developing and testing his software at his desk.

Toolkit component library. Most developers are well-practiced with existing languages and development environments. We leverage these existing practices by seamlessly integrating the toolkit into the familiar capabilities of a popular IDE, Microsoft Visual Studio (but our ideas are generalizable to other IDEs). For example, the toolkit in-

cludes a library of drag-and-drop components compatible with both *WPF* and *WinForms*. This includes representations of all tracked entities via `ProximitySpace` and `PresenceBase` components, and their relationships via a `RelationPair` component. As with other visual components in an IDE, the programmer can view and set all its properties and generate event handlers for all available events via direct manipulation rather than coding. This not only reduces tedium and coding errors, but reduces the threshold for inexperienced developers (such as students) as all properties and events are seen.

Templates, example library, and documentation. Our toolkit includes various facilities to ease learning of how to program with the Proximity Toolkit. First, programmers starting from scratch would almost always have to write some setup code to initialize their program to use proxemic interactions. We reduce start-up effort to almost zero by including a set of templates containing this code. Second, there are several standard patterns that we expect programmers to use when designing proxemic interactions. To ease learning, we provide a large set of teaching applications. Each illustrates, using a very simple example, the code required to implement a particular proxemic relationship. Third, programmers expect good documentation. Thus we include extensive API documentation and tutorial videos.

FLEXIBLE AND EXTENSIBLE ARCHITECTURE

Our first version of the toolkit [4] was tightly linked to a particular tracking technology. This means that other technologies could not be exploited. The current version of the toolkit decouples the API from underlying tracking technologies. We describe our extensible plugin architecture, the two tracking systems we integrated, and how those are reflected in the API.

Plugin architecture. The data providers of raw tracking input data are implemented as separate plugin modules, which are dynamically loaded into the proximity server at start-up. We currently have plugins for two different tracking technologies: the VICON motion capturing system that tracks infrared reflective markers, and the Microsoft KINECT depth camera. The plugin for each of these tracking systems accesses the underlying system software (the NEXUS software for VICON cameras, and the PRIMESENSE OPENNI for the depth camera [www.openni.org]) to get the raw data of tracked people, objects, and/or devices in 3D space. This raw data is then transmitted to the Proximity Toolkit server and stored in a unified data model as proxemic information of each entity. The server calculates the necessary proxemic relationships (distance, orientation, collision, etc.) for the entities present in the data model. To reduce computational overhead, the necessary 3D calculations are done only on demand, i.e., when any of the connected clients subscribe to the particular information. We foresee a variety of further plugins for tracking systems, such as other IR marker-based recognition systems.

Extensions. The Proximity Toolkit provides development templates, base classes, interfaces, and utility classes to facilitate integration of additional tracking technologies. To add a tracking system, programmers begin with the plugin template, derived from the plugin base class. They then implement several mandatory methods, including one that registers with the toolkit server on start-up, and another that implements the update method responsible to stream sensed tracking data into the toolkit. This base class also provides a set of utility methods, such as one for affine transformations from the tracking system's local coordinate system to the Proximity Toolkit's unified coordinate system (this affine matrix is calculated through a simple one time calibration process). As mentioned before, no high-level calculations on the raw input data are required for the plugin implementation, as these are performed by the proximity server.

Diverse tracking capabilities. In order to allow the integration of hardware with different tracking capabilities, developers specify the kinds of proxemic information (provided by that particular hardware) in the plugin implementation. For example, a tracking system might gather information about the position of an entity, but *not* its orientation. At any time, the visual monitoring tool allows to inspect all available types of proxemic information that are supported by the plugins (and therefore tracking systems) activated at that time. This can also be checked from within the client API through the `IsVisible` and `LastUpdated` properties of each available proxemic dimension.

Substitution. Tracking systems/plugins can be *substituted*, providing that their hardware gathers similar tracking information. For example, instead of using the depth camera for tracking people's position and posture, a programmer can use the IR motion capture system instead by attaching IR reflective markers to a person's body. A programmer's access to this proxemic information via the toolkit API remains unchanged, regardless of the tracking mechanism used.

Combination. In case different plugins provide *complementary* tracking information of a single entity, the information is combined in the proximity server's data model. For example, the KINECT and VICON systems could both track a person simultaneously: the KINECT system then provides information about the person's body position in 3D space, and the VICON system tracks a glove the person is wearing in order to retrieve fine-grained information of the person's finger movements. Both plugins then update the entity's data model in the server with their tracked information. If two systems in fact provide *overlapping/conflicting* tracking data (e.g., two systems provide information about an entity's location), the information will be merged in the server's data model. In principle, the plugins set a `Confidence` property (ranging from 0.0 to 1.0) when supplying tracking information of an entity to the server. The server then calculates a weighted average of all values received in a certain time frame (i.e., one update cycle) and updates the proxemic data model of that entity.

APPLICATIONS OF PROXEMIC INTERACTION

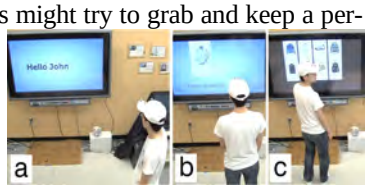
The Proximity Toolkit allowed our colleagues – most of whom were *not* involved in the toolkit design and coding – to rapidly design a large variety of proxemic-aware ubicomp systems. Suffice to say, the toolkit was invaluable. Instead of struggling with the underlying low level implementation details, both colleagues and students were able to focus on the design of novel interaction techniques and applications that considered people's use of space. This includes comprehensive systems such as the proxemic media player by Ballendat et al. [2], and other applications presented in [7].

To stress the ease of learning and developing with our toolkit, we summarize a few projects built by students in a graduate ubicomp class in Fall 2010. They received a one hour tutorial presentation and a demonstration of two programming examples. The students' assignment was simply to create a proxemic interface of their choosing, where they had to demonstrate it in the next class. Thus all examples (listed in Table 2 and briefly explained below) were built and demonstrated by the students within a week of the tutorial.

Application	*Proxemic relationships between*
Attention demanding advertisements	2 people, 1 large surface, 1 tablet
Spatial music experience	2 people, 4 objects
Proxemic-aware pong game	2 people, 1 large surface
Proxemic presenter	1 person, 1 large surface

Table 2. Overview of built proxemic-aware applications.

Attention-Demanding Advertisements explores how future advertisement displays might try to grab and keep a person's attention. A digital advertisement board tries to attract the attention of a passer-by. The board welcomes a person by addressing them with their name (a), shows items of interest to them (b), but then persistently tries to regain the attention of that person if they look or move away by playing sounds and flashing the background color (c).



Spatial Music Experience is an interactive music installation. The kinds of sounds generated and their volume is determined by the proxemic relationships of *people* and *physical objects* in the space. Generated sounds react fluently as one or both people move through the space, when they perform gestures, or when they grab and move physical objects.

Proxemic-aware Pong Game is inspired by Atari's Pong game. A person controls the paddle for bouncing the ball by physically moving left and right in front of a large screen. The system recognizes when a second person enters, and creates a second paddle for multiplayer game play. To increase the game play difficulty later during the game, the system increases the required physical distance to move the paddles. The system also considers the players' front-back movements: when moving close the screen they can adjust the paddle size through direct touch on the screen, and when both players sit down on the couch the game pauses.

Proxemic Presenter is a presentation controller that reacts to the presenter's position relative to a large display [7]. Presentation slides are displayed full screen on the large display. When the presenter stands at the side and turns his head towards the display, a small panel appears next to him, showing speaker notes, a timer, and buttons to navigate the slides. If he switches sides, the panel will appear at that side. When facing back to the audience, the panel disappears immediately. If the presenter moves directly in front of and turns towards the display, the system shows an overview of all slides as thumbnails. The presenter can directly select one of the slides through direct touch. When he turns back to the audience, the presentation reappears.

In these examples, what is important is how the Proximity Toolkit lowered the threshold for these students to begin their exploration of proxemics in the ubicomp context. The easy and direct access to proxemic information through the toolkit and API allowed them to rapidly prototype alternative system designs, all leading towards exploring the design space of future proxemic-aware ubicomp systems.

RELATED WORK

Our research is inspired by earlier toolkits enabling the rapid prototyping of ubicomp interactions. We sample and review related work in three areas: toolkit support in HCI, ubicomp development architectures, and 3D spatial tracking.

Post-GUI Toolkits

Several development toolkits facilitate the prototyping of physical and tangible user interfaces that bridge the connection between the digital and physical world [11]. Many of these toolkits focus on a low *threshold*, but simultaneously aim for maintain a relatively high *ceiling* [20]. For example, *Phidgets* [6] and the *iStuff* toolkit [1] provide physical building blocks (buttons, sensors) that programmers can easily address from within their software. *Shared Phidgets* took this concept further by simplifying the prototyping of distributed (i.e. remote located) physical user interfaces [18]. Hartmann's visual authoring environment in *dTools* [9] brought similar concepts to interaction designers. Other toolkits simplified the integration of computer vision techniques into novel user interfaces, such as Klemmer's *PapierMache* [13].

Ubicomp Development Architectures

On a somewhat higher level of abstraction, Dey introduced an architecture to compose context-aware ubicomp systems with the *Context Toolkit* [3]. They provide *context widgets* as encapsulated building blocks, working in conjunction with *generators*, *interpreters*, or *aggregators*. The context toolkit allows the composition of new applications through a concatenation of the basic components – and thus facilitates scaffolding approaches. Matthews applied similar concepts to the programming of peripheral ambient displays [19].

Other systems facilitate access to location information of devices in ubicomp environments. For example, Hightower's *Location Stack* [10] fuses the input data from various sources to a coherent location data model. Krumm and Hinckley's *NearMe* wireless proximity server [15] derives the position of devices from their 802.11 network connections (without requiring calibration), and thus informs devices about any other devices nearby. Li's *Topiary* [16] introduced prototyping tools for location-enhanced applications.

3D Spatial Tracking

Few development toolkits support the exploration of novel interfaces considering the presence, movements, and orientation of people, objects, and devices in 3D space. For example, some toolkits allow development of augmented reality (AR) applications. To illustrate, Feiner's prototyping system allows exploration of novel mobile augmented reality experiences (e.g., with a head mounted 3D display, or a mobile tablet like device) [5]. This was developed further in MacIntyre's *DART* [17], *Open Tracker* [21], and Sandor's prototyping environment [22] for handheld-based AR applications. These toolkits mostly focus on supporting augmented reality applications running on mobile devices, and not on ubicomp ecologies in small rooms. Some commercial systems track 3D data of objects. For example, the VICON Nexus software gives access to 3D spatial information of tracked objects. This information, however, only includes low level position data, which developers need to process manually in order to gain insights into proxemic relationships.

Our Proximity Toolkit builds on this prior work. Like post-GUI toolkits, it bridges the connection between the virtual and real world, but in this case by tracking proxemic information. Similarly, it extends ubicomp architectures and 3D spatial tracking by capturing and providing fine-grained information about 3D proxemic relationships in small ubicomp spaces (i.e., not only location, but also orientation, pointing, identity, etc.). Like the best of these, it supplies an API that, in our case, makes the *five essential proxemic dimensions* [7] easily accessible to developers. Like the more advanced tools, it also provide additional development tools, such as a monitoring tool for visualizing proxemic relationships, a record/playback tool to simplify testing; templates, documentation, examples, and so on.

CONCLUSION AND FUTURE WORK

The Proximity Toolkit enables rapid prototyping and exploration of novel interfaces that incorporate the notion of proxemic relationships. Through hiding most of the underlying access to tracking hardware and complex 3D calculations, our toolkit lets developers concentrate on the actual design and exploration of novel proxemic interaction.

We invite other researchers to use it. The Proximity Toolkit is available as open source: http://grouplab.cpsc.ucalgary.ca

ACKNOWLEDGMENTS

This research is partially funded by the iCORE/NSERC/SMART Chair in Interactive Technologies, Alberta Innovates Technology Futures, NSERC, and SMART Technologies Inc.

REFERENCES

1. Ballagas, R., Ringel, M., Stone, M., and Borchers, J. iStuff: a physical user interface toolkit for ubiquitous computing environments. *Proc. of CHI'03*, ACM (2003).
2. Ballendat, T., Marquardt, N., and Greenberg, S. Proxemic Interaction: Designing for a Proximity and Orientation-Aware Environment. *Proc. of ITS'10*, ACM (2010).
3. Dey, A.K., et al. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comp. Int. 16*, 2, L. Erlbaum (2001), 97-166.
4. Diaz-Marino, R. and Greenberg, S. The proximity toolkit and ViconFace: the video. *Ext. Abst. CHI '10*, ACM (2010).
5. Feiner, S., et al. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies 1*, 4 (1997), 208-217.
6. Greenberg, S. and Fitchett, C. Phidgets: Easy Development of Physical Interfaces Through Physical Widgets. *Proc. of UIST'01*, ACM (2001), 209–218.
7. Greenberg, S., et al. Proxemic interactions: the new ubicomp? *interactions 18*, ACM (2011), 42–50.
8. Hall, E.T. *The Hidden Dimension*. Doubleday, 1966.
9. Hartmann, B., et al. Reflective physical prototyping through integrated design, test, and analysis. *Proc. UIST*, ACM (2006).
10. Hightower, J., et al. The location stack: A layered model for location in ubiquitous computing. *Proc. of WMCSA'02*, (2002).
11. Ishii, H. and Ullmer, B. Tangible Bits: Towards Seamless Interfaces Between People, Bits and Atoms. *Proc. of CHI'97*, ACM (1997), 234–241.
12. Ju, W., et al. Range: exploring implicit interaction through electronic whiteboard design. *Proc. of CSCW'08*, ACM (2008).
13. Klemmer, S.R., et al. Papier-Mache: Toolkit Support for Tangible Input. *Proc. of CHI'04*, ACM (2004), 399–406.
14. Kortuem, G., et al. Sensing and visualizing spatial relations of mobile devices. *Proc. of UIST'05*, ACM (2005), 93-102.
15. Krumm, J. and Hinckley, K. The NearMe wireless proximity server. *Lecture notes in computer science*, (2004), 283-300.
16. Li, Y., et al. Topiary: a tool for prototyping location-enhanced applications. *Proc. of UIST '04*, ACM (2004).
17. MacIntyre, B., et al. DART: a toolkit for rapid design exploration of augmented reality experiences. *Proc. of UIST'04*, ACM (2004).
18. Marquardt, N. and Greenberg, S. Distributed Physical Interfaces with Shared Phidgets. *Proc. of TEI'07*, ACM (2007).
19. Matthews, T., et al. A toolkit for managing user attention in peripheral displays. *Proc. of UIST '04*, ACM (2004).
20. Myers, B.A., et al. Past, Present, and Future of User Interface Software Tools. *TOCHI 7*, 1, ACM (2000), 3–28.
21. Reitmayr, G. et al. OpenTracker: A flexible software design for three-dimensional interaction. *Virt. Reality 9*, (2005).
22. Sandor, C. and Klinker, G. A rapid prototyping software infrastructure for user interfaces in ubiquitous augmented reality. *Pers. and Ubiq. Comp. 9*, (2005).
23. Streitz, N., et al. Ambient displays and mobile devices for the creation of social architectural spaces. In *Public and Situated Displays*. Kluwer, 2003, 387-410.
24. Vogel, D. et al. Interactive public ambient displays: transitioning from implicit to explicit, public to personal, interaction with multiple users. *Proc. of UIST'04*, ACM (2004).
25. Weiser, M. The Computer for the 21st Century. *Scientific American 265*, (1991), 94.