WorldKit: Rapid and Easy Creation of Ad-hoc Interactive Applications on Everyday Surfaces

Robert Xiao **Chris Harrison** **Scott E. Hudson**
Human-Computer Interaction Institute
Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213
{brx, chris.harrison, scott.hudson}@cs.cmu.edu

Figure 1. Using a projector and depth camera, the WorldKit system allows interfaces to operate on everyday surfaces, such as a living room table and couch (A). Applications can be created rapidly and easily, simply by "painting" controls onto a desired location with one's hand - a home entertainment system in the example above (B, C, and D). Touch-driven interfaces then appear on the environment, which can be immediately accessed by the user (E).

ABSTRACT
Instant access to computing, when and where we need it, has long been one of the aims of research areas such as ubiquitous computing. In this paper, we describe the WorldKit system, which makes use of a paired depth camera and projector to make ordinary surfaces instantly interactive. Using this system, touch-based interactivity can, without prior calibration, be placed on nearly any unmodified surface literally with a wave of the hand, as can other new forms of sensed interaction. From a user perspective, such interfaces are easy enough to instantiate that they could, if desired, be recreated or modified "each time we sat down" by "painting" them next to us. From the programmer's perspective, our system encapsulates these capabilities in a simple set of abstractions that make the creation of interfaces quick and easy. Further, it is extensible to new, custom interactors in a way that closely mimics conventional 2D graphical user interfaces, hiding much of the complexity of working in this new domain. We detail the hardware and software implementation of our system, and several example applications built using the library.

Author Keywords
Interactive spaces; touch; smart rooms; augmented reality; depth sensing; ubiquitous computing; surface computing.

ACM Classification Keywords
H.5.2. User Interfaces: Input devices and strategies; Interaction styles; Graphical user interfaces.

CHI 2013, April 27–May 2, 2013, Paris, France.

INTRODUCTION
Creating interfaces *in the world*, where and when we need them, has been a persistent goal of research areas such as ubiquitous computing, augmented reality, and mobile computing. In this paper we discuss the *WorldKit* system, which supports very rapid creation of touch-based interfaces on everyday surfaces. Further, it supports experimentation with other interaction techniques based on depth- and vision-based sensing, such as reacting to the placement of an object in a region, or sensing the ambient light in one area of a room.

Our work draws together ideas and approaches from many systems. For example, we draw on aspects of Everywhere Displays [26] conceptually, LightWidgets [7] experientially, and LightSpace [34] technically. Based on very simple specifications (in the default case, just a simple list of interactor types and action callbacks) interfaces can be created which allow users to quite literally *paint* interface components and/or whole applications wherever they are needed (Figure 1) and then immediately start using them. Interfaces are easy enough to establish that the user could, if desired, produce an interface on the fly each time they entered a space. This flexibility is important because unlike an LCD screen, the world around is ever-changing and configured in many different ways (e.g., our lab is different from your living room). Fortunately, we can bring technology to bear to overcome this issue and make best use of our environments.

Like LightSpace [34], our system makes use of a projector and inexpensive depth camera to track the user, sense the environment, and provide visual feedback. However, our system does not require advance calibration of the spaces it operates in – it can simply be *pointed at* nearly any indoor space. Further, with a projector slightly smaller than the one used in our prototype, it could be deployed in a volume

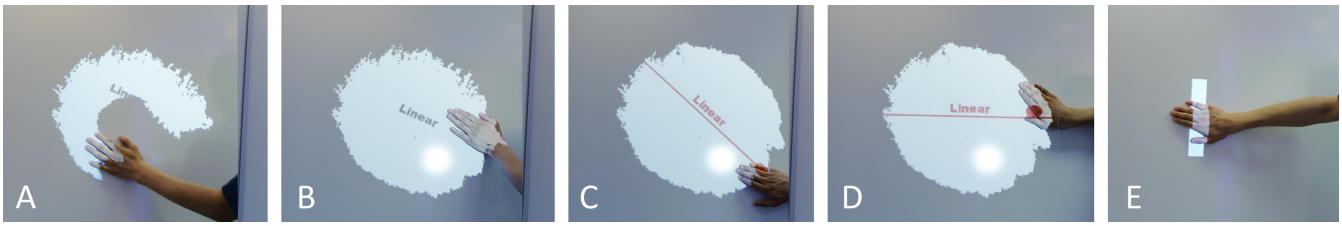

Figure 2. This sequence shows how a user can define the location, size and orientation of an interactor. First, the user starts painting an objects area (A, B), defining its location and size. Objects have a default orientation (C), which can be reoriented by dragging along the periphery (D). Finally, the interactor is instantiated and can be used (E).

similar to a modern laptop, and with likely future hardware advances (e.g., improved pico-projectors and smaller depth cameras) it may be possible to implement it in a truly mobile form. In addition, our system provides an extensible set of abstractions which make it easy and convenient to program simple interfaces while still supporting exploration of new interaction techniques in this domain.

In the next section, we will consider how users might make use of these created interfaces. We then turn to implementation details, discussing the hardware used, sensing techniques and other low level details. We will then consider how these basic capabilities can be brought together to provide convenient abstractions for *paint anywhere* interactive objects, which make them very similar to programming of conventional GUI interfaces. We then consider aspects of the software abstractions that are unique to this domain and describe an initial library of interactor objects provided with our system. Several example applications we built atop this library are also described. We conclude with a review of related work, noting that while previous systems have considered many of the individual technical capabilities built into our system in one form or another, the WorldKit system breaks new ground in bringing these together in a highly accessible form. The system enables both easy and familiar programmatic access to advanced capabilities, as well as a new user experience with dynamic instantiation of interfaces *when and where they are needed*.

INTERACTION

A core objective of our system is to make it simple for users to define applications quickly and easily, such that they *could* feasibly customize an application each time they used it. The default interaction paradigm provided by our system allows users to "paint" interactive elements onto the environment with their hands (Figures 1 and 2). Applications built upon this system are composed of one or more *in-the-world interactors*, which can be combined to create interactive applications.

By default, when an interface is to be deployed or redeployed, a list of interactor types and accompanying callback objects is provided – one for each element of the interface. The application *instantiates* each interactor using a specified instantiation method, defaulting to user-driven *painted* instantiations. For these elements, the system indicates to the user what interactor is to be "painted". The user then runs his or her hand over a desired surface (Figures 1B and 2A,B). Live graphical feedback reflecting the current selection is projected directly on the environment. When satisfied, the user lifts their hand. The system automatically selects an orientation for the interactor, which can optionally be adjusted by dragging a hand around the periphery of the selected area, as shown in Figure 2D. This completes the setup for a single interactor. The user then paints the next interface element, and so on.

Once all elements have been instantiated, the interface starts and can be used immediately. The entire creation process can occur very quickly. For example, the living room application sequence depicted in Figure 1 can be comfortably completed within 30 seconds. Importantly, this process need not occur every time – interactor placements can be saved by applications and reused the next time they are launched.

This approach offers an unprecedented level of personalization and responsiveness to different use contexts. For example, a typical living room has multiple seating locations. With our system, a user sitting down could instantiate a custom television interface using surfaces in their immediate vicinity. If certain controls are more likely to be used than others (e.g., channel switching), these can be placed closer to the user and/or made larger. Other functions could be omitted entirely. Moreover, users could layout functionality to match their ergonomic state. For example, if lying on a sofa, the arm rests, skirt or back cushions could be used because they are within reach.

Triggering Interfaces and Interface Design

Triggering the instantiation of an interface, including the design thereof can be achieved several ways. One option is for the system to be speech active. For example, the user could say "activate DVR" to bring up the last designed interface or "design DVR" to custom a new one. Alternatively, a free space gesture could be used, for example, a hand wave. A smartphone could also trigger interfaces and interface design, allowing for fine grain selection of functionality to happen on the touchscreen, and design to happen on the environment. Finally, a special (visible or invisible) environmental "button" could trigger functions.

SYSTEM IMPLEMENTATION

Hardware and Software Basics

Our system consists of a computer connected to a Microsoft Kinect depth camera mounted on top of a Mitsubishi

EX320U-ST short-throw projector (Figure 3). The Kinect provides a 320x240 pixel depth image and a 640x480 RGB image, both at 30 FPS. It can sense depth within a range of 50cm to 500cm with a relative error of approximately 0.5% [19]. Our short-throw projector has approximately the same field-of-view as the depth camera, allowing the two units to be placed in the same location without producing significant blind spots. As shown in KinectFusion [16], the depth scene can be refined over successive frames, yielding superior accuracy.

The software controlling the system is programmed in Java using the Processing library [27]. It runs on e.g., a MacBook Pro laptop with a 2GHz Intel Core i7 processor and 4 GB of RAM. The system runs at around 30FPS, which is the frame rate of the depth camera.

One-Time Projector / Depth Camera Calibration

We calibrate the joined camera-projector pair using a calibration target consisting of three mutually perpendicular squares of foamcore, 50cm on a side, joined at a common vertex. The seven non-coplanar corners of this target are more than sufficient to establish the necessary projective transform between the camera and projector, and the extra degrees of freedom they provide are used to improve accuracy via a simple least-squares regression fit.

As long as the depth camera remains rigidly fastened to the projector, the calibration above only needs to be performed once (i.e., at the factory). The setup can then be transported and installed anywhere – the depth sensor is used to automatically learn about new environments without requiring explicit steps to measure or (re-)calibrate in a new space. If the environment changes temporarily or permanently after interfaces have been defined by a user (e.g., a surface being projected on is moved), it may be necessary to re-define affected interfaces. However, our interactive approach to interface instantiation makes this process extremely lightweight for even novice users.

Basic Contact Sensing

Our system relies on surface contact sensing for two distinct purposes. First, when creating interfaces, touches are used to define interactor location, scale and orientation on the environment. This requires global touch sensing. Second, many interactor types (e.g., binary contact inputs), are driven by surface contact (i.e., touch or object contact or

Figure 3. A short throw projector with mounted Kinect.

presence) data. To achieve this, we mask the global scene with each interactor's bounds; data from this region alone is then passed to the interactor for processing. In some cases (e.g., counting interactor), additional computer vision operations are completed internally (e.g., connected components for blob detection).

To achieve the highest quality sensing possible, we employ several strategies to filter the depth image. First, when the system starts up, we capture 50 consecutive depth frames and average them to produce a background profile. Note that this implies that the scene must be stationary and in a "background" configuration when the system is initialized. It is also possible to automatically accumulate a background image over a longer period of time to provide some ability to handle dynamic reconfigurations, e.g., moved furniture, but our system does not currently do this. Within the background image, the standard deviation at each pixel location across the frames is used as a noise profile. Subsequently, each observed depth value is divided by the computed baseline deviation at that pixel, and values that are greater than 3 standard deviations from the mean are considered significant (Figure 4B). Significant pixels which differ from the background surface by at least 3mm and at most 50mm are considered candidate *contact pixels*. We then perform blob detection across these candidates using a connected-components algorithm to further eliminate erroneous pixels arising from noise in the depth sensor.

This process yields a number of *contact blob images* in the depth camera's coordinate space over each interactor (Figure 4C, red). Each image is projectively transformed into the local coordinate system of the corresponding interactor

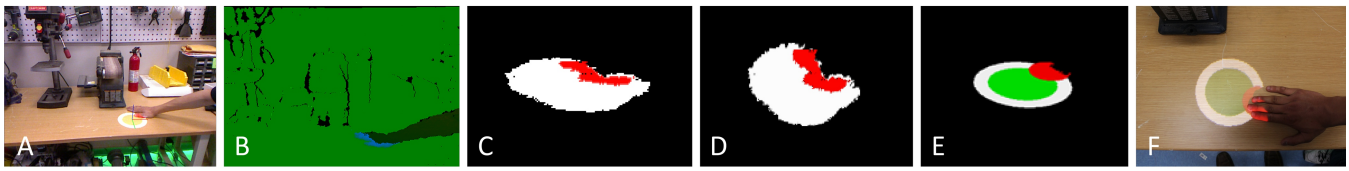

Figure 4. Touch event processing. User touches an interactor placed on a surface (A - view from Kinect). The depth differences from the background are computed (B - green, no significant difference; dark green, differences over 50mm; blue, candidate touch pixels). The candidate contact pixels (red) are masked by the interactor's depth-image mask (white) (C). The contact pixels are transformed into the interactor's local coordinate system, providing an orthographic view (D). For output, interactor graphics are warped into the projector's image space (E), so that they appear correctly on a surface (F).

(Figure 4D, red). From there, the blobs are passed to the corresponding interactor for type specific interpretation. For instance, the counting interactor from the system library simply uses the number of blobs intersecting that interactor, the multitouch interactor extracts the X-Y locations of each blob, and the area contact interactor determines the total number of pixels across all blobs within that interactor. Custom types extended from the library classes are free to perform additional processing for special purposes.

Software Structures

The WorldKit system provides a set of programming abstractions that aim both to make it very simple to create simple to moderately complex interfaces and to allow custom interaction techniques to be quickly created and explored in this new domain. Many aspects of the system structure are designed to be as close as possible to the abstractions now provided in nearly all conventional GUI interface toolkits.

For example, interfaces are constructed as trees of objects, which inherit from a base *interactor* class (which has been called a *component*, or *widget* in various other systems). That class establishes the central abstraction for the system and defines an API (and default implementations) for a variety of interface tasks such as: hierarchy (parent/child) management, event-oriented input handling, damage tracking, layout, interface (re)drawing, etc. Since our goal is to stay as close to existing abstractions as we can, we expect that many aspects of the system will already be familiar to developers. See Figure 7 for a complete sample application.

To create a new interactor type (primitive), the developer extends the base interactor class (or an existing interactor class with similar functionality), adding new event sources, drawing commands and interaction logic. This is functionally similar to how developers would create new interactors in e.g., Java Swing.

In the following sections we only consider the aspects of the system that are different from typical systems (e.g., interactor instantiation by end users) or require special treatment inside our system to make them appear ordinary (e.g., rectification between 2D drawing and input spaces and surfaces in the 3D world).

Instantiating Interactors

One major difference between WorldKit abstractions and typical GUI toolkits is in how interactors are instantiated. In conventional systems, the details of instantiation are typically determined simply by the parameters to the constructor for an interactor (which may come from a separate specification such as an XML document, and/or are originally determined with a visual layout editor).

In contrast, in WorldKit we provide three options for interactor instantiation: *painted*, *linked*, and *remembered*. By default, interactors use painted instantiation – allowing the user to establish their key properties by "painting" them on the world as described below. For the base interactor class, key properties include size, position, and orientation, but this may be defined differently in specialized subclasses. Alternately, the developer may ask for *linked* instantiation. In that case, a small bit of code is provided to derive the key properties for the interactor from another instantiated interactor. This allows, for example, one key interactor to be painted by the user, and then a related group of components to be automatically placed in relation to it. Finally, *remembered* instantiation can be performed using stored data. This data can come from the program (making it equivalent to conventional interactor instantiation) or from a data structure saved from a previous instantiation of the same interface. This allows, for example, an interface element to be placed "where the user last left it".

For painted instantiations, users define interactor size, location and initial orientation by using a hand painting gesture over the surface where they wish it to appear (Figure 2A). During this process an area for the interactor is accumulated. At each step the largest contact blob over the entire depth image is considered. If this blob is larger than a preset threshold, the blob is added to a mask for the interactor. If no blob is larger than the threshold, we determine that the user must have lifted their hand from the surface, and the accumulated mask is saved as the user's painting selection. Note that this mask is defined over the depth image (i.e., in depth image coordinates). We then take the (x,y,depth) points in the depth image indexed by the mask and transform them into a world-space point cloud. Averaging the surface normals over this point cloud produces the Z-axis of the planar region to be associated with the interactor. The X- and Y-axes lie in this plane, and their direction is controlled by the interactor's orientation.

The initial orientation aligns the Y-axis with the Y-axis of the depth image, which roughly corresponds to the direction of gravity if the depth sensor is mounted horizontally. As mentioned previously, the user can adjust the orientation by touching the interactor (Figure 2D).

Geometry Rectification for Input and Output

To provide a convenient API for drawing and input interpretation, the geometry of each interactor in our system is established in terms of a planar region in 3D space, derived as indicated above. Based on the X-, Y- and Z-axes of the interactor in the depth camera/projector coordinate system, we derive a *rectification matrix* mapping depth image coordinates into a local coordinate system for the interactor. This local coordinate system allows the developer to think about interaction drawing and input in simple 2D or surface terms. Full 3D information is available for use by advanced interactor classes if desired.

For input processing, the underlying depth and RGB images are updated 30 times per second. For each update we perform contact blob extraction as outlined earlier. For each interactor, we then intersect both the contact blob point cloud and the full depth image with the interactor's depth-image mask (Figure 4C, white). This produces raw depth

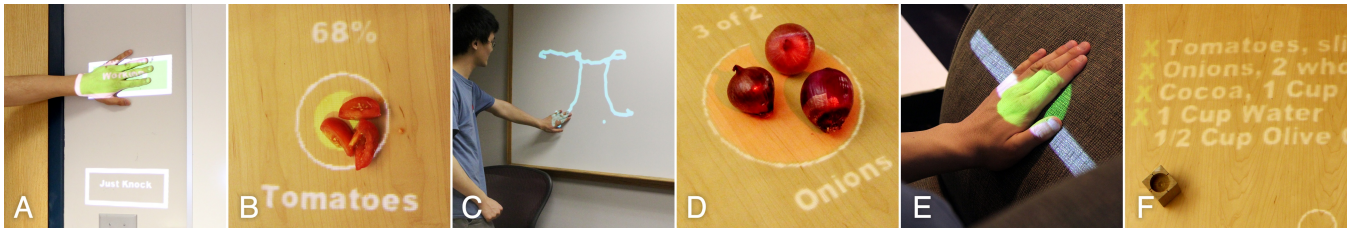

Figure 5. Our system provides a library of *interactor* classes which can be extended to perform many tasks. These including a binary contact interactor (A), percentage contact interactor (B), multitouch surface (C), object counting interactor (D), a linear axis interactor (E), as well as a simple output-only interactor (F). See also Table 1.

and RGB images as well as contact areas limited to the region over the interactor. The rectification matrix is then applied to individual interactor depth, RGB, and contact images to produce *rectified* images (Figure 4D), i.e. images represented in the interactor's local 2D coordinate system. In a rectified image, one pixel width corresponds to a known unit of distance on the real world. Finally, contact areas are further processed to produce simplified touch events. All of this information is then passed to the interactor(s) concerned.

At this point, the interactor may perform additional specialized processing depending on their type. For instance, a brightness interactor from our library will calculate its sensed brightness value based on the rectified RGB image, a contact interactor will update its touch state and fire pressed/released events if applicable, and a multitouch interactor will act based on the contact blobs visible in its rectified image.

Each interactor may also produce output in order to indicate its current state and provide feedback during interaction. To facilitate easy drawing, the system provides a conventional two-dimensional drawing context (a PGraphics object within the Processing system) which is passed to an interactor's draw() method as needed. This drawing context object is transformed so that interactor drawing is specified in real world units (e.g., millimeters) and oriented to correspond to the interactor's real-world orientation (e.g., aligned with its derived planar coordinate system as described above). The system takes draw commands on these graphics surfaces and automatically transforms them into the projector's image space for display (Figure 4E). Thus, when projected, interfaces render correctly on surfaces regardless of projector perspective, honoring interface layout and dimensions (Figure 4F). Finally, because we are projecting imagery onto real-world objects, head tracking is not required.

Interactor Type	Type of Associated Value
Binary contact	True or False
Area contact	Percentage of coverage
Presence	True or False
Contact counting	Number of items (contact blobs)
Linear axis touch	Centroid of touch (1D along axis)
Two axis touch	X/Y centroid of touch
Radial input touch	Angle to centroid of touch
Multitouch input	X/Y centroid of multiple touches
Brightness	Average brightness of surface
Color	Average color of surface

Table 1. Input-oriented base classes from our interactor library, organized by type of input

Interactor Library

As a part of our system, we created an initial interactor library to support various capabilities of the platform (Figure 5). Part of this library is a set of reusable input-oriented base classes (listed in Table 1). From these base classes, we derived a set of traditional UI elements featuring both input and output, such as buttons and sliders.

As examples: the binary contact interactor detects events on a surface by examining the set of contact blobs reported to it. If the total pixel count for these exceeds a small threshold (to filter out noise), the interactor detects a contact/touch. The area contact interactor additionally provides the proportion of depth values that are considered to be in contact range, allowing it to measure the contacted area. The presence interactor detects whether a background object is still present in its original configuration. For example, this can be used to sense if a door has been opened or a screen has been retracted.

A counting interactor counts and reports the number of distinct contact blobs on its surface. Linear axis interactors detect the position of a touch along one axis, which can be used to implement a variety of sliding controls, and multitouch interactors report the X and Y positions of each individual blob. Brightness interactors use the RGB camera feed to detect changes in the brightness of the sensed area. Similarly, color-sensing interactors measure the average color.

As in most systems, interactor types in our system are organized into a class hierarchy and may be extended from classes in the library by overriding methods associated with various interactive tasks. For example, advanced users might override an appropriate class to perform new or more advanced input processing such as hand contour analysis for user identification [29] or recognition of shoes [3].

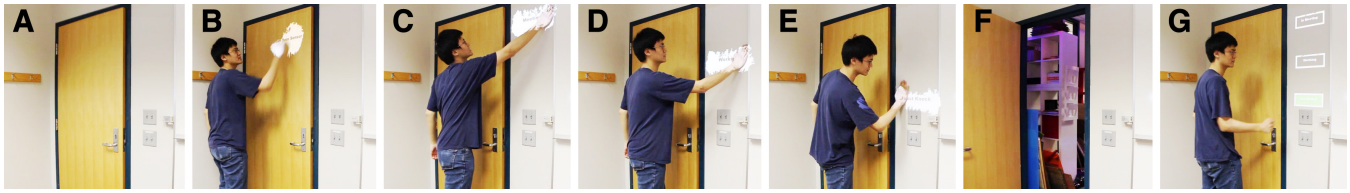

Figure 6. This sequence illustrates a user setting up a simple notification message application on an instrumented office (A). First, a user "paints" a presence interactor on an office door (B), which can detect if the door is open or closed. The user then paints three contact interactors, labeled "in meeting", "working" and "just knock", onto the wall adjacent to the door (C, D and E). When the door is open, the application is invisible (F). When the door is closed, three buttons appear, which are user selectable (G).

EXAMPLE APPLICATIONS

To illustrate the utility and capability of our system, we describe several example applications built using the accompanying library. Please see the accompanying Video Figure for a demonstration of each.

Living Room

The television remote control is frequently misplaced or lost in a typical home, leading to much consternation. With WorldKit, any surface can host the controls. This application instantiates a linear interactor to adjust the room's brightness, a radial interactor to adjust the TV volume and a Digital Video Recorder (DVR) interface to select a show of interest (Figure 1). Additionally, by adding a presence interactor to the sofa, we can even determine if the user is sitting and show or hide the interface as needed.

Office Door

A closed office door often gives no hints about the interruptible state of the occupant. A simple application of the WorldKit system allows the occupant to convey their status quickly when the door is closed. On the inside of the office, a large presence interactor is drawn on the closed door, and a number of smaller status buttons are drawn to the side (Figure 6). When the door is open, the status buttons are hidden, and the exterior indicator shows nothing. With the door closed, the status buttons appear. The exterior indicator reflects the chosen status button; thus, for instance, selecting the "In Meeting" status might cause "I'm in a meeting; please do not disturb" to appear on the outside.

Office Desk

This application uses a triggering contact interactor (in this case positioned over a keyboard) to activate a calendar display (which itself uses a linear interactor for scrolling) and a 2D position interactor for a simple whiteboard (Figure 8). When the user places his hands on the keyboard, the calendar appears. The user may then scroll the display through the calendar day by simply dragging up and down. Removing the hands from the keyboard causes the calendar to disappear. Users can also draw on the whiteboard, which is always visible regardless of the trigger state.

Kitchen

In the kitchen, it often becomes a chore to keep track of all the ingredients needed for a complex recipe. To address this, we created a simple recipe helper interface. The application prompts users to select a suitably-sized flat surface (e.g. kitchen counter) to prepare their ingredients. The user selects the desired recipe, and the application automatically lays out a set of interactors within that flat surface to hold each ingredient (Figure 9).

Interactors are customized for each ingredient to measure the amount or presence of the requested ingredient. For instance, if the recipe calls for a small number of countable items (e.g., eggs, whole onions), a counting interactor can be used. Ingredients that cannot be measured easily in the framework can be replaced by contact interactors, which simply record the presence or absence of the ingredient.

The flexibility of our system enables the interface to be quickly reconfigured to suit different sets of ingredients and quantities, without any cumbersome calibration or instrumentation.

LIMITATIONS

As it is currently implemented, our system has two notable drawbacks: the resolution of sensing and graphics can be lower than optimal, and the user may occlude the depth sensor and/or projector in certain configurations.

```
import worldkit.Application;
import worldkit.interactors.Button;
import worldkit.interactors.ContactInput.ContactEventArgs;
import worldkit.util.EventListener;

public class OneButtonApp extends Application {
  Button button;

  public void init() {
    button = new Button(this);
    button.contactDownEvent.add(
      new EventListener<ContactEventArgs>() {
        @Override
        public void handleEvent(Object sender,
                                ContactEventArgs args) {
          System.err.println("Got a button event!");
        }
      });
      button.paintedInstantiation("OneButton");
  }

  /* Boilerplate */
  public static void main(String[] args) {
    new OneButtonApp().run();
  }
}
```

Figure 7. Example code for a single button application. The application depicted in Figure 6 consists of three buttons.

In the current system, the projector displays imagery at a resolution of 1024x768 over a potentially wide area. In cases where this area is large, this results in a loss of visual detail. We feel that this is not an inherent flaw in the approach, but rather a technological limitation that will improve with time as projectors develop increased resolution. Similarly the Kinect depth camera is also limited in spatial, temporal and depth resolution. However, future depth cameras promise to overcome these limitations. Finally, we note that for the interactions presented in this paper, lack of resolution did not significantly impede the usability of the resulting applications.

Users may occlude the projector and/or depth camera during normal operation; this is fundamentally a limitation of using a single projector and camera setup. In our system, we avoid user confusion by positioning the Kinect on top of the projector (i.e., very closely aligning effective view and display frustums). This ensures that users receive feedback in the form of their own shadow if they occlude the view of the camera.

RELATED SYSTEMS

The notion of having computing everywhere has been a grand challenge for the HCI community for decades [31,32]. Today, users have achieved ubiquitous computing not through ubiquity of computing infrastructure, but rather by carrying sophisticated mobile devices everywhere we go (e.g., laptops, smartphones) [13]. This strategy is both cost effective and guarantees a minimum level of computing quality. However, by virtue of being portable, these devices are also small, which immediately precludes a wide range of applications.

In contrast, the environment around us is expansive, allowing for large and comfortable interactions. Moreover, applications can be multifaceted, supporting complex tasks, and allowing for multiple users. And perhaps most importantly, the environment is already present – we do not need to carry it around. These significant benefits have spawned numerous research systems for interacting beyond the confines of a device and on the actual surfaces of the world.

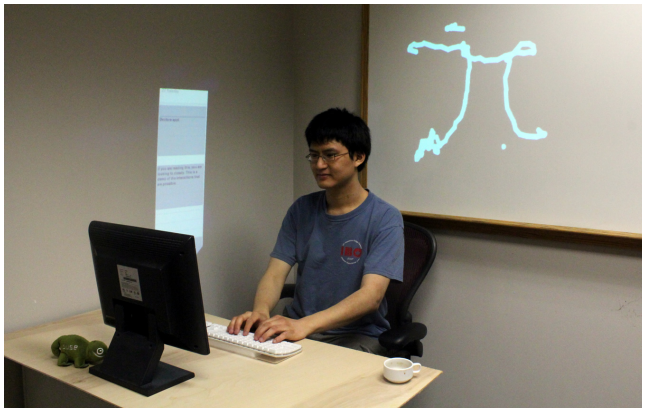

Figure 8. Simple office application. The whiteboard behind the user is an interactor. The calendar to the right of the user is visible only when the user's hands are on the keyboard.

A broad range of technical approaches has been considered for appropriating the environment for interactive use, including acoustic [11] and electromagnetic sensing [6]. A popular alternative has been camera/projector systems. By using light, both for input (sensing) and output (projection), system components can be placed out of the way, yet provide distributed functionality. This is both minimally invasive and potentially reduces the cost of installation (i.e. not requiring substantial wiring for sensors).

Interfaces in and on the World

Seminal work on such systems was initiated in the late 1990s. An early project, *The Intelligent Room* [5], eloquently described their objective as: "Rather than pull people into the virtual world of the computer, we are trying to pull the computer out into the real world of people." The system used cameras to track users, from which a room's geometry can be estimated. A pair of projectors allows one wall to be illuminated with interactive applications. Additional cameras were installed on this wall at oblique angles, allowing for finger touches to be digitized.

Of note, *The Intelligent Room* required all interactive surfaces be pre-selected and calibrated. The goal of the *Office of the Future* [28] was to enable users to designate any surface as a "spatially immersive display." This was achieved by capturing the 3D geometry of surfaces through structured light. With this data, interfaces could be rectified appropriately, and potentially updated if the environment was dynamic (as could [17]). The authors also experimented with head tracking (via a separate magnetically-driven system) so as to provide interfaces that appeared correct irrespective of a user's viewpoint, even when projecting on irregular surfaces. Although calibration to a surface would be automatic, the work does not describe any user mechanisms for defining surfaces. Once applications were running on surfaces, the system relied on conventional means of input (e.g., keyboard and mouse).

The Luminous Room [31] is another early exploration of projector/camera-driven interaction. It was unique in that it

Figure 9. Kitchen application. Various interactor types are composed into an ingredient management interface.

enabled simple input on everyday projected surfaces. Through computer vision, objects could be recognized through fiducial markers. It is also suggested that the silhouette of hands could be extracted, and incorporated into interactive applications. The system used a single conventional camera, so presumably hover/occlusion could not be easily disambiguated from touch. Like *The Intelligent Room*, this system also required pre-calibration and configuration of surfaces before they could be used.

The *Everywhere Displays* project [26] used a steerable mirror to enable a single projector to output dynamic graphics on a variety of office surfaces. To correct for distortion, a camera was used in concert with a known projected pattern. Using this method, a 3D scene could be constructed for desired projection surfaces. The authors speculate that touch sensing could be added by stereo camera sensing or examining shadows.

More recently, *Bonfire* [18] – a laptop mounted camera/projection system – enables interactive areas on either is side of the laptop. Because the geometry of the setup is known a priori, the system can be calibrated once, allowing graphics to be rendered without distortion despite oblique projection. Touch interaction is achieved by segmenting the fingers based on color information, and performing a contour analysis. The desk-bound nature of laptops means *Bonfire* also intersects with "smart desk" systems (see e.g., [20,33]), which tend to be static, controlled infrastructure (i.e., a special desk). Although both setups provide opportunities for interactive customization, the context is significantly different.

The advent of low cost depth sensing has led to a resurgence of interactive environment projects. A single depth camera can view a large area and be used to detect touch events on everyday surfaces [35,36]. *LightSpace* [34] uses an array of calibrated depth cameras and projectors to create a live 3D model of the environment. This can be used to track users and enable interaction with objects and menus in 3D space. Of particular relation to our work, *LightSpace* can create virtual orthographic cameras in a specified volume. For example, this is used to create thin planar volumes that can be used as multitouch sensors – one of the primitives our system provides. Few details about *OASIS* system [25] are public, though it appears to have similar capabilities and goals.

OmniTouch [12] is a worn depth camera and projection system that enables multitouch finger interaction on ad hoc surfaces, including fixed infrastructure (e.g., walls), handheld objects (e.g., books), and even users' bodies. Applicable surfaces within the system's field of view (~2m) are tracked and an approximate real-world size is calculated, allowing interfaces to be automatically scaled to fit. Orientation is estimated by calculating an object's average surface normal and second moment, allowing for rectified graphics. Users can "click and drag" on surfaces with a finger, which sets an interface's location and physical dimensions – a very simple example of user-defined interfaces. Finally, [17] allowed for interactive projections onto physical setups constructed from passive blocks; user interaction is achieved with an IR stylus.

User-Defined Interfaces

An important commonality of the aforementioned systems is a lack of end-user mechanisms for defining interactive areas or functionality. There is, however, a large literature regarding user defined interactions, going back as far as command line interfaces [9] and extending to the present, with e.g., unistroke characters [37] and touch screen gestures [24,38]. More closely related to our work is user-driven interface layout and composition. For example, end-users can author interfaces by sketching widgets and applications [21], which is far more approachable than traditional GUI design tools. Research has also looked into run-time generation of user interfaces based on available I/O devices, the task at hand, and user preferences and skill [8].

When moving out into the physical world, easily defining interfaces is only half the problem. Equally challenging is providing easy-to-use end-user tools that allow instrumentation of the physical world. Sensors, microprocessors and similar require a degree of skill (and patience) beyond that of the typical user. Hardware toolkits [2,10,14,22] were born out of the desire to lower the barrier to entry for building sensor driven applications. There have also been efforts to enable end users to easily create custom, physical, interactive objects with off the shelf materials, for example, Styrofoam and cardboard [1,4,15].

Most closely related to our technical approach are virtual sensing techniques – specifically, approaches that can sense the environment, but need not instrument it. This largely implies the use of cameras (though not exclusively; see e.g., [6,11]). By remote sensing on e.g., a video feed, these systems can sidestep many of the complexities of building interfaces physically onto the environment.

One such project is *Eyepatch* [23], which provides a suite of computer vision based sensors, including the ability to recognize and track objects, as well as respond to user gestures. These complex events can be streamed to user-written applications, greatly simplifying development. *Slit-Tear Visualizations* [30], although not used as inputs per se, are conceptually related. The interface allows users to draw sensing regions onto a video steam, which, through a simple visualization, allow users to readily distinguish environmental events, such as cars passing. Similarly, *Light Widgets* [7] allows users to select regions on everyday surfaces using a live video feed for sensing. Three widget types are provided: button, linear and radial. Sensing is achieved with a pair of cameras set apart (to disambiguate occlusion from touch on surfaces); user actions are detected by finding skin-colored blobs.

Discussion

As indicated above, many of the underlying technical components we bring together in our system have been consid-

ered in prior work, and in some cases more than one technical approach has been offered over time. Specifically, we were initially inspired by the visions put forward in the *Intelligent Room*, *Luminous Room,* and *Everywhere Displays* projects. The benefits and goals of having interactivity everywhere were clearly articulated in these early works. However, the instantiation and modification of interactive features was absent or limited. In general, example applications were custom built by the authors, carefully configured and calibrated, and largely inflexible.

Eyepatch, *Slit-Tear Visualizations* and *Light Widgets* put forward elegant approaches to allow end-users to quickly define simple interfaces on a video stream of the environment. We extend this idea to defining interfaces *in situ* – directly on the environment, without the need for a conventional computer. Furthermore, we expand the suite of controls, allowing for richer interactions, including multitouch input and non-human triggers like doors closing.

Moreover, our system projects coordinated graphical feedback onto user-defined areas. This builds on and provides reusable abstractions for the technical approaches presented in *Office of the Future*, *LightSpace*, and *OmniTouch*. Our system takes into account the geometry of user-defined surfaces, providing not only rectified projected output (so graphics appear correctly for all users), but also orthonormal processing of input data (providing higher accuracy and regular coordinate systems).

Only with *all* of these features in place (and with the functionality and low cost of the most recent hardware advances) could we begin to think about mechanisms that are suitable for end users to define interactive functions on everyday surfaces in a highly dynamic fashion. This has allowed us to produce a robust, extensible, and usable system that makes interfaces accessible where ever and whenever they are needed by an end-user.

FUTURE WORK

Although our system provides a very general set of interactive capabilities, there are several areas for future work. First, we have yet to fully explore the design space of interactors residing on real-world surfaces. Based on a wider exploration of this space enabled by our initial extensible tool, we anticipate that a more substantial and complete library of interactors (built with the existing extension mechanism) will increase the variety of applications that could be supported.

Another area of potential future work involves the expansion from surface interaction into free space interaction. In this area there are interesting challenges in determining how input spaces might be delimited and how end-users might quickly and easily instantiate interactors. In addition, there are significant interaction technique design challenges for this type of interaction. For example, it is as yet unclear what a base set of interaction types for this space might include. Furthermore, there are basic challenges such as how to provide feedback without an obvious surface to project coordinated graphical feedback onto.

As the underlying hardware for both projection and depth sensing improves, additional challenges and opportunities may arise in improved filtering, detection, recognition, and display. For example, with higher resolution depth cameras, it may be possible to include detailed finger gestures as a part of interaction, but recognition of these will be challenging. In addition, recognizing classes of everyday objects could introduce substantial new capabilities into this type of system.

Finally, an additional area for future research lies in the expansion of these techniques to other modalities. For example, the use of audio input and output in conjunction with touch offers potential benefits.

CONCLUSION

We have described our WorldKit system, which allows interactive applications to flourish on the real world. This system provides a convenient and familiar set of programming abstractions that make this new domain accessible for development and experimentation. Further, it supports applications that can be readily instantiated and customized by users with unprecedented ease. Additionally, our approach overcomes many challenges inherent in sensing on the environment and with low-resolution depth-sensing. As discussed in our future work, forthcoming improvements in depth cameras will one day enable touchscreen-quality interaction on the world around us. Our system also projects coordinated graphical feedback, allowing collections of interactors to largely operate as if they were sophisticated but conventional GUI applications. Overall, we believe this work is an important step in achieving the promise of the ubiquitous computing vision.

ACKNOWLEDGMENTS

Funding for this work was provided in part by a Qualcomm Innovation Fellowship, a Microsoft Ph.D. Fellowship, grants from the Heinz College Center for the Future of Work, NSERC, and NSF grant IIS-1217929.

REFERENCES

1. Akaoka, E., Ginn, T., and Vertegaal, R. DisplayObjects: prototyping functional physical interfaces on 3d styrofoam, paper or cardboard models. In *Proc. TEI '10*. 49-56.
2. Arduino. http://www.arduino.cc
3. Augsten, T., Kaefer, K., Meusel, R., Fetzer, C., Kanitz, D., Stoff, T., Becker, T., Holz, C., and Baudisch, P. Multitoe: high-precision interaction with back-projected floors based on high-resolution multi-touch input. In *Proc. UIST '10*. 209-218.
4. Avrahami D., and Hudson, S.E., Forming interactivity: a tool for rapid prototyping of physical interactive products. In *Proc. DIS '02*, 141-146.
5. Brooks, R. A. The Intelligent Room Project. In *Proc. International Conference on Cognitive Technology '97*, 271.

6. Cohn, G., Morris, D., Patel, S.N., and Tan, D.S. Your noise is my command: sensing gestures using the body as an antenna. In *Proc. CHI '11*. 791-800.
7. Fails, J.A. and Olsen, D. Light widgets: interacting in every-day spaces. In *Proc. IUI '02*. 63-69.
8. Gajos, K.Z., Weld, D.S., and Wobbrock, J.O. Automatically generating personalized user interfaces with Supple. *Artificial Intelligence*, vol. 174, 12-13 (August 2010), 910-950.
9. Good, M.D., Whiteside, J.A., Wixon, D.R., and Jones, S.J. Building a user-derived interface. *Commun. ACM* 27, 10 (October 1984), 1032-1043.
10. Greenberg, S. and Fitchett, C. Phidgets: easy development of physical interfaces through physical widgets. In *Proc. UIST '01*. 209-218.
11. Harrison, C. and Hudson, S.E. Scratch input: creating large, inexpensive, unpowered and mobile finger input surfaces. In *Proc. UIST '08*. 205-208.
12. Harrison, C., Benko, H., and Wilson, A.D. OmniTouch: wearable multitouch interaction everywhere. In *Proc. UIST '11*. 441-450.
13. Harrison, C., Wiese, J., and Dey, A. K. "Achieving Ubiquity: The New Third Wave." *IEEE Multimedia*, 17, 3 (July-September 2010), 8-12.
14. Hartmann, B., Klemmer, S.R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., and Gee, J., Reflective physical prototyping through integrated design, test, and analysis. In *Proc. UIST '06*, 299-308.
15. Hudson, S.E. and Mankoff, J. Rapid construction of functioning physical interfaces from cardboard, thumbtacks, tin foil and masking tape. In *Proc. UIST '06*. 289-298.
16. Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A. and Fitzgibbon, A. KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In *Proc. UIST '11*. 559-568.
17. Jones, B., Sodhi, R., Campbell, R., Garnett, G., and Bailey, B. Build your world and play in it: Interacting with surface particles on complex objects. In *Proc. ISMAR '10*. 165 - 174.
18. Kane, S.K., Avrahami, D., Wobbrock, J.O., Harrison, B., Rea, A.D., Philipose, M., and LaMarca, A. Bonfire: a nomadic system for hybrid laptop-tabletop interaction. In *Proc. UIST '09*. 129-138.
19. Khoshelham K., Elberink S.O. Accuracy and resolution of Kinect depth data for indoor mapping applications. *Sensors*, 12(2), 2012. 437-54.
20. Koike, H., Sato, Y. and Kobayashi, Y. Integrating paper and digital information on EnhancedDesk: a method for realtime finger tracking on an augmented desk system. *ACM Trans. on Computer-Human Interaction*, 8 (4), 307-322.
21. Landay, J.A. and Myers, B.A. Sketching Interfaces: Toward More Human Interface Design. *Computer* 34, 3 (March 2001), 56-64.
22. Lee, J.C., Avrahami, D., Hudson, S.E., Forlizzi, J., Dietz, P., and Leigh, D. The calder toolkit: wired and wireless components for rapidly prototyping interactive devices. In *Proc. DIS '04*, 167-175.
23. Maynes-Aminzade, D., Winograd, T., and Igarashi, T. Eyepatch: prototyping camera-based interaction through examples. In *Proc. UIST '07*. 33-42.
24. Nielsen, M., Störring, M., Moeslund, T.B. and Granum, E. (2004) A procedure for developing intuitive and ergonomic gesture interfaces for HCI. *Int'l Gesture Workshop 2003*, LNCS vol. 2915. Heidelberg: SpringerVerlag, 409-420.
25. Object Aware Situated Interactive System (OASIS). Intel Corporation. Retrieved April 7, 2012: http://techresearch.intel.com/ProjectDetails.aspx?Id=84
26. Pinhanez, C.S. The Everywhere Displays Projector: A Device to Create Ubiquitous Graphical Interfaces. In *Proc. UbiComp '01*. 315-331.
27. Processing. http://www.processing.org
28. Raskar, R., Welch, G., Cutts, M., Lake, A., Stesin, L., and Fuchs, H. The office of the future: a unified approach to image-based modeling and spatially immersive displays. In *Proc. SIGGRAPH '98*. 179-188.
29. Schmidt, D., Chong, M. K. and Gellersen, H. HandsDown: hand-contour-based user identification for interactive surfaces. In *Proc. NordiCHI '10*. 432-441.
30. Tang, A., Greenberg, S., and Fels, S. Exploring video streams using slit-tear visualizations. In *Proc. AVI '08*. 191-198.
31. Underkoffler, J., Ullmer, B., Ishii, H. Emancipated pixels: real-world graphics in the luminous room. In *Proc. SIGGRAPH '99*. 385-392.
32. Weiser, M. The Computer for the 21st Century. *SIGMOBILE Mob. Comput. Commun*. Rev. 3, 3 (July 1999), 3-11.
33. Wellner, P. Interacting with paper on the DigitalDesk. *Communications of the ACM*, 36 (7), 87-96.
34. Wilson, A.D. and Benko, H. Combining multiple depth cameras and projectors for interactions on, above and between surfaces. In *Proc. UIST '10*. 273-282.
35. Wilson, A.D. Using a depth camera as a touch sensor. In *Proc. ITS '10*. 69-72.
36. Wilson, A.D., Depth-Sensing Video Cameras for 3D Tangible Tabletop Interaction. In *Proc. Tabletop '07*, 201-204.
37. Wobbrock, J.O., Aung, H.H., Rothrock, B., and Myers, B.A. 2005. Maximizing the guessability of symbolic input. In *CHI '05 EA*. 1869-1872.
38. Wobbrock, J.O., Morris, M.R., and Wilson, A.D. User-defined gestures for surface computing. In *Proc. CHI '09*. 1083-1092.