2-2013

KinectArms: A toolkit for capturing and displaying arm embodiments in distributed tabletop groupware

Aaron GENEST

Carl GUTWIN

Anthony TANG
Singapore Management University, tonyt@smu.edu.sg

Michael KALYN

Zenja IVKOVIC

Citation

GENEST, Aaron; GUTWIN, Carl; TANG, Anthony; KALYN, Michael; and IVKOVIC, Zenja. KinectArms: A toolkit for capturing and displaying arm embodiments in distributed tabletop groupware. (2013). *CSCW '13: Proceedings of the 2013 conference on Computer supported cooperative work, San Antonio, Texas, USA, February 23-27*. 157-166.

Available at: https://ink.library.smu.edu.sg/sis_research/7992

KinectArms: A Toolkit for Capturing and Displaying Arm Embodiments in Distributed Tabletop Groupware

Aaron Genest[1], **Carl Gutwin**[1], **Anthony Tang**[2], **Michael Kalyn**[1], **and Zenja Ivkovic**[1]
[1]Department of Computer Science
University of Saskatchewan
110 Science Place, Saskatoon, Canada
[2]Department of Computer Science
University of Calgary
2500 University Dr. NW, Calgary, Canada
[aaron.genest, carl.gutwin, michael.kalyn, zenja.ivkovic]@usask.ca, tonyt@ucalgary.ca

ABSTRACT
Gestures are a ubiquitous part of human communication over tables, but when tables are distributed, gestures become difficult to capture and represent. There are several problems: extracting arm images from video, representing the height of the gesture, and making the arm embodiment visible and understandable at the remote table. Current solutions to these problems are often expensive, complex to use, and difficult to set up. We have developed a new toolkit – KinectArms – that quickly and easily captures and displays arm embodiments. KinectArms uses a depth camera to segment the video and determine gesture height, and provides several visual effects for representing arms, showing gesture height, and enhancing visibility. KinectArms lets designers add rich arm embodiments to their systems without undue cost or development effort, greatly improving the expressiveness and usability of distributed tabletop groupware.

Author Keywords
Distributed tabletops, gestures, embodiments, toolkits

ACM Classification Keywords
H.5.3 Group and Organization Interfaces: CSCW.

INTRODUCTION
Gestural communication is a ubiquitous and important part of co-located collaboration at real-world tables. People gesture over tables in many different ways, including pointing at objects, indicating paths and areas, emphasizing elements of the conversation, and illustrating actions [2,6]. At distributed tables, gestural communication is equally important, but gestures become much more difficult to reproduce compared to face-to-face environments. There are three main problems.

First, the complexity and subtlety of many over-the-table gestures requires that systems use video-based embodiments of hands and arms (e.g., [12,18,23]). However, extracting images of people's arms from the table background can be computationally expensive, prone to error (particularly if color is used for separation), and dependent on good lighting conditions [28].

Second, information about the height of the gesture can be a critical aspect of the communication [10], but height information is usually lost in distributed settings. Gesture height is difficult to capture (traditionally requiring expensive tracking technologies), and is difficult to convey through arm embodiments that are displayed as 2D images on the remote table surface.

Third, the reduced physical presence of remote arm embodiments, coupled with low frame rates and network jitter, make remote gestures difficult to see and interpret. Prior work has suggested visual traces as a way to increase the salience of a remote gesture [9,24], but traces are difficult to gather when arms move above the table.

The result of these three problems is that gestural communication in distributed tabletop groupware is much less expressive than at co-located tables. To address these problems, we have developed a new toolkit – called KinectArms – for capturing gestures over tables and displaying arm embodiments at remote sites (Figure 1).

Figure 1: KinectArms used in a photo-sharing application, showing shadow and height indictor (circle).

KinectArms solves all three of the issues mentioned above. First, it uses a depth camera to quickly and efficiently extract arm images from the video stream – the system runs easily at 30 frames per second. Second, KinectArms uses the depth camera to identify fine-grained height information about the gesture, and can attach that information to

CSCW '13, February 23–27, 2013, San Antonio, Texas, USA.

extracted features of the images such as hands and fingers. Third, KinectArms provides several visualization techniques to show gesture height, to enhance the visibility of embodiments, and to highlight arm movements (e.g., abstract height indicators, color and transparency modifications, and motion lines and traces). The toolkit also provides several other capabilities, including automatic calibration to the table surface, tracking of arms in the scene, and wrappers to allow development in several programming languages.

In this paper we describe the design, implementation, and use of KinectArms. Our work makes three contributions.

- KinectArms is the first toolkit to provide a simple solution to all the aspects of remote arm embodiments, including capture and representation of gesture height. It goes well beyond standard Kinect libraries, which are not set up to recognize arms over table surfaces.
- We demonstrate that KinectArms can reproduce a wide variety of previous designs (e.g., DOVE, VideoArms, and C-Slate) and show that the platform can be used to produce new representations as well, providing an opportunity to compare different designs.
- We evaluate the toolkit in terms of performance, developer effort, expressiveness, and simplicity, and show that KinectArms provides a powerful yet easy-to-use solution to the problem of remote gestures.

With KinectArms, designers can easily add support for remote gesture to tabletops, which can dramatically improve the usability and communicative capabilities of distributed tabletop groupware.

RELATED WORK

Researchers have looked at several aspects of gestures, including representation of gestures through embodiments, techniques for enhancing remote embodiments, and capturing and representing gesture height.

Representing Gestures Through Embodiments

Gestures are an important part of collocated settings and equally valuable when people collaborate in distributed settings [2]. The design of user embodiments for showing remote gestures has long been considered in CSCW (e.g., [2,4,8]). Two approaches to embodiment design have emerged: abstract representations that visualize particular aspects of user information, and realistic representations based on video that show more details of the user.

Realistic Representations

Realistic embodiments represent collaborators with full or partial-body video, providing a rich representation of people's actions and postures. Early techniques used analog video (e.g., VideoDraw, VideoWhiteBoard, or ClearBoard, [11,25,26]); more recent designs typically use digital video, such as DOVE [18], VideoArms [23], or C-Slate [12]. Realistic embodiments provide a large amount of information about gestures, and convey considerable subtlety, but are often complex and expensive [28]. In addition, video embodiments have difficulty conveying some aspects of remote action, such as gesture height.

Abstract Representations

Abstract visualizations of particular elements of user activity can provide valuable information for collaboration. Telepointers are a well-known example that represents the pointing locations of collaborators with shapes and colors [8]; in addition, researchers have shown that abstract embodiments can be augmented with several different types of user information [22]. The abstract approach to embodiment design can convey information that is not easily available through a realistic embodiment (e.g., height can be represented directly with a specific visualization).

Combinations of Realistic and Abstract Representations

A few researchers have examined combinations of real and abstract embodiments (an approach we follow with KinectArms). Kirk *et al.* used video combined with a stylus to permit sketching [14], and Tang *et al*. added contact traces to VideoArms, an augmentation that participants described as useful [24]. A more quantitative approach to evaluating abstract components added to realistic embodiments was adopted by Yamashita *et al.* [28] who replayed entire gestures using a form of motion blur to reduce the time needed for conversational grounding.

Enhancing Gesture Visibility in Distributed Settings

In distributed collaboration, remote embodiments are much less visually obvious than people's bodies in co-located settings, and may be affected by problems such as low frame rates or network jitter; as a result, it is easy to miss actions and motions [7]. Motion traces can help to solve these problems by smoothing motion and providing a longer-lasting representation of a gesture (e.g., [8,24,29]). Most trace techniques, however, have only considered mouse movements rather than hand gestures (with Tang et al's work the notable exception [24]), and only show traces when in contact with the surface. Combining historical traces with height visualizations has been discussed [5], but only for telepointer-style embodiments.

Capturing and Representing Gesture Height

The height of deictic gestures (which indicate things in a shared environment) is valuable in communication [6]. Height, however, is not well supported in embodiment design: although some embodiments and interaction techniques incorporate differences between table touches and hover states (e.g., [24,27]), most do not, and none express a full range of height above the surface.

Part of this problem is that height is difficult to capture. There has been considerable research on using height as an input variable and capturing that height with cameras (often depth cameras). Most often, height has been used to identify touch interactions on variably distant surfaces, such as with OmniTouch [9], which used a depth camera similar to the Microsoft Kinect. Detecting touch on fixed-distance (non-touch-sensitive) surfaces has been shown to be possible using the Kinect camera (e.g., [28]). Marquardt *et*

al. are one of the few to suggest that interactions should span the spaces between touch, hover, and above the surface [17]. More elaborate models of a workspace have been created with moving cameras, such as in KinectFusion [13]. For most surface-based computing, however, a fixed camera and fixed surface is a more common scenario.

Although height detection in the layer above (or in front of) a surface has been used in a variety of gesture recognition and input techniques (e.g., [1] and [12]), there are few embodiments that represent height in this space. Two exceptions are Fraser *et al.* [4], who showed that representing approaches to a surface can improve distributed interactions, and Genest and Gutwin [5], who showed that abstract representations of height can improve gesture interpretation in distributed settings. Neither solution, however, captured detailed information about the hand and arm: Fraser captured the position of the tip of a stylus and Genest used a single 3D location, tracked with a magnetic-field sensor on the user's pointing finger.

REQUIREMENTS FOR AN ARM-EMBODIMENT TOOLKIT

A new toolkit for supporting the visualization of gestures in distributed systems would be valuable for researchers and developers. With the advent of cheap depth sensors, it has become important to understand how all three dimensions of gesture can be represented in a variety of distributed settings. This cannot be easily done with the state of the art in software tools for capturing and representing gestures. A toolkit for arm embodiments has four main requirements:

1. Address the three problems described above – of easily separating arm images, capturing and displaying height information, and enhancing embodiment visibility;
2. Permit the rapid reproduction of existing embodiment techniques to allow experimentation and comparison;
3. Support easy creation of new embodiment types with both realistic and abstract representation techniques;
4. Provide simple setup and calibration.

THE KINECTARMS TOOLKIT

KinectArms is a toolkit that simplifies the capture of remote tabletop gestures and the display of those gestures through arm embodiments. KinectArms has two parts: a capture module that recognizes hands and arms above the surface, performs video separation, and identifies the height of each pixel of the separated image; and a display module that provides built-in effects to show height, to improve visibility, and to provide movement traces. Additional effects can be easily developed and added to the toolkit.

Overview and Setup

KinectTable uses an Xbox or PC Kinect sensor, fixed above the table (1.8m is optimal for the Xbox version) and pointed down. The camera need not be perfectly perpendicular to the display surface; the automatic table detection (see below) can handle small angular variation. The Kinect has a field of view of 57° horizontally and 43° vertically, so at 1.8 meters, the camera can accommodate a 1.42m by 1.95m table surface. This area can be increased for larger tables (at the cost of resolution) by raising the Kinect.

Although 2D resolution will increase as the Kinect approaches the surface, the depth image has an optimal resolution at 1.8m, with an accuracy of approximately 2mm at that distance [28]. This level of accuracy means that user touches on the table surface can be approximately determined (i.e., without requiring a touch-sensitive surface); however, our tests described below used a PQLabs touch overlay on a 60-inch LCD television.

After the Kinect is suspended above the table surface, the KinectArms software can be started (usually by a client application). On startup, KinectArms automatically detects the table surface (see details below). Table detection takes less than a second, and no further calibration is required – KinectArms is then ready to capture and process arm images, and add visual effects to these images.

The general algorithm for using KinectArms in an arbitrary tabletop application is as follows:

1. Create an instance of the KinectArms client
2. Set up initial visual effects
3. While the application is running:
   3.1 Get image data from the KinectArms client
   3.2 Apply visual effects to the current frame
   3.3 Draw the current frame to the table

KINECTTABLE: THE CAPTURE MODULE

The capture module of KinectArms uses a Microsoft Kinect as its primary sensor. The Kinect incorporates a full-colour camera and a depth camera (both 640x480), which can be aligned to produce a depth-mapped image. From this image, we accomplish both separation of arms and hands from the table background, and recognition of the structure of the arms in the scene. We note that this recognition is not already provided by the Kinect – although the SDK provides sophisticated body recognition, it is not designed to capture isolated arms and hands over a table surface (it needs to see the entire body for accurate recognition).

The main features provided by the KinectTable module are:

1. Fast setup and calibration, with automatic detection of the table surface.
2. An API for accessing information about arms and table:
   a. Image masks for the table and each of the arms
   b. Fingertip, palm, and arm locations in three dimensions
   c. Basic user tracking
   d. Geometric properties for each of the arms.

The KinectTable API

The C++ version of the KinectTable API provides access to image data, table information, and arm information.

Image Data

The raw color and depth images from the Kinect are available in a custom structure; the application programmer can use the raw information, or can obtain a processed image representation from the KinectViz module.

```
KinectArmsClient *client = KinectArmsGetClient();
```

```
KinectData data;
client->GetData(data);
DepthImage& depthImage = data.depthImage;
```

Table Information

KinectTable provides information about the table surface (from the automatic recognition step, as described below) including the height of the table, the table corners, and a bitmap mask where white pixels indicate the table.

```
BinaryImage& maskImage = data.tableMaskImage;
```

Arm and Finger Information

Information for each arm is stored in a C++ struct; these are provided in an array representing all arms above the table (stored in the order that they enter the space). Each struct contains the following arm information:

1. Geometric values:
   a. Mean height of the arm (using all pixels);
   b. Total number of pixels corresponding to the arm;
   c. The pixel at the geometric center of the arm;
2. An array of points defining the arm boundary;
3. A bitmap mask corresponding to the arm;
4. An array of points corresponding to fingertips;
5. An array of points for between-finger locations;
6. A point representing the center of the hand;
7. A point representing the base of the arm (i.e., the intersection of the arm with the edge of the table);
8. A unique ID, retained between frames.

The height of any point in the arm image can be obtained from the `getHeight(Point p)` function. For example, the following code retrieves the depth of an arm's first finger:

```
client->GetData(data);
Arm& arm1 = data.arms[0];
int fingerHeight = client->getHeight(arm1.fingers[0]);
```

KinectTable System Architecture

KinectTable is comprised of four components: a camera component, a table detector, an arm and hand detector, and an arm tracker (Figure 2). KinectTable makes use of the Kinect SDK or the OpenNI library (openni.org) to obtain the Kinect images, and the OpenCV library for image processing (opencv.willowgarage.com).

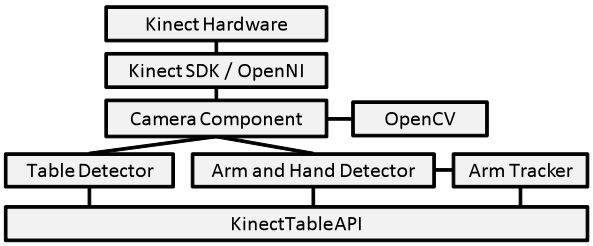

Figure 2: KinectTable system architecture.

Camera Component

The camera component fetches image data through the Kinect SDK or OpenNI sets the SDK to align the colour and depth images, and performs low-level processing to prepare the data for table, arm, and hand detection. This processing involves converting the Kinect images to our own custom C++ structure, and passing a 5x5 median filter over the image to fill in error pixels (some pixels fail to update every frame).

Table Detection

The height and extents of the table are critical for arm identification, and most systems use an interactive calibration procedure to determine these values [5,9,23]. KinectArms simplifies this step with an automatic table detector. Through the `RecalculateTableCorners()` function, KinectTable can be asked to identify the largest visible and roughly planar surface in the image as the table. After identification, any pixels in the depth map that are farther than the table, or outside its boundaries, are ignored.

Table detection is performed with the following steps:

1. A Laplacian filter and thresholding function are passed over the depth image to identify sharp changes in depth;
2. Depth edges are dilated to ensure continuity;
3. The center region enclosed by the continuous edge is filled – this region represents the table;
4. Table corners are found using k-curvature (as in [20]) on the region boundary (k-value of 30, threshold of 70°);
5. Table height is set as the lowest value of the table pixels.

Table identification is robust for slanted and curved tables, but any large non-linear changes in depth may cause incorrect table identification.

Arm and Finger Detection

Arms and fingers are identified using image processing techniques similar to those used for table detection (Fig. 3):

1. Background subtraction is performed by removing pixels in the depth image that are not above the table region. The remaining pixels are arm candidates.
2. A Canny edge detector (max threshold 200, min 150, Sobel filter order 5) finds the edges of the arms. Edges are dilated to make them continuous.
3. Contours of each arm candidate are found using OpenCV's Suzuki85 algorithm.
4. All small contours (< 40 pixels in length) are discarded as noise. The remaining contours are considered arms.
5. Geometric properties of each arm (mean depth, total pixels, center point) are calculated from the contours.
6. The base of each arm is determined by finding the intersection of the arm with the edge of the table.
7. Fingertip locations and the points between the fingers are found using k-curvature (k-value 30, threshold 85°).
8. The center of the hand is calculated using a Euclidean distance transform of the arm's boundary points.

Arm Tracking and User Identification

Arms are given unique ID numbers so that the application can track arms across multiple frames. Tracking takes advantage of the location where the arm intersects the edge of the table (a location that does not change rapidly in tabletop work). KinectTable matches arm bases in the current frame to known arm bases in the previous frame, by comparing distance and time values between the frames. If

an arm in a previous frame cannot be matched, we assume it has left the table area. If an arm does not return at the same base position within five seconds, we re-use the ID for the next arm that enters the table area.

.NET Wrapper

Developers can access KinectTable through a .NET wrapper that provides .NET versions of all data structures. The wrapper's API is similar to the C++ API but also provides event mechanisms whereby methods are called automatically when new data is available.

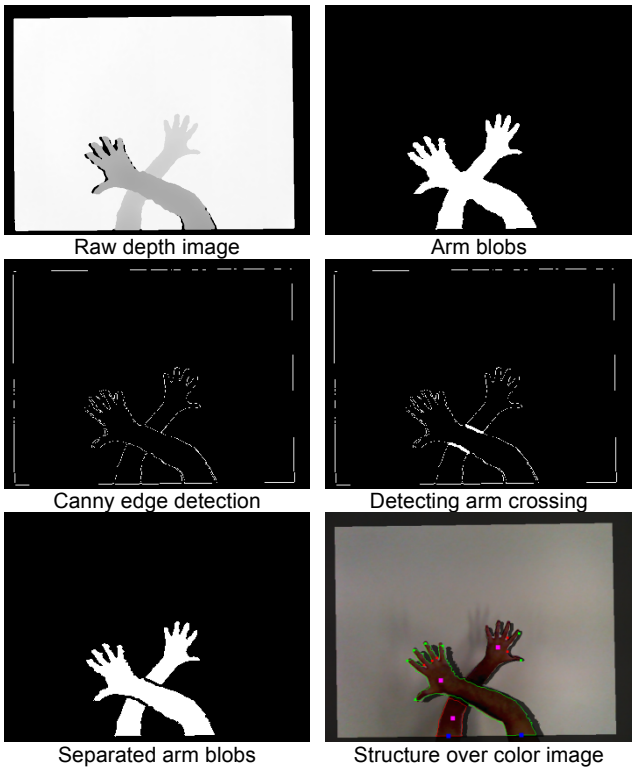

Figure 3: Steps in processing arm images.

KINECTVIZ: THE DISPLAY MODULE

KinectViz provides a set of standard effects that can be added to arm images obtained from KinectTable. The library provides effects to visualize height, increase or decrease arm visibility, improve user identification, and provide motion traces. Using these effects, it is possible to replicate many previous embodiments; KinectViz also allows programmers to design new effects. Programmers can set up effects with single API calls, and effects can be changed dynamically and can be assigned to specific height layers. Effects can be applied to all arms or to specific arms (the examples below show the global versions).

Basic Arm Representations

VideoArms. KinectViz provides full-colour background-subtracted images of hands and arms that can be drawn on top of existing tabletop objects. The basic representation (similar to VideoArms [23]) is built in a few lines of code:

```
client->GetData(data);
viz.updateData(data);
viz.applyEffects();
```

The `applyEffects()` method translates raw KinectTable data into a drawable image, and also alters the image to add any visual effects that have been chosen by the application programmer (no effects are needed for the standard arm).

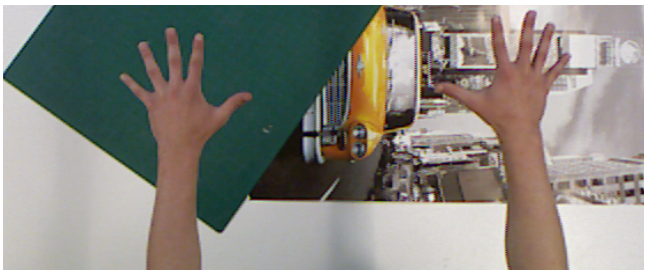

Figure 4: Standard arms drawn over table artifacts.

StructureArms. The extracted structure of the arm (base point, hand center, and finger points) can be used as the basis for an abstract representation of the arm. The main advantage over video-based embodiments is that structure information is much smaller than video, and can therefore be used even in poor network conditions. The structure points could also be used as anchors for artificial textures (e.g., cartoon arms or even images of people's real arms). The points on the structure can be accessed through KinectData (see above); as a demonstration, KinectViz includes a 'stick figure' effect in its API (Figure 5).

```
viz.getEffect(kSkeleton).enabled = true;
```

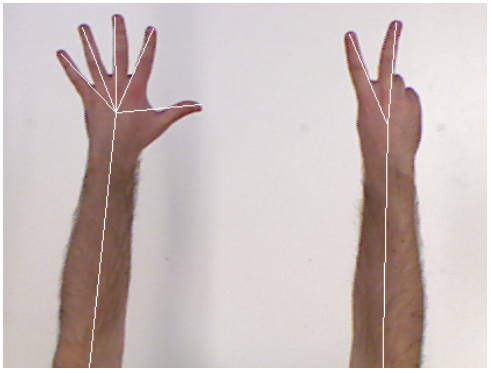

Figure 5. Stick-figure arms using structure points

Visual Effects 1: Height Indicators

Showing the height of a gesture can greatly improve people's ability to interpret the meaning of that gesture [5]. KinectViz provides three kinds of height indicator.

Circles. KinectViz can add abstract visualizations to the realistic embodiment. To show height, we add a circle that changes size and transparency based on the arm's height above the table (see Figure 6). KinectViz uses the lowest fingertip as the centre of the circle, which assumes that people are pointing downwards towards the surface.

Shadows. Arm shadows have been part of analog video representations (e.g., [24]), but are not usually captured in digital embodiments. KinectViz provides a shadow effect in

which the shadow is displaced to the side as the arm moves higher above the table (Figure 7).

```
viz.getEffect(kPointerCircle).enabled = true;
```

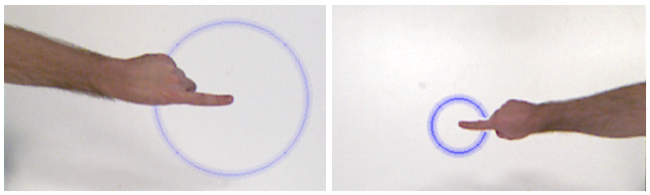

Figure 6: Circles – abstract visualization of gesture height.

```
viz.getEffect(kShadow).enabled = true;
```

Figure 7: Shadows – simulated representation of height.

Gradients. Richer representations of arm height are also possible with KinectVis. One novel technique shows height as a false-colour gradient (see Figure 8): parts of the arm closer to the table are shown in cool colors (blues and greens); higher parts of the arm are shown in warm colors.

```
viz.getEffect(kRainbow).enabled = true;
```

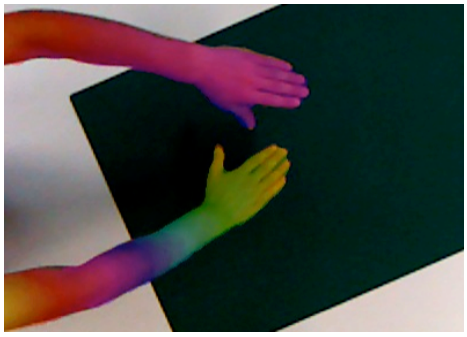

Figure 8: Gradients – richer representation of arm height.

Visual Effects 2: Visibility and Identification Enhancements

Virtual embodiments can suffer from several kinds of visibility problems: they are harder to notice and harder to see than real arms [7,19], they are often difficult to identify, and they can occlude objects on the table. To address these problems, KinectViz provides effects that manipulate the visibility of an embodiment and that assist identification.

Outlines. To increase noticeability, KinectViz includes an effect that draws a colored outline around the arm (Figure 9); the effect can be applied to one or all of the arms.

```
viz.getEffect(kOutline).enabled = true;
```

Transparency. Embodiment visibility can be varied with a transparency level defined by the application programmer, or with a level that varies according to the height from the surface (Figure 10). Transparency can be used in several ways: as a basic way to avoid occlusion, or as a dynamic effect (e.g., to reduce salience of less-active participants).

Figure 9: Outlines – visibility on complex backgrounds

```
viz.getEffect(kTransparency).enabled = true;
```

Figure 10: Transparency – reducing visibility and occlusion.

Tattoos. KinectArms can apply virtual tattoos or markings to enhance identification (Figure 11). The tattoos move with the user's arm and are maintained even if the user removes her arm from the Kinect's field of view (see description of IDs above). Tattoos can be any image or text, and are defined by providing a directory for tattoo images. Tattoos are applied in the order that arms enter the workspace, and are set up similarly to other effects:

```
viz.getEffect(kTattoo).enabled = true;
```

Figure 11: Tattoos – improving identification of embodiments.

Tinting. A second effect for helping people differentiate between several embodiments applies a semi-transparent color to the arm image. The effect is similar to the gradient effect of Figure 8, but with a single color.

```
viz.getEffect(kTint).enabled = true;
```

Visual Effects 3: Motion Traces

Gestures are easily missed in distributed settings because of the reduced salience of virtual representations, or network problems [8]. KinectViz provides two effects that help solve this problem: motion lines and motion blur.

Motion lines. Fingers can leave motion-line traces as they move above the table. This effect is created by storing the

finger point, connecting the points with lines, and fading older lines to avoid cluttering the space (Figure 12). No previous embodiment technique enables these kinds of traces above the table surface.

Motion blur. A blur effect on the entire arm can help people notice and understand arm movement and gesture [29]. KinectViz produces this effect by overlaying previous frames on the current image (Figure 12), and increasing the transparency on older frames.

```
viz.getEffect(kTraces).enabled = true;
viz.getEffect(kMotionBlur).enabled = true;
```

In both effects, variable `historySize` indicates how many previous samples to use in the effect, and `maxAge` controls how quickly traces fade away.

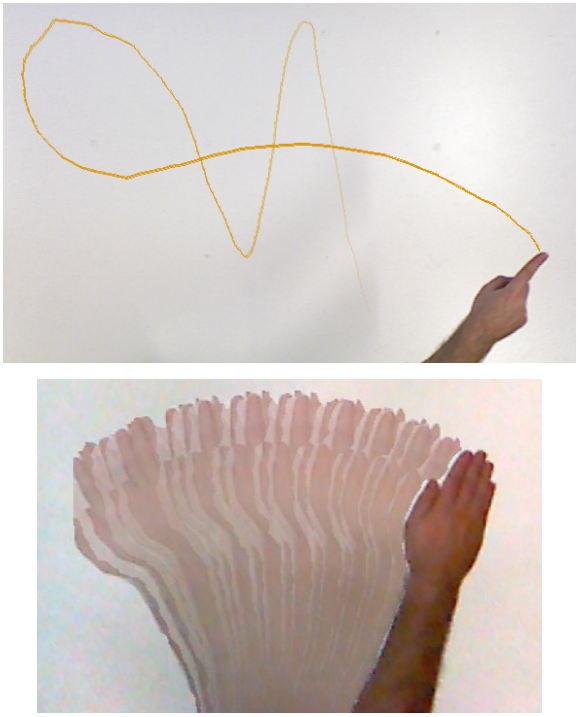

Figure 12: Motion lines (above) and motion blur (below).

Assigning Effects to Specific Height Layers and Users

People use different heights above the table surface to provide different information in a gesture [9] – for example, touching the surface, hovering, and medium and high levels above the table. Therefore, embodiments should be able to represent users differently depending on the height layer in which they are gesturing. All of the effects in KinectViz can be restricted to be visible only in certain layers above the table. For each of the API calls above, there are additional versions with two extra parameters – the lower and upper height of the layer in which the effect should be shown. This capability allows composite visualizations to be created without needing to add new code. For example:

```
viz.getEffect(kMotionBlur).minHeight = 0;
viz.getEffect(kMotionBlur).maxHeight = 1000;
```

Similar method calls allow programmers to assign effects to single users, using the user ID as determined by the library.

EVALUATION OF KINECTARMS

Our experiences with KinectArms allow an analytical evaluation of several issues: the toolkit's performance, the complexity it adds to applications, and its extensibility, generality, and expressive power.

Performance

KinectArms is fast, providing visualized arm images at about 30 frames per second, even with multiple users. KinectTable uses the depth camera and a set of fast image-processing algorithms to carry out video separation, so a major computational expense seen in other systems is avoided. The Kinect hardware produces 30 frames per second, and KinectTable easily processes at the same rate.

KinectVis carries out additional image processing and other computation to add visual effects to the arm images. Multiple or demanding effects in KinectViz, or a large number of arms that need to be processed, could reduce the frame rate depending on hardware. In our experience, with four users and simultaneous use of several of the effects described above, a standard PC (Windows 7, Core i5 processor) was able to maintain a rate of 30 frames per second. In distributed tables, networking constraints are more likely to impose the upper limit on frame rate (at remote sites) than KinectArms.

Complexity and Usability for Application Programmers

Using KinectArms with groupware applications is simple. API calls to KinectTable return arrays of hands and arms with unique identifiers that match users from frame to frame. Applications can choose how and where to display the hands and arms depending on their specific requirements. Similarly, KinectViz effects can be added with simple changes to the API, and the API allows developers to parameterize the effects (to different users and specific height layers) and change effects dynamically at runtime.

We carried out two informal tests of how easily KinectArms can be used for client application projects. First, we integrated KinectArms with a distributed photo-sharing application running on two interactive tables. With this application, users could manipulate shared images using the touch capabilities of the networked tables and see each other's gestures with KinectViz effects (e.g., Figure 1 and Figure 13). The integration of KinectArms and the photo-sharing application took less than one afternoon, suggesting that developers will be able to easily access KinectTable data and use KinectViz effects.

Second, we provided the KinectTable toolkit to another researcher in our lab (not an author) who needed a toolkit for exploring tabletop gesture recognition. Using calls to the KinectTable API, the researcher was able to begin developing gesture recognition systems within a day of starting development. The KinectTable hand structure

provided enough information for the researchers to write software that recognizes hand orientations and postures (e.g., fingers up or fingers down), grasping gestures, and pointing gestures. This research is currently ongoing, and is continuing to use the KinectArms toolkit.

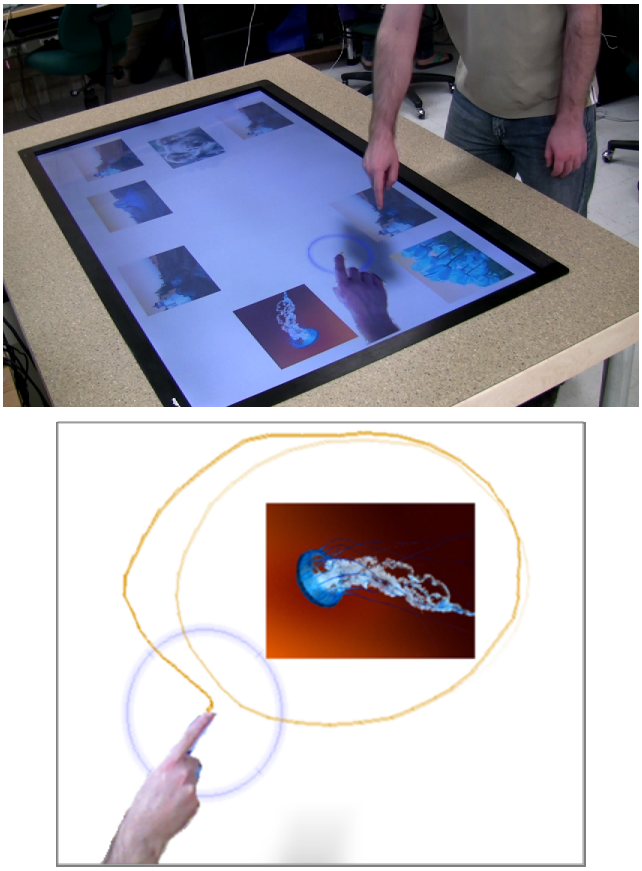

Figure 13: A remote user gesturing to indicate a shared image (below) in the photo-sharing application (above).

Expressiveness: Replicating Existing Techniques

A measure of KinectArms' breadth is its ability to duplicate other arm embodiments that have appeared in previous research. Using only the stock effects of KinectArms, we are able to reproduce embodiments from a wide range of previous work, including VideoArms [23], DOVE [18], C-Slate [12], Fraser's approach visualization [4], and Yamashita's gesture replay [29].

For example, Tang's VideoArms shows hands and arms as two-dimensional silhouettes over a workspace. A later version that used a Microsoft Surface added 'trace pearls': small, fading lines where fingers came in contact with the surface [24]. KinectViz can replicate these two techniques using its basic video arm representation, and a motion-line effect that is limited to a small layer just above the surface (0-1cm). When a pointing finger (defined as the lowest finger on a hand) enters the low layer, traces are drawn. This embodiment can be built with the following calls:

```
viz.getEffect(kTraces).enabled = true;
viz.getEffect(kTint).enabled = true;
((Tint&)(viz.getEffect(kTint))).handColors[0] =
  ColorPixel(255,100,100);
```

The monochrome image of users' hands and arms in VideoArms can also be changed to full colour at no cost (part of the basic video arm representation in KinectArms).

As a second example, both DOVE and C-Slate overlaid video embodiments on a workspace and used transparency either to reduce occlusion or to indicate that hands were further from the tablet surface. In addition, DOVE provide contact traces when the user touched the surface. These techniques can be replicated using transparency, shadows, and motion lines:

```
viz.getEffect(kTransparency).enabled = true;
viz.getEffect(kShadow).enabled = true;
viz.getEffect(kTraces).enabled = true;
```

In addition, KinectArms shows its expressive power in two other ways. First, by combining effects for different users and different height layers, KinectViz can create complex representations that go well beyond existing examples. Second, KinectViz provides several novel effects not seen elsewhere (e.g., outlines, color gradients, stick-figure arms, tints, and above-the-table motion lines); and these new effects were simple to build by using KinectArms' processed images, extracted arm structure, and detailed height information, capabilities which are not provided by any other current toolkit.

Extensibility: Defining New Effects

Although KinectViz has a wide variety of stock visualizations, developers may wish to create their own. The extensibility of the KinectArms toolkit is shown in the simplicity of the process of creating a new effect. KinectTable provides easy access to hand and arm position data and KinectViz automatically applies any enabled effect when `applyEffects` is called. Adding a new effect involves three steps:

1. Create a new class that extends class `Effect`, and implement method `applyEffect()` to manipulates the input image in the desired way.
2. Register the new effect with KinectViz, using the call `int handle = registerEffect<MyEffectClass>();`
3. The new effect can then be toggled on and off, and added to different height layers, using the handle.

Generality

KinectArms works with a wide variety of tabletop setups and table hardware. Tables with existing touch sensing can be used alongside KinectArms (and can be used to augment KinectViz effects by providing additional accuracy for touch events). Top-down projection, a problem for most video-separation techniques, works well with KinectArms, since separation is done with the depth camera, not the colour camera. KinectArms can also be used with non-interactive tables (similar to Ishii's TeamWorkStation). One-way connections can be used to show remote users the contents of a normal table, thus supporting collaboration over paper-based artifacts.

EXTENSIONS TO KINECTARMS

Although KinectArms already provides substantial support to developers of distributed tabletop groupware, there are several extensions that can further improve remote gestural communication, awareness, and system efficiency.

Texture + structure arm embodiments. Structure-based representations of hands and arms are far cheaper to transmit over networks than high quality images. KinectArms automatically extracts structure information about arms above the table, and this could allow designers to provide some of the benefits of realistic video-based embodiments without the costs of transmitting video. That is, network performance can be improved by transmitting hand and arm position data without images and adding custom images at the remote location. Images of users' arms could be taken at the beginning of a session or other kinds of representations could be used. Future research should determine the effectiveness of such cost rebalancing (GPU vs. network) and how users respond to reconstructed or artificial representations rather than live images.

Calibrating multiple distributed tables. Currently, KinectTable provides no built-in support for calibrating between two remote tables, and systems must be aligned by hand (by setting both Kinect sensors at fixed heights above the equal-sized surfaces). Based on this experience, we are currently adding an interactive calibration tool to the library, to permit the Kinect to be hung at different heights above different tables.

Storing the history of interactions. Some researchers have discussed the ability to view long term interaction traces – the historical movements of collaborators over an entire session [16]. Since KinectTable identifies hands and arms with unique IDs and stores their boundaries as well as finger and palm locations in 3D, it would be possible to archive all of the movements during a collaborative session in this way and reconstruct it at a later date. Such replays could be used to answer questions about what kinds of activity occurred during a collaborative session, or could be synchronized with audio recordings to visually identify when different spaces in the workspace were used.

Using the space around the table. KinectTable recognizes tables in its field of view and ignores everything outside of that space. However, collaboration operates in a context that includes the workspace and much research has shown that seeing what is happening outside of the workspace is valuable for both distributed and collocated activities (e.g., [4, 24]). Future versions of KinectTable could model the entire body in the same way as Microsoft's Kinect SDK, except from above. Understanding where users are situated at a table could, for example, allow identification of right and left hands, allow better identification of hands, and allow the tracking of users as they move around the table (something that currently disrupts user tracking). It can also improve awareness in distributed settings and allow systems to make assumptions about user behaviour depending on whether they are sitting or standing.

Supporting gesture recognition. KinectTable identifies several key components of hands and arms (e.g., fingertips and palms) and the current version of the toolkit is sufficient for basic, if rudimentary, gesture recognition. Future work will focus on higher levels of accuracy and added identification features, such as palm orientation, knuckle location, and wrist location. With these improvements, KinectTable can be a tool for gesture recognition as well as representation. We will use this capability to explore possible collisions of communicative gestures and command gestures in interactive systems.

Creating additional hybrid embodiments. Embodiments have previously been classified as realistic or abstract [6]. A third space, a hybrid of abstract and realistic, uses high quality images of hands and arms with abstract visual effects that accommodate for missing fidelity in the images, emphasize existing information, or add information not normally available in collocated settings. To date, few such hybrid embodiments have been built; KinectArms is the first tool to make the development of hybrid embodiments simple and fast. KinectViz allows designers to create and evaluate a wide range of hybrid embodiment designs. It will also help address some challenges with hybrid embodiment design, such as the placement of abstract elements to enhance pointing (our assumption that the pointing finger is always the lowest point of the hand and arms is correct for only some cases).

Interface gestures vs. communicative gestures. KinectArms can also help explore the tension between interface gestures meant as input, and communicative gestures between collaborators. Detailed gesture data about hands, fingers, and arms, along with information about where collaborators are positioned around a table will assist designers and researchers in determining how best to create systems that support both gestural input and gestural communication.

CONCLUSION

Representing rich and subtle gestures in distributed tabletop groupware – particularly those that use varying levels of height above the table – is often expensive, complex, and can require specialized hardware and lengthy calibration. As a solution to the problems of capturing and displaying arm embodiments for remote work, we developed KinectArms, a toolkit that helps groupware developers build arm embodiments with a minimum of effort. KinectArms handles video capture and separation, determines fine-grained height above the table, and provides visualizations that improve arm visibility, representation of gesture height, and movement. The KinectArms toolkit allows simple replication of most existing arm embodiment techniques, and enables the creation of many new types of representation.

In future work, we will carry out two kinds of further development with KinectArms. First, we will continue refinement of the current library – for example, to improve edge detection, effect speed, and camera/table alignment. Second, we will investigate the new extensions described above; these are longer-term enhancements and new functionality that will provide additional ways for people to interact around distributed tabletop systems, all made possible using cheap and easy-to-use components.

SOFTWARE AVAILABILITY

The KinectArms toolkit and the example programs used in this paper are available at www.hci.usask.ca/KinectArms.

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada, and the SurfNet Research Network.

REFERENCES

1. Ballendat, T., Marquardt, N., Greenberg, S. Proxemic interaction: designing for a proximity and orientation-aware environment, *Proc. ITS 2010*, 121-130.
2. Bekker, M., Olson, J., Olson, G. Analysis of gestures in face-to-face design teams provides guidance for how to use groupware in design, *Proc. DIS 1995*, 157-166.
3. Benford, S., Bowers, J., Fahlén, L., Greenhalgh, C., and Snowdon, D. User embodiment in collaborative virtual environments, *Proc. CHI 1995*, 242–249.
4. Fraser, M., McCarthy, M., Shaukat, M., and Smith, P. Seconds Matter: Improving Distributed Coordination by Tracking and Visualizing Display Trajectories, *Proc. CHI 2007*, 1303-1312.
5. Genest, A. and Gutwin, C. Evaluating the effectiveness of height visualizations for improving gestural communication at distributed tabletops, *Proc. CSCW 2012*, 519–528.
6. Genest, A. and Gutwin, C., Characterizing Deixis over Surfaces to Improve Remote Embodiments, *Proc. ECSCW 2011*, 519-528.
7. Gutwin, C. and Greenberg, S. A descriptive framework of workspace awareness for real-time groupware, *CSCW*, 11, 3, 2002, 411–446.
8. Gutwin, C. and Penner, R. Improving interpretation of remote gestures with telepointer traces. *Proc. CSCW 2002*, 49-57.
9. Harrison, C., Benko, H., and Wilson, A. OmniTouch: Wearable Multitouch Interaction Everywhere. *Proc. UIST 2011*, 441-450.
10. Hindmarsh, J. and Heath, C. Embodied reference: A study of deixis in workplace interaction. *Journal of Pragmatics*, 32, 12, 2000, 1855-1878.
11. Ishii, H. and Kobayashi, M. ClearBoard: a seamless medium for shared drawing and conversation with eye contact. *Proc. CHI 1992*, 525–532.
12. Izadi, S., Agarwal, A., Criminisi, A., Winn, J., Blake, A., Fitzgibbon, A. C-Slate: a multi-touch and object recognition system for remote collaboration using horizontal surfaces. *Proc. Tabletop 2007*, 3–10.
13. Izadi, S., Newcombe, R., Kim, D., Hilliges, O., Molyneaux, D., Hodges, S., Kohli, P., Shotton, J., Davison, A., and Fitzgibbon, A. KinectFusion: Real-Time Dynamic 3D Surface Reconstruction and Interaction. *Proc. SIGGRAPH 2011*, 559-568.
14. Kirk, D. and Fraser, D. Comparing remote gesture technologies for supporting collaborative physical tasks. *Proc. CHI 2006*, 1191–1200.
15. Li, J., Wessels, A., Alem, L., and Stitzlein, C. Exploring interface with representation of gesture for remote collaboration. *Proc. OZCHI 2007*, 179-182.
16. MacEachren, A., Brewer, I., Cai, G., and Chen, J. Visually enabled geocollaboration to support data exploration and decision-making. *Proc. International Cartographic Conference 2003*.
17. Marquardt, N., Jota, R., Greenberg, S., Jorge, J. The Continuous Interaction Space: Interaction Techniques Unifying Touch and Gesture on and above a Digital Surface. *Proc. Interact 2011*, 461-476.
18. Ou, J., Chen, X., Fussell, S., and Yang, J. DOVE: Drawing over video environment. *Proc. Multimedia 2003*, 100–101.
19. Pinelle, D., Guwin, C., Nacenta, M. The Effects of Co-Present Embodiments on Awareness and Collaboration in Tabletop Groupware. *Proc. GI 2008*, 1-8.
20. Segen, J. and Kumar, S. Human-computer interaction using gesture recognition and 3D hand tracking. *Proc. ICIP 1998*, 188-192.
21. Shoemaker, G., Tang, A., Booth, K. Shadow reaching: a new perspective on interaction for large displays. *Proc. UIST 2007*, 53-56.
22. Stach, T., Gutwin, C., Pinelle, D., Irani, P. Improving recognition and characterization in groupware with rich embodiments. *Proc. CHI 2007*, 11-20.
23. Tang, A., Neustaedter, C., Greenberg, S. Videoarms: embodiments for mixed presence groupware. *People and Computers 20*, 2007, 85-102.
24. Tang, A., Pahud, M., Inkpen, K., Benko, H., Tang, J., and Buxton, B. Three's company: understanding communication channels in three-way distributed collaboration. *Proc. CSCW 2010*, 271–280.
25. Tang, J. and Minneman, S. VideoWhiteboard: video shadows to support remote collaboration. *Proc. CHI 1991*, 315–322.
26. Tang, J. Minneman, S. VideoDraw: a video interface for collaborative drawing. *ToIS*, 9, 2, 1991, 170–184.
27. Wigdor, D., Williams, S., Cronin, M., et al. Ripples: utilizing per-contact visualizations to improve user interaction with touch displays. *Proc. UIST 2009*, 3-12.
28. Wilson, A. Using a depth camera as a touch sensor. *Proc. ITS 2010*. 69-72.
29. Yamashita, N., Kaji, K., Kuzuoka, H., and Hirata, K. Improving visibility of remote gestures in distributed tabletop collaboration. *Proc. CSCW 2011*, 95-104.