Kinected Browser: Depth Camera Interaction for the Web

Daniel J. Liebling, Meredith Ringel Morris
Microsoft Research, Redmond, Washington, USA
{ danl, merrie }@microsoft.com

ABSTRACT
Interest in and development of gesture interfaces has recently exploded, fueled in part by the release of Microsoft Corporation's Kinect, a low-cost, consumer-packaged depth camera with integrated skeleton tracking. Depth-camera-based gestures can facilitate interaction with the Web on keyboard-and-mouse-free and/or multi-user technologies, such as large display walls or TV sets. We present a toolkit for bringing such gesture affordances into modern Web browsers using existing Web programming methods. Our framework is designed to enable Web programmers to incrementally add this capability with minimum effort by leveraging Web standard DOM structures and event models. We describe our framework's design and architecture, and illustrate its usability and versatility.

Author Keywords
Kinect, depth cameras, JavaScript, HTML, toolkits.

ACM Classification Keywords
H.5.m. Information interfaces and presentation: Misc.

INTRODUCTION AND RELATED WORK
Interest in input methods beyond the mouse and keyboard has recently accelerated as new, inexpensive sensing hardware has become widely available. Such input is particularly suitable to emerging form-factors, such as large display walls, which lack mice and keyboards.

In 2010, Microsoft launched the Kinect sensor for its Xbox 360 gaming platform [kinect.com], followed by Kinect for Windows in early 2012 [kinectforwindows.org]. The wide availability and affordability of these devices ($150 USD) make them a popular platform for experimentation among researchers, practitioners, students, and hobbyists. The Kinect device and accompanying SDK provide access to color and infrared (depth) cameras at 640 × 480 pixel resolution, 18-point skeleton tracking for multiple simultaneous skeletons, and a microphone array. The SDK provides programming APIs in both C++ and C#.

Researchers have quickly appropriated depth cameras such as Kinect for a variety of scenarios including 3D scene reconstruction [3], simulated touch sensing [7], facilitating interaction with multi-display environments [9], and enabling on-body sensing [2].

In this note, we describe a toolkit that enables developers to easily augment any Web page with gesture or speech input. Recent standards such as HTML5 and CSS3 enable richly interactive Web applications. Despite these recent updates, the browser interaction model is still based on keyboard, mouse, and simple multi-touch interactions (the latter due largely to the recent surge in adoption of touch-enabled smartphones and tablets).

We anticipate that in a few years, PCs, laptops, tablets, and perhaps even smartphones will include depth cameras as standard peripherals (much like webcams today). By supporting depth camera browser interactions in addition to standard mouse, keyboard, and touchscreen inputs, we envision that a user could choose the most natural and appropriate modality for a given task. Our toolkit also supports easily augmenting web pages with speech input via the Kinect's microphone array, but the main focus is on incorporating gesture input.

Using gestures to interact with the Web may be appropriate for a variety of reasons, such as for a collaborative web search where the users outnumber the traditional input devices [1]; for ergonomic reasons (to enable productivity during typing breaks) [5]; for situations where traditional input devices are not available such as for interactions with large display walls [6] or TVs; or for casual web-based experiences such as browser-based games or the use of the web for other casual or whimsical tasks (e.g., "lean-back internet" [4]).

There have been a few recent forays into using Kinects with Web browsers. In June 2012, Microsoft announced a version of Internet Explorer for Xbox, which is a customized browser that uses voice commands for navigation; in contrast, we present a framework applicable to standard web browsers, which includes gesture support. SwimBrowser [swimbrowser.tumblr.com] is a whimsical application that allows a user to navigate the Web via "swimming" gestures, recognized by a Kinect. It shows how a depth camera and browser might be used together, but does not expose control over the development of such interactions to page authors. DepthJS [depthjs.media.mit.edu] is an early Kinect–Web browser integration for the Safari and Chrome browsers. DepthJS provides some high level affordances such as gesture-based tab switching, list item selection, and button presses;

ITS'12, November 11–14, 2012, Cambridge, Massachusetts, USA.

```
<html>
<head>
    <script type="text/javascript" src="jquery-1.7.min.js"></script>
    <script type="text/javascript" src="KinectedBrowser.js"></script>                    ①
    <script type="text/javascript">
    $(document).ready(function () {
        Kinect.init();                                                                   ②
        $("#positions").on("rightHandOver.kinect", function (state) {                    ③
            $("#positions").text("Right hand @ " + state.clientX + ", " + state.clientY);
        });
    });
</script>
<body>
    <div id="positions" style="width:200px;height:200px;border:1px solid black">Wave here!</div>
</body>
</html>
```

Figure 1. Using HTML and JavaScript with jQuery to react to the right hand moving. Key portions are highlighted in gray. (1) including script library; (2) initializing object; (3) attaching event listener on the document.

developers use JavaScript to access this functionality. DepthJS focuses on providing reusable code for particular hand movement and gesture patterns; in contrast, our toolkit focuses on providing a more general integration of the depth camera and browser in order to provide more flexibility in interaction design. Additionally, our contribution in this paper goes beyond the features of the toolkit itself to encompass a reflection on the design rationale behind the architecture of Kinected Browser and an evaluation of its usability and versatility.

DESIGN CONSIDERATIONS

HTML and JavaScript are the common denominators of the Web; browsers exist on almost every modern platform and form factor. We believe that as depth cameras become cheaper, more common, and expand beyond PCs, researchers and developers will want to explore gestural interfaces that can scale across devices, from phones to wall displays. Our toolkit enables such scenarios.

Gesture integration with a browser can take two forms: high-level manipulation that translates specific gestures into existing events such as clicks and navigations (the approach taken by DepthJS), or mapping depth and skeleton positions into lower-level objects from which page authors can develop new experiences. The "high-level" approach limits the gesture vocabulary in ways that prevent page authors from developing novel or customized experiences and pushing the boundaries of interactive Web sites. In contrast, we aim to support *versatility* in terms of the range of experiences that can be created with Kinected Browser. Our "low-level" design approach avoids constraining Web page authors with assumptions. Kinected Browser does not assume that the entire page is interactive, that there is only one user, that the screen has a specific size or aspect ratio, that only a small set of element types support interaction, or that a specific gesture or speech vocabulary is standard. Also, to maximize flexibility, our toolkit allows access to raw data from the sensor. Although most page authors are unlikely to use raw depth data, one can imagine Web applications using depth data, e.g., to scan physical artifacts and input them into 3D printing services.

Existing Web experiences are built around single-user interaction paradigms. Multiple users sharing a PC are not uncommon, from classrooms in developing nations [7] to co-located search [5]. As large interface spaces lend themselves to social interaction and the Web becomes more pervasive, Web developers will increasingly need multi-user input support. Our design supports as many users as the underlying Kinect system, and every event fired by the system includes a user identifier. We explored adding uniqueness of user IDs in the toolkit, but leave this up to the developer, since "uniqueness" can be defined in different ways – a person, a team, a family, etc. Furthermore, with increased availability of face recognition systems, developers can easily leverage a third party service to perform their own identity assignment.

Despite the wealth of data that systems like Kinect provide, the toolkit should be easy to use by programmers. The system should leverage existing Web interaction programming facilities. Our toolkit maps the skeleton information provided by Kinect into scriptable browser events, as described in the next section.

ARCHITECTURE

Kinected Browser consists of two modules (Figure 2): a browser plugin linked to the C++ Kinect SDK, and a JavaScript library. The SDK provides low level data including the positions of users' skeletons and the color and depth images. SDK consumer applications choose to receive data when new frames are ready, or poll for new data on their own schedule. Our toolkit reads this data and interprets them as higher level events, freeing page authors from worrying about frame rates and hit testing.

The natural point of integration in a Web browser is its existing UI event system. Most visible Document Object Model (DOM) elements fire events in response to mouse and keyboard activity. Adding additional gesture events on

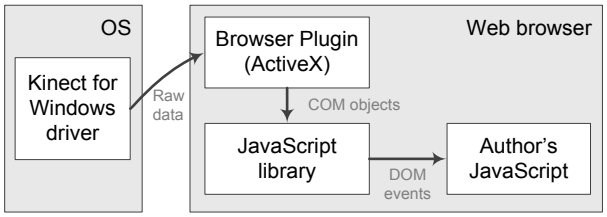

Figure 2. Kinected Browser toolkit architecture

each object is therefore consistent with the current programming model. Furthermore, it allows developers to reuse existing code if they simply want to use joints as virtual mouse cursors.

When Kinect data is ready, the browser plugin reads the raw data and assembles data structures that can also be read by scripting in the browser, since raw C-style interfaces are not usable there. We chose to implement most of the functionality in JavaScript to provide independence from the hardware and raw data; the browser-based portion of the code should work for new depth camera packages and new methods of providing data to the browser. For example, rather than a browser plugin, the low-level data could be read using the new WebSocket protocol, or the Kinect device might be on a remote machine. Furthermore, the JavaScript library is written to be browser platform neutral.

Pages enable use of the Kinected Browser system by placing a `<SCRIPT>` tag in the HTML that references the JavaScript library (Figure 1.1). After just as few as one lines of initialization code (Figure 1.2), authors can use data and events from the Kinect device using standard Web programming concepts.

Mapping Camera Space to Browser Space

In any vision-based application, the developer must choose how to map the camera coordinates into screen space. Kinected Browser provides two skeleton-to-screen-space mapping functions, but allows developers to specify custom mappers. By default, Kinected Browser maps screen space to the viewable area of the Web page rather than the entire scrollable page area. Since the camera space is in landscape aspect ratio, mapping to the entire page (the length of which may be several multiples of the width) would lead to severe vertical distortion. In practice, we find that users' browser viewports tend to have aspect ratios that are similar to the camera's; for such configurations, even when the horizontal and vertical multipliers are not precisely the same, the user experience is not adversely affected.

The system also supports the notion of egocentric mapping. That is, the centroid of the skeleton is roughly mapped to the center of the screen space. This allows multiple users to reach all controls on the page without having to occupy the same physical space. In egocentric mapping, we also scale the skeleton such that the user can physically reach any element in the viewport. To determine the outermost reachable points, we subtract arm length from the re-centered uppermost torso point in both the X and Y dimensions. These points, plus those of the feet, determine the physical skeleton bounding box. The system then scales between the physical bounding box and browser viewport. To aid debugging of skeleton interaction, the system supports drawing active skeleton mappings to an HTML canvas. We find this is useful for determining whether errors are due to camera noise or program bugs.

Controlling the DOM with Skeleton Input

Given the aforementioned camera-to-browser space mapping, the system uses skeleton points and motion to fire events on DOM elements. Page authors subscribe to skeleton movement events in a similar way to mouse events, adding event listener functions to any visual DOM element (Figure 1.3). Parallel to the DOM standard `mouseOver` and `mouseOut` events, the system publishes *`joint`*`Over` and *`joint`*`Out` events where *joint* is each of the 18 joints. The system hit tests skeleton points against a DOM element's bounding box, firing the appropriate event if necessary. Similarly, events in the *`joint`*`Move` family fire when a joint moves within an element's bounding box. The system can also map multiple joints into one "virtual joint" for scenarios where a set of joints is semantically equivalent. For example, we map *leftHand* and *rightHand* to a virtual *hand* joint. By using the virtual joint, authors can allow DOM elements to respond to either hand without duplicating event handlers; alternatively, developers can use the individual *left* and *right* versions of joints if they design interactions that make use of such distinctions.

In addition to having DOM elements respond to skeletal events, developers may also wish to associate input with geometrically specified regions of a page. For instance, a developer creating a game using only an HTML5 canvas might wish to make sprites react to joints moving over them. To support this scenario, Kinected Browser reuses the existing image map infrastructure. Web page authors use image maps to specify polygonal and circular regions of an image that respond to mouse interaction. Our system interprets the rectangular, circular, and polygonal areas to hit test for joint events.

Higher Level Inputs

In addition to over/out and movement events, Kinected Browser also supports joint hovers. Inspired by the Kinect for Xbox 360's hover selection mechanism, if a joint dwells above a DOM element and the joint's coordinates are fairly stable for 100 ms (allowing for noise and human joint wobble), the browser displays a large circle consisting of translucent arc segments that fill with a contrasting color as time progresses. If the hand remains over the target for 2.5 seconds, the system fires a target acquisition event and plays an audio cue. Target release occurs in the reverse manner; if a dwell occurs while the system is in an acquired state, then the target is released and a different audio cue plays. In one of our sample applications, we use acquisition and release to implement drag and drop.

Since our toolkit provides access to the underlying joint movement, developers can also easily incorporate an existing gesture recognizer such as the $1 Recognizer [10], which already has a JavaScript implementation.

Color and Depth Stream Data

Our toolkit also provides access to the color and depth camera image streams. Since the frame rate is up to 30 fps, the maximum bandwidth required across both streams is approximately 28.6 MB/sec. To reduce the load on JavaScript garbage collectors, we only copy data between the device and the browser on demand. This allows developers to request only as much data as they need without burdening the CPU unnecessarily. For convenience, the image data is available as a `CanvasImageData` object which can be drawn directly in the browser using the HTML5 Canvas API.

EVALUATION

To examine our toolkit's usability, we conducted a small informal user study. We recruited four participants from a Kinect interest mailing list at a large technology company. All participants were software professionals and had at least some experience using the Microsoft Kinect SDK as well as self-rated intermediate or expert experience with JavaScript and HTML. None were familiar with our toolkit.

We provided subjects with a short document describing the toolkit and detailing the available events. We then observed the developers completing two tasks each. In both tasks, participants were given an HTML file containing boilerplate code to initialize the framework (similar to Fig. 1, but without the event listener in Fig 1.3). In Task 1, we instructed them to update two regions of the page (`SPAN` tags) to display the X and Y coordinates of the right hand. All participants successfully completed the task in less than ten minutes. In Task 2, we created a simulated "scratch off" lottery ticket. In the HTML given to the participants, four rectangular `DIV`s with numbers were occluded by four opaque rectangular `DIV`s in front along the z-axis. Participants had to use the toolkit to make the opaque `DIV`s disappear so as to simulate scratching off the removable surface of a scratch-and-win lottery ticket. All participants successfully completed the task in 17 minutes or less.

After completing the tasks, subjects completed a post-study questionnaire. After less than an hour of exposure to the toolkit and minimal documentation, they expressed either Agree or Strongly Agree (on a 5-point Likert scale) that they understood how the toolkit mapped physical space into the browser and how to use the toolkit to respond to events. Furthermore, participants felt that expert proficiency was not required to use the system. We believe this is important given that Web development skill level varies and we want the system to be usable by a wide audience.

To illustrate the versatility of our toolkit, the authors and other members of our lab used Kinected Browser to create novel Web browser interactions. One team created an interactive information visualization that a user could manipulate using a combination of speech (to specify dimensions to manipulate) and gestures (to adjust the values of the target dimensions in the graph). Another developer used the system to build a multi-user search experience, in which voice queries directed to a search engine were color-coded based on user identity, and gesture recognition was used to implement voting to select a search result for viewing, by ensuring that all users simultaneously hovered over a common link in order to choose it. A third demonstration app let users select text on a page by pointing, then use the selected text to issue a search query initiated by speech control. These example applications showcase the flexibility that the Kinected Browser enables.

CONCLUSION

We introduced Kinected Browser, a toolkit that facilitates augmenting a Web browser with depth camera and gestural interactions using the familiar DOM-based event model of JavaScript programming, thus facilitating Web interaction for new form-factors such as large display walls, which may lack mice and keyboards or necessitate multi-user input. We discussed the design rationale and architecture involved in creating Kinected Browser and presented an initial evaluation of the toolkit's usability and versatility. Our toolkit is available for download at http://aka.ms/kib.

REFERENCES

1. Amershi, S. and Morris, M.R. CoSearch: A System for Co-located Collaborative Web Search. *CHI 2008*.
2. Harrison, C., et al. OmniTouch: Wearable Multitouch Interaction Everywhere. *UIST 2011*.
3. Izadi, S., et al. KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera. *UIST 2011*.
4. Lindley, S.E., et al. "It's simply integral to what I do:" Enquiries into how the web is weaved into everyday life. *WWW 2012*.
5. Morris, D., et al. SuperBreak: Using Interactivity to Enhance Ergonomic Typing Breaks. *CHI 2008*.
6. Paek, T., et al. Toward Universal Mobile Interaction for Shared Displays. *CSCW 2004*.
7. Pawar, U.S., et al. Multiple Mice for Retention Tasks in Disadvantaged Schools. *CHI 2007*.
8. Wilson, A.D. Using a Depth Camera as a Touch Sensor. *ITS 2010*.
9. Wilson, A.D. and Benko, H. Combining multiple depth cameras and projectors for interactions on, above, and between surfaces. *UIST 2010*.
10. Wobbrock, J., et al. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *UIST 2007*.