Least Privilege Rendering in a 3D Web Browser

John Vilk[1], David Molnar[2], Eyal Ofek[2], Chris Rossbach[2], Benjamin Livshits[2], Alexander Moshchuk[2], Helen J. Wang[2], and Ran Gal[2]

[1]University of Massachusetts Amherst

[2]Microsoft Research

ABSTRACT

Emerging platforms such as Kinect, Epson Moverio, or Meta SpaceGlasses enable *immersive experiences*, where applications display content on multiple walls and multiple devices, detect objects in the world, and display content near those objects. App stores for these platforms enable users to run applications from third parties. Unfortunately, to display content properly near objects and on room surfaces, these applications need highly sensitive information, such as video and depth streams from the room, thus creating a serious privacy problem for app users.

To solve this problem, we introduce two new abstractions enabling *least privilege* interactions of apps with the room. First, a *room skeleton* that provides least privilege for *rendering*, unlike previous approaches that focus on inputs alone. Second, a *detection sandbox* that allows registering content to show if an object is detected, but prevents the application from knowing if the object is present.

To demonstrate our ideas, we have built SurroundWeb, a *3D browser* that enables web applications to use object recognition and room display capabilities with our least privilege abstractions. We used SurroundWeb to build applications for immersive presentation experiences, karaoke, etc. To assess the privacy of our approach, we used user surveys to demonstrate that the information revealed by our abstractions is acceptable. SurroundWeb does not lead to unacceptable runtime overheads: after a one-time setup procedure that scans a room for projectable surfaces in about a minute, our prototype can render immersive multi-display web rooms at greater than 30 frames per second with up to 25 screens and up to a 1,440×720 display.

1. INTRODUCTION

Advances in depth mapping, projectors, and object recognition have made it possible to create immersive experiences inspired by Star Trek's Holodeck [2, 4, 9, 12, 13, 21]. Immersive experiences display content on multiple walls and multiple devices, potentially making every object in the room a target for interaction. These experiences can detect the presence of objects in the room and adapt content to match, as well as interact with users using gesture and voice.

Motivating example: Figure 1 shows a photograph of a presenter using SurroundPoint, one of the applications we explore in this paper, an immersive presentation application running in an office with a high definition monitor and a projector. One can think of SurroundPoint as *immersive PowerPoint*. The monitor in the center shows the main presentation slide, while the projector shows additional content "spilling out" of the slide and onto the walls of the room.

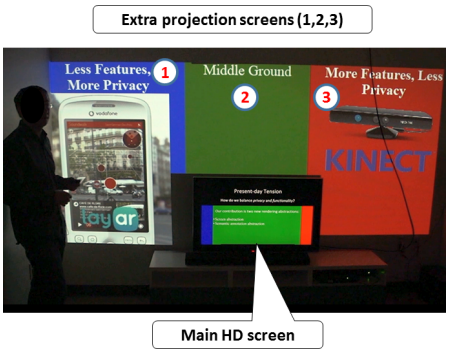
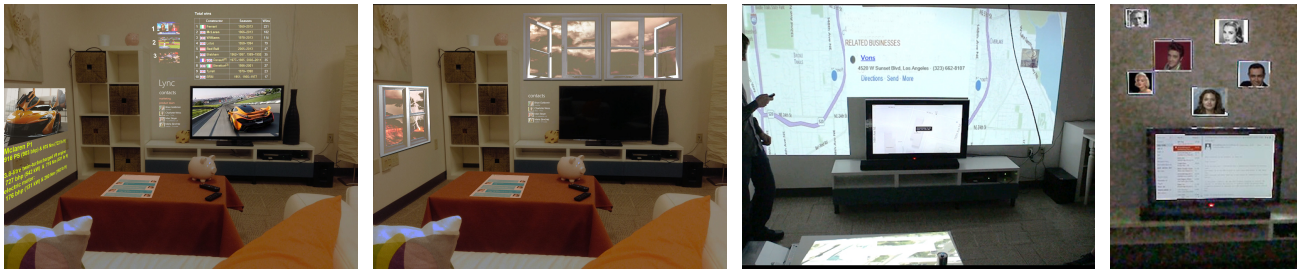


Figure 1: SurroundPoint: an immersive presentation experience.

Third-party apps raise privacy concerns: Emerging high-field-of-view head-mounted displays such as Epson Moverio or Meta SpaceGlasses [16], as well as established platforms such as Microsoft Kinect, allow third party developers to create applications with immersive experiences. App stores for these platforms enable users to download applications from untrusted third parties.

In these environments, an application detects objects using raw video or depth camera feeds, then renders content near detected objects on a display window. Unfortunately, giving raw depth and video feeds to an untrusted application raises significant privacy concerns. From mobile phones, we have learned how dangerous it is to give devices unrestricted access to sensor output. Even seemingly innocuous information such as GPS traces can betray sensitive information, such as inferring gender orientation from which bars a person frequents [15]. Similarly, from raw video and depth streams inside a home, it is likely possible to infer economic status, health information, and other sensitive information. Therefore we do not want to expose raw sensor data to applications. Our goal is to build applications in a *least privilege* way: they receive the information they need to operate and no more.

Least privilege rendering impossible today: Today's platforms can not provide *least privilege* rendering for immersive room experiences. The *window abstraction* in today's browsers and operating systems gives applications control over a two-dimensional rectangle of content on a display. To render content coherently on surfaces in the world or near detected objects, the application needs a mapping from window coordinates to world coordinates. Because today's operating systems do not provide such a mapping, the application must create it from video and depth camera feeds, which reveal



(a) Car racing news site. **(b)** Virtual windows **(c)** Road maps **(d)** Projected awareness IM

Figure 2: Four web rooms enabled by SurroundWeb, shown with multiple projectors and an HDTV.

Experience	Requires	Description
SurroundPoint	Room Skeleton	Each screen in the room becomes a rendering surface for a room-wide presentation (see Figure 1).
Car Racing News Site	Room Skeleton	Live video feed displays on a central monitor, with racing results projected around it (see Figure 2a).
Virtual Windows	Room Skeleton	"Virtual windows" render on surfaces around the room that display scenery from distant places (see Figure 2b).
Road Maps	Room Skeleton	Active map area displays on a central screen, with surrounding area projected around it (see Figure 2c).
Projected Awareness IM [4]	Room Skeleton	Instant messages display on a central screen, with frequent contacts projected above (see Figure 2d). This is an example of Focus+Context from UI research [3].
Karaoke	Room Skeleton	Song lyrics appear above a central screen, with music videos playing around the room (see Figure 5).
SmartGlass [17]	Satellite Screens	Xbox SmartGlass turns a smartphone or tablet into a second screen; a web page can use Satellite Screens to turn a smartphone or tablet into a screen for the web room.
Multiplayer Poker	Satellite Screens	Each user views their cards on a Satellite Screen on a smartphone or tablet, with the public state of the game displayed on a surface in the room.
Advertisements	Detection Sandbox	Advertisements can register content to display near particular room objects detected by the Detection Sandbox without knowing their locations or presence.
Kitchen Monitor	Detection Sandbox	The Kitchen Monitor displays alerts in the kitchen when water is boiling *without* knowing this information through registering content to display near boiling water.

Figure 3: Web applications and immersive experiences that are possible with SurroundWeb.

too much information to the application.

Recent work on new abstractions for augmented reality applications is also not sufficient. Examples of such work include adding a higher-level "recognizer abstraction" to the OS [10] or injecting noise into sensor data via "privacy transforms" [11] to limit sensitive information exposure. These approaches, however, focus on application *inputs*. In contrast, we need a way to manage *rendering*. For example, knowing that there are flat surfaces in the room, or even where they are, does not by itself let an application place content on those surfaces. Therefore no previous work provides a rendering mechanism that enables least privilege for immersive experiences.

Furthermore, previous approaches reveal when an object is present. Sometimes, the mere presence of an object may be sensitive, yet it would be beneficial to adapt content to the object's presence. Below we discuss an example application that checks for boiling water in a kitchen. If boiling water is present, the application displays an alert to the user in a place the user can see it. No previous approach enables an application to adapt itself to objects without leaking the object's presence to the application.

1.1 Least Privilege Rendering Abstractions

We introduce two novel abstractions that enable least privilege for immersive experiences. First, the *Room Skeleton*, which captures the minimal information needed for rendering in a room. The Room Skeleton contains a list of *Screens*. Each Screen is an object containing dimensions, relative location, and input capabilities of a display. A trusted kernel creates the Room Skeleton by scanning the room and looking for monitors or projectable surfaces to expose as Screens to the web page. In our prototype, this is a one-time scan, but future work could dynamically update the list of Screens as the room changes. We also show how to extend the Room Skeleton with *Satellite Screens* that host web page content on remote phones, tablets, or other devices. Web pages can query the Room Skeleton to discover the room configuration but cannot see raw sensor data. Based on the room configuration, applications adapt their content, then tell the trusted kernel to render specific content on a specific Screen. Therefore, applications can render without needing raw video, depth, or even a mapping of display coordinates to room coordinates.

Second, we introduce the *Detection Sandbox*, which mediates between applications and object detection code. The Detection Sandbox allows web pages to register content that should show if a specific object is present. Web pages use special CSS rules to inform SurroundWeb that content should be rendered into the room near a specific object. The web page, however, never learns whether the object is present. The Detection Sandbox also runs code to detect natural user inputs and automatically maps them into mouse events for web

pages. Using the Detection Sandbox does place limitations on applications. We discuss these limitations and directions for relaxing them in Section 6. Despite these limitations, a wide range of immersive experiences can be implemented using the Detection Sandbox.

1.2 SurroundWeb

In this paper we want to show that the privacy-enhancing abstraction above can be used to build realistic and attractive immersive experiences. We chose against developing a new platform, instead opting for showing how the least privilege rendering abstraction can be retrofitted onto the existing HTML stack, perhaps one of the most widely-used programming platforms today.

To this end, we have built a novel 3D Browser called SurroundWeb, which extends Internet Explorer to support room rendering, object detection, and natural user interaction capabilities. We use SurroundWeb as a platform for experimentation. Figure 3 describes some web applications that are possible with SurroundWeb, which we use for illustration throughout the paper. It is our hope that SurroundWeb will stoke innovation around novel 3D web applications and immersive room experiences.

Privacy guarantees: SurroundWeb makes three privacy guarantees using our new abstractions. The Detection Sandbox ensures *detection privacy*: the application execution does not reveal the presence or absence of an object. For example, an application can register an ad that should show if an energy drink can is present, but the application *never learns if an energy drink is in the room.* The Detection Sandbox also ensures *interaction privacy*: applications receive only inputs that users explicitly authorize by their actions.

The Room Skeleton ensures *rendering privacy*: applications can render on multiple surfaces in a room, but they learn only the number of surfaces, their relative positions, and input capabilities supported by each.

1.3 Contributions

This paper makes the following contributions:

- Two novel abstractions, the *Room Skeleton* and *Detection Sandbox* that enable *least privilege* for application display in immersive experiences. Previous work, in contrast, has focused solely on least privilege for application inputs.
- A novel system, SurroundWeb, that gives web pages access to object recognition, projected screens inside a room, and satellite screens on commodity phones or tablets. SurroundWeb provide *detection privacy*, *rendering privacy*, and *interaction privacy*, allowing users to run untrusted web pages with confidence.
- We evaluate the privacy and performance of SurroundWeb and conclude that it reveals less sensitive information to applications than previous approaches, and that its performance is encouraging.

2. SURROUNDWEB DESIGN

SurroundWeb exposes two novel abstractions to web pages: The *Room Skeleton* and the *Detection Sandbox*. These abstractions are provided by the *trusted core* of SurroundWeb, as distinguished from the untrusted web applications or pages which render using these abstractions.

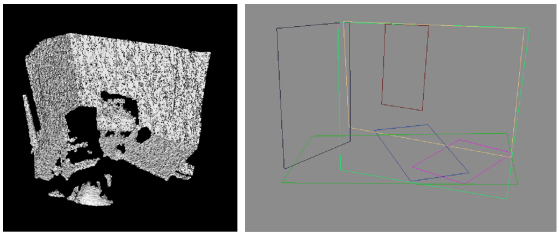


Figure 4: On the left, a 3D model reconstructed from raw depth data. SurroundWeb detects projectable "screens" to create the Room Skeleton, shown on the right. Web applications see only the Room Skeleton, never raw depth or video data.

In Section 4 we describe how these abstractions are implemented as extensions to the web programming model.

2.1 The Room Skeleton

Advances in depth mapping, such as KinectFusion [18], take raw depth information and output 3D models that reconstruct the volume and shape of items that have been "scanned" with depth cameras. These scans can be further processed to find flat surfaces in the room that can host two-dimensional content. Content can be shown on these surfaces either by projectors pointed at the surfaces, by head-mounted displays that overlay virtual content on surfaces (e.g. Meta SpaceGlasses [16]), or by looking through a phone screen and having the content superimposed on top of live video (e.g. Layar [14]). Figure 4, on the left, shows a 3D model of a real room and on the right shows the location of flat surfaces that could host content.

In our prototype, we perform a one-time setup phase which first detects flat surfaces in a room. Next, SurroundWeb discovers all display devices that are available and determines which of them can show content on the available surfaces. Finally, SurroundWeb discovers which input events are supported for which displays. For example, a touchscreen monitor supports touch events, and depth cameras can be used to support touch events on projected flat surfaces [20]. Future work could enable dynamic room scanning to account for movement of objects in the room that could create or obscure screens.

The result of scanning is the SurroundWeb *Room Skeleton*. We call this the Room Skeleton in analogy with the skeleton pose detector found in the Microsoft Kinect. The Kinect skeleton captures a core of essential information, the position and pose of the user. This essential information is sufficient for new experiences, but it does not include incidental information in a video and depth stream. Our goal with the room skeleton is to similarly capture a set of core information that enables web pages to render while leaving out unnecessary incidental information.

The Room Skeleton consists of a set of *Screens*. Each Screen has a resolution, a relative location to other screens, and a *capabilities* array. This is an array of strings that encodes the input events that can be accepted by the Screen. For example, these may include "none," "mouse," or "touch." Web pages loaded in SurroundWeb access the Room Skeleton through JavaScript. By querying the Room Skeleton, web pages can dynamically discover the room's capabilities and adapt their content accordingly. The web page can then explictly inform SurroundWeb which sub-portions of a page should be rendered on which Screens, similar to the way today's web pages make rendering decisions at the granularity

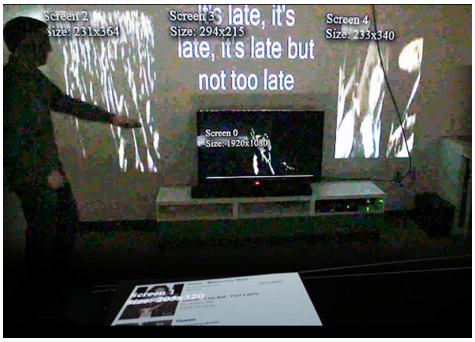


Figure 5: Karaoke uses the Room Skeleton to render karaoke lyrics across a TV and projectors.

of `div` elements. We describe the interface in Section 4.

Sample application: SurroundPoint: The SurroundPoint application described in Section 1 and pictured in Figure 1 uses the Room Skeleton. The presentation page contains several slides. Each slide has a set of "main" content, plus optional additional content. By querying the Room Skeleton, the page adapts the presentation to different settings. Consider the case where the room has only a single 1,080p monitor and no projectors, such as running on a laptop or in a conference room. Here, the Room Skeleton contains only one Screen: a single 1,920×1,080 rectangle. Based on this information, SurroundPoint knows that it should show only the "main" content. In contrast, consider the room shown in Figure 4. This room contains multiple projectable Screens, exposed through the Room Skeleton. SurroundPoint can detect that there is a monitor plus additional peripheral Screens that can be used for showing the optional additional content.

Sample application: Karaoke: Another application is Karaoke shown in Figure 5. It uses the Room Skeleton to render karaoke lyrics across the wall behind the TV, along with some images to the left and right of the lyrics.

2.2 The Detection Sandbox

Advances in object detection make it possible to quickly and relatively accurately determine the presence and location of many objects or people in a room. Object detection is a privacy challenge because the presence of objects can reveal sensitive information about a user's life. On the other hand, object detection makes possible new experiences. This creates a tension between privacy and functionality.

SurroundWeb resolves this tension by introducing a *Detection Sandbox*. All object recognition code runs as part of the trusted core of SurroundWeb. Web pages never receive events from this object recognition code directly. Instead, pages *register* content up front with the Detection Sandbox using a system of *rendering constraints* that can reference physical objects. In Section 4 we show how these are exposed via Cascading Style Sheets. After the page loads, SurroundWeb checks this registered content against a list of objects detected. If there is a match, SurroundWeb renders the content, but the web page does not receive notification that the content has been shown. SurroundWeb further suppresses input events to the registered content, which ensures that user inputs do not reveal to the web page whether the content has been shown or not.

Sample application: Kitchen Monitor: Using a detector that determines whether water is boiling, a Kitchen Monitor application could help users monitor their kitchens without leaking information to the web server.

2.3 Satellite Screens

In our discussion of the Room Skeleton above, we talked about fixed, flat surfaces present in a room. Today, however, many people have personal mobile devices such as mobile phones or tablets. SurroundWeb supports these through an abstraction called a *Satellite Screen*. By navigating to a URL of a SurroundWeb cloud service, phones, tablets, or anything with a web browser can register with the main SurroundWeb. JavaScript running in the web application discovers the device's screen size and input capabilities, then communicates these to the SurroundWeb trusted core. The SurroundWeb trusted core then adds the Satellite Screen to the Room Skeleton and notifies the web page. We describe our implementation in Section 4.

Sample application: Poker: Satellite Screens enable web pages that need *private displays*. For example, a poker web site might use a shared high-resolution display to show the public state of the game. As players join personal phones or tablets as Satellite Screens, however, the application shows each player's hand on her own device. Players can also make bets by pressing input buttons on their own device. More generally, Satellite Screens allow web sites to build multi-player experiences without needing to explicitly tackle creating a distributed system, as all Satellite Screens are part of the same DOM and exposed via the same Room Skeleton.

3. PRIVACY PROPERTIES

SurroundWeb provides three privacy properties: *detection privacy*, *rendering privacy*, and *interaction privacy*. We explain each in detail, elaborating on how we provide them in the design of SurroundWeb. We then discuss important limitations and how they may be addressed.

3.1 Detection Privacy

Detection privacy means that a web page can customize itself based on the presence of an object in the room, but the web server never learns whether the object is present or not. Without detection privacy, web applications or pages could scan a room and look for items that reveal sensitive information about a user's lifestyle.

For example, an e-commerce site could scan a room to detect valuable items, make an estimate of the user's net worth, and then adjust the prices it offers to the user accordingly. For another example, a web site could use optical character recognition to "read" documents left in a room, potentially learning sensitive information such as social security numbers, credit card numbers, or other financial data.

Because the *presence* of these objects is sensitive, these privacy threats apply even if the web page has access to a high-level API for detecting objects and their properties, instead of raw video and depth streams [10]. At the same time, as we argued above, continuous object recognition enables new experiences. *Therefore, detection privacy is an important goal for balancing privacy and functionality in immersive room experiences.*

SurroundWeb provides detection privacy using the Detection Sandbox. Our threat model for detection privacy in

SurroundWeb is that web pages are allowed to register arbitrary content in the Detection Sandbox. In SurroundWeb, this registration takes the form of *rendering constraints* specified relative to a physical object's position, which tell SurroundWeb where to render the registered content. We describe this mechanism in more detail in Section 4. Because the rendering process is handled by the trusted core of SurroundWeb, the web server never learns whether an object is present or not, no matter what is placed in the Detection Sandbox. Our approach places limitations on web pages, both fundamental to the concept of the Detection Sandbox and artifacts of our current approach. We discuss these in detail in Section 6.

3.2 Rendering Privacy

Rendering privacy means that a web page can render into a room, but it learns no information about the room beyond an explicitly specified set of properties needed to render. Without rendering privacy, web applications would need continuous access to raw video and depth streams to provide immersive room experiences. This, in turn, would reveal large amounts of incidental sensitive information, such as the faces and pictures of people present, items present in the room, or the contents of documents left in view of the system. Without this access, however, web applications would not know where to place virtual objects on displays to make them interact with real world room geometry. *Therefore, rendering privacy is an important goal for balancing privacy and functionality in immersive room experiences.*

The challenge in rendering privacy is creating an abstraction that enables *least privilege for rendering.* In SurroundWeb, this abstraction the Room Skeleton. Our threat model for rendering privacy is that web applications are allowed to query the Room Skeleton to discover Screens, their capabilities, and their relative locations, as we described above. Unlike with the Detection Sandbox, we explicitly allow the web server to learn the information in the Room Skeleton. The rendering privacy guarantee is different from the detection private guarantee, because in this case we explicitly leak a specific set of information to the server, while with detection privacy we leak no information about the presence or absence of objects. User surveys in Section 5 show that revealing this information is acceptable to users.

3.3 Interaction Privacy

Interaction privacy means that a web page can receive natural user inputs from users, but it does not see other information such as the user's appearance or how many people are present. Interaction privacy is important because sensing interactions usually requires sensing people directly. For example, without a system that supports interaction privacy, a web page that uses gesture controls could potentially see a user while she is naked or see faces of people in a room. This kind of information is even more sensitive than the objects in the room.

In SurroundWeb, we provide interaction privacy through a combination of two mechanisms. First, the trusted core of SurroundWeb runs all natural user interaction detection code, such as gesture detection. Just as with the Detection Sandbox above, web applications never talk directly to gesture detection code. This means that web applications cannot directly access sensitive information about the user.

Second, SurroundWeb maps from natural user gestures to existing web events, such as mouse events. We perform this remapping to enable interactions with web applications even if those applications have not been specifically enhanced for natural gesture interaction. These web applications are never explicitly informed that they are interacting with a user through gesture detection, as opposed to through a mouse and keyboard. Our choice to focus on remapping gestures to existing web events does limit web applications. In Section 6 we discuss how this could be relaxed while keeping the spirit of the SurroundWeb design.

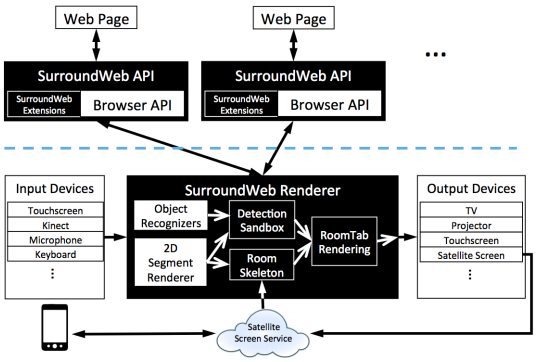


Figure 6: Architectural diagram of SurroundWeb.

4. IMPLEMENTATION

Our prototype, SurroundWeb, extends Internet Explorer by embedding a `WebBrowser` control in a C# application. Our core application implements the architecture shown in Figure 6. We first describe its core capabilities, then we show how they are exposed to web applications and pages through HTML, CSS, and JavaScript.

Figure 6 displays an architectural diagram of SurroundWeb. We show the parts we implemented in black. Items below the dashed line form the *trusted core* of SurroundWeb, while items above the dashed line are part of web pages running on SurroundWeb.

4.1 Core Capabilities

Screen detection: Our prototype is capable of scanning a room for unoccluded flat surfaces. Our prototype performs offline surface detection: after a one-time scan, our prototype maps segments of rendered content into a room using projectors. We use KinectFusion [18], as well as methods we have designed for finding flat surfaces from noisy depth data produced by Kinect.

Object detection sandbox: The trusted core of SurroundWeb receives continuous depth and video feeds from Kinect cameras attached to the machine running SurroundWeb. On each depth and video frame, we run classifiers to detect the presence of objects. In our prototype, we support detecting different types of soft drink cans, using a nearest-neighbor classifier based on color image histograms. After an object is detected, SurroundWeb checks the current web page for registered content, then updates its rendering of the room.

Natural user interaction remapping: In addition to object detection, the trusted core of SurroundWeb also continuously runs code for detecting people and gestures. We use the the Microsoft Kinect SDK to detect user position and gestures, including push and swipe gestures. Figure 7

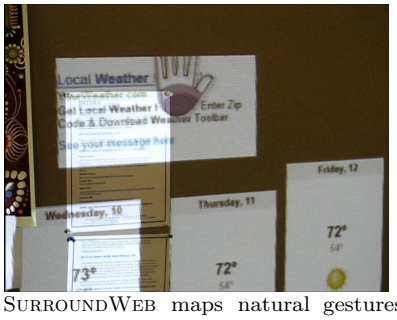


Figure 7: SurroundWeb maps natural gestures to mouse events. Here, the user uses a standard "push" gesture to tell SurroundWeb to inject a click event into the web application.

shows a photograph of SurroundWeb detecting a user's hand position and that the user is performing a pushing gesture. After a gesture is detected, SurroundWeb maps the gesture to a mouse or keyboard event, then injects that event into the running web page.

Satellite Screens: We host a SurroundWeb "satellite screen service" in Microsoft Azure. Users can point their phone, tablet, or other device with a browser to a specific URL associated with the running SurroundWeb instance. The front end runs JavaScript that discovers the browser's capabilities, then sets a cookie in the browser containing a screen unique identifier and finally registers this new screen with a service backend.

The backend informs the trusted core of the running SurroundWeb instance that a new satellite screen is available. The trusted core in turn creates an event informing the web application of the new screen. If the web application renders to this screen, the trusted core ships the rendered image to the backend, which then signals the front end to update the web application showing in the browser. Input events from the satellite screen are proxied to the web application running on SurroundWeb similarly.

4.2 HTML Extensions

Web room: A *web room* is a web application that uses SurroundWeb's extensions to HTML, CSS, and JavaScript to render content around a physical room. While a web application is limited to a browser window on a single monitor, web rooms can place content in multiple locations and displays at once. Much like web applications, web rooms can embed content, including scripts, from other web sites, which we describe below.

Room tab: A *room tab* is analogous to a tab in a regular desktop browser. Each room tab contains content from a single web room. When the user switches room tabs, all of the segments from the previous room tab vanish from the screens in the room. This feature avoids issues with content provenance; at any given time, the user can be assured that all of the displayed content comes from one web room.

Trusted user interface: A standard desktop web browser typically has a number of trusted components: A URL bar, navigation buttons, and a row of open tabs. SurroundWeb has a trusted UI that displays the URL of the currently-displayed web room, controls for switching between room tabs, and a display that outlines the segments that should be visible on each screen in the room. This lets the user determine the provenance of the content currently rendered in the room, and ensures that web applications cannot hide invisible content the user could unintentionally interact with.

Segments: SurroundWeb's rendering abstractions use rectangles of content called *segments*, which can be assigned to particular Screens. The web revolves around rectangular pieces of content called elements that web designers assemble together into a tree structure: the Document Object Model (DOM), which the browser displays as a web application. Many of these elements correspond to HTML container tags, which encapsulate a subtree of web application content.

We introduce the `segment` container tag to HTML, which annotates arbitrary HTML content as a segment. For example, "Hello World" would look like this:

```
<segment>Hello World!</segment>
```

The `segment` tag supports four size-related CSS attributes that other container tags support: `min-width`, `min-height`, `width`, and `height`. Other than these size-related attributes, `segment` tags do not influence the 2D layout of the content contained within them. `segment` tags differ from other HTML tags, such as `div`, in that they are not visible to the user unless the application specifies a target screen or object-relative constraint for them using CSS (Section 4.3) or JavaScript (Section 4.4).

By displaying subtrees of the DOM tree rather than creating a new DOM tree for each segment, we enable web rooms to function as a single unit rather than a set of separate units. This is convenient from a development standpoint, as web developers can develop web rooms in the same manner as web sites. If we instead implemented segments using frames, then it would be more difficult for a web application to update content across multiple segments at once, as each frame has its own separate DOM tree.

Content rendering: The architecture of SurroundWeb's renderer, displayed in Figure 6, resembles a conventional web browser. The block marked "SurroundWeb API" encapsulates existing browser HTML, CSS, and JavaScript functionality, and our extensions described below. The API communicates with the renderer, which renders the individual segments that the web room identifies using existing 2D browser rendering technology (the "2D Segment Renderer").

When the user navigates to a web application, our prototype first renders the entire web application using Internet Explorer. We then extract individual bitmaps for each `segment` tag. SurroundWeb combines each rendered segment with the information that the web room provides on where it should be placed. If the web room places a segment using the screen abstraction, then SurroundWeb is aware of the particular screen that it should display the content on, and can immediately render the content in the room. If the web room places a segment with object-relative constraints, then the renderer extracts constraints and solves them with a constraint solver to determine the rendering location of the segment.

All final rendering locations are passed to "Room Tab Rendering", which displays the rendered segments in the room using attached display devices. When the web room alters the contents of a segment, the prototype captures a new bitmap and updates the rendered segment in the room. As a result, our prototype supports dynamic animated content.

4.3 CSS Extensions

Web applications and pages use Cascading Style Sheets (CSS) to style and position content on the page. Sur-

Property	Description
`getAll()`	(Static) Returns an array of all of the screens in the room.
`id`	A unique string identifier for this screen.
`ppi`	The number of pixels per inch.
`height`	Height of the screen in pixels.
`width`	Width of the screen in pixels.
`capabilities`	List of JavaScript events supported on this screen.
`location`	Location of the screen in the room as an object literal, with fields 'ul' (upper-left) and 'lr' (lower-right) each containing an object literal with x, y, and z fields.

Figure 8: Properties of each screen object.

roundWeb adds CSS *object-relative constraints* for declaratively specifying the position of `segment` elements relative to physical objects in the room, such as `left-of` or `below`. CSS is a natural fit for these constraints, as CSS already controls the placement of content through various attributes, such as `position` and `margin`.

Each constraint can be assigned a list of object names, which specifies how the segment should be placed among objects detected in the room. For example, a web room may use the `below` constraint assigned to the object `EnergyDrink` on a `segment` containing a tea advertisement. We assume a central list of well-known list of names for objects whose detection may or may not be supported by the specific instance of SurroundWeb, just as the web today has a well-known list of names for events.

4.4 JavaScript Extensions

Current browsers expose comprehensive functionality through JavaScript for dynamically responding to events and altering page structure, content, and style. It is possible to construct an entire web page on-the-fly using JavaScript and standard browser APIs to inject HTML and CSS into the page. SurroundWeb continues this tradition through exposing all of its functionality through JavaScript.

Screens: Screens are a read-only global property of the current room. The web room can retrieve an array of all of the screens in the room through the `Screen.getAll()` procedure, and can use the properties of each screen to determine how to distribute content among them.

Figure 8 displays the properties available on each screen object. Two properties are worth discussing further in context of the web: `id` and `capabilities`. We add the `id` property to screens for the web environment so they can be referenced from dynamically constructed HTML and CSS. Since HTML and CSS are text formats, they are require this text-based identifier in order to reference individual screens.

The `capabilities` property is an array of strings that correspond to JavaScript events that are supported on that particular screen. Every JavaScript event type has a standardized name that web applications use to register event listeners with the browser. Modern web browsers have events for many devices, including accelerometers, touch screens, mice, and keyboards. Web rooms can use the `capabilities` property to determine which events are appropriate to listen for on a particular screen, and to make decisions on where certain interactive content should be placed.

Our prototype does not directly extend the CSS parsing of Internet Explorer. Instead, we emulate these extensions using a JavaScript API that can be called by web applications. The trusted SurroundWeb renderer as part of the rendering process compiles these constraints into a format accepted by a constraint solver, then attempts to solve them. If successful, the renderer uses the solution to place the segments in the room.

Segments: Segments can be dynamically constructed like any other HTML element: use `document.createElement("segment")` to create a new `<segment>` tag, modify its properties, and then insert it somewhere into the DOM tree so it becomes "active". To display an active segment on a particular screen, the web room must assign the screen's `id` property to the segment's `screen` property. The size of the segment and object-relative constraints can be specified using the standard JavaScript APIs for manipulating CSS properties.

4.5 Embedding Web Content

We have described *room tabs* as analogous to tabs in a desktop web browser: each room tab encapsulates content from a single web room. We now discuss how to handle embedding of content from different origins inside a single room tab. In this discussion, we use the same web site principal as defined in the same-origin policy (SOP) for web rooms, which is labeled by a web site's origin: the 3-tuple $\langle$`protocol`, `domainname`, `port`$\rangle$.

In an effort to be backward-compatible and to preserve concepts that are familiar to the developer, we extend the security model present on the web today: web rooms can embed content from other origins, such as scripts and images, and can use Content Security Policy (CSP) to restrict which origins it can embed particular types of content from. Like in the web today, scripts embedded from other origins will have full access to the web room's Document Object Model (DOM), which includes all browser APIs, SurroundWeb extensions, and segments defined by the web room.

We preserve this property for compatibility reasons, as many current web sites and JavaScript libraries rely on the ability to load popular scripts, such as jQuery, from Content Distribution Networks (CDNs). Since SurroundWeb extends HTML, JavaScript, and CSS, these existing libraries for web sites will still have utility in web rooms.

Web rooms can use the `iframe` tag to safely embed content from untrusted origins without granting them access to the SurroundWeb extensions. In the current web, a frame has a separate DOM from the embedding site. If the frame is from a different origin than the embedding site, then the embedded origin and the embedding origin cannot access each other's DOM. We extend CSP to allow web rooms to control whether or not particular origins can access the SurroundWeb extensions from within a frame. If a web room denies an embedded origin access to these extensions, then the `iframe` will render as it does today: to a fixed-size rectangle that the embedding origin can control the placement of. If the web room allows an embedded origin access to these extensions, then the `iframe` will be able to render content to that web room's room tab.

4.6 Walkthrough: Karaoke Application

To illustrate how a web room can be developed using SurroundWeb, we will walk through a sample Karaoke application, shown in Figure 5. This web room renders karaoke lyrics above a central screen, with a video on the central screen and pictures around the screen. Related songs are rendered on the table.

The web room contains the following HTML:

```
<segment id="lyrics"><!--Lyrics HTML--></segment>
<segment id="video"><!--Video HTML--></segment>
<segment id="related">
  <!--Related songs HTML--></segment>
```

The web room must scan the *Room Skeleton* to assign the segments specified in the HTML to relevant *Screens*.
1) Using JavaScript, the web room locates the vertical screen with the highest resolution, which will contain the video:

```
var screens = Screen.getAll(), bigVScn, maxPpi = 0;
function isVertical(scn) {
  var scnLoc=scn.location,ul=scnLoc.ul,lr=scnLoc.lr,
    zDelta = Math.abs(ul.z - lr.z),
    xDelta = Math.abs(ul.x - lr.x),
    yDelta = Math.abs(ul.y - lr.y);
  return zDelta > xDelta || zDelta > yDelta;
}
// Find the highest resolution vertical screen
screens.forEach(function(scn) {
  if (isVertical(scn) && scn.ppi > maxPpi)
    bigVScn = scn;
  maxPpi = bigVScn.ppi;
});
// Assign video to screen.
document.getElementById('video')
  .setAttribute('screen', bigVScn.id);
```

2) The web room determines the closest vertical screen above the main screen, and renders the karaoke lyrics to it. In the code below, z is the distance from the floor:

```
var aboveScn, bigLoc = bigVScn.location;
screens.forEach(function(scn) {
  if (!isVertical(scn) || scn === bigVScn) return;
  var scnLoc=scn.location,ul=scnLoc.ul,lr=scnLoc.lr;
  if (lr.z > bigLoc.ul.z) {
    // scn is above bigVScn
    if (aboveScn) {
      // Is scn closer to bigVScn than aboveScn?
      if (aboveScn.location.lr.z > lr.z)
        aboveScn = scn;
    }
    else aboveScn = scn;
  }
});
// Assign lyrics to screen.
document.getElementById('lyrics')
  .setAttribute('screen', aboveScn.id);
```

3) For the listing of related videos, the application locates the largest horizontal screen in the room:

```
var bigHScn, maxArea = 0;
screens.forEach(function(scn) {
  var area = scn.height*scn.width;
  if (!isVertical(scn) && area > maxArea) {
    maxArea = area; bigHScn = scn;
  }
});
// Assign related videos to screen.
document.getElementById('related')
  .setAttribute('screen', aboveScn.id);
```

4) Finally, the application assigns random media to render on other screens:

```
screens.forEach(function(scn) {
  if(scn!==aboveScn&&scn!=bigHScn&&scn!=bigVScn)
    renderMedia(scn);
});
function renderMedia(scn) {
  var newSgm = document.createElement('segment');
  newSgm.setAttribute('screen', scn.id);
  newSgm.appendChild(constructRandomMedia());
  document.body.appendChild(newSgm);
}
```

Now that the rendering code has finished, the karaoke application can update each screen in the same manner that a vanilla web site updates individual HTML elements on the page. Should the chosen configuration be suboptimal for the user, the Karaoke application can provide controls that allow the user to dynamically choose the rendering location of the segments.

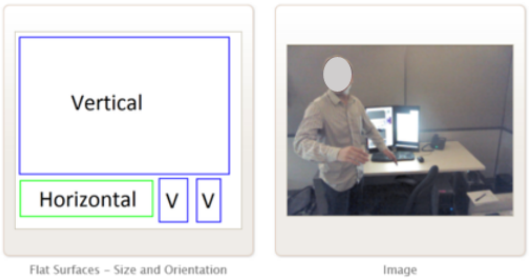


Figure 9: Excerpt from our survey asking respondents to evaluate information released by SurroundWeb. On the right, a color image of a room. On the left, a visualization of the information provided by our screen abstraction about the same room. Our instructions were as follows: *For each question, only consider the information explictly displayed in the two images below. Which image contains* **more information** *that you consider "private"?* To preserve the anonymity of the submission, we have modified the image to remove an author's face.

5. EVALUATION

Our focus in this section is two-fold. We want to evaluate whether our design delivers adequate privacy guarantees, which we do via a user study and whether the performance of SurroundWeb is acceptable.

5.1 Privacy

Room skeleton sensitivity: We used surveys to measure user privacy attitudes toward the Room Skeleton information that SurroundWeb reveals to web pages. We first asked survey participants to read a short summary of the capabilities of SurroundWeb-like systems.

Next, we showed users two pictures: a color picture of a room and a visualization of the data exposed from that room by the Room Skeleton. An example of such a survey is shown in Figure 9. Prior to running our surveys, we reviewed our questions, the data we proposed to collect, and our choice of survey provider with our institution's group responsible for protecting the privacy and safety of human subjects.

Out of 50 respondents, 78% claimed that they felt that the raw data was more sensitive than the information that SurroundWeb exposes to web pages. After combing through the data to filter out people who did not understand the survey and to reassign those who mistakenly chose the information they felt was *most* sensitive, that figure changes to 87.5%. This supports our choice of data to reveal in the Room Skeleton.

Web room-specific surveys: Next, we developed a survey that explored a broader set of possible data that could be released to web rooms by SurroundWeb in the context of different web rooms. We presented 50 survey-takers with three different web room descriptions: a "911 Assist" application that detects falls and calls 911, a "Dance Game" that asks users to mimic dance moves, and a "Media Player" that plays videos. We asked them which information they would feel comfortable sharing with each web room. We have the following additional findings.

Room type	Scanning Time (s)	# Planes Found
Living room # 1	30	19
Office	70	7
Living room #2	17	13

Figure 10: Scanning times and results for 3 representative rooms.

- Users have different privacy preferences for different applications. User privacy preferences depended on the benefit the application would provide. For example, when asked about a hypothetical *911 Assist* app, one person stated, "It seems like it would only be used in an emergency and only communicated to emergency personnel", and another said "Any info that would help with 911 is worth giving". Users also evaluated whether the application needed the information. For example, one person said, "A dance game would not need more information than the general outline or placements of the body". Finally, in some cases respondents thought creatively about how an application could use additional information. In particular, one respondent suggested that the Video Player application could adjust the projection of the video to "where you are watching". These support a design that gives different fine-grained permission levels to different web rooms.
- Users did not distinguish between screen sizes and room geometry. "Screen sizes" refers only to the number, orientation, and capabilities of screens, while "room geometry" refers to the relative position of screen surfaces to each other. Before conducting our surveys, we hypothesized that users would find room geometry to be more sensitive than screen sizes. In fact, our data does not confirm this hypothesis. This suggests that a reasonable default for the screen abstraction would give web rooms access to room geometry as well as screen size and capabilities. This result influenced the information that we decided to release through the web extensions.

5.2 Performance

Room skeleton performance: SurroundWeb performs a one-time scan of a room to extract planes suitable for projection, using a Kinect depth camera. Figure 10 shows the results of scanning three fairly typical rooms we chose. No room took longer than 70 seconds to scan. This is reasonable as a one-time setup cost for SurroundWeb.

Detection sandbox constraint solving time: SurroundWeb uses a constraint solver to determine the position of segments that web sites register with the Detection Sandbox in the room without leaking the presence or location of an object to the web application. The speed of the constraint solver is therefore important for web applications that use the Detection Sandbox. We considered two scenarios for benchmarking the performance of the constraint solver.

- We considered the scenario where the web application registers only constraints of the form "show this segment near a specific object." Figure 11 shows how solving time increases for this scenario as the number of registered segments in a single web application grows. While we expect pages to have many fewer than 100 segments registered with the Detection Sandbox, the main point of this experiment is that constraint solving time scales linearly as the number of segments grow.

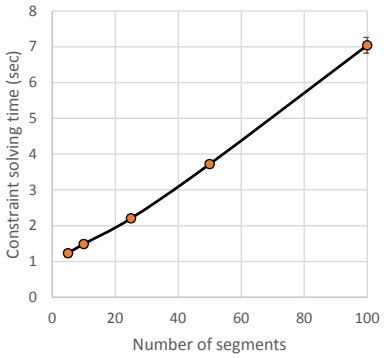


Figure 11: Solver performance as the number of segments registered with the Detection Sandbox increases. The error bars indicate 95% confidence intervals.

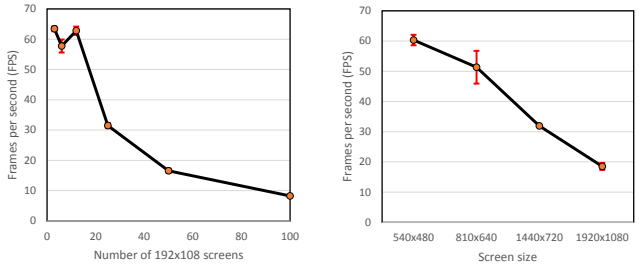


Figure 12: On the left, maximum rendering frame rate as the number of same-size screens increases. On the right, the maximum rendering frame rate of a single screen as its size increases. Error bars indicate 95% confidence intervals.

- Next, we considered the scenario where the web page uses the solver for more complicated layout. We tested a scene with 12 detected objects and 8 segments. We created a "stress" script with 30 constraints, including constraints for non-overlap in area between segments, forcing segments to have specified relative positions, specified distance bounds between segments in 3D space, and constraints on the line of sight to each segment. The constraints were solved in less than 4 seconds on one core of an Intel Core i7 2.2 GHz processor.

In both cases, only segments that use constraints incur latency. Segments rendered via Screens can display before solving finishes.

Rendering performance: We ran a benchmark that measures how fast our prototype can alter the contents of HTML5 `canvas` elements mapped to individual screens. Figure 12 displays the results. In the left configuration, the benchmark measures the frame rate as it increases the number of 192×108 screens.

In the right configuration, the benchmark measures the frame rate as it increases the size of a single screen. When there are 25 or fewer screens and screens with resolution up to 1,440×720, the prototype maintains an acceptable frame rate above 30 FPS. These numbers could be improved by tighter integration into the rendering pipeline of a web browser.

At present, our prototype must copy each frame multiple times and across language boundaries, as our prototype is

written in C# but we embed a native `WebBrowser` control. Despite these limitations, our prototype achieves reasonable frame rates.

6. LIMITATIONS AND FUTURE WORK

Detection, Rendering, and Interaction privacy presented in Section 3 are variants on a theme: enabling *least privilege* for immersive room experiences. In each case, we provide an abstraction that supports web pages, yet reveals the minimum information. We discuss limitations to the privacy properties provided by SurroundWeb, along with other future work.

Social engineering: Web applications can ask users to explicitly tell them if an object is present in the room or send information about the room to the site. These attacks are not prevented by SurroundWeb, but they also could be carried out by existing web applications.

Browser fingerprinting: *Browser fingerprinting* allows a web page to uniquely identify a user based on the instance of her browser. Our extensions add new information to the web browser that could be used to fingerprint the user, such as the location and sizes of screens in the room. We note that browser fingerprinting is far from a solved problem, with recent work showing that even seemingly robust countermeasures fail to prevent fingerprinting in standard browsers [1, 19]. We also do not solve the browser fingerprinting problem.

Clickjacking: Clickjacking is the problem of a malicious site overlaying UI elements on the elements of another site, causing the malicious site to intercept clicks intended for the other site. As a result, the browser takes an unexpected action on behalf of the user, such as authorizing a web site to access the user's information.

SurroundWeb forbids segments to overlap, guaranteeing that a user's input on a screen is received by the visible segment. This property allows web rooms to grant `iframes` access to the SurroundWeb API with the assurance that the `iframe` cannot intercept screen input events intended for the embedding site.

However, because SurroundWeb extends the existing web model for compatibility, it is possible that a web room has embedded a malicious script that uses existing web APIs to create an overlay *within* the segment. Thus, SurroundWeb does not solve the clickjacking problem as it is currently faced on the web. That said, we also do not make the clickjacking problem worse, and we do not believe our abstractions introduce new ways to perform clickjacking.

Side channels: The web platform allows introspection on documents, and different web browsers have subtly different interpretations of web APIs. Malicious JavaScript can use these differences and introspection to learn sensitive information about the user's session. One key example is *history sniffing*, where JavaScript code from malicious web applications was able to determine if a URL had been visited by querying the color of a link once rendered. While on recent browsers this property is not directly accessible to JavaScript, recent work has found multiple interactive side channels which leak whether a URL has been visited [22].

Because SurroundWeb extends the web platform, side channels that exist on the current web are still present in SurroundWeb. There may also be new side channels that reveal sensitive information about the room. For example, performance may be different in the presence or absence of an object in the room. For another example, our mapping from natural gestures to mouse events may reveal that the user is interacting with gestures or other information about the user. Characterizing and defending against such side channels is future work.

Extending the detection sandbox: In our prototype, the Detection Sandbox allows only for registering content to be displayed when specific objects are detected. We could extend this to enable matching color of an element to the color of a nearby object. As a further step, web applications might specify portions of a page or entire pages that are allowed to have access to object recognition events, in exchange for complete isolation from the web server.

These approaches would require careful consideration of how to prevent leaking information about the presence of an object through JavaScript introspection on element properties or other side channels. Our Detection Sandbox, however, does appear to rule out server-side computation dependent on object presence, barring a sandboxed and trusted server component. For example, cloud-based object recognition may require sandboxed code on the server.

Because our sandbox prevents user inputs from reaching registered content, in our prototype users can see object-dependent ads but cannot click on these ads. Previous work on privacy-preserving ads has suggested locally aggregating user clicks or using anonymizing relay services to fetch additional ad content [7]. We could explore these approaches to create privacy-friendly yet interactive object-dependent ads.

Multiple room tabs: In our prototype, we do not simultaneously show multiple tabs to prevent phishing attacks. This limitation might be overcome with a trusted UI that indicates which segments belong to which web pages. For example, the trusted core could pick a unique color for each page origin, then draw a solid border of that color around the segment when it renders.

7. RELATED WORK

Immersive Experiences: Illumiroom uses projectors combined with a standard TV screen to create gaming experiences that "spill out" of the TV and into the room [12]. Azuma surveys the broader field of augmented reality, defined as an experience with real-time registration of 3-D overlays on the real world [2]. Panelrama [23] extends HTML to enable building multi-screen experiences, but it does not handle room scanning or object detection. The ARGON mobile web browser also extends HTML to enable building augmented reality applications but it does not add support for satellite screens as we do [6]. Recent work argues these immersive experiences need first class support from operating systems, including new abstractions to enable improved security and improved performance [5].

Abstractions for Privacy: Previous work introduced the *recognizer* abstraction in an OS as the fundamental unit for application *input*. A recognizer wraps computer vision algorithms and allows applications to request only the data they need. Our approach in contrast maps natural inputs to traditional inputs such as mouse and keybaord. Darkly [11] performs *privacy transforms* on sensor inputs using computer vision algorithms (such as blurring, extracting edges, or picking out important features), but it has no support for rendering. In access visualization, sensor-access widgets [8] were proposed to reside within an application's display with an animation to show sensor data being collected by the ap-

plication. Previous work, however, does not handle the case of rendering.

8. CONCLUSION

We showed through surveys that the information revealed by SurroundWeb is acceptable. After a one-time setup procedure that scans a room for projectable surfaces in about a minute, our prototype can render immersive multi-display web rooms at greater than 30 frames per second with up to 25 screens and up to a 1440×720 display. Our abstractions are the first to enable least privilege for rendering in immersive experiences.

9. REFERENCES

[1] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. GÃijrses, F. Piessens, and B. Preneel. FPDetective: Dusting the web for fingerprinters. In *ACM CCS*, 2013.

[2] R. T. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. MacIntyre. Recent advances in augmented reality. *Computer Graphics and Applications*, 21(6):34–47, 2001.

[3] P. Baudisch, N. Good, and P. Stewart. Focus plus context screens: Combining display technology with visualization techniques. In *Proceedings of UIST '01*, 2001.

[4] J. Birnholtz, L. Reynolds, E. Luxenberg, C. Gutwin, and M. Mustafa. Awareness beyond the desktop: exploring attention and distraction with a projected peripheral-vision display. In *Proceedings of Graphics Interface 2010*, GI '10, pages 55–62, Toronto, Ont., Canada, Canada, 2010. Canadian Information Processing Society.

[5] L. D'Antoni, A. M. Dunn, S. Jana, T. Kohno, B. Livshits, D. Molnar, A. Moshchuk, E. Ofek, F. Roesner, S. Saponas, M. Veanes, and H. J. Wang. Operating system support for augmented reality applications. In *14th Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.

[6] B. M. et al. Argon mobile web browser, 2013. `https://research.cc.gatech.edu/kharma/`.

[7] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *Networked Systems Design and Implementation*, 2011.

[8] J. Howell and S. Schechter. What You See is What They Get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Web 2.0 Security and Privacy, IEEE*, 2010.

[9] E. Hutchings. Augmented reality lets shoppers see how new furniture would look at home, 2012. `http://www.psfk.com/2012/05/augmented-reality-furniture-app.html`.

[10] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *USENIX Security Symposium*, 2013.

[11] S. Jana, A. Narayanan, and V. Shmatikov. A scanner darkly: Privacy for perceptual applications. In *IEEE Symposium on Security and Privacy*, 2013.

[12] B. R. Jones, H. Benko, E. Ofek, and A. D. Wilson. IllumiRoom: peripheral projected illusions for interactive experiences. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 869–878, New York, NY, USA, 2013. ACM.

[13] S. K. Kane, D. Avrahami, J. O. Wobbrock, B. Harrison, A. D. Rea, M. Philipose, and A. LaMarca. Bonfire: a nomadic system for hybrid laptop-tabletop interaction. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, UIST '09, pages 129–138, New York, NY, USA, 2009. ACM.

[14] Layar. Layar catalogue, 2013. `http://www.layar.com/layers`.

[15] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 501–510, New York, NY, USA, 2012. ACM.

[16] I. Meta View. Meta developer kit, 2013. `http://www.meta-view.com/about`.

[17] Microsoft. Xbox SmartGlass, 2014. `http://www.xbox.com/en-US/SMARTGLASS`.

[18] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *10th IEEE International Symposium on Mixed and Augmented Reality*, 2011.

[19] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, S. Francisco, CA, May 2013.

[20] K. Parrish. Kinect for windows, ubi turns any surface into touch screen, 2013. `http://www.tomshardware.com/news/kinect-ubi-touch-screen-windows-8-projector,23887.html`.

[21] J. Rekimoto and M. Saitoh. Augmented surfaces: a spatially continuous work space for hybrid computing environments. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '99, pages 378–385, New York, NY, USA, 1999. ACM.

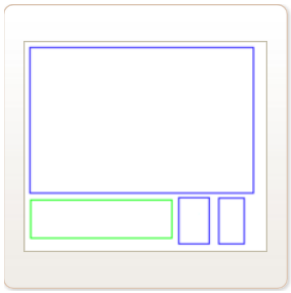
[22] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I still know what you visited last summer. In *IEEE Symposium on Security and Privacy*, 2011.

[23] J. Yang and D. Wigdor. Panelrama: enabling easy specification of cross-device web applications. In M. Jones, P. A. Palanque, A. Schmidt, and T. Grossman, editors, *CHI*, pages 2783–2792. ACM, 2014.

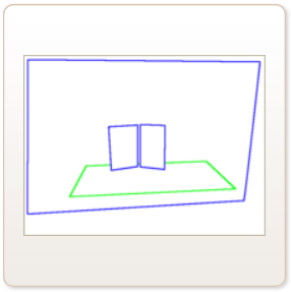
APPENDIX

As described above, we presented 50 survey-takers with three different web room descriptions: a "911 Assist" application that detects falls and calls 911, a "Dance Game" that asks users to mimic dance moves, and a "Media Player" that plays videos. We gave participants a description of the application, and asked: *What information would you be comfortable giving to this application? Choose as many as you like.* Participants chose from the visualizations of different data available to SurroundWeb shown in Figure 13.

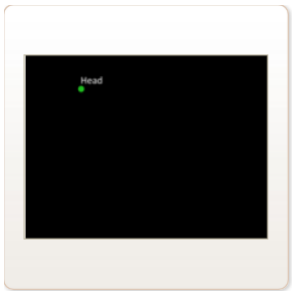
Participants could choose all, some, or none of the choices as information they would be comfortable revealing to the application. As we described above, we found that the information users were willing to reveal changed based on the application's description.



(a) Large flat surfaces: size (height and width) and orientation (standing up/laying down)

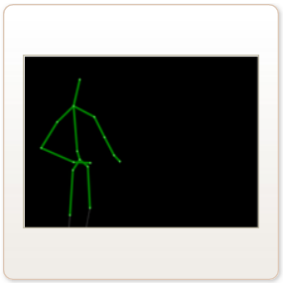


(b) Location of large flat surfaces

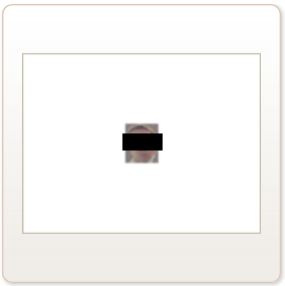


(c) Head position

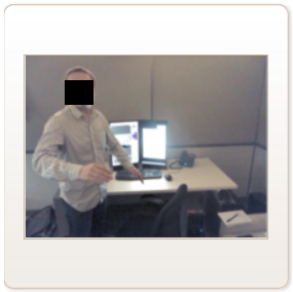
(d) Hand positions



(e) Body position



(f) Face



(g) Raw image

Figure 13: Examples of images used in the survey. To preserve the anonymity of the submission, we have anonymized the face of an author in these images.