SurroundWeb: Mitigating Privacy Concerns in a 3D Web Browser

MSR-TR-2014-147

John Vilk*, David Molnar†, Eyal Ofek, Chris Rossbach, Benjamin Livshits, Alexander Moshchuk‡, Ran Gal, and Helen Wang

*University of Massachusetts †Microsoft Research ‡Google Inc.

***Abstract*—Immersive experiences that mix digital and real-world objects are becoming reality, but they raise serious privacy concerns as they require real-time sensor input. These experiences are already present on smartphones and game consoles via Kinect, and will eventually emerge on the web platform. However, browsers do not expose the display interfaces needed to render immersive experiences. Previous security research focuses on controlling application access to sensor *input* alone, and do not deal with display interfaces. Recent research in human computer interactions has explored a variety of high-level rendering interfaces for immersive experiences, but these interfaces reveal sensitive data to the application. Bringing immersive experiences to the web requires a high-level interface that mitigates privacy concerns.**

This paper presents SurroundWeb, the first *3D web browser*, which provides the novel functionality of rendering web content onto a room while tackling many of the inherent privacy challenges. Following the *principle of least privilege*, we propose three abstractions for immersive rendering: 1) the *room skeleton* lets applications place content in response to the physical dimensions and locations of renderable surfaces in a room; 2) the *detection sandbox* lets applications declaratively place content near recognized objects in the room without revealing if the object is present; and 3) *satellite screens* let applications display content across devices registered with SurroundWeb. Through user surveys, we validate that these abstractions limit the amount of revealed information to an acceptable degree. In addition, we show that a wide range of immersive experiences can be implemented with acceptable performance.

I. Introduction

Immersive experiences mix digital and real-world objects to create rich computing experiences. These experiences can take many forms, and span a wide variety of sensor and display technologies. Games using the Kinect for Xbox can sense the user's body position, then show an avatar that mimics the user's movements. Smartphone translation applications, such as Word Lens, perform real-time translations in the video stream from phone cameras. A projector paired with a Kinect can enhance a military shooter game with a grenade that "bounces" out of the television screen and onto the floor [13]. Head mounted displays such as Google Glass, Epson Moverio, and Meta SpaceGlasses have dropped dramatically in price in the last five years and point the way toward affordable mainstream platforms that enable immersive experiences.

Toward a 3D web: In this paper, we set out to build a *3D web browser* that lets web applications project content onto physical surfaces in a room. While this is a novel and somewhat futuristic idea, it is possible to implement using modern sensors such as the Kinect. However, designing such a browser poses a plethora of privacy challenges. For example, current immersive experiences build custom rendering support on top of raw sensor data, which could contain sensitive data such as medical documents, credit card numbers, and faces of children should they be in view of the sensor. This paper is an exploration of how to reconcile 3D browser functionality with privacy concerns.

Tension between functionality and privacy: There is a clear *tension* between functionality and privacy: a 3D browser needs to expose high-level immersive rendering interfaces *without* revealing sensitive information to the application. Traditional web pages and applications are *sandboxed* within the browser; the browser interprets their HTML and CSS to determine page layout. Unfortunately, as prior experience with CSS demonstrates, these seemingly innocuous declarative forms of describing web content are fraught with privacy issues, such as subtle CSS-based attacks that use history sniffing [**?**] or more subtle ones that rely on scrollbars [16] and stateless fingerprinting [1].

In this work we discovered that a 3D browser needs to expose new rendering abstractions that let web applications project content into the room, while limiting the amount of revealed information to acceptable levels. Previous security research focuses on controlling application access to sensor data through filtering, access control, and sandboxing, but this research does not touch upon the complementary rendering interfaces that a 3D web browser requires [11, 12, 15]. Recent HCI research describes high-level interfaces for immersive rendering, but these approaches reveal sensitive data to the application [7, 13, 14].

SurroundWeb: In this paper, we present SurroundWeb, a 3D web browser that lets web applications display web content around a physical room in a manner that follows the *principle of least privilege*. We give applications access to three high-level interfaces that support a wide variety of existing and proposed immersive experiences while limiting the amount of information revealed to acceptable levels. First, the *Room Skeleton* exposes the location, size, and input capabilities of flat rectangular surfaces in the room, and a mechanism for associating web content with each. Applications can use this interface to make intelligent layout decisions according to the physical properties of the room. Second, the *Detection Sandbox* lets applications declaratively place content relative to objects in the room, without revealing the presence or locations of these objects. Third, *Satellite Screens* let applications display content across devices registered with SurroundWeb.

A. Contributions

This paper makes the following contributions:

- We implement SurroundWeb, a novel 3D web browser built on top of Internet Explorer. SurroundWeb enables web pages to display content across multiple surfaces in a room, across multiple phones and tablets, and to take natural user inputs.
- To resolve the tension between privacy and functionality, we propose three novel abstractions: *Room Skeleton*, *Detection Sandbox*, and *Satellite Screens*.
- We define the notions of *detection privacy*, *rendering privacy*, and *interaction privacy* as key properties for privacy in immersive applications, and show how SurroundWeb provides these properties.
- We evaluate the privacy and performance of SurroundWeb. To evaluate privacy, we survey people recruited by a professional survey provider and ask them about the information revealed by SurroundWeb compared to legacy approaches.[1] We discover that SurroundWeb has encouraging rendering performance, both from the Room Skeleton and the Detection Sandbox.

B. Paper Organization

The rest of this paper is organized as follows. In Section II, we discuss the threat model, and provide an overview of SurroundWeb. Section III describes our abstractions for immersive rendering. Section IV describes key privacy properties for immersive applications, and how SurroundWeb's abstractions provide these properties. Section V presents the implementation of SurroundWeb, and Section VI contains an evaluation of our design decisions and the performance of our prototype. Section VII discusses the limitations of our approach alongside future work, and Section VIII describes related work. Section IX concludes.

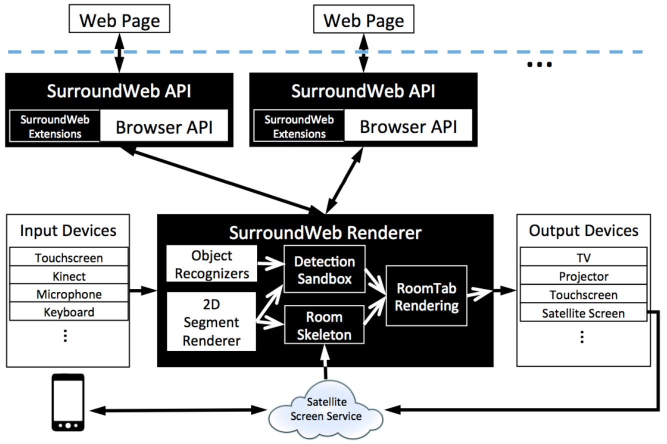


Fig. 1: Architectural diagram of SurroundWeb. Items below the dashed line are considered trusted, and items in white are off-the-shelf components.

II. Overview

In this work, we focus on the scenario where the computer, its operating system, the sensor hardware, and SurroundWeb itself are *trusted*, but SurroundWeb is executing an *untrusted* third-party web page or application. The SurroundWeb environment is identical to the browser sandbox used in regular browsers, except it has been augmented with SurroundWeb interfaces. As a result, the web application can only access sensor data and display devices indirectly through trusted APIs.

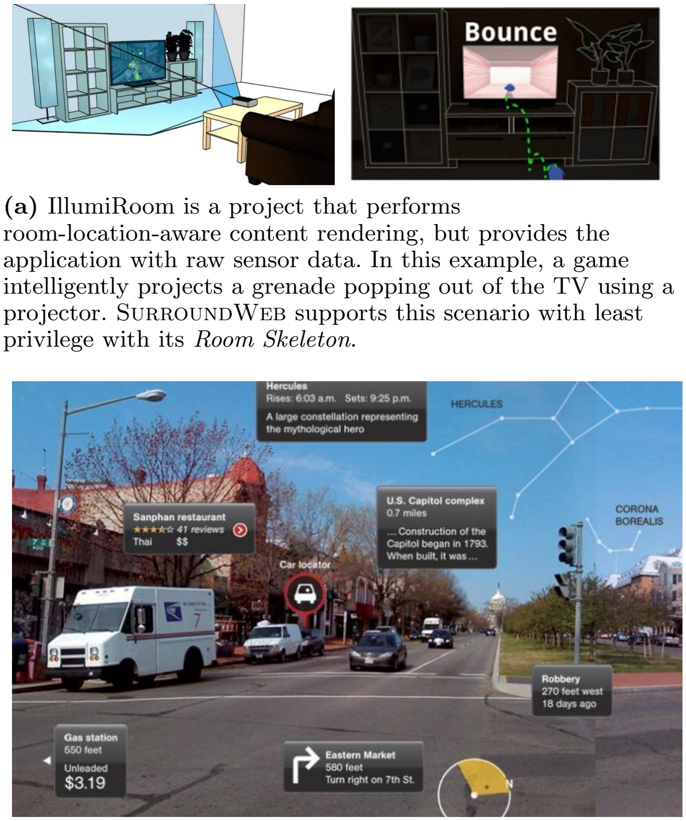
Figure 1 displays SurroundWeb's system diagram, with trusted components below the dashed line and off-the-shelf components in white. SurroundWeb consists of two main parts: the SurroundWeb API, which extends the browser API with immersive rendering support, and the SurroundWeb Renderer, which is responsible for interacting with sensors and display devices. Like in a regular web browser, web pages have no direct access to native resources (including sensors and displays), and are restricted to the interfaces furnished to them through JavaScript, HTML, and CSS.

Just like regular CSS, SurroundWeb supports both absolute and relative placement of web content within a room. For SurroundWeb, the notion of *placement* needs to be adapted to the context of 3D rendering. Figure 3 describes how SurroundWeb provides room-level positioning that is analogous to CSS positioning. Just like in the case of regular CSS, control over placement given to the web page or application opens the door to possible privacy violations. Liang *et al.* describe how CSS features like customizable scrollbars and media queries can be used to learn information about the user browser and, in some cases, to sniff information about browsing histories with the help of sophisticated timing attacks [16].

III. SurroundWeb Abstractions

We augment the browser with two privacy-preserving immersive rendering interfaces: the *Room Skeleton* for room-location-aware rendering tasks, and the *Detection*

[1] 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.



(a) IllumiRoom is a project that performs room-location-aware content rendering, but provides the application with raw sensor data. In this example, a game intelligently projects a grenade popping out of the TV using a projector. SurroundWeb supports this scenario with least privilege with its *Room Skeleton*.

(b) Layar and ARgon let applications render web content relative to objects, but they leak the presence of the objects to the application. SurroundWeb supports object-relative rendering without leaking the presence or location of the objects through the *Detection Sandbox*.

Fig. 2: Two examples of immersive experiences that reveal potentially sensitive information to applications.

Sandbox for object-relative rendering tasks. We also discuss *Satellite Screens*, which are an extension to the Room Skeleton that let applications display across devices. Section III describes all three of these abstractions in detail, and Section V describes their implemention in SurroundWeb. Figure 6 shows a visualization of the information that these interfaces reveal, and Figures 4 and 5 display example applications that are possible using these interfaces.

SurroundWeb provides web applications with two key rendering abstractions. The *Room Skeleton* lets applications place content relative to physical locations in the room. The *Detection Sandbox* lets applications place content relative to recognized objects in the room, without revealing the presence or location of those objects. We also extend the Room Skeleton with remote displays, which we call *Satellite Screens*.

A. *The Room Skeleton*

The *Room Skeleton* reveals the location and dimensions of all flat renderable surfaces in the room to the application as JavaScript objects, and exposes an API for displaying HTML content on each. We call these renderable surfaces *screens*. SurroundWeb handles rendering and projecting the content into the physical room. Using this interface, applications can perform room-location-aware rendering. Figure 6 visualizes the data that the Room Skeleton provides to the application.

In a static room, SurroundWeb can construct the Room Skeleton in a one-time setup phase, and can reuse it until the room changes. First, the setup process uses a depth camera to locate flat surfaces in the room that are large enough to host content. Next, it discovers all display devices that are available and determines which of them can show content on the detected surfaces. The process for doing this differs depending on the display device. Head-mounted displays can support rendering content on arbitrary surfaces, but projectors are limited to the area they project onto. For projectors, monitors, and televisions, SurroundWeb maps device display coordinates to room coordinates; this process can be automated using a video camera. Finally, the setup process 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 [23].

At runtime, the application receives *only* the set of screens in the room. Each screen has a resolution, a measure of pixel density (points-per-inch), a relative location to other screens, and a list of input events that can be accepted by the screen. For example, these input events include “none,” “mouse,” or “touch.” Applications can use this information to dynamically adapt how it presents its content in response to the capabilities of the room.

The application can associate HTML content with a screen to display content in the room. SurroundWeb uses a standard off-the-shelf browser renderer to render the content, and then displays it in the room according to information discovered at setup time. If the screen supports input events, the application can use existing standard web APIs to register event listeners on the HTML content.

Example 1 (SurroundPoint) SurroundPoint is a presentation application pictured in Figure 7. This application uses the Room Skeleton. The presentation 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 6. 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 additional content. □

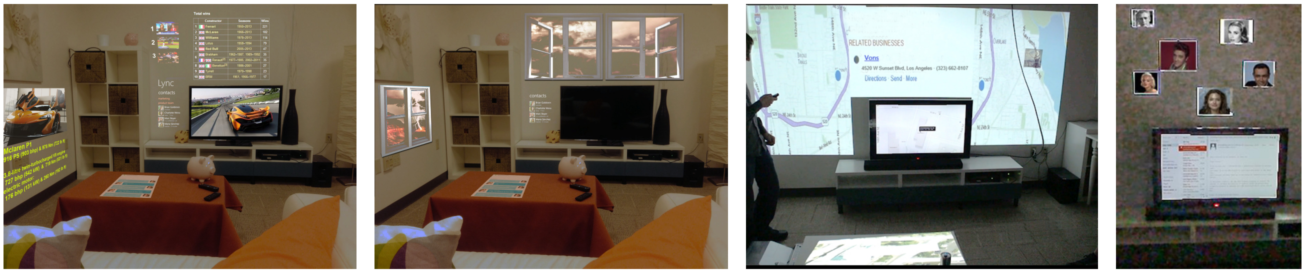
CSS		SurroundWeb	
Placement	**Example**	**Placement**	**Example**

Absolute	`#picture {position: absolute; top:400px;}`	Room-location-aware	`<segment screen="4">`
			`<img src="picture.jpg" />`
			`</segment>`

Content is placed at an absolute location, relative to the content's parent element. The example places a picture 400 pixels below its parent element.		Content is placed in an absolute location in the room. The example places `picture.jpg` on screen 4, which corresponds to a surface in the room.	

Relative	`#picture {position: relative; right: 50px;}`	Object-relative	`#picture {left-of: "chair";}`
Content is placed at a location relative to its normal position in the document. Here, a picture is shifted 50 pixels to the right.		Content is placed relative to a recognized object in the room. Here, a picture is placed to the left of a chair.	

Fig. 3: SurroundWeb supports positioning web content in the room in a way that is analogous to CSS positioning.



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

Fig. 4: Four web applications enabled by SurroundWeb, shown with multiple projectors and an HDTV.

B. 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 makes possible *object-relative rendering*, where an application specifies that content should be rendered by an object in the room. However, object detection is a privacy challenge because the presence of objects can reveal sensitive information about a user's life. For example, an application can detect if the user is holding a bottle of medicine, then show instructions for safely using the medicine as an "annotation" to the bottle. This creates a tension between privacy and functionality.

Our *Detection Sandbox* provides least privilege for object-relative rendering. All object recognition code runs as part of trusted code in SurroundWeb. Applications *register* HTML content up front with the Detection Sandbox using a system of *rendering constraints* that can reference physical objects. In Section V we show how SurroundWeb exposes these to web pages via Cascading Style Sheets, and discuss the FLARE constraint solver [25].

After the application loads, the Detection Sandbox immediately renders all registered content, regardless of whether or not it will be shown in the room. In doing so, we prevent the application from using tracking pixels to determine the presence of an object [26]. Then, SurroundWeb checks registered content against a list of objects detected. If there is a match, the Detection Sandbox solves the constraints to determine a rendering location, and places the content in the room.

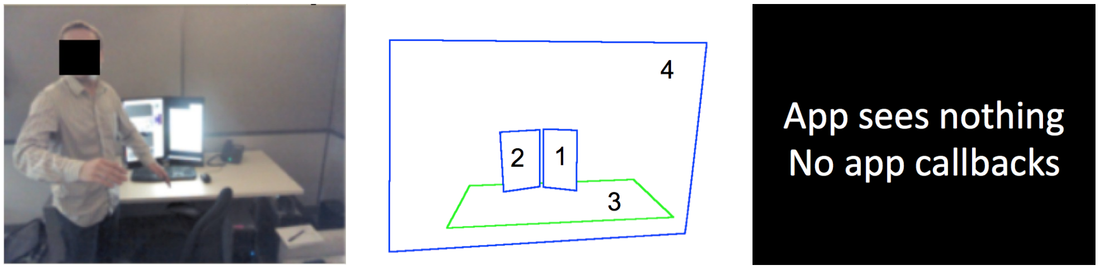
Constraint solving occurs asynchronously on a separate thread from the web application, preventing the application from directly timing constraint solving to infer whether or not an object is present. In addition, to prevent the application from determining the presence of an object, the Detection Sandbox suppresses user input events to the registered content and does not notify the application if content is displayed in the room.

Example 2 (Object-contextual ads) An application can register an ad to display if an energy drink can is present. The presence of the object is sensitive because excessive energy drink consumption may be correlated with poor health. When the can is placed in view of a camera in the room, the Detection Sandbox detects that the can is present and also detects that the application has registered content to display if an energy drink is present. The Detection Sandbox then displays the content, which in this case is an ad encoraging the user to drink tea instead. The application, however, never learns if the content displays or if the can is present.

With our Detection Sandbox, we cannot support users clicking on ads, because the input event would leak the presence of the object to the application. It may be possible to extend the Detection Sandbox with previous work on privacy-preserving ad serving which couples ad display with anonymizing routers such as Tor. With this approach, the network server would not be able to identify the user from the anonymized network connection. We discuss this idea in more detail in Section VII. □

Application	Requires	Description
SurroundPoint	Room Skeleton	Each screen in the room becomes a rendering surface for a room-wide presentation (see Figure 7).
Car Racing News Site	Room Skeleton	Live video feed displays on a central monitor, with racing results projected around it (see Figure 4a).
Virtual Windows	Room Skeleton	”Virtual windows” render on surfaces around the room that display scenery from distant places (see Figure 4b).
Road Maps	Room Skeleton	Active map area displays on a central screen, with surrounding area projected around it (see Figure 4c).
Projected Awareness IM [3]	Room Skeleton	Instant messages display on a central screen, with frequent contacts projected above (see Figure 4d). This is an example of Focus+Context from UI research [2].
Karaoke	Room Skeleton	Song lyrics appear above a central screen, with music videos playing around the room (see Figure 10).
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.
SmartGlass [20]	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 an additional screen.
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.

Fig. 5: Web applications and immersive experiences that are possible with SurroundWeb.



Raw Data **Room Skeleton** **Detection Sandbox**

Fig. 6: Our *Room Skeleton* and *Detection Sandbox* abstractions reveal significantly less information than raw sensor data. Here, we visualize the information that these interfaces provide to the application. The Detection Sandbox reveals *nothing*, as the application provides content to be rendered near objects, but is not informed if the content is rendered or if the object is present. We have censored an author's face in the raw feed to protect anonymity.

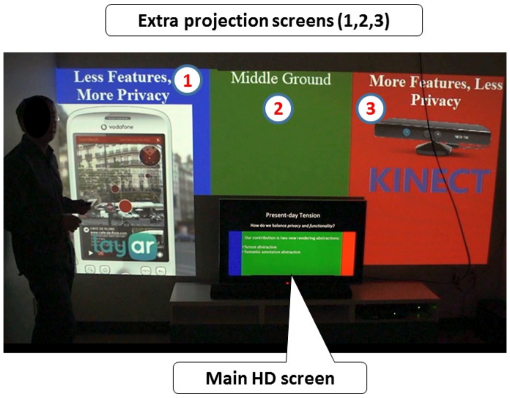

Extra projection screens (1,2,3)
Less Features, More Privacy
1
Middle Ground
2
More Features, Less Privacy
3
KINECT
Main HD screen

Fig. 7: The SurroundPoint presentation application uses the *Room Skeleton* to render an immersive presentation with least privilege. This image displays SurroundPoint running in our SurroundWeb prototype.

C. Satellite Screens

In our discussion of the Room Skeleton above, we talked about exposing fixed, flat surfaces present in a room as virtual *screens* for content. Today, however, many people have personal mobile devices, such as smartphones or tablets. To accomodate these devices, we extend the Room Skeleton with remote displays that we call *Satellite Screens*. By navigating to a URL of a cloud service, phones, tablets, or anything with a web browser can register a display with the Room Skeleton. JavaScript running on the cloud service discovers the device's screen size and input capabilities, then communicates these to the Room Skeleton. The Room Skeleton surfaces each device to the application as a new Screen, which can be rendered to like any other surface in the room.

Example 3 (Private displays for poker) Satellite Screens enable applications that need *private displays*. For example, a poker application 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, 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 applications to build multi-player experiences without needing to explicitly tackle creating a distributed system, as all Satellite Screens are accessible to a single application instance in the Room Skeleton. □

IV. Privacy Properties

Our abstractions provide 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 our abstractions. We then discuss important limitations and how they may be addressed.

A. Detection Privacy

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

For example, an e-commerce application 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, an application 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 application has access to a high-level API for detecting objects and their properties, instead of raw video and depth streams [11]. 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.*

The Detection Sandbox provides detection privacy. Our threat model for detection privacy is that applications are allowed to register arbitrary content in the Detection Sandbox. This registration takes the form of *rendering constraints* specified relative to a physical object's position, which tell the Detection Sandbox where to render the registered content. The rendering process is handled by trusted code in SurroundWeb, which renders content regardless of the object's presence. SurroundWeb also blocks input events to this content. As a result, the application never learns whether an object is present or not, no matter what is placed in the Detection Sandbox.

Our approach places limitations on applications, both fundamental to the concept of the Detection Sandbox and as artifacts of our current approach. We discuss these in detail in Section VII.

B. Rendering Privacy

Rendering privacy means that an application can render content into a room, but it learns no information about the room beyond an explicitly specified set of properties needed to render. Without rendering privacy, 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*, which we accomplish through the Room Skeleton. Our threat model for rendering privacy is that 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 application, while with detection privacy we leak no information about the presence or absence of objects. User surveys in Section VI show that revealing this information is acceptable to users.

C. Interaction Privacy

Interaction privacy means that an application 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, an application 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.

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

Second, we map natural user gestures to existing UI events, such as mouse events. We perform this remapping to enable interactions with applications even if those applications have not been specifically enhanced for natural gesture interaction. These 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 UI events does limit applications. In Section VII we discuss how this could be relaxed while maintaining our privacy properties.

V. Implementation

Our prototype 3D Web Browser, SurroundWeb, is written in C# on Windows. For rendering HTML, JavaScript, and CSS, SurroundWeb embeds a native `WebBrowser` control that provides an instance of the Internet Explorer rendering and JavaScript engines. SurroundWeb then exposes a set of objects through JavaScript to web pages rendered in the `WebBrowser`; these objects provide web pages with access to the room skeleton and detection sandbox. After the `WebBrowser` control finishes rendering a web page, the SurroundWeb renderer uses a special interface of the control to extract rendered PNG files for each element of the resulting web page.

When necessary, SurroundWeb uses the FLARE library for 3D constraint solving. SurroundWeb then places these elements in the room according to the Room Skeleton and Detection Sandbox requests registered by the web page. SurroundWeb also links against the Kinect For Windows SDK, enabling it to recognize gestures and map them into web page events.Figure 1 displays an architectural diagram of SurroundWeb, with the parts we implemented in black. Items below the dashed line form the *trusted core* of SurroundWeb, while items above the dashed line are untrusted.

In the rest of the section, we first describe in more detail the *core capabilities* of our prototype. We then detail how these capabilities are exposed to web applications through HTML, CSS, and JavaScript.

A. *Core Capabilities*

Our SurroundWeb prototype works with instrumented rooms, each of which contains multiple projectors and Kinect sensors. Similarly to IllumiRoom, SurroundWeb uses the Kinect sensors to scan the geometry of the room, then uses the projectors to display application content at appropriate places in the room. In addition, unlike IllumiRoom, SurroundWeb takes advantage of tablets, phones, and other devices in the room to show additional personal content or controls. As a result, we implement the following components:

Room Skeleton building: 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 [21], 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.

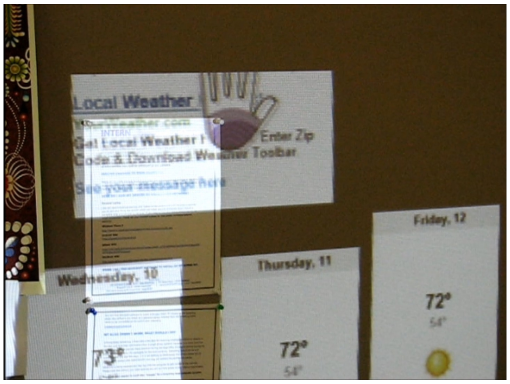


Fig. 8: SurroundWeb maps natural gestures to mouse events. Here, the user uses a standard "push" gesture to tell SurroundWeb to inject a click event into the web application.

We use the the Microsoft Kinect SDK to detect user position and gestures, including push and swipe gestures [19]. Figure 8 shows a photograph of SurroundWeb detecting that the user is performing a pushing gesture over a particular part of the wall. 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.

B. *HTML Extensions*

The trusted core of SurroundWeb embeds a native `WebBrowser` control, which allows the user to specify a URL containing HTML, CSS, and JavaScript. As a web page renders, it has access to several extensions of HTML that give it access to SurroundWeb core capabilities. We refer to a web page that uses these capabilities as a *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.

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 V-C) or JavaScript (Section V-D).

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.

Trusted user interface: In addition to extending HTML, the SurroundWeb prototype presents an interface to the user. 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.

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 content 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.

Content rendering: The architecture of SurroundWeb's renderer, displayed in Figure 1, resembles a conventional web browser. The block marked "SurroundWeb API" encapsulates existing browser HTML, CSS, and JavaScript functionality, and our extensions to these interfaces. 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"). In our prototype, this is the Trident rendering engine inside the Internet Explorer `WebBrowser` control.

When the user navigates to a web application, our prototype first renders the entire web application inisde the `WebBrowser` control. We then extract individual bitmaps for each `segment` tag using an API of the control. 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 the FLARE constraint solver [25] 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. This allows SurroundWeb to support animated content.

C. CSS Extensions

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

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. As part of the rendering process, the trusted SurroundWeb renderer compiles these constraints into the constraint language of the FLARE constraint solver [25], and invokes the solver to determine a rendering location. We use FLARE because it is specifically designed to solve layout constraints for objects in an arbitrary three-dimensional space as quickly as possible, making it a good

Property	Description
`getAll()`	(Static) Returns an array of all of 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.

Fig. 9: Properties of each screen object.

fit for the Detection Sandbox. As FLARE only operates on three-dimensional objects, we represent each segment in FLARE as a rectangular thin object whose texture is the rendered bitmap of the segment. If FLARE is able to solve the rendering constraints, then the solution will be a valid rendering location in the room, which SurroundWeb will use to render the supplied content.

D. 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 9 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 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.

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.

E. 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 $\langle\texttt{protocol}, \texttt{domainname}, \texttt{port}\rangle$.

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 interfaces, 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 SurroundWeb’s interfaces. 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 interfaces from within a frame. If a web room denies an embedded origin access to these interfaces, 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 interfaces, then the `iframe` will be able to render content to that web room’s room tab.

Example 4 (Karaoke Application Walkthrough) To illustrate how a web room can be developed using SurroundWeb, we will walk through a sample Karaoke application, shown in Figure 10.

This web room renders karaoke lyrics above a central screen, with a video on the central screen and pictures

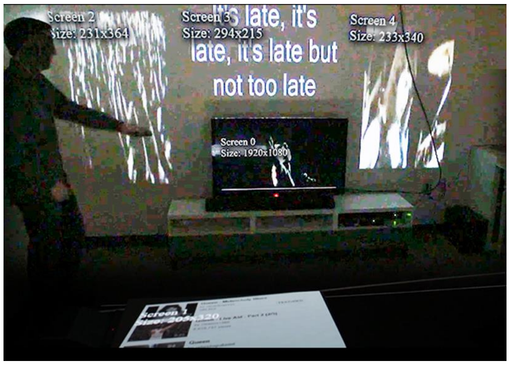


Fig. 10: The Karaoke application uses the Room Skeleton to make room layout decisions. The application dynamically decides to show a video on the high resolution TV. Lyrics and photos go on projected screens, and related videos are projected on the table.

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. □

VI. Evaluation

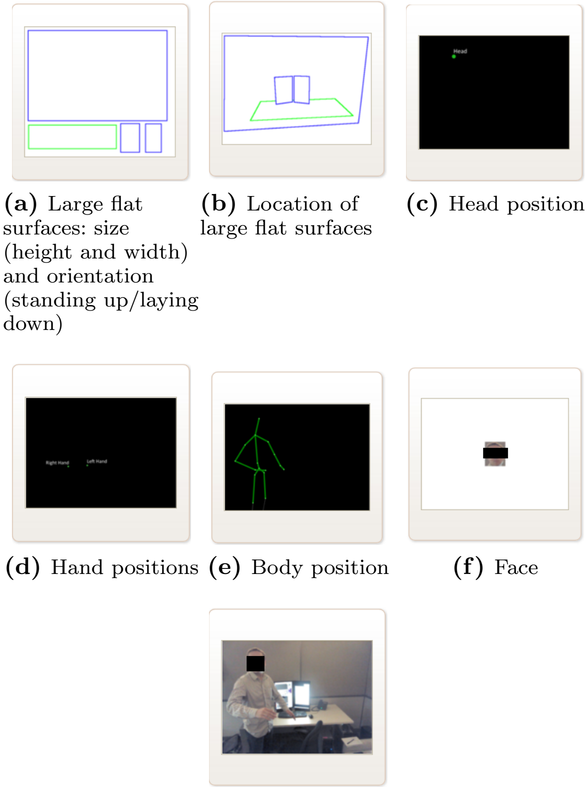
The focus of this evaluation is two-fold. We want to evaluate whether our abstractions deliver adequate privacy guarantees, which we do via a user study, and whether the performance of SurroundWeb is acceptable.

A. *Privacy*

To evaluate the privacy of our abstractions, we conducted two surveys of random US Internet users using the Instant.ly survey service [27]. Below are the research questions we were trying to address via surveys and the answers we discovered experimentally.

- **RQ1**: Is the information revealed by specific abstractions considered less sensitive than raw data? (Yes) We asked this question to evaluate whether our abstractions in fact mitigate privacy concerns.
- **RQ2**: Do user privacy preferences change depending on the application asking for data? (Yes) We asked this question to evaluate whether follow-on work should support different abstractions or different permissions for different web pages.
- **RQ3**: Is there a difference in user perceived sensitivity between giving the relative locations of flat surfaces vs. just stating the dimensions of flat surfaces? (No) We asked this question because we were not sure whether to include relative locations in the Room Skeleton; the answer justified including them because it increased functionality without changing perceived sensitivity of data given to the application.

We did not ask any questions that required participants to supply us with personally identifiable information. Prior to running our surveys, we reviewed our questions, the data



(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

Fig. 11: Survey participants chose which data should be given to three hypothetical applications by choosing zero or more of the above options. To preserve the anonymity of the submission, we have anonymized the face of an author in these images.

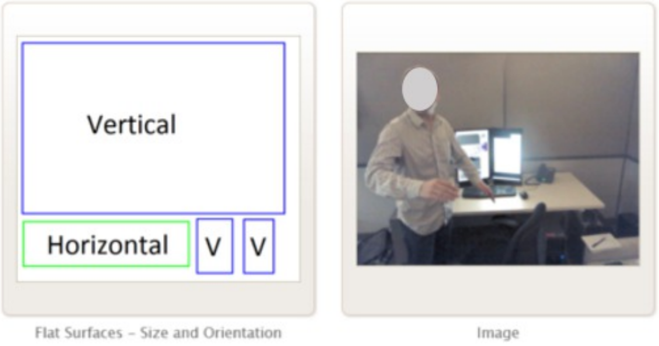


Fig. 12: Excerpt from our Room Skeleton sensitivity survey. We asked users which image contains more information that they consider private; 87.5% believe the raw image contains more private information. We have modified the image to remove an author's face.

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.

Room Skeleton sensitivity: The first survey measures user privacy attitudes toward the information that the Room Skeleton reveals to applications. The survey began with a briefing, summarizing the capabilities commonly available to immersive experiences. Next, we showed users two pictures: a color picture of a room, and a visualization of the data exposed from that room in the Room Skeleton. Figure 12 displays these pictures. The survey asked the following question: *Only consider the information explictly displayed in the two images. Which image contains* **more information** *that you consider "private"?* We also asked respondents to justify their answer in a free-text field.

Finding 1: Out of 50 respondents, 78% claimed that they felt that the raw data was more sensitive than the information that the Room Skeleton exposes to web pages. However, we noticed from the written justifications that some people did not understand the survey and answered randomly. Others mistakenly chose the information they felt was *most* sensitive. After filtering out the random respondents and accounting for those who misinterpreted the question, 87.5% claimed that they felt the raw data was more sensitive than the information in the Room Skeleton. This supports our choice of data to reveal in the Room Skeleton.

Application-specific surveys: Our second survey explores a broader set of possible data that could be released to applications in the context of different application scenarios. We presented 50 survey-takers with three different application 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 application. Participants chose from the visualizations of different data available to SurroundWeb shown in Figure 11. Participants could choose all, some, or none of the choices as information they would be comfortable revealing to the application. Then, we asked participants to justify their answer in a free-text field. From this survey, we have the following findings:

Finding 2: Users have different privacy preferences for different applications. 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 were more skeptical of releasing information to the dance game; one user stated, "A dance game would not need more information than the general outline or placements of the body". In some cases respondents thought creatively about how an application could use additional information; 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 supports fine-grained permission levels for individual applications, which we view as future work.

Finding 3: Users did not distinguish between the sensitivity of screen sizes and room geometry. *Screen sizes* includes the number, orientation, and size of screens, but does not include their positions in the room. *Room geometry* refers to the information revealed through our Room Skeleton abstraction. Before conducting our surveys, we hypothesized (RQ3) that users would find room geometry to be more sensitive than screen sizes. In fact, our data does not confirm this hypothesis. In response to this finding, we decided that the Room Skeleton would expose the room's geometry.

B. 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 13 shows the results of scanning three fairly typical rooms we chose. No room took longer than 70 seconds to scan, which is reasonable for a one-time setup cost.

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

Fig. 13: Scanning times, in seconds, and results for three representative rooms.

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. Note that constraint solving occurs on its own thread, so applications have no way of measuring this information. 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 14 shows how solving time increases for this scenario as the number of registered content 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.

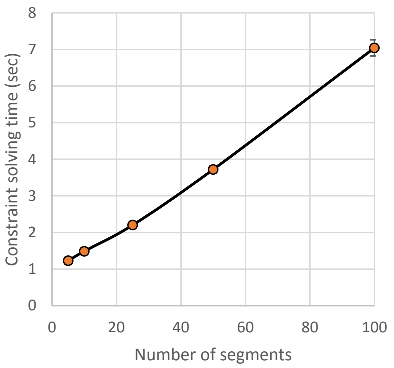


Fig. 14: Solver performance as the number of segments registered with the Detection Sandbox increases. The error bars indicate 95% confidence intervals.

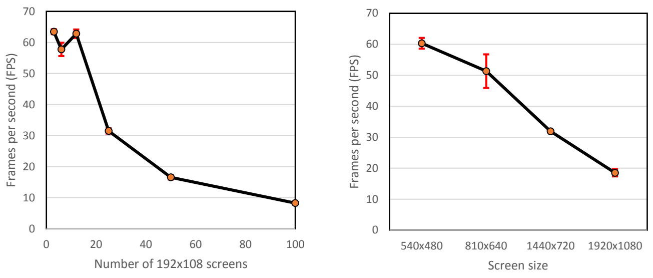


Fig. 15: On the left, maximum rendering frame rate as the number of same-size screens increases. On the right, the maximum rendering frame rate of a single screen as its size increases. Error bars indicate 95% confidence intervals.

- Next, we considered the scenario where the web page uses the solver for a more complicated layout. We tested a scene with 12 detected objects and 8 content 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. Note that since the constraint solver operates on its own thread, the constraint solver does not delay the display of content rendered via the Room Skeleton.

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 15 displays the results. In the left configuration, the benchmark measures the frame rate as it increases the number of 192×108 screens. To stress the novel pieces of the SurroundWeb rendering pipeline, we must render across multiple screens. For a single screen, SurroundWeb simply uses the embedded Trident rendering engine from Internet Explorer.

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.

VII. Limitations and Future Work

Detection, Rendering, and Interaction privacy presented in Section IV are variations on a theme: enabling *least privilege* for immersive room experiences. In each case, we provide an abstraction that supports immersive experiences while revealing minimal information. We discuss limitations to the privacy properties provided by our abstractions and limitations to our abstraction implementations in SurroundWeb, along with other future work.

Social engineering: 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 our abstractions, but they also could be carried out by existing applications.

Browser fingerprinting: *Browser fingerprinting* allows a web page to uniquely identify a user based on the instance of her browser. Our interfaces 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, 22]. 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 [10]. 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, 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 [16, 30].

Because SurroundWeb extends the web platform, side channels that exist on the current web are still present in SurroundWeb. For the Detection Sandbox, we avoid these side channels by having a separate trusted renderer place object-relative content registered with applications. Application interactions with the Detection Sandbox are strictly one-way; the application hands content and constraints to the Detection Sandbox, and has no way to query the content to determine its room location, or determine if the content is displayed in the room at all. The Detection Sandbox renders content immediately, regardless of if it appears in the room or not, to prevent the application from using tracking pixels or Web Bugs to determine if content is displayed [26]. Furthermore, applications receive *no* input from content placed in the Detection Sandbox. In this way we isolate the content in the Detection Sandbox and mitigate potential side channels that could potentially reveal to the application whether the content is present and, if so, where.

There may also be new side channels that reveal sensitive information about the room. For example, although constraint solving occurs on its own thread, 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 the 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.

Supporting clickable object-relative ads: 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 [9]. We could use 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.

VIII. Related Work

SurroundWeb bridges the gap between emerging research in HCI on augmented reality rendering abstractions and research in security on preserving user privacy in augmented reality settings. Table 16 summarizes the capabilities that these research systems surface compared with SurroundWeb.

Regulating Access to Sensor Data: Existing research in security focuses explicitly on regulating sensor data provided to untrusted applications; none provide any capabilities for rendering content in an augmented reality environment. This research is orthogonal to SurroundWeb, and could be applied to expose rich sensor data to 3D web pages in a privacy-preserving manner.

Darkly [12] performs *privacy transforms* on sensor inputs using computer vision algorithms (such as blurring, extracting edges, or picking out important features) to prevent applications from extracting private information from sensor data. For sensitive information, Darkly provides the application with opaque references that it can pass to trusted library routines, and allows the application to apply basic numerical calculations to the data. Darkly also provides a trusted GUI component, but applications are unable to introspect on the GUI's contents, which is an important feature on the web. In addition, Darkly does not view the *presence* of input events on the trusted GUI as sensitive, whereas SurroundWeb must necessarily block input events to content registered with the Detection Sandbox to prevent the application from learning which objects are in the room.

As a separate approach to the same problem, previous work introduced the *recognizer* abstraction to provide applications with the appropriate permissions access to higher-level data constructed from sensor data by trusted code in the OS [5, 11]. SurroundWeb itself uses a trusted recognizer to map natural user input to existing web events. πBox takes a third approach to this problem, and sandboxes code that interacts with sensitive sensor information [15]. SurroundWeb's Detection Sandbox, which uses a constraint-based layout system, could be extended to support full-fledged sandboxed layout decisions that make use of sensitive sensor information.

High-level Rendering Abstractions: On the other hand, recent research in HCI has created useful abstractions for rendering content in an augmented reality environment, but they are either limited in functionality or reveal potentially sensitive information to the application. Many of these abstractions can be reimplemented completely or partially in SurroundWeb to preserve user privacy. Illumiroom uses projectors combined with a standard TV screen to create gaming experiences that "spill out" of the TV and into the room, but to do this the application has access to the raw sensor feed [13]. SurroundWeb can create a similar experience using its Room Skeleton abstraction.

Panelrama lets web pages intelligently split their content across local devices using a "Panel" abstraction, but it makes layout decisions using only static device properties, such as screen size, rather than data available through sensors, such as room location [31]. In addition, because Panelrama does not depend on support in the web browser, application developers must explicitly manage state synchronization across multiple communicating pages, each with their own DOM tree. In contrast, with SurroundWeb the developer writes a single page with all elements in the same DOM tree. Panelrama could be reimplemented on top of SurroundWeb's Room Skeleton and Satellite Screens, and extended to incorporate room location into its layout algorithm.

Layar uses tags in printed material and GPS coordinates to superimpose interactive content relative to a tagged object or location in a mobile phone's camera feed [14]. Unlike with SurroundWeb's Detection Sandbox, accessing Layar content reveals to the application that the user either possesses the tagged object, or that the user is near a tagged physical location. The Argon mobile browser extends HTML to let web applications place content at particular GPS locations, which the user can view through her mobile phone [7]. Like with Layar, viewing and loading this content reveals to the application that the user is near a tagged GPS location.

Access Control: Recent work on world-driven access control restricts sensor input to applications in response to the environment, e.g. it can be used to disable access to the camera when in a bathroom [24]. Mazurek et al. surveyed 33 users about how they think about controlling access to data provided by a variety of devices, and discovered that many user's mental models of access control are incorrect [18]. Vaniea et al. performed an experiment to determine how users notice and fix access-control permission errors depending on where the access-control policy is spatially located on a website [28].

Browser Privacy: Previous research in web security has unearthed a variety of privacy issues in existing web browsers. While a great deal of focus has been on stateful user tracking via cookies and policies such as DoNotTrack, there are some inherent additional issues with browser design that may compromise privacy. Most notably, a variety of methods exist for *history sniffing*, which allow web sites to learn about users' visits to other sites. Weinberg *et al.* describe an interactive side channel that leak whether a URL has been visited [30]. Liang *et al.* discovered a novel CSS timing attack for sniffing browser history [16].

Other security research discusses browser fingerprinting, which allows a web page to uniquely identify a user based on the instance of the browser. The work of Mayer [17] and Eckersley [6] presents large-scale studies that show the possibility of effective stateless web tracking via only the attributes of a user's browsing environment. These studies prompted some follow-up efforts [8, 29] to build better fingerprinting libraries. Yen *et al.* [32] performed a fingerprinting study by analyzing month-long logs of Bing and Hotmail and showed that the combination of the User-agent HTTP header with a client's IP address were enough

Field	Related work	Application Sensor Input				Rendering Support		
		Unrestricted	Filtered	Sandboxed	Object Presence	Multi-device	Room-location-aware	Object-relative
Security	SurroundWeb		✓	✓		✓	✓	✓
	Darkly [12]		✓					
	OS-level Recognizers [11]		✓					
	World-driven ACL [24]		✓					
	πBox [15]			✓				
HCI	Illumiroom [13]	✓					✓	
	Panelrama [31]					✓		
	Layar [14]				✓			✓
	Argon [7]				✓			✓

Fig. 16: Summary of related work in security and HCI. Emerging research in security focuses solely on restricting application access to sensitive sensor data, while HCI research generally focuses on application rendering capabilities without considering their privacy implications. SurroundWeb bridges the gap between these two bodies of research.

to track approximately 80% of the hosts in their dataset.

Chapman and Evans show how side channels in web applications can be detected with a black box approach; future work could apply this to looking for detection sandbox side channels [4].

IX. Conclusion

This paper presents SurroundWeb, the first 3D web browser that enables web applications to project web content into a room in a manner that follows the principle of least privilege. We described two rendering abstractions for the web platform that support a wide variety of existing and proposed immersive experiences while limiting the amount of information revealed to acceptable levels. The *Room Skeleton* lets applications place web content on renderable surfaces in the room. The *Detection Sandbox* lets applications declaratively place web content relative to objects in the room without revealing any object's location or presence. We defined *detection privacy*, *rendering privacy*, and *interaction privacy* as key properties for privacy in immersive applications, and demonstrate that our interfaces provide these properties. Finally, we validated that our two abstractions reveal an acceptable amount of information to applications through user surveys, and demonstrated that our prototype implementation has acceptable performance.

References

[1] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for Fingerprinters. In *ACM Conference on Computer & Communications Security*, 2013.

[2] P. Baudisch, N. Good, and P. Stewart. Focus plus context screens: Combining display technology with visualization techniques. In *ACM Symposium on User Interface Software and Technology*, 2001.

[3] 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 *Graphics Interface*, 2010.

[4] P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 263–274, New York, NY, USA, 2011. ACM.

[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 *Workshop on Hot Topics in Operating Systems*, 2013.

[6] P. Eckersley. How Unique Is Your Browser? In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, pages 1–17, 2010.

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

[8] E. Flood and J. Karlsson. Browser fingerprinting, 2012.

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

[10] R. Hansen and J. Grossman. Clickjacking. http://www.sectheory.com/clickjacking.htm.

[11] 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.

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

[13] B. R. Jones, H. Benko, E. Ofek, and A. D. Wilson. IllumiRoom: peripheral projected illusions for interactive experiences. In *Conference on Human Factors in Computing Systems*, 2013.

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

[15] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. Box: A platform for privacy-preserving apps. In *Symposium on Networked Systems Design and Implementation*, 2013.

[16] B. Liang, W. You, L. Liu, W. Shi, and M. Heiderich. Scriptless timing attacks on web browser privacy. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[17] J. R. Mayer. Any person... a pamphleteer. Senior Thesis, Stanford University, 2009.

[18] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access control for home data sharing: Attitudes, needs and practices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010.

[19] Microsoft Corporation. Kinect for Windows SDK, 2013. http://www.microsoft.com/en-us/kinectforwindows/.

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

[21] 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 *IEEE International Symposium on Mixed and Augmented Reality*, 2011.

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

[23] 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.

[24] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, and H. J. Wang. World-driven access control. In *ACM Conference on Computer and Communications Security*, 2014.

[25] L. Shapira, R. Gal, E. Ofek, and P. Kohli. FLARE: Fast Layout for Augmented Reality Applications. In *IEEE International Symposium on Mixed and Augmented Reality*, September 2014.

[26] R. M. Smith. The web bug faq, 1999. https://w2.eff.org/Privacy/Marketing/web_bug.html.

[27] uSample. Instant.ly survey creator, 2013. http://instant.ly.

[28] K. Vaniea, L. Bauer, L. F. Cranor, and M. K. Reiter. Out of sight, out of mind: Effects of displaying access-control information near the item it controls. In *IEEE Conference on Privacy, Security and Trust (PST)*, 2012.

[29] V. Vasilyev. fingerprintjs library. https://github.com/Valve/fingerprintjs, 2013.

[30] 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.

[31] J. Yang and D. Wigdor. Panelrama: enabling easy specification of cross-device web applications. In *ACM Conference on Human Factors in Computing Systems*, 2014.

[32] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the Web: Privacy and security implications. In *Proceddings of the Network and Distributed System Security Symposium (NDSS)*, 2012.