2015 IEEE Symposium on Security and Privacy

SurroundWeb: Mitigating Privacy Concerns in a 3D Web Browser

John Vilk
University of Massachusetts, Amherst

David Molnar, Benjamin Livshits, Eyal Ofek
Microsoft Research

Chris Rossbach
VMWare Research

Alexander Moshchuk
Google

Helen J. Wang, Ran Gal
Microsoft Research

***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.

***Index Terms*—augmented reality; JavaScript; web browser; projection mapping**

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 pages and 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. In addition to this paper, we produced an informative video that we encourage readers to watch to familiarize themselves with this scenario: `http://research.microsoft.com/apps/video/?id=212669`.

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 surprising CSS-based attacks that use history sniffing [30] and even 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

DOI 10.1109/SP.2015.33

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 lets web pages display content across multiple surfaces in a room, run 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 users recruited through a professional survey provider and ask them about the information revealed by SurroundWeb compared to legacy approaches.[1] We discover that SurroundWeb has acceptable performance through benchmarking the rendering and layout speed of the room skeleton and 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 an assessment of the performance of our prototype. Section VII discusses the limitations of our approach alongside future work, and Section VIII describes related work. Finally, Section IX concludes.

[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.

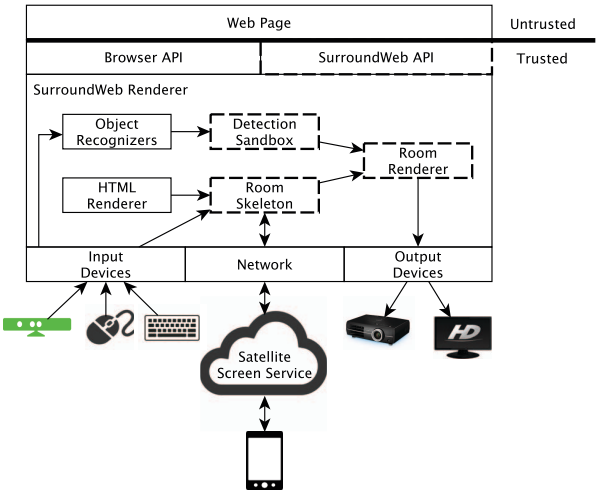


Fig. 1: Architectural diagram of SurroundWeb. Items below the thick line are considered trusted, and items with solid borders are off-the-shelf components.

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 that we describe in this paper.

Figure 1 displays SurroundWeb's system diagram, with trusted components below the thick line and off-the-shelf components with solid borders. 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*

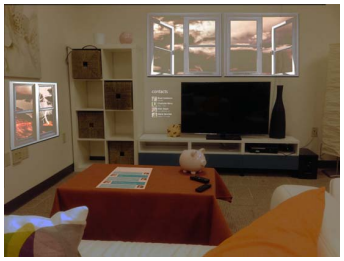
CSS		SurroundWeb	
Placement	**Example**	**Placement**	**Example**

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. 2: 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. 3: Four web applications enabled by SurroundWeb, shown with multiple projectors and an HDTV.

needs to be adapted to the context of 3D rendering. Figure 2 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's 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 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. We introduce these abstractions with the help of example applications, described in Figures 3 and 4, which clarify their intent and utility.

A. The Room Skeleton

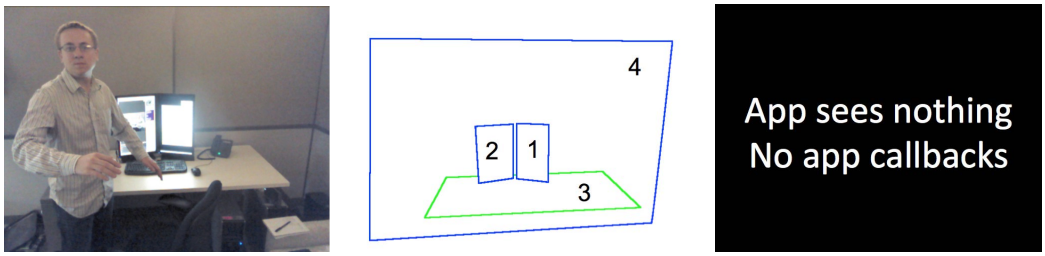
The *room skeleton* reveals the location and dimensions of all flat renderable surfaces in the room to the application as JavaScript objects, and exposes an API for displaying HTML content on each. We call these renderable surfaces *screens*. SurroundWeb handles rendering and projecting the content into the physical room. Using this interface, applications can perform room-location-aware rendering. Figure 5 visualizes the data that the room skeleton provides to the application.

Room Setup: 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 [25].

Runtime: 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

Application	Requires	Description
SurroundPoint	Room Skeleton	Each screen in the room becomes a rendering surface for a room-wide presentation (see Figure 6).
Car Racing News Site	Room Skeleton	Live video feed displays on a central monitor, with racing results projected around it (see Figure 3a).
Virtual Windows	Room Skeleton	"Virtual windows" render on surfaces around the room that display scenery from distant places (see Figure 3b).
Road Maps	Room Skeleton	Active map area displays on a central screen, with surrounding area projected around it (see Figure 3c).
Projected Awareness IM [3]	Room Skeleton	Instant messages display on a central screen, with frequent contacts projected above (see Figure 3d). 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.
Calorie Informer	Detection Sandbox	The Calorie Informer displays calorie count and serving information by recognized food products *without* knowing what products the user has.
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. 4: Web applications and immersive experiences that are possible with SurroundWeb.



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

Fig. 5: 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.

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 full-room presentation application pictured in Figure 6. While traditional presentation software is limited to a single screen, SurroundPoint can display information around the entire room. Each slide in a SurroundPoint presentation has "main" content to present on a primary display, plus optional additional content. By querying the room skeleton, SurroundPoint adapts the presentation to different room layouts.

Consider the case where the room has only a single 1080p 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 1920×1080 rectangle.

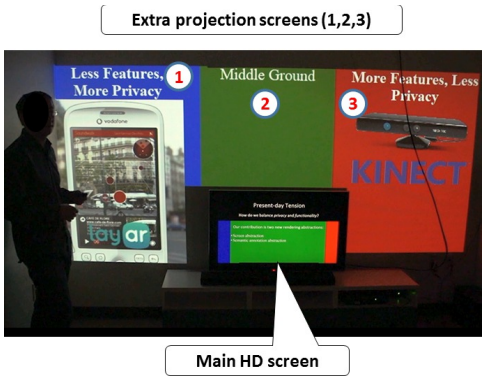


Fig. 6: The SurroundPoint presentation application uses the *room skeleton* to render an immersive presentation with least privilege. This image displays SurroundPoint running in our SurroundWeb prototype.

Based on this information, SurroundPoint knows that it should show only the "main" content.

In contrast, consider the room shown in Figure 5.

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. □

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 enables *object-relative rendering*, where an application specifies that content should be rendered by an object in the room. 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. However, object detection is a privacy challenge because the presence of objects, such as particular medicines, can reveal sensitive information about a user's life. 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.

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 [29]. Then, SurroundWeb checks registered content against a list of objects detected. If there is a match, the detection sandbox solves the rendering 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. In Section V, we show how applications specify rendering constraints via CSS and discuss the FLARE constraint solver [28].

Example 2 (Object-contextual ads) An application can register an ad to display if an energy drink can is present. 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 encouraging 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. However, we believe that we can leverage the sizeable amount of previous work on privacy-preserving ad delivery to make interaction possible in the future [9, 32]. We discuss this idea in more detail in Section VII. □

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 accommodate these devices, we extend the room skeleton with remote displays that we call *satellite screens*.

Satellite screens let applications build multi-player experiences without needing to explicitly tackle creating a distributed system. 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 of a particular SurroundWeb instance. 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. When a user closes the web page, the cloud service notifies the room skeleton to remove the screen from the room skeleton.

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. □

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

Property: *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.*

Mechanism: 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 display the registered content. The rendering process is handled by trusted code in SurroundWeb, which internally renders content regardless of the object's presence to defeat tracking pixels. 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. However, 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

Property: *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 have access to additional incidental information, which may be sensitive in nature. For example, many existing immersive room applications process raw camera data directly to determine where to render content. The raw camera feed can contain large amounts of incidental sensitive information, such the faces of children or documents in front of the camera. However, the application needs access to a subset of this data so it can intelligently determine where to place virtual objects in the physical room. *Therefore, rendering privacy is an important goal for balancing privacy and functionality in immersive room experiences.*

Mechanism: The challenge in rendering privacy is creating an abstraction that follows the *principle of 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

Property: *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. *Therefore, interaction privacy is an important goal for balancing privacy and functionality in immersive room experiences.*

Mechanism: We provide interaction privacy through a combination of two mechanisms. First, trusted code in SurroundWeb 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

Excluding external dependencies, our SurroundWeb prototype is written in 10K lines of C#, and 1.5K lines of JavaScript. We describe how we implement the core SurroundWeb abstractions from Section III, and how we expose the abstractions to web applications in a natural manner. We discuss how SurroundWeb extends CSP to let web sites safely embed content from other origins using `iframe`s. We walk the reader through writing the frontend to a karaoke web application, which illustrates how easy it is to write a SurroundWeb application, and describe a variety of items we have built using our prototype.

A. Abstraction Implementations

Our SurroundWeb prototype works with instrumented rooms, each of which contains multiple projectors and Kinect sensors. 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. Figure 1 displays an architectural diagram of the prototype.

Content rendering: The prototype uses the C# `WebBrowser` component to render HTML content in Internet Explorer. The prototype maps the rendered content to room locations, which are mapped back to display device coordinates. The prototype displays the content at the resulting display coordinates.

Room skeleton: The room skeleton exposes the sizes and relative locations of flat surfaces in the physical room that are able to display content. Our prototype constructs the room skeleton offline using KinectFusion [22]. Since Kinect depth data is very noisy, the prototype applies custom filters to the data stream that smooth the data to make it simpler to identify flat surfaces. Once the prototype detects the flat surfaces, it maps each back to display device coordinates.

Detection sandbox: The detection sandbox lets applications place content relative to recognized objects in the room without leaking the presence of those objects to the application. Our prototype detection sandbox consists of two core components: *object detection* to detect and locate objects in the physical room, and *constraint solving* to determine where to place web content in the room.

To detect objects, our prototype runs object classifiers on continuous depth and video feeds from Kinect cameras. While detecting arbitrary objects is a difficult computer vision problem, existing experiences use QR-like identifiers to recognize objects, which can be read using consumer-grade cameras. In addition, while our prototype is limited to vision-based object detection, object detection can be accomplished through other means, such as through RFID tags or sensors located in objects. After an object is detected, the prototype checks the current web page for registered content, then updates its rendering of the room.

SurroundWeb compiles the location of detected objects and application-specified object-relative content locations into 3D constraints. These constraints are fed into the FLARE constraint solver, which finds appropriate room coordinates for the content [28]. SurroundWeb uses these coordinates to render content appropriately in the room.

Satellite screens: We host a SurroundWeb satellite screen service in Microsoft Azure. Users 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 screen appears in the room skeleton as a new renderable surface with no location information, and the Azure service proxies user input to SurroundWeb.

Natural user interaction: The prototype uses the Kinect SDK to detect people and gestures [19]. Figure 7 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 generic mouse or keyboard event, then injects that event into the running web page.

Fig. 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.

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. 8: Properties of each screen object, which are exposed through JavaScript.

B. API Design

The prototype exposes the room skeleton, satellite screens, and detection sandbox to web pages through extensions to HTML, CSS, and JavaScript. We briefly describe these extensions below.

Room skeleton: To display content on a screen in the room skeleton, the application must first identify which screen it wants to use. The application queries for available screens by calling `Screen.getAll()`. Figure 8 displays all of the properties available on each `Screen` object. Then, it must encapsulate the web content in a `<segment>` HTML tag, which is a new tag that we add to the web platform. The segment tag must have a `screen` property with the relevant screen's `id` value assigned. Once this is done, the content will appear in the room on the specified screen.

```
// Display tweet on first screen in room skeleton.
$('#tweet').attr('screen', Screen.getAll()[0].id);

<segment id="tweet"><div class="tweet">
  @TypeScriptLang: TypeScript 1.4 now available!  Now
      with union types, type aliases, better inference,
      and more ES6 support!
</div></segment>
```

Constraint	Description
`left-of`	Place the segment to the left of the object.
`right-of`	Place the segment to the right of the object.
`above`	Place the segment above the object.
`below`	Place the segment below the object.
`valign`	Vertically align the segment with the object relative to the plane perpendicular to the ground.
`halign`	Horizontally align the segment with the object relative to the ground.

Fig. 9: Our prototype lets applications specify object-relative layout constraints on content via new CSS properties. These can be applied to `segment` HTML elements, which we add to the web platform. Each property takes a list of object names.

Satellite screens: The process for displaying content on a satellite screen is identical to displaying content on a screen in the room skeleton. Satellite screens appear in the list of screens returned through `Screen.getAll()`, so applications simply associate their id with the segment it wishes to display on each.

Detection sandbox: Our prototype augments CSS to support object-relative content positioning in the room.[2] Web applications encapsulate the web content in a `<segment>` tag, and specify object-relative CSS constraints on the tag. We describe the constraints that our prototype supports in Figure 9, which translate into FLARE 3D constraints in a straightforward manner [28].

```
// Advertise lightbulb prices near a lamp in the room.
#lightbulb-prices { left-of: "lamp" }

<segment id="lightbulb-prices"><div class="prices">
  GE 60-Watt Incandescent Light Bulbs: 4 for $6.49 @
      Amazon.com
</div></segment>
```

C. Embedding Web Content

We now discuss how to handle embedding of content from different origins inside of a web page running in SurroundWeb. In this discussion, we use the same web site principal as defined in the same-origin policy (SOP) for SurroundWeb web pages, which is labeled by a web site's origin: the 3-tuple ⟨`protocol`, `domainname`, `port`⟩.

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: SurroundWeb web pages 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 page's Document Object Model (DOM), which includes all browser APIs, SurroundWeb interfaces, and segments defined by the web page.

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 SurroundWeb.

[2]Since we are unable to modify Internet Explorer's CSS engine, our prototype implementation exposes constraints through equivalent HTML attributes on segment elements.

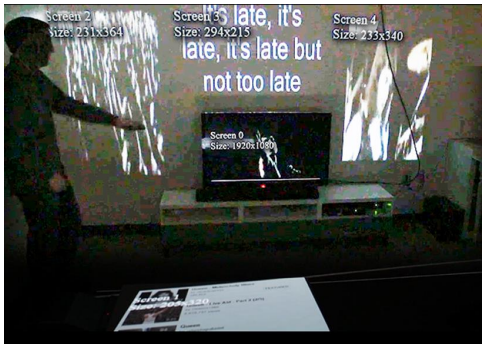


Fig. 10: The Karaoke application uses the room skeleton to make room layout decisions. The application dynamically decides to show a video on the high resolution TV. Lyrics and photos go on projected screens, and related videos are projected on the table.

Web pages 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 pages to control whether or not particular origins can access the SurroundWeb interfaces from within a frame. If a web page 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 page allows an embedded origin access to these interfaces, then the `iframe` will be able to render content in the room skeleton and detection sandbox.

D. Walkthrough: Karaoke Application

To illustrate how a web page can be developed using SurroundWeb, we will walk through a sample Karaoke application, shown in Figure 10.

This web page renders karaoke lyrics above a central screen, with a video on the central screen and pictures around the screen. Related songs are rendered on a table. The web page 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 page must scan the *room skeleton* to assign the segments specified in the HTML to relevant *screens*.

1) Using JavaScript, the web page locates the vertical screen with the highest resolution, which will contain the video:

```
<html><head>
<script type="text/javascript">
  // Wait for HTML to load before running code.
  window.onload = function() {
    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);

    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);

    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);

    // Assign random related media to 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);
    }
  };
</script></head>
<body>
  <segment id="lyrics"><!--Lyrics HTML--></segment>
  <segment id="video"><!--Video HTML--></segment>
  <segment id="related"><!--Related songs HTML-->
    </segment>
</body></html>
```

Fig. 11: The complete SurroundWeb code for the Karaoke Application.

```
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 page 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. Figure 11 shows all of the rendering code put together. Should the chosen configuration be suboptimal

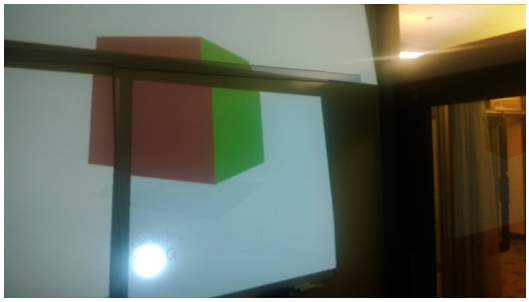


Fig. 12: Using our adaptation library, a 3D-cube demo runs across multiple surfaces in our SurroundWeb prototype after a simple one-line change. While the demo believes it is drawing a 3D scene to a single HTML5 canvas, it is actually drawing the scene across three SurroundWeb screens.

for the user, the Karaoke application can provide controls that allow the user to dynamically choose the rendering location of the segments.

E. Application Showcase

Using the SurroundWeb prototype, we implemented room-wide presentation software, a soda can object detector, and a JavaScript library that adapts existing HTML5 canvas applications to work across screens in the room skeleton.

Room-wide presentations: SurroundPoint is a room-wide presentation application where each slide can span multiple surfaces in the room. A SurroundPoint presentation can be seen in Figure 6. Each slide contains "main" content, and a set of optional additional content. By querying the room skeleton, SurroundPoint adapts the presentation to different environments. We used SurroundPoint to deliver multiple presentations to the press [21, 31].

Soda can detector: We implemented a soda can detector that supports detecting different types of soft drink cans using a nearest-neighbor classifier based on color image histograms. When the classifier detects a soda can, the detector alerts the detection sandbox, which updates the rendering of the room. Note that object detectors are considered to be a trusted part of SurroundWeb, and web pages do not directly interact with them.

Adapting existing web content: Since SurroundWeb merely *adds* additional features to the web platform, existing web sites already function in SurroundWeb, but they are limited to a single screen. To ease the transition into a multi-screen layout, we wrote a JavaScript library called JumboTron that automatically adapts applications that use the HTML5 canvas element to render across multiple surfaces in the room. The library detects screens in the room skeleton that are close together, and creates distinct canvas elements for each screen. It then binds the canvas elements together in a JumboTron object, which emulates the canvas interface. The application interacts with the JumboTron object, which proxies the input to the relevant sub-canvases. Figure 12 displays a `three.js` demo that required a one-line change to use our library and render across surfaces.

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

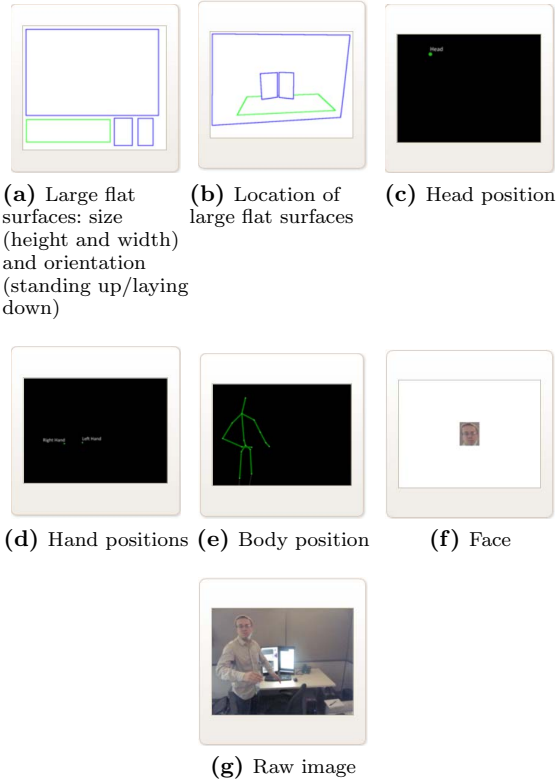
To evaluate the privacy of our abstractions, we conducted two surveys of random US Internet users using the Instant.ly survey service [33]. Below are the research questions we were trying to address via surveys and the answers we discovered experimentally.

- **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 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 14 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



(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. 13: Survey participants chose which data should be given to three hypothetical applications by choosing zero or more of the above options.

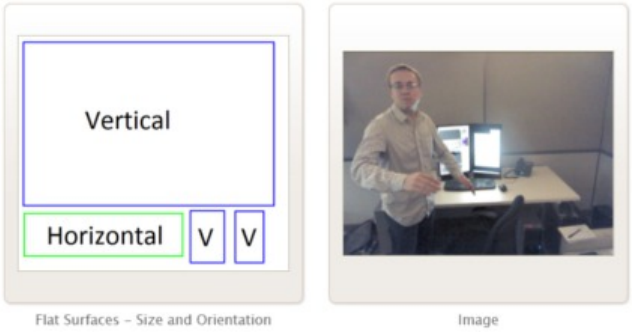


Fig. 14: Excerpt from our room skeleton sensitivity survey. We asked users which image contains more information that they consider private; 87.5% believe the raw image contains more private information.

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 13. 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 15 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.

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

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

Fig. 15: Scanning times, in seconds, and results for three representative rooms.

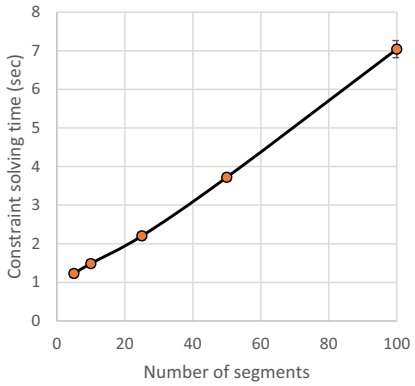


Fig. 16: Solver performance as the number of segments registered with the detection sandbox increases. The error bars indicate 95% confidence intervals.

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 16 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.
- 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.

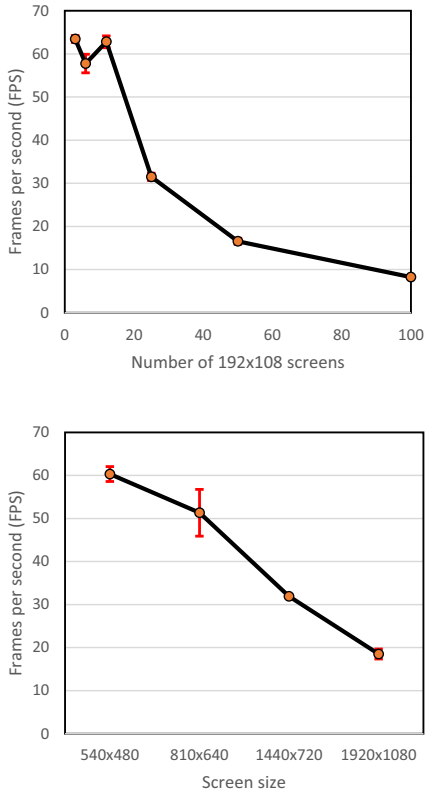


Fig. 17: On the left, maximum rendering frame rate as the number of same-size screens increases. On the right, the maximum rendering frame rate of a single screen as its size increases. Error bars indicate 95% confidence intervals.

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 17 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, 24]. 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 sites 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 site 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, 36].

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 by 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 [29]. 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. We could use information flow control to prevent sensitive information from affecting data sent back to web servers, but implementing sound information flow tracking for all of JavaScript and its natively-implemented browser environment without unduly impacting performance presents a formidable challenge.

The current detection sandbox is limited to trusted object classifiers, preventing applications from providing application-specific object classifiers. The detection sandbox could be extended to support untrusted object classifiers through native sandboxing [27]. These untrusted classifiers would be provided with raw sensor data, and would be able to alert the detection sandbox when they detect

objects, but would not be able to leak any information outside of the sandbox.

Our detection sandbox 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.

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 18 summarizes the capabilities that these research systems surface compared with SurroundWeb.

A. 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.

B. 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 [37]. 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. SurroundWeb avoids this side channel by immediately loading and rendering any content registered with the detection sandbox, regardless of whether or not it will be displayed in the room.

C. 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 [26]. 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

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	[26]		✓					
	πBox	[15]			✓				
HCI	Illumiroom	[13]	✓					✓	
	Panelrama	[37]					✓		
	Layar	[14]				✓			✓
	Argon	[7]				✓			✓

Fig. 18: 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.

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 [34].

D. 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 [36]. Liang *et al.* discovered a novel CSS timing attack for sniffing browser history [16].

Nikiforakis *et al.* [23] propose a way to combat browser-based stateless fingerprinting by applying randomization policies. These are evaluated to minimize page breakage they might cause. This approach is proposed as a complement to the private browsing mode designed to protect against stateful fingerprinting.

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, 35] to build better fingerprinting libraries. Yen *et al.* [38] 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 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 technique to search 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 three 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. *Satellite screens* let applications display content across multiple devices. 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.

Acknowledgements

We thank Lydia Chilton for suggesting the terms "room skeleton" and "detection sandbox." We thank Janice Tsai for help with survey design and human protection review. We thank Doug Burger, Hvroje Benko, Shuo Chen, Lydia Chilton, Jaron Lanier, Dan Liebling, Blair MacIntyre, James Mickens, Meredith Ringel Morris, Lior Shapira, Scott Saponas, Margus Veanes, and Andy Wilson for helpful discussions and feedback. We also thank the anonymous reviewers for their useful feedback, which greatly improved this paper.

References

[1] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for Fingerprinters. In *Proceedings of the 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 *Proceedings of the 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 *Proceedings of Graphics Interface*, 2010.

[4] P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the Conference on Computer and Communications Security*, 2011.

[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*, 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. Master of Science Thesis, Chalmers Univesity of Technology, 2012.

[9] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *Proceedings of the Symposium on 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 *Proceedings of the USENIX Security Symposium*, 2013.

[12] S. Jana, A. Narayanan, and V. Shmatikov. A Scanner Darkly: Privacy for Perceptual Applications. In *Proceedings of the 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 *Proceedings of the Conference on Human Factors in Computing Systems*, 2013.

[14] Layar. Layar Solutions, 2015. `https://www.layar.com/solutions/`.

[15] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. Box: A Platform for Privacy-Preserving Apps. In *Proceedings of the 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 *Proceedings of the 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 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] D. Molnar, E. Ofek, and Microsoft Corporation. SurroundWeb: Spreading the Web to Multiple Screens. `http://research.microsoft.com/apps/video/?id=212669`.

[22] 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 *Proceedings of the IEEE International Symposium on Mixed and Augmented Reality*, 2011.

[23] N. Nikiforakis, W. Joosen, and B. Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *Proceedings of the International World Wide Web Conference*, 2015.

[24] 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*, 2013.

[25] 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`.

[26] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, and H. J. Wang. World-Driven Access Control. In *Proceedings of the Conference on Computer and Communications Security*, 2014.

[27] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the USENIX Security Symposium*, 2010.

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

[29] R. M. Smith. The Web Bug FAQ, 1999. `https://w2.eff.org/Privacy/Marketing/web_bug.html`.

[30] S. Stamm. Plugging the CSS History Leak, 2010. `http://mzl.la/1mzshEY`.

[31] R. Taylor. Internet everywhere? How the web could be on every surface of a room. `http://www.bbc.com/news/technology-27243375`.

[32] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

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

[34] 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 *Proceedings of the IEEE Conference on Privacy, Security and Trust*, 2012.

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

[36] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson. I Still Know What You Visited Last Summer. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

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

[38] 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 *Proceedings of the Network and Distributed System Security Symposium*, 2012.