CellMate: A Responsive and Accurate Vision-based Appliance Identification System

Kaifei Chen
Takeshi Mochida
Jonathan Fürst
John Kolb
David E. Culler
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-154
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-154.html

September 14, 2016

Acknowledgement

This work is supported in part by the National Science Foundation under grant CPS-1239552 (SDB).

CellMate: A Responsive and Accurate Vision-based Appliance Identification System

Kaifei Chen[†], Takeshi Mochida[†], Jonathan Fürst[‡], John Kolb[†],
David E. Culler[†], Randy H. Katz[†]
[†]University of California Berkeley, [‡]IT University of Copenhagen,
{kaifei, tmochida}@berkeley.edu, jonf@itu.dk, {jkolb, culler, randykatz}@berkeley.edu

ABSTRACT

Identifying and interacting with smart appliances has been challenging in the burgeoning smart building era. Existing identification methods require either cumbersome query statements or the deployment of additional infrastructure. There is no platform that abstracts sophisticated computer vision technologies to provide an easy visual identification interface, which is the most intuitive way for a human. We introduce **CellMate**, an responsive and accurate vision-based appliance identification system using smartphone cameras. We innovate on optimizing and combining the advantages of several of the latest computer vision technologies based on our unique constraints of accuracy, latency, and scalability. To evaluate CellMate, we collected 4008 images from 39 room-size areas across five campus buildings, making the size one order of magnitude greater than prior work. We also collected 1526 human-labeled images and tested them on different groups of areas. With existing indoor localization technologies, we can easily narrow down the location to ten areas and achieve a success rate of more than 96% within less than 60 ms server processing time. We optimized average local network latency to 84 ms and therefore expect around 144 ms total identification time on the smartphone end.

1. INTRODUCTION

Cameras are on most smartphones today but are not well utilized to provide better interactions between humans and their environment. Advances in computer vision over the past several years enable many potential applications using smartphone cameras to allow users to understand and interact with the environment. However, there is no system that leverages them to provide a simple interface to users or a simple utility service for building applications. Imagine you walk into a meeting room and need to use a projector but cannot find the remote control to turn it on and connect to it. Wouldn't it be useful if you could point your smartphone camera at the projector and the control interface just appears on the phone's screen?

There are many ways to identify and interact with smart appliances. Existing approaches are not intuitive or convenient for two major reasons. First, some approaches require users to describe the appliance in cumbersome ways. For example, users may have to query appliances following a strict syntax [3]. Although there are natural language based solutions (e.g., Amazon Echo), it is generally difficult to have a unique human-friendly description of a device, especially when multiple identical instances are close to each other. Moreover, speaking vocal commands is not courteous in quiet environments like offices. Second, many approaches involve additional infrastructure. Some require setting up laser [15] or infrared [22] signal receivers on appliances, and a user carries a signal transmitter for interaction. Some merely depend on indoor localization [9], which suffers from several meters in error when restricted to existing infrastructure [11] and a lack of orientation information. This approach also does not work for interactions at a distance. Fiducial marker (e.g., QR code) based identification [21, 5] requires attaching markers to appliances, and these markers cannot be recognized at distance, from a large angle, or with poor illumination. Therefore, we propose a vision-based approach that identifies appliances in an image by localizing it in 3D space. Compared to existing approaches, it needs no additional infrastructure, requiring only labeled videos, and yields higher accuracy.

Augmented Reality (AR) overlays objects or information in a continuous camera view by understanding all visible objects locally. We differ from AR in two aspects. First, we cannot have an always-on smartphone camera for various reasons (e.g., energy, privacy). Instead of localizing a stream of images along a continuous trajectory, we localize a single image in a much larger 3D space, which will have a radius of at most 10 meters if we use other indoor localization technologies [11]. Second, AR focuses on the semantics and shapes of objects in a user's view, whereas our application seeks to use localization in order to identify the specific appliance a user is looking at.

In this paper, we introduce **CellMate**, a new kind of visual appliance identification system that leverages advantages of different computer vision technologies and organizes them to optimize single image queries for fast response, high accuracy, and scalability. To use CellMate, a user needs to capture a video of appliances she is concerned about using a depth camera, convert it to a 3D model using RTAB-Map [8], label one image of every appliance using the CellMate labeling tool, and send the resulting database file to the CellMate

BuildSys '16 Nov 16–17, 2016, Stanford, CA, USA

DOI: 10.475/123_4

server. CellMate collects these database files, extracts the data, and runs a RESTful server over HTTP for appliance identification using selected computer vision technologies. A client application can simply post a query image (without depth information) to the server. The server then extracts its distinct visual features, localizes the query image in the 3D model, and projects all labeled points onto the image to identify any appliances it contains. The client will receive a response containing the identified appliance at the center of the image. We also built an Android app as an example client, which is shown in Figure 1.

CellMate tackles two important problems. First, the visual contexts of appliances change over time. Because we crowdsource videos, users can update an area with a new video when things change and identification becomes inaccurate. Second, there are no automatic appliance labeling technologies. The CellMate labeling tool allows users to label an appliance only once among all images in a database by projecting its 2D label onto its 3D model, allowing the same label to be applied to all images of the same appliance.

To evaluate CellMate, we collected 4008 images from 39 videos, each of which covers an area from one of five different campus buildings. To our knowledge, this dataset size is one order of magnitude larger than those used in prior work [6]. After collecting these videos, we also collected 1526 test images of appliances from different angles and distances and manually labeled them. In the evaluation, CellMate achieves more than 96% accuracy when identifying appliances among 10 areas. We also achieve an average of 60 ms server computation time, with an average of 84 ms local network latency. The total identification latency 144 ms is much lower than the 400 ms identification response time limit [13].

We summarize our main contributions as follows:

- We build a holistic system[1] involving sophisticated computer vision technologies and provide an easy-to-use vision-based appliance identification service.
- We explore the placement of computations and combination of algorithms to optimize both responsiveness and accuracy.
- We build a labeling tool that requires only one label per appliance among all images of that appliance.
- We conduct evaluations from different real world scenarios with a dataset that is significantly larger than those featured in prior work to demonstrate scalability, while also achieving lower latency and better accuracy.

The rest of this paper is organized as follows: Section 2 gives two use cases and talks about our system requirements. Section 3 describes an overview of the system architecture and algorithms. Section 4 evaluates CellMate by its success rate, latency, and several other aspects. Section 5 gives some discussions and our future work. Section 6 describes related work in the area of object identification. Section 7 concludes the paper.

2. SYSTEM DESIGN

This Section discusses our system design requirements. To motivate them, we provide two use cases and three measurements on computation time and network latency under different circumstances.

2.1 Use Cases

[1]https://github.com/SoftwareDefinedBuildings/CellMate

Figure 1: CellMate Android App

2.1.1 *Identification and Activation*

A user enters a room and wants to change the brightness of the light on the south side of the room. Using state-of-the-art Building Management System (BMS) apps, this involves typing "light", the room number, and "south" into a search bar, which is tedious and error-prone. Using the CellMate app, she simply points her phone camera at the light and the light's control interface appears instantaneously. She changes the brightness as desired and closes the app.

2.1.2 *Data Crowdsourcing*

A user wants to add controllable appliances around her workspace into CellMate. She first captures a short video that contains all appliances she wants to control. Using the CellMate app, she can browse the video frames, long press on any appliance and type its label. Finally, she uploads the video to a CellMate server. After this, she is able to take a picture to identify any appliance she captured in the video. Over time, her workspace will change, such as new posters on the wall, moving a space heater, etc. She notices this when the CellMate identifications begin to fail. To resolve this, she can re-capture and upload another video, which replaces the old one on the CellMate server.

2.2 Location of Computation

It is important to know where to run some essential computations considering both computation power and network latency. If it's beneficial to run some logic on the client, for example, we need to design our system with this consideration in mind.

2.2.1 *Speeded Up Robust Features (SURF)*

We first look at Speeded Up Robust Features (SURF), which are visual features of an image that are both fast to compute and robust to scaling, rotation, viewing angle, and varying illumination [1]. These features are needed to localize an image. We can extract SURFs on the phone to potentially reduce the amount of data transmitted to server.

In our first experiment, we compare the computation time of SURFs on different platforms, which include a Qualcomm Snapdragon Quad-core 1.2 GHz Cortex-A7 CPU on a LG G2 Mini Android phone, an Intel i7-4790 CPU on an Ubuntu machine, and a NVIDIA GeForce GTX 970 GPU on a Ubuntu machine. In the experiment, we randomly select 1000 images from our test image set, down-sample them to 640×480 resolution (or 480×640 depending on the image orientation), and perform both SURF detection and descriptor computa-

tion on these platforms using OpenCV.

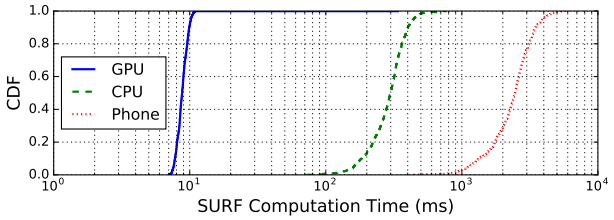


Figure 2: CDF of SURF Computation Time on Different Platforms

Figure 2 shows the CDF of SURF computation time on different platforms. The x axis is in log scale. We can see that there is an order of magnitude difference between the phone CPU and the server CPU as well as between the server CPU and the server GPU. Because SURF itself will take several seconds on the phone, we run it on the server GPU. This means we need to transmit every query image to the server.

2.2.2 *Image Upload Time*

Because we must use a server, we measure how much time it takes to upload a query image. We randomly select 1000 640×480 (or 480×640) images and send them from a LG G2 Mini Android phone over the UC Berkeley campus network to four servers at different locations: a campus server in the local network, an Amazon Web Services (AWS) instance in San Jose (43 miles away), an AWS instance in Virginia (2350 miles away), and an AWS instance in Ireland (5050 miles away). We obtain raw pixels from the camera in YUV format from Android, take only the Y component, and transmit the raw pixels over HTTP, which is $640 \times 480 = 307200$ bytes.

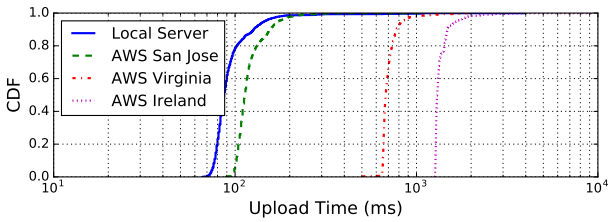


Figure 3: CDF of Upload Time of Raw Pixels of a 640×480 Image to Different Servers

Figure 3 shows the time it takes to upload the raw pixels of an image to servers running in different locations listed above. The local network generally has a delay of less than 100 ms, while a nearby cloud service can typically induce less than 110 ms in latency. This suggests that we may need to use nearby servers for best performance.

2.2.3 *Using JPEG Compression*

JPEG has been very efficient in terms of both computation time and compression ratio [20]. We explore the benefits of trading time to compress the image for a reduced amount of data to upload. After we get raw Y component pixels, we use cross-compiled OpenCV for Android to encode the raw pixels to JPEG format with a quality factor of 95 (on a 0 to 100 scale) and measure the total time of the JPEG compression and JPEG data transmission. We ignore the server side JPEG decompression because it easily takes less than 1 millisecond for a 640×480 image [16].

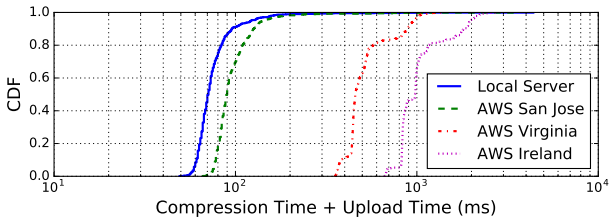


Figure 4: CDF of JPEG Compression Time + Upload Time of a 640×480 Image to Different Servers

Figure 4 shows the CDF of the combined JPEG compression time and upload time. On average, JPEG produces 83267 bytes from 307200 bytes (compression ratio is 3.69). Compared to transmitting all the raw pixels, it takes about 20 ms less to compress and upload an image to a local server, so we chose to perform JPEG compression for every query image.

In summary, based on these three measurements, we recommend that other clients also compress and upload query images in JPEG format, and we perform all the remaining computations in the cloud with a GPU.

2.3 Design Choices

Motivated by the use cases and the computation placement measurements, we make the following design choices.

Thin Client: As we conclued from our experiments, clients are not required to perform any computation and only need to send the query image over HTTP POST to a CellMate server and wait for the result.

Crowdsourced Data: For our use case, crowdsourcing is the ideal method for gathering new data and updating stale data. As shown in Section 4, we only require a short video to cover an area like a meeting room or office cubicle. The video can contain as few as 25 images to achieve an average identification success rate of 90%. For now, we require videos with depth information, which can be obtained from a depth camera. Our final goal is to support RGB videos as well.

Manual Data Labeling: Because state-of-the-art object detection and recognition is not effective enough (up to 63% accuracy [19]), and we differentiate identical appliances at different locations, we decide to manually label appliances on crowdsourced videos. To make this easy, we need a labeling tool where the user can browse images and click to label any appliance. Labeling one appliance should be as easy as clicking on one pixel and typing in the appliance name, as opposed to labeling all occurrences in all images.

Scalable and Responsive Server: Because all computations are on the server, it needs to be scalable and responsive. We aim to scale to a space that can be determined using existing indoor localization systems that do not require any dedicated infrastructure, which can already provide sub-10 meter accuracy [11]. As proposed in [13], an identification system should have a response time of less than 400 ms. As our measurements show in Section 2.2, the average network latency to upload an image to a nearby cloud server is about 100ms, so CellMate's server-side processing should incur less than 300 ms of latency.

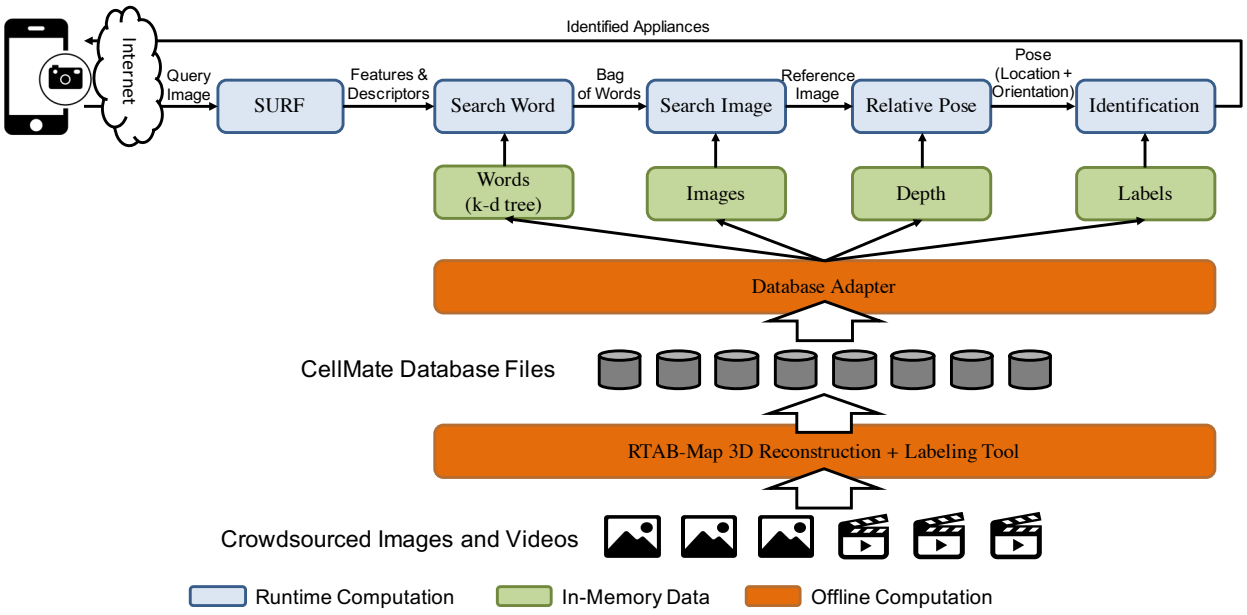


Figure 5: CellMate System Architecture. The offline components capture, label, and prepare data for runtime, and the runtime components localize the query image on a 3D model to identify the appliances it contains using a five-stage pipeline.

3. SYSTEM OVERVIEW

Figure 5 shows the architecture of the end-to-end CellMate system. In the offline phase, users collect videos with a depth camera using RTAB-Map [8], label appliances using the CellMate labeling tool, and send the database files to the CellMate server. At runtime, the CellMate server passes every incoming query image to a five-stage pipeline for appliance identification. The pipeline extracts SURFs from the image and converts the SURFs to a bag-of-words (BOW) using their nearest words in a kd-tree. Next, the pipeline finds the most similar image as a reference using the BOW, and uses the reference image to localize the query image in 3D space. Finally, we can identify the appliance that appears in query image based on its location. We discuss the details below.

3.1 Android Mobile Application

An example screenshot of the CellMate Android app is shown in Figure 1. It has a full screen see-through camera view with control widgets overlaid on top. A capture button is at the bottom right. The bottom left shows the currently identified appliance. Appliance control buttons are shown at the bottom center. Without loss of generality, we currently only support "ON" and "OFF" buttons.

Using the mobile app involves two steps: identification and interaction. When the capture button is clicked, an image of the current camera view (query image) is immediately sent to a CellMate server over a RESTful API for identification. The server returns the label of the identified appliance that is closest to the image center. For privacy reasons, the client only uploads the image that a user intends to capture.

After receiving the appliance label, the client retrieves its control interface and metadata from the BMS and updates the UI accordingly. The control buttons are initially disabled and translucent, and will be enabled after parsing the control interface and metadata. The control logic is defined by the appliance metadata and integrated with the BMS. In this paper, we focus on identification.

In the Android app implementation, there are two main tasks. First, all raw camera frames are directly streamed to a full screen image viewer. Second, when the capture button is clicked, the app sends the most recent camera frame to the server after down-sampling and compressing it. To minimize the identification latency, we disabled auto focus, auto exposure, and auto white balance, which can take up to 1 second. We believe there is no need to worry about blurring images, because a user usually points the camera at the appliance steadily when she wants to control it.

3.2 Offline Processing

We process crowdsourced videos and labels offline to build a database for runtime appliance identification. There are two main steps in the offline processing: 3D reconstruction and labeling.

3.2.1 Crowdsourcing Videos

Users can take a video of any area in a building to feed into the CellMate database, such as an office cubicle, a home kitchen, or a meeting room. The amount of data required is very small. For example, in our evaluation data, 60 images can cover a 4-person office room. When an area undergoes large changes, such as when furniture is moved or wallpaper is changed, a new video covering the same area is required to update the CellMate database. Currently, CellMate only supports RGB-D videos collected using RGB-D cameras (e.g., Microsoft Kinect, ASUS Xtion PRO LIVE[2]), which is very easy to do using the RTAB-Map data collection tool. Crowdsourced data is processed by RTAB-Map [8] for 3D reconstruction and appliances are labeled manu-

[2]https://www.asus.com/us/3D-Sensor/Xtion_PRO_LIVE/

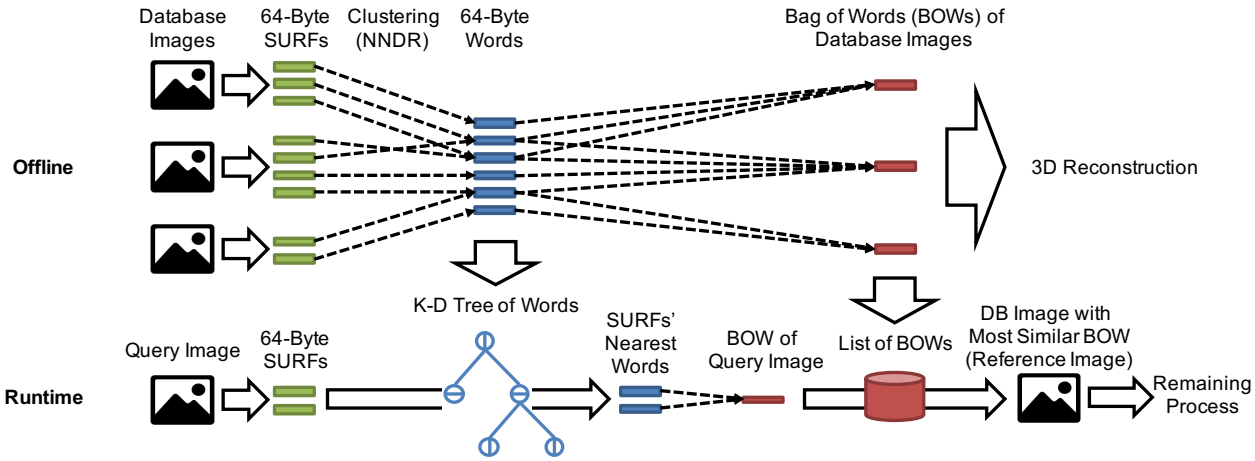
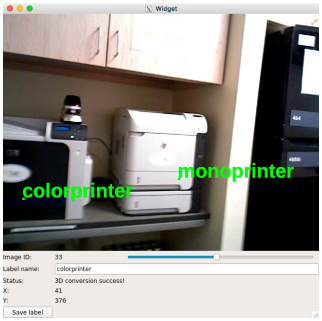


Figure 6: CellMate Data Flow. The offline part extracts SURFs from captured images, clusters them to words, and forms bag-of-words (BOW) to describe images. The runtime extracts SURFs from query image, finds their nearest words in a kd-tree to create its BOW, which is used to look for a reference image.

ally using CellMate labeling tool, which adds a new "Label" table into the RTAB-Map database. The details are in the next two sections.

3.2.2 3D Reconstruction

(a) A Point Cloud Example



(b) Labels of two printers from a left angle

(c) Same labels of two printers from a right angle

Figure 7: CellMate Labeling Tool

We use RTAB-Map to reconstruct 3D models from videos. The purpose is to significantly simplify the labeling process. With a 3D model, one label in one image can be shared with all other images containing the same physical object. Specifically, by projecting the labeled pixel to its 3D point in the 3D model, we can project it back to any other image containing the same point. As shown in Figure 6, to build a 3D model, RTAB-Map first extracts SURFs from all images in a video. Then it clusters SURF descriptors to 64-byte words, where SURF descriptors in one cluster are meant to be from the same 3D point. Following best practices, it uses the Nearest Neighbor Distance Ratio (NNDR) clustering approach [10]. RTAB-Map then finds common words between images using their Bag-of-Words (BOWs) and uses the common words and their pixels to compute the relative poses between images. A pose is the combination of a 3 Degree-of-Freeedom (DOF) location and 3 DOF orientation. After all relative poses are calculated, it combines all images into one uniform coordinate system to form a 3D model. Figure 7a shows a 3D model of an office kitchen generated from its video. RTAB-Map saves all original data and results in a SQLite database.

3.2.3 Labeling Tool

Because labeling relies on a human, we have built a tool to visualize images and add labels to simplify this process. Figure 7b and Figure 7c give two example screenshots of our labeling tool from the same area as in Figure 7a. As we can see, the labeling tool displays the images and the labels read from a 3D model. The user can click on any pixel and type a new label to save into the database. As we mentioned before, each appliance only needs to be labeled once, because every labeled pixel is projected to a 3D point so the label can propagate to other images. In this specific example, the two labels "colorprinter" and "monoprinter" are visible in both images, even though these labels are added from only one image. However, this 2D-to-3D projection will fail if the depth value of the pixel is missing, which is not uncommon with today's RGB-D cameras. To address this problem, we display a message indicating whether the projection is successful and only enable the save button when the selected pixel has a valid depth value. Many techniques can estimate missing depth values, and we plan to adopt them in our future work.

3.3 Runtime Identification Pipeline

At initialization, the CellMate server reads data from a list of CellMate databases and organizes the data structures for optimal runtime performance. It starts an HTTP server

and five threads for five stages of the pipeline. As shown in Figure 5, there is no overlap between the data used by the different stages. Therefore, it will be straightforward to distribute and parallelize the computation. On arrival of a query image, the HTTP server starts a worker thread to decode and pass the image to the first stage of the pipeline, and then waits for the result before responding. We describe an overview of each step in the pipeline below.

SURF: In the five-stage pipeline, the server first computes SURFs on the query image using its GPU. This step involves SURF detection and SURF descriptor computation. SURF detection finds interesting keypoints (e.g., corners) at different scales. A SURF descriptor is a 64-byte array that describes a SURF keypoint in a robust way against various scales, orientations, and illuminations. Figure 6 shows the SURFs extracted from the query image. To avoid excessive computation, we cap the number of SURF features of an image at 400, an empirically optimal number also used by RTAB-Map.

Search Word: This step finds the words nearest to the image's SURFs in the database to form a Bag-of-Words (BOW) describing the query image. As shown in Figure 5 and Figure 6, our runtime builds a kd-tree of all words in the database offline to speed up the look-ups. A node in a k-d tree is a data point that partitions all data on the branch based on their values in a particular dimension, which enables average $O(\log n)$ nearest neighbor search complexity. We use the kd-tree implementation in the Fast Library for Approximate Nearest Neighbors (FLANN) [14] and its default parameters, which builds 4 randomized kd-trees to reduce the possibility of all unbalanced kd-trees.

Search Image: This step finds a reference image to localize the query image. A reference image should have enough common words with the query image to compute their relative pose in next stage. In addition, if there are multiple SURFs whose nearest words are the same, we won't know which SURF is corresponding to the same word in the other image, so we only use the words that appear once in an image. Therefore, we define our objective function as the cardinality of the set of common unique words between two images, which we call similarity:

$$\text{sim}(A, B) = |\text{Unique}(BOW_A) \cap \text{Unique}(BOW_B)| \quad (1)$$

where BOW_A and BOW_B are the sets of words in each image respectively, and Unique(BOW) gives a set of words that only appear once in BOW. Figure 6 shows how we find a reference image with the highest similarity for the remaining processes. However, because similarity is different from Euclidean distance, we must iterate over all images instead of using a kd-tree. Due to noise in the data and errors in SURFs and the clustering algorithm, words and 3D points may not have a perfect one-to-one correspondence, and can reveal large inconsistencies in the next stage. Thus, we pass the ten most similar images to next stage for extra reference image candidates.

Relative Pose: This step calculates the global pose (location and orientation) of the query image using a reference image and their common words. Because the reference image has depth data, we can get the 3D points of its words in the global coordinate system. This is therefore a Perspective-N-Point (PnP) problem [17], which is to compute the query image's global location as a vector $t_{3\times1}$ and orientation as a matrix $R_{3\times3}$ from the homogeneous coordinates of its N pixels as a matrix $P^{2D}_{3\times N}$ and homogeneous coordinates of their corresponding 3D points as a matrix $P^{3D}_{4\times N}$:

$$P^{2D}_{3\times N} = K_{3\times3}[R_{3\times3}|t_{3\times1}]P^{3D}_{4\times N} \quad (2)$$

where $K_{3\times3}$ is the intrinsic matrix that characterizes the camera lens. We iterate through the most similar reference images to solve Equation 2, until we achieve sufficiently small error.

Since our data is a list of independent videos, each of them has its own coordinate system. Our query image's global pose will be in the final reference image's coordinate system.

Identification: After getting the query image's pose, we can identify the appliances it contains. We only look at labels in the databases generated from the video containing the final reference image. We project all labels that are in front of the camera (if the Z axis value is positive in the camera's coordinate system) onto its 2D plane, and return them sorted by their pixels' distances to the center of the image. The first appliance in the list will be the final identified appliance.

4. EVALUATION

We describe several evaluations of our current CellMate implementation in this section. We evaluate accuracy and latency when coverage area and data are scaled up, scalability with concurrent users, robustness to environmental changes, and energy consumption on the mobile client.

4.1 Experimental Setup

In our current CellMate implementation, the Android app contains 1549 lines of Java code, the server contains 3202 lines of C++ code, and the labeling tool contains 392 lines of C++ code. We use RTAB-Map to read RGB-D videos from an ASUS Xtion PRO LIVE RGB-D camera and perform 3D reconstructions from these videos. After recording a video, we use an LG G2 Mini Android phone to capture test pictures of appliances from different angles and distances. To ensure a large enough dataset, we consider both controllable objects (e.g., lights) and non-controllable objects (e.g., bookshelves) in our evaluation.

Our dataset contains 39 videos of room-size areas across five buildings in two university campuses. The areas include conference rooms, office cubicles, lounges, kitchens, hallways, etc. There are 4008 images for 3D reconstructions, 263 labels for 179 appliances, and 1526 images for testing. This dataset represents a large enough part of a building that can be narrowed down using existing indoor localization techniques [11]. Compared to prior work [6], we have more than an order of magnitude more modeling images, which enables us evaluate our system at scale.

Our setup includes a CellMate server running in a docker container on an Ubuntu machine, which has an 8-core Intel i7 CPU, 16 GB of memory, and a NVIDIA GeForce GTX 970 GPU. Given that we already have network studies in Section 2, we focus here on server side evaluations. We run a Python script as a client in the same docker container for testing. It simply uploads test images to the server and compares the returned results with the ground truth.

4.2 Success Rate

We run several rounds of tests and study the identification success rates and latencies. Each round initializes a

CellMate server with a combination of CellMate areas (each has its own database), and tests using their test images. Because there are too many possible combinations for 39 databases (i.e., $\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$, $n = 39$), we just generate a random permutation, start with the first area, and add a new area in each subsequent round.

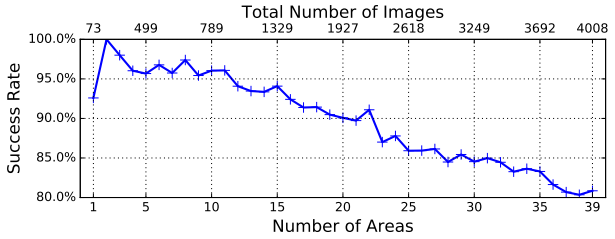


Figure 8: Success Rate vs. Number of Areas

Success rate is defined as the percentage of images that are identified correctly. The more databases we query an image against, the more likely there is a similar image from the wrong database that is used as the reference image, lowering the success rate. Figure 8 shows the success rates with different numbers of areas. The bottom x axis shows the total number of area databases included in the test, while the top shows the total number of images in those databases. The success rate starts at around 93%, increases to 100% afterwards, and drops to 81% when identifying among 39 areas (or 4008 images). Fortunately, existing indoor localization can practically yield less than 10 meters in error [11], so we can easily narrow down our search space to less than 10 rooms and achieve more than a 96% success rate.

4.2.1 *Failure Analytics*

(a) Labels Too Close

(b) Label Occluded

(c) Similar Scenes

(d) Not Covered in DB

(e) Lack of Features

(f) Incorrect Label

Figure 9: Common Failure Cases (left is query image, right is reference image)

To examine the causes of failures in our system, we summarize six categories of common failures when identifying among all 39 areas. Figure 9 shows these failure cases with examples. In each example, the left side image is the query image, and the right side image is the selected reference image to localize the query image. Both (a) and (b) use a correct reference image. However, (a) contains a water machine and a coffee machine that are very close to each other and so are their labels. This situation is very sensitive to image localization accuracy. (b) shows that the user intends to identify the fridge, but a printer is occluded by the fridge and has a label that is closer to the image center. (c) shows two scenes that look very similar, and our system cannot differentiate them using their SURFs. In (d), the flowerpot is the target appliance, but it's barely captured by any image in the training video, so our system fails to find a correct reference image. (e) shows that the ceiling light does not have a sufficiently unique visual context, so our system can only find another ceiling light as the reference image. (f) shows the query image was identified as the light, but labeled incorrectly as the projector. These failures cases can be very useful to guide our future work on improving the success rates, and demonstrate that most failures are due to external factors rather than artifacts of the system itself.

4.3 Server Processing Time

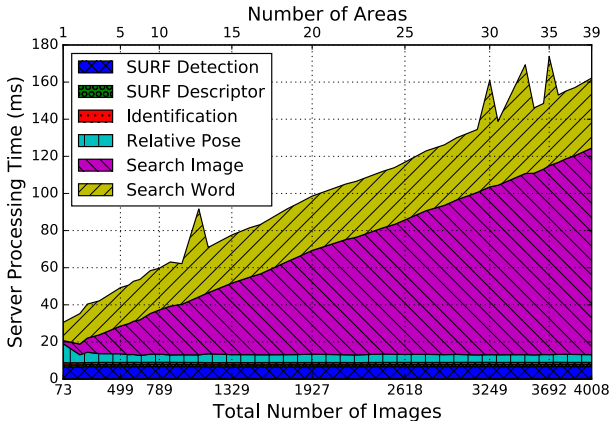


Figure 10: Server Processing Time vs. Total Number of Images

Figure 10 breaks up the server processing time with different numbers of images. As we can see, SURF detection, SURF descriptor, identification, and relative pose all require a nearly constant and small amount of time because they are one-time computations regardless of the number of images in the database. For the image search, we cannot use a kd-tree with our definition of similarity, as discussed in Section 3. Image search time thus increases linearly because the system is iterating over all images to compute similarities. However, even searching among all 4008 images from all 39 areas only takes 120 milliseconds, which still keeps the end-to-end latency imperceptible. Searching nearest words in the kd-tree increases slowly as expected and typically takes less than 40 ms. We notice there are spikes in kd-tree search time when certain combinations of areas are used. This is because the distribution of the words makes the the kd-tree unbalanced, which can induce the worst case $O(n)$ search time, even when we use four randomized kd-trees to mitigate the issue. Overall, when searching among ten areas (or 789 images), the total server processing time is about 60 ms.

4.4 Scalability with Concurrent Users

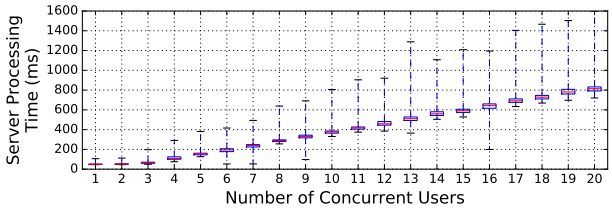


Figure 11: Computation Time vs. Number of Concurrent Users

The last two evaluations have shown CellMate's ability to scale with more data. Here we look at scalability with users. We run a CellMate server with data from one area and start several test scripts on the same machine that are each constantly sending query images. During this experiment, the success rates are not influenced by the number of concurrent users, so we only look at the server processing time.

Figure 11 shows a boxplot of server processing time with different numbers of concurrent clients. The boxes show the 25^{th}, 50^{th}, and 75^{th} percentiles of the times, and the lower whiskers and upper whiskers are the 1^{st} and 99^{th} percentiles respectively. The outliers are not shown for conciseness. As we can see, the average server processing time increases linearly as concurrent users increase, and we can support up to eight concurrent users to keep average server processing time less than our design goal of 300 ms as discussed in Section 2.3. In a typical commercial building deployment, we do not expect to have more than eight concurrent users, keeping latency reasonable.

Note that because we only run different stages of the pipeline in threads on one core, we are not yet exploring the benefits of pipelining or parallelization here. We can also see the distributions of latencies are concentrated because the boxes are short, which shows that the CellMate server is stable.

4.5 Area Image Density

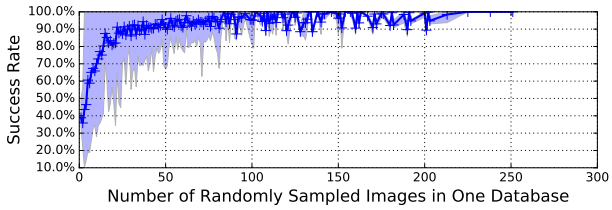
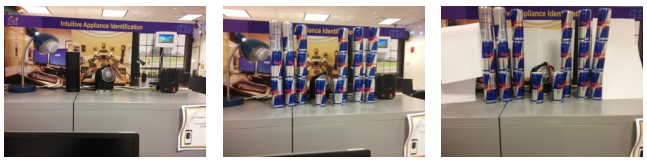


Figure 12: Success Rate vs. Number of Random Sampled Images in One Database

We explore the possibility of using fewer images to cover a given area to potentially further reduce image search time without losing accuracy. Our areas have various but similar enough physical sizes, and we ignore this factor to simplify the experiment without loss of generality. We modify the CellMate server so it initializes with only a subset of the images randomly sampled from a source video. For every area, we start with a 5% sub-sample rate, increase by 5% intervals, and observe the success rates. We perform four rounds of experiments for every area with all sub-sample rates.

Figure 12 shows the relationship between the success rate and the number of images we use for an area. The solid line shows the mean success rate for each number of images in the subsamples. The lower and upper edges of the filled area are the minimum and maximum accuracies we achieve in our experiment. As we can see, we can cover a typical room-size area with less than 80 images and get more than 90% average success rate. This inspires us to explore the trade-off between number of images (directly related to latency) and success rate, and we plan to work on mechanisms that are better than random selection in future work.

4.6 Robustness to Environmental Changes



(a) Original View of Fan (Center of Image) **(b) Fan with 20 cans (18 on side, 2 in front)** **(c) Fan with 20 cans and background blocked**

Figure 13: Examples of Changing Environment

As objects move and change all the time, it is very important that our system performance does not significantly degrade. To study CellMate's robustness to changing environments, we pick a small area from our dataset that is easier to make change. As shown in Figure 13a, this area contains three appliances: a lamp on the left, a fan in the middle, and a heater on the right, each of which has a decorative background image (also used as a unique visual context). We study the success rate identifying the fan while the environment changes.

We first add 18 beverage cans on the left and right sides of the fan, then add two cans in front of it such that it's partially blocked, and finally use six pieces of white paper to block all background images and the other appliances. We add these objects one by one and measure the success rates among all 4008 images from 39 databases using 10 test images from different angles and distances after every change. Figure 13 shows three example steps. The first image shows the original view of the fan, the second shows the fan with all 20 cans on both sides and in front, and the third shows all changes we add to the appliance.

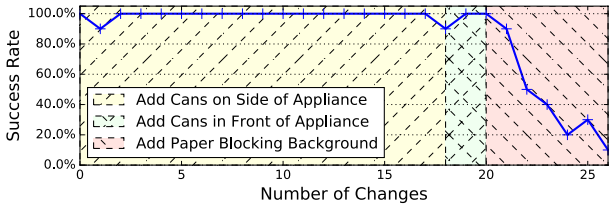


Figure 14: Success Rate with Number of Items Added around an Appliance

Figure 14 shows the success rate as changes are made to the appliance's environment. The three filled areas show the three types of change. As we can see, even though cans block some part of the image and add new SURFs, they don't have much influence on the success rate, no matter whether we

put them on the side or in front of the fan. This means visual context plays an important role in correctly identifying an appliance. As we add white paper to block more context information, the success rate starts to decrease, which is expected. This shows our system is robust to moderate changes of environment, meaning users do not need to frequently update areas with new videos.

4.7 App Energy Usage

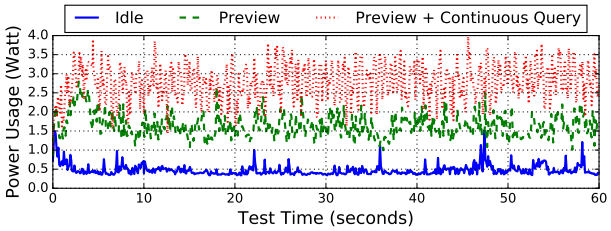


Figure 15: Energy Usage over Time

Our design goal is to make the CellMate mobile client power efficient. Because the LG G2 Mini uses the Qualcomm MSM8226 Snapdragon 400 SoC, we use the Qualcomm Trepn Power Profiler to measure accurate per-app power usage by leveraging specific Snapdragon features[3]. We adjust the screen brightness to 100% and measure three different power usages for 60 seconds: overall power usage while showing the home screen (idle), CellMate power usage while streaming camera view to screen (preview), and CellMate power usage while sequentially sending query images as fast as possible using WiFi. Figure 15 shows the the power usage trends, where idling consumes on average 0.48 Watts, preview consumes on average 1.64 Watts, and preview with continuous queries consumes on average 2.78 Watts. With the 3.8V 2440 mAh battery in the phone, the ideal battery life using CellMate to preview and continuously identify appliances is 2.84 hours ($\frac{(2.44\text{ Amp hour})*(3.8\text{ Volt})}{0.48+2.78\text{ Watt}}$).

5. DISCUSSION AND FUTURE WORK

As we build and deploy CellMate, we see some potential improvements that also put forward interesting research challenges.

Database Size Reduction: As we show in Section 4.5, we don't need all images from all videos to achieve a good identification success rate. Also, image search time increases linearly with the number of images. We plan to study this trade-off between success rate and image search time, and work on an algorithm to choose an optimal subset of images from all crowdsourced videos.

Database Coverage Improvement: Besides shrinking the dataset, we also find it important to have better coverage of appliances' appearances from different angles and distances. However, current systems rely on users to capture more images for better coverage. It's not clear how to quantify coverage or how much coverage is sufficient for general usage. In future work, we plan to come up with a metric for coverage and provide feedback to users to collect data with better quality.

Online Dataset Update: For areas that have changed a lot over time, users need to upload a new video to guarantee high success rate. Another approach is to use query images to update the dataset at runtime. However, this involves multiple challenges. For example, we need to know when to delete an old image and which labels can or cannot be transferred to new images.

Labeling Tool Improvement: Besides data collection, labeling is the aspect of CellMate that requires human intervention. To minimize human effort, we have a labeling tool that enables one simple click-and-type for every appliance. We plan to push this further with more intuitive and automatic mechanisms. For example, we can make the user click on the mobile app to label, and use convolutional neural network based object recognition to propose an initial guess for the labels in an image.

Screen and Whiteboard Handling: Screens and whiteboards are objects that change much more frequently than others. We did not design our system to cope with these objects. In future work, we may need to recognize them first and ignore their visual content for further processing.

6. RELATED WORK

Much work has been done to enable easier interaction between humans and appliances. As we discussed in Section 1, many of these solutions require extra infrastructure, such as laser pointers [7], infrared transceivers [22], or face-mounted eye-tracking cameras [2]. Other works use of commodity smartphone sensors but have different limitations. For example, reading fiducial markers [18] and signal strength based indoor localization [9] both fail to identify appliances at a distance. Moreover, natural language processing based approaches suffer from cumbersome and ambiguous vocal commands.

In comparison to these, visual information is more intuitive, straightforward, and convenient. Heun et al. [5] build an AR interface for smart devices, but rely on self-defined markers for image localization. Mayer et al. [12] and Jain et al. [6] collected a set of images and manually labeled all the images, whereas we only need 179 labels for the 179 appliances among 4008 images. When the camera view can match a labeled image, they display its label as relevant textual information, which doesn't require the same localization accuracy as we do. [5] and [12] have no evaluation on accuracy, latency, or scalability. [6] relies on continuous user video and sensor streams to narrow down the search space, and gets less than 90% success rates 30% of the time (average value not available) among 200 labeled images with 180 ms server computation time using a higher-end GPU (Titan Black). In comparison, we get average 96% success rate among 789 images with 60 ms server computation time.

iMoon [4] uses WiFi signal strength based indoor localization to reduce the search space and localizes a continuous stream of camera views for navigation. In the evaluation, they localize 2220 locations using 2197 images, where each location contains 12 images take from a 360° panorama view at 30° intervals. This significantly simplifies the problem because only one of the 12 images needs to be correctly localized. With the help of WiFi signal strength based localization, the system has more than 2 meters in error with more than 4 seconds of latency using both a higher-end CPU (Xeon E5-2650) and a higher-end GPU (Tesla K20C). Although we don't have ground truth 3D positions, our system must have much lower error to achieve more than 90% success rate identifying appliances that are tens of centimeters

[3]https://developer.qualcomm.com/software/trepn-power-profiler

in size while using only a single image among a dataset of around 2200 images.

7. CONCLUSION

In this paper, we presented an accurate and responsive vision-based appliance identification system called CellMate. It crowdsources user-captured videos and provides a labeling tool to easily label every appliance only once among all images containing that appliance. To optimize runtime responsiveness, CellMate decomposes the identification process into to a five stage pipeline and incorporates state-of-the-art computer vision data structures and algorithms. It is tested using 4008 model images and 1526 test images, and can achieve a success rate of more than 96% with 144 ms of client side latency if we use existing indoor localization techniques to narrow down the search space. CellMate forms a significant step in enabling users to interact with their environments using computer vision technologies in a more intuitive and accurate way. As future work, we plan to improve the performance and deploy the system at a larger scale.

Acknowledgements

This work is supported in part by the National Science Foundation under grant CPS-1239552 (SDB).

8. REFERENCES

[1] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[2] A. Bulling, D. Roggen, G. Tröster, G. Tröster, and G. Tröster. *Wearable EOG goggles: Seamless sensing and context-awareness in everyday environments*. ETH, Eidgenössische Technische Hochschule Zürich, Wearable Computing Laboratory, 2009.

[3] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. smap: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 197–210. ACM, 2010.

[4] J. Dong, Y. Xiao, M. Noreikis, Z. Ou, and A. Ylä-Jääski. imoon: Using smartphones for image-based indoor navigation. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 85–97. ACM, 2015.

[5] V. Heun, S. Kasahara, and P. Maes. Smarter objects: using ar technology to program physical objects and their interactions. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 961–966. ACM, 2013.

[6] P. Jain, J. Manweiler, and R. Roy Choudhury. Overlay: Practical mobile augmented reality. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 331–344. ACM, 2015.

[7] C. C. Kemp, C. D. Anderson, H. Nguyen, A. J. Trevor, and Z. Xu. A point-and-click interface for the real world: laser designation of objects for mobile manipulation. In *Human-Robot Interaction (HRI), 2008 3rd ACM/IEEE International Conference on*, pages 241–248. IEEE, 2008.

[8] M. Labbe and F. Michaud. Appearance-based loop closure detection for online large-scale and long-term operation. *Robotics, IEEE Transactions on*, 29(3):734–745, 2013.

[9] J. Lifton, M. Mittal, M. Lapinski, and J. A. Paradiso. Tricorder: A mobile sensor network browser. In *Proceedings of the ACM CHI 2007 Conference-Mobile Spatial Interaction Workshop*, 2007.

[10] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[11] D. Lymberopoulos, J. Liu, X. Yang, R. R. Choudhury, V. Handziski, and S. Sen. A realistic evaluation and comparison of indoor location technologies: experiences and lessons learned. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 178–189. ACM, 2015.

[12] S. Mayer, M. Schalch, M. George, and G. Sörös. Device recognition for intuitive interaction with the web of things. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 239–242. ACM, 2013.

[13] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[14] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[15] S. N. Patel and G. D. Abowd. A 2-way laser-assisted selection scheme for handhelds in a physical environment. In *UbiComp 2003: Ubiquitous Computing*, pages 200–207. Springer, 2003.

[16] B. Pieters, J. De Cock, C. Hollemeersch, J. Wielandt, P. Lambert, and R. Van de Walle. Ultra high definition video decoding with motion jpeg xr using the gpu. In *Image Processing, 2011 18th IEEE International Conference on*, pages 377–380. IEEE, 2011.

[17] L. Quan and Z. Lan. Linear n-point camera pose determination. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 21(8):774–780, 1999.

[18] J. Rekimoto and Y. Ayatsuka. Cybercode: designing augmented reality environments with visual tags. In *Proceedings of DARE 2000 on Designing augmented reality environments*, pages 1–10. ACM, 2000.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[20] D. Santa-Cruz, R. Grosbois, and T. Ebrahimi. Jpeg 2000 performance evaluation and assessment. *Signal Processing: Image Communication*, 17(1):113–130, 2002.

[21] J.-t. Wang, C.-N. Shyi, T.-W. Hou, and C. Fong. Design and implementation of augmented reality system collaborating with qr code. In *Computer Symposium (ICS), 2010 International*, pages 414–418. IEEE, 2010.

[22] B. Zhang, Y.-H. Chen, C. Tuna, A. Dave, Y. Li, E. Lee, and B. Hartmann. Hobs: head orientation-based selection in physical spaces. In *Proceedings of the 2nd ACM symposium on Spatial user interaction*, pages 17–25. ACM, 2014.