Real-Time Screen-Camera Communication Behind Any Scene

Tianxing Li, Chuankai An, Xinran Xiao, Andrew T. Campbell, and Xia Zhou
Department of Computer Science, Dartmouth College, Hanover, NH
{tianxing, chuankai, campbell, xia}@cs.dartmouth.edu xinranxiao@gmail.com

ABSTRACT

We present HiLight, a new form of real-time screen-camera communication without showing any coded images (e.g., barcodes) for off-the-shelf smart devices. HiLight encodes data into pixel translucency change atop *any* screen content, so that camera-equipped devices can fetch the data by turning their cameras to the screen. HiLight leverages the alpha channel, a well-known concept in computer graphics, to encode bits into the pixel translucency change. By removing the need to directly modify pixel RGB values, HiLight overcomes the key bottleneck of existing designs and enables real-time unobtrusive communication while supporting any screen content. We build a HiLight prototype using off-the-shelf smart devices and demonstrate its efficacy and robustness in practical settings. By offering an unobtrusive, flexible, and lightweight communication channel between screens and cameras, HiLight opens up opportunities for new HCI and context-aware applications, e.g., smart glasses communicating with screens to realize augmented reality.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless communication

Keywords

Screen-camera communication; visible light communication; alpha channel

1. INTRODUCTION

In a world of ever-increasing smart devices equipped with screens and cameras, enabling screens and cameras to communicate has been attracting growing interests. The idea is simple: information is encoded into a visual frame shown on a screen, and any camera-equipped device can turn to the screen and immediately fetch the information. Operating on the visible light spectrum band, screen-camera communication is free of electromagnetic interference, offering a promising out-of-band communication alternative for short-range information acquisition. The most popular example is QR code [6], where information (typically a URL) is encoded into a 2D barcode. Recent research endeavors have led to innovative barcode designs that boost the data rate [14, 26] or enhance the transmission reliability [15, 16, 28, 36].

These efforts are exciting; however, they commonly require displaying visible coded images, which interfere with the content the screen is playing and create unpleasant viewing experiences. In this paper, we seek approaches to enable unobtrusive screen-camera communication, which allows the screen to concurrently fulfill a dual role: displaying content and communication. Ultimately, we envision a screen-camera communication system that transmits and receives dynamic data in real time, while ensuring communication occurs unobtrusively regardless of the content the screen is displaying — let it be an image, a movie, a video clip, a web page, a game interface, or any other application window. As the user interacts with the screen and switches the content, the communication sustains. Hence, communication is truly realized as an additional functionality for the screen, without placing any constraints on the screen's original functionality (displaying content).

We are not alone in working towards this vision. Recent efforts have made valuable progress on designing unobtrusive screen-camera communication [10, 37, 39, 41, 42]. However, a fundamental gap to the vision remains, mainly because existing designs all require direct modifications of the pixel color (RGB) values of the screen content. This methodology cannot enable real-time unobtrusive communication atop arbitrary screen content that can be generated on the fly with user interactions. The reason is two-fold. First, modifying pixel RGB values in real time has to rely on the Graphics Processing Unit (GPU) by directly leveraging related GPU libraries. However, the operating system typically does not allow a third-party application to pre-fetch or modify the screen content of other applications or system interfaces (e.g., the home screen of a smartphone or tablet). Thus, to achieve real-time communication, existing designs are limited to screen content within a single application or standalone files (e.g., image, video). Second, although the main processor (CPU) can pre-fetch and modify arbitrary pixel RGB values, modifying RGB values at the CPU incurs a significant delay (hundreds of milliseconds), which makes these designs unable to support real-time communication atop dynamic content such as video.

To overcome this barrier, we propose a new design paradigm for unobtrusive screen-camera communication, which *decouples* communication and screen content image layers. Our key idea is to create a separate image layer (a black matte, fully transparent by default) dedicated to communication atop all existing content image layers. We encode data into the pixel translucency change at the separate image layer, so that the receiver (camera) can perceive the color intensity change in the composite image and decode data.

MobiSys'15, May 18–22, 2015, Florence, Italy.

http://dx.doi.org/10.1145/2742647.2742667.

To control pixel translucency, we leverage the alpha channel [27], a well-known concept in computer graphics. We control the level of pixel translucency change so that it is perceivable only to cameras but not human eyes. Furthermore, by controlling the translucency of each pixel independently, we enable multiple simultaneous transmitter elements on the screen. This creates a MIMO communication channel between screens and cameras [7], which can further boost data rate or improve transmission reliability.

Our methodology has two key benefits. *First*, operating on the alpha channel, the system no longer needs to directly modify pixel RGB values while achieving the same effect. Since alpha values are blended by the GPU, the encoding is almost instantaneous, which is critical to support real-time communication atop arbitrary dynamic content. More importantly, since alpha blending is a mature feature offered by the GPU, existing platforms (e.g., Android, iOS) provide application-layer functions that use related GPU APIs to modify pixel alpha values. Thus, the CPU can call these functions at the application layer without directly dealing with GPU APIs or modifying the OS or low-level drivers. *Second*, by realizing communication at a separate image layer, the system makes unobtrusive communication universally available and truly parallel to the content playing on the screen – no matter if it is a static, dynamic, or multi-layer content, and regardless of the frame rate it is playing at and the frame resolution. Users can use the screen as it is, while the communication between the screen and the camera occurs behind the scene in real time, unobtrusively.

Following our methodology, we design and build HiLight[1], the first system that supports real-time, unobtrusive screen-camera communication atop arbitrary screen content using off-the-shelf smart devices. We address the following specific design challenges. *First,* when determining the level of pixel translucency change, we face a tradeoff between user viewing experiences and transmission reliability. HiLight seeks the best tradeoff by adapting the translucency change based on the color distribution of screen content and the frame transition. It smooths the pixel translucency change across the screen while ensuring the change is reliably detectable by the receiver. *Second*, the whole encoding has to finish within a tight time limit (16 ms) to support video's normal playing speed and avoid the flicker effect [9]. We design lightweight encoding algorithms that require only sampled color information of sampled content frames. This reduces the overhead of pre-fetching content frames and ensures that data encoding and decoding can be finished within 8 ms. *Third*, the receiver perceives a mixture of color intensity change caused not only by encoded translucency change, but also by other interfering sources such as screen content change, ambient light, and camera noise. We design efficient algorithms to extract the desired color intensity change, and adapt our strategy to the scene type. This design ensures the system robustness under diverse practical settings.

We evaluate the HiLight prototype with an Samsung Tab S tablet as the transmitter and an iPhone 5s as the receiver. We test HiLight using 112 images, 60 video clips, and various web browsing and gaming scenarios. Our key findings are as follows:

- HiLight supports unobtrusive communication atop *arbitrary* screen content as the user is interacting with the screen, achieving 1.1 Kbps throughput with 84 – 91%+ accuracy for all scene types.

- HiLight realizes real-time data processing using off-the-shelf smart devices. Its encoding and decoding delays are both within 8.3 ms even for high-resolution video frames (1080p), sufficient to support video playing at 60 FPS and 120 FPS frame rates.

- HiLight is robust across diverse practical settings with hand motion and perspective distortion, supporting up to 1-meter viewing distance and 60° viewing angle for the 10.5-inch transmitter screen and maintaining stable performance once the ambient light is above 40 lux in indoor environments.

[1]We have made the demo video clips of HiLight available at `http://dartnets.cs.dartmouth.edu/hilight`.

Contributions. We summarize our key contributions as follows:

- We analyze existing unobtrusive screen-camera communication designs using detailed measurements and identify their limitations to enable real-time unobtrusive communication atop any screen content.

- We propose a new design paradigm for unobtrusive screen-camera communication, which decouples communication from the screen content and uses the alpha channel to encode data into pixel translucency change [21]. Since alpha values are blended by the GPU, data encoding is almost instantaneous, which is the key to enabling real-time communication atop arbitrary content.

- We design and build HiLight using off-the-shelf smart devices, the first system that realizes on-demand data transmissions in real time unobtrusively atop *arbitrary* screen content. We extensively evaluate our prototype under diverse practical settings, examine its potential performance on a wide range of devices (e.g., smartphones, tablets, laptops, and high-end cameras), and assess its user perception.

By offering an unobtrusive, flexible, and lightweight communication channel between screens and cameras, HiLight presents opportunities for new HCI and context-aware applications for smart devices. For example, a user wearing smart glasses can acquire additional personalized information from the screen (e.g., TV screen, smartphone, or tablet screen) without affecting the content users are currently viewing. HiLight also provides far-reaching implications for facilitating new security and graphics applications.

2. SCREEN-CAMERA COMMUNICATION: LIMITATIONS AND SOLUTION

In this section, we first present the design goals for a universal screen-camera communication system. We then analyze the inefficacy of existing designs to achieve these design goals. Finally we introduce our new methodology.

2.1 Design Goals

We bear in mind the following goals when designing a universal screen-camera communication system.

Staying Unobtrusive to Users. The screen-camera data communication is completely hidden from users and does not require showing any visible coded images on the screen. Hence, data communication does not interfere with any content that the user is viewing at the screen, and can be easily embedded into any existing screens (e.g., smartphone screens, laptop screens) without sacrificing their original functionality (i.e., displaying content).

Supporting *Any* Scene. The data communication can occur regardless of the screen content, let it be an image, a video clip, a movie, a gaming scene, a multi-layer application windows, or the home screen of a smartphone or tablet. Furthermore, the communication continues as the screen content changes on the fly (e.g., the

user browses a web page, plays games, and watches a movie). To support all types of screen content, the communication needs to be independent of the screen content, which can be generated on the fly and vary over time as the user interacts with the screen.

Processing Dynamic Data in Real Time. The system can transmit and receive dynamic data on the fly, rather than pre-processing an image or a video file to embed data. This is necessary when the data comes on the fly or when the screen content is not known in advance (e.g., interactive gaming scene). The challenge is that when the screen is displaying video (e.g., movies, video clips), the frame rate is at least 24 frames per second (FPS). Thus, the system has to finish encoding data into a frame within 42 ms before the screen displays the next frame. When the video is played at 60 FPS or 120 FPS, the encoding has to finish within 16 ms or 8 ms.

Operating on Off-the-Shelf Smart Devices. The system can transmit and receive data in real time using existing smart devices (e.g., smartphones, tablets). These devices have limited computational power, and their cameras are not as sophisticated as high-end single-lens reflex (SLR) cameras. Hence the encoding and decoding designs have to be lightweight and robust, so that smart devices as transmitters and receivers can process data in time while supporting dynamic content such as video.

Next, we analyze existing proposals on unobtrusive screen-camera communication and examine whether they can achieve the above goals.

2.2 Current Limitations

Researchers have proposed designs [10, 37, 39, 41, 42] to remove the need of showing visible coded images while enabling screen-camera communication. Despite the tangible benefits of existing designs, they are unable to achieve all the design goals described in § 2.1, mainly because they directly deal with the color values of each pixel, specified as the color intensity values in red, green, and blue (RGB) diodes [34]. To ensure that RGB value changes are unnoticeable to users, they have to pre-fetch the RGB values of all content pixels to subtly modify RGB values and encode data. As a result, existing designs cannot serve as a unified solution that can achieve real-time data communication decoupled from the screen content (e.g., the home screen of a smartphone, the interface windows of other independent applications). Next, we analyze the two possible approaches to modifying pixel RGB values using existing designs and identify their limitations.

GPU-Based RGB Modifications. To enable real-time modifications of pixel RGB values, one has to rely on GPU's parallel processing power and leverage GPU-related libraries (e.g., OpenGL [1] on the Android and iOS). The key idea is to supply GPU with two programs, known as *shaders* or *kernels*, which are executed independently per pixel. Modern shaders are fast. Based on our experiments on a Google Nexus 6 (with Android 5.0 OS) phone, it takes only 0.13 ms to modify RGB values of all the pixels in a single video frame (1920 x 1080 pixels). This approach, however, has two limitations. *First*, it is limited to modifying screen content within a single application or a standalone file (e.g., a video or image file). Because of the system security model of existing platforms (e.g, Android), third-party applications cannot pre-fetch screen content from other applications or system interfaces (e.g., home screen). *Second*, by replacing the original screen content with an opaque, modified new content layer, the modifications can nullify the existing rendering optimization. For example, consider the user is browsing a web page in a browser. To enable data communication, we have to run an OpenGL application and display the modified page through this OpenGL app. This additional step can remove the rendering optimization made by the web browser. Therefore, although existing designs can leverage GPU programming to enable real-time unobtrusive communication, they are limited to the screen content within a single application or static standalone files and are unable to support all dynamic screen content.

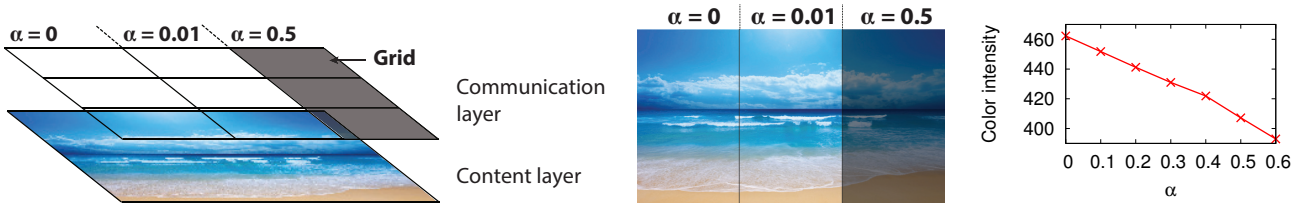
CPU-Based RGB Modifications. Another approach to modifying pixel RGB values is to use the CPU to pre-fetch pixel RGB values of the current screen content. Although this approach allows the system to enable unobtrusive communication atop any scene, it entails long encoding delay and fails to support real-time data communication. We further quantify the processing time using off-the-shelf smart devices (Nexus 4, Nexus 5, Samsung Note 3, Samsung S5, Samsung Tab S, all with Android 4.4). To capture the screen content on an Android device, we read the frame buffer under the `/dev/graphics/` directory. We leverage the OpenCV library [2], a popular computer vision library, to calculate the complementary RGB values for each pixel. This calculation is a representative of the RGB modification step in all existing designs [10, 37, 39, 41, 42]. We then use the Simple Direct Layer (SDL) library to render an image on the screen. We choose SDL library because it is much more flexible than the rendering library [3] in the Android framework. For all smart devices we tested, the processing time of data encoding far exceeds 42 ms, the timing threshold (§ 2.1) required to support real-time screen-camera data transmission. Especially for high-resolution (1080p) video, CPU-based RGB modifications need 1–2 s to embed data into a frame. While the encoding time slightly drops as the frame resolution decreases, it never drops below 300–600 ms.[2] Note that these steps are the minimal for embedding data into the RGB values of the screen content frame. Some designs require additional steps (e.g., copying frames [37]), leading to even longer delays. To speed up encoding, one can modify only a subset of pixels. Yet this reduces the number of pixels used to transmit data, sacrificing either the data rate or the transmission reliability. More importantly, so far we have assumed a single image layer. For content with multiple image layers, the cost of pre-fetching and modifying pixel RGB values grows linearly with the number of image layers. Thus, CPU-based RGB modifications cannot achieve real-time data communication.

Summary of Observations. To sum up, existing unobtrusive screen-camera communication designs require direct modifications of content pixel color (RGB) values and fail to transmit dynamic data atop arbitrary dynamic content, where either data or screen content comes on the fly. Furthermore, by creating an opaque content layer with modified RGB values, these designs can nullify the existing rendering optimization. To mitigate the problem, current designs will have to give up whole-screen communication and sacrifice either the data rate or the transmission reliability. In addition, integrating communication with screen content greatly limits their applicable scenarios.

2.3 Solution: Decoupled Communication Using Alpha Channel

To overcome the above limitations, we propose a new design paradigm for unobtrusive screen-camera communication. Motivated by the fact that the tight integration of communication and screen content is problematic, we propose to decouple screen-camera communication from the image layers of screen content. Furthermore, we design lightweight data encoding that leverages the power

[2]We also test devices (e.g., Samsung S5, Nexus 5) with the recent Android 5.0, which provides a new screen capture API to speed up the process of pre-fetching screen content. Our results show that CPU-based RGB modifications still take more than 200 ms.



(a) Overview (b) Resulting composite image (c) Pixel color intensity perceived by a camera (receiver)

Figure 1: Decoupling screen-camera communication from the content image layer on the screen. (a) Atop the content image layer, we create an additional image layer (a black matte, fully transparent by default) dedicated to data communication, referred to as the *communication layer*. To transmit data, we divide the communication layer into grids, and encode data into pixel translucency change of each grid without affecting users' viewing experiences. (b) shows the resulting composite image after modifying the pixel translucency of the communication layer. (c) Camera-equipped devices perceive color intensity change as α value increases.

of GPU and does not require direct modifications of content RGB values. More importantly, since alpha blending is a mature feature offered by the GPU, alpha values can be modified by calling a system function at the application layer without any OS-level modifications (see more in § 5). At the high level, our design principle is two-fold:

- We create a separate image layer (a black matte, fully transparent by default) dedicated to communication, referred to as the *communication layer*, atop all content image layers (Figure 1(a)). By *decoupling* communication from screen content, we allow the screen to better fulfill its dual role (communication and displaying content).
- We encode data by changing the pixel transparency of the communication layer. This removes the need to directly modify pixel RGB values while achieving the same effect. In particular, we leverage *alpha channel*, a well-known concept in computer graphics, to control pixel transparency and encode data [21]. Since alpha values are blended by GPU (< 1 ms), our design significantly drives down the encoding delay, which is the key to support real-time, unobtrusive communication atop any screen content.

Next, we begin with the background on alpha channel and how we utilize it to transmit data. We then elaborate the key benefits of our methodology and design challenges.

Communication Using Alpha Channel. As a well-known concept in computer graphics, alpha channel is widely used to form a composite image with partial or full transparency [27]. It stores a value α between 0 and 1 to indicate the pixel translucency. 0 means that the pixel is fully transparent, and 1 means that it is fully opaque. By default, we set $\alpha = 0$ for all pixels in the communication layer to make it fully transparent and not interfere with the screen content. As we increase the α values of the communication layer (Figure 1(a)), we change how users perceive the color of the content layer. Specifically, when $\alpha = 1$, the pixel of the top communication layer is opaque, hence users perceive this pixel as black, the color of the communication layer. Thus we can essentially dim an area by increasing the α values of this area on the communication layer. As an example, Figure 1(b) shows the composite image on the screen, where we set pixel α values of the communication layer to 0 (left), 0.01 (middle), and 0.5 (right), illustrated in Figure 1(a). A higher α leads to a darker appearance in the composite image.

This dimming effect is perceived as color intensity (summation of RGB channel values) change by a camera. We further examine how cameras on existing smart devices perceive color intensity change. We set up a Samsung Note 3 phone as the transmitter, and a Samsung S5 phone 15 cm away as the receiver. We create a top black image layer in Note 3, adjust pixel α values of this top image layer uniformly, and measure the color intensity of each pixel captured by the S5's camera. Figure 1(c) shows the color intensity averaged over all pixels as α increases. It confirms that increasing α value on the top image layer linearly decreases the perceived color intensity. We also observe that even for the α change of 0.01, imperceptible to human eyes (Figure 1(b)), the S5's camera can still detect the difference.

Motivated by the above observations, we encode data into pixel translucency (α) change on the top communication layer. For today's smart devices, the screen refresh rate is typically 60 Hz. Therefore, to ensure the change imperceptible to human eyes, we change α values by only 1%–8%, sufficient to avoid the flicker effect[3] [9]. Since the change can be perceived by cameras, camera-equipped devices can examine the perceived color intensity change in captured frames and decode data. Furthermore, we can divide the communication layer into grids and change the α values of the pixels in each grid independently. *Hence each grid functions as an independent transmitter element, resulting into a MIMO channel between screens and cameras.*

Note that the same principle holds when using a white matte as the communication layer, where increasing the α values of an area in the communication layer brightens this area in the composite image. However, black and white are the only two colors feasible for realizing alpha channel based communication. The reason is as follows. In general, assume the communication layer's color intensity in a color channel (i.e., red, green, or blue channel) is C_1 with the alpha value of α. Based on the alpha blending principle, for a pixel in the content layer with color intensity C_2 in this color channel, this pixel's color intensity C in this color channel in the final composite image is calculated as $C = \alpha \cdot C_1 + (1 - \alpha) \cdot C_2$, where $C_1, C_2 \in [0, 255]$ and $\alpha \in [0, 1]$. Hence setting C_1 to the minimal (0) or the maximal (255) in all color channels, which corresponds to the black or the white color, always dims or brightens a pixel in the composite image, regardless of the pixel's original color (C_2), the key for encoding data into the pixel translucency change atop any screen content. In contrast, setting C_1 to any value within (0, 255) in a color channel does not uniformly dim or brighten pixels in all colors and thus is unable to enable communication atop any pixel color of the content layer.

[3]The occurrence of the flicker effect depends on the screen refresh rate and the degree of change across frames.

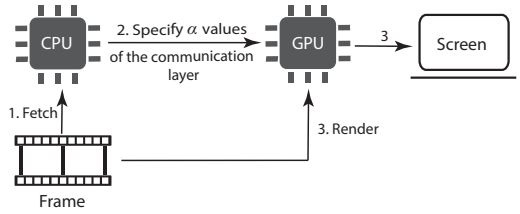


Figure 2: Flowchart of modifying alpha values. The CPU passes specified α values to GPU using a system function at the application layer. The GPU then applies the alpha blending technique to combine α values with an image frame.

Benefits. Our methodology offers three benefits. *First*, operating on the alpha channel significantly reduces the encoding time. Figure 2 shows the procedure for the system to process a frame using alpha-channel communication. The main processor passes specified α values of the communication layer to the GPU, which blends them with the content layer using the alpha blending technique [27]. The blending can be finished in 1 ms, several orders of magnitude faster than modifying pixel RGB values on CPU. *Second*, since we enable communication using a separate image layer (communication layer), communication can occur regardless of the screen content, its frame rate, and the number of content image layers. Thus our design allows the screen-camera communication channel to support arbitrary scene, including a multi-layer scene (e.g., Angry Birds) that needs to modify the α values of the content layer. Also, since data encoding is in the separate image layer, we can parallelize data encoding and content playing and further reduce encoding delay. *Third*, by adding a transparent communication layer and slightly adjusting its translucency, our design does not interfere with the GPU rendering optimization for the underlying image layers while achieving the same effect of directly modifying pixel RGB values.

Challenges. To implement our methodology, we face three key challenges. *First*, determining the level of translucency change is nontrivial. The change has to be unobtrusive to human eyes while reliably detectable by cameras. Also, α value changes RGB values by a percentage. Thus for darker colors, the absolute color intensity change is smaller and harder to detect. *Second*, when transmitting dynamic data atop a video, all encoding steps need to finish within 42 ms to support video's normal playing speed. Furthermore, to reduce the flicker effect [9], the frequency of the translucency change at the communication layer has to match the screen's refresh rate (60 Hz). Hence the encoding needs to be lightweight to finish within 16 ms, and can operate on smart devices. *Third*, for a dynamic scene, the screen content changes over time. The resulting color intensity change interferes with our encoded color intensity change. Even for static content, environmental factors such as ambient light and camera noise can affect decoding. All these interfering sources make it challenging to extract data from the perceived color intensity change.

We address the above challenges and design HiLight. In the next two sections, we describe in detail our design of HiLight on both the transmitter and the receiver.

3. HiLight TRANSMITTER

HiLight transmitter encodes data into the pixel translucency (α value) change at the communication layer. The translucency change is fixed at the screen's refresh rate (60 Hz), independent of the content frame rate that can change at any time. We face two design challenges. *First*, it is nontrivial to determine the α value change ($\Delta\alpha$) for each pixel. On one hand, $\Delta\alpha$ should stay minimal to be imperceptible to users. On the other hand, a larger $\Delta\alpha$ means a more detectable change in the perceived color intensity at the receiver. Hence we need to seek the best tradeoff between user experience and communication reliability. *Second*, the per-frame encoding needs to finish within 16 ms to support video's normal playing speed and avoid the flicker effect. While we remove the need to directly modify RGB values, the remaining steps for data encoding (fetching and rendering in Figure 2) can still take more than 16 ms. We need to further reduce the encoding delay.

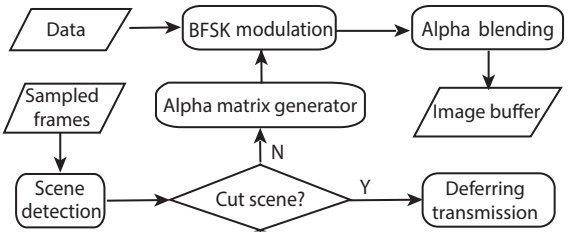


Figure 3: The encoding procedure of the HiLight transmitter.

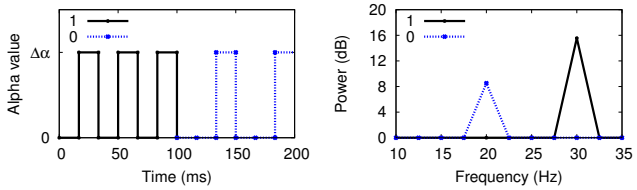
We address the challenges as follows. *First*, we set the $\Delta\alpha$ small (1–8%) to keep the translucency change unobtrusive to users. To ensure that the change can be reliably detected, we configure the $\Delta\alpha$ of each pixel based on the color distribution of the content frame and the transition across frames. The configuration helps receivers deal with dark areas or frames with drastic content change. *Second*, to further reduce the data encoding delay, we fetch only one content frame per sampling window (0.1 s in our implementation) and sample a subset of pixels for their color values. This is not doable for existing designs to achieve whole-screen communication. Since existing designs rely on directly modifying content color values, they have to fetch every frame and obtain color values of all pixels in a frame.

Next, we first overview the overall encoding procedure, and then focus on two main design components in detail.

3.1 Overview

Figure 3 overviews the encoding procedure. We sample content image frames to configure $\Delta\alpha$ for each pixel at the communication layer. Note that the content frame information only helps optimize the $\Delta\alpha$ configuration and is not an enabler for the communication. Unlike current designs that require modifying content image frames, we only need to read (not write) coarse-grained information of content image frames. Given two adjacent sampled frames, the transmitter examines their color transition to adapt the encoding strategy. If the frame content changes drastically, the resulting color intensity change can overwhelm the intensity change encoded with data. We refer to these frames as *cut scene* frames, which typically occur rarely (1.3% in a test video stream in Figure 5(d)). We skip the cut-scene frames and defer encoding. Once detecting a non-cut scene frame, we determine the $\Delta\alpha$ based on the color brightness of the area in the content layer and the frame transition.

Given the $\Delta\alpha$ for each pixel, the transmitter modulates bits using Binary Frequency Shift Keying (BFSK). We choose BFSK because of its simplicity and its robustness to interference (e.g., ambient light). In our implementation, bit 0 and 1 are represented by translucency change at 20 Hz and 30 Hz respectively, over six frames (Figure 4). We set six frames as a frame window, because it is the minimal to achieve 20 Hz and 30 Hz change under the screen refresh rate of 60 Hz. By removing the low-frequency components



(a) Time domain (b) Frequency domain

Figure 4: Encoding bits into pixel translucency (α) change. (a) shows the sequence of α values for symbols '01'. (b) shows the translucency change in the frequency domain for bit 0 and 1 after applying FFT.

($<$ 15Hz), the frequency power of the data bit is less affected by environmental noise (e.g., ambient light). In comparison, modulation schemes (e.g., Manchester code) that rely on phase transition in the time domain to extract data can suffer performance degradation caused by environmental noise. In addition, since FSK encodes data as relative color intensity changes, HiLight is robust against image quality degradation (e.g., block boundary effects caused by video compression).

After generating the sequence of α values for each pixel, the main processor passes these values to GPU. The GPU then applies the alpha blending technique [27, 34] to generate image layers (communication layer) with specified translucency, combines them with content image frames, forms composite images, and outputs them to the image buffer.

Next we describe two main design components (scene detection and determining α value changes $\Delta\alpha$) in detail.

3.2 Scene Detection

Scene detection is the key component for the system to support arbitrary screen scene. It runs continuously in the background, samples content frames, and identifies the screen scene type based on the frame transition. The scene type is crucial for deciding whether the encoding should be performed, and for determining the degree of pixel translucency change at the communication layer (§ 3.3).

Specifically, the transmitter periodically samples content frames every 0.1 s. We choose 0.1 s as the sampling window because it is sufficient for detecting frame transition accurately while missing a small number of frames. For each sampled frame, the thread further uniformly samples one-sixteenth of the pixels and obtains their color information.

Given the sampled pixels of two sampled frames, we aim to categorize the current frame into one of the following scene types: *the static, the gradual, and the cut scene*. These categories capture the degree of frame transition, and are widely used in computer vision [17]. Figure 5 shows example frame pairs for each scene type. While screen content can change drastically over a long time, the change between two adjacent frames is gradual and minor. Therefore, static and gradual scenes are the most common, while cut scene occurs rarely (Figure 5(d)). Because of the drastic color intensity change of the cut scene and its rare occurrence, we encode data only atop the other two types of scenes.

To determine the scene type, we measure the dissimilarity of two adjacent sampled frames using two existing metrics. The first metric [44], referred to as the *pixel-based* metric, calculates the difference of the pixel color intensity values of two frames. For frames F and F' each with $X \times Y$ pixels, the pixel-based metric $d_p(F, F')$ is calculated as:

$$d_p(F, F') = \frac{\sum_{\substack{1 \le i \le X \\ 1 \le j \le Y}} \mid C(p_{ij}) - C(p'_{ij}) \mid}{X \times Y}, \qquad (1)$$

where $C(p_{ij})$ and $C(p'_{ij})$ are the color intensity values of pixels p_{ij} and p'_{ij} in frame F and F', respectively. The second metric [25, 31], referred to as the *histogram-based* metric, examines the difference in the color distribution of two frames. It partitions the whole color intensity range into bins, counts the number of pixels with color values within each bin, and calculates the difference as:

$$d_h(F, F') = \sum_{\substack{i \in [1,K] \\ \max(H(i), H'(i)) \neq 0}} \frac{(H(i) - H'(i))^2}{\max(H(i), H'(i))}, \qquad (2)$$

where $H(i)$ and $H'(i)$ are the number of pixels in the i-th bin for frame F and F' respectively, and K is the number of bins. We choose these two metrics because they are easy to compute, and complement each other. Pixel-based metric captures small local changes (e.g., a small moving object), while histogram-based metric captures the global change.

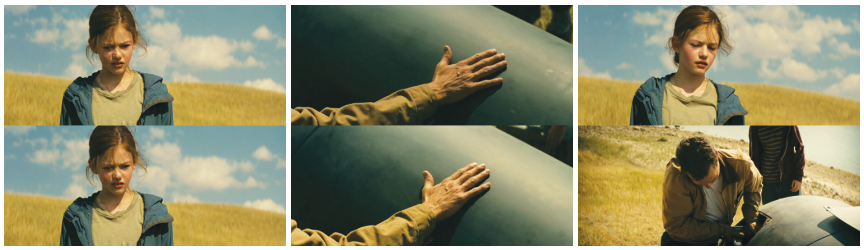
We combine these two metrics to classify a frame. For both metrics, a higher value indicates a more drastic frame transition. Hence we define two thresholds for each metric to signal a static and a cut scene. We denote the thresholds for the metric d_p as d_p^{static} and d_p^{cut}, and d_h^{static} and d_h^{cut} for the metric d_h. We classify a frame as a cut scene if the values of both metrics are above their cut scene thresholds, i.e., $d_p > d_p^{cut}$ and $d_h > d_h^{cut}$. We classify a frame as a static scene if both metric values are below the static scene thresholds, i.e., $d_p < d_p^{static}$ and $d_h < d_h^{static}$. A frame is a gradual scene if neither condition is satisfied. In our implementation, we empirically set d_p^{static}, d_p^{cut}, d_h^{static}, and d_h^{cut} as 10, 100, 0.1, and 1, respectively. These values also align with the literature in computer vision [29, 30, 32].

3.3 Determining Alpha Value Change ($\Delta\alpha$)

Armed with the content frame information (scene type, and frame color distribution), the second key component determines the degree of pixel translucency change at the communication layer. The goal is to keep the resulting color intensity change unobtrusive while ensuring they are still reliably detectable by receivers.

To achieve the goal, we configure the $\Delta\alpha$ of each pixel based on the color distribution of the content frame and frame transition. Our configuration is driven by two observations. *First*, $\Delta\alpha$ changes the pixel color intensity by a percentage. Thus, for a fixed $\Delta\alpha$, the resulting absolute color intensity change of dark pixels is smaller and less detectable than that of bright pixels [21]. *Second*, frame content change also leads to color intensity change, which interferes with the intensity change encoded with data. Motivated by these observations, we increase the $\Delta\alpha$ for pixels atop dark content areas or upon a gradual scene frame. This enhances the coded intensity change in these two challenging scenarios, allowing the receiver to better extract data.

Specifically, we determine the α value change $\Delta\alpha$ for each pixel in two steps (Algorithm 1). The first step is to divide the screen into grids and decide the $\Delta\alpha$ at the grid level. For each grid G_k, we calculate its average color intensity value $C(G_k)$ over all sampled pixels in G_k. We configure the grid-level α change $\Delta\alpha_G$ within $[1\%, 8\%]$ based on the grid color intensity $C(G_k)$, and the scene type. A side effect of this step is that for adjacent grids with different α values, the grid border appears as a noticeable edge. To minimize this side effect, the second step is to fine-tune the $\Delta\alpha$ for each pixel within a grid and diminish the α value difference



(a) Static scene ($d_p = 0.2$, $d_h = 0$) (b) Gradual scene ($d_p = 12.2$, $d_h = 0.2$) (c) Cut scene ($d_p = 216$, $d_h = 1.6$) (d) Frame scene type in a video stream

Figure 5: (a)-(c) Example image pairs of three scene types (the static, the gradual, and the cut scene). We evaluate the difference of each pair of frames using both the pixel-based metric d_p and the histogram-based metric d_h. (d) shows the scene type of each frame in a video stream.

Algorithm 1: Determine alpha value change ($\Delta\alpha$).

input : 1) F, sampled content frame; 2) S_T, scene type.
output: α value change $\Delta\alpha$ for each pixel.

Divide F into X grids: G_1, ..., G_X
for $k \leftarrow 1$ **to** X **do**
  // Computing grid-level $\Delta\alpha$
  $C(G_k) = \texttt{Mean}\{C(p_{ij}) \mid p_{ij} \in G_k\}$
  if $C(G_k) \leq 50$ **then** $\Delta\alpha_G = 4\%$
  else if $C(G_k) \leq 100$ **then** $\Delta\alpha_G = 3\%$
  else if $C(G_k) \leq 150$ **then** $\Delta\alpha_G = 2\%$
  else $\Delta\alpha_G = 1\%$
  if $S_T =$ gradual **then** $\Delta\alpha_G = 2 \cdot \Delta\alpha_G$
  // Fine-tuning $\Delta\alpha$ within a grid
  $(x_c, y_c) \leftarrow \texttt{FetchCenter}(G_k)$
  for $p_{ij} \in G_k$ **do**
    $\Delta\alpha_{ij} = \frac{\Delta\alpha_G}{sqrt(2\pi)} \times e^{-\frac{(i-x_c)^2+(j-y_c)^2}{2}}$
  end
end

among pixels on the grid border. In particular, for pixels in the center of the grid G_k, their $\Delta\alpha$ is the same as the grid-level α change $\Delta\alpha_G$. This ensures that the receiver can still differentiate translucency changes of adjacent grids. For pixels further away from the grid center, we gradually decrease their $\Delta\alpha$ following a standard normal distribution $\mathcal{N}(0, 1)$ (Figure 6). The fine-tuning smooths the translucency change across grids. Finally the $\Delta\alpha$ values for all pixels are fed into the modulator, which modulates each bit into translucency (α) change over time.

4. HiLight RECEIVER

HiLight receiver captures incoming frames, examines the perceived color intensity changes, and decodes data. The main design challenge is to extract the desired color intensity change out of the perceived change. This is challenging because the color intensity change perceived by the receiver comes from not only encoded α changes, but also changes in screen content, ambient light condition, as well as camera noises. The receiver needs to extract data from the perceived color intensity change given the presence of all these interfering factors.

To address the challenge, we design strategies to filter out color intensity change associated with interfering factors, and adapt our strategy to the current scene type. In particular, for the static scene, we leverage an audio beamforming algorithm [33, 35] to minimize the impact of interfering factors and extract the desired color intensity change encoded with data; for the dynamic scene, we identify

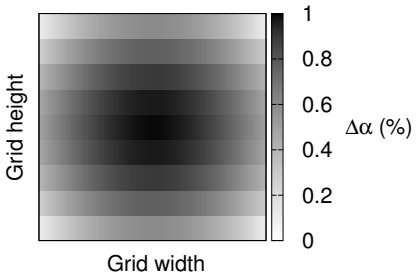


Figure 6: Configuring $\Delta\alpha$ for each pixel within a grid, assuming the grid-level α change is 1%. Here the darkness indicates the $\Delta\alpha$ value, not the actual pixel appearance. We gradually decrease $\Delta\alpha$ as pixels become further away from the grid center. This diminishes the appearance of the grid border and optimizes the viewing experience.

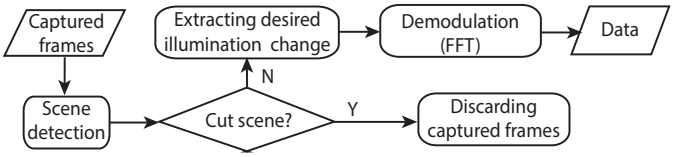


Figure 7: An overview of the decoding procedure in HiLight.

patterns caused by the encoded change. These patterns help recover the bits hidden in the mixed intensity change. Note that we integrate these scene-dependent strategies into a single framework, which allows the system to support arbitrary scenes in a unified solution.

Next we first overview the decoding procedure, followed by a detailed description on extracting the desired color intensity change.

4.1 Overview

Figure 7 summarizes the main steps of decoding. The receiver keeps monitoring the incoming frames, and buffers frames in a frame window (six frames). It then samples a frame in this frame window, compares it to the last sampled frame, and applies the same scene detection algorithm as that in the transmitter (§ 3.2) to identify the scene type. If the scene type is the cut scene, the receiver discards this frame window since the transmitter does not encode data on these frames. For frames of other scene types (i.e., the gradual or static scene), the receiver divides the captured transmitter screen into grids, identifies and enhances the desired color intensity change in each grid (§ 4.2).

The extracted color intensity change is then passed to a BFSK demodulator, which applies Fast Fourier Transform (FFT) to project translucency change into the frequency domain. To reduce the impact of ambient light noise with frequency components close to 0, the receiver filters out frequency components below 15 Hz. Among the frequency components above 15 Hz, the receiver identifies the component with the highest power, maps it to the corresponding bit, and outputs the bit into a candidate pool. The receiver then slides the frame window by one frame, and repeats the above procedure to generate another candidate bit. The final bit for this frame window is the bit with the most occurrences in the pool.

4.2 Extracting Desired Color Intensity Change

The receiver perceives mixed color intensity change caused by encoded α values, screen content, ambient light, and camera noise. Among them, the color intensity change associated with encoded α values is the desired change that the receiver aims to extract, and the rest are interfering sources. To minimize the impact of these interfering sources, our design is driven by the fact that these interfering sources have nonuniform impact across the screen, depending on the brightness of the area. Therefore, we can further divide a grid into smaller regions, evaluate the impact of interfering sources on each region, and assign higher weight to regions less affected by interfering sources. This allows us to reduce the impact of the interfering factors, and enhance the desired change associated with data. The key question then is to determine the weight of each region within a grid. An effective weight assignment should leverage the scene type of the screen content. Next we describe our strategy for each scene type (the static and gradual scene) in detail.

Static Scene. For the static scene, we are inspired by an audio beamforming algorithm called Minimum Variance Distortionless Response (MVDR) [33, 35]. This algorithm was originally designed to enhance audio signals in a desired direction while canceling out noise signals. The key idea is to leverage the correlation (e.g., delay spread and energy distribution in the frequency domain) across noise signals and cancel the noise energy.

We develop a variant of this algorithm in our context, where the goal is to enhance the contribution of regions less affected by interfering sources. Specifically, for a grid with N regions, we calculate the weight matrix for its regions as:

$$W = R^{-1}C \times R_{sum}, \tag{3}$$

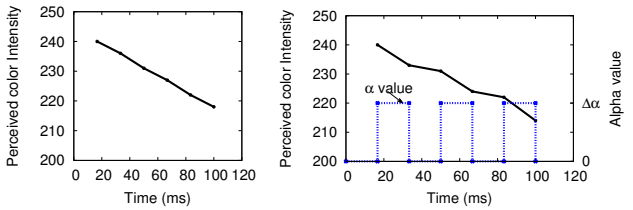
where R is a $N \times N$ noise correlation matrix capturing the correlation across regions of this grid, C is a $N \times 1$ constant matrix with all elements equal to 1, and R_{sum} is the summation of all elements in R^{-1}.

We obtain the noise correlation matrix R via a training process for a static scene. In particular, once the receiver detects a static scene, it buffers frames for M frame windows. It then applies FFT to compute the frequency components of each frame window, and estimates the correlation R_{ij} between region i and j as:

$$R_{ij} = \frac{1}{M} \times \sum_{1 \le k \le M} X_{ik} X_{jk}^{T},\ i, j \in [1, N], \tag{4}$$

where X_{ik} and X_{jk} are the vectors of frequency components of the kth frame window, for region i and j respectively. From our experiments, we find that $M = 5$ is sufficient to train the correlation matrix and cancel out the contributions of regions more affected by interfering sources. Thus the training takes 0.5 s overall.

Gradual Scene. When the screen content is dynamic, it is hard for training-based methods to catch up with the content change in real time. Thus we design a lightweight scheme without training to assign weights to regions within a grid.



(a) w/o encoding bit 1 (b) w/ encoding bit 1

Figure 8: Identifying patterns caused by encoded α values when examining the perceived color intensity changes over a frame window (0.1 s), assuming bit 1 is encoded.

Our design is based on a simple observation: while frames can change drastically over a long time, when examining adjacent frames within a frame window (i.e., 0.1 s) and focusing on a tiny region, we observe that the color intensity change is often monotonic at a constant speed (Figure 8(a)). As a result, when it is mixed with the encoded α change, the α change creates variations in the speed of perceived color intensity change. As an example, Figure 8(b) shows the perceived color intensity of a region when mixed with α value change encode for bit 1. When the α value decreases (no dimming effect) at the second and fourth frames, it slows down the perceived color intensity change at both points. These speed variations are reflected by the sign of the second derivative of the color intensity change over time. Similarly, α value change for encoding bit 0 also leads to a pattern in the second derivative of color intensity change in a frame window. We leverage these patterns to infer the encoded bit.

Clearly, for regions less affected by interfering sources, these patterns resulting from encoded bits are less noticeable. Therefore, we can determine the weight of each region by examining whether these patterns in the second derivatives exist. In our implementation, for regions where these patterns do not exist, we set their weights as zero to minimize their impact. By focusing on regions that exhibit detectable patterns, we reduce the impact of interfering sources and make the system robust in diverse settings. Furthermore, the pattern detection only requires computing the second derivatives. Hence we can update the weight matrix on the fly based on current screen content.

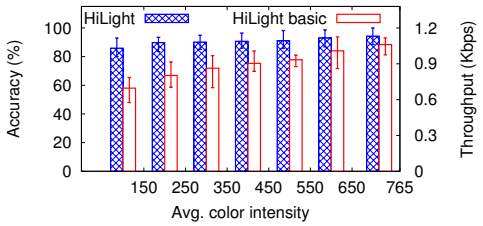
5. SYSTEM IMPLEMENTATION

We implement HiLight at the application layer using off-the-shelf smart devices. We implement HiLight transmitter at the Android platform, and the receiver at the iOS framework using iPhone 5s. We choose iPhone 5s as the receiver because for the receiver to decode data, its camera needs to be able to capture at least 60 frames per second, and the iOS AVFoundation framework supports 120 FPS video recording on iPhone 5s. In contrast, the current Android framework (Android 4.4 or lower) does not support video recording at 60 FPS[4], and phone manufactures (e.g., Samsung, HTC) do not make their phone camera API/library public.

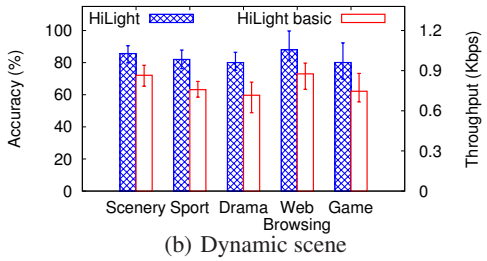
To implement the HiLight transmitter, we use `ImageView` (an Android UI container) to create the communication layer (Figure 1(a))

[4]The recent Android 5 framework [4] includes a new camera API, which supports 4K resolution video recording with 65 Mb/s bandwidth. It supports 60 FPS or a higher frame rate. We plan to implement HiLight on the new platform once the hardware support is available.

Figure 9: Experimental setup with Samsung Tab S (left) as the transmitter and iPhone 5s as the receiver (right).



(a) Static scene



(b) Dynamic scene

Figure 10: HiLight's performance atop arbitrary scene assuming 120 grids on the communication layer. We also plot the performance of HiLight basic [21] that excludes two design enhancements (§ 3.3 and § 4.2) to examine their contribution. Since the throughput is equal to the accuracy multiplied by the number of transmitted bits, we plot a single bar reflecting numbers in both accuracy and throughput.

atop all image layers. We use the `setImageAlpha` method to specify α values at the application layer. This method passes the given α values down to the `SurfaceFlinger` layer of the Android framework, which further leverages the `glBlendFunc` API in the OpenGL ES library [24] to run the alpha blending on the GPU. Given the ubiquity of hardware accelerated compositing, a similar implementation can be realized on other mobile platforms (e.g., iOS, Windows). We implement scene detection and α value generation as two separate threads. In the first thread, we sample the frame buffer under `/dev/graphics/` to fetch the screen content every 0.1 s. We leverage the OpenCV library to implement the scene detection algorithm, which runs continuously to detect the scene type of the incoming sampled frames. The first thread passes the detection result and frame information to the second thread, which then generates α values and modulates each bit using BFSK. The transmitter signals the start of a transmission by sending 1 in all grids for 0.1 s. Finally, to avoid touch events being blocked by the communication layer, which is atop all content image layers, we register a `View.OnTouchListener` to return false so that the system ignores the communication layer and passes the touch event to the applications below the communication layer.

To implement the HiLight receiver, we use the AVCapture session in the iOS AVFoundation framework to capture frames. To reduce the camera noise in the dark environment, we leverage the built-in light sensor to detect the dark environment, and increase the camera ISO value to raise its sensitivity accordingly. We apply an existing screen detector [42] to locate the transmitter screen, extract the screen area from each captured frame, and divide the captured screen area into grids. To speed up the frame decoding, we fetch only a quarter of pixels in the screen area to decode bits. We implement the main components (scene detection, extracting desired color intensity change, and FFT decoding) in three separate threads. We display the decoding results in a pop-up window on the receiver's screen.

6. HiLight EXPERIMENTS

We perform detailed experiments to evaluate the HiLight prototype, focusing on its ability to support unobtrusive transmissions atop arbitrary scene in real time and its robustness in practical settings. We also seek to understand how hardware choice on screen and camera affects its performance, as well as user's perception on the unobtrusiveness of our system.

Experimental Setup. We use the Samsung Tab S as the default transmitter (we test other types of screens in § 6.4). We set up the transmitter and receiver (iPhone 5s) at their normal viewing distance (30 cm for the transmitter's 10.5-inch screen, see Figure 9).

Table 1: Statistics of test images.

Avg. color intensity	<150	[150, 250)	[250, 350)	[350, 450)	[450, 550)	[550, 650)	[650, 765)
# of images	10	31	23	23	14	6	5

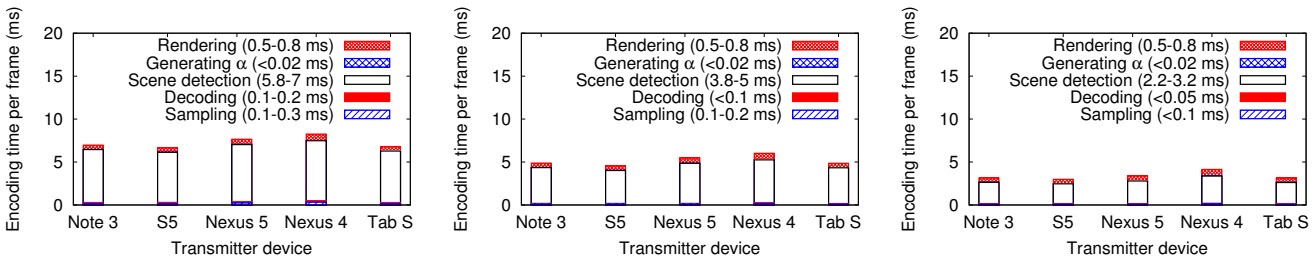
We increase the viewing distance when testing HiLight on transmitters with larger screens in § 6.4. While we fix them on two phone holders for most experiments, we also test the hand-held scenario in § 6.3. By default, we divide the transmitter screen into 120 grids, where each grid is 2.7 cm^2 in size (Table 2). We repeat each experiment for five rounds, and perform all experiments under a fluorescent lamp (100 lux) in an office environment. We focus on two performance metrics: 1) accuracy, the percentage of bits successfully received over all bits transmitted; and 2) throughput, the number of bits successfully received per second.

6.1 Supporting Arbitrary Scene

To evaluate HiLight's ability to support arbitrary scene, we randomly select 112 images with all levels of average color intensity (Table 1), 60 video clips, a 30-min web browsing scene, and 6 gaming apps[5]. We run HiLight to transmit random bits atop each screen content, and measure the throughput and accuracy at the receiver. In all of our graphs, we plot error bars covering the 90% confidence interval.

Static Scene. Figure 10(a) plots the average accuracy and throughput as the average pixel color intensity value varies, where a higher color intensity value indicates a brighter image. We divide the whole color intensity range into seven intervals, and average the results of images with color intensity values in the same interval. Intuition says that dark images lead to less detectable translucency changes and thus much lower throughput. Surprisingly, we observe that HiLight is not sensitive to image color intensity, maintaining 91%+ accuracy and 1.1 Kbps throughput for any images with average color intensity above 150. Even for dark images (color intensity values below 150), HiLight still achieves 85% accuracy and 1 Kbps throughput. This sets a notable improvement over our prior design, referred to as HiLight basic [21]. The improvement can be attributed to the effectiveness of two design components: the adaption of α value change (§ 3.3) and the strategy of extracting encoded color intensity change (§ 4.2). Our hypothesis is confirmed by the results of HiLight basic [21], which fixes the $\Delta\alpha$ to 1% and does not use weight matrix to extract desired color intensity

[5]All the test images and video clips are available at `http://dartnets.cs.dartmouth.edu/hilight-test`.



(a) High resolution (1080p) (b) Medium resolution (720p) (c) Low resolution (480p)

Figure 11: HiLight's encoding time per frame under different frame resolution. Across all test devices, HiLight encodes a single frame within 8 ms, sufficient to support both 60 FPS and 120 FPS frame rates.

changes. To future improve HiLight's performance on very dark scenes (color intensity below 150), we can set white as the grid color of the communication layer atop dark areas. We plan it for future work.

Dynamic Scene. For the dynamic scene, we test three types of video clips including scenery video (e.g., campus video), sports, and drama (e.g., TV shows, movie trailers). They represent video with minor, medium, and drastic frame transition, respectively. For each type, we test 20 video clips and each clip lasts 1–2 mins. We also test two example scenarios when the screen content is generated on the fly. These include 1) web browsing, where the user is reading and scrolling down a web page, refreshing the page, and switching to other web pages; and 2) gaming, where the user is playing games (e.g., Angry Bird, Super Mario).

Figure 10(b) plots the average throughput and accuracy under all types of dynamic scene. Our key observations are as follows. *First*, overall HiLight maintains 84% accuracy with 1 Kbps throughput across these scene types, even for drama video clips that contain drastic screen content change. By examining the performance of HiLight basic, we validate the efficacy of our two key design components, which lead to 18% improvement in general and 20% for drama video in particular. *Second*, dynamic scene leads to slightly lower throughput and accuracy than the static scene, mainly for two reasons. The first reason is that to configure α changes, the system uses sampled color information of sampled frames to estimate actual frames. The sampling leads to estimation errors as frames vary over time, making the adaptation of $\Delta\alpha$ sub-optimal. One solution is to sample more frequently upon more drastic frame transition. We plan it for future work. Another reason is that we defer transmissions upon cut scene. Yet since cut scene occurs rarely (only in sports and drama), the resulting throughput loss is less than 0.6% and 2.4% for sports and drama. *Third*, among different dynamic scenes, drama scene is the most challenging as it contains more drastic content changes that greatly interfere with encoded color intensity change. Web browsing scene achieves the best performance among all because it has a fair portion of static scene when the user reads web pages. Most games we test often change background images drastically, and thus the gaming scene is similar to drama.

Varying the Grid Size. We also examine how different grid size on the communication layer affects HiLight under the static or dynamic scene. We test seven grid sizes (Table 2) using 10 images and 10 video clips, and list HiLight's average throughput and accuracy in Table 2. Overall, as the number of grids increases, HiLight's throughput grows rapidly and its accuracy drops slowly. This is because more grids mean more concurrent transmitter elements, leading to more transmitted bits. Yet as each grid has fewer pixels, the inter-symbol interference [26] kicks in, making it more challenging to differentiate pixel translucency changes across adjacent grids and causing more bit errors. However, the increase in the transmitted bits outweighs the accuracy drop, leading to the throughput growth. In particular, for the grid size of 0.5 mm^2 (600 grids), HiLight achieves 4 Kbps under both static and dynamic scene. The minimal grid size is ultimately limited by the resolution of the camera sensor and the distance between the transmitter and the receiver.

Table 2: Impact of grid size on HiLight performance.

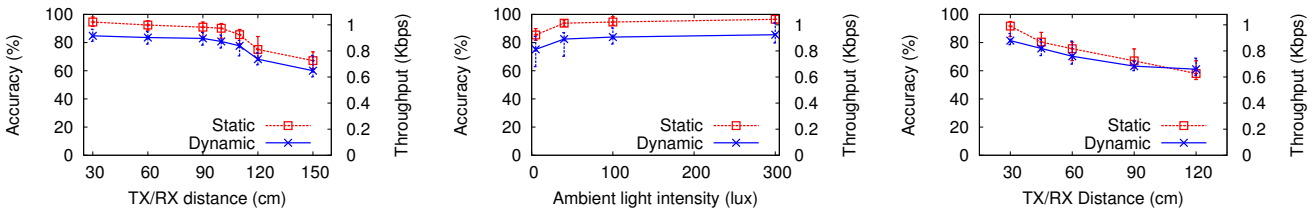
# of grids	6	30	120	180	240	360	600
Size (cm^2)	54.2	10.8	2.7	1.8	1.4	0.9	0.5
# of pixels	683K	137K	34K	23K	17K	11K	6.8K
Static scene							
Accuracy (%)	99.4	93.9	89.4	86.8	82.9	78.7	75.2
Throughput (Kbps)	0.06	0.3	1	1.6	2	2.8	4.5
Dynamic scene							
Accuracy (%)	91.0	87.1	85.4	80.3	78.3	73.2	67.2
Throughput (Kbps)	0.05	0.3	1	1.4	1.9	2.6	4

6.2 Processing Time

To evaluate HiLight's ability to support real-time communication, we measure its processing time of per-frame encoding and decoding. We measure the data encoding time as the duration starting from fetching a frame to the end of rendering. The decoding time is the time taken to process frames in a frame window (six frames) upon the arrival of a new frame.

Encoding Time. We measure HiLight's encoding time on five Android devices. Since the encoding time is not affected by the number of grids on the screen and heavily depends on the frame resolution, we test three resolution levels, each with 10 video clips. Figure 11 plots the total per-frame encoding time as well as the breakdown. We make three key observations. *First*, across all devices and frame resolution levels, HiLight's encoding delay is consistently below 8.3 ms. This is mainly because HiLight leverages GPU to process α values and removes the need to directly modify pixel RGB values. With the per-frame encoding time below 8.3 ms, HiLight can further support high-resolution video streaming at 120 FPS while transmitting data unobtrusively.

Second, among all encoding steps, scene detection occupies 78–87% of the encoding time and ranges from 2.5 ms to 7 ms. It is longer for the higher frame resolution, because there are more pixels for the algorithm to sample and process. *Third*, our sampling strategy significantly reduces the overhead of fetching and decoding a frame. Since we sample one frame per frame window (six frames) and sample one-sixteenth of pixels in the sampled frame,



(a) Supporting distance (b) Ambient light (c) Hand motion

Figure 12: Impact of practical factors on HiLight's performance.

Table 3: HiLight's decoding time over a frame window under different grid size, using iPhone 5s.

Decoding steps	Decoding time (ms) under varying # of grids						
	6	30	120	180	240	360	600
Sampling	2.95						
Scene detection	1.22						
Extracting changes	0.02						
Demodulation	0.07	0.21	0.81	1.1	1.73	2.47	3.31
Overall	4.26	4.4	5	5.29	5.92	6.66	7.5

fetching pixel color values entails 0.35 ms, which is 1/96 of fetching every frame and obtaining the color values of all pixels.

Decoding Time. We further examine HiLight's data decoding time as the number of grids on the communication layer varies. Table 3 lists the time taken by each decoding step when processing frames in a frame window on iPhone 5s. Aiming to capture the decoding time upon each incoming frame on the fly, we do not include the MVDR training delay (§ 4.2) in the table, because we do not need to perform the training for each frame, but only in the beginning of the whole decoding process for a static scene[6]. From Table 3, our key observations are as follows. *First*, under all the grid sizes, HiLight is able to decode frames in a frame window within 8 ms. This demonstrates that even when the transmitter is sending data at 120 Hz (supported by higher screen refresh rate), the receiver is able to process all incoming data in real time. *Second*, for all decoding steps except BFSK demodulation, their running time is independent of the number of grids. This is because these steps are performed upon sampled pixels of each frame. Only the number of sampled pixels affects their running time. In comparison, BFSK demodulation performs FFT for each grid independently to decode each bit. More grids lead to more FFT computations and thus a longer processing time.

6.3 Practical Considerations

We now evaluate the impact of practical factors on HiLight's performance. We test 10 images and 10 video clips, and repeat all experiments for five rounds.

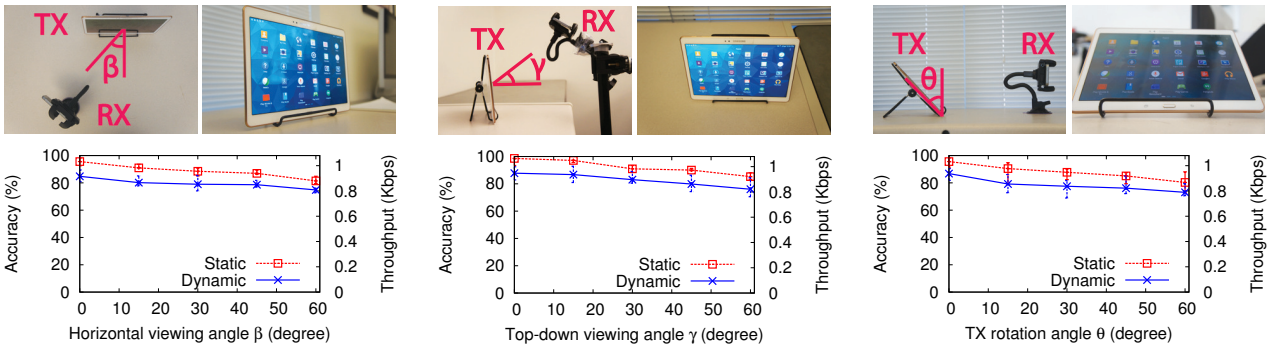
Supporting Distance. We start with examining the transmission distance HiLight supports. We increase the distance between the screen and the camera from 30 cm (the normal viewing distance) to 150 cm, and plot the resulting throughput and accuracy under both static and dynamic scene in Figure 12(a). Overall HiLight's performance is relatively stable when the distance is within 1 m, maintaining 90%+ accuracy for the static scene and 85%+ for the dynamic scene. This demonstrates the efficacy of HiLight design. As the distance further increases, throughput and accuracy start to drop quickly. This is expected, because light attenuates over distance, leading to weaker illumination changes that are harder to detect. In addition, since the receiver camera does not have an optical zoom, as the receiver is further away, the transmitter screen becomes smaller in the captured frame. This reduces the captured pixels for decoding and lowers the decoding accuracy.

Ambient Light. Next, we examine HiLight's sensitivity to ambient light. We test four indoor ambient light conditions (1 lux, 40 lux, 100 lux, and 300 lux), and plot HiLight's performance in Figure 12(b). Intuitively, dark settings are challenging for HiLight, where pixel translucency changes are harder to detect and camera noise is also higher. Yet we observe that the HiLight's performance is fairly stable once the light illuminance is above 40 lux (far below the normal ambient light, which is around 100 lux). HiLight achieves the robustness because the system leverages the built-in light sensor on iPhone 5s to sense the ambient light and raises the camera sensitivity in dark settings. This allows the receiver to better sense the pixel translucency change. While increasing camera sensitivity also introduces camera noise, most camera noise is filtered out by our weight matrix used to extract the desired color intensity change (§ 4.2).

Hand Motion. We now move on to examining the impact of hand motion on HiLight. We hold the receiver in the air facing a fixed transmitter, and vary the distance between the transmitter and the receiver. Figure 12(c) plots the results when the transmitter screen is displaying static or dynamic content. As expected, hand motion causes misalignment between two devices, leading to image blur and thus lowering the accuracy. Yet at the normal viewing distance (30 cm), the accuracy drop is minor (<3–4% compared to the perfectly aligned scenario in Figure 12(a)). As the distance increases, the performance gap to the perfectly aligned scenario increases. This is because at longer distances, hand motion causes more significant screen misalignment, and the receiver is more likely to misidentify the grids on the screen. Comparing the two types of scene, we observe that the accuracy of the static scene drops more quickly. This is because the weight matrix for the static scene is calculated once during decoding, while for the dynamic scene, the weight matrix is updated for every frame window and thus can better deal with screen misalignment. To minimize the impact of hand motion, one can leverage image tracking and alignment algorithms in computer vision to track the transmitter screen. The challenge is to ensure that these algorithms can be performed in real time. We plan to study this as part of our future work.

Viewing Angle. Finally, we examine HiLight with varying horizontal and vertical viewing angles to assess the impact of the misalignment between the transmitter and the receiver. In the first two experiments, we fix the transmitter screen and rotate the receiver either horizontally or vertically by 0° to 60° while keeping their

[6]Our measurements show that MVDR training takes 1 ms to 27 ms as the number grids increases from 6 to 600, respectively.



(a) Horizontal viewing angle β (b) Top-down viewing angle γ (c) TX rotation angle θ

Figure 13: Impact of the horizontal and vertical viewing angle on HiLight's performance.

Table 4: Impact of screens on HiLight's performance, using iPhone 5s as the receiver.

TX	Screen	Scene type	Through-put (Kbps)	# of grids	Accuracy (%)
Samsung Tab S	OLED 10.5" 2560 × 1600	Static	1.1	120	91
		Dynamic	0.3	30	90
Nexus 5	LCD 5.0" 1920 × 1080	Static	0.6	60	91
		Dynamic	0.2	20	91
Mac Air	LED 13.3" 1440 × 900	Static	1.1	120	91
		Dynamic	0.3	30	90
iMac	LED 27" 2560 × 1440	Static	1.6	180	91
		Dynamic	0.6	60	91

Table 5: Impact of cameras on HiLight's performance, using Samsung Tab S as the transmitter.

Receiver	Cam-era	Scene type	Through-put (Kbps)	# of grids	Accuracy (%)
iPhone 5s	8 MP	Static	1.1	120	91
		Dynamic	0.3	30	90
Samsung Note 3	13 MP	Static	4.9	540	90
		Dynamic	4.9	540	90
Samsung S5	16 MP	Static	6.7	720	93
		Dynamic	4.9	540	90
Canon 60D (SLR camera)	18 MP	Static	6.6	720	92
		Dynamic	5.0	540	92

distance the same (30 cm). In the third experiment, we fix the receiver and make the transmitter screen self-rotate from 0° to 60°. Figure 13 shows the three experimental setups, the observed perspective distortion in each setup[7], and HiLight's throughput and accuracy. Overall, HiLight supports up to 60° viewing angle in all directions for both types of scenes. Its performance gracefully degrades as the viewing angle increases, because the transmitter screen captured by the receiver distorts as a trapezoid, making it challenging for the receiver to identify grids. In addition, since different pixels on the screen have different depths to the receiver, some portion of the captured screen can be out of focus. However, the performance degradation is less than 7–8% for both scene types. This is because HiLight relies on the relative color intensity change to extract data and is insensitive to static noises such as the image quality degradation caused by being out of focus. To further enhance HiLight's robustness against perspective distortion, one can leverage image projection algorithms (e.g., affine transformation [8]) to avoid grid misidentification. Our experiments show that the current implementation of these algorithms requires 100 ms to process a frame, thus we need an optimized implementation to support real-time communication.

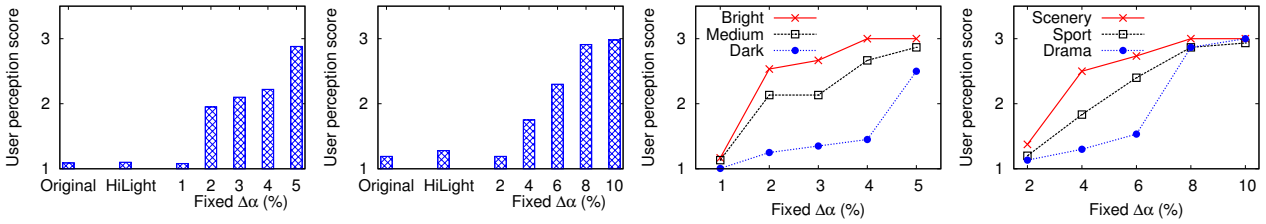
6.4 Impact of Screen/Camera Hardware

We next examine how the physical capability of screens and cameras affects HiLight's performance. We test HiLight on different screens and cameras and examine its highest throughput while maintaining accuracy above 90%. We adjust the distance for each pair of devices to ensure the captured screen size in the camera preview the same.

[7]Using the 10-inch OLED screen, we observe minor pixel brightness degradation caused by viewing angle change (<10% at 60° viewing angle). Its impact on HiLight's performance is negligible.

Varying the Screen. We first fix iPhone 5s as the receiver and test different types (LCD, LED, OLED) of screens in varying sizes (5–27 in) as transmitters. Since our current implementation of the HiLight transmitter is based on the Android framework, to test HiLight on devices such as Mac Air and iMac, we make video clips with data embedded into a static or dynamic scene, play these video clips on each screen, and measure the real-time accuracy and throughput at the receiver. Table 4 lists the results for 10 static and 10 dynamic scenes. Clearly screen resolution and size are the two key factors affecting the performance. iMac's screen works the best with the largest size and second-highest resolution. Among different screen types, OLED screens are the most preferable, because OLED screens do not have backlight and each pixel emits light independently. Therefore, colors on OLED screens are brighter with higher contrast, making color intensity change easier to detect. In comparison, both LCD and LED screens have backlight that reduces the color contrast. LED screens outperform LCD screens because the LED backlight renders color change more precisely.

Varying the Camera. Next, we fix the Samsung Tab S as the transmitter, use different cameras to capture frames, and process captured frames offline to examine the accuracy and throughput. We test four types of cameras including a high-end SLR camera with 18 MP resolution. Results in Table 5 demonstrate the significant performance gain brought by high-resolution cameras for both types of scene. Higher-resolution cameras capture more pixels on the transmitter screen, and thus they support smaller grids on the transmitter screen and achieve higher throughput. In particular, the high-end SLR camera supports 720 grids on the 10-in screen enabling 6.6 Kbps throughput, six times higher than that achieved by iPhone 5s in our prototype. Even the cameras on some existing smart devices (Note 3 and S5) are sufficient for HiLight to reach 5 – 6 Kbps, similar to that of the high-end camera. This demonstrates



(a) Static scene (b) Dynamic scene (c) Static scene, fixed $\Delta\alpha$ (d) Dynamic scene, fixed $\Delta\alpha$

Figure 14: User perception on HiLight's unobtrusive transmissions, where the perception scores are explained in Table 7. Achieving scores of 2 or below is acceptable. Comparing to changing pixel translucency uniformly with a fixed amount, HiLight adapts the pixel translucency changes ($\Delta\alpha$) non-uniformly across pixels, and thus diminishes the visual effects caused by data communication.

Table 6: Statistics of the 20 video clips used in our user study.

Scene type	Static scene			Dynamic scene		
	Bright	Medium	Dark	Scenery	Sport	Drama
# of video clips	3	3	4	3	3	4

Table 7: User perception scores.

Score	User perception
1	Certainly no noticeable visual effects. Good viewing experience.
2	Uncertain about visual effects. Still good viewing experience.
3	Certainly noticeable visual effects.

HiLight's robustness on diverse hardware platforms and its great potential on future smart devices with more sophisticated cameras.

6.5 User Perception

Finally, we conduct a user study to examine whether HiLight causes any noticeable visual artifacts on the original screen content. Our user study is conducted with 10 participants (5 male and 5 female) in the age range of 18 to 30. We select 20 original video clips (each lasts for 10 s) covering diverse scenes (Table 6). For each video clip, we create six modified versions by encoding data into pixel translucency (α) changes. In one version, we use HiLight to encode data, which adapts $\Delta\alpha$ non-uniformly across pixels (Algorithm 1). In the other five versions, we change the α values of all pixels uniformly with a fixed amount, varying from 1% to 5% for the static scenes and 2% to 10% for the dynamic scenes. Among the total 140 video clips (20 original and 120 modified), we randomly assign 100 of them to each participant. To avoid participant response bias [12], we do not inform participants which video clips are modified and which are original. Before the study, each participant watches the original video clips on a Samsung Tab S screen to gain an initial sense of the video content. During the study, each participant watches the assigned 100 video clips (mixed with the original and modified versions) in a random order on the Tab S screen and rates from 1 to 3 to indicate the certainty of observing visual effects in each video clip[8]. Table 7 shows the perception scores. Achieving a score of 1 is ideal and 2 is acceptable.

We plot the average user perception scores in Figure 14(a) and (b). We make two key observations. *First*, by adapting the amount of translucency changes ($\Delta\alpha$) non-uniformly across the screen, HiLight significantly reduces the visual effects caused by the $\Delta\alpha$ in both static and dynamic scenes. User's perception level on HiLight is very close to that of watching the original video clips. In contrast, a fixed $\Delta\alpha$ across all pixels easily leads to noticeable visual effects even for small $\Delta\alpha$ values (4% for static scenes and 6% for the dynamic scenes). This result demonstrates the effectiveness of HiLight's adaptation of $\Delta\alpha$ based on sampled screen content pixels and its gradient smoothing of $\Delta\alpha$ within each grid (Figure 6).

Second, the maximal amount of translucency changes that users cannot perceive depends on the screen content. Between static and dynamic scenes, dynamic scenes allow larger pixel translucency changes that are unnoticeable by users. We hypothesize that this is because human eyes are less sensitive to subtle changes in pixel brightness when the screen content itself contains more drastic color intensity changes. We observe a similar pattern as we analyze different types of dynamic scenes. Figure 14(d) plots the average user perception score for three types of dynamic scenes, as the fixed $\Delta\alpha$ increases. For drama scenes that have the most drastic color intensity changes, 6% of fixed α changes are still almost unnoticeable by users. Among the static scenes, we observe that users can tolerate larger pixel translucency changes ($\Delta\alpha$) atop darker screen content (Figure 14(c)). This is because the absolute brightness changes of darker areas are smaller than that of brighter areas. These observations together justify the $\Delta\alpha$ configuration in HiLight (Algorithm 1), which increases the $\Delta\alpha$ for dark areas and dynamic scenes.

[8]A participant can replay a video clip multiple times to determine the score.

7. POTENTIAL APPLICATIONS

Augmented Reality. HiLight can enable innovative interaction designs for augmented reality (AR) apps. For AR glasses with heads-up displays and cameras, they can leverage HiLight to obtain additional information layers from any surrounding screens. Users wearing smart glasses can acquire personalized information from any screen, including the subtitles of a movie or TV program, customized coupon or URL link within an advertisement, and hints in gaming apps, without affecting the content shown on the screen.

Continuous Authentication. Operating on the visible light spectrum, HiLight communication occurs only when the camera is physically facing the screen. This allows us to continuously authenticate user's physical presence, highly valuable for many existing apps. Consider a user wearing smart glasses in front of a screen. Online banking or health-related apps can leverage HiLight to continuously authenticate user's presence and immediately log out the user when the user leaves the screen. In addition, online video streaming providers such as Amazon can leverage HiLight to monitor the exact portion of video a user has watched, and define accurate pricing scheme based on the information.

Video Tagging. When capturing a video, it is hard to add object tags on the fly. Using HiLight, we can embed information associated with the local object or location into all public screens. Thus, when a camera captures video, it also simultaneously fetches

the additional information from surrounding screens and attaches it with the video. Social apps such as Facebook can then automatically extract the information in the captured video and add tags for the user.

8. RELATED WORK

We categorize existing research on screen-camera communication into two categories.

Obtrusive Screen-Camera Communication. Active research has examined screen-camera communication using visible coded images [6]. One class of work focuses on boosting data rate, by leveraging better modulation schemes [26] or innovative barcode designs [14, 43]. Another class of work aims to enhance link reliability, by tackling the frame synchronization problem [15, 22, 28], or enabling barcode detection at long distances [16, 23], or designing multi-layer error correction schemes [36]. A recent study [40] generates dynamic QR code to transmit sensor data on the fly. Our work leverages insights from these studies, but differs in that we aim to enable screen-camera communication without showing visible coded images.

Unobtrusive Screen-Camera Communication. Prior work has studied how to hide information in a given screen content while enabling screen-camera communication. Yuan *et al.* leverage watermarking to embed messages into an image [41, 42]. [10, 37, 39] enable unobtrusive communication by switching barcodes with complementary hue. PiCode and ViCode [19] integrate barcodes with existing images to enhance viewing experiences. All these methods have greatly reduced the visual artifacts of visible codes. Yet they all require direct modifications of content pixel RGB values, which prevents them from supporting arbitrary scene in real time. The unique contribution of HiLight is to broaden the applicable scenario of unobtrusive screen-camera communication. It removes the need to directly modify RGB values and decouples communication from screen content. HiLight is similar in spirit to prior efforts that modulate bits by changing the screen's backlight [13, 18, 20]. All these designs, however, require special hardware support (e.g., shutter plane). HiLight differs in that it works upon off-the-shelf smart devices.

9. CONCLUSION

We presented HiLight, the first system enabling real-time, unobtrusive screen-camera communication atop any screen scene (can be generated on the fly). We implemented HiLight on off-the-shelf smart devices and evaluated HiLight in diverse practical settings. By placing no constraint on the screen's original functionality (displaying content) while enabling communication, HiLight opens up opportunities for new HCI and context-aware applications.

Our current system has several limitations and possible extension that we plan to address in future work. *First*, the throughput of our system is currently limited by the screen refresh rate (60 Hz). For OLED screens, the physical response time of a pixel can be less than 0.01 ms [5], translating into the pixel change frequency higher than 100 KHz. We plan to seek methods to better approach this physical limit and boost the system throughput. *Second*, we plan to explore advanced modulation and coding schemes to improve the transmission reliability. We observe that most bit errors are randomly spread out. Since the current communication channel is one-way without any feedback from the receiver, error handling schemes such as low-density parity-check (LDPC) code [11] or Reed-Solomon code [38] are good candidates. *Third*, our current design sets a uniform color (black) for the communication layer. We plan to vary the color (between black and white) across different grids of the communication layer based on the content area color (e.g., use white color for dark areas and black for bright areas). This color adaptation can enhance the system reliability atop a wide range of screen content. *Forth*, to better support device mobility (e.g., a user wearing smart glasses as the receiver), we plan to integrate low-complexity object tracking algorithms into HiLight, so that the receiver can keep tracking the screen area and decode data even during constant hand or head movement. *Finally*, our system is still a one-way channel. To realize a two-way channel, we can place two devices with screens facing each other and each device uses its front-facing camera to receive data. Obtaining receiver feedback allows the transmitter to adapt its configuration (e.g., grid size, frame window). We plan to explore the associated systems and design challenges.

10. ACKNOWLEDGMENTS

The authors sincerely thank shepherd Chunyi Peng and the reviewers for their valuable feedback and Jiawen Chen at Google for his insights on GPU programming. We also thank DartNets Lab members Zhao Tian, Rui Wang, Fanglin Chen, and Xiaole An for their support on our study. This work is supported in part by the Dartmouth Burke Research Initiation Award and the National Science Foundation under grant CNS-1421528. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding agencies or others.

11. REFERENCES

[1] `https://www.opengl.org/`.

[2] `http://opencv.org/platforms/android.html`.

[3] `http://developer.android.com/guide/topics/resources/drawable-resource.html`.

[4] `https://developer.android.com/about/versions/android-5.0.html`.

[5] `http://en.wikipedia.org/wiki/OLED`.

[6] Automatic identification and data capture techniques - QR code 2005 bar code symbology specification. ISO/IEC 18004:2006.

[7] ASHOK, A., ET AL. Challenge: Mobile optical networks through visual MIMO. In *Proc. of MobiCom* (2010).

[8] BERGER, M. *Geometry I*. Springer Science & Business, 2009.

[9] BULLOUGH, J., ET AL. Effects of flicker characteristics from solid-state lighting on detection, acceptability and comfort. *Lighting Research and Technology 43*, 3 (2011), 337–348.

[10] CARVALHO, R., CHU, C.-H., AND CHEN, L.-J. IVC: Imperceptible video communication. In *Proc. of HotMobile (poster)* (2014).

[11] DAVEY, M. C., AND MACKAY, D. J. Low density parity check codes over gf (q). In *Information Theory Workshop, 1998* (1998), IEEE, pp. 70–71.

[12] DELL, N., VAIDYANATHAN, V., MEDHI, I., CUTRELL, E., AND THIES, W. "Yours is Better!": Participant Response Bias in HCI. In *Proc. of CHI* (2012).

[13] FATTAL, D., ET AL. A multi-directional backlight for a wide-angle, glasses-free three-dimensional display. *Nature 495*, 7441 (2013), 348–351.

[14] HAO, T., ZHOU, R., AND XING, G. COBRA: Color barcode streaming for smartphone systems. In *Proc. of MobiSys* (2012).

[15] HU, W., GU, H., AND PU, Q. LightSync: Unsynchronized visual communication over screen-camera links. In *Proc. of MobiCom* (2013).

[16] HU, W., MAO, J., HUANG, Z., XUE, Y., SHE, J., BIAN, K., AND SHEN, G. Strata: Layered coding for scalable visual communication. In *Proc. of MobiCom* (2014).

[17] HUANG, C.-L., AND LIAO, B.-Y. A robust scene-change detection method for video segmentation. *IEEE Transactions on Circuits and Systems for Video Technology 11*, 12 (2001), 1281–1288.

[18] HUANG, S. Backlight modulation circuit having rough and fine illumination signal processing circuit, Mar. 27 2012. US Patent 8,144,112.

[19] HUANG, W., AND MOW, W. H. PiCode: 2D barcode with embedded picture and ViCode: 3D barcode with embedded video (poster). In *Proc. of MobiCom* (2013).

[20] KIMURA, K., MASUDA, S., AND HAYASHI, M. Display apparatus and method for controlling a backlight with multiple light sources of a display unit, Sept. 11 2012. US Patent 8,264,447.

[21] LI, T., AN, C., CAMPBELL, A., AND ZHOU, X. HiLight: Hiding bits in pixel translucency changes. In *Proc. of the 1st ACM MobiCom Workshop on Visible Light Communication Systems (VLCS)* (2014).

[22] LIKAMWA, R., RAMIREZ, D., AND HOLLOWAY, J. Styrofoam: A tightly packed coding scheme for camera-based visible light communication. In *Proc. of the 1st ACM MobiCom Workshop on Visible Light Communication Systems (VLCS)* (2014).

[23] MOHAN, A., WOO, G., HIURA, S., SMITHWICK, Q., AND RASKAR, R. Bokode: Imperceptible visual tags for camera based interaction from a distance. In *Proc. of SIGGRAPH* (2009).

[24] MUNSHI, A., GINSBURG, D., AND SHREINER, D. *OpenGL ES 2.0 programming guide*. Pearson Education, 2008.

[25] NAGASAKA, A., AND TANAKA, Y. Automatic video indexing and full-video search for object appearances. In *Proc. of the IFIP TC2/WG 2.6 Second Working Conference on Visual Database Systems II* (1992).

[26] PERLI, S. D., AHMED, N., AND KATABI, D. PixNet: Interference-free wireless links using LCD-camera pairs. In *Proc. of MobiCom* (2010).

[27] PORTER, T., AND DUFF, T. Compositing digital images. In *ACM SIGGRAPH Computer Graphics* (1984).

[28] RAJAGOPAL, N., LAZIK, P., AND ROWE, A. Visual light landmarks for mobile devices. In *Proc. of IPSN* (2014).

[29] ROSIN, P. L. Thresholding for change detection. In *Computer Vision, 1998. Sixth International Conference on* (1998), IEEE, pp. 274–279.

[30] ROSIN, P. L., AND IOANNIDIS, E. Evaluation of global image thresholding for change detection. *Pattern Recognition Letters 24*, 14 (2003), 2345–2356.

[31] SETHI, I. K., AND PATEL, N. V. Statistical approach to scene change detection. In *IS&T/SPIE's Symposium on Electronic Imaging: Science & Technology* (1995), pp. 329–338.

[32] SMITS, P. C., AND ANNONI, A. Toward specification-driven change detection. *IEEE Transactions on Geoscience and Remote Sensing 38*, 3 (2000), 1484–1488.

[33] SUR, S., WEI, T., AND ZHANG, X. Autodirective audio capturing through a synchronized smartphone array. In *ACM Mobisys 2014*.

[34] TAN, K. W., ET AL. FOCUS: a usable & effective approach to OLED display power management. In *UbiComp* (2013).

[35] VAN DE SANDE, J., AND ASSIGNOR, T. Real-time beamforming and sound classification parameter generation in public environments. *TNO report TNO-DV* (2012), S007.

[36] WANG, A., MA, S., HU, C., HUAI, J., PENG, C., AND SHEN, G. Enhancing reliability to boost the throughput over screen-camera links. In *Proc. of MobiCom* (2014).

[37] WANG, A., PENG, C., ZHANG, O., SHEN, G., AND ZENG, B. InFrame: Multiflexing full-frame visible communication channel for humans and devices. In *Proc. of HotNets* (2014).

[38] WICKER, S. B., AND BHARGAVA, V. K. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[39] WOO, G., LIPPMAN, A., AND RASKAR, R. VRCodes: Unobtrusive and active visual codes for interaction by exploiting rolling shutter. In *Proc. of ISMAR* (2012).

[40] YONEZAWA, T., OGAWA, M., KYONO, Y., NOZAKI, H., NAKAZAWA, J., NAKAMURA, O., AND TOKUDA, H. SENSeTREAM: Enhancing online live experience with sensor-federated video stream using animated two-dimensional code. In *Proc. of UbiComp* (2014).

[41] YUAN, W., DANA, K., VARGA, M., ASHOK, A., GRUTESER, M., AND MANDAYAM, N. Computer vision methods for visual mimo optical systems. *Proceedings of the IEEE International Workshop on Projector-Camera Systems (held with CVPR)* (2011), 37–43.

[42] YUAN, W., ET AL. Dynamic and invisible messaging for visual mimo. In *IEEE Workshop on Applications of Computer Vision (WACV)* (2012).

[43] ZHANG, B., ET AL. SBVLC: Secure barcode-based visible light communication for smartphones. In *Proc. of INFOCOM* (2014).

[44] ZHANG, H., KANKANHALLI, A., AND SMOLIAR, S. W. Automatic partitioning of full-motion video. *Multimedia Systems 1*, 1 (1993), 10–28.