Towards Fully Offloaded Cloud-based AR: Design, Implementation and Experience

Ryan Shea
Simon Fraser University
Burnaby, British Columbia, Canada
ryan_shea@sfu.ca

Andy Sun
Simon Fraser University
Burnaby, British Columbia, Canada
hpsun@sfu.ca

Silvery Fu
Simon Fraser University
Burnaby, British Columbia, Canada
dif@sfu.ca

Jiangchuan Liu
Simon Fraser University
Burnaby, British Columbia, Canada
jcliu@cs.sfu.ca

ABSTRACT

Combining advanced sensors and powerful processing capabilities smart-phone based augmented reality (AR) is becoming increasingly prolific. The increase in prominence of these resource hungry AR applications poses significant challenges to energy constrained environments such as mobile-phones.

To that end we present a platform for offloading AR applications to powerful cloud servers. We implement this system using a thin-client design and explore its performance using the real world application Pokemon Go as a case study. We show that with careful design a thin client is capable of offloading much of the AR processing to a cloud server, with the results being streamed back. Our initial experiments show substantial energy savings, low latency and excellent image quality even at relatively low bit-rates.

CCS CONCEPTS

•Networks → Cloud computing; Mobile networks; **•Human-centered computing → Mobile phones; Mobile devices;** *Virtual reality; Ubiquitous and mobile computing design and evaluation methods; Empirical studies in ubiquitous and mobile computing;* **•Computer systems organization → Cloud computing;** **•Hardware →** *Sensor devices and platforms;*

KEYWORDS

Augmented Reality, Cloud Offloading, Mobile Gaming

ACM Reference format:
Ryan Shea, Andy Sun, Silvery Fu, and Jiangchuan Liu. 2017. Towards Fully Offloaded Cloud-based AR: Design, Implementation and Experience. In *Proceedings of MMSys'17, Taipei, Taiwan, June 20-23, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3083187.3084012

MMSys'17, Taipei, Taiwan

978-1-4503-5002-0/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3083187.3084012

1 INTRODUCTION

Integrating powerful sensing capability and unparalleled mobile communication abilities the smart-phone has led to a plethora of new and exciting applications. From instant world wide communication to real-time point by point navigation the smart-phone has become a near ubiquitous part of modern day life.

The combination of accurate sensors, cameras, and powerful processing capabilities has led to a new wave of applications that augment the physical world with digital information. These augmented reality (AR) applications allow us to instantly get information about real-world objects or translate foreign languages. However, with any new technology there are drawbacks and smart-phone enabled AR is no exception. The amount of processing power required to analyze data from our smart-phones' many sensors, render the augmented reality elements, and finally compose the AR scene takes considerable power from our devices. Moreover, mobile devices are energy limited due to battery restrictions meaning that a rich AR application can quickly drain even the largest smart-phone batteries. For example, measurements with the popular AR gaming app Pokemon Go show that the Pokemon uses nearly three times the amount of battery when compared to browsing social media[1]. In Section 2.3, we closely profile Pokemon Go and show much of the energy usage comes from rendering 3D objects.

Based on the previously discussed energy consumption issues we propose that a complex augmented reality application such as Pokemon Go could benefit from cloud offloading. Many pioneering works have explored offloading different aspects of augmented reality applications. In 2013, researchers explored many potential improvements that could be made to mobile applications by gearing them with powerful clouds [9]. In 2014, Huang *et al* presented CloudRidAR, a frame work for offloading some computation intensive aspects of AR to the cloud [5]. Shi *et al* explored computational offloading guidelines between wearable devices and mobile devices [14]. Enhancements using cloud based processing and live object retrieval were presented in [6][7]. Recently, work has been done on quantify the performance implications of edge computing on latency sensitive applications such as AR [3][10].

Further, because of the latency sensitive nature of AR offloading many similar techniques to the ones found in cloud gaming

[1] https://www.cnet.com/how-to/pokemon-go-battery-test-data-usage/

are also relevant. In 2013, researchers explored the existing commercial offerings of cloud gaming in terms of architecture and performance [13]. Huang *et al* released the first open source cloud gaming platform [4]. Finally, the future of cloud gaming was discussed in a recent article by Cai *et al.* The article forecasted changes in the programing paradigm of gaming applications to facilitate better integration between games and cloud offloading [1]. It is likely similar advances in the programing paradigms of augmented reality applications could facilitate seamless offloading for augmented reality applications.

Despite these pioneering works, to our knowledge no existing work has been made to fully offload an AR application to the cloud. To this end we develop CloudAR, our prototype thin client based approach which is capable of producing an AR scene using cloud offloading. In one experiment, our client drew up to 65% less energy in comparison to the representative AR application Pokemon Go. Further, the client had an average end-to-end interaction delay of 55 ms, which provides a low-latency user experience with respect to the integration of the virtual and physical scenes. This end result is a client that achieves excellent video quality with low interaction delay, all the while consuming less energy than other popular AR applications.

The rest of the paper is organized as follows. In Section 2 we dissect and profile the popular augmented reality game Pokemon Go to determine possible improvements. In Section 3, we discuss different visual elements inherent to augmented reality applications. Section 4 and Section 5 discusses the design and evaluation of the CloudAR platform. Finally, Section 6 provides some further discussion and concludes the paper.

2 EXPLORING AR: POKEMON GO AS A CASE STUDY

Combining augmented reality, edge computing, pervasive smartphone use and location based massively multi-player features, Pokemon GO exploded onto smart-phones in the summer of 2016. It is estimated that at the peak of the craze Pokemon GO was installed by over 10% of smart phone users in the USA[2]. On the surface Pokemon GO seems to be little more than a simple gaming app, however under the hood many technological advances are brought together to make this game a success. While one can not disregard the marketing and popularity of the Pokemon franchise when discussing its near meteoric rise to popularity, it is important to point out that new Pokemon games are released on an almost yearly schedule, with very few of these releases achieving such initial success as Pokemon GO. Much of Pokemon GO's success comes from its leveraging of many existing technologies in very effective ways.

Pokemon Go can trace its origin back to one of Google's many April Fool's pranks: the Google Maps Pokemon Challenge. In 2014, Google, in conjunction with Nintendo and The Pokemon Company, announced a new job position of "Pokemon Master" and required applicants to capture all 721 Pokemon before being offered the role. This harmless video drew in an enormous positive response, and set in motion a series of events which culminated the creation of Pokemon Go by Niantic, an internal Google start-up.

[2]https://www.similarweb.com/blog/pokemon-go-update

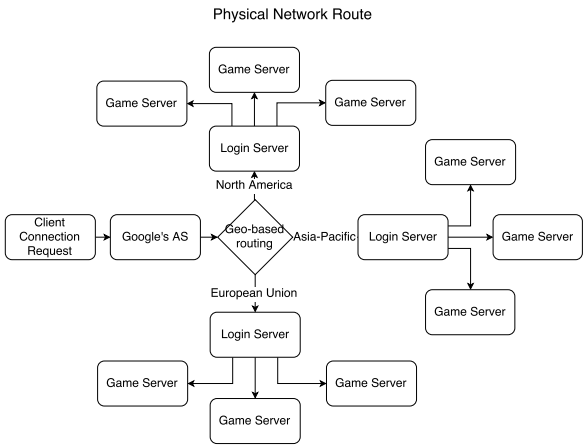


Figure 1: PGO Server Selection

Pokemon Go is one of few games that are truly mobile as it forces the player to physically roam an area by utilizing a mobile device's capabilities. This new genre of gaming creates architectural challenges and design choices that must be carefully considered. As the player's avatar is now attached to a physical location, strong game server scalability becomes a necessity due to the high clustering seen in human population density. Game client optimization also becomes a high priority, as the mobile device's sensors are constantly on and causing drain on the battery. With these in mind, we begin our analysis by examining the architectural model of Pokemon Go.

Prior to Pokemon Go, Ingress was among the first location based mobile games that paved the way for the later popularity of location based AR games [2]. Cloud computing is the driving force behind these successes. Both games are powered by the Google Cloud and the Google edge network to achieve global, high-quality coverage [15].

2.1 Networking Architecture of Pokemon Go

We observed Pokemon Go to have a logical networking topology resembling a star topology. Namely, a central URL, pgorelease.nianticlabs.com/plfe/rpc, directs a client to a regional server which then proceeds to serve all future requests until the connection is terminated. To reconnect back to a game server, the client must contact the central server and wait for a response. The client is not guaranteed to connect to the same edge server. It is important to note that while exploring the network topology we found that although the connections all appear to route to a single Google IP located in Mountain View California the connection is actually being serviced by a server closer to the client. We performed an investigation using trace-route and RTT analysis, and by inspecting the autonomous systems our packets traversed. Our discoveries are presented in Figure 1. In our measurements from servers located in six geo-distributed locations we discovered that Google handles authentication requests in at least three distinct locations. Further, regardless of which region we resolve the URL

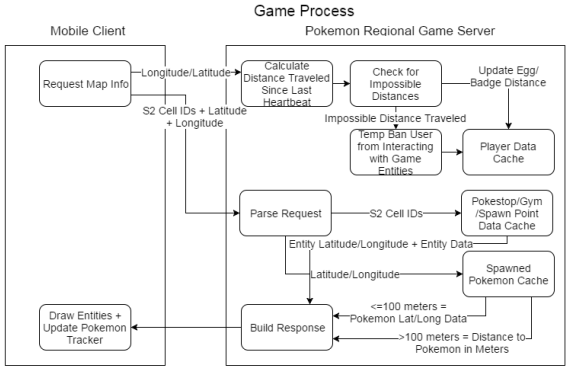
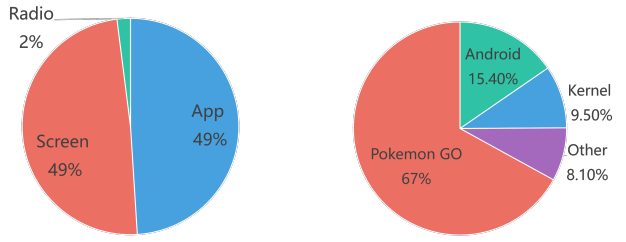


Figure 2: Pokemon Go Architecture



(a) Power total **(b) Power app**

Figure 3: Power profiling on the mobile device (Total: 3544 mW)

pgorelease.nianticlabs.com/plfe/rpc in our experiments, we are always given the same IP address corresponding to a location in Mountain View California. Based on our network analysis we find that it is likely Google uses agreements with major Internet exchanges in order to service authentication requests closer to the clients.

All communications between the client and the server are handled over HTTPS and all data is exchanged in the protobuf format. The bulk of network transactions are the retrieval and updating of map objects from the server, based on the player's location.

Specific API calls are wrapped and sent in the repeated requests field, and the authentication ticket received from the central server is added verbatim into a auth_ticket. The request hash signature is generated using xxHash followed by an in-house encryption algorithm.

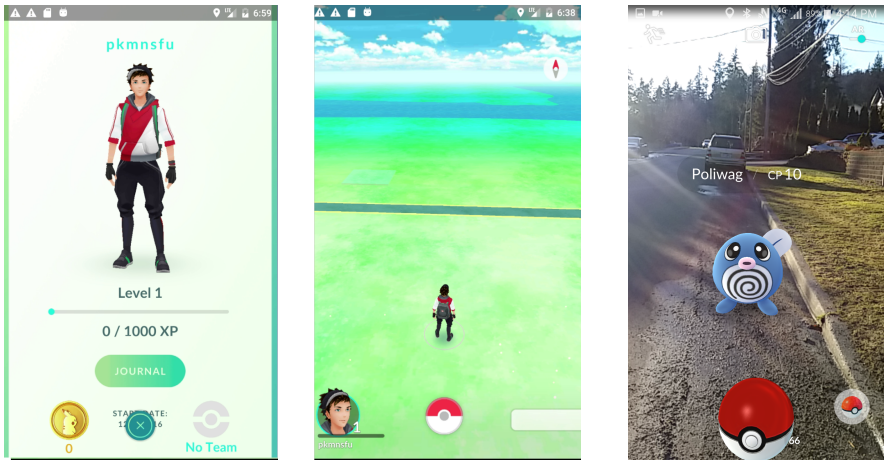
2.2 Client Architecture and Dataflow of Pokemon Go

In Figure 2, we provide an abstracted datapath of the augmented reality update process in Pokemon Go. The heart of the application is the update process to retrieve new Pokemon spawns and other map entities. We have discovered two processes to update map entities, a major update and a minor update. The major update occurs immediately post-login and retrieves all map entities, i.e. Pokemon, Pokestops, gyms, and spawn points, over a large area centered on the player. The minor update is identical to the major update, the difference being that the minor update receives only Pokemon, Pokestop, and gym data in a local area, and that it occurs more frequently than the major process, about once every 6 seconds as opposed to once every 60 seconds. Figure 2 describes the major update, where the client requests all map entities around a specified latitude and longitude. The request is composed of the player's current latitude and longitude coordinates, and a list of S2 cell IDs[3] to retrieve Pokestop and gym data for. Once the request is received by the server, it performs a displacement check between the current coordinates sent and the last coordinates received. If this displacement exceeds a threshold of what is considered physically possible, the player is silently temporarily banned from interacting with game entities for an unknown duration. If the displacement does not exceed this threshold, a secondary check occurs to determine if the displacement should be counted as valid for the game mechanics of egg incubation and badge credit. To pass this check, the displacement must not exceed 300 meters per minute. Since this distance check is not pertinent to producing a server response, we conjecture that this happens asynchronously while the server builds a response. To construct the response, the server takes the latitude and longitude of the request and constructs a list of all Pokemon currently active within a 200 meter radius of the player. If a Pokemon is less than 100 meters away from the player, the exact coordinates and time-to-live in milliseconds of the Pokemon are included as attributes. Otherwise, only the distance in meters is included as an attribute. Simultaneously, the server also takes the S2 cell IDs sent and constructs a list of all Pokestops, gyms, and spawn points. Pokestop entities contain an attribute for active modifiers (currently the only modifier is a lure module), gyms contain attributes for the current prestige, team owner, and the current highest combat power (CP) pokemon in the gym. Once both lists are generated, the response is sent to the client.

2.3 Mobile Client Power Consumption

During the initial distribution of Pokemon Go there were many reports of the app having a deleterious effect on the battery life of mobile devices. Motivated by these reports, we investigated and quantified the power usage of a smartphone running Pokemon GO, in order to determine what improvements might be introduced. We devised a measurement strategy involving a real world Android device, namely the Moto G 3rd Generation. Our test platform specifications include Quad-core 1.4 GHz Cortex-A53 CPU, a Adreno 306 GPU, 2 GB of RAM and 16 GB of internal flash memory. The device's operating system was updated to latest available Android version 6.0 (Marshmallow). We used the phone's built in battery discharge sensor and the measurement application *GSam Battery Monitor* to profile the Pokemon Go application. To make a stable testing environment we ran the Pokemon Go app for 30 minutes and collected the average battery discharge rate. The adaptive brightness setting of the screen was disabled to ensure that changes in the testing environments ambient environment would not effect the measurements.

[3] docs.google.com/presentation/d/1Hl4KapfAENAOf4gv-pSngKwvS_jwNVHRPZTTDzXXn6Q/



(a) Menu Scene (b) Game-play Scene (c) AR Scene

Figure 4: Different Scenes in a AR application

We find that under our testing conditions Pokemon Go uses a system wide power consumption of 3544 mW. Figure 3 illustrates the percent of energy and which subsystem is consuming it as well as a breakdown of which running apps are consuming power. As can be seen we have an even split at 49% for both the screen and apps, and only 2% being consumed by the radio. We find that the device's screen consumes exactly 1736.71 mW, Pokemon Go app 1169.62 mW, other apps 567.10 mW, and finally the radio only 70.90 mW. Our results make it clear that the augmented reality app itself has a very high energy consumption cost.

It is well established in the literature that the screen can be a large drain on the battery of a smart phone. Why the augmented reality app Pokemon Go in particular consumes so much power required further exploration. To that end we further profiled the application using the development platform *Android Studio*. By profiling the app we find that over 80% of the CPU cycles are being used by the function call UnityPlayer.nativeRender, which is responsible for processing 3D objects for display. We conjecture that this function call is likely where the in-game and AR objects are composed for viewing by the user, and that this task is extremely computationally expensive. Of the approximatively 20% remaining CPU time, the largest contributer is a function call to the Android system's "ContextService". The context service is responsible for gathering data from the sensors such as the GPS, accelerometer, and gyroscope. Pokemon Go makes heavy use of this service to feed data from the phone's sensors to the game engine to update the game world.

3 AUGMENTED REALITY VISUAL COMPONENTS

Broadly speaking, all software-based game applications contain at least two visual components: the menus and the gameplay window. The menu allows players to interact with the metacontrols (e.g., what type of Pokeball to use) and metagame (e.g., what Pokemon to send into battle), while the gameplay window provides an interface to the game itself by rendering the game world. If these components can be fully decoupled from each other visually they can be independently offloaded and rendered on cloud servers potentially increasing performance and saving battery life.

As we previously discussed, Pokemon Go is a location-based augmented reality game. It has two fundamental visual components, the menus and the gameplay window. Additionally, it contains an additional component not previously discussed, the AR gameplay window. The game contains one main menu and six auxilary menus that can be accessed by tapping the player icon on the bottom left or the pokeball in the bottom centre. These menus mainly load metagame knowledge such as the user's unlocked achievements, a microtransaction shop, purchased/owned items, and the user's captured Pokemon. An example of each type of visual component can be seen in Figure 4. Therefore, they are largely independent of the game state and engine. As a result, it is possible to factor out the menus and overlay them on top of the gameplay window. Conversely, the gameplay is also capable of being independently drawn without integrating the menus into the game window.

This is the core principle that our proposed offloading platform exploits to transfer the energy and computationally-heavy game engine to the cloud. The menus and gameplay window can be independently rendered by cloud servers and combined back into a single opaque video stream to the client. Interestingly, the same idea can be applied to the augmented reality gameplay of Pokemon Go as well, the physical scene that the virtual world is augmenting or building upon is independently observed. Virtual objects are placed relative to a central position, the virtual camera, whose absolute position is irrelevant with respect to the physical scene. Therefore, it is possible to remotely generate a transparency-enabled video stream of the virtual camera, and, locally on the client, combine the two scenes together to create an immersive experience.

This type of abstraction can be extended to all GUI-based software applications in a similar manner; instead of a gameplay window, it is an interactive canvas of the application view. For example, a web browser's menu would be the URL bar while the canvas

would be the displayed web page. Consequently, we propose a new archetype of cloud-enabled AR applications that offload some or all aspects of client rendering to the cloud. The computationally-heavy core logic of the application can be rendered on a cloud server and streamed back to the client, while the menus can be rendered natively or streamed alongside the application view. Augmented reality views can also be generated on a server and sent back to a client, however it does require the usage of an alpha channel.

4 CLOUDAR: DESIGN AND IMPLEMENTATION

As shown in the previous sections, AR applications with pervasive sensing capabilities such as Pokemon Go can consume significant battery power of mobile devices. Since the Pokemon Go infrastructure is largely built atop Google's cloud, there are great opportunities if we can offload more heavy lifting workloads to the cloud end. In this section, we present our design and implementation of a video streaming-based, cloud-offloaded AR platform. The platform is motivated by the following observations:

- While AR content rendering consumes the largest amount of power, due to the pervasiveness of video applications mobile devices contain power-efficient hardware chips for video processing (e.g., decoding and encoding); processing the AR content video stream can be much more power efficient than rendering the AR content locally.
- The cloud-based framework could significantly reduce the hardware requirements of the mobile devices. From the App developers' perspective, they do not have to deal with the vast heterogeneity of mobile devices, adapting and testing the game against different OS platforms.
- Hosting AR content generation in the cloud could substantially reduce the time-to-market of the App. It also reduces the complexity of applying patches and updates to the app for the game makers.

Offloading AR content rendering to the cloud is not trivial. First, given the rendering engine is located in the cloud, we need to send rendered objects from the cloud to the mobile device *in real-time*. In the meantime, we need to cast the user's input and the device's sensor data to the cloud to preserve the user interactivity. We need a mechanism to compose the rendered scene. Finally, the additional computation for handling the offloading at the mobile devices should consume less energy than the local-render scheme.

4.1 System Overview

Current augmented reality systems are rendered locally on the same device that contains the necessary sensors. This is done to allow the engine direct access to the raw sensor feeds to generate and project the virtual objects onto a scene. For example, Pokemon Go's AR system utilizes a mobile device's gyroscope and compass to project virtual Pokemon into the device's camera feed. However, this approach has a few drawbacks: it is extremely energy intensive and requires relatively powerful hardware to support the computations. We propose an alternate, cloud-based AR system that eliminates these disadvantages while providing a similar user experience.

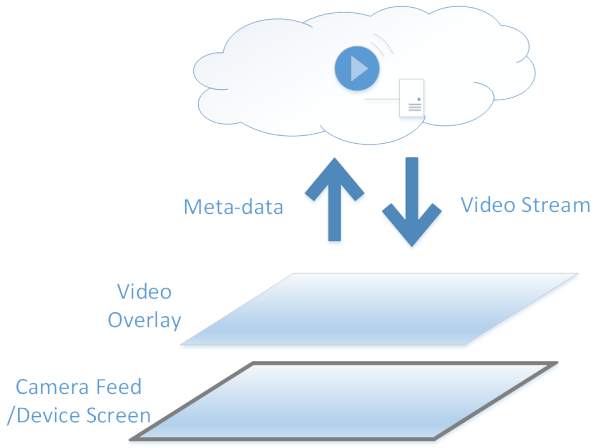


Figure 5: AR Video Overlay

In our system, the mobile device (thin client) provides and sends sensor data to the cloud, and receives an alpha channel enabled video stream as a response. The client is then responsible for rendering this stream as an overlay on top of the scene being augmented. The only hardware requirements at the client side are the sensors that are being used and video decoding capability. This technique minimizes the energy consumption by offloading all possible AR related computations to the cloud.

4.2 Implementation: AR Video Overlay (AVO)

As discussed in the previous section, the GUI presented to the user can be decomposed into several data-decoupled visual layers. With augmented reality, the main layers are the physical, underlying scene and the projection of the virtual world. In our case study with Pokemon Go, the AR scene shown in Figure 4c can be broken down into three separate layers:

- Menu/Control - This layer contains the visual elements that the user interacts with; it contains the item selection on the bottom-right corner and Pokemon metadata label in the centre of the screen.
- AR Overlay - The virtual world is projected onto this layer; it contains the Pokemon itself and the 3D Pokeball at the bottom-centre.
- Physical Scene - This is the captured scene from the camera, i.e., the base layer for the game scene.

Each of these layers can be rendered independently and with minimal knowledge of the other layers. Following this intuition, we constructed a prototype client designed to mimic the AR capabilities of Pokemon Go while being as unconstrained as possible with respect to hardware and software limitations. To achieve this, we designed a browser-based client using only widely available web technologies: HTML5, JavaScript, and CSS. The client essentially rebuilds the 3 layers locally, the physical scene is obtained from the device camera, the AR layer is abstracted into a video element (AVO) that is stacked on top of the camera, and any menu elements can be locally constructed via CSS.

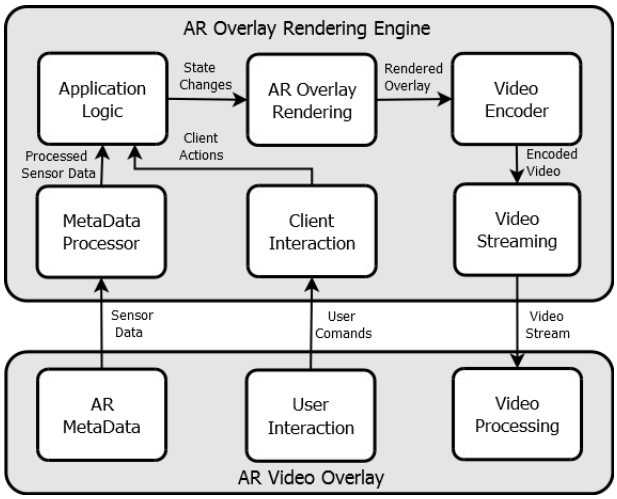


Figure 6: Architecture: Cloud and Thin-Client

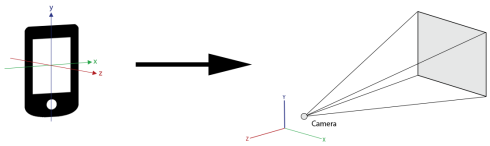


Figure 7: Controlling the virtual camera using CloudAR

By abstracting the AR content into a video overlay, the AVO achieves four major design goals: AR rendering is decoupled from the physical device, sensor data collection and processing are decoupled from each other, highly optimized video codecs can be leveraged to reduce data rates, and the device hardware requirements are alleviated, allowing for and accommodating wider device heterogeneity. The decoupled aspects are then offloaded entirely into the cloud, and in exchange, the client receives a rendered AR scene as shown in Figure 5. Additionally, as the AVO is ultimately rendered in a controlled server, a greater level of consistency and reliability in performance is guaranteed due to this decoupling from the mobile device. In the specific case of Pokemon Go, their minimum Android requirement could be reduced to Android 2.3 (Gingerbread) as opposed to 4.4 (KitKat) by utilizing this approach.

4.3 Implementation: CloudAR Server

In Figure 6, we depict the architecture of our cloud based AR streaming platform, CloudAR.

On the server side, the first two modules are the *MetaData Processor* (MDP) and the *Client Interaction*. These modules ingest, validate, and process the client data. In the case of AR, the MDP also performs the role of sensor fusion to predict and reduce noise from the sensor data. The *Application Logic* is essentially the game instance. It processes the sensor data and client actions from the previous two modules, and computes the updates to the game world, based on which, the rendering is then performed by the *AR Overlay Rendering* module. The rendered scenes are passed to the *Video Encoder* module that contains a video encoder and a discrete framer. The video encoder is selectable, consisting of either a software or hardware h264 or VP8 encoder. For scenes such as game-play and menus h264 encoding can be used. However, VP8 is required for the AR scenes as it is one of the few video compression formats that fully support an alpha channel. The encoded video stream is then encapsulated in the webM format and transported using web sockets to the client.

The AVO is generated by the Application Logic module as it constructs a virtual world using the mobile device's initial orientation in 3D space relative to the Earth. A virtual camera is set-up to mirror the viewing angle of the client and provides the viewpoint for the AR Overlay Rendering module. The virtual world's Y-axis is defined to be the vector pointing towards the magnetic north, Z-axis is defined to be the vector oppositely directed to gravity, and the X-axis is the cross product of Y and Z (See Figure 7[4]). Using this coordinate system, entities are then placed relative to the camera location. Consequently, as everything is relative to the camera, the relationship between 1 virtual unit to a physical unit is arbitrarily defined by the application and allows for complex, granular scenes to be constructed.

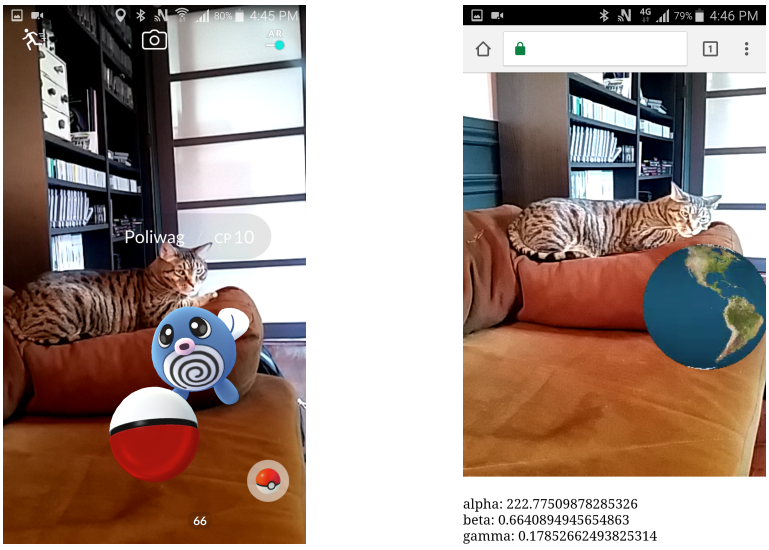
The AR Overlay Rendering module then takes the view from the virtual camera and outputs a sequence of raw RGBA bitmaps. The rendering is done through a headless OpenGL context using either the EGL API or an X server. Since most AR scenes feature few prominently displayed entities, the majority of what the camera sees is empty space. This empty space is removed using the alpha channel, which results in an image that is completely transparent in most areas. An interesting consequence is that the encoder can essentially ignore the RGB values of the majority of the frame, as the alpha channel is set to 255 for full transparency.

In practice, our implementation of the server utilized a NodeJS backbone due to having non-blocking I/O for the event-driven nature of incoming data. Communication between the server and client is handled using the WebRTC protocol, and FFMPEG is used as the Video Encoder module to transcode the output into a VP8 WebM stream. This architecture is highly scalable both vertically and horizontally. It can be scaled vertically by simply storing different virtual scenes and device orientations, while it can be scaled horizontally by adding more NodeJS servers and routing the data to the appropriate instance.

4.4 Implementation: CloudAR Mobile Client

Expanding on the implementation of the client mentioned in section 4.2, the mobile device's camera and orientation is accessed via JavaScript APIs and sent to the CloudAR platform. Once the user has granted access to the device camera, it is used as an HTML5 video source and displayed in real-time. Concurrently, the browser attempts to establish a SRTP connection to the CloudAR platform while continually updating the server with device orientation. Once this connection is fully established, a transparent VP8-encoded video stream is then set to be a second video source that is then overlayed on top of the existing camera scene. The alpha channel

[4]Visual elements from this figure were composed from: developer.android.com



(a) Pokemon GO AR Scene (b) CloudAR Scene (Cloud Rendered)

Figure 8: Augmented Reality Scene

in the AVO is a requirement since it allows the underlying physical scene to "punch through" the AR video element. Control elements, such as the item selection and Pokemon label, can be rendered locally by the browser as HTML elements on top of the video feeds since most, if not all, user interactions with these controls will transition from an AR scene to a menu scene. The end result is an AR system that provides the same functionality as our reference system, Pokemon Go at a lower energy cost. These energy savings are further enhanced in the presence of an enabled hardware decoder. By using these features we can enable resource-constrained (in terms of computation, memory, battery, etc.) mobile clients to run advanced applications.

The *AR MetaData* module is in charge of casting the various sensor data collected at the mobile device to the cloud server, interfaced with MetaData Processor module. Hence, when the video is received by a mobile client, our client-side *Video Processing* module is configured to utilize the hardware decoder with real-time optimizations to decode the video and display it on the client device. The *User Interaction* module supports input, which could include input from touch screens, game-pads, keyboards or mice.

Figure 8, shows a comparison between a AR scene from Pokemon GO and one rendered remotely using CloudAR. For CloudAR we designed a 3D spinning globe scene and render it in a specific location using the phone's sensor data.

5 CLOUD AR EVALUATION

The selected test system mobile device for game-play and menu scene offloading is a Moto G 3rd Generation smart phone which includes Quad-core 1.4 GHz Cortex-A53 CPU, a Adreno 306 GPU, 2 GB of RAM and 16 GB of internal flash memory. We updated the device's operating system to the latest available Android version 6.0 (Marshmallow). Due to the lack of a gyroscope on the Moto G, a second test device for the AR offloading had to be used. This

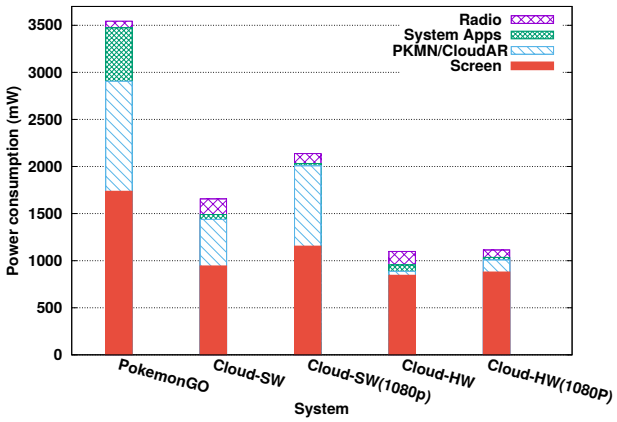


Figure 9: Game/Menu Scene Offload: Energy Savings

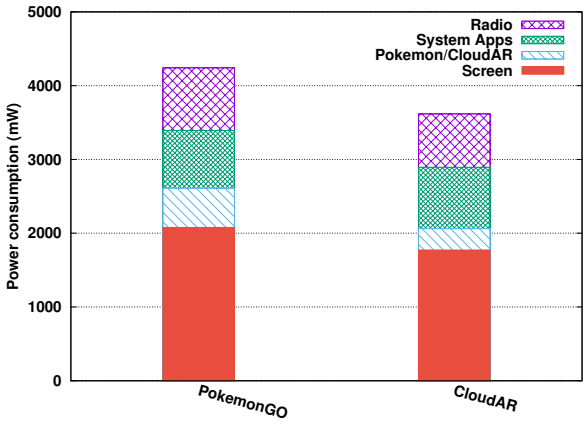


Figure 10: AR Scene Offload: Energy Savings

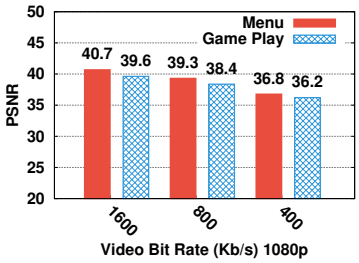
second device is a Samsung Galaxy S7 on Android 6.0.1, featuring a dual quad-core CPU (2.3 GHz M1 Mongoose, 1.6 GHz Cortex-A53), a Mali-T880 GPU, 4 GB RAM, and 32 GB of internal storage. The client browser used for the AR scene experiments is Google Chrome on version 55.0.2883.91.

For our cloud offloading server we leverage our research platform, SFUcloud, an advanced infrastructure as a service (IaaS) cloud. Physically SFUcloud is backed by 9000 GB of ram and 1000 logical CPUs. This processing power is split over three physical racks, with switching provided by 10 Gb/s Ethernet. The cluster is managed using Apache Cloud Stack and the Xen Hypervisor, which provides us reliable fine grained resource control. Our cloud offloading server instance was provisioned with 4x2.4 GHz Intel Xeon E5-2665 cores, 16 GB of RAM, and a NVIDIA GRID-K1 GPU. The GRID's on-board hardware video encoder was used to encode the h264 streams and vp8 encoding was done using FFMPEG on the CPU.

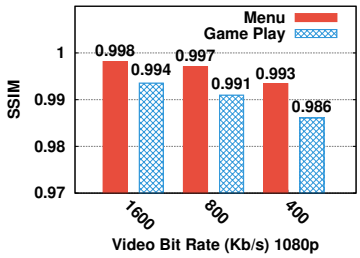
5.1 Mobile Device Power Profiling

In our first experiment we captured live frames from an instance of Pokemon Go running on our cloud server. The client sends GPS coordinates, which we use to update the player's location in Pokemon Go. We encode each frame with h264 and stream back to the client. We use h264 for this experiment as our test device supports hardware decoding of h264; this allows us to investigate the difference between hardware and software decoding in terms of energy consumption. In Figure 9, we depict the power profiling results on the Moto G test device when offloading menu and game-play scenes from Pokemon Go. Regardless of which video decoder is being used, CloudAR is able to substantially reduce the energy consumption of Pokemon Go. We see slight increases in energy consumption on the radio, which is due to the increased usage of radio for video streaming. Another part of the power saving comes from the screen. CloudAR is able to adjust the Frame Rate per Second (FPS) based on players' preference, helping reduce the energy consumption on the screen when players do not require high frame rate.

In our second experiment we use the VP8 encoder, as our AR scenes require an alpha channel to render it at the cloud side. The client streams the gyroscope and compass sensor data to our cloud server. We employ the spinning globe scene from Figure 8b as our AR scene. Figure 10 shows the results of offloading augmented reality scenes on the Galaxy S7. Overall, the browser-based CloudAR client consumed 15% less energy than the Pokemon Go client. These numbers are fairly conservative as there is a non-trivial amount of energy consumption due to using a browser, as seen in the 6% increase in the system applications. Consequently, these results show that even in the worst case, there is still a sizable energy consumption benefit to offloading AR scenes. Further, it is likely a client implementation using a native app would have a lower energy consumption than what was observed, as we have considerable overhead due to the use of Chrome. As a future work we plan to implement a fully native Android application to test this hypothesis.



(a) PSNR 1080p



(b) SSIM 1080p

Figure 11: Menu/Game Play Offload (Image Quality)

BitRate	1600Kb/s	800Kb/s	400Kb/s	200Kb/s
PSNR	47.7	47.0	45.3	43.3
SSIM	0.9997	0.9996	0.9990	0.9982

Table 1: AR Offload VP8 720p (Image Quality)

5.2 Streaming Quality

We describe the streaming quality of CloudAR in Table 1, which states the image quality and bit-rate of the AR Scene offload portion of CloudAR. We analyze the video using two classical metrics, namely *Peak Signal-to-Noise Ratio* (PSNR) and *Structural Similarity Index Method* (SSIM).

The PSNR and bit rate of the menu and game-play scene offloads are given in Figure 11a, and the SSIM are given in Figure 11b. The PSNR method quantifies the amount of error (noise) in the reconstructed video, which has been added during compression. The SSIM method calculates the structural similarity between the two video frames. In terms of both metrics, CloudAR is able to attain high streaming quality even at the low bit rate, which allows our system to be used without excessive bandwidth requirements. Offloading processing tasks to the cloud helps alleviate a key limitation of the PokemonGO app, namely its reputation for rapidly draining batteries. This offloading also frees up resources and battery life for other tasks such as running advanced sensors.

Finally, in terms of bandwidth usage sending sensor data to the cloud requires less than 2 KB/s. Consequently, the overall bandwidth of CloudAR is approximately the chosen video bit rate with an additional 2 KB/s for sensor meta data.

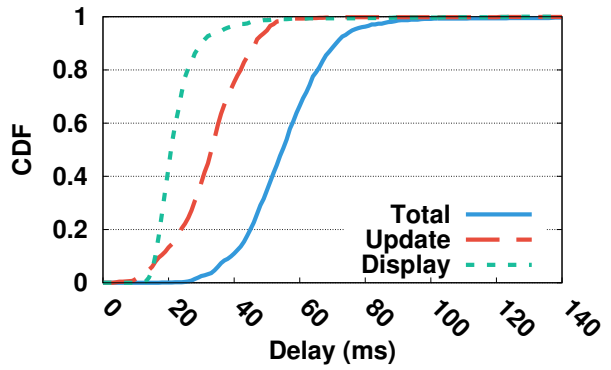


Figure 12: Interaction delay

5.3 Interaction Delay

Using the same prototype client, we obtained the total interaction delay by summing up the server update delay and the client display delay. We define the server update delay as the delay between sensor data updates and the receiving of the corresponding AR overlay frame. We define the client display delay as the delay between receiving the new overlay frame and the next full repaint. We obtained these measurements by using a high-resolution timer that is accurate to 5 microseconds on the client and half a microsecond on the server. The results are plotted in a CDF shown in Figure 12. The total interaction delay of CloudAR is 55 ms in the 50th percentile and 96 ms in the 99th percentile. Similarly, the client display delay and the server update delay had 50th and 99th percentile times of 22 ms/41 ms and 33 ms/54 ms, respectively. Finally, our network RTT from client to server was stable in the experiments with an average of 8.4 ms.

The client display delay is entirely bounded by the frequency of the browser's internal repaint frequency, which we found to be 60 Hz or 16.67 ms. This means that this portion of interaction delay is largely determined by when a video frame is received relative to the next browser repaint. Theoretically, if the incoming frames perfectly align with the browser's repaint, the client display delay will effectively be reduced to a few milliseconds to process the incoming pixel data. Practically, the effective client display delay is the browser's repaint frequency, 16.67 ms, plus the true display delay, which in our case turns out to be about 5 ms.

On the other side, the server update delay is largely determined by the networking delays. Rendering a single frame from the virtual camera takes an average time of about 3 ms, while encoding the frames into a video stream takes about 12 ms. These aspects combined make up for about a third of the observed 41 ms update delay. Interestingly, this shows that CloudAR is capable of pushing 60 FPS scenes to the client. Finally, like any offloading technique our interaction delay is greatly affected by the overall network latency. However, given our measured average interaction delay of 55 ms we conjecture we can support network RTTs of up to 45 ms with minimal impact on QoE.

6 FURTHER DISCUSSION AND CONCLUSION

We have examined and measured the performance of offloading augmented reality scenes in addition to proposing a generic cloud offloading framework. The results show that the potential energy savings of remotely rendering scenes far outweighs the relatively small interaction delays. These AR offloading techniques are not limited to games, but can likely be applied to many applications. With the increase in battery constrained devices and high quality network availability, cloud offloading architectures appears to be a strong contender for future augmented reality applications.

There are many directions for future research in scene-based cloud offloading. Investigation and testing is needed on localized object-recognition based AR use cases. The AR scenes described in this paper are geolocation-based and displayed based on an estimated distance from the user's current geocoordinates. Similarly, local depth-of-field is not being considered while rendering the virtual entities. An attractive avenue of "full" augmented reality is then possible once object distance and local field topology can be established from the local camera.

Conversely, another direction for further work would be to analyze some of the byproducts of cloud offloading, such as cheat protection and digital rights management. Since the core logic of an application is run on a cloud server, a malicious user is blocked off from directly interacting with the application's memory space and resources, thereby removing any possibility for software cracking or botting. Additionally, as we have shown the immense potential of cloud offloading, we conjecture there is a strong case towards the inclusion of this use case in many existing World Wide Web Consortium specifications, predominately in the addition of an alpha channel in WebRTC video streams.

In conclusion, our prototype intelligent thin client is capable of producing an AR scene that is nearly identical to Pokemon Go. In one experiment, our client drew considerably less energy in comparison to Pokemon Go while presenting an image with a structural similarity of 99.4% at 1080p using our offloading techniques. In another, the client had an average end-to-end interaction delay of 55 ms and provided a low latency user experience with respect to the integration of the virtual and physical scenes. The result is a system that achieves excellent video quality with low interaction delay, while providing significant energy savings.

ACKNOWLEDGMENTS

This work was supported by an Industrial Canada Technology Demonstration Program, an NSERC Discovery Grant, and an NSERC E.W.R. Steacie Memorial Fellowship.

REFERENCES

[1] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. 2016. The Future of Cloud Gaming [Point of View]. *Proc. IEEE* 104, 4 (2016), 687–691.

[2] Hal Hodson. 2012. Google's Ingress game is a gold mine for augmented reality. *New Scientist* 216, 2893 (2012), 19.

[3] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2016. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 5.

[4] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. GamingAnywhere: an open cloud gaming system. In *Proceedings of the 4th ACM multimedia systems conference*. ACM, 36–47.

[5] Zhanpeng Huang, Weikai Li, Pan Hui, and Christoph Peylo. 2014. CloudRidAR: A cloud-based architecture for mobile augmented reality. In *Proceedings of the 2014 workshop on Mobile augmented reality and robotic technology-based systems*. ACM, 29–34.

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

[7] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. 2016. Low Bandwidth Offload for Mobile AR. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 237–251.

[8] Li Lin et al. 2014. Liverender: A cloud gaming system based on compressed graphics streaming. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 347–356.

[9] Fangming Liu, Peng Shu, Hai Jin, Linjie Ding, Jie Yu, Di Niu, and Bo Li. 2013. Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications. *IEEE Wireless communications* 20, 3 (2013), 14–22.

[10] Nayyab Zia Naqvi, Karel Moens, Arun Ramakrishnan, Davy Preuveneers, Danny Hughes, and Yolande Berbers. 2015. To cloud or not to cloud: a context-aware deployment perspective of augmented reality mobile applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 555–562.

[11] B. Richerzhagen, D. Stingl, R. Hans, C. Gross, and R. Steinmetz. 2014. Bypassing the cloud: Peer-assisted event dissemination for augmented reality games. In *Proc. IEEE International Conference on Peer-to-Peer Computing (P2P)*. 1–10.

[12] Mahadev Satyanarayanan. 2015. A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets. *GetMobile: Mobile Computing and Communications* 18, 4 (2015), 19–23.

[13] Ryan Shea, Jiangchuan Liu, Edith C-H Ngai, and Yong Cui. 2013. Cloud gaming: architecture and performance. *IEEE Network* 27, 4 (2013), 16–21.

[14] Bowen Shi, Ji Yang, Zhanpeng Huang, and Pan Hui. 2015. Offloading Guidelines for Augmented Reality Applications on Wearable Devices. In *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 1271–1274.

[15] Luke Stone. 2016. Bringing Pokémon GO to life on Google cloud. https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html. (29 09 2016).