AUGMENTED REALITY STREAMING ON COMMODITY MOBILE PHONES USING AR CORE

ABSTRACT

The advancement in cloud technologies has enabled video and game streaming possible today. These streaming technologies offloads the heavy processing to an edge server so that the client does not need high-end hardware to support it. Augmented reality applications are becoming more widespread, but only high-end flagship mobile phones provide the support to run AR applications. To support AR applications, mobile phones are equipped with high-end processors and sensors. In Android, one of the basic requirements of the mobile device is to have AR Core to support ARCore applications. In the proposed system, we aim to prove that streaming computationally intensive ARCore applications on commodity mobile phones without ARCore support from a remote or edge server is possible.

1. INTRODUCTION

To support ARCore, the mobile device needs to have a fast CPU and better calibration of the camera and IMU[1]. These factors limit ARCore support to flagship mobile phones and are not supported by old devices or commodity mobile phones. Streaming AR applications from a remote server is a lot more challenging than video or game streaming. AR applications require various kinds of inputs to determine the scene or environment around the user. It is difficult to offload the AR computation to a remote server because we have to make the remote server understand the surrounding environment of the user. So we need to send all possible inputs like camera feed, sensor values to the server. Streaming AR applications would enable a wide range of devices to run ARCore applications.

2. ARCORE

ARCore uses three key capabilities to integrate virtual content with the real world. They are motion tracking, environmental understanding, and light estimation. Motion tracking allows the phone to track its position relative to the world. Environmental understanding allows the phone to detect the size and location of all types of surfaces. And light estimation allows the phone to estimate the environment's current lighting conditions. ARCore works by tracking the position of the mobile phone as it moves through space and builds its own understanding of the real world. It finds key points called features and tracks how those points move over time. With these features and the data from the mobile phone’s IMU (Inertial sensors), the ARCore can determine both the position and orientation of the phone as it moves through space [12].

There are a few problems with ARCore. For a mobile device to support ARCore, it needs to pass a certain certification from Google, who developed ARCore. Google checks the quality of the camera, motion sensors, and the design architecture to ensure it performs as expected to support ARCore. Mobile phones need to have fast and powerful enough CPU [13] to run the map-building algorithms. The mobile phones need better calibration of the camera and IMU. Also, they need to provide consistent timestamps across exposure & CPU load [1]. Further sensor requirements to make a mobile phone ARCore ready can be seen in this reference link [14]. These requirements make it difficult for ARCore to be available in all the devices and thus only a limited number of devices support ARCore. The solution that we propose eliminates these requirements, since mobile phones need not support ARCore but have a basic camera and sensors that send data to remote servers and receive the streamed AR content.

3. PROPOSED SYSTEM

Since ARCore is not open-source, we need to build a system around it for achieving our goal. The basic idea is to make an android virtual device with ARCore support running on a remote server have a virtual environment similar to the real environment by providing the virtual device all the necessary inputs like the camera feed and sensor values from the client mobile device. This would create a clone of the client's mobile device but with higher specs on the server. Later the output from the virtual device is screen recorded and is streamed back to the client.

4. SYSTEM ARCHITECTURE

The proposed system consists of a mobile device as a client and a remote or edge server with a virtual device that supports ARCore. The client forwards its sensor values like Accelerometer and Gyroscope to the server. It also streams the camera feed to the server via a UDP or RTSP protocol. The sensor and camera inputs are necessary in order to simulate the user's environment in the server. An android virtual device (AVD) is created in the server with ARCore enabled. The server receives the camera stream and forwards it to a virtual camera which is used as the camera source by the android virtual device. The incoming sensor values are also set to the virtual device. Using these values, the AR application installed on the AVD can detect the surrounding environment. Then the output from the AVD is streamed back to the client via a UDP or RTSP protocol. The client device can use the streaming URL to receive the output of the AVD from the remote server. Figure 1 represents the entire architecture of the proposed system.

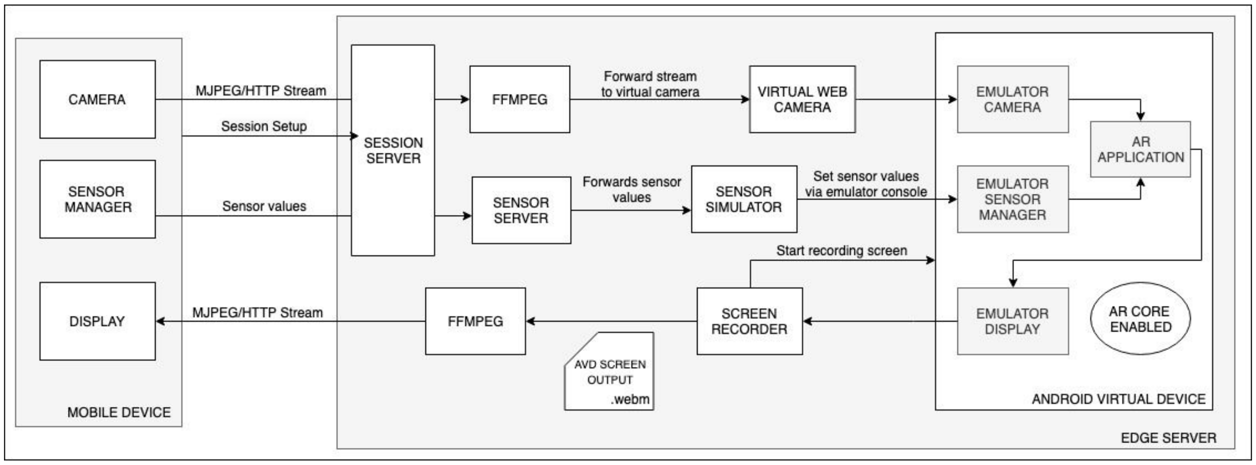

Figure 1: System Architecture

5. PROOF OF CONCEPT

To make sure that the proposed idea would work, we exercised a small experiment. We used an Ubuntu machine as our server and installed Android Studio. We created an android virtual device and enable it to support ARCore. We installed a hello_ar_java sample ARCore application from Google for testing the system. We then used a virtual camera device as the camera source for the android virtual device and fed sample footage to the virtual camera. Now we tried moving the AVD's 3D model to simulate real movement. We found out that, one out of five times, the sample ARCore application was able to detect key points in the footage and event detect surfaces. Usually, a virtual environment provided by Android Studio is used to test ARCore applications. This helped us to come to the conclusion that video input is accepted by the ARCore application in a virtual device.

We used an Android application [2] that would record video as well as all sensor values. The sensor values were saved in a CSV file and the footage was saved as an MP4 file. These files were used to simulate the video and sensor feed in the virtual device for determining if the proposed system would work.

6. CLIENT SETUP

6.1. CAMERA FEED STREAM

We were able to find many live protocols that provide real-time video transmission over a network. MJPEG, HLS (Http Live Streaming), RTSP/RTP, MPEG-DASH, MPEG-TS, WebRTC were some popular protocols that we considered to use [15]. We were able to find that most protocols had been designed to stream data to multiple clients at adaptive bit-rates and heavily compress/encode data leading to latencies of more than a second. These protocols were useful for broadcasting sports events, streaming movies and many more. But such latencies would render any live AR app unresponsive and difficult to use. In order to minimise latency (at the cost of increased bandwidth), we found MJPEG over Http to be the best streaming format, since it does not involve heavy compression. This protocol is widely used in IP cameras which cannot make heavy computations and stream video to the local servers.

We tried implementing the following 2 protocols, to see which would suit our use case best.

6.1.1 RTSP Streaming

Streaming protocol supported by popular media frameworks, such as VLC, FFMPEG. Widely used to control media streams over RTP, using methods options, record, play, describe, setup, announce, pause [16]. RTSP does not deliver data, though the RTSP connection may be used to tunnel RTP traffic for ease of use with firewalls and other network devices. RTP is a transport protocol for the delivery of real-time data, including streaming audio and video. RTP is used widely used in VoIP and video conferencing applications.

RTSP Protocol steps,

- The client requests Session Descriptor from the server, specifying media format.
- The server transmits RTP packets in conjunction with RTCP.
- The client uses RTSP to control the media stream.

Latency - approx. 1s, mostly due to delays caused by mp4 video compression buffer and client-side buffer used to reduce jitter. Jitter is caused by data not arriving in order over UDP resulting in the need for a buffer at the client.

6.1.2 MJPEG Streaming

This protocol is widely in IP cameras which cannot make heavy computations and stream video to the local server [17]. Each image frame along with timestamp is transmitted over HTTP in a persistent connection. In our setup, the protocol follows these steps,

- Android’s Camera preview is sampled at 30 FPS (ARCore supports 30FPS/60FPS) through the Camera API.
- The server requests stream from the device through an HTTP request.
- The device compresses each frame into a JPEG byte stream, which is included in the body of a multipart/x-mixed-replace HTTP response.
- The server hosts a media framework/tool like FFMPEG to read the MJPEG stream.

6.2. SENSOR VALUES STREAM

We initially created an Android application that would stream accelerometer and gyroscope sensor values. The application requires the IP address of the remote server and the port were the sensor simulator server is listening to. Using the sensor manager, the sensor values are received and using a socket, the values are sent to the remote server. The sample rate that we found to obtain the best frequency was with the SENSOR_DELAY_FASTEST hint, which yielded a frequency of around 500 to 1000Hz.

6.3. APPLICATION

We created an application that would stream the camera feed and the sensor values together. A client at the mobile device would connect to a server on the remote machine (where AR rendering will be done) to stream sensor values. A server on the mobile device would then be initialized to stream camera feed to the remote machine. The Camera API's onPreviewFrame listener is used to get a byteArray representing a Camera frame, this is then compressed to a JPEG format and added to the HTTP response. The SensorManager API's onSensorChanged listener is used to obtain sensor values that are streamed to the remote machine, through a TCP socket.

In order to manually synchronize both sensor and camera streams, we introduced a buffer at the mobile device that could be controlled through a slider to inject delays in either of the streams.

7. SERVER SETUP

7.1. ANDROID VIRTUAL DEVICE

In the server, an Android Virtual Device is created using Android Studio. Since ARCore is best supported in Google's Pixel devices, we create a pixel virtual device with a target of Android 8.1 and an API level of 27. To support AR applications created using Sceneform, we have to set the OpenGL ES of the emulator to 3.0 or higher. To support ARCore, we have to install Google Play Services for AR into the virtual device. Now the virtual device is ready to support ARCore applications. The steps to set up the virtual device and enable ARCore is given in reference [5]. By default, the camera is set to a virtual scene in the virtual device, but here we'll be providing the camera's input as a video stream. So we set the back camera of the virtual device to the virtual camera device created in section 6.2.

7.2. VIRTUAL CAMERA

For an AVD, we can use a webcam as the virtual device's camera. By doing this, the virtual device's camera shows the output of the machine's web camera. So we try to create a virtual web camera device and set it as the camera source for the android virtual device. The virtual camera is created using v4l2loopback. V4l2loopback is a kernel module to create V4L2 loopback devices. This open-source module allows us to create virtual video devices that appear as physical devices to applications while taking input from an application. Applications will read these devices as if they were ordinary video devices, but the video will not be read from a capture card but instead, it is generated by another application [6]. This allows us to send the client's camera feed into the virtual device. To install and create a virtual camera device, follow the steps in the reference link [6] & [8]. To list the available virtual devices, use the command,

v4l2-ctl --list-devices

In order to feed the client's camera feed into the virtual device, we need to use an encoder/decoder module. We use FFmpeg [7] to convert the camera feed stream to a suitable format accepted by the virtual camera device. FFmpeg is an open-source collection of libraries and tools to process multimedia content such as audio, video, subtitles and metadata. The FFmpeg converts the incoming UDP/RTSP stream to v4l2 format supported by the virtual camera device.

To test if the virtual camera device is streaming the video properly, use the following commands,

ffmpeg -re -i <sample_footage>.mp4 -map 0:v -f v4l2 </dev/video0> (for video file)
ffmpeg -f video4linux2 -input_format mjpeg -i rtsp://<stream_url> -f v4l2 </dev/video0> (for streaming URL)

NOTE: *The angle brackets are not included in the command. It represents that the parameter varies depending on the file name or streaming URL or the virtual device name.*

Use the command ***ffplay /dev/video0*** to see the output of the virtual camera device. If the AVD's camera is set to /dev/video0, opening the camera application of the AVD would show the output of the virtual camera device.

7.3. SENSOR SIMULATOR

A simple java server is set up in the remote server which listens to port 5000. The client sends the sensor values via a web socket. Once the sensor values are received by the server, a sensor simulator script sets these sensor values to the virtual device using the emulator console [9]. It uses telnet to access the emulator console. After authentication of the emulator console using the auth command, the sensor values are set using the command, ***sensor set <sensor_name> x:y:z***. Using these commands, we can set the accelerometer and gyroscope values. To verify if the virtual device's sensor values are getting set, go to the extended controls of the virtual device and click the Virtual sensors menu item. This will show you the real-time sensor values of the virtual device.

7.4. SCREEN RECORDER

To get the AR applications output back to the client, we again use the emulator console of the virtual device. Using telnet, we access the emulator console and use the command, ***screenrecord start /path_to_save_file/output.webm***. Using this webm file and the FFmpeg module, we can stream the video file to the client. The streaming URL can be used by the client to view the AR application's output.

8. EXPERIMENTAL SETUP

As the client device, we used a Google Pixel 2 device. Even though the device supports the ARCore application, we did not install any AR application in the device. The IP Camera and sensor streaming applications are installed and is run in the background.

For the server, we used a machine with Ubuntu 16.04 OS and installed the latest Android Studio. We created a Google Pixel 2 virtual device with a target of Android 8.1 and an API level of 27.

We installed the Google_Play_Services_for_AR_1.12.1_x86_for_emulator.apk [10] version 1.12.1 to enable ARCore in the virtual device. We installed the hello_ar_java sample ARCore application [11] on the virtual device. This application was used as the test ARCore application that would compute and stream the output to the client.

9. OBSERVATIONS

9.1. SENDING CAMERA FEED AND SENSOR VALUES TOGETHER

In the proof of concept, the system only worked one out of five times. But, by sending real-time camera feed along with the sensor values of the mobile phone to the server, and setting both the virtual camera and the sensor values simultaneously, improves the keypoint detection, surface detection and retention of the surface by the ARCore application. But the system is still inconsistent because the detected surface vanishes when the phone's view changes and comes back to the same position in a few trials.

9.2. SYNCHRONIZING CAMERA AND SENSOR INPUTS TO AVD

To reduce the inconsistency in surface detection, we tried to synchronize the camera feed and the sensor values that are fed into the android virtual device. This is because one of the requirements of ARCore is to provide consistent timestamps of camera and sensor input. Since the camera and sensor input feeds are injected into the AVD through different protocols and tools, the streams are bound to be out of sync.

We had two approaches to deal with this problem. We tried introducing delays in the streams manually, and observe changes in the AR application. So we built a delay mechanism in the client streaming application. This would help us delay either the camera stream or the sensor stream. We experimented by delaying the sensor stream using a buffer and see if we could match the latency between the two input streams using trial and error method. In a few cases, it did improve the performance, but still, it was inconsistent. This is because the latency is due to a number of factors like bandwidth of the network, the load on the server and many more. So the latency tends to change over time and is not constant.

Another way we could try dealing with this problem is to synchronize AVD inputs, through a native implementation built on top of the emulator. But this is something that can be done in the future to improve the performance and for a better solution.

10. FUTURE WORK

The system has a lot of areas of improvement. We need to optimize the streaming pipeline to have a better synchronization of camera feed and sensor value injection into the AVD. By minimizing the latency and synchronizing the camera feed frame with the sensor value, the surface detection and retention rate can be improved.

In the client end, the camera feed streaming, sensor streaming, and output video display can be made into a single universal application. Also, this application can be made to send the touch inputs to the server and make it more interactive.

In our study, we experimented on motion tracking and environmental understanding of the ARCore while streaming. We need to study more about the effects of light estimation and understand it's requirements like any additional sensor input.

11. CONCLUSION

Given more access to ARCore itself, the system could be improved in numerous ways. Even tapping into the source code of the Android emulator could help us build a more working solution. But from the working of the system, we can conclude that it is possible to stream ARCore applications from a remote server to a mobile device. Even though the system doesn't work perfectly, it clearly proves that the concept of streaming AR applications is possible.

12. REFERENCES

[1] https://github.com/google-ar/arcore-android-sdk/issues/217
[2] https://github.com/e-lab/VideoSensors
[3] https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en
[4] https://github.com/fyhertz/spydroid-ipcamera

[5] https://developers.google.com/ar/develop/java/emulator

[6] https://github.com/umlaeute/v4l2loopback

[7] https://github.com/FFmpeg/FFmpeg

[8] https://askubuntu.com/questions/881305/is-there-any-way-ffmpeg-send-video-to-dev-video0-on-ubuntu

[9] https://developer.android.com/studio/run/emulator-console.html

[10] https://github.com/google-ar/arcore-android-sdk/releases/download/v1.12.1/Google_Play_Services_for_AR_1.12.1_x86_for_emulator.apk

[11] https://github.com/google-ar/arcore-android-sdk/tree/master/samples/hello_ar_java

[12] https://developers.google.com/ar/discover

[13] https://developers.google.com/ar/discover/supported-devices

[14] https://github.com/google-ar/arcore-android-sdk/issues/594

[15] https://www.wowza.com/blog/streaming-protocols-latency

[16] https://searchvirtualdesktop.techtarget.com/definition/Real-Time-Streaming-Protocol-RTSP

[17] https://stackoverflow.com/questions/14815103/android-streaming-the-camera-as-mjpeg