Power Efficient Mobile VTuber Live Streaming

Zichen Zhu
Rutgers University
Piscataway, NJ, USA
zichen.zhu@rutgers.edu

Stefano Petrangeli
Adobe Research
San Jose, CA, USA
petrange@adobe.com

Viswanathan Swaminathan
Adobe Research
San Jose, CA, USA
vishy@adobe.com

Sheng Wei
Rutgers University
Piscataway, NJ, USA
sheng.wei@rutgers.edu

ABSTRACT

Virtual YouTuber (VTuber) live streaming, which renders and streams a virtual avatar of the real-person streamer on top of the live camera view, has gained significant popularity recently. Despite the engaging user experience, the intensive and power-consuming computations required by VTuber, such as facial feature extraction and avatar rendering, pose significant challenges to the constrained battery life of the mobile device. We develop a power efficient VTuber live streaming system by offloading the camera view and the computation-intensive operations from the mobile device to an edge server, which not only significantly reduces the power consumption of the mobile device but also enables larger-scale rendering of multiple avatars that are not feasible in the existing mobile VTuber systems. Furthermore, to reduce the bandwidth overhead caused by the camera view offloading, we develop an adaptive framerate control mechanism to dynamically adjust the framerate of the offloaded camera view based on the variations of inter-frame luminance. Our evaluations on the end-to-end VTuber live streaming system demonstrate significant power savings with limited bandwidth, latency, and quality overhead.

CCS CONCEPTS

• **Information systems → Multimedia streaming**.

KEYWORDS

Live streaming, virtual YouTuber, power efficiency

ACM Reference Format:
Zichen Zhu, Stefano Petrangeli, Viswanathan Swaminathan, and Sheng Wei. 2023. Power Efficient Mobile VTuber Live Streaming. In *ACM Multimedia Asia 2023 (MMAsia '23), December 06–08, 2023, Tainan, Taiwan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3595916.3626427

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MMAsia '23, December 06–08, 2023, Tainan, Taiwan
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0205-1/23/12…$15.00
https://doi.org/10.1145/3595916.3626427

1 INTRODUCTION

Virtual YouTuber (VTuber) has become a popular live streaming application on commercial video streaming platforms, such as YouTube and Twitch [3]. Different from traditional live streaming that directly broadcasts the camera view of the streamer, VTuber streaming renders a virtual avatar that synchronizes with and replaces the real person streamer in the camera view in real time, which provides a mix of virtuality and reality experiences to engage the viewers [18, 29]. Since its emergence in 2010, VTuber has become a unique and important category of video content providers leading to rapidly growing cultural communities and commercialized industry [29]. The trend has been witnessed by major video streaming platforms. For example, VTuber content on Twitch increased 467% year-over-year in 2021 [4], and VTuber views on YouTube grew to over 1.5B views per month by October 2020 [2].

Figure 1 shows a generic workflow of VTuber live streaming. The real-person streamer's facial expressions and movements are captured by a webcam and processed in real time by the VTuber software to generate the virtual avatar, including face detection, facial feature extraction, and avatar rendering. The rendered VTuber frames are then pushed to the streaming server and streamed to remote viewers. During this process, the VTuber software involves sophisticated image/video processing and computer vision computations, which are typically conducted on powerful desktop computers to ensure the real time performance. However, it constrains VTuber streaming to an indoor environment with physical access to a desktop computer but little interaction with the physical world, resulting in non-satisfactory viewing experiences [29].

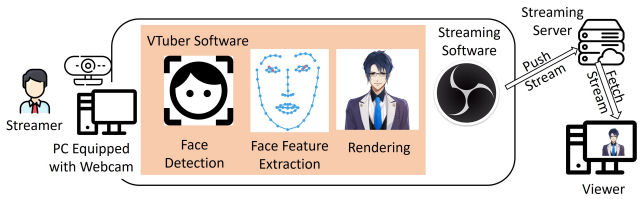

Figure 1: Generic workflow of VTuber live streaming.

To address this key limitation of deployment, the recent developments of VTuber software, such as VTube Studio [15], have supported the mobile mode of VTuber streaming, where the VTuber processing and rendering tasks can be executed in a smartphone application. However, the intensive computations required

by the VTuber software pose significant challenges to the resource-constrained mobile device. First, the power consumption caused by the intensive computations would drain the limited battery of the smartphone at a faster speed, resulting in reduced streaming duration and thus downgraded VTuber/viewer experiences. Second, the limited computation resource on the smartphone and the real time requirement would make it difficult for co-streaming involving multiple avatars, which is a desired feature of VTuber streaming [29]. For example, VTube Studio only supports a single avatar being rendered and streamed without the co-streaming feature [15].

We develop a power efficient mobile VTuber live streaming framework, namely *PEV*, to address the aforementioned technical challenges. The main design principle of *PEV* is to offload the computation-intensive operations, namely face detection, facial feature extraction, and avatar rendering, from the resource-constrained mobile device to a resource-rich edge server for power savings. To achieve the offloading-based computation, *PEV* passes the mobile-end camera view to the edge server for processing and rendering of the virtual avatar. Furthermore, to reduce the bandwidth consumption between the mobile device and the edge server caused by the camera view offloading, we develop an adaptive framerate control algorithm to dynamically adjust the framerate of the camera view in transmission based on the variations of inter-frame luminance, which reduces the quality impact on the rendered virtual avatar.

We implement *PEV* in an end-to-end VTuber live streaming system and compare it with various modes of state-of-the-art VTuber systems in terms of power efficiency and bandwidth/latency/quality overhead. Overall, our results demonstrate that *PEV* achieves significant power savings compared to the functionally equivalent baseline systems with limited overhead. Furthermore, the offloading-based design enables *PEV* to support co-streaming with multi-face rendering without increasing the on-device power consumption. To summarize, we have made the following contributions:

- To the best of our knowledge, *PEV* presents the first power efficient VTuber live streaming study in the community.
- The edge offloading-based solution does not only achieve power savings but also enable new computation-intensive features (e.g., multi-avatar) and eliminate the indoor limitation in VTuber streaming with minimum overhead.
- The proposed inter-frame luminance-based framerate control mechanism effectively reduces the bandwidth overhead of *PEV* while maintaining acceptable rendering quality.

2 BACKGROUND AND RELATED WORK

2.1 VTuber Avatar Generation

To achieve the desired experiences of virtuality and reality, the VTuber streaming system requires a sequence of three computation-intensive tasks to generate the virtual avatar in real time based on the real-person streamer's camera view, as illustrated in Figure 1. Taking the popular commercial software VTube Studio [15] as an example, the three tasks work as follows. **Face Detection** locates the face to be rendered in the input video frames. In VTube Studio, the OpenSeeFace library [10] is used to process the input video frames from the webcam, which employs a neural network to detect faces in the frame and outputs the coordinates of the detected face. The coordinates are then utilized to crop face images from the input video frames in the next step. **Facial Feature Extraction** employs a neural network that takes cropped face images as input and outputs the landmarks of the faces representing key facial points. The landmarks are then transformed into facial features that describe the facial expression. **Rendering** renders the virtual character based on the extracted facial features that drive the facial movements. Since different people may have different ranges of motions for facial expressions, the renderer scales the input facial feature parameters into the virtual character's range for rendering. The three tasks involve computation-intensive machine learning or video processing tasks and are thus not suitable for execution on a resource-constrained mobile device, in terms of the power efficiency and the real time requirement. Therefore, in the design of *PEV* we propose an edge offloading-based approach to accommodate the required computations.

2.2 Related Work

VTuber Live Streaming as an emerging and rapidly growing streaming application is still in its infancy in terms of research in the community. Several early research works so far have mainly focused on understanding the social and technical perspectives of the VTuber applications and the cultural community [18, 29]. Several other works focus on developing a specific VTuber [25] or using AI or computer vision techniques to accomplish character generation in VTuber streaming [20]. To date, there have not been intensive studies of the VTuber application in terms of the important streaming optimization topics, such as rate adaptation, live latency, and power efficiency. **Power Efficient Streaming** has been widely studied in the past decade for both 2D and immersive videos. Hans *et al.* [19] and Jiang *et al.* [24] reveal the energy consumption issue in mobile gaming and immersive video streaming. Many research works have been devoted to optimizing the power consumption in traditional 2D video streaming using a variety of hardware and software approaches [17, 21, 27, 28, 33, 34]. More recently, power efficient immersive streaming or VR/AR applications have drawn great attention in the research communities [16, 23, 26, 32]. Among them, one category of research employs edge offloading to optimize on-device power consumption due to rendering or deep learning computations [26, 35]. To date, the power efficiency of VTuber streaming has not been studied in the community, which, combined with the computation-intensive avatar generation tasks discussed in Section 2.1, leaves an important gap in deploying VTuber streaming applications on mobile devices.

3 PROPOSED APPROACH: PEV

3.1 System Overview

We develop a power efficient mobile VTuber live streaming framework, namely *PEV*, which employs edge offloading to reduce the computational load on the mobile device. Figure 2 illustrates the overall design of the proposed *PEV* framework, which consists of a *client* component and a *server* component. The *client* component is intended to be a lightweight smartphone at the streamer's end with minimum computation load for power efficiency. It captures the streamer's face with the camera, sends the camera stream to the server, and displays the server-rendered virtual character. The *server* component is composed of a *content server* and a *rendering*

server. The *content server* acts as a content and communication hub to connect the *client* and the *rendering server*, as well as distribute the rendered VTuber live stream to the remote viewers. The *rendering server* is in charge of the computation-intensive tasks for virtual avatar generation based on the camera stream.

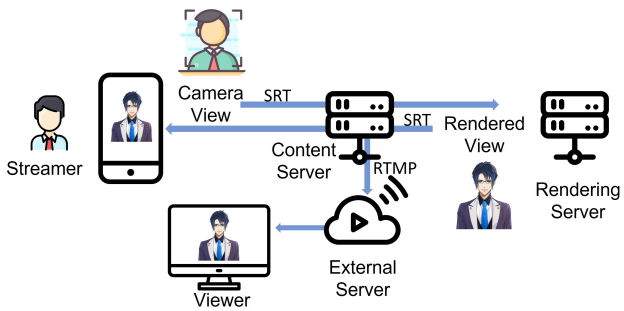

Figure 2: Workflow of the proposed *PEV* System.

As shown in Figure 2, the overall workflow of the *PEV* system is as follows. The smartphone captures the streamer's live video using the camera and delivers the camera stream to the content server using a real time communication protocol (e.g., SRT [11]). Upon receiving the camera stream, the content server relays it to the rendering server in real time for further processing. The rendering server conducts the avatar generation operations based on the incoming camera view, including face detection, facial feature extraction, and rendering. Once rendered, the rendering server returns the virtual avatar to the content server, which then streams it back to the smartphone of the streamer through the established real time communication channel, as well as broadcasts the video to the remote viewers in real time (e.g., using RTMP [1]). Following this workflow, the streamer is able to have the virtual avatar frames generated and displayed on the smartphone in real time during the VTuber streaming without requiring power-consuming computation on the mobile device. Also, the remote viewers can view the VTuber stream and interact with the streamer in real time.

3.2 Adaptive Camera Framerate Control

A key challenge in the proposed offloading-based solution is that the client must deliver the camera stream to the server, which consumes a significantly higher amount of network bandwidth than the original mobile-only VTuber streaming. While the rendering server requires a high-quality camera stream to accomplish the avatar generation computations accurately, we observe that there are plenty of inter-frame redundancies in the camera view, as the motion of the streamer is typically limited in most VTuber streaming scenarios. Following this thought, we design a client-server collaborative framerate control mechanism to determine and configure the optimal framerate for the camera stream, which reduces bandwidth consumption between the client and server while maintaining an acceptable viewing quality for the generated avatar. It is worth noting that determining the optimal framerate by itself may involve power consuming computations, which we deploy on the server to avoid compromising the power savings at the client.

As the client delivers the camera stream to the rendering server, the server maintains a moving window of frames (e.g., a 1-second segment) for analysis and prediction of the target framerate for the next segment and inform the client accordingly. To determine the target framerate, we measure the temporal information (TI) metric [22] of the frames in the moving window, which represents the average inter-frame difference between consecutive frames, as described in Equation (1), where $F_n(i, j)$ is the pixel at location (i, j) in frame n.

$$TI = max_{time}\{std_{space}[F_n(i,j) - F_{n-1}(i,j)]\} \tag{1}$$

Generally speaking, a lower TI value indicates higher similarity between consecutive frames, and thus a higher chance of skipping one of the frames to save bandwidth without significantly impacting the rendering and viewing quality. Following this thought, we define a TI threshold, namely TI_{th}, to represent the upper bound of TI value, below which the frame can be skipped to reduce the framerate. Considering a moving window of N frames and the original framerate is f, the target framerate can be measured by the number of frames in the past N frames where TI is no less than T_{th}:

$$FR = \frac{f}{N} \sum_{i=n-N+1}^{N} \begin{cases} 1 & if\ TI_i \geq TI_{th} \\ 0 & else \end{cases} \tag{2}$$

To calculate the TI threshold, we apply a pre-set percentage on the range of TI measurements in the moving window, which is configurable to achieve various levels of framerate control:

$$\begin{aligned} TI_{th} &= min(F_{n-N+1}, \cdots, F_n) \\ &+ (max(F_{n-N+1}, \cdots, F_n) - min(F_{n-N+1}, \cdots, F_n)) * a \end{aligned} \tag{3}$$

where a is the pre-set percentage, $min(F_{n-N+1}, \cdots, F_n)$ and $max(F_{n-N+1}, \cdots, F_n)$ are the minimal and maximal TI measurements in the past N frames, respectively. Algorithm 1 presents the detailed procedure for calculating the target framerate based on the moving window of frames and the TI threshold.

Algorithm 1 Calculation of target framerate.

```
Input: F_n, the frame received from the client
Output: FR, the target framerate
N = 30                                  ▷ Size of moving window
buf = FIFO_Queue(N)                     ▷ Moving window
while not end of stream do
    TI_n = TI(F_n, F_{n-1})
    buf.remove(TI_{n-N})
    buf.add(TI_n)
    TI_th = min(buf) + (max(buf) − min(buf)) * a
    FR = Σ_{i=n-N+1}^{N} TI_i | TI_i ≥ TI_th
end while
```

4 SYSTEM IMPLEMENTATION

Figure 3 shows the system implementation of the proposed *PEV* system, including the detailed technical component of the *mobile device*, the *content server*, and the *rendering server*.

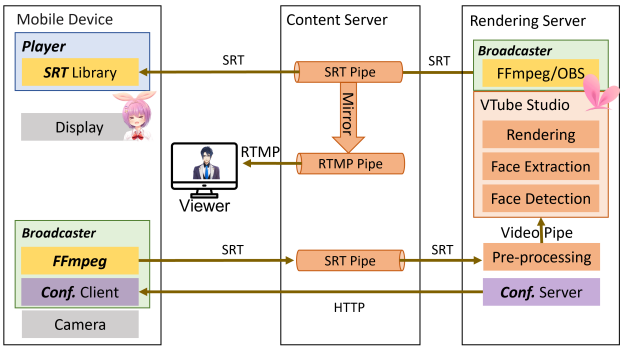

Figure 3: System architecture of the proposed *PEV* system.

4.1 Mobile Device

4.1.1 Broadcaster. The broadcaster is responsible for capturing and transmitting the camera view to the rendering server. To accomplish this task, we utilize *FFmpeg* as the low-level library to send the camera view to the rendering server via the SRT protocol [11]. The camera view is encoded in H.264 without B frames to minimize delay. Following the typical workflow of using a webcam, the camera view is transmitted at a resolution of 720p and a framerate of 30 fps. Also, we deploy a framerate configuration unit in the broadcaster, which dynamically adjusts the framerate of the camera stream to the target framerate recommended by the server.

4.1.2 Player. The player is responsible for managing and displaying the rendered view on the streamer's mobile device. We adopt the lightweight *Larix Player* [7] for this purpose, as it enables seamless playback of SRT streaming on a mobile device. Once the rendered view is received from the content server, the player displays it immediately without additional buffering that could introduce delays. This ensures a smooth and uninterrupted viewing experience.

4.2 Rendering Server

4.2.1 Pre-processing. The pre-processing unit serves as the first point of contact for the received camera stream, which identifies the locations of multiple faces by utilizing the OpenSeeFace library [10]. It then selects one face at a time to transmit to the avatar rendering software via a virtual camera created by v4l2loopback [14].

4.2.2 Renderer Software. The renderer software plays a crucial role in rendering the virtual avatar based on the streamer's face transmitted by the pre-processing unit. We adopt the commercial virtual character renderer *VTube Studio* [15] to accomplish this task. However, it is worth noting that other renderer software capable of retrieving the streamer's view from the webcam device can also be used. The renderer software executes the face detection, facial feature extraction, and rendering steps described in Section 2.1.

4.2.3 Server Broadcaster. We employ *Open Broadcaster Software* (OBS) [9] as the server side broadcaster, which supports multiple codecs and protocols. It streams the generated virtual avatar to the content server through the SRT protocol by recording the renderer window displaying the virtual character. This approach enables the compatibility with other renderer software, while still maintaining seamless integration with the rest of the system. Additionally, by using a webcam as input and capturing the window as output, we are able to leverage the capabilities of OBS while keeping the system lightweight and flexible.

4.2.4 Server Configuration Unit. The server configuration unit follows the framerate control algorithm described in Section 3.2 to calculate the target framerate based on the moving window of existing frames received from the client. The determined target framerate is returned to the client configuration unit (as part of the client broadcaster) to configure the framerate.

4.3 Content Server

The content server plays a crucial role in managing SRT sessions between the mobile device and the rendering server, facilitating connection requests and coordinating data transfer responsibilities. This is made possible through the use of the *Simple Realtime Server* (SRS) [12] that supports a variety of streaming protocols. With SRS, the content server also transcodes the streaming output from the rendering server to other popular protocols like RTMP, enabling seamless distribution of the video stream to the remote viewers.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

5.1.1 Hardware Setup. We implemented the *PEV* system as illustrated in Figure 3. We use a Pixel 3 smartphone as the mobile device, which is equipped with a 1.6 & 2.5 GHz processor, 4 GB of RAM, and a 2915 mAh battery. The rendering server is deployed on a gaming laptop with an Intel i7-11800H processor and NVIDIA GeForce RTX 3070 GPU. The content server is hosted on a virtual machine with 8 cores of Intel Core i9-10980XE CPU at 3.00GHz and 16 GB memory assigned running on a workstation serving for Kubernetes. The input videos are captured from the front camera of the smartphone at 720p resolution. For each test case, the input video is captured live with similar background, motion, and duration (~120 seconds).

5.1.2 Test Cases. We evaluate and compare *PEV* with baseline systems based on VTube Studio [15].

- ***VTS-Local*** deploys and performs all the VTuber streaming on the mobile device, which is implemented using VTube Studio without the PC streaming feature.
- ***VTS-Stream*** conducts face detection and facial feature extraction on the mobile device and renders the virtual character on the rendering server, which is implemented by enabling the streaming mode of VTube Studio. Since the rendered avatar is only presented on the rendering server, it requires both the mobile device and the rendering server in proximity of the streamer.
- ***VTS-Stream-Play*** complements *VTS-Stream* with the playback feature that streams and presents the rendered avatar view to the streamer in real time.
- ***PEV*** deploys our proposed offloading-based approach without framerate control, which has bi-directional video streams between the mobile device and the server for offloading the virtual avatar rendering task.
- ***PEV-FC*** deploys the framerate control mechanism to *PEV* to reduce the bandwidth consumption caused by offloading.

Table 1: Comparisons of functionality, power consumption, latency, and bandwidth between *PEV* and the baseline systems.

		VTS-Local	VTS-Stream	VTS-Stream-Play	*PEV*	*PEV-FC (a=0.7)*
Functionality	Outdoor Streaming	N/A	✓	✓	✓	✓
	Presenter Feedback View	N/A	N/A	✓	✓	✓
	Multi-Face	N/A	N/A	N/A	✓	✓
Measurement	Avg. Power (W)	4.18	3.92	4.52	3.53	3.55
	Avg. Latency (s)	0.56±0.06	0.61±0.03	1.06±0.06	1.57±0.06	1.62±0.08
	Bandwidth (Mbps)	0	0.50	2.80	4.97	3.34

Table 1 compares the functionalities of the 5 systems for outdoor streaming, presenter feedback view, and multi-face features. *PEV* and *PEV-FC* support all the features given their offloading design.

5.2 Power Evaluation

We use the Monsoon Power Monitor [6] to measure the power consumption of the mobile device and conduct post-processing to remove outliers in the power traces that are off by over 3x standard deviations from the mean value, as shown in Table 1. Our proposed *PEV* and *PEV-FC* systems can save 0.99 W and 0.97 W of power compared to the functionally equivalent baseline *VTS-Stream-Play*. Furthermore, Figure 4 presents the boxplot of power distribution based on raw power traces without post-processing over a 120-second period.

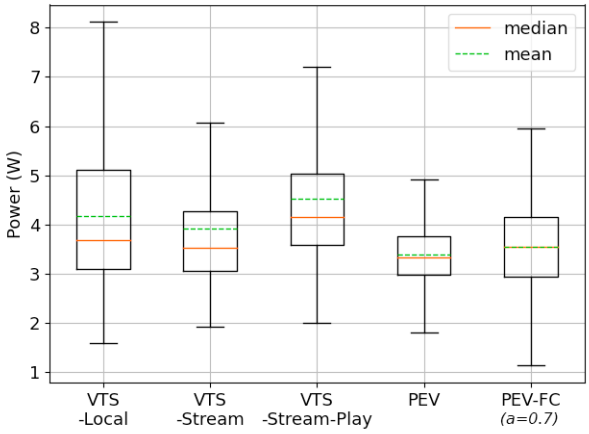

Figure 4: Distribution of power consumption.

Furthermore, Table 2 shows the power results of *PEV* when processing multiple faces. Since VTube Studio does not support multi-face rendering, none of the aforementioned baseline systems can accomplish this task; therefore, we compare with the OpenSeeFace library [10], where we conduct face detection on 10% of the input frames and feature extraction on every frame, considering that the face position does not change significantly over a short period of time. We observe that the power consumption of *PEV* does not depend on the number of faces by offloading the computation workload to the server. The OpenSeeFace library, on the other hand, shows a correlation between the number of faces and power consumption, since each face results in power-consuming computations on the mobile device. The results show that the power consumption increased from 6.63 W for 1 face to 7.03 W for 2 faces. The power drop (6.27 W) in the 3-face case is caused by the decrease in framerate due to the intensive computation. Under the varying framerates, the per-frame power consumption would be a more accurate power efficiency metric, which increases significantly with OpenSeeFace (0.23-0.34 W) and stays constant with *PEV* (0.12 W).

Table 2: Average power consumption and framerate evaluations with multiple faces using OpenSeeFace (*OSF*) and *PEV*.

Test Case	Avg. Power (W)		Framerate (FPS)		Power per Frame (W)	
	OSF	*PEV*	OSF	*PEV*	OSF	*PEV*
1 Face	6.63	3.53	28.34	30	0.23	0.12
2 Faces	7.03	3.57	21.97	30	0.32	0.12
3 Faces	6.27	3.54	18.30	30	0.34	0.12

5.3 Overhead Evaluation

5.3.1 Latency. We measure the latency between significant facial or body movements (e.g., eye opening) of the streamer and the corresponding virtual character movements using a timer and taking the average of 10 measurements, as shown in Table 1. The value with ± after the average latency is the standard deviation of the measurements to represent the uncertainty caused by manual operations of starting/stopping timers. *VTS-Local* has the lowest latency at 0.56 seconds, followed by *VTS-Stream* at 0.61 seconds; in both cases, the camera view is processed locally on-device. *VTS-Stream-Play* has a higher latency of 1.06 seconds, as it offloads the facial features to the server for processing and streams the rendered avatar view back to the client. *PEV* and *PEV-FC* have 1.57 and 1.62 seconds of latency, respectively. In comparison to *VTS-Stream-Play*, all the 3 cases involve bi-directional traffic between the mobile device and the rendering server to support the presenter view feature; however, *PEV* and *PEV-FC* offload the camera stream to the rendering server, which requires higher latency than just offloading the facial features. The latency can be improved by optimizing the communication protocol [13], which is out of scope for this paper.

5.3.2 Bandwidth. To evaluate the bandwidth consumption, we employ Android bug reports [5], which provide a comprehensive snapshot of the system state for each application. As shown in

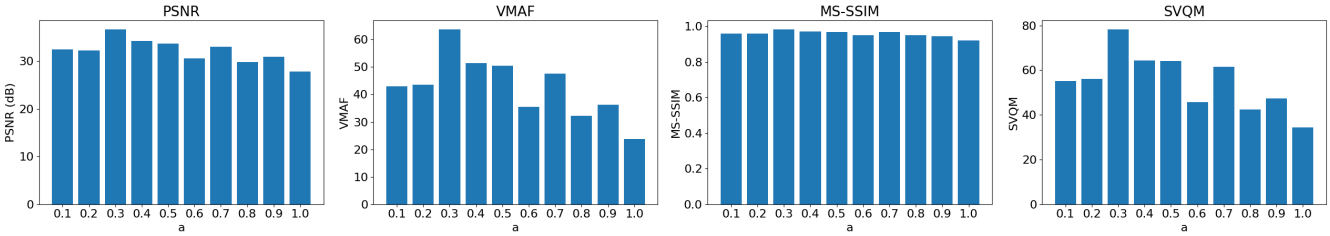

Figure 5: Quality evaluations of *PEV-FC* under different configurations of a in Algorithm 1.

(a) $a = 0.0$ **(b)** $a = 0.7$ **(c) Diff. (a) & (b)**

Figure 6: Rendered views with and without framerate control.

Table 1, *VTS-Local* consumes no bandwidth because all the computations are conducted locally on-device. *VTS-Stream* requires only 0.50 Mbps to send face features extracted on-device to the rendering server. The facial features are transmitted in plaintext JSON format using the TCP protocol, resulting in reduced bandwidth usage at the expense of increased power consumption on the mobile device. In addition, *VTS-Stream* does not support playback of the rendered avatar view on the mobile devices and thus saves the bandwidth for streaming back the rendered view. *VTS-Stream-Play* shares the similar streaming functionality and workflow with *PEV* and *PEV-FC*, and its bandwidth consumption hits 2.80 Mbps. *PEV* requires 4.97 Mbps for the bi-directional video streaming (i.e., offloading and playback) using the SRT protocol. *PEV-FC* (a=0.7) achieves significantly lower bandwidth consumption (3.34 W) compared to *PEV* thanks to the framerate control mechanism.

To further evaluate the framerate reduction mechanism adopted in *PEV-FC*, we set the parameter a in Algorithm 1 to different ratios and measure the framerate, power consumption, and the resulting latency and bandwidth overhead. In this experiment, we use a 120-second pre-recorded video with a bitrate of 2.11 Mbps to measure the framerate and bandwidth consumption, and the live videos to measure power and latency. Table 3 shows the evaluation results, where *PEV-FC* keeps the trend that a higher a value (i.e., lower framerate) achieves reduced bandwidth with a slight increase in latency. Also, the power consumption is independent of the a values.

5.3.3 Quality. To evaluate the rendering quality of the proposed *PEV* system, we use the pre-recorded video in the bandwidth evaluation to feed the renderer and calculate four quality metrics, including PSNR, VMAF [8], MS-SSIM [31], and SVQM [30], with the 30 FPS video (without framerate reduction) as the reference. Figure 5 shows the quality results under various a values for framerate reduction, and Figure 6 shows the visual demonstration of rendered avatar view for Frame #170 with (a=0.7) and without (a=0) framerate control, together with the delta between the two views. The quantitative and visual results indicate acceptable quality impact posed by the framerate reduction mechanism.

Table 3: Framerate, power consumption, latency and bandwidth evaluations of *PEV-FC* under different configurations of a in Algorithm 1.

a	Avg. Framerate (FPS)	Avg. Power (W)	Avg. Latency (s)	Bandwidth (Mbps)
0.0	30.0	3.53	1.57±0.06	4.97
0.1	26.4	3.55	1.58±0.05	4.62
0.2	23.4	3.55	1.59±0.05	4.37
0.3	20.5	3.55	1.57±0.05	4.14
0.4	18.2	3.55	1.57±0.06	3.97
0.5	15.6	3.55	1.64±0.06	3.73
0.6	13.2	3.55	1.64±0.06	3.56
0.7	10.4	3.55	1.62±0.08	3.34
0.8	8.8	3.55	1.67±0.11	3.17
0.9	6.2	3.55	1.80±0.12	2.93
1.0	3.6	3.55	1.88±0.27	2.77

6 CONCLUSION

We have developed a power efficient VTuber live streaming system, namely *PEV*, to address the power consumption challenge in the emerging streaming application. *PEV* achieves significant power savings by offloading computation-intensive face detection, facial feature extraction, and rendering operations to the server, while dynamically adjusting the framerate of the offloaded camera stream to control the bandwidth overhead. We evaluated the power, latency, bandwidth, and quality of *PEV* on an end-to-end VTuber live streaming system, in comparison with baseline VTuber streaming systems. Our evaluation demonstrates significant power savings and limited overhead achieved by *PEV*. The repository of the project is at https://github.com/hwsel/PEV.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under award 1910085 and by the gift donation from Adobe Inc.

REFERENCES

[1] 2012. *Adobe's Real Time Messaging Protocol.* https://rtmp.veriskope.com/pdf/rtmp_specification_1.0.pdf.

[2] 2020. *Source information: Innovation in Adverse Times.* https://www.youtube.com/trends/articles/report-sources-tr20/.

[3] 2022. *VTubers are making millions on YouTube and Twitch.* https://techcrunch.com/2022/08/20/vtubers-are-making-millions-on-youtube-and-twitch/.

[4] 2022. *What is VTubing and why do brands need to know?* https://www.thedrum.com/news/2022/08/26/what-vtubing-and-why-do-brands-need-know.

[5] 2023. *Capture and read bug reports.* https://developer.android.com/studio/debug/bug-report.

[6] 2023. *High Voltage Power Monitor.* https://www.msoon.com/high-voltage-power-monitor.

[7] 2023. *Larix Player for Android.* https://softvelum.com/player/android/.

[8] 2023. *Netflix/Vmaf: Perceptual Video Quality Assessment Based on Multi-Method Fusion.* https://github.com/Netflix/vmaf.

[9] 2023. *Open Broadcaster Software (OBS).* https://obsproject.com/.

[10] 2023. *OpenSeeFace.* https://github.com/emilianavt/OpenSeeFace.

[11] 2023. *Secure Reliable Transport (SRT) Protocol.* https://github.com/Haivision/srt.

[12] 2023. *Simple Realtime Server (SRS).* https://ossrs.io/lts/en-us.

[13] 2023. *SRT: Support Publish by SRT for Live Streaming, New Generation Protocol for Broadcasting.* https://github.com/ossrs/srs/issues/1147.

[14] 2023. *v4l2loopback - a kernel module to create V4L2 loopback devices.* https://github.com/umlaeute/v4l2loopback.

[15] 2023. *VTube Studio.* https://denchisoft.com/.

[16] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V Krishnamurthy, and Amit K Roy-Chowdhury. 2019. Frugal Following: Power Thrifty Object Detection and Tracking for Mobile Augmented Reality. In *Internatoinal Conference on Embedded Networked Sensor Systems (SenSys).* 96–109.

[17] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *USENIX Annual Technical Conference (ATC).* 21–21.

[18] Patrick Gwillim-Thomas. 2023. The Actualizing Platform: 2.5 Dimensions in the VTuber Media Ecology. *Mechademia* 15, 2 (2023), 49–69.

[19] Ronny Hans, Ulrich Lampe, Daniel Burgstahler, Martin Hellwig, and Ralf Steinmetz. 2014. Where Did My Battery Go? Quantifying the Energy Consumption of Cloud Gaming. In *IEEE International Conference on Mobile Services (MS).* 63–67.

[20] Li Hu, Bang Zhang, Peng Zhang, Jinwei Qi, Jian Cao, Daiheng Gao, Haiming Zhao, Xiaoduan Feng, Qi Wang, Lian Zhuo, Pan Pan, and Yinghui Xu. 2021. A Virtual Character Generation and Animation System for E-commerce Live Streaming. In *ACM International Conference on Multimedia (MM).* 1202–1211.

[21] Wenjie Hu and Guohong Cao. 2015. Energy-Aware Video Streaming on Smartphones. In *IEEE Conference on Computer Communications (INFOCOM).* 1185–1193.

[22] International Telecommunication Union (ITU). 2019. BT.500: Methodologies for the Subjective Assessment of the Quality of Television Images. https://www.itu.int/rec/R-REC-BT.500

[23] Nan Jiang, Yao Liu, Tian Guo, Wenyao Xu, Viswanathan Swaminathan, Lisong Xu, and Sheng Wei. 2020. QuRate: Power-Efficient Mobile Immersive Video Streaming. In *ACM Multimedia Systems Conference (MMSys).* 99–111.

[24] Nan Jiang, Viswanathan Swaminathan, and Sheng Wei. 2017. Power Evaluation of 360 VR Video Streaming on Head Mounted Display Devices. In *Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV).* 55–60.

[25] Youna Kang. 2021. CodeMiko: An Interactive VTuber Experience. In *SIGGRAPH Asia 2021 Real-Time Live!* 1–1.

[26] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2019. Energy-efficient Video Processing for Virtual Reality. In *International Symposium on Computer Architecture (ISCA).* 91–103.

[27] Robert LiKamWa, Zhen Wang, Aaron Carroll, Felix Xiaozhu Lin, and Lin Zhong. 2014. Draining Our Glass: An Energy and Heat Characterization of Google Glass. In *Asia-Pacific Workshop on Systems (APSys).* 10:1–10:7.

[28] Yao Liu, Mengbai Xiao, Ming Zhang, Xin Li, Mian Dong, Zhan Ma, Zhenhua Li, and Songqing Chen. 2015. Content-Adaptive Display Power Saving in Internet Mobile Streaming. In *ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV).* 1–6.

[29] Zhicong Lu, Chenxinran Shen, Jiannan Li, Hong Shen, and Daniel Wigdor. 2021. More Kawaii than a Real-Person Live Streamer: Understanding How the Otaku Community Engages with and Perceives Virtual YouTubers. In *CHI Conference on Human Factors in Computing Systems (CHI).* 1–14.

[30] Yang Peng and Eckehard Steinbach. 2011. A Novel Full-Reference Video Quality Metric and Its Application to Wireless Video Transmission. In *IEEE International Conference on Image Processing (ICIP).* 2517–2520.

[31] Z. Wang, E.P. Simoncelli, and A.C. Bovik. 2003. Multiscale Structural Similarity for Image Quality Assessment. In *Asilomar Conference on Signals, Systems & Computers, 2003.* 1398–1402.

[32] Zhisheng Yan, Chen Song, Feng Lin, and Wenyao Xu. 2018. Exploring Eye Adaptation in Head-Mounted Display for Energy Efficient Smartphone Virtual Reality. In *International Workshop on Mobile Computing Systems & Applications (HotMobile).* 13–18.

[33] Jingyu Zhang, Gan Fang, Chunyi Peng, Minyi Guo, Sheng Wei, and Viswanathan Swaminathan. 2016. Profiling Energy Consumption of DASH Video Streaming Over 4G LTE Networks. In *International Workshop on Mobile Video (MoVid).* 3.

[34] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. 2010. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES-ISSS).* 105–114.

[35] Zichen Zhu, Xianglong Feng, Zhongze Tang, Nan Jiang, Tian Guo, Lisong Xu, and Sheng Wei. 2022. Power-Efficient Live Virtual Reality Streaming Using Edge Offloading. In *Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV).* 57–63.