Foveated Streaming of Virtual Reality Videos

Miguel Fabián Romero Rondón, Lucile Sassatelli, Frédéric Precioso and Ramon Aparicio-Pardo
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
{romero,sassatelli,precioso,raparicio}@i3s.unice.fr

ABSTRACT

While Virtual Reality (VR) represents a revolution in the user experience, current VR systems are flawed on different aspects. The difficulty to focus naturally in current headsets incurs visual discomfort and cognitive overload, while high-end headsets require tethered powerful hardware for scene synthesis. One of the major solutions envisioned to address these problems is foveated rendering. We consider the problem of streaming stored 360° videos to a VR headset equipped with eye-tracking and foveated rendering capabilities. Our end research goal is to make high-performing foveated streaming systems allowing the playback buffer to build up to absorb the network variations, which is permitted in none of the current proposals. We present our foveated streaming prototype based on the FOVE, one of the first commercially available headsets with an integrated eye-tracker. We build on the FOVE's Unity API to design a gaze-adaptive streaming system using one low- and one high-resolution segment from which the foveal region is cropped with per-frame filters. The low- and high-resolution frames are then merged at the client to approach the natural focusing process.

CCS CONCEPTS

• **Information systems → Multimedia streaming**; • **Human-centered computing → Virtual reality**;

KEYWORDS

Virtual Reality, Foveated Rendering, Streaming

1 INTRODUCTION

Virtual Reality (VR) has taken off in the last years thanks to the democratization of affordable head-mounted displays (HMDs). A main challenge to the massive adoption of VR is the delivery through streaming over the Internet. Several works have proposed to provision better qualities in the restricted area the user can watch from the sphere, called the "viewport", lowering the quality of areas outside the viewport [2]. Current VR systems are however flawed on different aspects. First, it is hard for the Human Visual System (HVS) to focus naturally in current headsets, in particular owing to the vergence-accommodation conflict [1], incurring visual discomfort and cognitive overload. One of the major solutions envisioned to address this problem is foveated rendering. It exploits the radially-decreasing human visual acuity between the fovea and the eye's periphery [5]. Instead of adapting to the wider viewport, foveated rendering adapts to the narrower user's gaze position by blurring away the regions not in the gaze's target so as to reproduce and help the natural focusing process. Second, high-end HMDs (e.g., HTC Vive and Oculus Rift) currently require powerful hardware tethered to the headset for scene synthesis. These hardware requirements can be reduced by employing foveated rendering [4].

There was no affordable (less than $1000) VR foveated rendering systems until recently (second-half of 2017), mainly because of the high costs of integrating an eye-tracker into the VR hardware. This has changed with the release of the FOVE[1]. A number of works have looked at foveated streaming to couple the visual and computational gains with bandwidth gains, yet not for in-headset VR (e.g., see [3] and references therein). We consider the problem of streaming stored 360° videos to a VR headset equipped with eye-tracking and foveated rendering capabilities (the FOVE). Our end research goal is to make high-performing foveated streaming systems allowing the playback buffer to build up to absorb the network variations, which is permitted in none of the current proposals. To do so, we intend to build on proper gaze prediction and foveal manipulation to anticipate and drive the user's gaze.

Towards this goal, and as a first step, we present our foveated streaming prototype, based on the FOVE HMD. Our contributions are:

• We build on the FOVE's Unity API to design a gaze-adaptive streaming system.
• The client is designed to inform the server of the current gaze position, receives the video sphere in low-resolution and additionally the foveal region in high-resolution, and is responsible for the merging of textures.
• The server prepares the content upon reception of a request. It computes the equirectangularly projected mask, crops the frame of the segment and formats the resulting piece for transmission without overhead.
• To enable full freedom in future design, we provide the ability to apply different masks over each frame of a segment, and verify that the whole system can work online.

Our work is mainly related to [3] and [6] addressing foveated streaming for mobile cloud gaming (not VR) and in-headset VR without foveation, respectively. In [3], Illahi et al. live-encode frames by setting the quantization parameter of each macroblock depending on the gaze location, consider a Gaussian-smoothed circular foveal region and assert the processing latency to be order of 20 ms. To prevent changing the encoder and keep the complexity very low for the larger frames to be processed in 360°, we use the MPEG-DASH principle and make the video available in 2 resolutions, cropping the

MMSys'18, June 12–15, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5192-8/18/06.

[1] http://getfove.com/

foveal region from the high resolution for streaming. This thresholding is chosen for simplicity in our prototype owing to the non-circularity of the foveal region in the equirectangular projection. Generating one second-long segment made of 30 frames requires about 700 ms in our CPU implementation (FFmpeg cannot access the GPU through Virtualbox), amounting to about 23 ms per frame, which the same order as in [3]. In [6] (akin to [7]), VR videos are streamed with the viewport selected from high-resolution tiles, and the client reassembles the different regions at the destination. We leverage the same idea (with the same high-speed H.264 encoding flags) but develop the whole system for Unity and the FOVE, and specifically design the cropping filters to allow dynamic foveation over one-second segments to preserve the feeling of immersion.

2 BUILDING BLOCKS

We first introduce our working material: the FOVE headset and its Unity API with the components employed, then we define the eye-tracking data and how they are used with our FFmpeg-based cropping module.

2.1 FOVE headset and Unity API

FOVE is a VR headset including an eye-tracker that allows to follow the user's gaze. It provides two programming interfaces, one is the Unity Plugin and the other is the C++ SDK. These APIs allow, among other tasks, to connect to the FOVE compositor, to capture the HMD orientation, to get the direction where the user is looking at, and to know if the user is blinking. We decided to work with the Unity API, because the Unity engine is a widely used industry standard that offers different resources built from a large community.

Unity[2] is a game engine used to develop three-dimensional simulations across several platforms. The basic components of a Unity simulation are a scene, where the entire virtual world is designed, and a camera that captures and displays the virtual world to the viewer. To give the stereoscopic effect, our Unity scene contains two cameras, mapped to the movements of the user's head through the FOVE SDK. To give the immersion illusion, the cameras are fixed in the center of a sphere, where the 360° video is projected.

VR videos, which are spherical in nature, are mapped onto a planar texture, one of these mappings is the commonly used equirectangular projection. With this panoramic projection we can consider that the 360° video has the same rectangular shape as a regular video. In Unity, a video can be considered as a 2D-texture that changes in time, to play the 360° video onto a texture, we used the `VideoPlayer` component, since it supports the `H.264` video codec when playing a video from a URL. The VideoPlayer component can be tuned to playback videos streamed from a server, using the following event-handlers:

- `prepareCompleted`. Invoked when the VideoPlayer has downloaded some frames, and reserved resources so that the video can be played.
- `loopPointReached`. Invoked after the VideoPlayer has finished the playback of the video.

To provide adaptive streaming capabilities, the video needs to be segmented in time: we chop the high-resolution video into segments with constant duration of 1 second. The `VideoPlayer` component can be tuned to request each segment of the video from the server by manipulating the requested URL, simply adding the id of the segment, and in our case, the user's gaze parameters.

[2]https://unity3d.com/

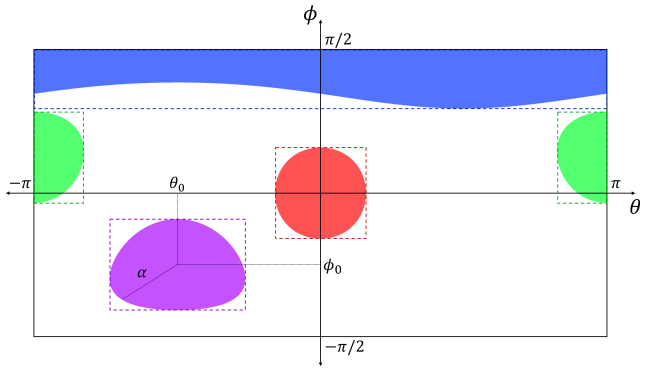

Figure 1: Deformation of the circular foveal area (and its respective bounding box) when the video-sphere is mapped to a plane using the equirectangular projection.

2.2 User's gaze parameters and foveal cropping

The fovea of the viewer is modeled as a circle in the spherical coordinates system by the parameters described in Table 1. We can communicate the current user's gaze position to the server by sending the tuple (θ, ϕ), and set the size of the fovea with the parameter α, thereby controlling the 'feeling' of natural vision of the user and the bandwidth needed.

Table 1: Parameters of the user's gaze in the spherical coordinate system

Param.	Range	Description
r	$\in \mathbb{R}_+$	Radius of the sphere.
θ	$\in [-\pi, \pi]$	Inclination, also known as yaw axis.
ϕ	$\in [-\frac{\pi}{2}, \frac{\pi}{2}]$	Azimuth, also known as pitch axis.
α	$\in \mathbb{R}_+$	Controls the radius of the foveal area.

Since the video is projected with the equirectangular projection, even though the fovea has a circular shape, it gets deformed depending on its location, and the size of the rectangular bounding box enclosing it varies as shown in Figure 1.

Each segment request includes the timestamp t, the spherical angles (θ_t, ϕ_t) and the size of the fovea α_t. With this information, the server can crop each of the high-resolution segments to fit only the bounding box of the foveal area. As segments are 1 second-long, we need to crop and prepare the frames of each segment as low under a second as possible, for this purpose we used `FFmpeg`[3].

3 DESIGN AND IMPLEMENTATION OF THE FOVEATED STREAMING SYSTEM

In this section we present how all the building blocks are composed to make the proposed foveated streaming system. The full design

[3]https://ffmpeg.org/

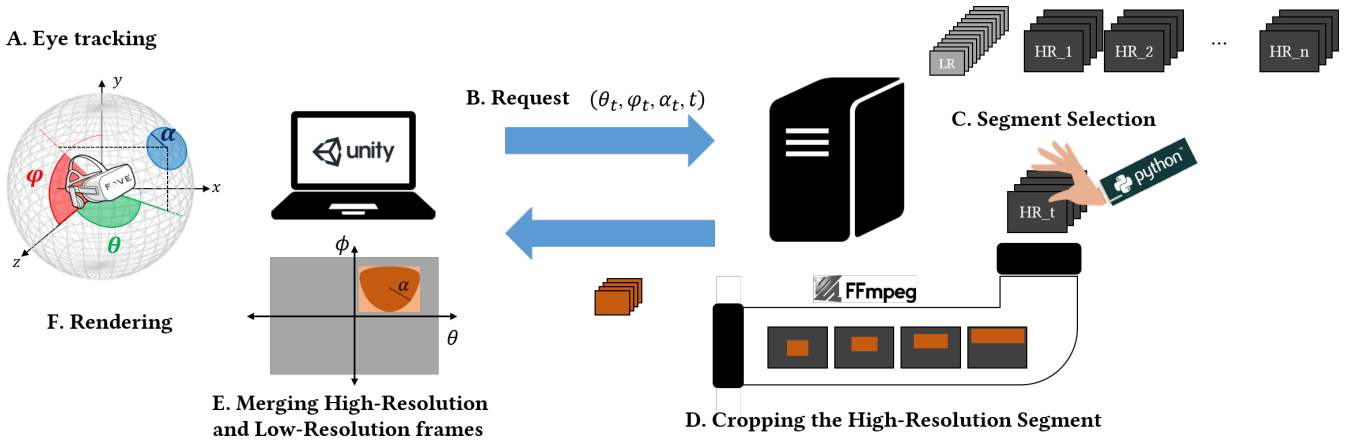

Figure 2: Workflow of the Foveated Streaming System. The steps are in the following order: A. Determine the gaze position with the FOVE headset. B. Request a new segment with the parameters of the gaze (Section 3.1). C. Select the segment according to the request. D. Crop the high-resolution segment (Section 3.2). E. Merge the high- and low-resolution frames (Section 3.3). F. Render the resulting frame.

of the system is shown in Figure 2. The client runs under Windows 10, where the Unity engine is in charge of the following tasks:

- Read user's gaze position from the FOVE HMD.
- Control the buffer to know when to request a new segment.
- Receive, decode and merge the high- and low-resolution frames.
- Project the result in the sphere.
- Use the Unity cameras to capture the viewport of the user in the sphere and render it in the HMD.

The server side is simulated using a Virtual Machine with Ubuntu 16.04 where the videos are stored in two resolutions: low-resolution (1024x512) and high-resolution (4096x2048). A regular system would stream both the high-resolution and low-resolution segments over time, but for the sake of simplicity, we have chosen to have the client fetching the complete low-resolution video at the beginning, and then stream only the cropped high-resolution segments.

3.1 Unity VideoPlayer Module for Streaming

Using Unity in the client side, we assign one distinct VideoPlayer to each requested segment to be able to load, prepare and play the segments independently. As described in Algorithm 1, the data structure that holds the VideoPlayers also acts as the buffer of the system. The event-handler `Prepared` is used to start the playback of the video and to play possible subsequent unprepared segments, it also synchronizes the high-resolution and low-resolution frames. Once the playback of the current segment is finished, the event-handler `EndReached` allows to play the next segment if it is already prepared in the buffer, otherwise it pauses the playback of the low-resolution video, that would be played again in the `Prepared` event-handler. The function `RequestNextSegments()` requests the next video segments, passing the parameters $(\theta_t, \phi_t, \alpha_t, t)$ to the request url, starting from segment t until filling the buffer.

3.2 Cropping the High-Resolution Segment

The server side is implemented in `Python`. When the user requests a high-resolution segment, a smooth transcoder using `FFmpeg` cuts on the fly the frames to contain only the fovea of the user, the bounding box of the fovea is a rectangle (x, y, w, h) with origin (top-left vertex) in the point (x, y), width w and height h. Since segment duration is 1 second-long, we need to crop and prepare each segment as low under a second as possible. Importantly, we design the FFmpeg filter not to simply crop a whole segment at once using the same mask, but instead we want to provide the ability to apply different masks over each frame of a segment. Indeed, we want to guarantee such full freedom to enable in our future work attention driving with refined foveal region manipulations. For this purpose we implemented our own custom `FFmpeg` filter called "gazecrop", and it is executed using the following command:

`ffmpeg -i input_`t`.mp4 -vf gazecrop="bbox_expr='` θ_0, ϕ_0,α_0,θ_1,ϕ_1,α_1,. . .,θ_{n-1},ϕ_{n-1},α_{n-1}`'" -vcodec 'libx264' -preset veryfast -tune zerolatency -movflags 'frag_keyframe + empty_moov' -an -f mp4 pipe:1`

In the `gazecrop` filter, the string after `bbox_expr` expresses the triplets $(\theta_i, \phi_i, \alpha_i)$ of the user's gaze for each frame i, and n is the number of frames in each video segment. This filter crops each frame i, after computing the foveal bounding box (x, y, w, h) using equations (1-4) from [6]. This command is executed in Python, the output video is piped out to a Python variable and then it can be simply written out as the response of the HTTP request.

As mentioned in Sec. 1, the total server delay is about 700 ms per segment, that is ca. 23 ms per frame (with the filters running on the CPU only, owing to the virtualized server implementation in the prototype). Pipelining the emission of frames before the completion of the segment is part of future work.

Data: *minBuffSize*, *maxBuffSize*, *serverUrl*, θ_t, ϕ_t, α_t
t = 0; currSegId = 0; currBuffSize = 0;
buffHighRes = new Array(*maxBuffSize*);
RequestNextSegments();
lowResVP = new VideoPlayer();
lowResVP.url = *serverUrl* + 'lr';

Function *RequestNextSegments()* **do**
  while *minBuffSize* < *currBuffSize* < *maxBuffSize* **do**
    i = t mod *maxBuffSize*;
    buffHighRes[i] = new VideoPlayer();
    buffHighRes[i].loopPointReached = **EndReached**;
    buffHighRes[i].prepareCompleted = **Prepared**;
    buffHighRes[i].url = *serverUrl*+($\theta_t, \phi_t, \alpha_t, t$); t = t+1;
    currBuffSize = currBuffSize + 1;
  end
end

Function *Prepared(VideoPlayer highResVP)* **do**
  if *highResVP.id == buffHighRes[currSegId].id* **then**
    highResVP.play(); lowResVP.play();
  end
end

Function *EndReached(VideoPlayer highResVP)* **do**
  currSegId = currSegId+1; currBuffSize = currBuffSize-1;
  RequestNextSegments();
  nextHighResVP = buffHighRes[t mod *maxBuffSize*];
  if *nextHighResVP.isPrepared()* **then**
    nextHighResVP.play();
  else
    lowResVP.pause();
  end
end

Algorithm 1: Basic Foveated Streaming Client

$$x = \begin{cases} \theta_i - \cos^{-1}\sqrt{\frac{\cos^2\alpha_i - \sin^2\phi_i}{\cos\phi_i}} & \text{if } \cos^2\alpha_i \geq \sin^2\phi_i \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$$w = \begin{cases} 2\theta_i - \cos^{-1}\sqrt{\frac{\cos^2\alpha_i - \sin^2\phi_i}{\cos\phi_i}} & \text{if } \cos^2\alpha_i \geq \sin^2\phi_i \\ 2\pi & \text{otherwise} \end{cases} \quad (2)$$

$$y = \phi_i - \alpha_i \quad (3)$$

$$h = 2\alpha_i \quad (4)$$

3.3 Merging High-Resolution and Low-Resolution Frames

When the client receives back the response from the server with the high-resolution and low-resolution frames, before projecting it onto the sphere, it fuses both frames using a fragment shader that assigns the value of the pixel in the high-resolution texture if it belongs to the foveal area, otherwise it sets the value of the pixel in the low-resolution texture. An illustration of the result is shown in Figure 3.

Figure 3: On the left: Resulting foveated rendering effect. On the right: Comparison between total size of the frame against viewport size in red, and size of the cropped section of the foveated rendering system in blue.

4 DEMONSTRATION

For this demo we will use a virtual machine to simulate the server. The host machine is a laptop with Intel Core i7 processor, 64GB of RAM powered by a Geforce GTX 1070 GPU. The server (guest OS) contains the videos to be streamed. The users can select the video from a list containing the description, the duration and a thumbnail. The first step before starting the playback of the video is to calibrate the eye-tracker. To do this we will run the calibration process provided with the FOVE SDK that consists in following, with the gaze, a green dot in a gray background. Once the calibration is finished, the video starts, the user will interact with the HMD by moving his head and eyes to explore the 360° video.

5 CONCLUSION

In this demo, we present and run our VR foveated streaming system for the FOVE with Unity. To our best knowledge, there exists no similar system in the literature. This is the first step to make high-performing foveated streaming systems, with proper gaze prediction and foveal manipulation to anticipate and drive the user's gaze. We will run the client and the virtual server in the same laptop (Intel Core i7 processor, 64 GB of RAM powered by a Geforce GTX 1070 GPU) and welcome the MMSys conference attendees to try different videos in the FOVE headset.

ACKNOWLEDGMENTS

This work has been partly supported by the French government, through the UCA^{JEDI} Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-15-IDEX-01.

REFERENCES

[1] K. Carnegie and T. Rhee. 2015. Reducing visual discomfort with HMDs using dynamic depth of field. *IEEE Computer Graphics and Appl.* 35, 5 (2015), p. 34–41.
[2] X. Corbillon, G. Simon, A. Devlic, and J. Chakareski. 2017. Viewport-adaptive navigable 360-degree video delivery. In *IEEE ICC*.
[3] G. Illahi, M. Siekkinen, and E. Masala. 2017. Foveated video streaming for cloud gaming. In *IEEE Int. Workshop on Multimedia Signal Proc. (MMSP)*.
[4] A. Patney, J. Kim, M. Salvi, A. Kaplanyan, C. Wyman, et al. 2016. Perceptually-based foveated virtual reality. In *ACM SIGGRAPH Emerging Technologies*.
[5] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, et al. 2016. Towards foveated rendering for gaze-tracked virtual reality. *ACM Trans. on Graphics* 35, 6 (2016), p. 179.
[6] P. Rondao Alface, M. Aerts, D. Tytgat, S. Lievens, C. Stevens, et al. 2017. 16K Cinematic VR Streaming. In *ACM Multimedia Conf. (MM)*.
[7] J. Ryoo, K. Yun, D. Samaras, S. R. Das, and G. Zelinsky. 2016. Design and Evaluation of a Foveated Video Streaming Service for Commodity Client Devices. In *ACM Int. Conf. on Multimedia Sys. (MMSys)*.