UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

360-DEGREE VIDEO LIVESTREAMING OVER WIRED AND WIRELESS LINK IN VIRTUAL REALITY ENVIRONMENT

Diploma Thesis

Dallas Dimitrios

Supervisor: Korakis Athanasios

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΖΩΝΤΑΝΗ ΡΟΗ ΒΙΝΤΕΟ 360 ΜΟΙΡΩΝ ΜΕΣΩ ΕΝΣΥΡΜΑΤΟΥ ΚΑΙ ΑΣΥΡΜΑΤΟΥ ΣΥΝΔΕΣΜΟΥ ΣΕ ΠΕΡΙΒΑΛΛΟΝ ΕΙΚΟΝΙΚΗΣ ΠΡΑΓΜΑΤΙΚΟΤΗΤΑΣ

Διπλωματική Εργασία

Δάλλας Δημήτριος

Επιβλέπων: Κοράκης Αθανάσιος

Βόλος 2021

ACKNOWLEDGMENTS

Five years ago, writing my own thesis seemed impossible to me. As I am about to graduate, this belief has disappeared. Therefore I dedicate this document to my family, for always being there for me, for always supporting me both emotionally and financially so that I can walk my own path from now on.

I would like to thank deeply my supervisor Professor Korakis Athanasios, for giving me the opportunity to follow my field of interest and for his constant guidance and support on the implementation of this project.

Special thanks to all my fellow coworkers at NITlab and especially to Theodosiou Georgios for helping me surpassing obstacles and providing feedback on this project.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

ABSTRACT

This thesis intends to examine three major technologies and the various ways they can be combined to improve multimedia applications and fulfill their emerging needs in tomorrow’s society. These technologies are the spherical or 360^{o} video, multimedia livestreaming and Virtual Reality environments.

The technical background is described in detail in the beginning of this document, followed by demonstrating implementations of 360^{o} livestreaming and 360^{o} VR livestreaming. Hardware equipment used is a 360^{o} camera, Ricoh THETA V, and some virtual reality headsets, HTC Vive and VR Box. Software was selected through research regarding livestreaming protocols, namely RTMP and MJPEG, and VR functionality, namely YouTube platform, Unity game engine, WebXR Device API and A-Frame web framework. For browser approaches, a Reverse Proxy server was implemented as part of their distributed architecture.

ΠΕΡΙΛΗΨΗ

Η παρούσα πτυχιακή εργασία, η οποία έχει γραφτεί στην Αγγλική γλώσσα, σκοπεύει να εξετάσει τρεις κύριες τεχνολογίες και τους ποικίλους τρόπους που μπορούν να συνδυαστούν για να βελτιώσουν τις εφαρμογές πολυμέσων και να καλύψουν τις μελλοντικές τους ανάγκες που θα προκύψουν από την πρόοδο της κοινωνίας. Οι αναφερόμενες τεχνολογίες είναι το «σφαιρικό βίντεο» ή «βίντεο 360 μοιρών», η ζωντανή μετάδοση πολυμέσων και τα περιβάλλοντα Εικονικής Πραγματικότητας.

Το τεχνικό υπόβαθρο περιγράφεται αναλυτικά στην αρχή του συγκεκριμένου εγγράφου και ακολουθείται από την επίδειξη υλοποιήσεων ζωντανής ροής βίντεο 360^o και ζωντανής ροής βίντεο 360^o σε περιβάλλον εικονικής πραγματικότητας. Ο υλικός εξοπλισμός που χρησιμοποιήθηκε είναι μία κάμερα 360^o, η Ricoh THETA V, και κάποια γυαλιά εικονικής πραγματικότητας, το HTC Vive και το VR Box. Το λογισμικό επιλέχθηκε ερευνώντας πρωτόκολλα ζωντανής μετάδοσης πολυμέσων, αναφορικά το RTMP και το MJPEG, και την λειτουργικότητα εικονικής πραγματικότητας, αναφορικά η πλατφόρμα YouTube, η παιχνιδομηχανή Unity, η διεπαφή WebXR και η δομή A-Frame για φυλλομετρητές. Για τις προσεγγίσεις των φυλλομετρητών, υλοποιήθηκε ένας διακομιστής αντίστροφης μεσολάβησης ως κομμάτι της αρχιτεκτονικής τους.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1: A blend of advanced technology

From the beginning of time, humans always wanted to communicate with each other, as direct as possible. This need, once covered by a simple letter, has grown through the years leading to video chats over mobile phones. Living in the 21st century, human needs push technologies to advance, but also technologies inspire some of these needs.

Telecommunication and network technologies have reached levels that offer real-time communication and interaction. Audiovisual media have evolved in such ways that special multi-lens cameras, called 360^o cameras, allow spherical video capturing in combination with spatial audio. At the same time, a new type of technology emerged, that being Virtual Reality, which changes the user's optical view thus letting him into a simulated experience. Now all of the above can be combined to offer a 360^o video livestream in Virtual Reality.

1.2: Applications

The need for a service like that can be established from a variety of applications. Video chatrooms, teleconferences and virtual tours have already made their first steps. Further down the line we could see journalistic coverage, industry inspection, or even robot teleoperation for health and surgical purposes. All of the technologies composing 360^o video livestreaming in VR are considered state-of-the-art technologies and are expected to evolve tremendously in the years to come.

1.3: Motivation

The idea for this thesis' subject has occurred from a partnership work for an undergraduate lesson. The concept of the assignment was to be able to remotely navigate an Unmanned Ground Vehicle (UGV) through VR.

This, simple as it may sounds, sets some conditions that need to be followed. Livestream video should be from a 360^o camera to offer an immersive VR experience to the user. The UGV should be able to move freely everywhere, so the livestreaming should be transmitted wirelessly, either from the UGV's microcontroller or the camera device directly. And lastly, the livestream feed should be as close to real-time as it could be, because immediate teleoperation requires instantaneous reflexes to the environment changing.

All those requirements led to the “immersive wireless real-time livestreaming” direction and composed the central axis around which the research for this thesis was conducted.

CHAPTER 2: TECHNICAL BACKGROUND

2.1: Video

Video is the technology that captures moving images electronically. Those moving images are actually just a series of still images that change so fast that our eye perception makes us think that the image is moving.

Analog video represents moving visual images in the form of analog signals. It is created in a video camera by scanning an electron beam across a phosphor.

Digital video is, in simple words, moving images in rapid succession in the form of encoded digital data, and not analog information like a continuous signal. Digital video can be easily shared, copied, and stored, with the data quality not degraded. It can also be distributed with multicast in case of streaming.

2.1.1: Characteristics of video streams

- Frame rate

  It is the number of still images (or frames) recorded per unit of time of video. Usually frame rate is expressed in frames per second (fps) and ranges from six or eight fps for old mechanical cameras to 120+ fps for new professional cameras. Most video cameras record at 30 fps.

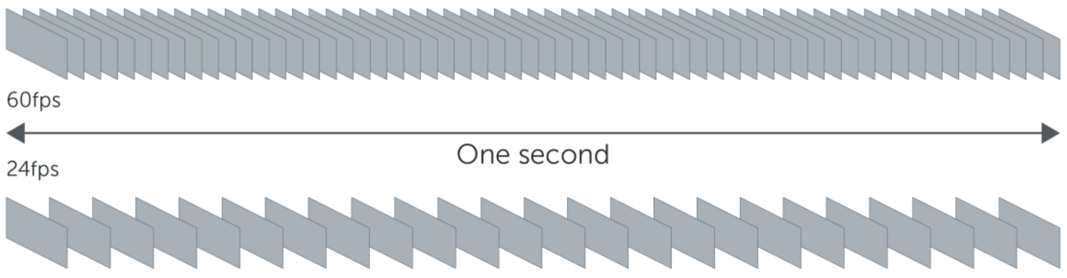


Figure 1: Visualization of frame rate with 24 and 60 frames per second

- Resolution

  Pixels are called the tiny picture elements that compose each frame when it is recorded. An image’s resolution is basically expressing the amount of pixels the image is composed of. Usually expressed by multiplication of horizontal pixels by vertical pixels:

640x480 means 640 pixels wide, by 480 pixels tall. The most popular resolutions in today's video streams are 640x320, 1024x512 and 1920x960.

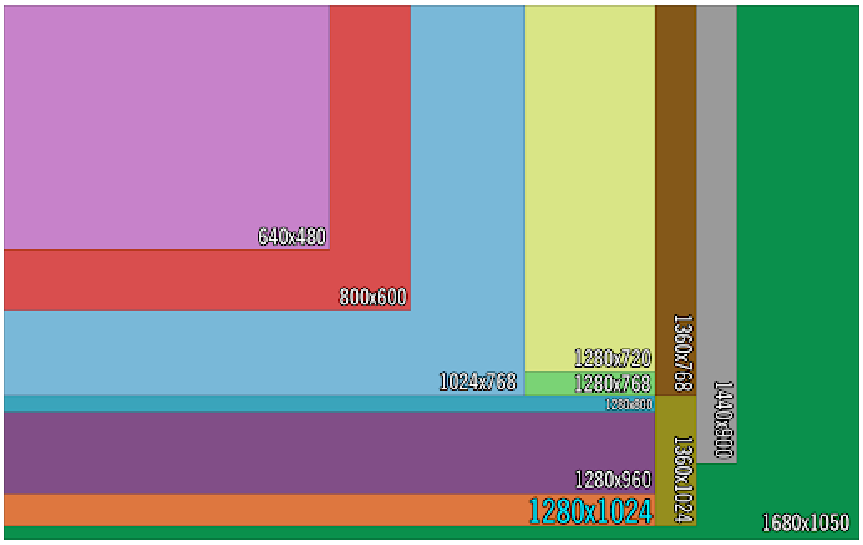


Figure 2: Most popular and common resolutions in comparison with each other

- Aspect ratio

The width and height of video screens and video elements have a proportional relationship called aspect ratio. The most common aspect ratios are rectangular, like 16:9, which is about 1.78:1.

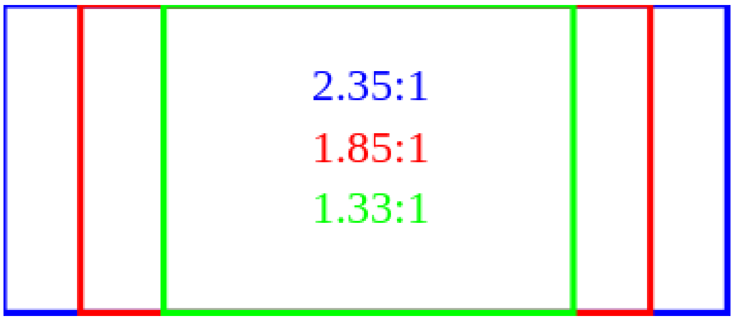


Figure 3: Various aspect ratios for different monitor types

- Bit rate

Bit rate is a measurement for the information content from a digital video stream. For uncompressed videos, bit rate corresponds to the quality of given video because bit rate is proportional to every property affecting video quality. When transmitting video, bit rate is an important factor because the transmission link must be capable of supporting it in the first place. Additionally, bit rate is important for the storage of a video because the video size is proportional to the bit rate and the duration. In order to greatly reduce the bit rate while having little effect on quality, video compression techniques are used.

2.1.2: Video Compression

Uncompressed video offers maximum quality, with the cost of a very high data rate. The most effective ways to compress video streams are reducing spatial and temporal redundancy with a group of pictures, meaning that they register the differences between the parts of single frame (spatial reduction) or between successive frames.

The most common modern compression standards are those developed by the Motion Picture Experts Group (MPEG). MPEG-1, in 1991, was designed to compress VHS-quality video and after twelve years in 2003 H.264/MPEG-4 AVC was developed, which has become the most widely used video coding standard for AVCHD, mobile phones and Internet.

- Video Coding Formats & Video Codecs

  A video coding format is the way in which digital video content is represented when stored or transmitted. It typically uses a standardized video compression algorithm.

  A specific software or hardware implementation, consisting of an encoder and a decoder, capable of compression or decompression, respectively, to/from a specific video coding format is called a video codec. Codecs are very important for streaming media. There are codecs for data (PKZIP), still images (JPEG, PNG), audio (MP3) and of course video (MPEG-4/H.264).

- Lossless & Lossy Compression

  Codecs are divided into two kinds, those being lossless and lossy. Lossless codecs, upon decompression, reproduce the same exact file as the original. Some lossless video codecs can’t compress video to data rates low enough for streaming.

  One the other hand, lossy codecs produce a facsimile of the original file upon decompression, but not the original file. This kind of codecs has one immutable tradeoff. The lower the data rate, the less the decompressed file resembles the original file. In simpler words, more compression equals more quality lost.

- Intra-frame & Inter-frame

Now, more specifically for lossy compression technologies, they use two types of compression: intra-frame and inter-frame. The intra-frame type is essentially a compression applied on still images (or told frames) of the video, in which each frame is compressed without reference to any other frame. An example of this type of video compression is Motion-JPEG (explained below).

The inter-frame type, on the contrary, uses redundancies between frames to compress video. If, for example, the background image in a video remains static between frames, the inter-frame techniques store this static information once (in a keyframe) and then store only the changing information in subsequent frames (delta frames). These techniques are much more efficient than intra-frame because they leverage redundant information between frames. But inter-frame compression complicates because it copies data from one frame to another, if the original frame is lost in transmission, the following frames cannot be reconstructed properly.

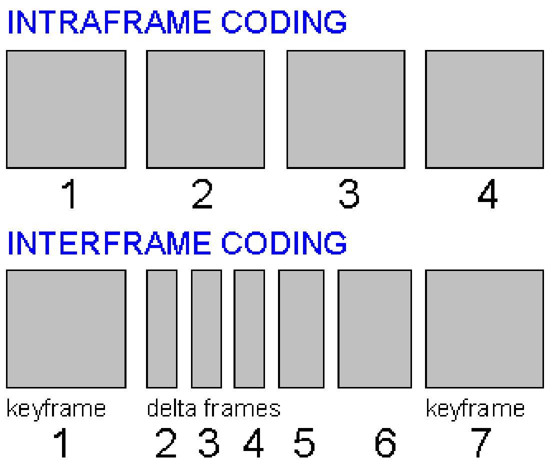


Figure 4: Visual comparison of intra-frame and inter-frame compression

- Motion-JPEG

As it was mentioned before, Motion-JPEG (MJPEG) is a lossy type of video compression and, more accurately, belongs to the intra-frame type. It functions according to JPEG image compression, which stands for “Joint Photographic Experts Group”.

It is a standard image format for containing lossy and compressed image data which is based on the discrete cosine transform (DCT). This mathematical operation converts each frame of the video source from the spatial (2D) domain into the frequency domain. A perceptual model based loosely on the human psychovisual system discards high-frequency information, e.g., sharp transitions in intensity and color hue.

In simpler words, when an image is saved as a JPEG some of the data of the original image is discarded to reduce the file size down to just around 10% of their uncompressed file size.

The compression procedure compares every 8 by 8 block of pixels to a linear combination of 64 standard patterns. Each of those 64 patterns has weight contributing to the 8 by 8 block of pixels. The patterns with higher frequency (more checkerboard-like) have their respective weights lowered depending on the JPEG quality setting. This results in a smaller file size.

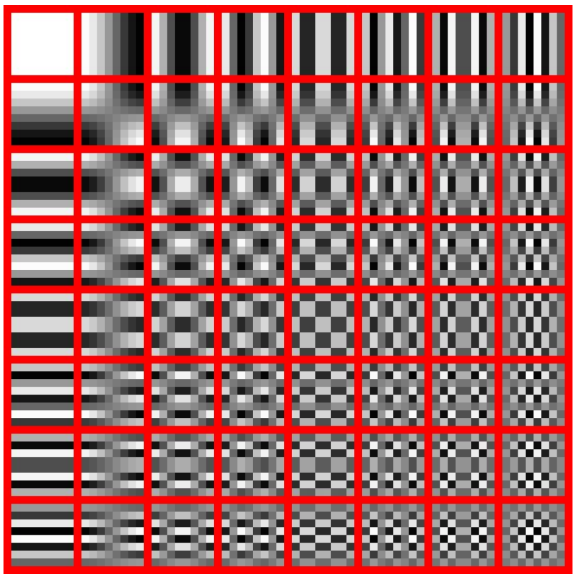


Figure 5: Two-dimensional DCT frequencies from the JPEG DCT

Returning to the MJPEG, it is an intraframe-only compression scheme. Each video frame of a digital video sequence is compressed individually as JPEG image. The MJPEG standard is being widely used by IP cameras, webcams, web browsers, media players, and streaming servers. MJPEG has one clear advantage to everything else and that is that it works in any browser.

Modern interframe video formats, such as H.264/MPEG-4 AVC, achieve compression ratios of 1:50 or better, but MJPEG limits its efficiency to 1:20 or lower because of its lack of interframe prediction. MJPEG also takes 5-20x the processing power of H.264, so it is usually limited to low resolution or HD cameras or cameras with lower FPS rates.

The image quality of MJPEG is directly affected from each video frame's spatial complexity, meaning frames with large smooth transitions compress well and are more likely to hold their original details. MJPEG compressed video is also insensitive to motion complexity. It is neither hindered by highly random motion, nor helped by the absence of motion which are two opposite extremes commonly used to test interframe video formats.

2.2: 360º video

360-degree (360º) videos, also known as immersive videos or spherical videos, are a fairly recent technology in which omnidirectional cameras, or a collection of cameras are used to construct a spherical capture of a space, rather than the rectangular capture. The perspectives of the cameras are stitched together to generate an immersive experience for viewers to experience, placing the viewer within the context of a scene or event rather than presenting them as an outside observer. Playback on normal flat display resembles a panorama view and it is common to give the viewer the ability to control the orientation of the scene and viewing direction.

2.2.1: 360º cameras

One could record a 360º video using either a rig combining multiple cameras or using a dedicated omnidirectional camera [1] which embeds multiple camera lenses. Both ways record overlapping angles simultaneously.

- Field of view

  Field of view (FOV) is the observable area a person can see through its eyes or via an optical device. In the case of optical devices, FOV describes the angle through which the devices can pick up electromagnetic radiation and allows for coverage of an area rather than a single focused point. In VR, as covered below, a large FOV is essential to getting an immersive, life-like experience.

  In human vision, there are two monocular FOVs stitched together by the brain to form one binocular FOV. Individually, horizontal FOV is about 135 degrees and vertical FOV is just over 180 degrees. The binocular FOV is necessary for depth perception and gives about 114 degrees of horizontal view.

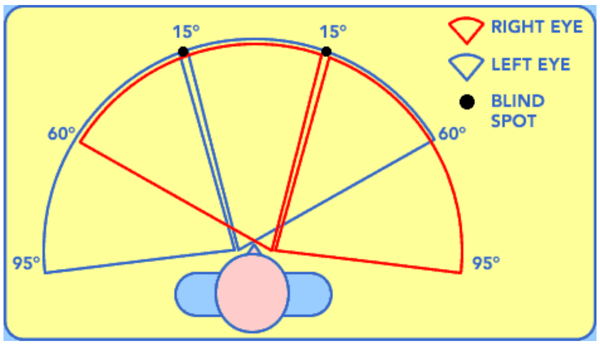


Figure 6: Human vision horizontal FOV

- Image/video stitching

  To solve the FOV limitations of typical images, stitching technology emerged. Multiple overlapping images are stitched together to generate a larger image with wider FOV. To produce seamless results, nearly exact overlaps between images and identical exposures are typically required.

  Further extending image stitching, comes the video stitching. Selected frames of original videos are stitched together by performing image stitching algorithms to generate a stitching template. The subsequent frames of the video are stitched according to the generated template. Footage from two or more separate video streams are usually combined to result to 360^o video. The areas of overlap between the lenses that have been stitched together, called "stitch lines" , are clearly meant to be continuous, though they appear as disconnected lines. This process is commonly performed either by computer software or by built-in camera software that can analyze common features and synchronize and link the different lenses' feeds together. The only area that cannot be viewed is the view toward the camera support.

- Camera systems

  There are mainly two categories of 360^o camera systems; those who consist of multiple wide-view action cameras installed within a frame and those who are containing multiple camera lenses, connected internally in a single camera device. The first category includes rigs such as GoPro's Omni and Odyssey (which consist of multiple GoPro HERO models). These camera systems generally do not perform the video stitching internally, thus not supporting livestreaming. Each camera records its separate video file in its own memory card and the user must stitch and synchronize the video files after the recording.

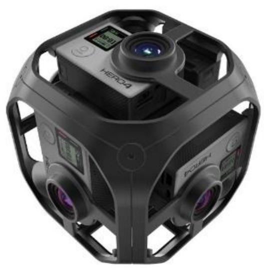


Figure 7: Rigged 360^o camera system, GoPro Omni

The second category cameras can be divided in two more branches; one with cameras consisting of three or more lenses, providing with higher definitions but at a value cost, and one with cameras of exactly two lenses (dual-lens cameras) covering 180^o hemispheres each. Widely known examples of the former are devices like the professional Insta360 Pro 2 (8K definition) and Panono 360 (with up to 36 camera lenses).

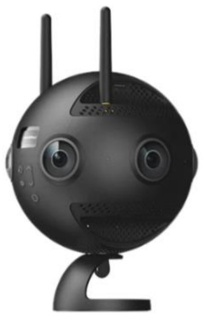


Figure 8: Insta360 Pro 2

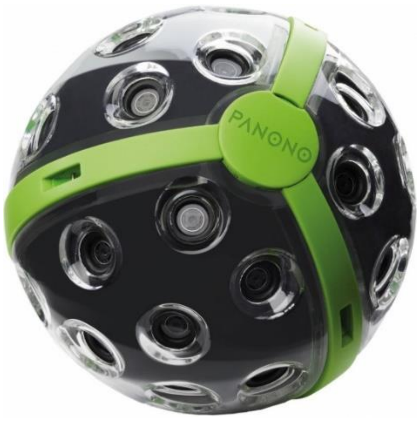


Figure 9: Panono 360 with 36 lenses

Finally, the most widely and financially accessible kind of 360^o camera systems, the dual-lens cameras, can be either handheld or mobile phones' accessories. Handheld examples are the Ricoh THETA V (described extensively in CHAPTER 3) and the Insta360 One X. The accessory type is represented by devices like Insta360 Air and Insta360 Nano, connected with USB Type-C adapters or Lightning connector.

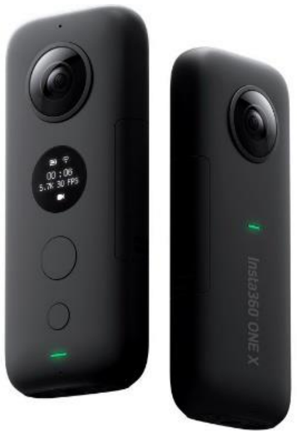


Figure 10: Handheld device, Insta360 One X

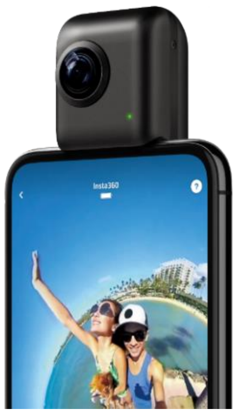


Figure 11: Mobile phone accessory, Insta360 Nano

2.2.2: Map projections

For image segments that have been taken from the same point in space, stitched images can be arranged using one of various map projections [2]. A globe's surface is flattened into a plane in order to make a map. Map projections inevitably distort the sphere surface in some extent and in a variety of ways. Therefore, different map projections exist in order to preserve some properties of the sphere-like body at the expense of other properties.

- Cube mapping

  In cube mapping the image is projected onto the six sides of a cube and is either stored as six separate square textures or unfolded into six regions of a single texture. Each cube face showing 90-degree by 90-degree area of the panorama having such a distortion so when wrapped together the whole environment appear as a sphere. Cube mapping is preferred over sphere mapping because it eliminates many problems such as image distortion, viewpoint dependency, and computational inefficiency. That is why cube mapping is used greatly used for applications like constructing skybox images.

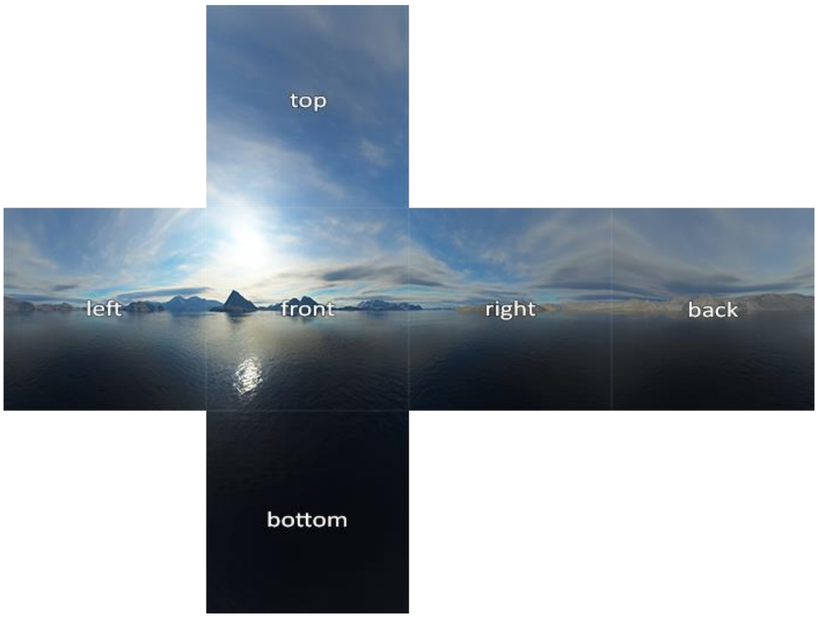


Figure 12: Cube mapping texture regions for a skybox

- Equirectangular/Spherical

  In equirectangular projection or spherical projection the stitched image shows a 360° horizontal by 180° vertical field of view. Panoramas in this technique are perceived as though the image is wrapped into the inside surface of a sphere. In a 2D plane, horizontal lines appear curved, while vertical lines remain vertical in the *front, right, back* and *left* directions.

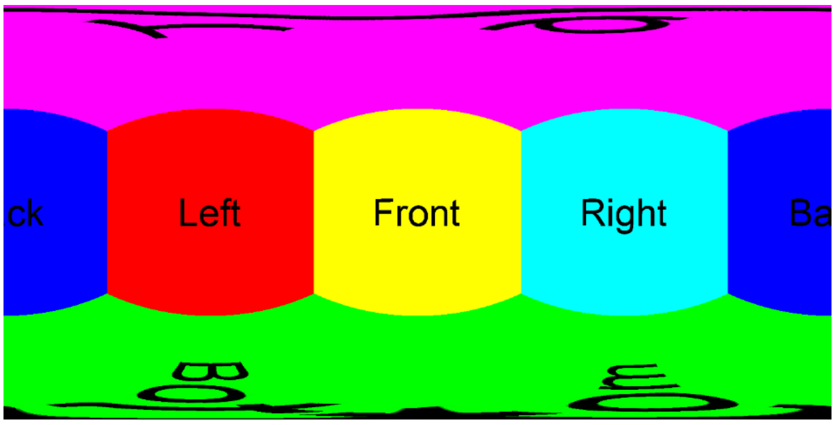


Figure 13: Equirectangular panorama with directional tile mapping

Equirectangular projection is the typical format for 360^o video and is either monoscopic (one image directed to both eyes) or stereoscopic (two distinct images directed individually to each eye). The main difference between equirectangular and spherical projection is that the former refers to the panorama view and the later refers to the view occurring when these panoramas are wrapped on the inside surface of a sphere.

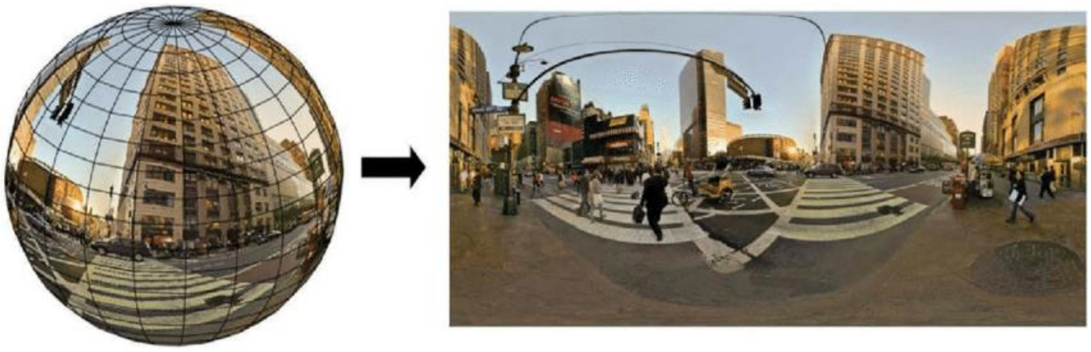


Figure 14: Equirectangular panorama in spherical projection

2.2.3: Video display

360^o videos are typically viewed via personal computers, mobile devices, or dedicated head-mounted displays.

The most common format for computer display is the equirectangular one. Usually, viewing the video via personal computer, meaning the screen display only, might not give to the user the ability to interact with the view. Even if the center of the equirectangular projection can be rotated, it probably will not be possible to change the map projection to a spherical one. When possible, users can navigate around the view by clicking and dragging on the screen if the video projection displays only a FOV of the whole map.

On smartphones, internal sensors such as the gyroscope can also be used to pan the video based on the orientation of the device. Taking advantage of this behavior, stereoscope-style enclosures for smartphones (such as Google Cardboard, Samsung Gear VR or VR Box) can be used to view 360^o videos in an immersive format, much like to virtual reality. The phone display is viewed through lenses contained within the enclosure, as opposed to virtual reality headsets that contain their own dedicated displays.

Software applications are very crucial for this part of the 360^o video distribution. These are the “middleware” that provide us, the viewers, the video stitching, the map projection or the 360^o navigation. Some of them are listed below:

- VideoLAN VLC
  VLC is an open-source multimedia player with cross-platform compatibility. The VLC framework is perfect for most audiovisual media because of the multiple supported media codecs, video formats and streaming protocols. The media player supports 360^o photos, panoramas and in Windows and Mac computers it is able to play spherical video formats. The video can be rotated with the mouse or the arrow keys on the keyboard. Unfortunately, VLC does not fully support 360^o livestreaming videos.
- YouTube
  YouTube is an American social media platform used for online video sharing. The platform took a major leap with 360^o video coverage and have multiple ways to display and navigate in it. One can use a mouse to click and drag the picture to look around in the desktop mode. If the personal computer is connected to a high-end HMD, then the view can be redirected to the immersive VR lenses of the HMD. On mobile phone, there are also two possible ways to display the video. The basic one is a single view; one can tap and drag on the phone screen, similarly with the desktop view, or simply look around utilizing the gyroscope sensor. The second one needs a cardboard or a VR Box so that that the phone is inserted into the low-cost headset for a VR experience.

2.3: Livestreaming

There is a difference between streaming and livestreaming. Streaming is a way to transmit, deliver and playback a completed video file, often from a remote storage location. By transmitting a few seconds of the file at a time, the client devices do not have to download the entire video to play it. Livestreaming, on the other hand, is when the streamed video is broadcasted over the Internet in real time, without first being recorded and stored. Today, TV broadcasts, video game streams, and social media video can all be livestreamed. Therefore, the difference between the two is the creation time and storing of the content in relation to being relayed to the receiver.

2.3.1: Background procedure

The typical procedure, or steps, taken behind the scenes in a livestream is starting with the video, followed by the compression, encoding, segmentation, content delivery network (CDN) distribution and caching, decoding and concludes with the video playback.

- Video capture

  Starting a livestream, one thing is crucial; raw video data, captured by either a camera or a screen recording application. This visual information is then represented as digital data to be processed on the next steps.
- Compression & encoding

  The data is compressed by removing redundant visual information. Commonly, intraframe compression is applied which does not let identical parts to re-render for any subsequent frames. Encoding is the process of converting the livestreaming video data into an interpretable digital format, widely recognized from a variety of devices. Compression and encoding usually come hand in hand with software as video codes, like H.264, VP9 and AV1.
- Segmentation

  On most occasions, video files are far bigger than common images or text files, contain more digital information and take longer when being transmitted. The video data is divided into segments, a few seconds per chunk, before being sent. This part of the livestream process is called segmentation.
- CDN distribution & caching

This step of the process is not mandatory if only the livestream flow is for a real-time peer-to-peer (P2P) connection. If that is not the case, then a Content Distribution Network (CDN) should mediate in order to maintain high quality with minimal latency while serving the stream to multiple viewers in different locations. Basically, a CDN is a distributed network of servers that serve content on behalf of an origin server, cutting down its workload. These servers will also cache each segment of the livestream and most viewers will get the livestream from the CDN cache instead of from the origin server. Even though the cached data is a few seconds behind, the livestream closer to real-time, because it cuts down on round-trip time (RTT) to and from the origin server.

- Decoding & video playback

  After the distribution of the livestream segment, the server is done working on his part. The rest of the procedure is up to the user device to receive, decode, and decompress the segmented video data. Finally, a media player on the user's device interprets the data as visual information.

2.3.2: Types of Livestreaming Communication

The world nowadays has become far more comfortable with video communications in plenty aspects of daily life. From one-to-one talks to mass lectures, there are a lot of events with live videos. But a livestream video can be comprehended with multiple different definitions according to the application of the specific requirements a live event should have.

There are two ways of categorizing livestream events: One is by determining the number of the participants of the event, the other being by determining the delay which those participants will have on their livestream. But those two are not irrelevant with one another, and the participant-based categorization comes under the delay-based dichotomy.

Real-time communication:

Real-time communications, or more accurately “near real-time", usually match with one-to-one and many-to-many streaming applications, where minimum latency is mandatory for QoE. The benefit of real-time communications is there is no delay with the livestream with low quality. The downsides are that it is not adaptive to the end viewer's bandwidth, so low bandwidth end devices will often get a bad experience of the video, maybe sound or pixilation problems.

- One-to-One

  This subcategory of livestreaming is application specific and not for general use. Basically, it is a one-way video data stream from a single camera to a single viewer application. Providing sound can be optional for a wide variety of applications. Servers or CDNs mediate only if there is a need for reverse proxying, link extension or data caching and saving. But the majority of one-to-one video stream are either utilizing physical links (like USB cables) or carried out over Local Area Network (LAN) making use of private links. Some examples of this subcategories are local Webcam applications, real-time device control and closed-circuit television (CCTV).
- Many-to-Many

  Mainly referring to face-to-face video chats and online conferences with more than two users, featuring both image and sound. A common limitation is that it can typically only support a small number of concurrent people on the call, for example the small hundreds, especially when everyone is using a shared internet connection. This type of livestreams serves the need of reducing the physical distance between the users, so it is usually managed by companies acting as CDNs.

Non-real time (NRT) communication or Broadcast:

Non-real time communications or Broadcasts, respectively, go hand in hand with one-to-many streaming events in which the viewers can tolerate with the small delay produced after the processing of the CDN. Not to be confused with pre-recorded video-on-demand streaming.

- One-to-Many

  One-to-many livestreaming is mainly referred to as "Broadcast" and represents events like presentations, premiers, sports coverage. The benefits of broadcast are that they can support a very large number of concurrent video recipients, for example many tens or hundreds of thousands of people, watching the webcast at the same time. This is because of the adaptive bitrate technology which adapts to the bandwidth of the recipient, meaning even low bandwidth devices get a good viewing experience. The downside is that the stream needs processing before it can be viewed. This processing will generate a delay of about 20-30 seconds between the presenter and the viewer.

2.3.3: Streaming Protocols

A protocol, in the network and telecommunications field, is a set of rules defining the way data travels from one system to another. When these rules are distinguished in layers and stack up on top of one another, they form a protocol stack. Each layer can focus on a specific function and cooperate with the other layers. Each layer above the foundation adds complexity.

Video streaming protocols focus on delivering multimedia data (video and/or audio). A video streaming protocol sends chunks of content from one device to another. The way these chunks are reassembled into playable content on the other end is also defined by the protocol. Each individual chunk usually contains compressed video, compressed audio, and metadata such as subtitles, timing info, etc, and how this content is stored while streamed is defined by the transport format. Both sending and receiving device must support the same protocol. These protocols usually sit in the top three layers of the Open System Interconnection (OSI) model, namely application, presentation, and session layers.

Checking a quick background of the transport layer is necessary to understand the foundations on which the streaming protocols are residing on:

- Transmission Control Protocol (TCP)

  TCP operates with a three-way handshake on initialization. The connection starts with the client requesting the server for a handshake, the server responds, and the client acknowledges the response. TCP maintains a session between either end, so it is quite reliable when it comes to packet loss and ordering.

- User Datagram Protocol (UDP)

  UDP, does not require a handshake to transport data. It does not take in concern any bandwidth constrains, providing in that way higher transmission speed but with higher risk for packet loss. It is common for UDP to corrupt the data on route because it does not support error handling.

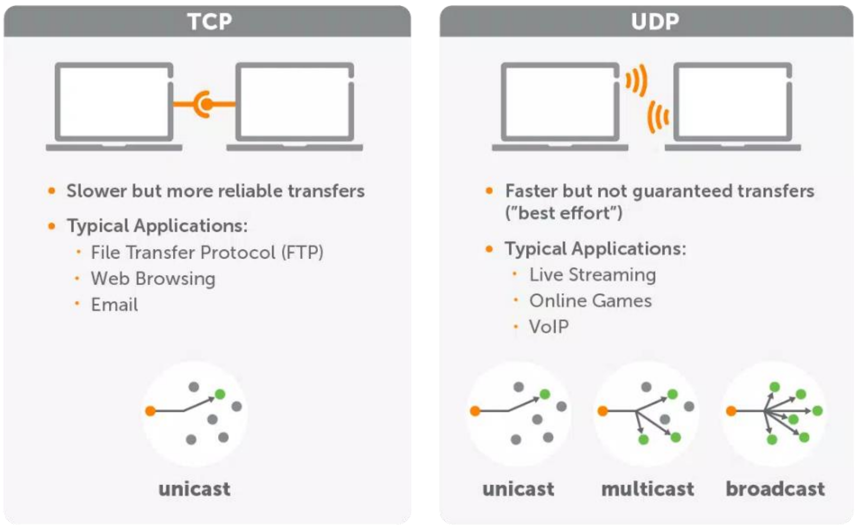


Figure 15: TCP vs UDP comparison for transport layer applications

Traditional Streaming Protocols:

These older protocols were made to support low-latency streaming. Such speed is achieved by transmitting the data using a firehose approach (a large number of messages are broadcast rapidly, repetitively, and continuously) rather than requiring local download or caching. They are no longer the tool of choice for viewer-facing delivery and are not natively supported on most endpoints (browsers, mobile devices and computers). Nevertheless, traditional streaming protocols are still highly useful as ingest formats. In other words, they are no longer used for showing video, but they are used from broadcasters for transporting live streams from source to the media server. From there they can transcode it into an HTTP-based technology for multi-device delivery as part of a larger streaming workflow

- Real-Time Messaging Protocol (RTMP)

  The RTMP specification was developed by Adobe at the dawn of streaming. The protocol could transport audio and video data between a dedicated streaming server and the Adobe Flash Player, reliably and efficiently. Adaptive bitrate streaming eventually edged RTMP out. As Flash began to be phased out over recent years, RTMP remained a useful format for encoding and ingesting video feeds but declined in relevance as a way for delivery.

RTMP runs over TCP, so it establishes and maintains communication between an RTMP Client and an RTMP Server for fast, and reliable data transmission. It breaks a multimedia stream into fragments with their size usually being debatable between the two communicating ends. Large fragment sizes can cause delays in write-operations and very small fragments can increase the load on the CPU, yet small fragment sizes could carry less payload than the bandwidth supports.

There are several variants of the RTMP specification which are worth mentioning:

- RTMPS
  RTMPS is simply RTMP over a TLS/SSL connection. Setting up RTMPS is considered complicated, but it guarantees a level of security. Livestreaming to platforms like Facebook Live needs an RTMPS source.
- RTMPT
  RTMPT is the use of RTMP with encapsulation within HTTP requests. This allows RTMP messages to pass through firewalls, and the message that is encapsulated can be either RTMP Proper, RTMPS, or RTMPE packets within.
- RTMFP
  RTMPF uses RTMP over UDP instead of TCP. The RTMFP was designed for low latency, real-time, live audio, and video communication directly between peers (P2P) without the need for going through an RTMP server.

- Real-Time Streaming Protocol (RTSP)
  RTSP was created as an open-source "network remote control" for media servers. It was developed to control streams without the need for downloading locally. RTSP is not designed to transmit data. The actual transmission of the video data is done by the Real-time Transport Protocol (RTP) and the Real-time Control Protocol (RTCP) delivers the media. RTSP often refers to the entire stack of RTP, RTCP, and. It is dependent on a dedicated media server for viewer-facing content. In the case of IP cameras, IoT devices and other applications requiring close to instant playback from a source, RTSP remains a perfect choice for livestreaming.

RTSP supports several commands such as play, pause, setup, etc which are called control request operations. The initial “OPTIONS” command reports back to the client which of these operations are available in the current stream. When a user initiates a When a video stream is initiated by a user from an IP camera using RTSP, the camera first sends an RTSP request to the streaming server. This jumpstarts the setup process. An end-to-end connection with TCP is maintained and achieves high transmission speed without requiring any local download or caching. Because RTSP depends on a dedicated server for streaming and relies on RTP to transmit actual media, the protocol does not support content encryption or the retransmission of lost packets. And while RTSP and HTTP have similar syntax and operation, due to incompatibilities there is no way to stream it in a web browser without adding additional software.

- RTMP vs RTSP

  Both protocols are on application-level and used to control media streams, having low-latency through a stable connection and being able to deliver in nearly real-time.

  RTMP benefits from its low latency (3-5 seconds) with minimal buffering but it is not optimized for quality of experience or scalability. Playback compatibility is no longer accepted by iOS, Android and most browsers, thus is limited to Flash Player, Adobe AIR and RTMP-compatible players like VLC. It supports H.264, VP6 and VP8 video codecs. It is the go-to solution for first-mile screen-recording or web cameras streaming video ingesting it to larger media broadcasters

  For RTSP, the latency is relatively smaller (1-2 seconds) and it is ubiquitous protocol for IP cameras even in today’s technology, with the same drawbacks as RTMP. It is not widely supported and rarely used for playback, but a few exceptions are Quicktime Player, VideoLAN VLC media player and 3Gpp-compatible mobile devices. The video codecs RTSP supports are H.264, H.265, VP88 and VP9. For applications including P cameras and IoT devices and close to instant playback from a source, RTSP is the best solution.

  On today’s internet, video streaming workflows use RTMP or RTSP to ingest streams (per its designation as a “contribution protocol” or “first-mile” technology) and then utilize another means of delivery to repackage and send the content to be watched on a wide range of devices.

HTTP-Based Adaptive Protocols:

Modern stream delivery protocols take advantage of Adaptive Bitrate Streaming (ABR). With ABR, the protocol adapts to the bandwidth and CPU capacity of the receiving device to provide the smoothest experience possible.

In ABR streaming, a video is transcoded into multiple resolutions and bitrate combinations, and each is referred to as a “rendition”. These renditions can be “1080p 5.0mbps”, “720p 4.0mbps”, “640p 3.2mbps”, “480p 2.0mbps” and “270p 1mbps”. A collection of renditions is a bitrate ladder. When the player starts to playback the video, it senses the available bandwidth. If this is much greater than the highest bitrate on the bitrate ladder, the player safely downloads the highest bitrate for a few segments. the player senses the bandwidth again and if it is still very high, it asks for the highest bandwidth again. If the bandwidth suddenly drops, then the player will probably request for smaller rendition chunk from the server in order to not have loading issues. This process continues throughout the video. This is how the bitrate and quality are adaptively varied to adapt to the varying bandwidth conditions.

- Apple HTTP Live Streaming (HLS)

  HLS is currently the most widely used video streaming protocol by professional broadcasters. It was developed and released in 2009 and was originally developed by Apple to make the iPhone able to access live streams, but now nearly every device supports this format. HLS, developed by Apple, is an ABR protocol but it does not include an encoder which is required to bring in live streams. This is where traditional streaming protocols remain useful for input. Content in HLS is delivered through standard HTTP web servers, with no special infrastructure required, and it is less likely to be blocked by firewalls. Its major benefits are the adaptive bitrate and widely supported playback compatibility, thus leading to fairly high-latency (6-30 seconds).

  In HLS multiple files are created and distributed to the player via different streams, which change adaptively for playback experience optimization. No streaming server is required because it is an HTTP-based protocol, thus the switching is done on the player.

For client distribution, the source is encoded into short chunks at different data rates, about 5-10 seconds long. These are loaded onto an HTTP server along with a text-based manifest file with a .M3U8 extension that directs the player to additional manifest files for each of the encoded streams.

The player monitors changing bandwidth and CPU conditions. If a stream change must occur, the player finds the location of additional streams in the original manifest file, and then acquires the stream-specific manifest file for next chunk of video data. Stream switching is generally seamless to the viewer.

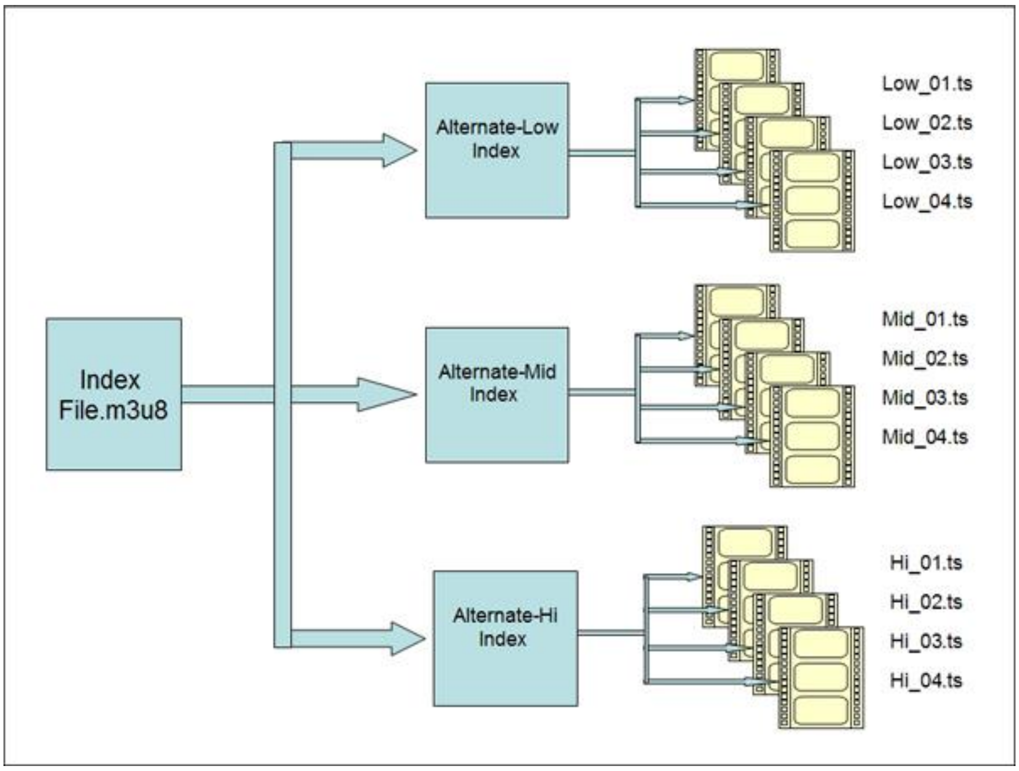


Figure 16: HLS directs the player to different streams and chunks of data within them

Low-Latency HLS (LL-HLS) is a variant of HLS which promises to deliver streams globally with a latency of under 3 seconds. It also offers backward compatibility to existing clients, meaning that any players that are not optimized for LL-HLS can fall back to standard HLS behavior. It is designed to significantly shrinking the latency of HLS while having the same simplicity, scalability, and quality in content delivery. Its major drawback as an emerging spec is that vendors are still implementing support.

- Dynamic Adaptive Streaming over HTTP (DASH)

DASH, or preferably MPEG-DASH, one of the newest streaming protocols that came to be one of the most popular, is developed by MPEG in 2010-2011, published as a standard in 2012 and provided some strong competition for HLS dominating the streaming world. MPEG-DASH also is an ABR protocol but in contrast with HLS' proprietorship of Apple, MPEG-DASH is an open-source option. It also can use content encoded in any format which makes it "codec-agnostic".

Describing in detail the functionality of this standard, a set of encodes (or renditions) of a livestream is packaged by a MPEG-DASH packaging service or software. This packager splits each rendition into chunks of a small duration and records the order in which they are to be delivered into a manifest, called MPD. The origin server stores the packaged video along with the manifest and waits to deliver it to a player, usually via a CDN. The MPEG-DASH compliant player at the client side has an ABR-streaming engine. On start the video player requests the video's MPD file. After receiving the MPD, the player parses it to understand how to play the video. The player then starts requesting the video's chunks in the pre-defined order, decodes the chunks, and displays the video to the user. The player monitors bandwidth conditions and requests from the CDN to send the next chunk of video for suitable bitrate advertised in the MPD. This process continues until either the livestream comes to an end, or the user stops the playback session.

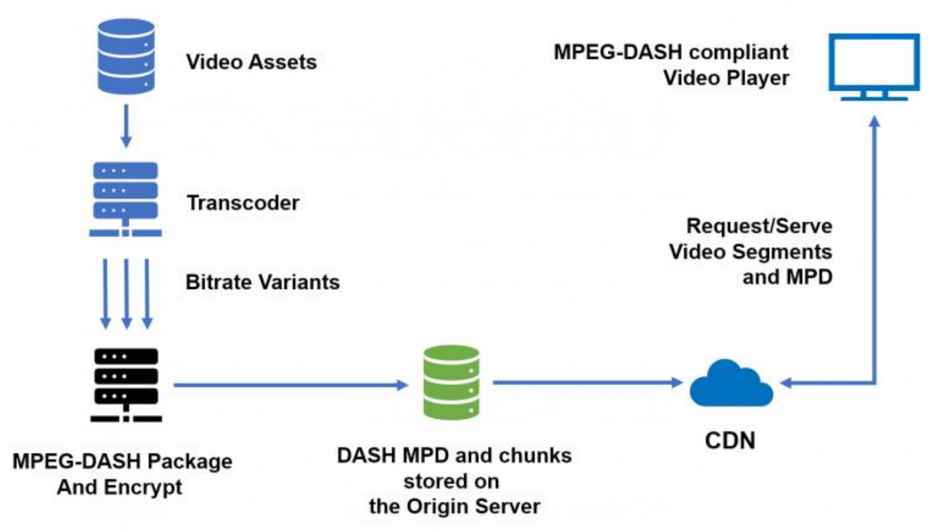

Video Assets
Transcoder
Bitrate Variants
MPEG-DASH Package And Encrypt
DASH MPD and chunks stored on the Origin Server
CDN
Request/Serve Video Segments and MPD
MPEG-DASH compliant Video Player

Figure 17: Course of actions in the MPEG-DASH protocol

Like HLS and LL-HLS, MPEG-DASH has its own low-latency counterpart and that is “Low-Latency Common Media Application Format (CMAF) for DASH” – another emerging technology for speeding up HTTP-based video delivery. Basically, the need for CMAF was emerged due to the inefficiency of HLS using .ts format and DASH using the .mp4 containers based on ISOBMFF. Content distributors had to encode and store the same data twice for reaching both Apple and Microsoft devices. So MPEG established the new standard for reducing complexity when delivering video online. Although vendors have yet to prioritize support for this protocol, the technology shows promise delivering superfast video (at 3 seconds latency or less) at scale by using shorter data segments. When a player is not optimized for low-latency CMAF for DASH it can fall back to standard DASH behavior.

New Technologies:

- Secure Reliable Transport (SRT)

  SRT is designed by Haivision to support low-latency streams when there is noise over the network. By combining UDP’s high speed and TCP’s error-correction qualities, SRT delivers reliable, low-latency livestreams, regardless of network quality. It utilizes the Automatic Repeat reQuest (ARQ), an error-correction mechanism for packet recovery. If there is a gap in the stream, the server recognizes and re-requests those packets from the encoder. Even though the server and encoder keep communicating throughout the transmission, only the missing packets get retransmitted. In this way, it does not cause a mess of overhead or extensive latency.

  SRT carries many benefits for the future of livestreaming. For starter, it provides high quality video withstanding up to 10% packet loss without detectable degradation and maintains stream integrity by accounting jitter and fluctuating bandwidth. It is media-agnostic meaning it acts as a wrapper around audiovisual content, be it MPEG-2 or H.264. It is secure, providing 128/256 bit AES encryption for secure transmission over the internet and finally uses simplified firewall traversal. All that in addition to being open-source, delivering interoperability and longevity.

- Web Real-Time Communication (WebRTC)

  WebRTC is an open web framework for real-time communication. It is combination of standards and protocols alongside JavaScript APIs, which enables peer-to-peer connections between browsers for near-simultaneous exchange of data. This supports browser-to-browser communication and interactive live streaming between individuals.

WebRTC employs HTML5 APIs so that browsers can capture, encode, and transmit live streams between one another with two-way communication. While some streaming workflows require an IP camera, encoder, and/or streaming software, the simplest WebRTC deployments can accomplish everything with a connected webcam and browser. WebRTC can be played back on any HTML5 player unlike Flash-based video.

WebRTC supports H.264, VP8 & VP9, while being compatible with the most widely used browsers (Chrome, Firefox, Safari) without any plug-in. The latency of this protocol averages at 500 milliseconds but it does not provide scalability without the of a streaming server or service.

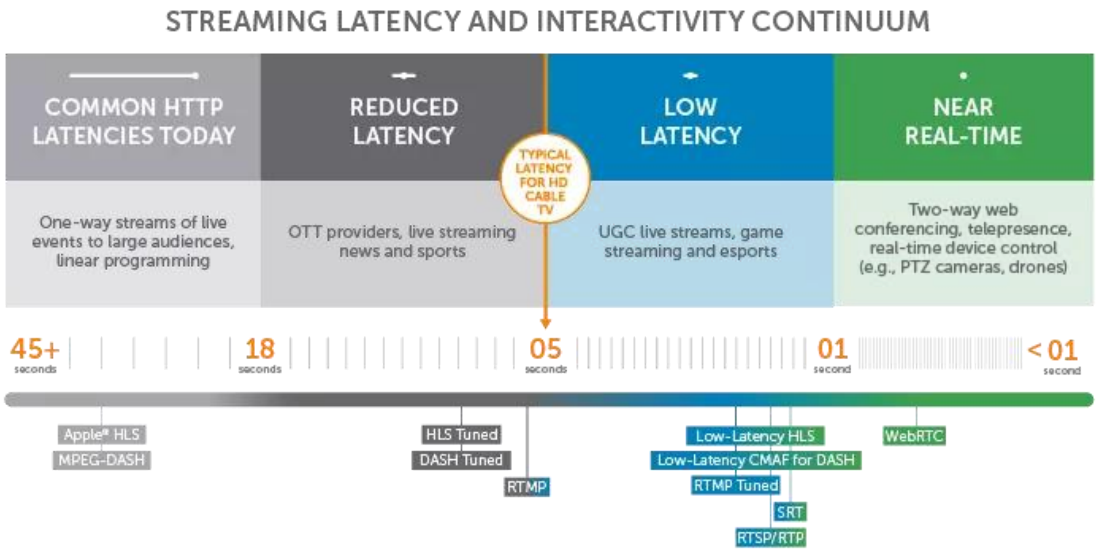

STREAMING LATENCY AND INTERACTIVITY CONTINUUM
COMMON HTTP LATENCIES TODAY
REDUCED LATENCY
LOW LATENCY
NEAR REAL-TIME
TYPICAL LATENCY FOR HD CABLE TV
One-way streams of live events to large audiences, linear programming
OTT providers, live streaming news and sports
UGC live streams, game streaming and esports
Two-way web conferencing, telepresence, real-time device control (e.g., PTZ cameras, drones)
45+ seconds
18 seconds
05 seconds
01 second
< 01 second
Apple® HLS
MPEG-DASH
HLS Tuned
DASH Tuned
RTMP
Low-Latency HLS
Low-Latency CMAF for DASH
RTMP Tuned
SRT
RTSP/RTP
WebRTC

Figure 18: Protocols comparison for streaming latency and interactivity continuum

2.4: Virtual Reality

The concept of Virtual Reality (VR) has been around since the early 20th century, when a science fiction writer, Stanley G. Weinbaum, described the idea of interactive movies which can provide sight, sound but also taste, smell, even touch. Since then, there have been decades of experimentation, with the 1992 film “The Lawnmower Man” shaping mainstream perceptions of VR for some time afterwards. Nevertheless, the current age of VR began in 2010 with the first VR headset prototype, which then evolved into Oculus Rift. A few genres of “realities” exist because of today’s technology but there is a clear distinction between them.

2.4.1: Categories

- Virtual Reality (VR)

  A VR environment is a completely computer-generated. All scenery and objects appear to be real in sight, thus making the user feel immersed in his surroundings. Therefore, the typical VR is called “immersive VR”. This environment is presented to the user through a device, the VR headset or Head Mount Display (HMD). Immersive VR is famous for video gaming and sports exercising because it simulates the whole experience as if the playable character is the user himself.

  VR applications do not stop on gaming and sports. Various areas have already taken advantage of this technology such as medicine, culture, education, and architecture. From guided virtual museum visits to practical experience in working positions, VR allows humanity to cross boundaries that would otherwise be unimaginable.

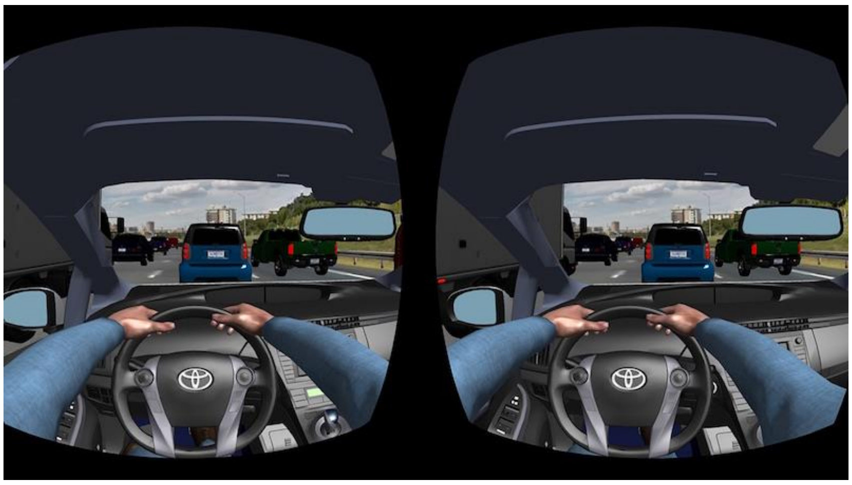


Figure 19: Inside a VR headset playing VR Rush Hour: Austin

- Augmented Reality (AR)

  AR is when the real physical world is enhanced using digital visual elements, sound, or other sensory stimuli. It is a growing trend among companies involved mostly in mobile computing and business applications, like marketing. Retailers and other companies can use AR to promote products or services, launch novel marketing campaigns, and collect unique user data.

  Smart eyewear devices could be a breakthrough for AR because they do not show a tiny portion of the user's landscape (as smartphones and tablets do), but they provide a more complete link between real and virtual realms if it develops enough to become mainstream.

Figure 20: AR in gaming, Pokémon GO

Figure 21: AR applicated in smart eyewear

- Mixed Reality (MR)

  While VR allows the creation of a virtual world from scratch, what AR does is add virtual elements to the real surrounding environment.

  Mixed Reality brings together real world and digital elements, meaning it combines elements of both VR and AR. In MR, interaction is possible with both physical and virtual items and environments, using next generation sensing and imaging technologies. MR offers real view of the surrounding world but also immersion and interaction with a virtual environment without using any special equipment besides the MR headset. MR breaks down basic concepts between real and imaginary, offering an experience that can change the comprehension of everyday activities like working, gaming, and education to name a few.

Figure 22: Industrial worker using MR headset Microsoft's HoloLens

2.4.2: Hardware

- Degrees of Freedom

  Degrees of Freedom (DoF) is a term referring to motion around or along an axis in a 3D space. It is the number of ways a rigid object is able to move in that 3D space. According to the Cartesian coordinate system of 3D space, there is a triplet of coordinate axis, X-Y-Z. Each of these can allow a translational movement along the axis and a rotational movement around the axis. The three rotational movements are known as pitch, yaw and roll while the three translational movements are called forward/backward, left/right and up/down, depending on the "positive" direction. In VR, DoF is an essential concept with which human movement converts into movement in the VR environment.

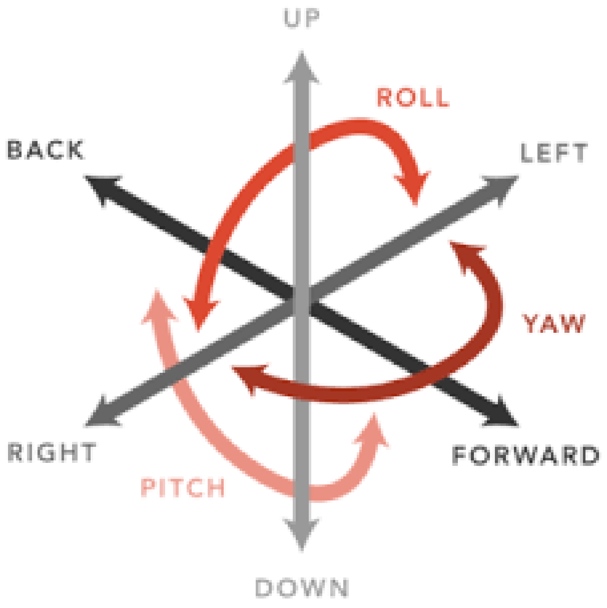


Figure 23: Degrees of Freedom in a 3D space

- VR headsets

A Head Mount Display (HMD) is wearable display device for the human head that provide visuals for the human eye. It can be either monocular or binocular. The difference with VR headsets is that the latter are basically HMDs combined with Inertial Measurement Units (IMUs). IMUs measure the rotational movements on a 3D space and positional tracking sensors, either HMD built-in or external, measure the translational movements. With this hardware, a VR headset is considered an input device which tracks the 3D movements of the user’s head. A VR headset can either be “3-DoF” or “6-DoF”. The 3-DoF headsets, like Google Cardboard or VR Box, are usually glasses for smartphones, which only have gyroscopes. So, they cannot perceive a change in the user’s translational movement. On the other hand, 6-DoF headsets, like Oculus Rift and HTC Vive, allow full transitional and rotational tracking, thus offering the user a lot more freedom and making the experience closer to immersive.

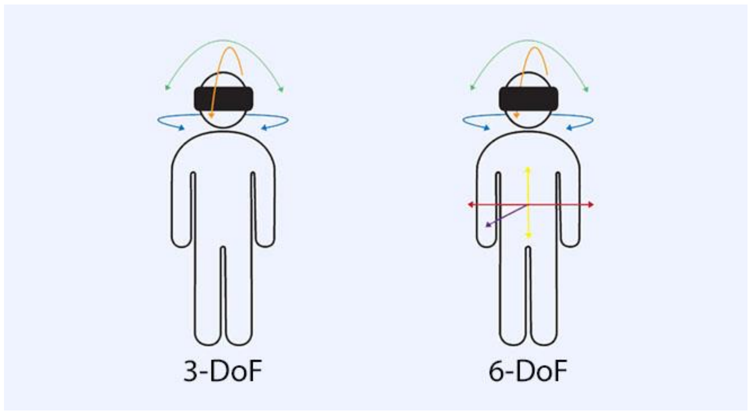


Figure 24: Difference between a 3-DoF headset and 6-DoF headset

2.4.3: Software

A VR environment is fundamentally a 3D environment which has some special features when it comes to sensory input and output. So, before developing a VR environment, it is firstly required to develop a 3D environment.

- Unity game engine

For this task, the preferred choice are game engines, which are software frameworks primarily used for developing video games. They typically consist of a rendering engine for 2D and 3D graphics, a physics engine for managing the interactions in the environment and they include relevant libraries and other software tools for scripting, networking, memory management and much more.

The Unity game engine is ideal for cross-platform developing of 3D environments, and not only for desktop usage. It is considered an easy-to-use engine because of its simplicity, flexibility in language scripting and a great variety of tools.

Unity mainly supports C# and JavaScript, and its scripting API is fully documented for both of them. Though, for the majority of developers and for the contents of this thesis, the language of choice is the former.

- UnityEngine namespace in Scripting API

  In C#, a namespace is, like a library, a method of preventing name conflicts while containing declarations and implementations. In the *UnityEngine* namespace there is the *MonoBehaviour* class, the base class from which every Unity script derives.

  The *MonoBehaviour* class offers basic callback functionality, public and static methods, and properties for Unity's *GameObjects* to employ on runtime. Some of the basic functions are explained below:

  - Awake: This function is used to initialize variables or states before the application starts so it is called when the script instance is being loaded
  - Start: This function is called when the script becomes enabled, exactly once in the lifetime of the instance. This happens after the initialization in *Awake* and before the first *Update* call.
  - Update: If the MonoBehaviour script is enabled, the *Update* function can be considered a loop method because it is called before the rendering of every frame. That is why it usually implements the main functionality of the script. There is also the *FixedUpdate* and the *LateUpdate* variant, from which the former is frame-rate independent and used for physics calculations, while the latter is useful for ordering script execution because it is called after all *Update* functions have been called.
  - OnDestroy: The *Destory* function actually belongs to the *Object* class because it removes every possible component, script or *GameObject* in runtime.

MonoBehaviour class though provides the *OnDestroy* callback, which is invoked just before the script gets destroyed by the *Destroy* function.

An evenly important piece of the *UnityEngine* namespace is the *Coroutine* class. A *Coroutine* is a type of function that can pause its execution with the *yield* keyword and automatically resume in the next frame. With the *WaitForSeconds* class, the coroutine execution can be yielded for a given number of seconds. Coroutines start within a *MonoBehaviour* script with the *StartCoroutine* function, which needs an *IEnumerator* routine as an argument, and can be stopped with the *StopCoroutine* function. The *Corutine* type is very useful for multitasking and is well-suited for implementing event loops and iterators.

- Steam VR

SteamVR is a runtime, provided by Valve's Steam client, which powers VR experiences on a desktop. It runs over the OpenVR API which allows access to VR equipment without the applications knowing the specific hardware they are targeting. SteamVR is also a Unity Plug-in, maintained by Valve, to smoothly interface the homonym runtime with the game engine. It handles the VR equipment positional and rotational tracking, the loading of the VR controllers' 3D models and also their input and haptic output.

The class of interest in the SteamVR Plug-in is the *Player* class, inside *Valve.VR.InteractionSystem* namespace. This is a *MonoBehaviour* derived class which includes references for other *MonoBehaviour* scripts handling the HMD and both controllers.

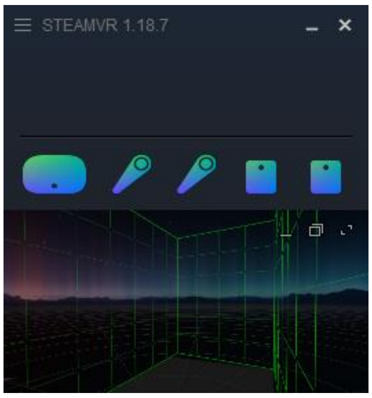


Figure 25: SteamVR runtime with room setup

CHAPTER 3: IMPLEMENTATIONS

In the content of this thesis, 360^o video livestreaming through VR [3] is implemented in various ways. The physical equipment available is limited to a single 360^o camera, that being Ricoh THETA V, and to two HMDs, one being HTC Vive and the other being VR Box. Nevertheless, this limitation allows to test different streaming protocols and different end-platforms for visualizing the video. All implementations were tested on Windows 10 Home 64-bit, versions 2004 (OS Build 19041.804) and Version 21H1 (OS Build 19043.1237).

Figure 26: Ricoh THETA V

Figure 27: HTC Vive headset

Figure 28: VR Box for smartphones

3.1: Describing Ricoh THETA V

Ricoh THETA V was selected after a thorough research around 360^o cameras. Research was conducted based on some criteria, which are onboard stitching, live preview capabilities, wireless connectivity, and affordable price. Ricoh THETA V is a perfect choice for developers and has a lot of bonus features against its competitors. Namely having an open-source Application Programming Interface (API), and running on Android, which allows the development of various plug-ins. All that come along with a detailed documentation.

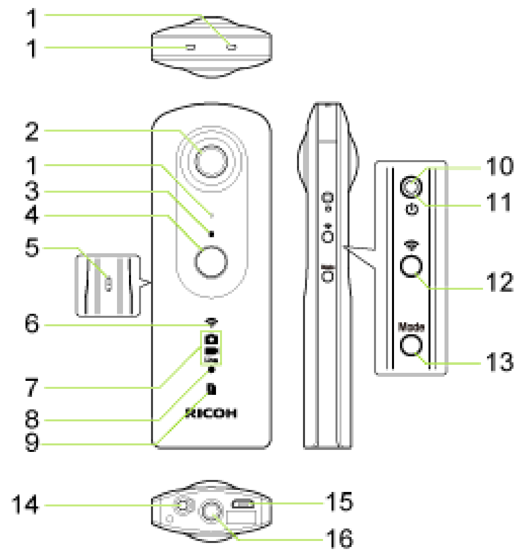


Figure 29: THETA V components

1. Microphone
2. Lens
3. Camera status lamp
4. Shutter button
5. Speaker
6. Wireless lamp
7. Capture mode lamp
8. Video recording lamp
9. Memory warning lamp
10. Power lamp
11. Power button
12. Wireless button
13. Mode button
14. Microphone terminal
15. USB terminal
16. Tripod mount hole

3.1.1: Characteristics

It has 19 GB internal memory, capable of 4K video at 30 FPS and HDR shooting. It also has a four-channel microphone allowing spatial audio in video capturing.

3.1.2: Ricoh Desktop Application

Ricoh provides a desktop application which supports all models in the series, THETA V included. It is used for viewing 360^o still images and videos on a computer by dragging them to the main app and rotate them as well as manipulate them by dragging them or with the navigation panel. Converting them to other formats as well as save them as 360^o videos is also possible. The app can update the device firmware to the latest version as it is recommended by Ricoh.

3.1.3: Plug-ins

RICOH THETA V utilize an Android-based operating system, so in addition to the basic 360° camera functionality, installing plug-ins - basically Android applications - allows even greater flexibility in controlling the camera.

In order to enable developer mode, registering for the RICOH THETA Partner Program is required. Developing is done through the Android Studio having downloaded the Ricoh THETA Plug-in Software Development Kit (SDK). Plug-ins, either self-developed or from other partners, can be viewed from the Plug-in Store webpage, but they are downloaded via the Ricoh Desktop App. The same app is used for plug-in management when the camera is connected via USB to the PC.

3.1.4: THETA Web API v2.1

This is the API used to access the camera's functions through commands and options. THETA Web API v2.1 conforms to Open Spherical Camera (OSC) API v2.0 by Google. OSC API is a proposed set of commands for various spherical cameras with built-in Wi-Fi capabilities. Developing any app against OSC lets it control any connected spherical camera that implements OSC.

Each camera creates an Access Point (AP) and serves a discoverable, WPA2-PSK password protected, Wi-Fi network with a human-readable Service Set Identifier (SSID). After that, the OSC camera must implement a HTTP 1.1 webserver on default HTTP port 80, which must response to GET and POST requests. Commands input/output JSON and are part of HTTP requests/responses respectively. In addition to official OSC commands and parameters, THETA-specific commands are distinguished by an underscore prepended to their name. More on THETA Web API will be discussed in the below implementations.

3.1.5: Wi-Fi modes

THETA V has to modes of Wi-Fi connectivity. The first is Direct mode or "AP mode" where the device creates a local Wi-Fi network without internet access and operates as an Access Point. For accessing the Web API, the viewer device must connect on THETA's WLAN and reach the gateway's IP 192.168.1.1, which is the camera's IP.

The second and more convenient Wi-Fi mode is the Client mode or "CL mode" in which the device connects as a client to an already existing local Wi-Fi network provided by a Wi-Fi router. For this mode to be activated, there are two possible ways. First is by Wi-Fi Protected Setup (WPS) configuration if the router supports it. By pressing the WPS button on the router and by pressing shutter & Wi-Fi & mode buttons on the camera at the same time. THETA V will connect automatically on the main WLAN of the router. Second way is by utilizing the smartphone App and is described below.

3.1.6: THETA Smartphone Application

THETA Smartphone App is the easiest way to control everything on Ricoh THETA V. The mobile device should firstly connect to the camera's local Wi-Fi network. The mobile app will automatically recognize and connect to THETA V because the IP (192.168.1.1) is already known by the network's gateway (the camera). In the settings section there is an option for connecting THETA V on a WLAN by providing wanted SSID and password. After that, the mobile app can make the connection via WLAN, because both devices ae connected to that. The app offers full 360^o and dual-lens VR navigation for images and videos in camera's internal memory. It utilizes almost every command from THETA Web API v2.1 thus providing live preview feed with image and video capturing, resolution and frame rate control, firmware update from internet, plug-in management and a lot of other settings and configurations.

3.2: Bridged Wireless Access Point

This thesis required working from a PC with a wired connection to a 10.64.44.0/23 network while the WLAN was a different network with 192.168.1.0/24 domain. Ricoh THETA V would only connect wirelessly and in order to communicate with the PC, they should be on the same LAN. A TL-WN821N wireless USB adapter did not solve the problem due to be problematic. The preferred workaround was having a Raspberry Pi 3 Model B (RPi) function as a bridged wireless AP, providing THETA V with a 10.64.44.0/23 IP. The procedure is described below with Shell sessions on an *Raspbian gnu/linux 10 (buster)* operating system.

To connect the Ethernet and wireless networks a bridge network device is added named br0 by creating a file */etc/system/network/02-br0.netdev* with contents:

```
[NetDev]
Name=br0
Kind=bridge
```

Adding the built-in Ethernet interface (*eth0*) as a bridge member by creating */etc/system/network/04-br0_add-eth0.network* containing:

```
[Match]
Name=eth0
[Network]
Bridge=br0
```

The software will add the wireless interface (*wlan0)* to the bridge when the service starts, so no actions need to be taken.

Enabling the *systemd-networkd* service to create and populate the bridge

```
sudo systemctl enable system-networkd
```

Network interfaces which are members of a bridge device should not have an IP address, but the bridge device itself shall in order to reach RPi within LAN. The DHCP client in RPi, *dhcpd*, requests an IP for every active interface according to the */etc/dhcpcd.conf* file. So *eth0* and *wlan0* must be blocked and *bro* needs to be defined in the file, by adding the lines

```
denyinterfaces wlan0 eth0
interface br0
```

For the configuration of the AP software, *wpa_supplicant* is used, and the below lines must be added in */etc/wpa_supplicant/wpa_supplicant-wlan0.conf*:

```
country=GR
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1

network={
    ssid="10.64.44.0/23"
    mode=2
    frequency=2412
    key_mgmt=WPA-PSK
    proto=RSN WPA
    psk="********"
}
```

A later addition was done in order to have a static IP on the Bridged AP, creating */etc/system/network/12-br0_up.network* with contents:

```
[Match]
Name=br0
[Network]
MulticastDNS=yes
Address=10.64.44.242/23
Gateway=10.64.44.1
DNS=10.64.44.1
```

This step, creating a Bridged Wireless Access Point is very crucial for the PC-camera communication in the following implementations and is simultaneously used by the smartphone having THETA App. The IPs assigned were 10.64.44.242 (RPi), 10.64.45.228 (THETA V), 10.64.45.161 (PC Windows) and 10.64.45.224 (PC Ubuntu), stable during all implementations.

3.3: Equirectangular and non-VR livestream solutions

While putting through this thesis, many approaches either led to an immersive VR implementation or could not be considered for further research due to missing features.

3.3.1: OBS webcam

Open Broadcaster Software (OBS) is a free and open-source cross-platform streaming and recording program built with Qt and maintained by the OBS Project. Since 2016, the software is now referred to as OBS Studio. OBS 27.0.1(64-bit, windows) is used in this implementation on operating system Windows 10. It is recommended to always update to latest firmware on the camera either by Desktop App or Mobile App. As of the release date of this thesis, downgrading firmware is not possible on THETA cameras. This implementation was checked through time with firmware versions 3.50.1, 3.60.1 and 3.70.1 for THETA V.

Ricoh THETA V communicates with PCs over USB cable implementing Media Transfer Protocol (MTP). The PC or OBS is capable of recognizing THETA V as a webcam but in order to enable livestream support, it is required to install the THETA UVC Blender application. UVC is basically the computer driver for THETA cameras which supports 360^o features when livestreaming and in OBS it is interpreted as different device from THETA V camera. Older versions of UVC were using uncompressed video formats like MJPEG, but since v1.5.0 it uses H.264 compression for high-resolution videos and outputs equirectangular data. There are two available resolutions, 1920x960 defined as FullHD or 2K and 3840x1920 defined as 4K, and the frame rate is constant at 29.97fps (30fps for convenience) in regular conditions. In this implementation the UVC version tested was the latest, Ricoh THETA UVC V/Z1 v2.0.0 (which works for both THETA V and THETA Z1 cameras) and has two available “devices”, one for each resolution mentioned above.

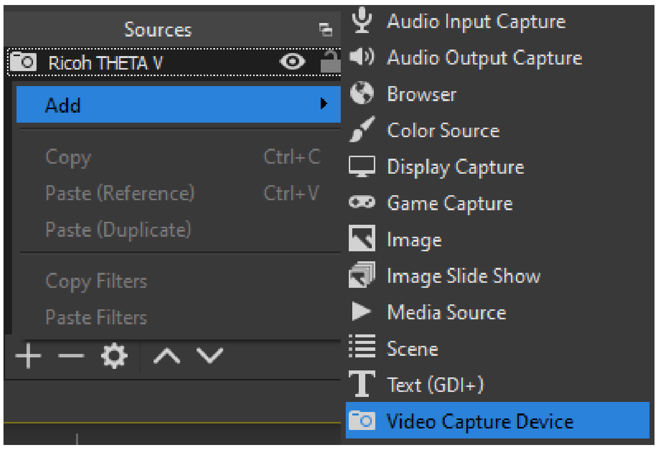


Figure 30: Creating Video Capture Device in OBS

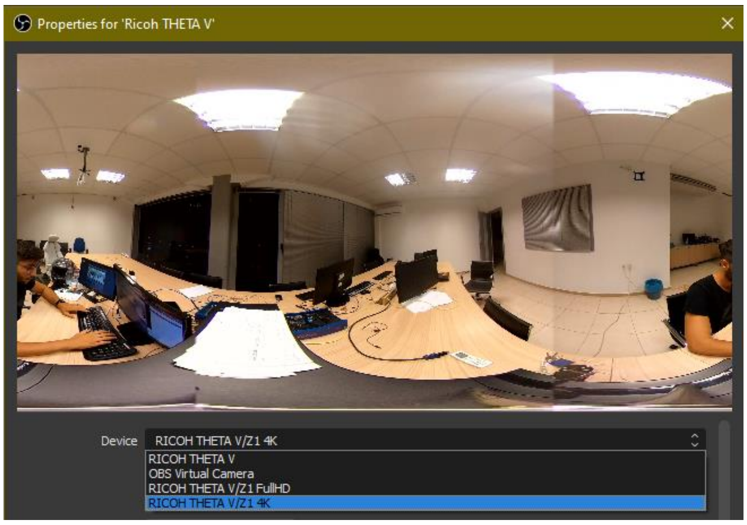


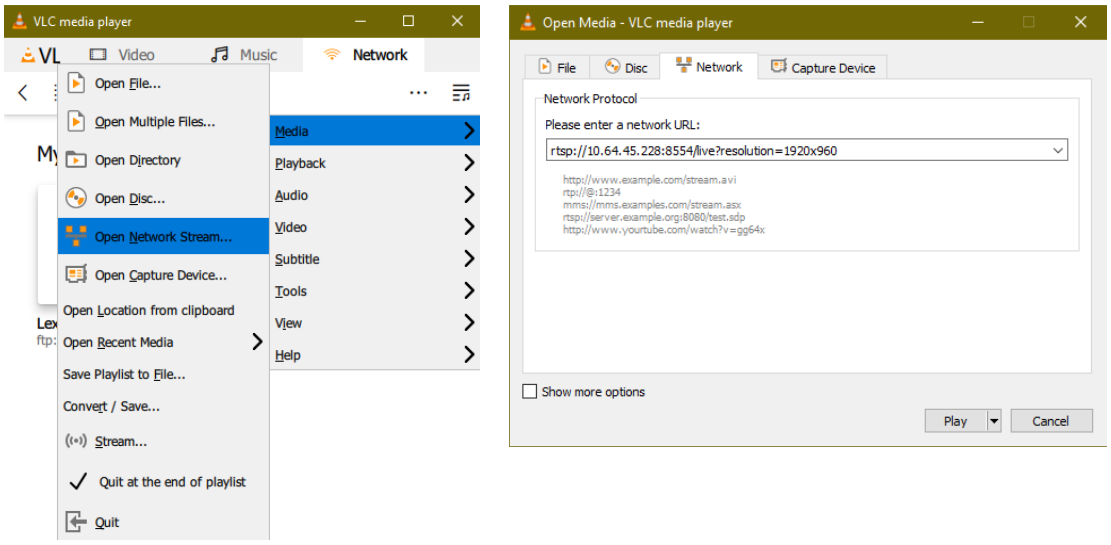
Figure 31: Devices from UVC v2.0.0 and equirectangular USB live preview in OBS

3.3.2: VLC capturing RTSP

VLC was also mentioned in chapter 2.3.5 and here it is used as a streaming tool to catch an RTSP livestream, one exposed by THETA V itself using a Plug-in. Two product versions were tested, 3.0.12 stable version and 4.0.0 nightly build, an experimental version which is unstable. 3.0.12 was already supposed to support 360° video and 3D audio, but with no success, so the experiment version 4.0.0 needed to be tested. Again, 360° navigation was not supported from an equirectangular stream. Therefore, both versions are used for equirectangular video display.

There were two Partner Plug-ins available, “THETA RTSP Streaming” (here tested with v1.0.3) and “Device WebAPI Plug-in” (here tested with v1.1.0) which implemented an RTSP server on THETA V.

The former is using the port 8854 and the /live endpoint for exposing the RTSP stream. A URL parameter, “resolution”, is available for configuration and the four possible options are "640x320", "1024x512", "1920x960" or "3840x1920". The framerate is fairly stable and unconfigurable at 30fps. An example of the livestream URL for the 1920x960 resolution can be formatted in the following: *rtsp://10.64.45.228:8554/live?resolution=1920x960*



rtsp://10.64.45.228:8554/live?resolution=1920x960

Figure 32: VLC 4.0.0 steps to opening a URL stream

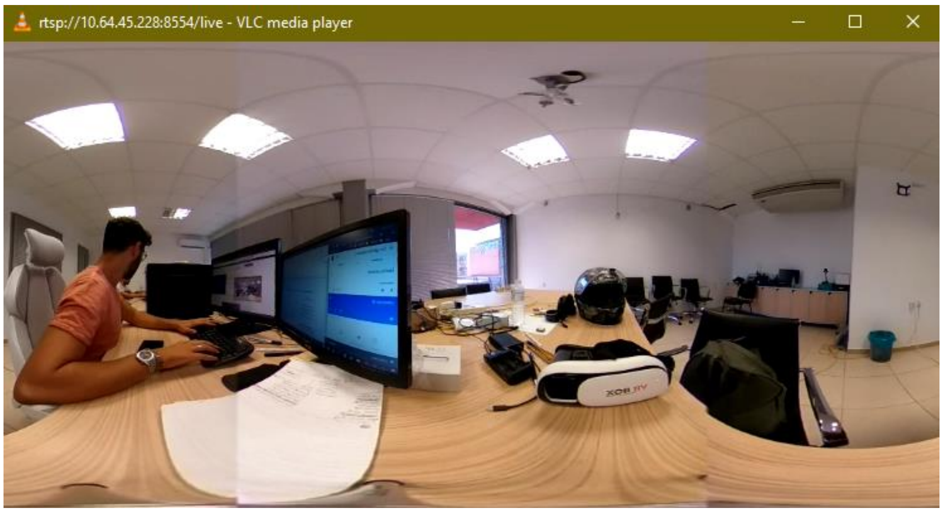


Figure 33: Equirectangular 1920x960 livestream from THETA RTSP Streaming Plug-in

The latter, as its name suggests, is using the THETA Web API to function upon. The Plug-in is developed in Japanese so not much info is presented here. It offers an HTTP webserver exposed on port 8888. The RTSP stream though is exposed on port 8086, with no available resolution setting, therefore the URL is constantly *rtsp://10.64.45.228:8086*. An important difference with the THETA RTSP Streaming Plug-in is that the live preview format is not equirectangular. Instead, the livestream appears to be in spherical projection, and it is cropped out in a specific FOV of about 90^o wide and 60^o tall. Due to VLC capabilities, it does not support spherical navigation

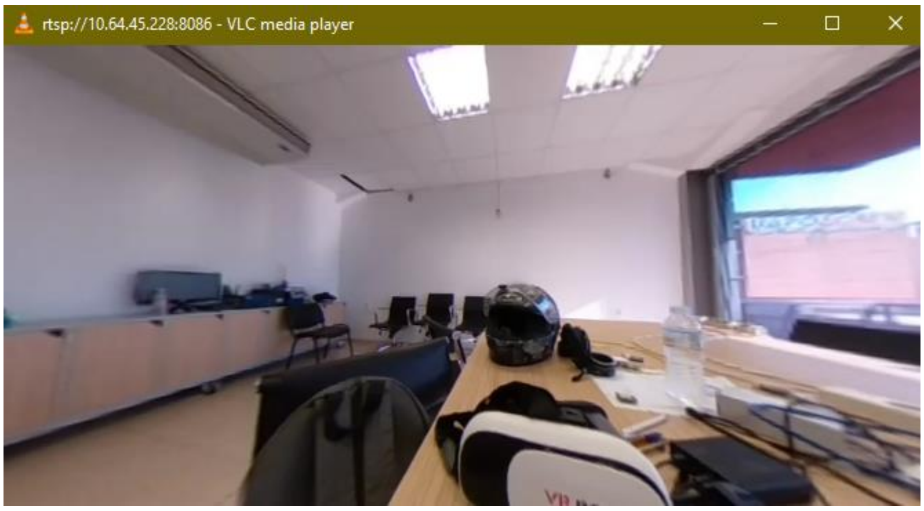


Figure 34: Cropped spherical RTSP livestream from Device WebAPI Plug-in

3.3.3: MJPEG with spherical navigation

Continuing with the Device WebAPI Plug-in, in the Live Preview section, it offers another viewable form of the 360^o livestream. This time it exposes an MJPEG stream on a randomly produced endpoint on port 9000. Copying the exact URL and opening it in another tab will result on the same view which the RTSP option provides. Although, when the “MJPEG Display” button is pressed, the Plug-in webserver provides a web browser spherical view supporting mouse navigation. In the Live Preview section there is also a “Camera Settings” field in which the MJPEG resolution can be set with the options "640x320", "1024x512", "1920x960" or "3840x1920". The MJPEG livestream cannot be seen simultaneously with the RTSP livestream and vice versa. The Plug-in’s authors report that there is an option to support split screen mode, in order to view the live preview with a smartphone VR headset. No such feature was found as of the release of this thesis.

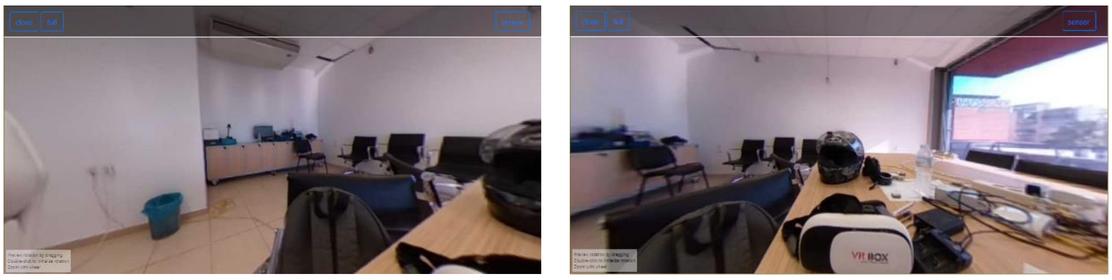


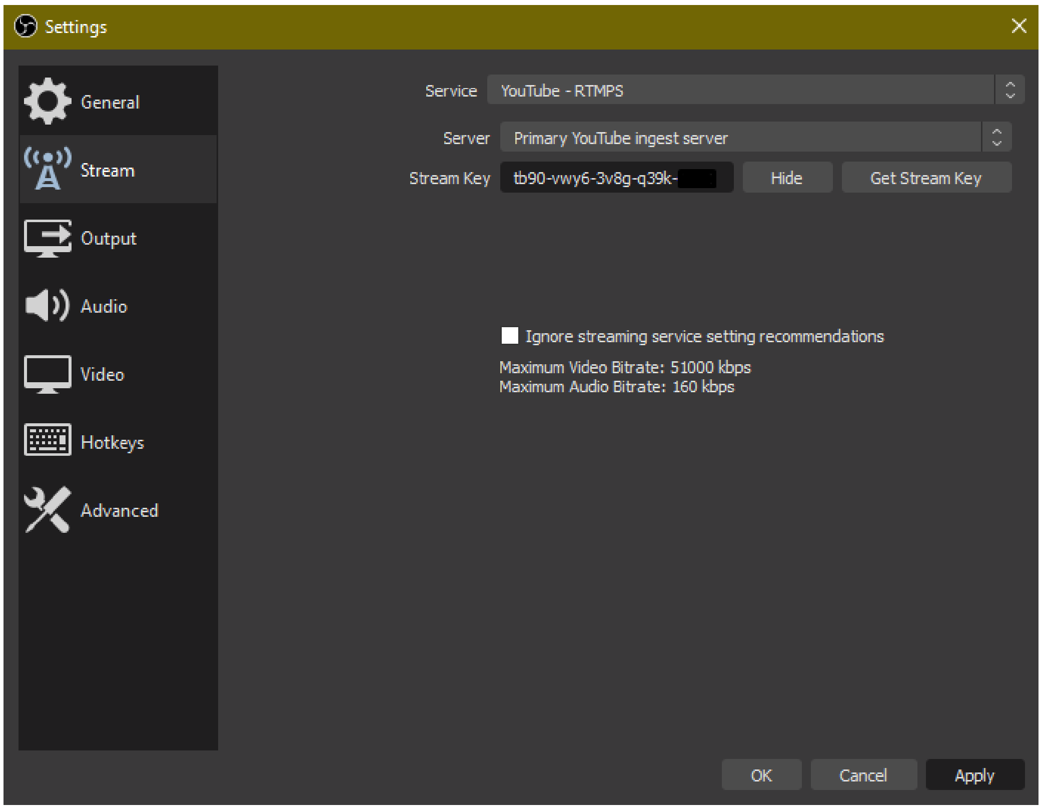
Figure 35: MJPEG livestream with spherical navigation

3.4: YouTube livestreaming with RTMP

The same implementation is also supported from Facebook but, in this document, only YouTube is examined. YouTube lets anyone with a registered channel to create and publish livestream. By clicking the YouTube Studio option in the Account section, the site opens the Channel Dashboard. On the top right corner there is a “Go Live” button and clicking this will load the livestream configuration page. The important settings here is enabling “360° video” and selecting “Ultra low-latency”, which does not support 1440p and 4K resolutions. The URL *rtmp://a.rtmp.youtube.come/live2* is the Primary YouTube ingest server, which receives the streams using RTMPS, authorizes and registers if their *Stream key* is valid and lastly prepares the stream for the viewers.

3.4.1: OBS redirecting to YouTube

In OBS (v27.0.1), in the *Stream* section, the right configuration is to select “YouTube – RTMPS” in the *Service* field. *Server* field remains as it is and in the *Stream key* field, it must be filled in with the stream key provided in the YouTube livestreaming dashboard. After that, pressing “Start Streaming” in the OBS should start the livestream in YouTube platform.



Service YouTube - RTMPS

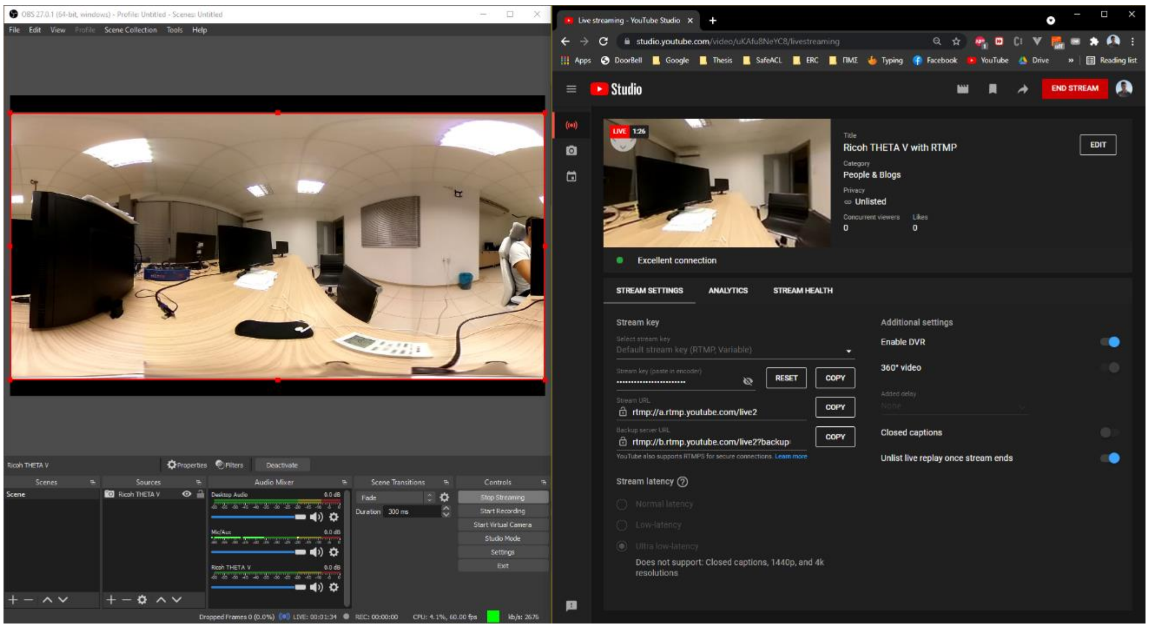
Server Primary YouTube ingest server

Stream Key tb90-vwy6-3v8g-q39k-[redacted]

Ignore streaming service setting recommendations

Maximum Video Bitrate: 51000 kbps
Maximum Audio Bitrate: 160 kbps

Figure 36: OBS setting for YouTube streaming



Title: Ricoh THETA V with RTMP
Category: People & Blogs
Privacy: Unlisted
Concurrent viewers: 0
Likes: 0
Excellent connection

Stream key
Select stream key: Default stream key (RTMP, Variable)
Stream key (paste in encoder): ••••••••••••••••••••••••
Stream URL: rtmp://a.rtmp.youtube.com/live2
Backup server URL: rtmp://b.rtmp.youtube.com/live2?backup
YouTube also supports RTMPS for secure connections. Learn more

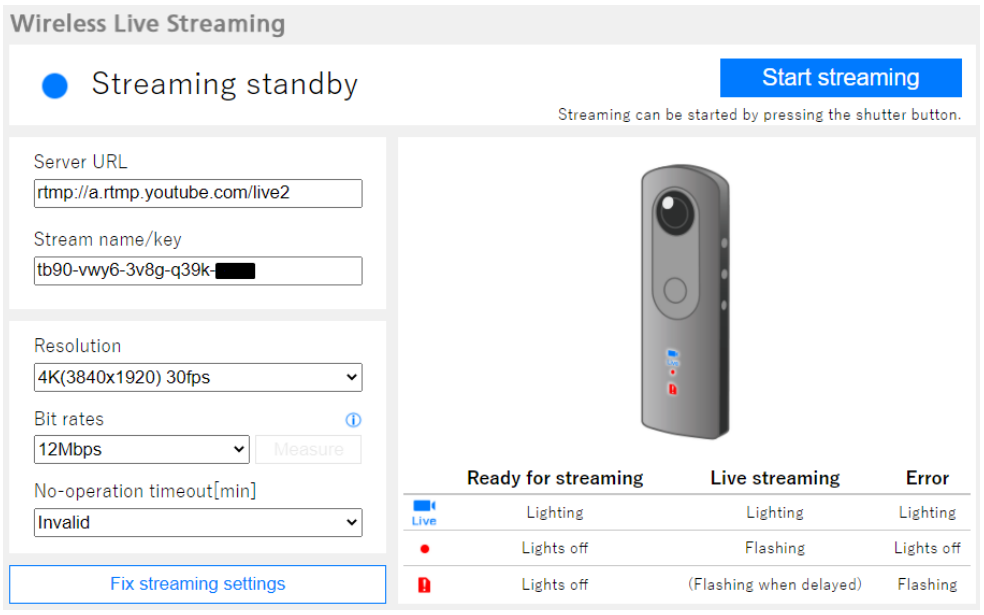
Stream latency
Normal latency
Low-latency
[✓ Ultra low-latency]
Does not support: Closed captions, 1440p, and 4k resolutions

Additional settings
Enable DVR: on
360° video: off
Added delay: None
Closed captions: off
Unlist live replay once stream ends: on

Figure 37: OBS and YouTube livestream side by side

3.4.2: Wireless Live Streaming Plug-in

Wireless Live Streaming Plug-in (here tested with v1.1.1), is an official Ricoh Plug-in and supports livestreaming to an RTMP server. THETA V Wi-Fi should function on CL mode for this implementation because access to the internet is mandatory to reach YouTube server. After activating the *Wireless Live Streaming Plug-in,* a webserver is exposed on the cameras IP, on port 8888. Within this page there are again the same setting fields as in OBS, the *Server URL* field and the *Stream key* field, to be filled with YouTube information. There are also configuration options for *Resolution*, *Bit rates* and *No-operation timeout*, for turning off the camera. After filling in the fields, *Fix streaming settings* button must be pressed to send the configurations to THETA V. Final step is pressing either *Start streaming* button or the camera's shutter button.


Wireless Live Streaming

● Streaming standby

Start streaming

Streaming can be started by pressing the shutter button.

Server URL: rtmp://a.rtmp.youtube.com/live2

Stream name/key: tb90-vwy6-3v8g-q39k-[redacted]

Resolution: 4K(3840x1920) 30fps

Bit rates: 12Mbps

Measure

No-operation timeout[min]: Invalid

Fix streaming settings

	Ready for streaming	Live streaming	Error
Live	Lighting	Lighting	Lighting
●	Lights off	Flashing	Lights off
🟥	Lights off	(Flashing when delayed)	Flashing

Figure 38: Wireless Live Streaming Plug-in default dashboard

3.4.3: VR features

Once the livestream is up, it can be viewed via PC on the YouTube platform, or via smartphone on the YouTube application. The former supports spherical navigation with a mouse and VR navigation if SteamVR is open and HTC Vive connected to the PC. The latter support spherical touchscreen and gyroscope navigation as well as VR navigation with dual screen view to function within VR Box.

3.4.4: Conclusion

Livestreaming from OBS is more reliable, with a stable USB link offering better quality video and with a wired Ethernet link offering higher bit rates. But its main disadvantage is requiring operating from PC in addition to having all those cables limiting the camera’s movement. Wireless Live Streaming Plug-in on the other hand solves these problems by operating wirelessly and autonomously within the already operating camera, thus allowing the camera to move freely within the Wi-Fi coverage area. This makes the Plug-in a very convenient and ready-to-go solution for applications regarding teleoperating vehicles.

3.5: Unity livestreaming with webcam

Unity was used with version 2019.4.6f1 on Windows 10. For VR functionality SteamVR Plug-in for Unity was selected for its simplicity. The specific version of camera driver tested was RICOH THETA UVC v1.0.2 because the earliest v2.0.0 supports H.264 compression instead of MJPEG, resulting in the error "Could not connect pins - RenderStream()". This version of UVC offers "RICOH THETA V 4K" and "RICOH THETA V FullHD" devices.

The concept is to receive the camera's equirectangular livestream feed from wired USB connection and wrap it inside a sphere, resulting to a spherical projection. There are two different Unity features which can be utilized for that application - a sphere with inverted normals (or inversed sphere) and the environment's skybox. Common start for both techniques is to identify THETA V inside if the Unity environment, assign its livestream feed on a 2D texture object and the project this texture on a material component.

In the *UnityEngine* namespace there is *WebCamTexture* class, whose objects are textures onto which live video input is rendered. The *WebCamTexture* constructor's parameters are *deviceName*, *requestedWidth*, *requestedHeight* and optionally *requestedFPS* (for max frame rate). After construction, the *Play()* method must be called to start the camera livestream and render the feed onto the texture. After that, the texture must be assigned to the material object's *mainTexture* property. Below is the *getRicohStream* script used to implement this functionality.

```
using UnityEngine;

public class getRicohStream : MonoBehaviour
{

    static WebCamTexture ricohStream;  // Equirectangular feed texture
    public string camName = "RICOH THETA V 4K"; // Name of device.
    public Material camMaterial;  // Skybox material

    void Start()
    {
        if (ricohStream == null)
            ricohStream = new WebCamTexture(camName, 3840, 1920);

        if (!ricohStream.isPlaying)
            ricohStream.Play();

        if (camMaterial != null)
            camMaterial.mainTexture = ricohStream;
    }
}
```

3.5.1: Inverse sphere technique

In Unity, objects appear to be 3D but, in reality, they are “hollow”, meaning that they are only rendered on the exterior side. This happens for performance issues. This exterior side is not objective, but it depends on the normals of an object. Normal is defined as the vector perpendicular to the surface tangent at a point on the surface of an object. In simple words, the direction of an object’s normals defines the way this object will be rendered. In this scenario, the camera must be in the center of the spherical livestream, the livestream must render on the inside of the sphere, so the normals must be flipped.

For this task, a 3D Computer Graphics software was needed and the selected one was Blender, because Unity natively imports Blender files. Blender version 2.72 was used for this implementation, although version 2.79b is demonstrated in this thesis.

The sphere to be created on an empty scene must have an Icosphere mesh with 6 subdivisions and with the “Generate UVs” option checked. Icopheres are polyhedral spheres made up of triangular faces structured to appear like a sphere. Subdivisions determine the vertices, thus the triangles, used to define the icosphere. At level 1 the icosphere is an icosahedron, a solid with 20 equilateral triangular faces. Each increase in the number of subdivisions splits each triangular face into four triangles. Configuring 6 subdivisions results to 20.480 triangles, a good enough number to maintain high quality on the texture and avoid high performance rendering. Generating UVs is needed for adding texture coordinates in curve objects. Then in edit mode, by pressing the "W" key, "Flip Normals" option comes into view. From there, saving the *.blender* file inside the Unity Project's *Assets* folder, lets the Unity environment interpret it as a model (*inside-sphere*) to be used the *Hierarchy* prefab (*Inverse_sphere*).

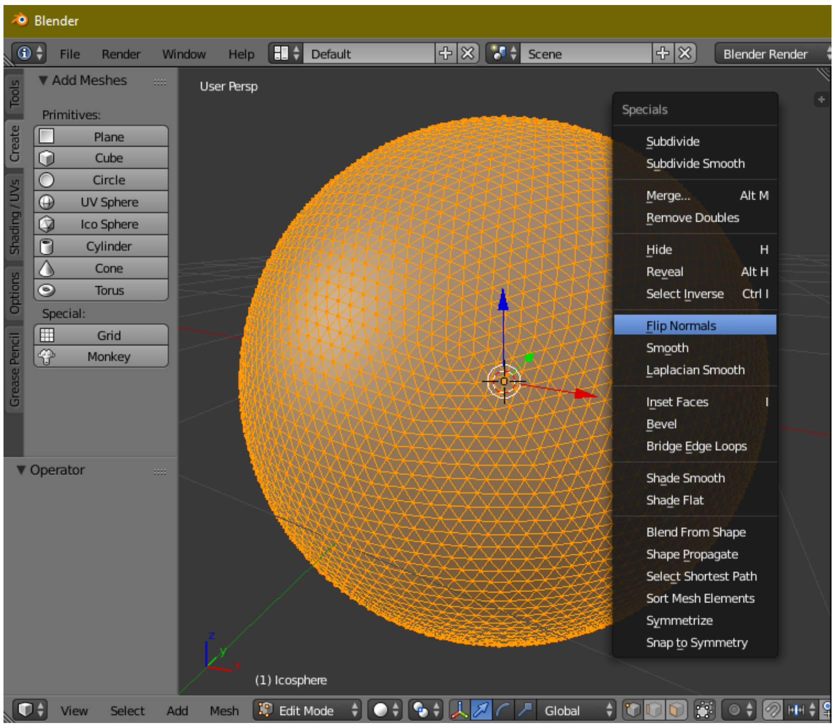
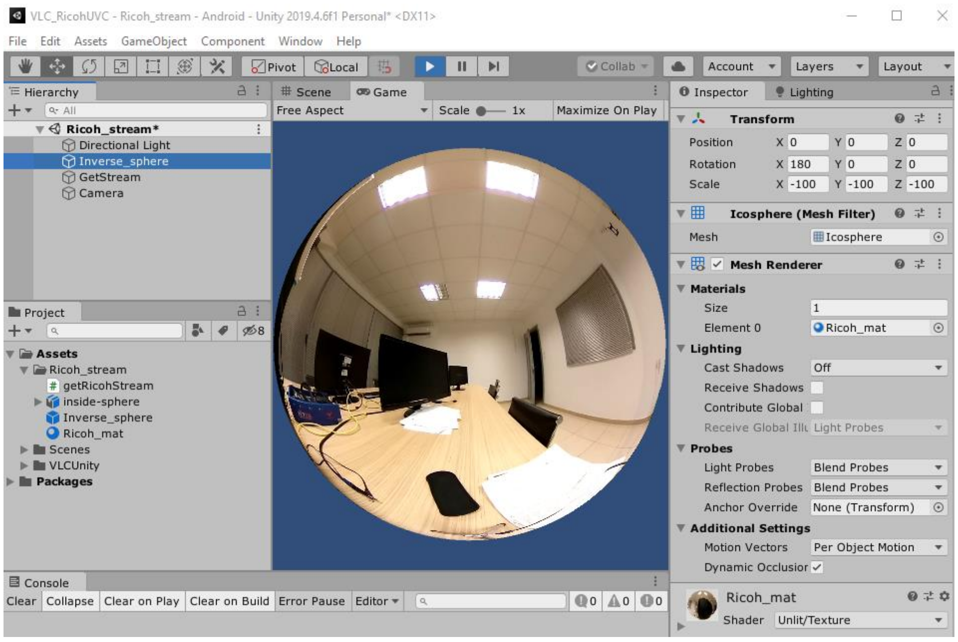


Figure 39: Creating an inverse sphere in Blender 2.79b environment

Next step is creating a Material object (*Ricoh_mat*) in the Unity Project's *Assets* folder and in the *Inspector* window selecting *Shader -> Unlit -> Texture*. *Ricoh_mat* must be assigned to the sphere's *Mesh Renderer* component, in the *Materials* field.

From there, final action is creating in the *Hierarchy* window an empty object (*GetStream)* and assigning to it a *getRicohStream* Script component with the *Ricoh_mat* dragged in the *Cam Material* field.

After all this process, image inside the sphere needs some modifying. Firstly it needs a positive scaling (up to 100 in each axis) for it to be appearing distant. If image is mirrored after flipping the normals, then it needs a negative sign in front of the scale values and maybe a rotation of 180° on the X-axis.



VLC_RicohUVC - Ricoh_stream - Android - Unity 2019.4.6f1 Personal* <DX11>

Hierarchy: Ricoh_stream* — Directional Light, Inverse_sphere, GetStream, Camera

Project: Assets — Ricoh_stream (getRicohStream, inside-sphere, Inverse_sphere, Ricoh_mat), Scenes, VLCUnity; Packages

Inspector:

Transform

	X	Y	Z
Position	0	0	0
Rotation	180	0	0
Scale	-100	-100	-100

Icosphere (Mesh Filter): Mesh — Icosphere

Mesh Renderer

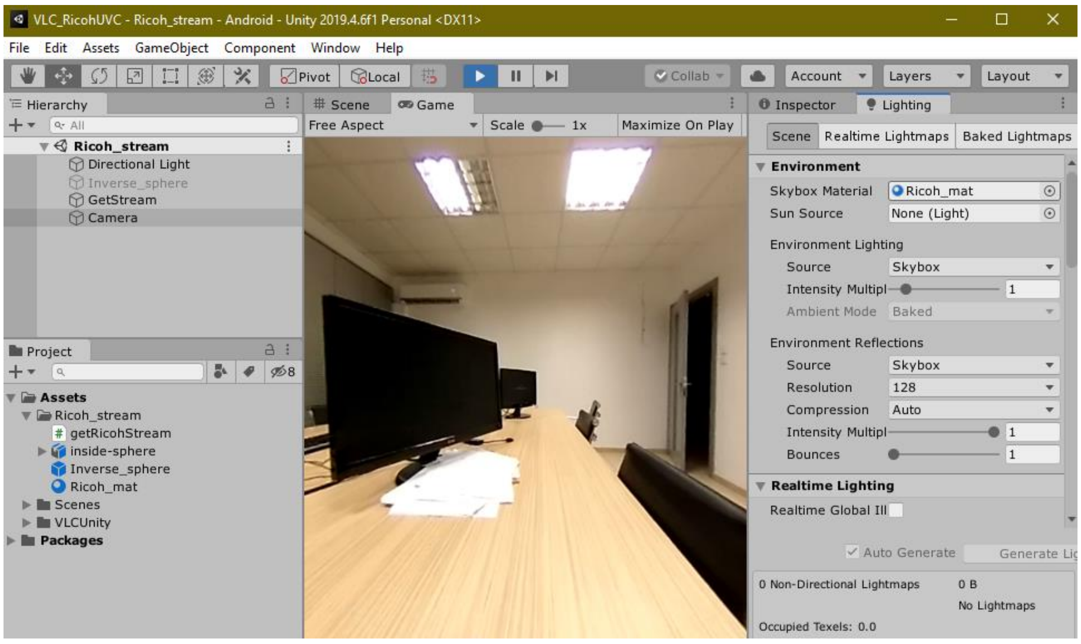
Materials	
Size	1
Element 0	Ricoh_mat
Lighting	
Cast Shadows	Off
Receive Shadows	
Contribute Global	
Receive Global Illu	Light Probes
Probes	
Light Probes	Blend Probes
Reflection Probes	Blend Probes
Anchor Override	None (Transform)
Additional Settings	
Motion Vectors	Per Object Motion
Dynamic Occlusion	✓

Ricoh_mat — Shader: Unlit/Texture

Figure 40: Inverse sphere technique setup in Unity

3.5.2: Skybox technique

After describing the above technique in detail, this Skybox technique is fairly similar. Skybox [4] is in fact a cube map projection wrapped as a sphere to deceive the eye. So this time, *Ricoh_mat* must be set as *Shader -> Skybox -> Panoramic*, because the texture provided will be equirectangular (panoramic). It is already rendered from inside out so all that remains is assigning *Ricoh_mat* in the *Lighting* window *-> Environment -> Skybox Material* and having *GetStream* object running the *getRicohStream* script.



VLC_RicohUVC - Ricoh_stream - Android - Unity 2019.4.6f1 Personal <DX11>

Hierarchy: Ricoh_stream — Directional Light, Inverse_sphere, GetStream, Camera

Project: Assets — Ricoh_stream (getRicohStream, inside-sphere, Inverse_sphere, Ricoh_mat), Scenes, VLCUnity; Packages

Lighting — Scene:

Environment	
Skybox Material	Ricoh_mat
Sun Source	None (Light)
Environment Lighting	
Source	Skybox
Intensity Multipl	1
Ambient Mode	Baked
Environment Reflections	
Source	Skybox
Resolution	128
Compression	Auto
Intensity Multipl	1
Bounces	1
Realtime Lighting	
Realtime Global Ill	

0 Non-Directional Lightmaps 0 B
No Lightmaps
Occupied Texels: 0.0

Figure 41: Skybox technique setup in Unity

3.5.3: Conclusion

Both techniques work great with high-quality livestream video, minimal latency due to wired USB connection, which has a benefit of keeping the THETA V fully charged. A minor issue is that in Inverse Sphere technique, player should be standing still. If he is to walk away from the starting point, he is basically walking towards the “wall” of the sphere’s interior, resulting to spoiling livestream image. Skybox technique is usually preferred because of two reasons: 1) not having to deal with mirroring issues and 2) it allows the player of walking freely by staying always in the far distance of the environment.

3.6: Unity livestreaming with MJPEG decoding

This implementation has many similarities with the previous one in chapter 3.5 and even more with the following one in chapter 3.7. Once more, Unity version 2019.4.6f1 is used with SteamVR Plug-in for Unity, this time working alongside THETA Web API and the camera working on Wi-Fi CL mode. Both projection techniques described above can be used, although Skybox technique is preferred for its simplicity.

3.6.1: THETA Web API functionality

Web API was slightly described in chapter 3.1 and here will be the main subject. Basically, Web API is the device's built-in HTTP/1.1 server on port 80, responding to GET and POST requests. Port 80 default HTTP port, so it can be skipped when writing a URL for Web API. Along the few endpoints provided, the one responsible for configuring and controlling the camera is */osc/commands/execute*, which is responding to POST requests with JSON content. This is why in the request headers, the *Content-Type* key needs to have a value of either "application/json;charset=utf-8" or simpler "application/json". The final URL for web requests to THETA Web API would be

```
http://10.64.45.228/osc/commands/execute
```

Regarding the specific commands needed in this implementation, these are *camera.getLivePreview* and *camera.setOptions*. The *camera.getLivePreview* command is as simple as it appears requiring only the request body to contain below JSON:

```
{
    "name": "camera.getLivePreview"
}
```

The server responds with an MJPEG stream in the form of binary data. This binary data is transferred with the headers:

```
Connection: Keep-Alive
Content-Type: multipart/x-mixed-replace;boundary="---osclivepreview---"
X-Content-Type-Options: nosniff
Transfer-Enconding: Chunked
```

These headers will be explained separately in chapter 3.7, but briefly, they serve the purpose of keeping the HTTP (operating over TCP) connection alive so that the server keeps responding in the same request, without the client resending it. Responses are chunks of data, each one depending on the previous chunk to make sense on the receiving side. And because these chunks have not constant size and are not interpreted separately, the "boundary" string indicates when a new chunk is beginning- thus when the previous one has ended.

The *camera.setOptions* command is for device configuration and has to be send along the *options* parameter. From the many options provided from OSC and Ricoh, the one needed to alter the MJPEG live preview quality is *previewFormat*, using object with *framerate, height,* and *width* fields. The resolutions available are 640x320, 1024x512 and 1920x960, in height-by-width format. Frame rate can be set to 8fps for all possible resolutions and to 30fps only for 640x320 and 1024x512. Below is an example of the request body for setting MJPEG live preview:

```
{
    "name": "camera.setOptions",
    "parameters": {
        "options": {
            "previewFormat": {
                "framerate": 30,
                "height": 512,
                "width": 1024
            }
        }
    }
}
```

The Web API requires the incoming requests to have an *Authorization* field of type *Digest Auth*. This means that it needs authentication credentials (username and password) or else it will not respond or even check the request body. Under the THETA V, there is a code, eg. "YL12345678". The username required for the Web API is "THETAYL12345678" and the password is the numeric side of the username "12345678".

3.6.2: MJPEG decoding in C#

For the Unity and code side of this implementation, just like chapter 3.5, the livestreaming should load on a 2D texture which will be rendered on the panoramic material assigned to the Skybox. Only this time, there is no ready texture. The program should send a request to Web API and keep receiving every response while decoding the chunks of data to construct separate JPEG images. A single script component will be assigned on an empty object in Unity hierarchy.

Starting with the namespaces used, System.IO has major part in response streaming and System.Net in the HTTP communication.

```
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Net;
using UnityEngine;
```

The main functionality of the script is within a coroutine named *GetLivePreview* which is called in the *Start* function.

```
private void Start()
{
    StartCoroutine(GetLivePreview());
}
```

Inside *GetLivePreview* coroutine, the *Digest Auth* authorization is a priority.

```
var credentialCache = new CredentialCache();
credentialCache.Add(
    new System.Uri("http://10.64.45.228/osc/commands/execute"),
    "Digest",
    new NetworkCredential("THETAYL12345678", "12345678")
);
```

Two separate requests will be sent, one with *camera.setOptions* and one with *camera.getLivePreview* command, thus creating two separate byte arrays to be the requests' bodies.

```
byte[] previewBody = Encoding.Default.GetBytes("{\"name\" :
\"camera.getLivePreview\"}");
byte[] formatBody = Encoding.Default.GetBytes("{"+
                                  "\"name\":\"camera.setOptions\","+
                                  "\"parameters\":{"+
                                  "\"options\":{"+
                                  "\"previewFormat\":{"+
                                  "\"framerate\":30,"+
                                  "\"height\":512,"+
                                  "\"width\":1024"+"}}}}");
```

For the Web API communication, the System.Net class *HttpWebRequest* was used which makes a request to a Uniform Resource Identifier (URI) or in this case the *http://10.64.45.228/osc/commands/execute* URL. It is an old and reliable class and much simpler than the other networking features which Unity provides.

```
var request =
HttpWebRequest.Create("http://10.64.45.228/osc/commands/execute");
request.PreAuthenticate = true;
request.Credentials = credentialCache;
request.Timeout = (int)(30 * 10000f);
request.Method = "POST";
request.ContentType = "application/json;charset=utf-8";
request.ContentLength = previewBody.Length;
ServicePointManager.Expect100Continue = false;
Stream reqStream = request.GetRequestStream();
reqStream.Write(previewBody, 0, previewBody.Length);
```

Now, regarding the response, the initial steps is to receive the response byte stream and be able to read it from binary to ASCII characters, using System.IO classes, *Stream, BinaryReader* and *BufferedStream*.

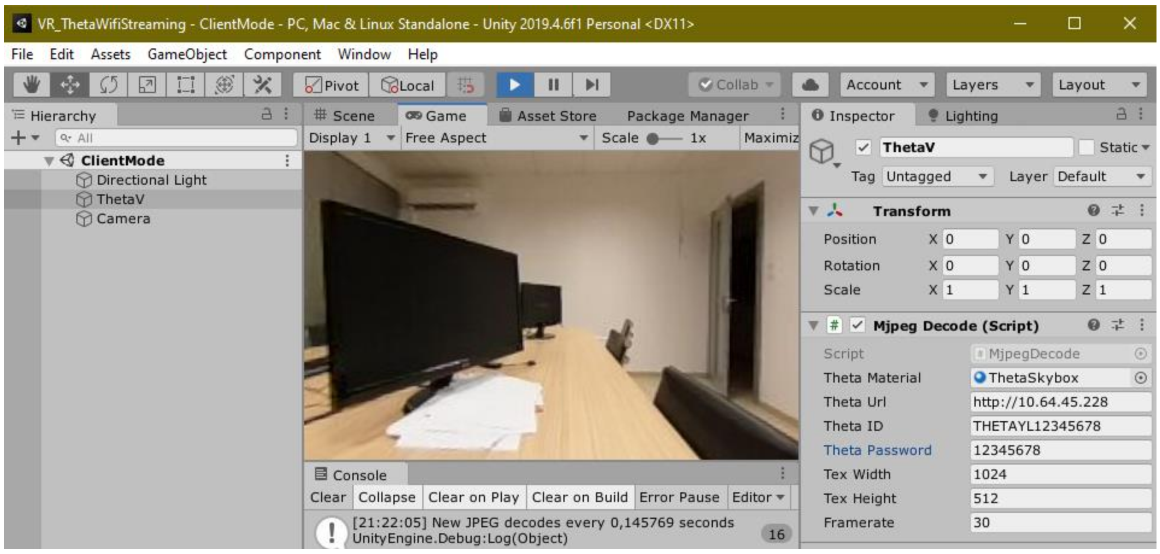
```
using (var stream = request.GetResponse().GetResponseStream())
{
    BinaryReader reader = new BinaryReader(new
BufferedStream(stream), new ASCIIEncoding());
    {
    ...
    }
}
```

For the missing part of the snippet, is the main functionality of this program, the MJPEG decoding. An MJPEG streams consists of many consecutive JPEG images. JPEG images start with a header, the hexadecimal values “FF D8”, or “0xFF 0xD8” in binary. No file length is embedded in a JPEG image, so there is also a trailer, the hexadecimal values “FF D9”, or “0xFF 0xD9” in binary. Now this searching method must be done continually and endlessly as the livestreaming continues and Web API keeps sending data.

Concerning code, the task requires a data structure for dynamic insertion and removal of bytes - a byte list. Every two successional bytes received are checked for being the JPEG header, signifying the start of the image. Previously checked bytes are discarded and the following ones are added in the list unchecked. They are checked though after their insertion in searching of the JPEG trailer. Once the trailer pair of bytes is found, the JPEG image is constructed and loaded on the skybox material’s texture. The procedure has come to an end and loops back.

```
List<byte> imageBytes = new List<byte>();
bool isLoadStart = false;
while (true)
{
    byte byteData1 = reader.ReadByte();
    byte byteData2 = reader.ReadByte();
    if (!isLoadStart)
    {
        if (byteData1 == 0xFF && byteData2 == 0xD8)
        {
            imageBytes.Add(byteData1);
            imageBytes.Add(byteData2);
            isLoadStart = true;
        }
    }
    else
    {
        imageBytes.Add(byteData1);
        imageBytes.Add(byteData2);
        if (byteData1 == 0xFF && byteData2 == 0xD9)
        {
            tex = new Texture2D(texWidth,texHeight);
            tex.LoadImage((byte[])imageBytes.ToArray());
            thetaMaterial.mainTexture = tex;
            imageBytes.Clear();
            yield return null;
            isLoadStart = false;
        }
    }
}
```

The *MjpegDecode.cs* script needs to be assigned to an empty *ThetaV* object in order to run. Some public variables can be added for easy playtesting settings.


VR_ThetaWifiStreaming - ClientMode - PC, Mac & Linux Standalone - Unity 2019.4.6f1 Personal <DX11>
File Edit Assets GameObject Component Window Help
Pivot Local Collab Account Layers Layout
Hierarchy Scene Game Asset Store Package Manager Inspector Lighting
Display 1 Free Aspect Scale 1x Maximiz
ClientMode
Directional Light
ThetaV
Camera
Console
Clear Collapse Clear on Play Clear on Build Error Pause Editor

```
ThetaV                      Static
Tag Untagged   Layer Default
Transform
Position   X 0   Y 0   Z 0
Rotation   X 0   Y 0   Z 0
Scale      X 1   Y 1   Z 1
Mjpeg Decode (Script)
Script            MjpegDecode
Theta Material    ThetaSkybox
Theta Url         http://10.64.45.228
Theta ID          THETAYL12345678
Theta Password    12345678
Tex Width         1024
Tex Height        512
Framerate         30
```

```
[21:22:05] New JPEG decodes every 0,145769 seconds   16
UnityEngine.Debug:Log(Object)
```

Figure 42: Unity Livestreaming with MJPEG decoding

3.6.3: Conclusion

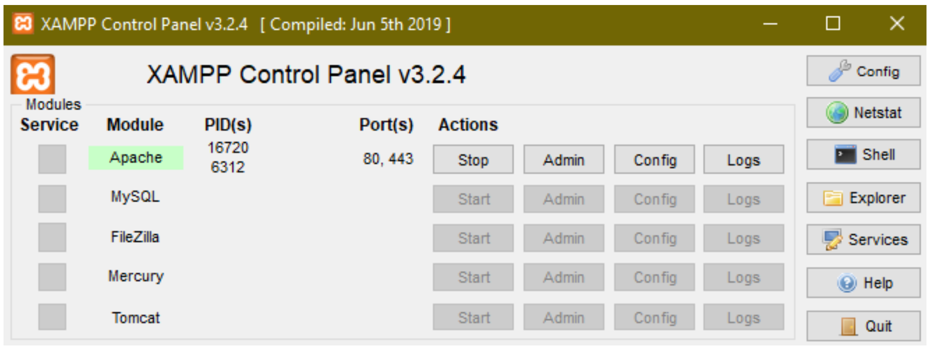
The application was tested and works perfectly over WLAN and allows the camera to move freely. The result is decent in terms of quality and latency, because the decoding time of each JPEG is nearly 135ms, not affecting the small latency of MJPEG streaming. Implementing livestreaming in Unity with MJPEG decoding unravels a whole lot of possibilities in VR livestreaming because of the SteamVR plugin capabilities. Yet, the Unity platform is a restrictive tool if the livestreaming needs to be exposed in public and for general use, thus inspiring the next implementation.

3.7: Browser livestreaming with Reverse Proxy Server

Following the steps of the previous chapter, the initial idea was to convert the Unity MJPEG decoding application into a browser application, which if made public it would be accessible to everyone. This means that that the frontend application should be served over a web server, like Apache HTTP Server, and have VR functionality, which can be provided by WebXR Device API alongside Unity WebGL or by A-Frame web framework. Down the road it became clear that the implementation is complex and would be limited by the camera's simple HTTP server. That is why it was necessary to extend its capabilities by deploying a reverse proxy server.

3.7.1: Apache HTTP Server

Both WebXR and A-Frame require the content to be served over HTTPS, because of the Secure Sockets Layer (SSL) protocol. Simple development servers would not do the trick if they do not support SSL certificates. Apache HTTP Server is a free and open-source web server and can be easily set up by installing the XAMPP stack package. In this thesis, XAMPP was installed with version 8.0.3 on Windows 10 and the XAMPP Control Panel v3.2.4.


XAMPP Control Panel v3.2.4 [Compiled: Jun 5th 2019]
XAMPP Control Panel v3.2.4
Modules
Config
Netstat
Shell
Explorer
Services
Help
Quit

Service	Module	PID(s)	Port(s)	Actions			
	Apache	16720 6312	80, 443	Stop	Admin	Config	Logs
	MySQL			Start	Admin	Config	Logs
	FileZilla			Start	Admin	Config	Logs
	Mercury			Start	Admin	Config	Logs
	Tomcat			Start	Admin	Config	Logs

Figure 43: XAMPP Control Panel for Apache control

After the installation, any build folder, e.g. *browserLivestreaming*, must be located inside *C:\xampp\htdocs*. When Apache starts, the built application can be reached on a browser through the URL *https://10.64.45.161/browserLivestreaming/index.html*.

3.7.2: Reverse Proxy Server

Reverse proxy [5], in a cloud architecture, is a server that sits on the backend side of applications to accomplish load balancing, fault tolerance and auto scaling, smoothly and efficiently. It is responsible for receiving all the requests from the clients and redirecting them to the many servers of the system. In this case, the reverse proxy redirects the requests only to Web API and extends it by implementing additional functionality, like serving over HTTPS, responding to OPTIONS requests or caching [6] the MJPEG stream as separate JPEGs.

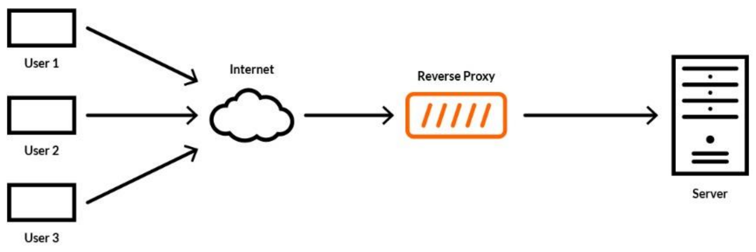


Figure 44: Reverse Proxy server diagram

The initial need for a reverse proxy server was CORS policy. Cross-Origin Resource Sharing (CORS) is a mechanism that allows a server, in this case Apache, to indicate any resources about to be loaded on browser and whose origins are not the same with its own. Browsers make a "pre-flight" request as a safety measure to ensure that the request being done is trusted by the server hosting the cross-origin resource. This pre-flight is an OPTIONS method request which it indicates the HTTP method and the headers that will be used in the actual request. The foreign origin server will check the pre-flight and will either permit the actual request or not.

Simple requests are those which do not trigger a CORS pre-flight. The conditions for one request to be “simple” and that are relevant to this implementation are the following:

1. Its method must be *GET*, *HEAD* or *POST*.
2. The only headers which are allowed to be manually (not by the user agent) set are *Accept, Accept-Language, Content-Language* and *Content-Type*.
3. *Content-Type* header must be valued as *application/x-www-form-urlencoded, multipart/form-data* or *text/plain*.

Conditions 1 and 2 are met but for the request on Web API for *camera.getLivePreview* command has a *Content-Type: application/json* header. So *simple request* is not a solution in this case.

There are some browser extensions, like *Moesif CORS* and *Allow CORS*, that allow CORS by adding in the server response the following headers if necessary: *Access-Control-Allow-Origin*, *Access-Control-Allow-Methods, Access-Control-Allow-Headers* and *Access-Control-Allow-Credentials*. With these extensions, an OPTIONS pre-flight request is still triggered, which THETA Web API cannot response to.

Finally, the only acceptable solution would be the reverse proxy server responding to *OPTIONS* method requests and at the same time adding the required CORS response headers.

For the reverse proxy server application, *Flask* was chosen, a micro web framework written in *Python*. It was installed as a python module with *pip* command, not natively as a bash script. It was installed in version *0.10.1* alongside *Jinja2 2.11.3* and *Werkzeug 1.0.1* as the Web Server Gateway Interface (WSGI) and is running with *Python 3.7.3*. The device running the *Flask* application is the RPi used as a bridged AP in chapter 3.2, thus the reverse proxy's IP is *10.64.44.242* , the same as RPi's IP.

The *FlaskProxy.py* script is enough to run a web server, on port 5000 using HTTP, just by writing the below code:

```
from flask import Flask

app = Flask(__name__)
THETA_IP = 'http://10.64.45.228'
default_headers = [
    ('Connection','Keep-Alive'),
    ('X-Content-Type-Options','nosniff')
    ]

@app.route('/')
def index():
    return 'Flask is running ' + request.environ.get('SERVER_PROTOCOL')

if __name__ == '__main__':
    app.run(host="0.0.0.0", debug=True)
```

Flask needs to work over HTTPS for WebXR and A-Frame to work. That can be achieved with SSL certificates. Usually the certificates acts as identification for the server, as it includes the server's name and domain. To ensure that the information provided by the server is correct, the certificate is cryptographically signed by a certificate authority. is one where the signature is generated using the private key that is associated with that same certificate. After the client verifies the certificate, it creates an encryption key to use for the communication with the server. To make sure that this key is sent securely to the server, it encrypts it using a public key that is included with the server certificate. The server is in possession of the private key that goes with that public key in the certificate, so it is the only party that is able to decrypt the package. From the point when the server receives the encryption key all traffic is encrypted with this key that only the client and server know.

Not having access to a validated certificate authority, the same task can be fulfilled with a self-signed certificate. With *OpenSSL* toolkit installed with version *1.1.1d*, the following command must run on the same folder as *FlaskProxy.py*:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout local.key -out local.crt
```

And __*main*__ function of the python program will be:

```
if __name__ == '__main__':
    context = ('local.crt', 'local.key')
    app.run(host="0.0.0.0", debug=True, ssl_context=context)
```

Now the Flask server is running on *https://10.64.44.242:5000/* and the reverse proxy functionality must be added. *Flask* supports route handling for fixed paths, with constant URLs, or dynamic route handling where the path can have any value. Basically the *path* parameter in the handler URL can be any string. And when the server starts handling the URL following the pattern, this string is interpreted as a string variable. In simple words, when */osc/commands/execute* is reached, the *path* variable has a value of '*osc/commands/execute*'.

```
from flask import Response, request
import requests
from requests.auth import HTTPDigestAuth

@app.route('/<path:path>',methods=['GET','POST','OPTIONS'])
def proxy(path):
  if request.method=='GET':
    resp = requests.get(THETA_IP+'/'+path,
                        auth=HTTPDigestAuth('THETAYL12345678',
                                            '12345678')
                        )

    headers = [(name, value) for (name, value)
              in resp.raw.headers.items()]
    response = Response(resp.content, resp.status_code, headers)
```

The above code handles the *GET* method requests. Client does not need to send authorization header to the proxy since the proxy itself uses the *HTTPDigestAuth* function for authentication with the camera. As soon as the response is back from THETA Web API server, a new *Response* is constructed with the same content, status code and headers as the real response which THETA sent to the reverse proxy server.

```
from flask import stream_with_context

@app.route('/<path:path>',methods=['GET','POST','OPTIONS'])
def proxy(path):
  ...
  elif request.method=='POST':
    if request.get_json() is not None:
      # LIVE PREVIEW COMMAND
      if request.get_json()["name"] == "camera.getLivePreview":
          resp = requests.post(THETA_IP+'/'+path,
                               stream=True,
                               json=request.get_json,
                               auth=HTTPDigestAuth('THETAYL12345678',
                                                   '12345678')
                               )

          headers = [(name, value) for (name, value)
                    in resp.raw.headers.items()]
          response = Response(
              stream_with_context(resp.iter_content(chunk_size=None)),
              resp.status_code,
              headers
          )
      # REGULAR POST
      else:
          resp = requests.post(THETA_IP+'/'+path,
                               json=request.get_json(),
                               auth=HTTPDigestAuth('THETAYL12345678',
                                                   '12345678')
                               )
          headers = [(name, value) for (name, value)
                     in resp.raw.headers.items()

          response = Response(resp.content, resp.status_code, headers)
```

About handling regular *POST* method requests, the process is the same with the slight difference of redirecting the client request's JSON body to the Web API server. Other than that, the complicate part is when the reverse proxy needs to handle a streaming response.

The *Response.iter_content* function is basically a generator which iterates over the response body data and does not read the content at once into memory. *Chunk_size* is to *None* so that the program reads the data in whatever size of chunks they are received, not by a predefined size that could delay the streaming speed. The *flask.stream_with_context* helper wraps around this generator and keeps sending the generated chunks back to the client as individual bodies but keeping the same HTTP connection with him, thus sending only on response that keeps adding to its body.

```
from flask import make_response

@app.route('/<path:path>',methods=['GET','POST','OPTIONS'])
def proxy(path):
  ...
  elif request.method=='OPTIONS':
          response = make_response()
```

OPTIONS pre-flight requests must be handled, even with an empty response from *flask.make_response* function. After all, the *OPTIONS* responses are not concerning the client application.

For the final part of the reverse proxy basic functionality, in every *response* constructed in whatever *if else branch*, CORS policy headers are added and then the response is returned to the client. The whole *route handler* would look like this:

```
@app.route('/<path:path>',methods=['GET','POST','OPTIONS'])
def proxy(path):
  if request.method=='GET':
    ...
  elif request.method=='POST':
    ...
  elif request.method=='OPTIONS':
    ...

  response.headers.add("Access-Control-Allow-Origin", "*")
  response.headers.add('Access-Control-Allow-Headers', "*")
  response.headers.add('Access-Control-Allow-Methods', "*")
  return response
```

Having developed the reverse proxy server this way, the previous implementation in chapter 3.6 works flawlessly, because the MJPEG stream is redirected to Unity exactly as it was before. Some additional feature will be added for the next chapter concerning WebXR.

3.7.3: WebXR in Unity WebGL

Unity allows building a project as a web application, using tools as HTML5/JavaScript, WebAssembly, WebGL rendering API and other web standards which are all supported by most major browsers like Google Chrome and Mozilla Firefox. Although, WebGL builds have some limitations due to constrains of the platform. The three limitations, which will be explained further below, are the networking specifications of WebGL, the lack of threading support in JavaScript and the memory allocations of Unity Heap.

For the VR features in the web application, Unity WebGL builds support WebXR JavaScript API which is integrated via the *Mozilla WebXR Exporter* asset. The compatible Unity versions are *2019.4.7* and up or *2020.1* and up, so in this implementation *2019.4.24f1* is used.

- *UnityWebRequest* features and architecture

According to *WebGL Networking* page, .NET networking classes, like *HttpWebRequest*, cannot be used because as mentioned before, JavaScript code does not have direct access to IP sockets directly, due to the browsers' forbidding it. Though, it supports the *UnityWebRequest* (UWR) class, which is based on the *XMLWebRequest,* by using the JavaScript Fetch API. By default, this imposes security restrictions with CORS resources, but the reverse proxy server takes care of it already.

UWR does not a have a *Digest Auth* feature. This should not be a problem as soon as the proxy acts as a middleman and do the authentication for the client. But for direct communication with the THETA Web API, the digest authentication must be disabled. THETA V must operate in AP mode. The client device must send a *POST* request on *http://192.168.1.1/osc/commands/execute* with the below JSON contents:

```
{
    "name": "camera.setOptions",
    "parameters": {
        "options": {
            "_authentication": "none"
        }
    }
}
```

For the C# script used in the Unity project, the procedure starts with setting the resolution and frame and the proceeds to request the stream. Although this time, UWR class has some impact on the code because of its architecture. It breaks down to three distinct operations: 1) supplying data to server, 2) receiving data from server and 3) HTTP flow control. The first two are managed by an *UploadHandler* object and a *DownloadHandler* object respectively. These two objects and the HTTP flow control are managed by the UWR object itself. Their classes can also be used to implement derived classes which will have custom functionality based on the user’s needs.

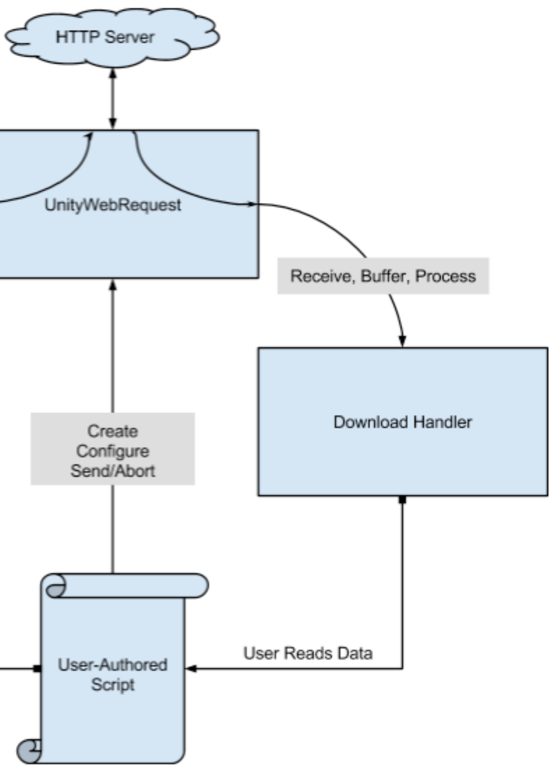


Figure 45: UntiyWebRequest architecture

For usage over HTTPS (like WebXR requires), there is another handler class called *CertificateHandler*, which is responsible for validating SSL certificates against certificate authorities. However, the reverse proxy server uses a self-signed SSL certificate. Therefore, for this thesis' development usage, a derived class *HttpsCertificateHandler* is implemented from *CertificateHandler*. This class overrides the *ValidateCertificate* method and returns *true* without checking.

```
using UnityEngine;
using UnityEngine.Networking;
public class HttpsCertificateHandler : CertificateHandler
{
    protected override bool ValidateCertificate(byte[] certificateData)
    {
        return true;
    }
}
```

The basic code flow for a simple request, like the *camera.setOptions* request, where *setOptionsBytes* contain the JSON body, can be demonstrated with the following snippet:

```
private UnityWebRequest request;

IEnumerator SetOptions()
{
    request = new UnityWebRequest(url, "POST")
    {
        uploadHandler = new UploadHandlerRaw(setOptionsBytes),
        downloadHandler = new DownloadHandlerBuffer(),
        certificateHandler = new HttpsCertificateHandler()
    };
    request.SetRequestHeader("Content-Type", "application/json");

    yield return request.SendWebRequest();
    while (!request.isDone)
    {
        yield return new WaitForEndOfFrame();
    }

    request.Dispose();
    request = null;
}
```

The *Dispose* method signals that the UWR object is no longer being used and should clean up any resources it is using. These resources include all handler objects used within this UWR.

- Pseudo-MJPEG streaming

WebGL is single threaded while the JavaScript Fetch API is asynchronous. This means that while UWR coroutine handles an HTTP response, it occupies the one and only thread and returns control to the browser after it finishes the download. The *DownloadHandler.ReceiveContentLengthHeader* callback is invoked whenever a *Content-Length* header is received within a server response. With that information the *UnityWebRequest.downloadProgress* returns a floating point between 0.0 and 1.0 indicating the progress of the downloading body data from the server. When that reaches value 1.0, the *UnityWebRequest.isDone* property return true and the coroutine can finally terminate and release the main thread. This is not the case with MJPEG streaming response, because the response from THETA Web API does not contain a *Content-Length* header. The callback is never invoked, expected response length is unknown, progress is constantly on zero and the UWR never terminates to release the main thread.

In the meantime, no image is rendered on the skybox material. That is because the *DownloadHandler.ReceiveData* callback never acquires the WebGL thread. This callback is the one which has to be overridden when received data needs to be tampered with, like in the case of MJPEG decoding.

With that said, it was essential to develop a mechanism in the reverse proxy to break up the MJPEG stream and caching it as individuals JPEG images in a buffer and then sending them as individual responses to individual Unity requests. In that way, the client application can just request rapidly single JPEGs and project them, thus creating a pseudo MJPEG stream.

This is implemented with a thread process constantly “recording” the MJPEG, meaning that it receives the stream, finds the JPEG header and trailer in the way that was described in 3.6.2, and caches the image in a buffer until the client application requests it. To avoid race conditions, a semaphore is used, controlling access to the common buffer between the thread and the main process. The selected data structure for the buffer is queue for inserting and deleting elements in O(1) time. The Python data-type for implementing a queue is a double-ended queue (deque) which supports adding and removing elements from either end, but here used as First-In/First-Out (FIFO) from one end. Caching memory size is limited for performance optimization, so older JPEGs are replaced by newer ones.

```python
import collections
import threading

recThread = None
sem = threading.Semaphore()
buffer = collections.deque([], maxlen=2)

def recordMjpeg(response):
    bytes = b''
    a = -1
    b = -1

    for line in response.iter_content(chunk_size=None):
        bytes += line
        if a == -1:
            a = bytes.find(b'\xff\xd8')

        b = bytes.find(b'\xff\xd9')
        if a != -1 and b != -1:
            jpg = bytes[a:b+2]
            bytes = bytes[b+2:]
            a = -1
            b = -1
            sem.acquire()
            buffer.append(jpg)
            sem.release()


def generateJpeg():
    jpg = None
    while True:
        sem.acquire()
        if len(buffer) > 0:
            jpg = buffer.popleft()
        sem.release()
        if jpg is not None:
            return jpg
```

The *recordMjpeg* function is the target function for the *recThread* and the *generateJpeg* function is called by the main program serving the client.

```python
@app.route('/<path:path>',methods=['GET','POST','OPTIONS'])
def proxy(path):
    global recThread

    if request.method=='POST':
        if request.get_json() is not None:
            if request.get_json()["name"] == "camera.recordMjpeg":
                #Start recordMjpeg thread
                if recThread is None:
                    request.get_json()["name"]="camera.getLivePreview"
                    resp = requests.post(THETA_IP+'/'+path,
                                         stream=True,
                                         json=request.get_json()
                                        )
                    recThread = threading.Thread(target=recordMjpeg,
                                                  args=(resp,),
                                                  daemon=True
                                                 )
                    recThread.start()
                #Reset recordMjpeg thread
                else:
                    sem.acquire()
                    buffer.clear()
                    sem.release()
                response = make_response()

            elif request.get_json()["name"] == "camera.getJpeg":
                response = Response(
                    generateJpeg(),
                    200,
                    default_headers
                )

    response.headers.add("Access-Control-Allow-Origin", "*")
    response.headers.add('Access-Control-Allow-Headers', "*")
    response.headers.add('Access-Control-Allow-Methods', "*")
    return response
```

The two commands, *camera.recordMjpeg* and *camera.getJpeg* are not native THETA Web API commands, but rather exist exclusively for client-proxy communication.

Concerning the Unity side of the system and the C# code, *JpegHandler* is a custom class implemented for this task, derived from *DownloadHandlerScript* (basically a *DownloadHandler* interface). When constructing an object for continuous and long-term use, memory allocation is a constraint. The *DownloadHandlerScript* constructor eliminates memory allocation by permiting the pre-allocation of a managed-code byte array, reusing it to deliver downloaded data to *DownloadHandler.ReceiveData* callback. This byte array limits the amount of data delivered each frame. There are three possible options for the byte array size, provided by the main process: 40000, 80000 and 210000, because of its direct relation with the *jpegLength* variable – the average JPEG size. With empirical estimation during tryouts, the average JPEG size seems to depend on the resolution of the live preview of THETA V: ~35000 bytes for 640x320, ~65000 for 1024x512 and ~210000 for 1920x960.

```
public class JpegHandler : DownloadHandlerScript
{
    private Texture2D tex;
    private Material thetaMaterial;
    private List<byte> dataStream;
    private int jpegLength;

    public JpegHandler(byte[] buffer,
                       Material thetaMaterial,
                       Texture 2D tex,
                       List<byte> dataStream)
        : base(buffer)
    {
        this.tex = tex;
        this.thetaMaterial = thetaMaterial;
        this.dataStream = dataStream;

        switch (buffer.Length)
        {
            case 40000: jpegLength = 35000;
                break;
            case 80000: jpegLength = 65000;
                break;
            case 210000: jpegLength = 200000;
                break;
        }
    }
    ...
    callbacks
    ...
}
```

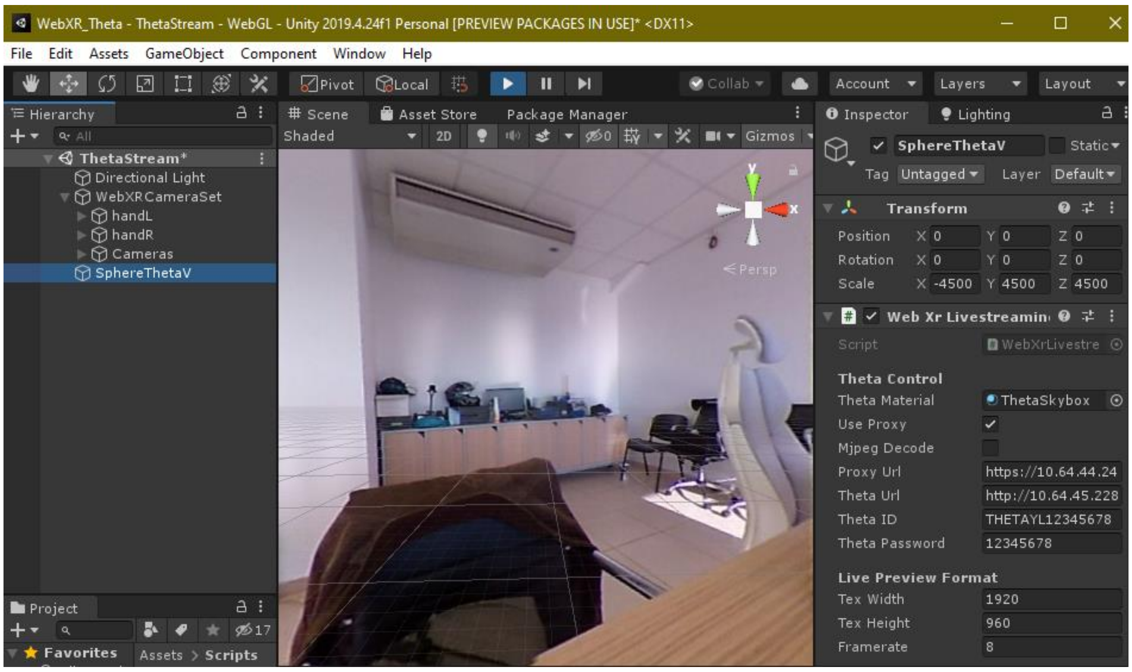
Seeing inside the *ReceiveData* callback of the *JpegHandler,* it operates the same way as in chapter 3.6.2, regarding textures and materials for skybox rendering. Its algorithm has been designed to manage the sometimes chunked-transfer response of the data, thus waiting for the minimum *jpegLength* bytes to be assembled before loading the image on the material.

```
public class JpegHandler : DownloadHandlerScript
{
    ...
    protected override bool ReceiveData(byte[] jpegBytes, int numBytes)
    {
        if (numBytes < 1)
        {
            return false;
        }

        for (int i = 0; i < numBytes; i++)
        {
            dataStream.Add(jpegBytes[i]);
        }
        if (dataStream.Count > jpegLength)
        {
            tex.LoadImage(dataStream.ToArray());
            thetaMaterial.mainTexture = tex;
            dataStream.Clear();
        }

        return true;
    }
}
```

In the main process A coroutine is used again for the repetitive JPEG requests. The *StartRecording* coroutine needs no display, since it resembles the above *SetOptions* coroutine, only this time sending a *POST* request with the *camera.recordMjpeg* command, and later, on server response, it flips the *recordingStarted* variable.

```
IEnumerator RepetitivePreviewRequests()
{
    yield return StartCoroutine(StartRecording());

    byte[] byteBuffer;
    switch (texHeight)
    {
        case 320: byteBuffer = new byte[40000];
            break;
        case 512: byteBuffer = new byte[80000];
            break;
        case 960: byteBuffer = new byte[210000];
            break;
        default: byteBuffer = new byte[80000];
            break;
    }
    byte[] postBytes;
    postBytes = Encoding.Default
                        .GetBytes("{\"name\" :\"camera.getJpeg\"}");
    Texture2D texture = new Texture2D(2, 2);
    List<byte> dataStream = new List<byte>();

    while (true)
    {
        if (!recordingStarted)
        {
            yield break;
        }

        request = new UnityWebRequest(url, "POST")
        {
            uploadHandler = new UploadHandlerRaw(postBytes),
            downloadHandler = new JpegHandler(byteBuffer,
                                               thetaMaterial,
                                               texture
                                               dataStream),
            certificateHandler = new HttpsCertificateHandler()
        };
        request.SetRequestHeader("Content-Type", "application/json");

        while (!request.isDone)
        {
            yield return new WaitForEndOfFrame();
        }

        request.Dispose();
        request = null;

        GC.Collect();
    }
}
```

The last piece of the code not discussed is the *System.GC.Collect* method, which forces an immediate garbage collection of all generations and cleans them from memory. That is because WebGL content will run inside a browser, so any memory has to be allocated by the browser within the browser's memory space.



SphereThetaV Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X -4500 Y 4500 Z 4500

Web Xr Livestreamin
Script WebXrLivestre

Theta Control
Theta Material ThetaSkybox
Use Proxy ✓
Mjpeg Decode
Proxy Url https://10.64.44.24
Theta Url http://10.64.45.228
Theta ID THETAYL12345678
Theta Password 12345678

Live Preview Format
Tex Width 1920
Tex Height 960
Framerate 8

Figure 46:Reverse Proxy Server pseudo-MJPEG livestream in Unity with WebXR

Finally, the end result after building the project in the *WebXR_JPGs* folder of the *xampp/htdocs* is this:

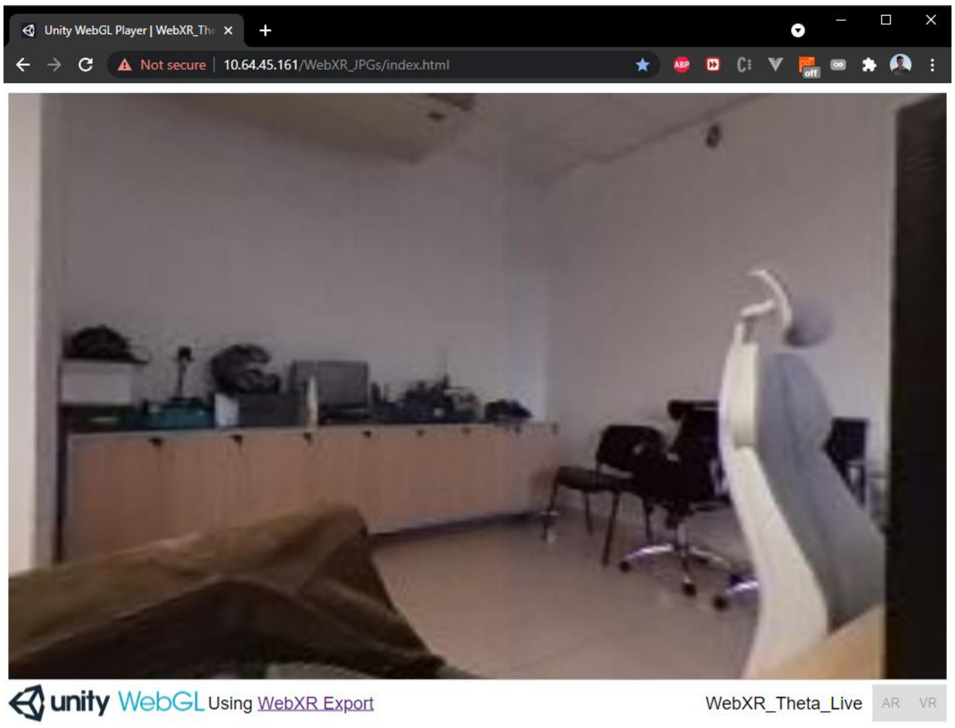

Unity WebGL Player | WebXR_Th
Not secure | 10.64.45.161/WebXR_JPGs/index.html
unity WebGL Using WebXR Export
WebXR_Theta_Live
AR
VR

Figure 47: Livestreaming in browser with Unity WebGL and WebXR

3.7.4: A-Frame

Starting with Three.js, a cross-browser JavaScript library and API used to create and display of GPU accelerated 3D animations and computer graphics in a web browser using WebGL. It uses the JavaScript language as part of a website without relying on proprietary browser plugins. It includes many features and components known from game engines like scenes, cameras, geometries, meshes, materials and so on.

A-Frame is an open-source web framework for building VR experiences, originally developed by Mozilla VR team. It is an entity-component-system (ECS) framework for Three.js where developers can create 3D and WebXR scenes using HTML without having to know WebGL. HTML provides a familiar authoring tool for web developers and designers while incorporating a popular game development pattern used by game engines, such as Unity. Nevertheless, developers have unlimited access to JavaScript, DOM APIs, three.js, WebXR, and WebGL. One of its top advantages, and true to the reason of this implementation, is that it is cross-platform that works on standard desktop, smartphones and is compatible with all major VR headsets.

Initially, the reverse proxy server needs to handle an extra URL, */get_stream*, from which it will expose the MJPEG livestream in equirectangular data while still provide with CORS headers.

```
THETA_IP = 'http://10.64.45.228'
EXECUTE_PATH = '/osc/commands/execute'
LIVE_CMD = '{"name": "camera.getLivePreview"}'

@app.route('/get_stream', methods=['GET'])
def stream():
    print("get stream")
    resp = requests.post(THETA_IP+EXECUTE_PATH,
                         stream=True,
                         json=json.loads(LIVE_CMD))
    headers = [(name, value) for (name, value)
               in resp.raw.headers.items()]

    response = Response(
        stream_with_context(resp.iter_content(chunk_size=None)),
        resp.status_code,
        headers
    )

    response.headers.add("Access-Control-Allow-Origin", "*")
    response.headers.add('Access-Control-Allow-Headers', "*")
    response.headers.add('Access-Control-Allow-Methods', "*")
    return response
```

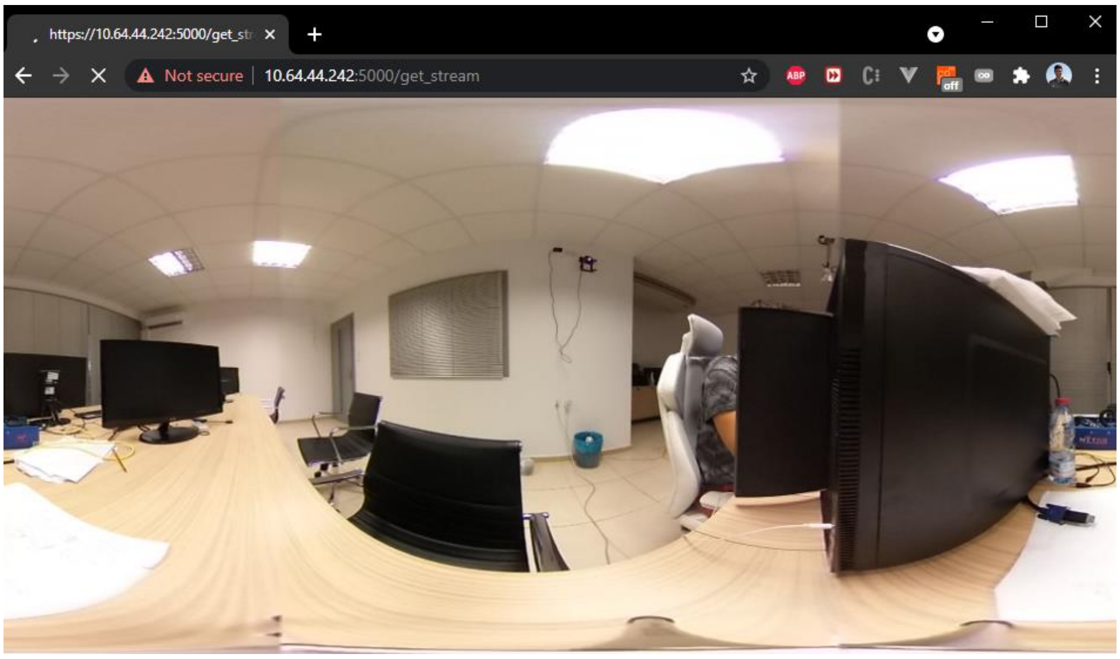

https://10.64.44.242:5000/get_str
Not secure
10.64.44.242:5000/get_stream

Figure 48: Equirectangular livestream from reverse proxy server in browser

Concerning A-Frame, it can be developed from a plain HTML file without having to install anything. It needs an index.html file inside a folder in C:\xampp\htdocs, including a script tag pointing to the source script *aframe.min.js*, of the wanted release, in the <head> section.

```html
<html>

<head>
    <script
      src="https://aframe.io/releases/1.2.0/aframe.min.js">
    </script>
</head>

<body>
<script>
...
</script>

<a-scene>
    <a-entity a-theta></a-entity>
</a-scene>

</body>

</html>
```

The *<a-scene>* component is the one handling 3D boilerplate, VR setup, and default controls. Inside of it, is an *<a-entity>* of the ECS mentioned above. The entities are container objects, the base for all scene objects, in which components are attached. Components, like the above *<a-theta>,* are reusable modules which provide visuals and/or functionality and can also contain other components inside them to have more complex behaviour. The *<a-theta>* component contains the main implementation and resembles the one on chapter 3.6, where an MJPEG livestream is projected and wrapped inside a sphere.

```
<script>
    AFRAME.registerComponent('a-theta', {
        schema: {
            radius: { type: 'number', default: 15 },
            width: { type: 'number', default: 64 },
            height: { type: 'number', default: 32 },
            color: { type: 'color', default: '#AAA' }
        },
        init: function () {
            var data = this.data;
            var el = this.el;

            this.loader = new THREE.TextureLoader();
            this.geometry = new THREE.SphereGeometry(
                                    data.width,
                                    data.height,
                                    data.depth
                                    );
            this.geometry.scale(-1,1,1)
            this.material = new THREE.MeshBasicMaterial({
                map: this.getImage()
            });
            this.material.needsUpdate = true;
            this.mesh = new THREE.Mesh(this.geometry, this.material);
            this.mesh.position.set(0,0,0)
            el.setObject3D('mesh', this.mesh);
        },

        tick: function (time, timeDelta) {
            this.mesh.material.map.img = this.getStream();
            this.mesh.material.map.needsUpdate = true;
        },

        getStream: function () {
            var stream = "https://10.64.44.242:5000/get_stream"
            return this.loader.load(stream);
        }
    });
</script>
```

The component needs a *schema* that defines and describes its properties, like dimensions and color. This is stored in the *data* variable used in the component initialization to create an imaginable sphere with *THREE.SphereGeometry* and invert its scale on x-axis. The *THREE.MeshBasicMaterial* is the one receiving the livestream from the reverse proxy server with the help of *THREE.TextureLoader*, which renders it on the material. The real 3D object is the *THREE.Mesh*, combining both geometry and material. The material is updated whenever *tick* callback is triggered, on each render loop. With this simple snippet of code, the result appears below.

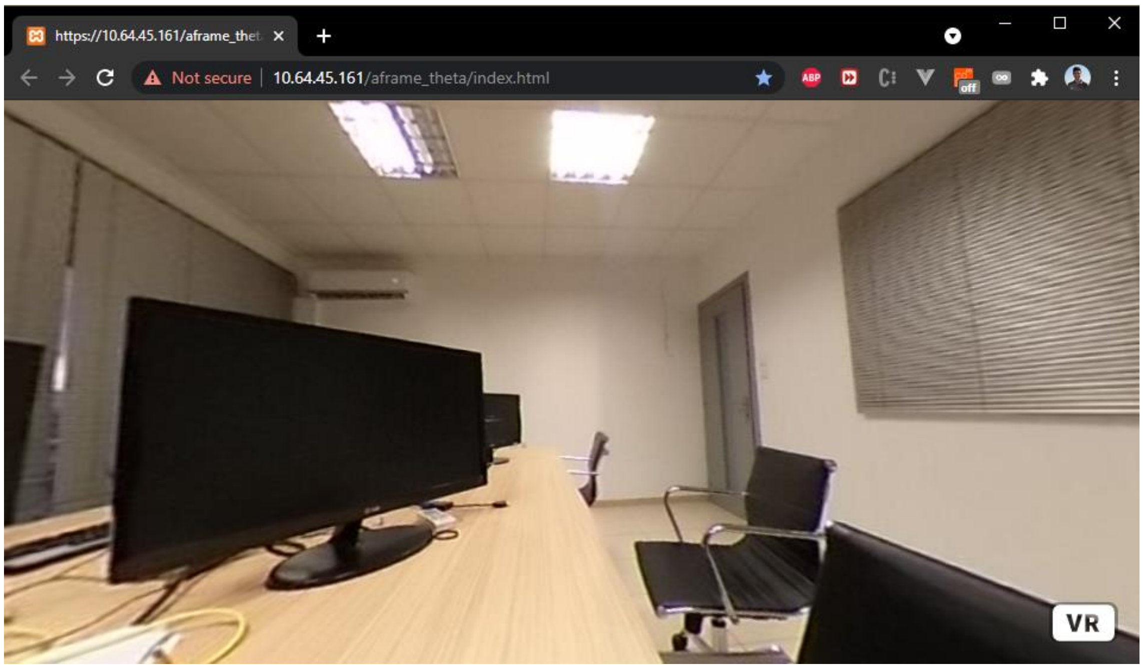

https://10.64.45.161/aframe_thet
Not secure
10.64.45.161/aframe_theta/index.html
VR

Figure 49: MJPEG livestreaming in browser using A-Frame

3.7.5: Conclusion

Browser VR livestreaming was a difficult task to implement with many obstacles and constraints due to the requirements of browsers with VR and with their CORS policy. The WebGL-WebXR solution is inconvenient because of its networking and memory-allocation management. Although, if those difficulties are known, there can be a workaround. Because of its synergy with the Unity game engine, it can be a powerful tool for developing a great variety of features inside immersive and interactive VR livestreaming applications. The A-Frame solution was by far easier, with much less constraints than WebGL. It is a web application environment, with game engine traits, offering a vast variety of networking features and more than enough functionality to build VR livestreaming applications in browser. These two frameworks open a great field in browser VR content distribution.

CHPATER 4: CONCLUSION

4.1: Summary

The goal of this thesis was to examine various approaches and implementations to achieve VR functionality side by side with 360^o video livestreaming. 360^o images and videos already are being used for virtual tours in the real estate market. Livestreaming gains more ground in the gaming and the entertainment community, day by day. Virtual Reality finds more and more practical applications in everyday life. The 360^o video livestreaming is a technology that currently few developers experiment on. Virtual Reality on the other side has a huge community when it comes to developing. Blending those two together is a great step for the many inconceivable applications to come in tomorrow’s society.

4.2: Discussion

In the contents of this thesis, the solutions we examined were either easy to implement or more complex. YouTube’s approach provides VR features while 360^o video livestreams and also permits users to spectate over the Internet, not only over LAN. Easy as this may be, the camera needed to be tethered with a USB cable and the resolution options are limited in order to keep the minimum livestream latency, which is clearly greater than latency over LAN. like having to deploy a reverse proxy server to achieve the same output over LAN.

On the other hand, Unity and A-Frame implementations are providing a freedom in the developing of the applications, extending them further than just livestreaming purposes. The current infrastructure was only allowing deployment over LAN. This resulted in greater resolutions with much smaller latency times while also having the camera connected wirelessly, allowing it to move freely.

4.2: Future work

As a future step, to enhance the survey conducted in the current thesis, we should carry out a detailed comparison of the final implementations. The characteristics to be compared should be the resolution, the frame rate and latency of the livestream but also the camera connectivity, the VR features capabilities, and the distribution potential.

360^o video livestreaming through VR could also benefit from the technology of machine vision and would result in an advanced version of mixed reality environment. A personal goal for the future of this project is to integrate it in a robotic arm teleoperation project.

BIBLIOGRAPHY

[1] D. Ochi, Y. Kunita, A. Kameda, A. Kojima and S. Iwaki, "Live streaming system for omnidirectional video," in *2015 IEEE Virtual Reality (VR)*, 2015.

[2] M. Jamali, F. Golaghazadeh, S. Coulombe, A. Vakili and C. Vazquez, "Comparison of 3D 360-Degree Video Compression Performance Using Different Projections," in *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, 2019.

[3] I. D. D. Curcio, H. Toukomaa and D. Naik, "360-Degree Video Streaming and its Subjective Quality," in *SMPTE 2017 Annual Technical Conference and Exhibition*, 2017.

[4] M. Zheng, Y. Tie, F. Zhu, L. Qi and Y. Gao, "Research on Panoramic Stereo Live Streaming Based on the Virtual Reality," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021.

[5] A. Erfanian, F. Tashtarian, A. Zabrovskiy, C. Timmerer and H. Hellwagner, "OSCAR: On Optimizing Resource Utilization in Live Video Streaming," in *IEEE Transactions on Network and Service Management*, 2021.

[6] P. Maniotis and N. Thomos, "Tile-based edge caching for 360° live video streaming," in *IEEE Transactions on Circuits and Systems for Video Technology*, 2021.