ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Design and implementation of a software-based vision mixer

AUTOR: Oier Lauzirika Zarrabeitia

TITULACIÓN: Grado en Ingeniería de Sonido e Imagen

TUTOR: José Luis Rodríguez Vázquez

DEPARTAMENTO: Ingeniería Audiovisual y Comunicaciones

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: Agustín Rodríguez Herrero

TUTOR: José Luis Rodríguez Vázquez

SECRETARIO: Enrique Rendón Angulo

Fecha de lectura:

Calificación:

El Secretario,

Los ingenieros inventamos la televisión.
De la invención de la *caja tonta* se encargaron otros.

Resumen

Cualquier producción audiovisual en vivo requiere de un mezclador de vídeo para poder componer y conmutar entre fuentes de vídeo. Pese a que su invención se remonta a los comienzos de las retransmisiones televisivas, siguen siendo un sistema clave en todos los programas en directo que empleen más de una fuente de vídeo (la inmensa mayoría).

Se puede asegurar que siempre habrá programas que, dada su naturaleza, el público quiera disfrutar de ellas en directo, como es el caso de algunos eventos deportivos, informativos o culturales. Pese a que en la última década el panorama audiovisual ha sufrido un vuelco con la disrupción del streaming, siempre que se sigan usando sistemas de vídeo para dar cobertura a estos eventos, los mezcladores seguirán existiendo de una u otra forma.

La larga trayectoria de los mezcladores no implica que no exista lugar para la innovación. Es más, dado que se encuentran tan estrechamente relacionados con un campo en rápido desarrollo como es la multimedia, existe mucho interés en mejorarlos. Una de las innovaciones que se están llevando a cabo en el ámbito de los mezcladores de vídeo es la utilización de software para emular estos sistemas. Esto presenta varias ventajas, primordialmente relacionadas con la flexibilidad, la agilidad y el bajo costo de la solución adoptada.

Por otro lado, también se está ahondando en la posibilidad de usar redes de ordenadores para el transporte de señales de vídeo dentro de un estudio de televisión, lo que se conoce como producción IP. Esto presenta ventajas muy similares a las descritas para los mezcladores definidos por software y además estas dos tecnologías se compenetran excelentemente.

El presente proyecto consiste en la implementación de un mezclador definido por software que adquiera las señales de vídeo a través de la red. Tiene como objetivo perseguir las ventajas que ofrecen estas dos tecnologías. Además, pretende realizar nuevas aportaciones al campo de los mezcladores permitiendo recortar cuadros de vídeo por curvas paramétricas.

Abstract

Vision mixers are used in all live broadcasts to switch and composite among all available video feeds. Despite them being introduced in the early days of television, they are still a key system in all live productions that use more than one video source (almost all).

The audience will always demand live coverage of some events due to their nature, as is the case of sporting events, news and cultural happenings. In the last decade, the disruption of video streaming has revolutionized the multimedia industry. In spite of this, vision mixers continue to exist in one form or another, regardless of the used transmission medium.

Even though vision mixers are a very old technology, this does not imply that there is no room for innovation in this field. What is more, as they are tightly related to a rapidly changing industry, as is the case of the multimedia sector, there is a lot of interest on improving them. One of the most important ongoing advancements in the field of vision mixers is the usage of computer software to emulate their functionality. This has several advantages, such as increased flexibility, agility and lower operation costs.

Additionally, there is a trend of using computer networks to carry video signals across television production facilities. This is known as IP production and it offers similar advantages to the ones enumerated earlier. Moreover, this technology fits very well with the software-based vision mixers.

This project consists in implementing a software-based vision mixer that ingests video from the network. The project pursues all the benefits of these two technologies. In addition, it pretends to introduce some features that are not offered by commercial mixers. For instance, it is able to crop video frames with an arbitrary shape defined by parametric curves.

Contents

List of Figures

List of Tables

Acronyms

ADC analog-digital converter. 16

API application programming interface. 31, 66

BOM bill of materials. 89

CAD computer aided design. 75

CCW counter clockwise ↺. 79

CG character generator. 13, 19, 25, 26

CLI command line interface. 30, 37, 44, 48, 51, 92

CPU central processing unit. 24, 28, 32, 34, 58–60, 76, 84

CSS Cascading Style Sheets. 31, 51

CW clockwise ↻. 79

DAC digital-analog converter. 16

DMA direct memory access. 32

DSK downstream keyer. 17, 19, 40–42

DVE digital video effect. 16, 17, 22, 24, 29, 43

EBU European Broadcasting Union. 26

EOTF electro-optical transfer function. 29, 32, 34

ETIST Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación. 55

FHD Full High Definition (1920 × 1080). 27, 58–60

FOSS free and open-source. 26, 51, 63

GLSL OpenGL Shading Language. 44, 84

GPU graphics processing unit. 24, 29, 31, 32, 34, 35, 59–61, 75, 79, 84

HSV Hue-Saturation-Value. 46

HTML HyperText Markup Language. 31

HTTP HyperText Transfer Protocol. 48

I/O input-output. 16, 25, 26, 31, 66, 69, 89

IC integrated circuit. 16

IDE integrated development environment. 31

IP Internet Protocol. 13, 16, 25, 28

JS JavaScript. 31, 51

LED light emitting diode. 89

M/E mix/effect. 16, 17, 22, 29, 37, 43, 49, 53, 60, 61, 65

MD Material Design. 51

NDI Network Device Interface. 28, 29, 31, 37, 40, 53, 54, 58, 59, 65, 69

NIC network interface card. 58

OBS Open Broadcaster Software. 26

OS operative system. 27, 31, 37

PC personal computer. 25

PISO parallel in serial out. 89

pixel picture element. 22, 32, 34, 35, 47

RGB Red-Green-Blue. 31–33

SDF signed distance field. 75, 84

SDI Serial Data Interface. 16, 25, 26, 28, 59, 66

SIMD single instruction multiple data. 84

SIPO serial in parallel out. 89

SOC system on a chip. 59, 61

SVT Sveriges Television. 26

TCP Transmission Control Protocol. 30, 48, 92

texel texture element. 35

TMU texture mapping unit. 34

TV television. 13, 15, 17, 28, 49, 67, 69, 89

UHD Ultra High Definition (3840 × 2160). 27, 28, 58, 59

UI user interface. 51, 66

UPM Universidad Politécnica de Madrid. 55

UPS uninterruptible power supply. 27

USART Universal Synchronous/Asynchronous Receiver Transmitter. 92

USB Universal Serial Bus. 92

USK upstream keyer. 17, 22, 40–42

Chapter 1

Introduction

Vision mixers are a central element in a live television (TV) production facility. Their purpose is to switch and compose among all the available video sources, such as cameras, media players, character generators (CGs), etc... Therefore, they need to operate in real-time, this is, they must be able to handle the data on their inputs in a reasonable and invariant amount of time. Historically, they have been discrete systems with highly specialized hardware, but advancements in computer technology have led to the possibility of emulating them using software.

This project consists in implementing a software-defined vision mixer that can be run on any modern computer. The primary advantage of software-based solutions is that they are much cheaper to operate, as broadcast-grade equipment is costly. Therefore, software-based solutions enable small groups of people to produce live video broadcasts with a reduced budget. This has been important in the recent pandemic situation, as many artists could not perform in front of an audience, so streaming their work was their only option to make a living. Other potential scopes of use involve the educational environment, as each of the students can be provided with its own vision mixer.

The project has been named after the cenital camera angle. This camera angle looks at the subject from the sky, hence its name. More specifically, the *Cenital* name refers to the implemented mixer and the control application is named as *Cenital Web Control*.

This report firstly describes in chapter 2 the history and operating principles of modern vision mixers. Moreover, this chapter also addresses the existing software-based solutions and introduces the Internet Protocol (IP)-based production. The next chapter (chapter 3) describes the specifications of the implemented mixer. Here, the international colorimetry standards covered by this mixer will be enumerated. Then, in the chapter 4, the implementation details and general architecture of the software are deeply described. However, some ancillary implementation details are left for the appendices A and B. Finally the chapter 5 discusses the results obtained for this project. Moreover, it serves as a guide to properly scale computer hardware to obtain the desired results at the lowest possible cost.

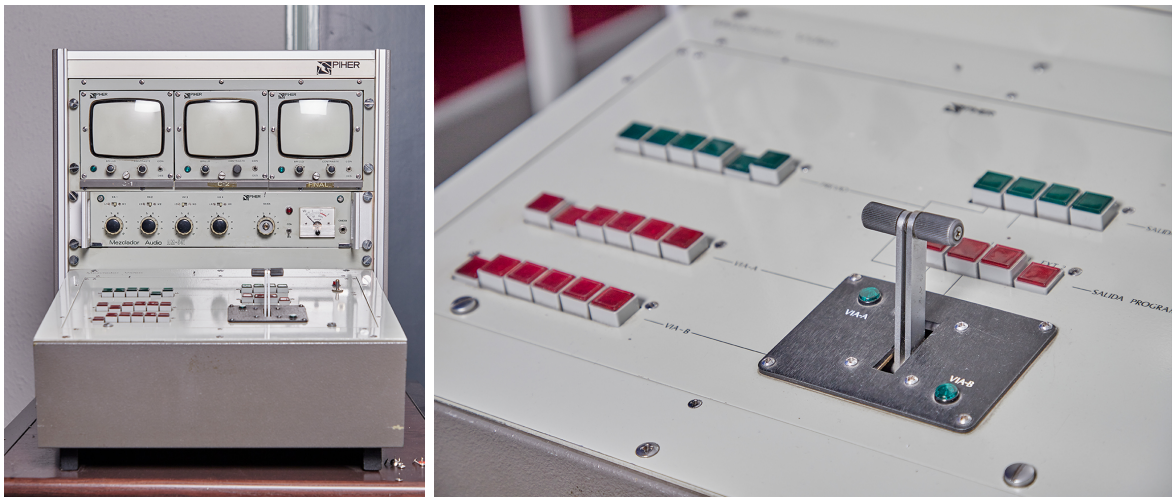
Chapter 2

State of the art

2.1 Historical evolution of vision mixers

The first vision mixers were introduced into the market as soon as the first commercial TV broadcasters began operations in the mid 1930s. At that time, vision mixers operated in a mechanical manner, this is, they merely consisted of switch matrices used to route the input signals to the main output. As no signal processing was performed, all inputs needed to be *genlocked*; In other words, their horizontal sync pulses needed to happen at the same time[1].

As time passed, these commutation matrices evolved into complex analog mixers, which allowed to perform several effects using analog electronics, although they still needed *genlocked* sources. For instance, these mixers were able to fade or wipe between sources, or compose using *luma key*s[2]. This same time, the standard program-preset operation model was introduced, as opposed to the older A-B model. The mixer displayed in the figure 2.1 features both an A-B and a program-preview bank.



Photos by Facultad de Ciencias de la Información, Universidad Complutense de Madrid

Figure 2.1: Piher vision mixer, circa 1960

Shortly after, in the late 1960s, some TV stations began broadcasting colour signals. This implies that vision mixer technology needed to evolve to handle these new signals. The presence of colour was useful to introduce a powerful technique that is still used: *chroma keying*[3][4]. This gives the ability to delete parts of a frame based on its colour. It is typically used in weather forecasts; where the presenter sits in front of a green screen which gets replaced with a weather

map by the vision mixer.

Several years later, advancements in digital integrated circuit (IC) manufacturing made possible to manipulate video digitally[5]. This introduced discrete digital effect processors, known as digital video effects (DVEs). These allowed spatial transformations of video frames, which were impractical with analog electronics, as high bandwidth signal delays would be needed. Shortly after, these discrete effect processors were integrated into the mixers. Nowadays, although the whole pipeline of the mixers is digital, the DVE term is still used to refer to those effects which involve spatial transformations of a frame[6][7].

There was a short period in the late 1980s where all video processing was performed with digital electronics but the video input-output (I/O) was performed using analog signals by the means of analog-digital converters (ADCs) and digital-analog converters (DACs). This suddenly changed with the introduction of the Serial Data Interface (SDI) communications protocol, which allows transmitting video alongside ancillary data over a 75Ω coaxial cable[8]. This protocol is still used on its latest revision, which specifies a stream of 12Gb/s. Moreover, it has been adapted to support other transmission mediums such as optical fiber.

The latest advancements came in the late 2000s with the introduction of the first software-based mixers by NewTek. At this same time, the first IP based production protocols were also introduced. In short terms, IP based production replaces the dedicated coaxial cables with the more versatile Ethernet cables. These in turn can be twisted-pair shielded cables or optical fiber. IP based production and software-based mixers work well together, as they facilitate live video production with no dedicated broadcasting equipment.

2.2 Common vision mixer features

Modern vision mixers usually integrate many simpler mixers, known as mix/effect (M/E) banks. Each of these units is a full-fledged mixer, which will be responsible of generating a single composition. However, they might generate more than one distinct output signal for reasons that will be explained later. In rough terms, the category of a vision mixer can be measured by its amount of M/E units, although this is quite simplistic, as the feature set of each one of these banks may be more important, but more often than not, there is a certain amount of linearity among these two specifications. Most mixers in the market have 1 to 4 M/E units, topping out at 8.

Each M/E has a signal matrix known as the *crosspoint*. It serves as a router for input signals, which are represented as columns. Meanwhile, each of the feeds required by the M/E will be represented as a row, which can only be fed by one or zero input signals.

M/Es are based on the program/preview operation model. This operation model consists in configuring the next composition in advance at the preview bus. Meanwhile, the program bus is permanently on air. When the time is right, the contents of the preview bus are swapped with the contents of the program bus, either by an instantaneous cut or a transition. Therefore, each M/E generates two video signals: program and preview. This gives the operator the ability to double check for errors in the composition before it is broadcasted. However, manual changes in the program bus can be performed, although it is not a good operation practice, as any composition errors will be instantaneously transmitted.

Each of these composition buses are founded on the background layer. This layer is always visible -although it may be black or transparent- and encompasses the whole screen. As suggested

by its name, its only ability is to display a video signal in the background. If no compositing is required, it is used to display the main video feed.

Several layers called *keyers* may be hierarchically added on top of this background. Low-indexed *keyers* can be occluded by higher indexed ones. Moreover, they can be classified as upstream keyers (USKs) and downstream keyers (DSKs). The difference between these two is that USKs are added before the transition generator, whilst DSKs are added to the signal that has the transition already on it. If the transition is inactive, it can be considered that DSKs have a higher priority, so they will occlude USKs. Usually the USKs are used to perform compositions that are background dependant. Meanwhile, the DSKs are used to add more stationary contents, such as banners and TV station watermarks. Therefore, most vision mixers offer more USKs compared to the number of DSKs. A possible case of each one of them is shown in the figure 2.2.

Note: Both USKs are DVEs. The DSK is a linear keyer.

Figure 2.2: Possible use case of USKs and DSKs

A general view of the signal path of a hypothetical 8-input M/E unit can be seen in the figure 2.3.

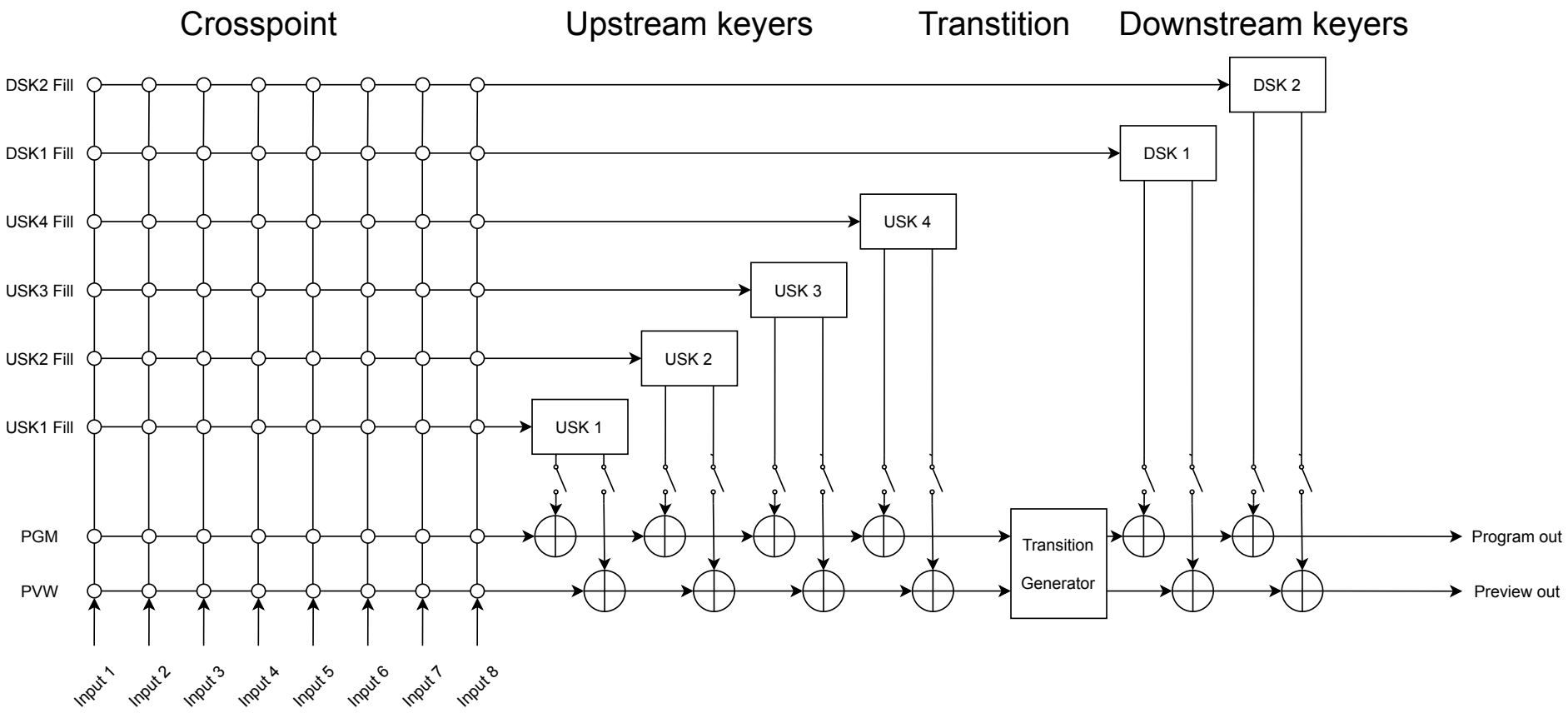


Figure 2.3: Signal path of a hypothetical ME with 8 inputs, 4 USKs and 2 DSKs

2.2.1 Mixing equation

Most of the effects performed by a vision mixer can be modelled using the equation (2.1), where a is the background image and b is the foreground image. These two images get weight-averaged using the control signal c and its complementary. This control signal is also an image. Therefore, places where c is high, the foreground image will predominate. The reverse is also true, if c is low, the background image will predominate. In the context of digital image processing, this is a linear point operation.

$$r(x,y) = (1 - c(x,y)) \cdot a(x,y) + c(x,y) \cdot b(x,y) \qquad c(x,y) \in [0,1] \tag{2.1}$$

To obtain colorimetrically correct results, a and b must be linear, this is, they must not contain a gamma correction. The most basic form of gamma correction is shown in the equation (2.2) where x' denotes the gamma corrected value and x is the linear value. Typically, γ equals 2.2. Similarly, ITU-R described in the BT.1886 standard[9] the gamma transfer function that is shown in the equation (2.3). These are not the only gamma corrections used in the industry, but they are the most important ones.

$$x = x'^{\gamma} \Longleftrightarrow x' = x^{\frac{1}{\gamma}} \qquad x, x' \in [0,1] \tag{2.2}$$

$$x = \begin{cases} \frac{x'}{4.5} & \text{if} \quad x' \leqslant 0.081 \\ (\frac{x'+0.099}{1.099})^{\frac{1}{0.45}} & \text{otherwise} \end{cases} \Longleftrightarrow x' = \begin{cases} 4.5x & \text{if} \quad x \leqslant 0.018 \\ 1.099x^{0.45} - 0.099 & \text{otherwise} \end{cases} \tag{2.3}$$

2.2.2 Keying

As mentioned earlier, keyers are used to overlay an image on top of the background. However, to be able to see that background, parts of the foregound image need to be conditionally transparent. The process of discriminating parts of the overlay is known as keying. Several techniques have been developed to achieve such a task, the most common of which will be explained hereunder.

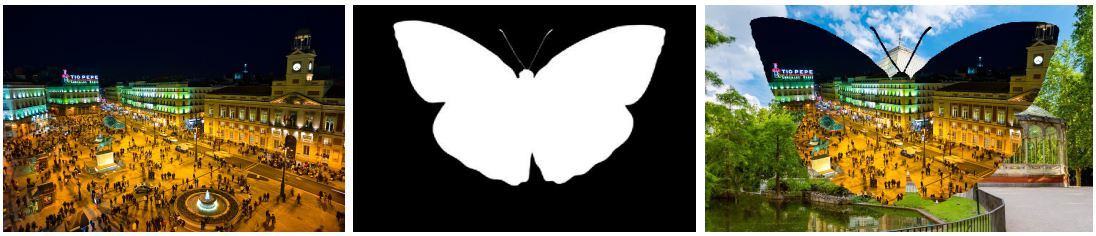
Linear keying

This is one of the oldest keying techniques. It consists in using a matte signal which represents the transparency of the filling signal. Considering the mixing equation (2.1), the background and fill images would be a and b respectively. Meanwhile, the matte signal would be equivalent to the control signal.

If a signal carries its own matte information, it is said that it has an alpha channel. The most common usage of the linear key is to overlay CGs or other types of computer generated graphics. For instance, the DSK shown in the figure 2.2 could be using this technique. Unlike other types of keying, it requires two video sources -apart from the background signal- as seen in the figure 2.4. Even though this example uses a binary matte signal -black or white-, it is worth noting that it supports gray-scale matte images which smoothly blend from the background image to the keyed image.

Pattern keying

This is similar to the previous type of keying. However, the matte signal is a internally generated preset. In the figure 2.5 a list of patterns offered by a commercial mixer can be observed.



(a) Fill image (b) Matte image (c) Linear keyed image

Figure 2.4: Linear key example

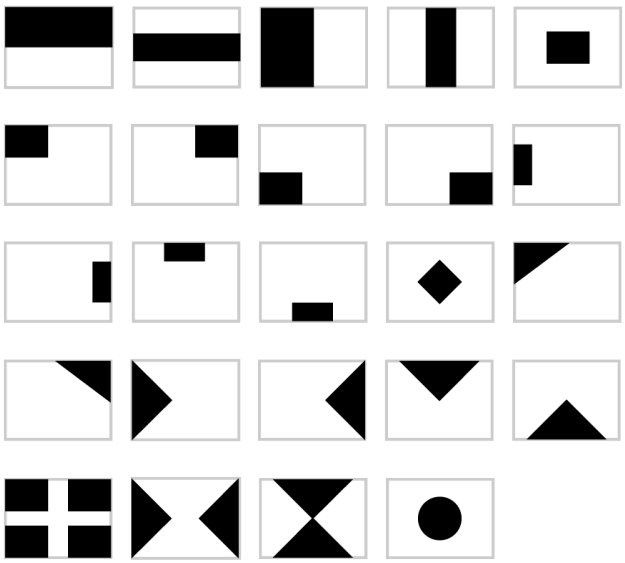


Figure 2.5: Patterns offered by Ross vision mixers

Luma keying

This type of keying is very similar to the linear key. However, instead of using a dedicated signal for transparency information, it uses the filling signal's own luminosity. Similarly to linear keying, a and b terms of the mixing equation (2.1) would be the background and fill images, but the control signal would be a function of the filling singnal's luminance channel: $c(x, y) = f(Y(x, y))$, hence its name.

This is only useful if the transparent parts are very black and opaque parts are very white or vice versa. Since the luminosity information can not be controlled separately, these keyers usually have more advanced settings that model the behaviour of the function that relates the luminance to the control image. Possible settings for this transfer function could be a lower and upper threshold, a toggle for inverting the result, etc... An example of this type of keying is displayed in the figure 2.6.

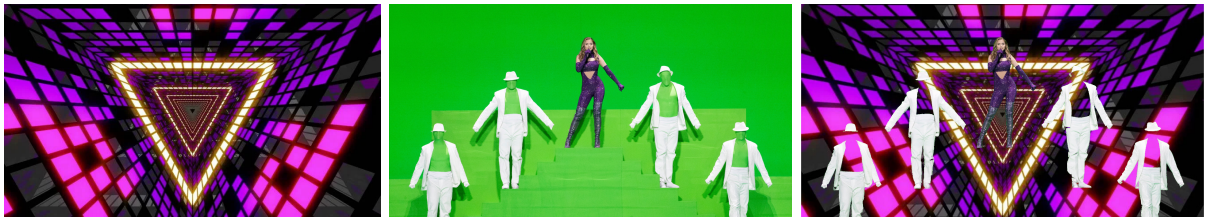
(a) Background image (b) Key source (c) Luma keyed image

Figure 2.6: Luma key example

Chroma keying

As opposed to luma keying, chroma keying uses the color information of a video signal to discriminate the transparent and opaque parts of an image. In this case, the obtaining of the control signal is more complex and highly implementation dependant, so it will not be described here. However, it is worth mentioning that it should be done in a color model where the color information is explicit, as is the case of YUV, YIQ, YCbCr, YCtCp, HSV or HSL.

This keying is sometimes referred as the *green-screen effect*, as it is usually configured to reject the green color[8]. Historically, its most widespread usage has been in weather forecasts, where the presenter gets overlaid in front of a weather map. The green color is commonly used because it rarely appears on skin-tones and costumes. However, if this is the case, a blue screen is used instead. The example shown in the figure 2.7 makes a more creative use of this effect, as it uses green suits on purpose.



(a) Background image (b) Key source (c) Chroma keyed image

Figure 2.7: Chroma key example

DVEs

DVE stands for Digital Video Effects. Although modern mixers are fully digital, this name dates to the analog mixers, when certain effects could not be achieved with analog circuitry. Mainly, this refers to the ability to perform geometrical transformations on an overlay, such as scaling, rotating and positioning it in space. For example both USKs in the figure 2.2 are using a DVE. Among all the keying techniques described here, this is the only one that can not be modelled using the mixing equation, although some implementations may use it to embed the transformed image into the background[8].

It is usually paired with animation controls, as it is useful to give dynamism to these effects. Moreover, it may offer the ability to generate borders around the transformed frame[10] and perform complex spatial transformations like 3D effects[11].

2.2.3 Transitions

As mentioned earlier, M/Es have two compositing buses which can be swapped with a transition. This transition is applied to the signal that carries the background information with the USKs overlaid. Mixers offer a wide range of transition effects, the most basic of which will be explained hereafter.

Similarly to keyers, most transition effects also use the mixing equation (2.1). However, as opposed to keyers where the control signal is generated from an external source (except for the pattern keying), transition's control signal is internally generated and depends on the progress of the transition. This progress will be denoted with $t \in [0, 1]$. Therefore, the control signal will also be a function of the progress: $c(x, y, t)$.

From the operator's point of view, the progress of the transition can be controlled in two manners. The first one is pressing the automatic transition button. This will animate the t parameter to travel from 0 to 1 in a preconfigured amount of time. The other way is to manually animate the t parameter using a potentiometer. This potentiomenter is usually referred as the *T-bar* due to its characteristic 'T' shape, as shown in the figure 2.8.

Mix

This is the most basic transition effect. It consists in steadily fading from one source to another, as displayed in the figure 2.9. In its most basic form, it can be modelled using a control signal that is constant throughout the frame: $c(x, y, t) = t$.

Some mixers have the ability to transition from one feed to another passing through a third source, as illustrated by the figure 2.10. This can be considered two mix transitions executed one after the other. Although the middle image can be anything, more often than not, it is a solid color[10].

Wipe

As opposed to the previous transition, this one uses a control signal that is not the same for all picture elements (pixels) of an image. As the shape of this signal can be arbitrary, this leads to a wide fan of possible variants for this kind of transitions. Some example patterns are shown in the figures 2.11 and 2.12. The generation of these patterns is usually similar to the patterns described

Figure 2.8: *T-bar* of a Grass Valley Kayak 1 M/E vision mixer

Figure 2.9: Mix transition example

Figure 2.10: Dip transition example

for pattern keying.

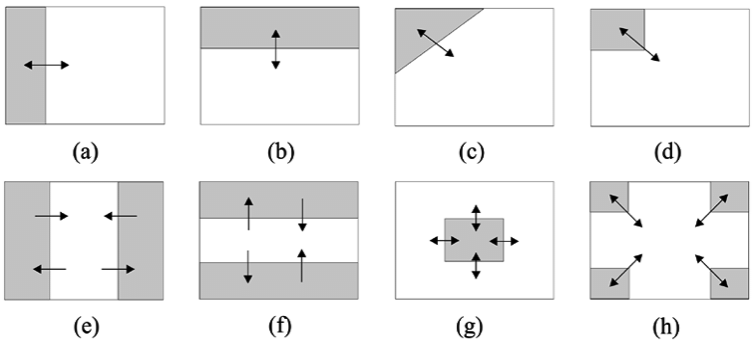


Figure 2.11: Typical wipe transition patters

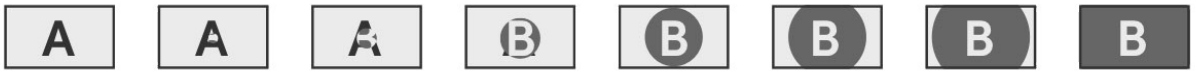


Figure 2.12: Wipe transition example

DVE-based

DVE transitions treat the in-going and out-going feeds as overlays where at least one of them has an animation which makes is appear or disappear from the screen. Obviously, the the usage of DVE effects implies that it will not use the mixing equation. Similarly to wipe patterns, DVE transitions also have many possible animations. An example of one of them is displayed in the figure 2.13.

Figure 2.13: DVE transition example

2.3 Software based vision mixers

Recent increases in computational power have led to the possibility of manipulation video using software tools. This brings up the possibility of emulating the vision mixers described earlier using computer software. In essence, this allows using any computer for live video production, something that previously could only be done using dedicated equipment[12].

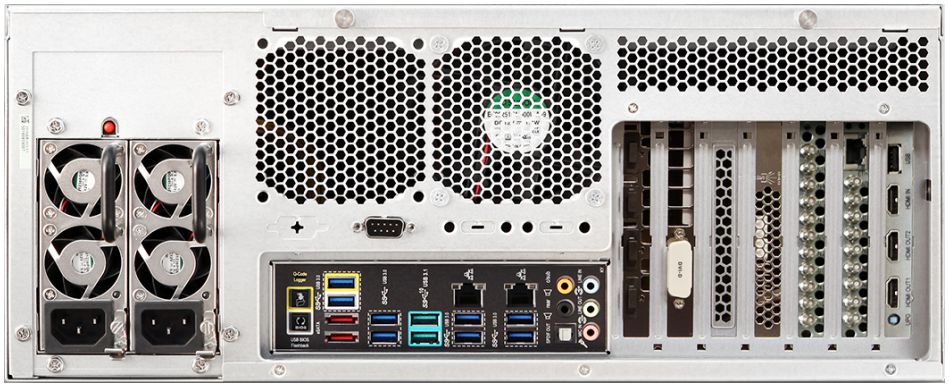
With the purpose of achieving the maximum performance, software-based vision mixers use graphics processing unit (GPU) acceleration, this is, they use computer's dedicated graphics processing hardware to offload image processing from the central processing unit (CPU). The main limiting factor will be bit-rates across different buses of the computer, which will impact the maximum amount of sources that can be handled by a given machine.

All vision mixers need to be able to ingest/output video from/to external sources, such as cameras. Most mixers do this though the SDI interface. In order to do this using a software solution, the host computer must feature a SDI I/O card. However, if this is not present, network based solutions exist, namely, IP based production. Moreover, all computers can output video though the graphic card's consumer-grade video outputs. Therefore, in a IP based production environment, any computer can play the role of the vision mixer.

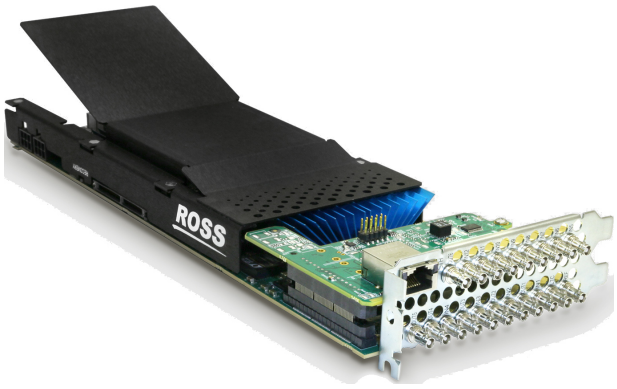
2.3.1 Existing solutions

According to experts, the first software mixer was NewTek's Amiga Video Toaster, released in 1990. However, this was an expansion card for the Amiga 2000 computer which implemented a hardware vision mixer alongside a companion app, defeating many purposes of software vision mixers. Nevertheless, it allowed to produce video in a centralized manner, converging in the same machine a 4 channel vision mixer and a CG. It was not until 2005 when this same company introduced its TriCaster line of products[13]. NewTek TriCasters are a hardware solution, but under the hood they are built with off-the-shelf personal computer (PC) parts and video I/O cards.

Many traditional brands have moved towards NewTek's first approach, this is, the integration of mixer-specific hardware inside PCs. The primary advantage of this is that the most part of handling video can be done in hardware, allowing to obtain a higher throughput whilst keeping the flexibility of software-based solutions. A example of this approach is Ross' Graphite line of products, that as seen in the figure 2.14 is based on a standard PC motherboard with a proprietary PCIe expansion card[14].



(a) I/O of the mixer



(b) Add-on PCIe card

Figure 2.14: Ross Graphite vision mixer

Later in 2010, another company called vMix emerged with a video switching software which could be run on any hardware. As of writing this report, this software still exists and its developers have been pushing new releases periodically.

The next year, in 2011, the first version of Open Broadcaster Software (OBS) was released, which at the time was a simple free and open-source (FOSS) video streaming tool. After years of development, this software has evolved into a scene-based vision mixer with lots of features and I/O capabilities[15]. Its main users are streamers, which more often than not have modest video production requirements such as embedding their webcam in the screen for streaming it to popular streaming platforms.

Another related solution is CasparCG, a general purpose FOSS CG, developed by the Sveriges Television (SVT). Although it is not strictly a vision mixer, it has been included here because it can not only output video, but also consume it and perform complex compositions with it, as seen in the figure 2.15. According to its authors, it could be used as a vision mixer, but this is not its intended operating purpose. It powers many broadcasts produced by members of the European Broadcasting Union (EBU)[16].

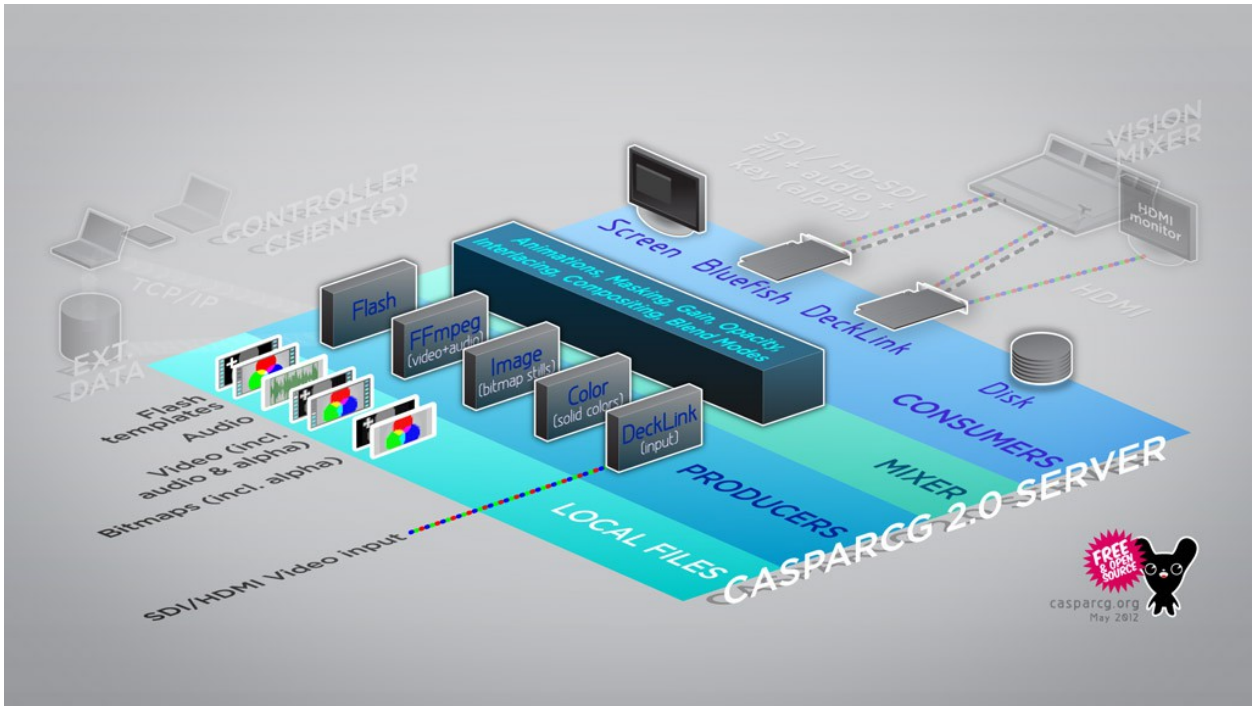


Figure 2.15: CasparCG's architecture

2.3.2 Benefits and drawbacks

The major benefits of using software solutions are the cost savings and their flexibility. Even considering the cost of a high-end workstation with multiple SDI I/O cards, it will easily outperform a higher prized hardware solution in terms of composition capabilities. Moreover, software-based vision mixers tend to integrate many other production systems such as CGs, media players, virtual set generators, etc... [17][18]. This not only reduces the cost on auxiliary equipment, but also its own physical I/O requirements.

What is more, computational resources can be dynamically allocated. For instance, the same computer may be used to produce a simple Ultra High Definition (3840 × 2160) (UHD) show or to perform a very complex Full High Definition (1920 × 1080) (FHD) compostion, or even multiple FHD compositions. As opposed to this, hardware mixers have dedicated resources for dedicated tasks. This means that for the previous example, a buyer would need to choose a UHD mixer capable of producing multiple complex compositions, even if it is not going to use all features simultaneously.

However, software-based vision mixers also present some drawbacks. First of all, the latency is orders of magnitude higher. For istance, Black Magick Atem switchers have a latency lower than one scan-line when genlocked[10] (≈ 30µs for 1080p30), whilst software solutions have latencies in the neighborhood of a few frames[17] (≈ 66ms for the same situation). This last number rubs the the delays noticeable by human brains. Therefore, this makes them inadvisable for use cases where the audience has other means of sensory input, as is the case of image magnification in a concert, as shown in the figure 2.16. For the rest of broadcast use cases, this is not a serious concern.

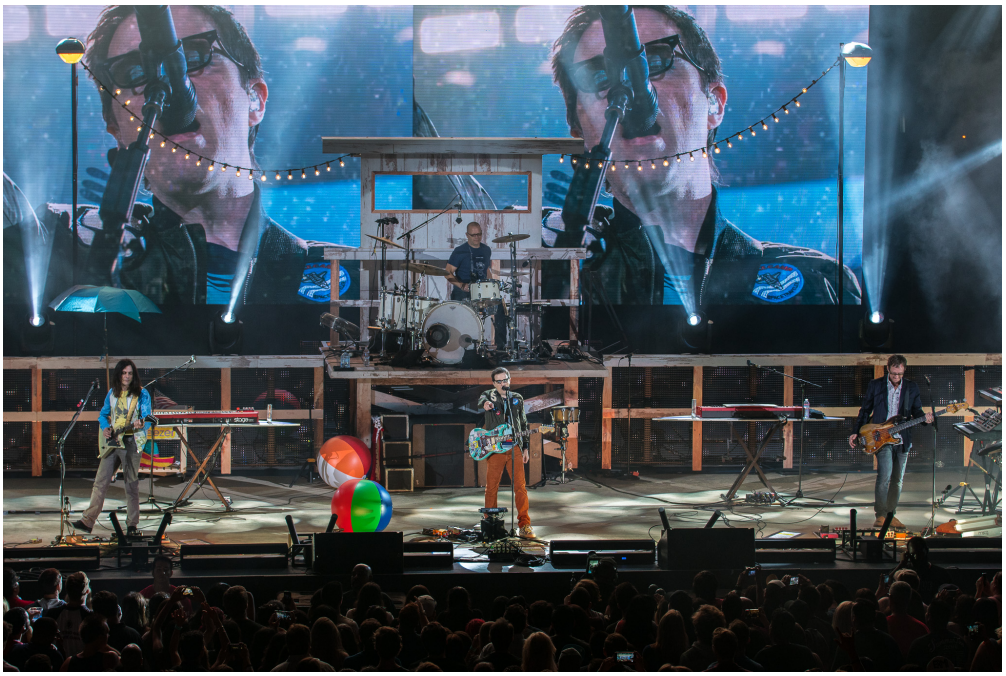


Figure 2.16: Image magnification in a Weezer's concert

Another drawback is that software-based solutions are not as reliable as hardware-based ones. As broadcasting equipment is produced with very high quality standards, physical vision mixers are virtually unbeatable, being able to run 24/7 for years with very low maintenance, as long as they are protected against power outages with a uninterruptible power supply (UPS). As opposed to this, software-based mixers are susceptible to software-bugs, operative system (OS) updates, etc... This last concern rules out using them for continuity control.

Finally, current computers can not meet the input requirements of large-scale shows. However, as computer's internal bandwidths and computational power rise, this gap will become narrower, eventually disappearing.

2.4 IP based production

As mentioned earlier, IP based production consists in using standard computer networking infrastructures to transport video inside a TV production facility. This is a very current topic, as many TV stations are undergoing through a conversion from SDI based infrastructures to IP based ones[19].

As opposed to the traditional streaming, IP based production must meet some additional requirements. For instance, the quality of the image must be comparable to the quality provided by uncompressed SDI. This rules out using any heavy compression algorithms. Moreover, the latency must be as low as possible, making it impossible to use bidirectional frame compression. Additionally, the signal must fit into a 1Gb/s or 10Gb/s link. Finally, it must be able to carry some ancillary data from and to the source.

The first protocol to perform such a task was introduced in 2007 as the SMPTE-2022. This protocol allows sending a MPEG-2 stream or a packetized SD-SDI (270Mb/s) stream over a 1Gb/s Ethernet cable[20]. Later in 2017, this standard was upgraded to the SMPTE-2110, which brings many benefits, such as better synchronization mechanisms and support for any resolution, including those larger than UHD, as long as there is enough bandwidth [21]. The largest problem with the SMPTE-2110 protocol is that it is not meant to be used by computers, specially in the case of UHD signals, as the bitrate is too high for modern CPUs.

Another proliferating protocol is NewTek's Network Device Interface (NDI), which is a royalty-free standard. As opposed to SMPTE's IP based production protocols, it is paired with a intra-frame compression codec, although it is able to transport existing video codecs such as H.264 and H.265. This makes it possible to use it with standard networking gear and modest computers whilst keeping resolutions up to UHD. Their own codec has been designed to meet the quality requirements mentioned earlier.

Chapter 3

Specifications

This project will attempt to implement a software-based vision mixer which receives live video feeds through the NDI protocol described in the previous chapter. It will also have the capability of playing pre-recorded video clips and images. The processed images will be extracted from the mixer using the integrated video ports of the host computer.

All image processing will have to be performed hardware-accelerated with GPUs to obtain the maximum performance. If possible, video decoding will be hardware-accelerated as well.

Similarly to other software-based vision mixers, it will have to be as flexible as possible, allowing multi-M/E configurations with an arbitrary number of keyers. These keyers will be configurable as a combination of linear, luma, chroma or pattern keys. As opposed to all mixers on the market, the pattern will not be a preset. It will be user-definable by parametric vector-graphics instead. 3D DVE effects will also be available. All aforementioned effects may be simultaneously applied on the same keyer.

The mixer will have to be able to work with arbitrary configurations, such as nonstandard resolutions and framerates. Moreover, the configuration across M/E banks may be heterogeneous.

For the sake of obtaining colorimetrically correct results, it will support the following colorimetry standards and electro-optical transfer functions (EOTFs):

- ITU-R BT.601-7[22]
- ITU-R BT.709-6[23]
- ITU-R BT.2020-2[24]
- SMPTE 240M[25]
- SMPTE 431 (Display-P3)[26]
- SMPTE 432 (DCI-P3)[27]
- SMPTE ST.2084 (PQ)[28]
- ARIB STD-B67 (HLG)[29]
- IEC61966-2-1 (sRGB)[30]
- IEC61966-2-4 (xvYCC)[31]
- Adobe RGB

Architecturally, the mixer will be engineered to follow the server-client pattern. The server will have the duty of ingesting, processing and outputting video, while the client(s) will be the front-end commanding to the server changes in the configuration. In case there are multiple clients connected to the same server, the server will also be responsible for synchronizing the state across all clients. Client-server communications will be performed using a purpose-built command line interface (CLI) over Transmission Control Protocol (TCP) or Web-Sockets.

Clients will serve as the user interface of this software. The server is agnostic of this interface, so multiple variants of the client may be implemented. In this project, the primary focus will be on developing a web application that allows to configure most aspects of the server. However, a hardware prototype of a control panel will be considered as well.

Chapter 4

Description

The adopted solution can be divided into three parts. The first part relates to the low-level handling of video signals in the server. The second one is about the implementation of the server itself and the third one treats the client web application. Additionally, details about the implementation of the physical control panel prototype can be found in the Appendix B.

The parts regarding the server have been implemented using C++17 and they are meant to be run under Linux. However, no OS-specific libraries have been used, so it should work with any other OS with minor modifications. The web application has been implemented using the standard web development languages, namely HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript (JS). Therefore, there is no OS nor web browser limitation regarding it. Both pieces of software were developed using Microsoft's Visual Studio Code integrated development environment (IDE).

4.1 Video manipulation library

This section describes how low-level video data is handled by this software. To do so, it uses a video manipulation library called *Zuazo* which is a personal project of the author of this work. This software library allows to manipulate real-time video using GPU acceleration though the *Vulkan* graphics application programming interface (API).

Zuazo is divided into multiple modules. A project may choose to include an arbitrary set of them. The modules used in this project are the following:

- **zuazo**: Core functions for the library. Must be always included if any component of the library is used.
- **zuazo-compositor**: Video composition utilities.
- **zuazo-ndi**: NDI I/O. Wraps NewTek's official NDI library in a *Zuazo* coherent interface.
- **zuazo-ffmpeg**: Compressed video input through the *FFmpeg* library.
- **zuazo-window**: Window and monitor output the *GLFW* library.

4.1.1 RGB conversion pipeline

Most image operations are intended to be performed in a linear Red-Green-Blue (RGB) color model. This means that the component values must be proportional to the tristimulus received by the camera. However, this signal is rarely used in transmission and storage. Therefore, all ingested

signals are converted to an adequate pixel format. The general conversion pipeline is shown in the figure 4.1.

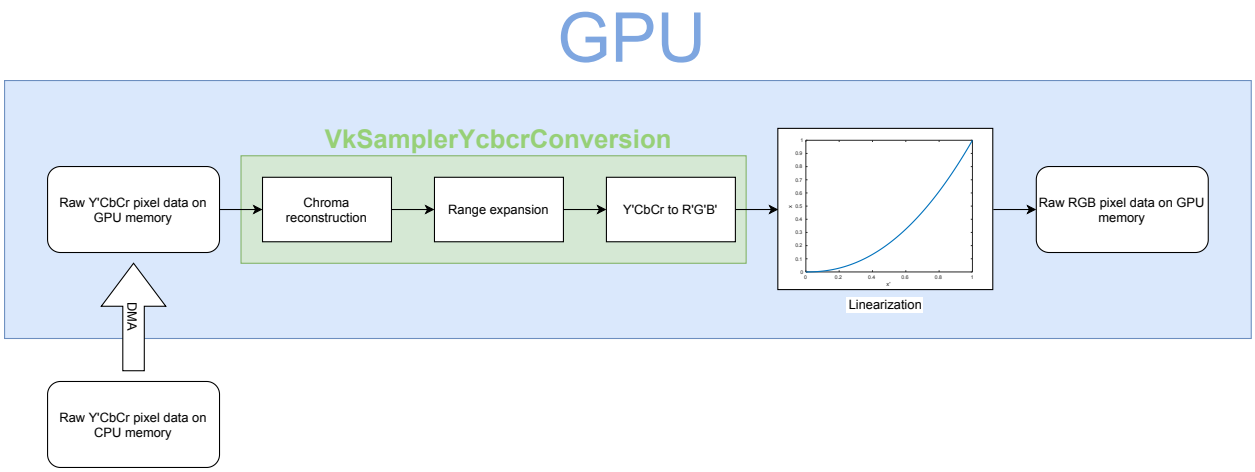


Figure 4.1: Y'CbCr to RGB conversion pipeline

Data upload

Independently of the source of the video signal, the starting point of this process is CPU's memory, where the source has deposited raw pixel data. This data may have an arbitrary bit-depth, multiplex, colour model, EOTF, etc... As the objective is to obtain raw linear RGB data, this buffer is uploaded to another buffer that lies on the GPU's own memory. The GPU is responsible of converting this untreated data to a easily manageable format. The data transfer is performed through a asynchronous direct memory access (DMA) operation, so that the CPU can handle other tasks meanwhile.

Chroma reconstruction

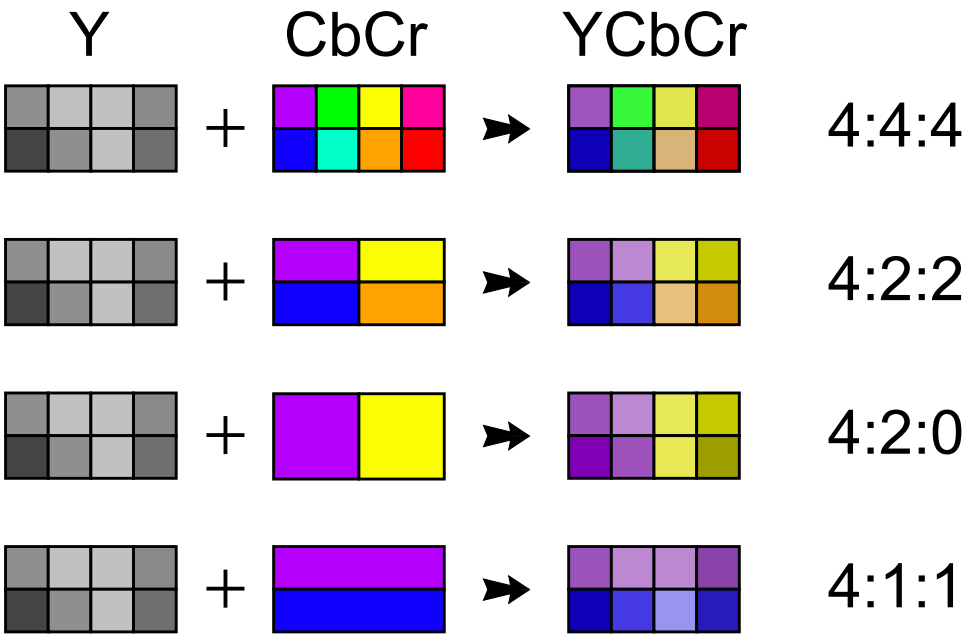
More often than not, when using Y'CbCr-like color models, chrominance components (Cb and Cr) are subsampled, this is, less values of them are stored, as shown in the figure 4.2. To represent this operation, the following notation is widely used:

<luma samples on any line>:< C_b, C_r samples on even lines>:< C_b, C_r samples on odd lines>

The following subsampling schemes are supported:

- 4:4:4 (no subsampling)
- 4:2:2
- 4:2:0
- 4:1:1
- 4:1:0
- 3:1:1
- 3:1:0

Moreover, there are several ways of aligning the chroma sample grid with the luma sample grid. This alignment is also taken into account. Considering these two parameters, the chroma samples are linearly interpolated to match the luma sample count.



Source: Wikipedia

Figure 4.2: Common chroma subsampling schemes

Range expansion

Due to legacy reasons, many video signals do not use the whole available range for their respective bit-depth. If this is the case, the components are expanded according to the ITU-R BT.601 standard[22]. If an alpha channel is present, it may bypass this step, depending on the information provided by the source. The aforementioned range expansion is performed with the equation (4.1) for $R, G, B, A, Y \in [0, 1]$ components and with the equation (4.2) for $C_b, C_r \in [-0.5, +0.5]$, where n is the bit-depth.

$$Y' = \frac{y' \cdot (2^n - 1) - 16 \cdot 2^{n-8}}{219 \cdot 2^{n-8}} \tag{4.1}$$

$$C' = \frac{c' \cdot (2^n - 1)}{224 \cdot 2^{n-8}} \tag{4.2}$$

Colour model conversion

If using a Y'CbCr-like colour model, this needs to be converted to R'G'B' (gamma corrected RGB). This is a simple linear transformation that is performed using a 3x3 matrix. The values of the matrix depend on the actual Y'CbCr specification. Obviously, if the source data is in RGB space, no conversion will be performed.

The following Y'CbCr specifications are supported:

- RGB (no conversion)
- YIQ
- YUV
- ITU-R BT.601[22]

- ITU-R BT.709[23]
- ITU-R BT.2020[24]
- SMPTE 240M[25]

If *Vulkan* supports the `VkSamplerYcbcrConversion` extension and this extension supports the requested configuration, the last three operations are executed by *Vulkan* itself. Otherwise, a fallback method that performs them on a fragment shader is available. Moreover, partially supported configurations may also take advantage of the extension.

Component linearization

The last step is to linearize the inverse EOTF applied to the pixel data. This is done because mixing gamma-corrected values leads to colorimetrically incorrect results. Alpha component bypasses this stage, as it is not gamma corrected.

The following EOTFs are supported:

- Linear (no linearization needed)
- ITU-R BT.1886[9]
- Gamma 2.2
- Gamma 2.6
- Gamma 2.8
- IEC61966-2-1[30]
- IEC61966-2-4[31]
- SMPTE 240M[25]
- SMPTE ST.2084[28]
- ARIB STD-B67[29]

The resulting RGB(A) image is stored on a *Vulkan* texture local to the GPU memory. All components are represented by floating-point numbers whose mantissa has more bits than the source component. In other words, if the source format has strictly less than 12 bits per component, a 16 bit floating-point (half precision) number will be used. Otherwise, 32 bits (single precision) will be used[32].

There are some exceptional cases where the aforementioned conversions are not performed. For those cases, the data is copied directly from the CPU to the definitive GPU buffer, preserving the original pixel format. These exceptions are the following:

(a) Source data is full-range multiplexed RGB(A) with no gamma correction. After all, none of the stages would be executed.

(b) Source data is full-range multiplexed RGB(A), with 8bits per component, IEC61966-2-1 encoding and *Vulkan* supports sRGB formats. In this case, all operations except the linearization are a no-op. When sRGB formats are supported, this linearization is performed on the texture mapping unit (TMU) of the GPU each time the texture is accessed.

The reverse pipeline has been partially implemented, but none of the features provided by this mixer make extensive use of it.

4.1.2 Video processing

Once a manageable video frame is on GPU memory, it is considered to be a texture. In the context of computer graphics, textures are the basic form of interpreting image data on GPUs. These textures are mapped to objects in a process known as UV-mapping. For this application, the whole texture will be assigned to a rectangle object.

This presents a new problem. If the aspect ratio of the rectangle does not match the aspect ratio of the frame, special consideration needs to be taken. Several solutions have been implemented and it is up to the user to choose one of them:

- **Stretch**: The image will be distorted to fit the new aspect ratio.
- **Box**: The image will be resized by the most restrictive axis.
- **Crop**: The image will be resized by the least restrictive axis.
- **Clamp horizontally**: The image will be resized by the horizontal axis.
- **Clamp vertically**: The image will be resized by the vertical axis.

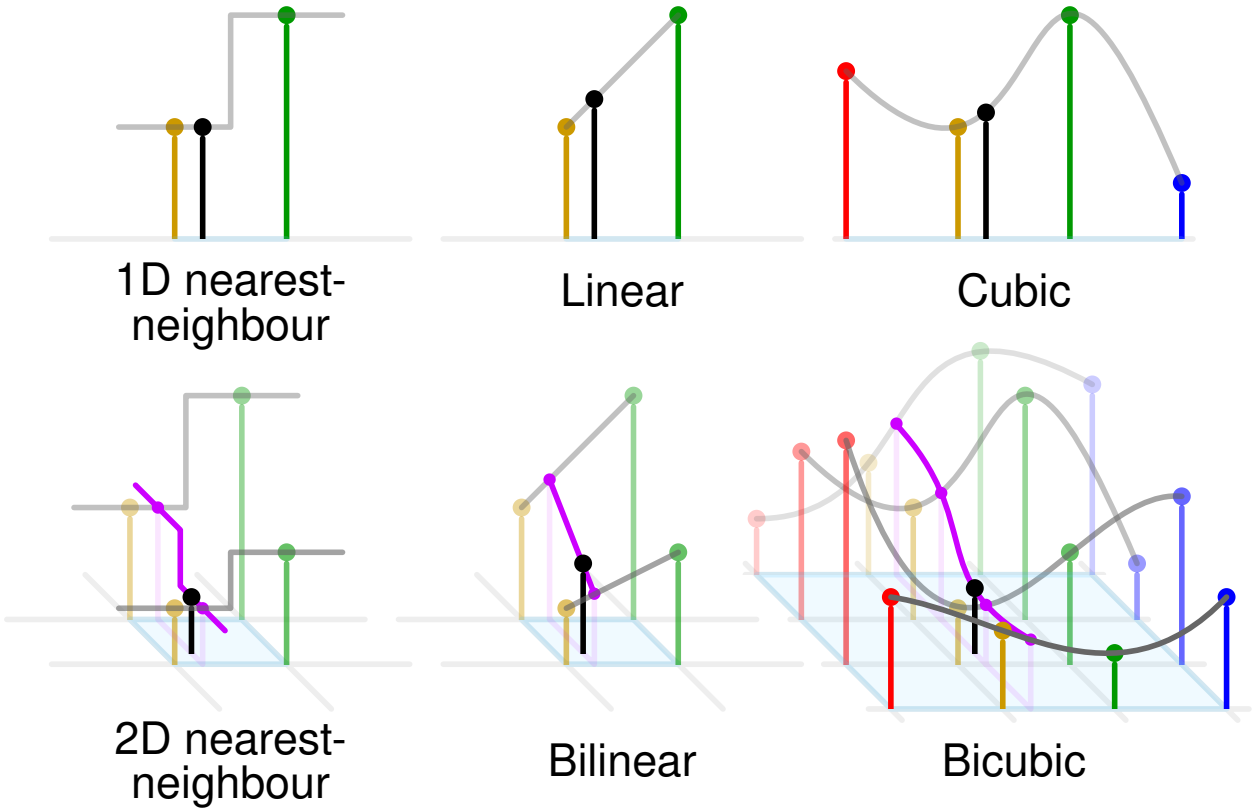
Another aspect regarding the textures is the sampling. When the aforementioned rectangle is rendered, each of its pixels will have to be assigned a color. This color comes from the texture, but it may correspond to a intermediary position between texture elements (texels). Therefore, neighbouring texels need to be interpolated to obtain the actual color. This interpolation must be performed with linear color values. Luckily, in the previous section, this premise was ensured. The interpolation techniques illustrated in the figures 4.3 and 4.4 have been implemented.

Using nearest filtering is the cheapest option, as it requires a single texture access per sample. Bilinear filtering requires 4 accesses and 16 for bicubic filtering. Therefore, the later one is the costliest one and should not be used indiscriminately.

From the implementation point of view, nearest filtering is always performed by *Vulkan*. In most cases, bilinear filtering is also executed by *Vulkan* natively, but if this is not possible, a manual fallback method exists that implements it using 4 nearest accesses. Lastly, *Vulkan* rarely supports bicubic filtering, but if this is the case, the `VK_IMG_filter_cubic` extension is used. For the rest of the cases, it is implemented through 16 nearest accesses or 4 bilinear accesses. The later technique is based on a publicly available algorithm[33].

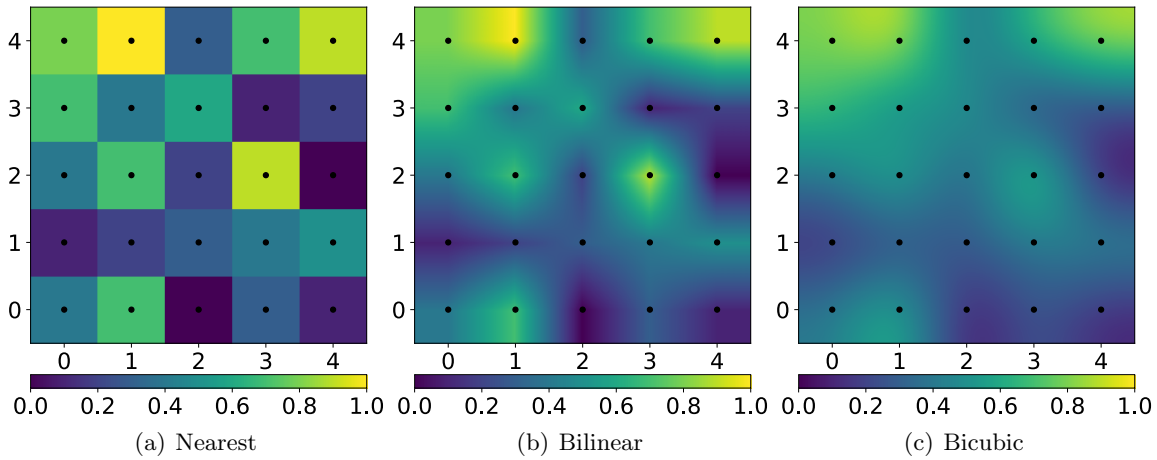
The resulting object is embedded into the framebuffer, the memory area where the GPU works on. If the object presents transparency information, either because the original frame does or the processing has added to it, it will be blended with the existing contents in the framebuffer according to one of the following compositing modes. An example of each one of them can be seen in the figure 4.5

- **Write**: The framebuffer will be overwritten without considering its contents: $C = C_{object} \cdot \alpha$
- **Opacity**: Weighted average of the foreground and background colours. It is the most common blending mode: $C = C_{object} \cdot \alpha + C_{framebuffer} \cdot (1 - \alpha)$
- **Add**: Additive blending. The resulting value may be saturated: $C = C_{object} \cdot \alpha + C_{framebuffer}$



Source: Wikipedia

Figure 4.3: Interpolation filters



(a) Nearest (b) Bilinear (c) Bicubic

Source: Wikipedia

Figure 4.4: Texture sampling filters

- **Inverse difference**: Subtracts the object's value to the existing colour. Useful to perform comparisons $C = C_{framebuffer} - C_{object} \cdot \alpha$
- **Difference**: Subtracts the existing value to the object's colour. Useful to perform comparisons: $C = C_{object} \cdot \alpha - C_{framebuffer}$
- **Darken**: The resulting colour will have the darkest components of the foreground and the background: $C = min\{C_{object} \cdot \alpha, C_{framebuffer}\}$
- **Lighten**: The resulting colour will have the brightest components of the foreground and the background: $C = max\{C_{object} \cdot \alpha, C_{framebuffer}\}$
- **Multiply**: The resulting colour will be the multiplication of the foreground and the background: $C = C_{object} \cdot \alpha \cdot C_{framebuffer}$
- **Screen**: Multiplication of complementary values: $1-C = (1-C_{object} \cdot \alpha) \cdot (1-C_{framebuffer}) \Leftrightarrow C = C_{object} \cdot \alpha + C_{framebuffer} \cdot (1 - C_{object} \cdot \alpha)$

4.2 Server and communications

The server itself has been developed using the previously described libraries. This server makes use of them to expose a CLI that resembles a vision mixer.

This mixer is able to take live video feeds from NDI sources or pre-recorded video files. Several heterogeneous M/Es can be configured to manipulate those sources. Moreover, these M/Es can be configured with arbitrary video parameters, such as nonstandard resolutions. The resulting video feeds can be outputted through a OS window or used as an input for other M/Es. The windows can be configured to be full-screen on a monitor attached to the host computer, being able to use the video outputs of the computer.

4.2.1 Architecture

The relations among the most crucial classes of the mixer have been illustrated in the figure 4.6. A brief explanation about the duty of each one of the classes shown in this figure is described hereunder. The architecture can be divided into two parts. The first one relates to the infrastructure needed to configure the video manipulation. This includes all the classes owned by the Mixer class. The other part relates to the control of the former classes, this is, the classes owned by the Controller class.

The mixer has been architected to allow easy expansion of both parts. This expansion can be done by defining new classes that inherit from the base classes.

Mixer

The primary purpose of this class is to hold all elements that in some way interact with video signals. This includes inputs, outputs and M/E banks. These elements can be dynamically constructed/destroyed on demand. They are held in a hash table where the key is provided by the unique name assigned to each of the elements. Moreover, it also serves as a signal matrix, as it has methods to route any output of any element to one of the inputs of another element.

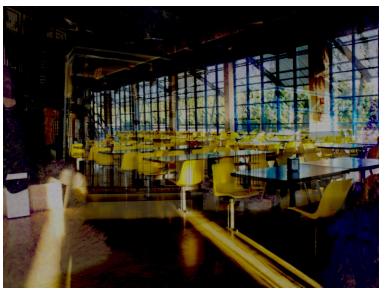
(a) Background

(b) Foreground

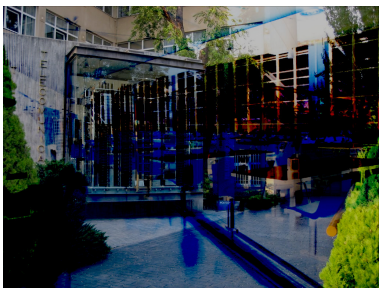
(c) Write

(d) Opacity

(e) Add



(f) Inverse difference



(g) Difference

(h) Darken

(i) Lighten

(j) Multiply

(k) Screen

Figure 4.5: Results of performing the blending operation with $\alpha = 0.5$ and all supported modes.

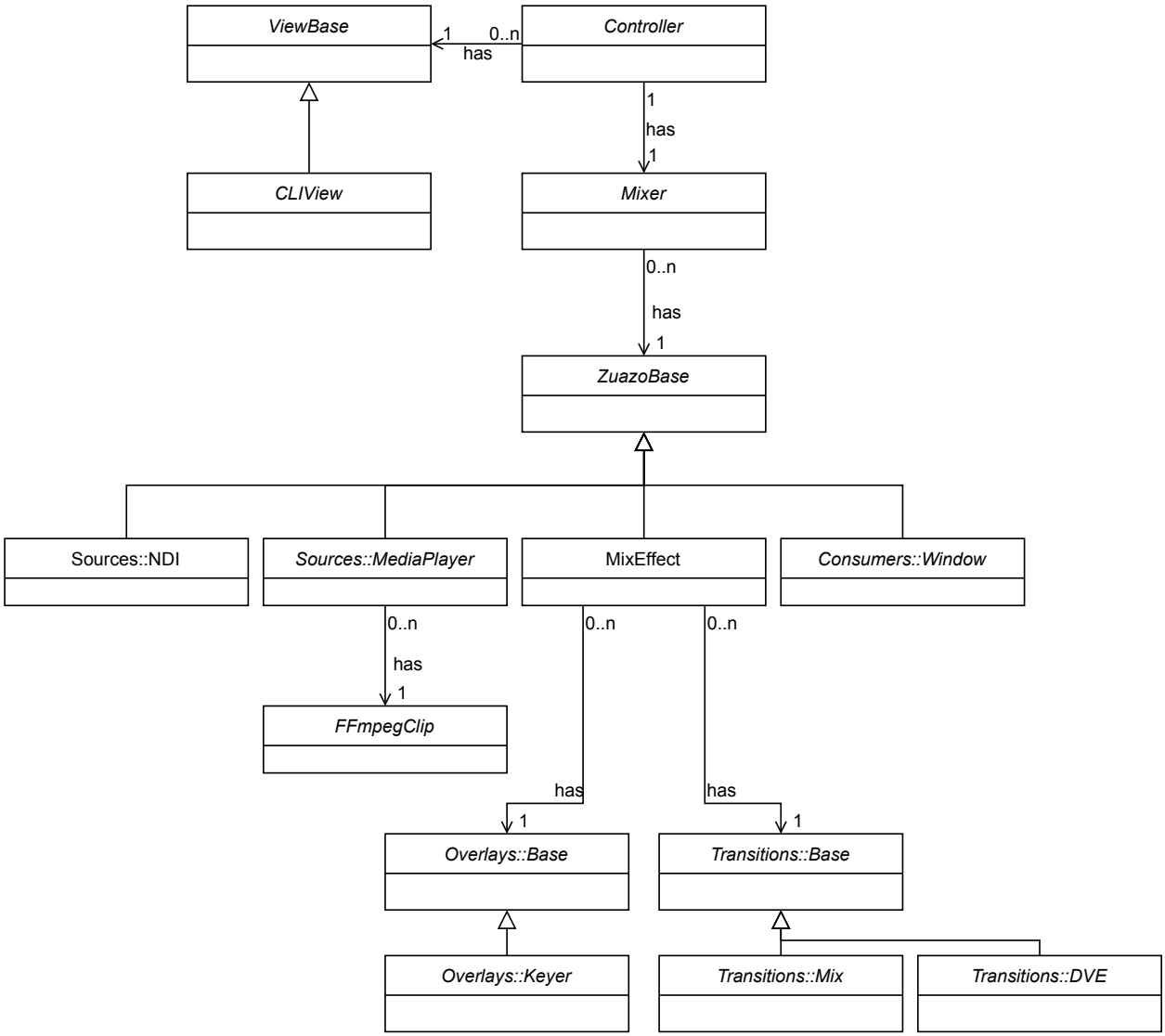


Figure 4.6: Software architecture of the server side of the mixer

ZuazoBase

Strictly speaking, this class belongs to the *Zuazo* library, but it is described here because it serves as the base class for all the classes regarding video manipulation. The hash table of the previous element holds instances of this class.

Sources::NDI

This class provides a NDI source as a input for the mixer. The one available NDI feeds is selected as the source.

Sources::MediaPlayer

Similarly to the mixer class, this class holds a hash indexed list of pre-recorded video clips. One of those clips is selected to be in the output. New clips can be loaded/unloaded dynamically.

FFmpegClip

These is the class held by the MediaPlayer class. It holds all the infrastructure needed to play a pre-recorded video file using the *FFmpeg* library. This library is very versatile and allows to play almost all imaginable video containers and codecs. If possible, it will take advantage of hardware video decoding acceleration.

The wrapper of this library provided by *zuazo-ffmpeg* allows to set different repeat modes, play speeds, etc...

Consumers::Window

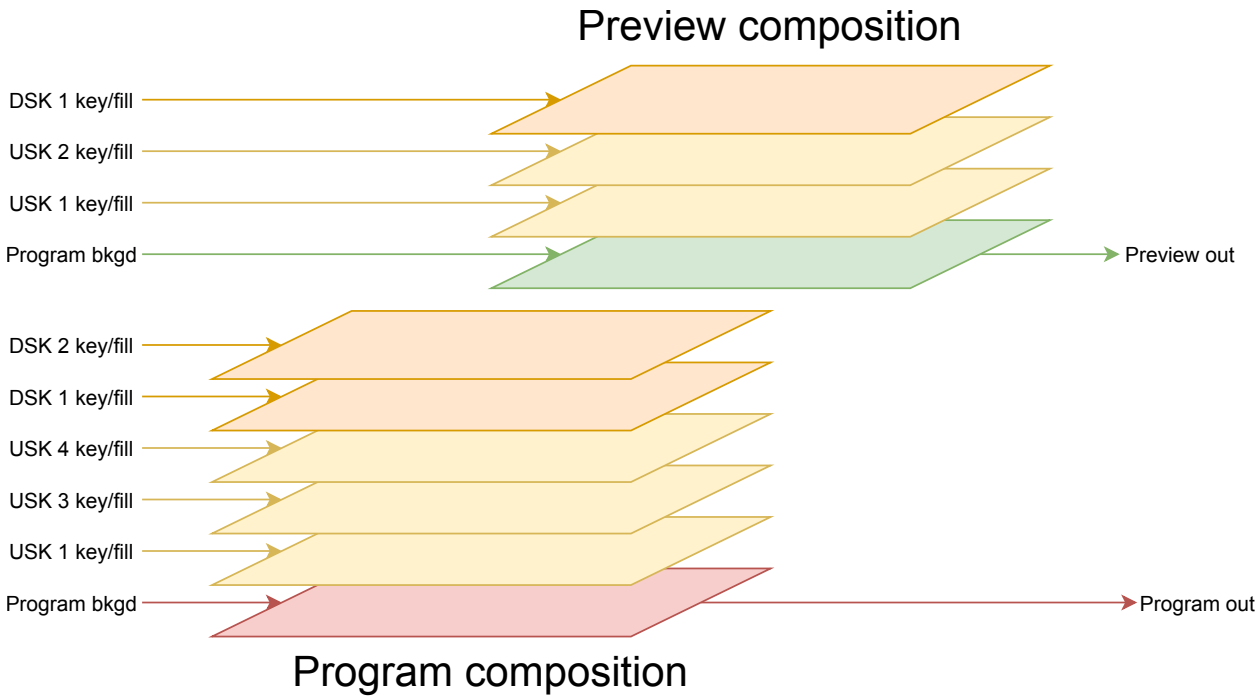
This class allows to create a window for showing video. This window can be configured to be full-screen on a monitor connected to the computer. Moreover, several other parameters such as window decoration, opacity and resizability can be tweaked.

MixEffect

Currently, this is the only class that manipulates video. The video compositing is performed using four instances of the `Compositor` component from the *zuazo-compositor* module.

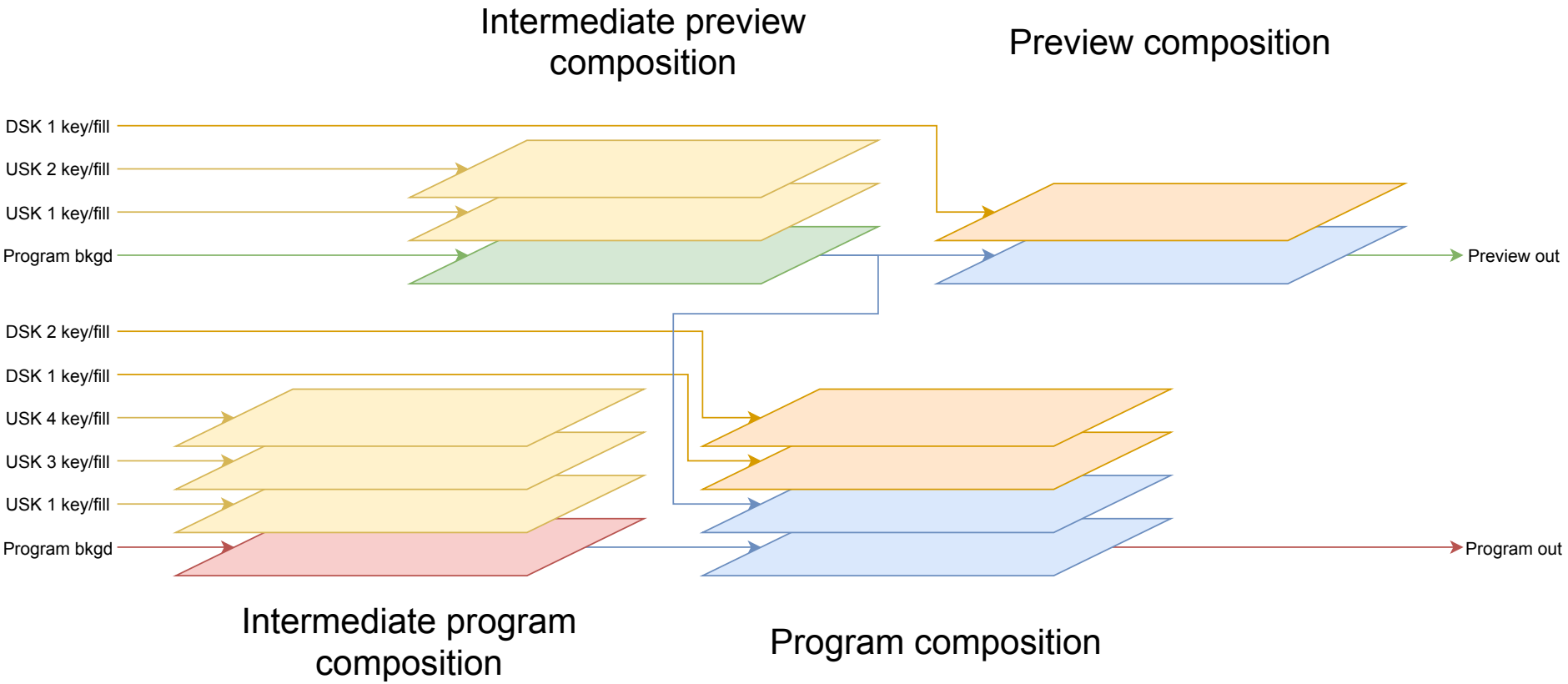
As shown in the figure 4.7, generally, only two of the compositors are used, one for each output bus. These compositors render the background with all the overlays on top. Indeed, the overlays are ordered according to their priority, this is, high indexed overlays occlude lower indexed ones and DSKs occlude USKs.

However, if a transition is in progress, this behaviour changes, as DSKs must be applied after the transition. Therefore, the other two compositors are used to generate a intermediate image that contains the background with all the USKs. Both program and preview intermediate images are used to generate the transition on the configured output bus, as displayed in the figure 4.8. The other output bus is also composited but without the transition effect.



Layer legend: Program background Preview background USKs DSKs

Figure 4.7: Compositing layers when no transition is in progress



Layer legend: Program background Preview background USKs DSKs Transition

Figure 4.8: Compositing layers when a transition is in progress in the program bus

Overlays::Base

This class serves as the base class for overlays. It exposes the interface required by the mix effect to perform the compositing. Currently, the only class that implements this interface is the `Keyer` but other overlays such as 3D models can be implemented to enhance the functionality of the M/Es.

Overlays::Keyer

As mentioned earlier, this is the only class that implements the `Overlay` interface. This has one of the most advanced implementations, hence, it will be explained in depth later in its own section. In short terms, this allows to embed images on top of the background layer.

Transitions::Base

Similarly to the overlays, all transitions also inherit from a base class. However, the transitions are not a single layer but a collection of layers. Usually they consist of two layers, the in-coming and out-going layers. Two classes implement this interface: `Mix` and `DVE`

Transitions::Mix

This transition has both layers encompassing the whole viewport, similarly to the background layers. These two layers are progressively blended using one of the following techniques:

- **Mix**: Both layers are weight-averaged using the mixing equation described on the chapter 2.
- **Add**: Both layers are added together. The first half of the transition, the foreground layer is attenuated, while in the second half, the background layer will be.
- **Fade**: In the first half the background layer fades out and in the second half the foreground layer fades in.

Transitions::DVE

As the name suggests, this is a DVE-based transition. Unlike the `Mix` transition, the layers used to create the `DVE` are not generally rendered full-screen. Several variations of this transition have been implemented:

- **Cover**: The incoming image appears from one of the edges occluding the outgoing image. The travel direction can be determined by a continuous angle.
- **Uncover**: The outgoing image disappears towards one of the edges, uncovering the incoming image. The travel direction can be determined by a continuous angle.
- **Slide**: Both of the previous effects are executed at the same time. Therefore, it can be considered that both images move together, as if they were welded.
- **Rotate 3D**. The outgoing image starts rotating around a user-configured axis that is parallel to the screen. When the image is normal to the screen, the rotation continues but with the incoming image.

Control::Controller

The controller receives the commands from the `ViewBase`s. Then the `Controller` is responsible of applying the changes requested by the command, responding to the invoker `ViewBase` and broadcasting the mutations to the rest of the `ViewBase`s. These messages are interchanged as a list of strings, where each of the elements represents a token.

Control::ViewBase

A `ViewBase` represents a manner of exporting and importing commands from and to the mixer. Therefore, as mentioned earlier, it will request to the mixer mutations or queries, hear a response back and react to state changes.

Control::CLIView

This class inherits from `ViewBase` and exposes the mixer as a CLI. Therefore, its responsibility is to perform the above-mentioned tasks as a plain text interface.

4.2.2 Keying

The keying operation involves the implementation of a low-level *Zuazo* component. Therefore, it is worth dedicating a section to this process. It has been implemented using a OpenGL Shading Language (GLSL) fragment shader. In the context of digital image processing, all the underlaying operations are point operations, so the keying process is also a point operation.

In the figure 4.9 it can be seen that the keyers take as input three frames. The first one is the existing contents of the framebuffer, as the resulting image will be overlayed on top of it. The second and third ones are keying and filling images. The opacity of the later one will be determined based on the former one. This opacity will be used to select between the filling image and the background according to the blending modes described earlier. This process is executed repeatedly for each keyer. The *mix out* of the first one will be the *background in* of the next one, and so on. The first keyer will receive as the background input the actual background layer.

The keying signal can determine the opacity of the filling signal in the traditional three manners: Linear keying, luma keying and chroma keying. Moreover, a pattern and a global gain can also be used as the key source. These five opacity values are multiplied together and used as the α source in the blending stage. Any set of these five transparency sources can be used. If not enabled, it can be considered that a '1.0' is fed to the multiplier.

Linear key

Linear keying refers to the use of an existing component as a transparency source for the fill signal. In this case, any of the R, G, B, A and Y components of either key or fill signals can be used as a transparency source. Moreover, the result can be complemented. As the source image is in RGB(A) colour model, in case the Y component is used, it needs to be computed according to the YUV/YIQ formula shown in the equation (4.3).

$$Y = 0.299R + 0.587G + 0.114B \tag{4.3}$$

The most common usage would be to use the A component of the filling image when this presents an alpha channel. Moreover, if the alpha is provided as a separate matte signal, the logical configuration would be to use the keying signal's luminance.

Luma key

Luma key is very similar to a linear key configured to listen to the keying signal's luminance. Similarly to the previous keying technique, it implements an optional result complementation.

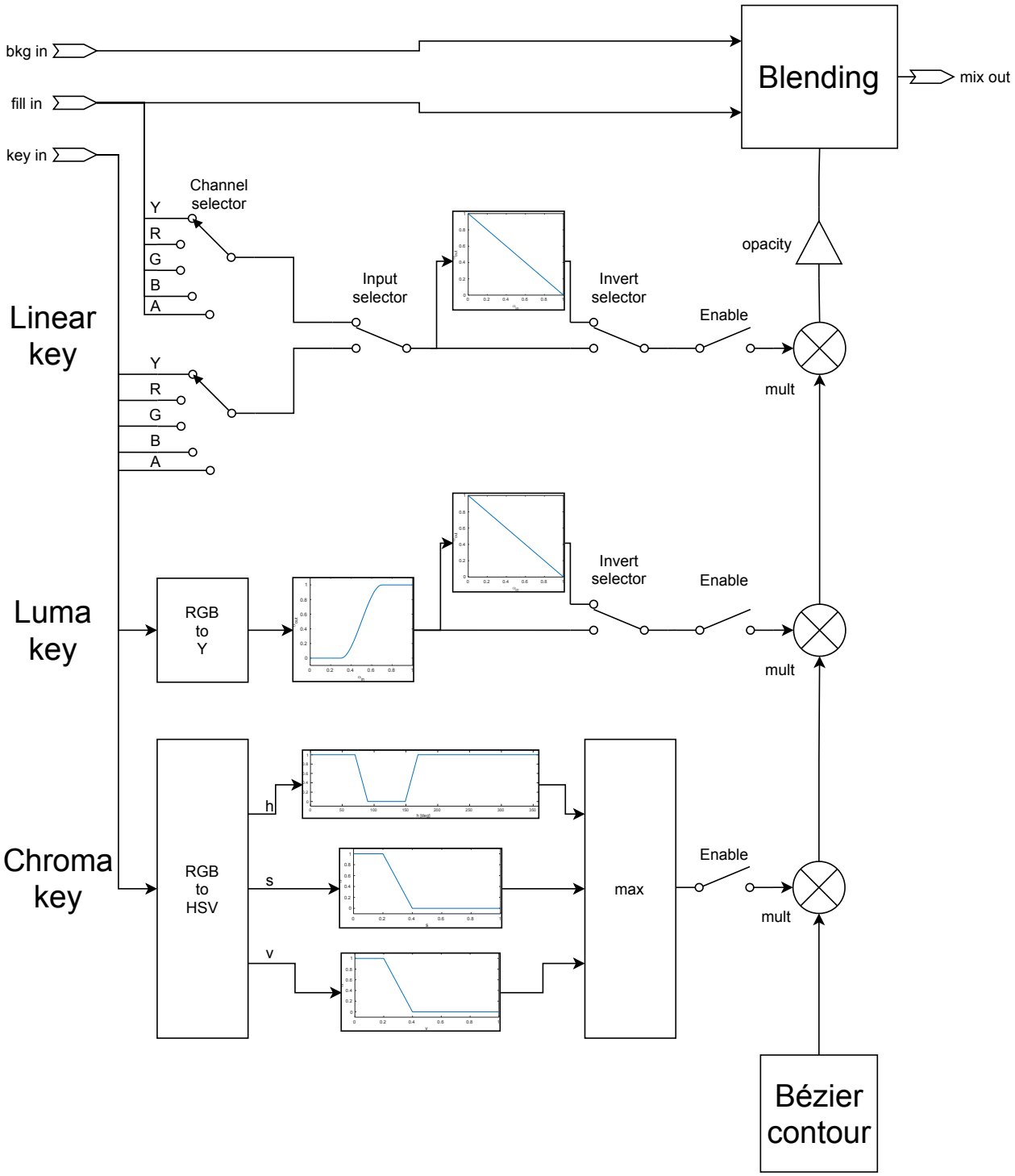


Figure 4.9: Signal path of the keyer

The only difference with this approach is that the luminance to transparency conversion is not performed linearly. Instead, a transfer function is applied to it. This transfer function is a Hermite function, which can be configured with a lower and upper threshold, as shown in the figure 4.10. If $max \leqslant min$, this transfer function is a umbralization at min.

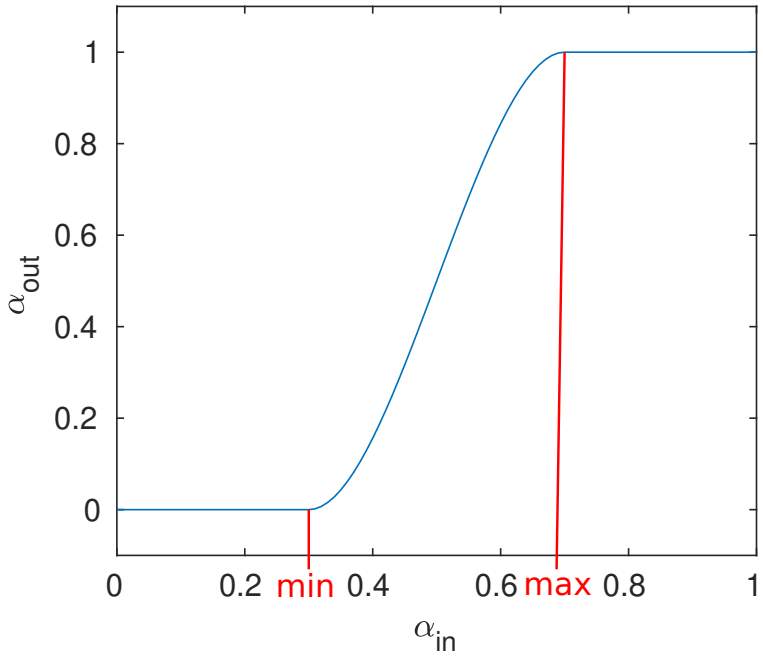


Figure 4.10: Luma-key's transfer function

Chroma key

Chroma keying must be done in a colour model where the colour data is explicit, as this type of keying consists in using the tonality to discriminate parts of the filling image. In this case, the Hue-Saturation-Value (HSV) color model has been used. As a consequence, the first step is to convert the source pixels to this model using the equations (4.4), (4.5) and (4.6). Although those equations may seem to be trivial to implement, they have many branches, which considerably hurts performance. Therefore, a highly optimized version written by the authors of *LOLengine* has been used[34].

$$MAX = max\{R, G, B\} \qquad MIN = min\{R, G, B\}$$

$$H = \begin{cases} 60° \cdot \frac{G-B}{MAX-MIN} + 0° & \text{if} \quad MAX = R \quad \text{and} \quad G \geqslant B \\ 60° \cdot \frac{G-B}{MAX-MIN} + 360° & \text{if} \quad MAX = R \quad \text{and} \quad G < B \\ 60° \cdot \frac{B-R}{MAX-MIN} + 120° & \text{if} \quad MAX = G \\ 60° \cdot \frac{R-G}{MAX-MIN} + 240° & \text{if} \quad MAX = B \end{cases} \tag{4.4}$$

$$S = \begin{cases} 0 & \text{if} \quad MAX = 0 \\ 1 - \frac{MIN}{MAX} & \text{otherwise} \end{cases} \tag{4.5}$$

$$V = MAX \tag{4.6}$$

HSV colour model allows to easily implement the chroma key, detecting if the hue lies on a particular interval, as shown in the figure 4.11. This interval is configured by a central value and

width. Moreover, the edges can be smoothed to steadily fade from transparent to opaque. Due to the periodicity of the hue component, the transfer function also has this nature, wrapping from 360° back to 0°.

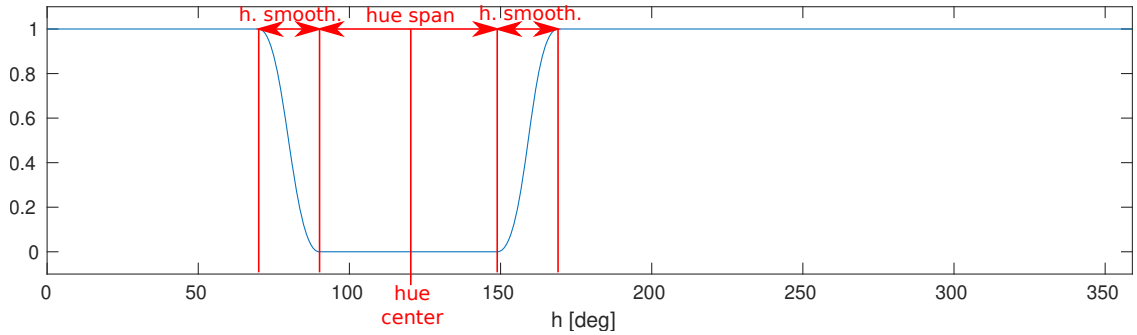


Figure 4.11: Chroma key's hue transfer function

However, the hue component is not well defined for low saturation and brightness values, as it gets very sensitive to noise. To solve this problem, a threshold is applied to the saturation and brightness components. Similarly to the hue component, this threshold can be smoothed as seen in the figures 4.12 and 4.13.

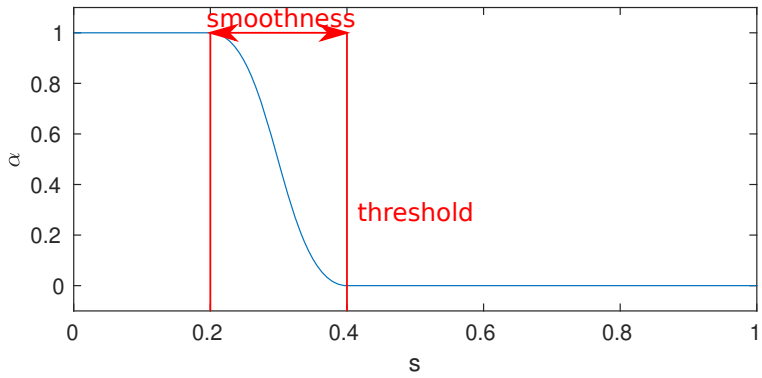


Figure 4.12: Chroma key's saturation transfer function

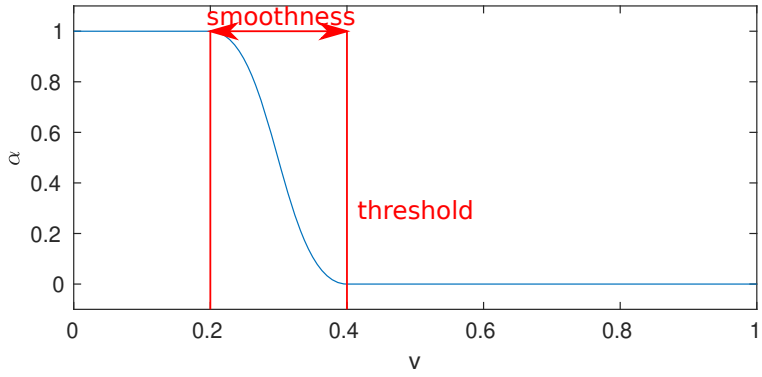


Figure 4.13: Chroma key's brightness transfer function

Finally, the most restrictive of the three transparency values is selected, namely, the maximum. In this way, low saturated or dark pixels avoid to be transparent due to the restriction imposed by the saturation and brightness transfer functions. Similarly, pure colors (maximum saturation and

brightness) far away from the discriminated color will not be transparent due to the hue transfer function.

Pattern key

The pattern keying can be configured to use multiple Bézier contours to crop the filling image with an arbitrary shape. This shape is completely user definable. Implementation details about the techniques used to achieve such a task are in the Appendix A.

4.2.3 Communications

The mixer is designed to be remotely controlled using network protocols. The control is performed using a CLI, which can be transported either over TCP or WebSockets. The first protocol has been selected as it is widely used and it is easy to develop new applications with it. The second protocol allows interacting with web applications and works as a thin layer on top of HyperText Transfer Protocol (HTTP), as shown in the figure 4.14. The operation of the CLI has three principles:

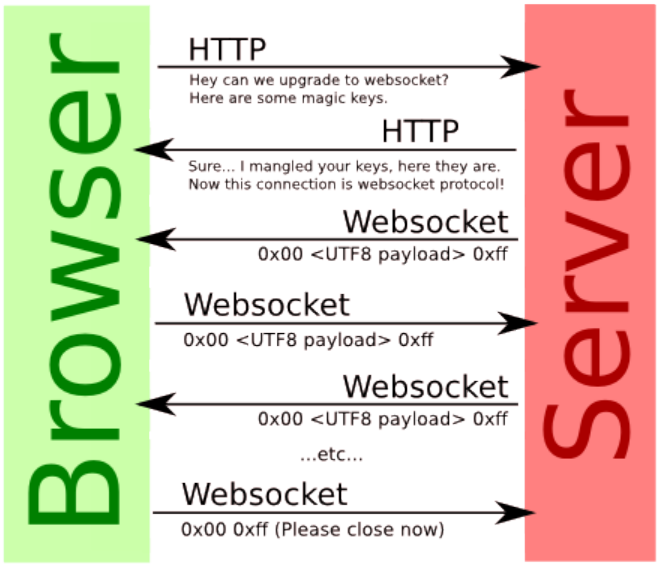


Figure 4.14: WebSocket handshaking process

1. Each of the commands can be considered as a chain of space separated tokens.
2. All sent commands will receive a response from the server. This response will be either success (`OK`) or failure (`FAIL`) and may contain a tokenized payload afterwards. If a token starting with `#` followed by a identifier is prepended to the request, this token is also prepended to the response. This allows to have multiple requests in flight.
3. If a command mutates the state of the mixer, this command is broadcasted to all connected clients, including the invoker. However, the broadcast will suppress the identifier token.

In-depth details about this CLI are enclosed in the mixer's manual.

This control paradigm offers great flexibility for setting up multiple control environments. Some example configurations are described hereafter. Note that they are just examples and not hermetic

configurations. Users may decide to vary and combine them.

Basic configuration

In most cases, none of the advanced control features is used. In this case, the server is controlled by a single client, as shown in the figure 4.15.

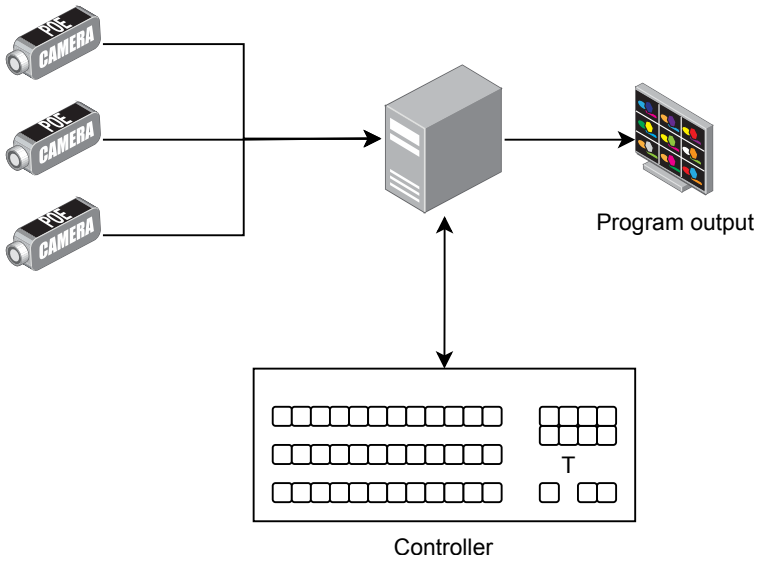


Figure 4.15: Basic mixer control configuration

Multi-program configuration

Some events may require to produce multiple audiovisual feeds. For instance, if a large concert is going to be broadcasted on TV, two signals may be generated, one for TV use and the other for image magnification inside the venue. These are usually two distinct mixes, as the image magnification feed usually focuses on close shots of the performers, while the TV signal has more general shots. Instead of using two separate vision mixer, this task can be accomplished with a single mixer with two M/E banks.

However, performing two distinct mixes may overwhelm even the most experienced producers, so two producers, each one with its own controller, are required. The control mechanism allows to point each controller to each of the M/Es, so they do not mess the work of each other. This configuration is displayed in the figure 4.16.

Redundant configuration

The last example relates to the possibility to synchronize the states of two mixers using a proxy client. This allows to implement a fail-over mechanism in case of one of the controllers or the mixers fail. This configuration is shown in the figure 4.17 and allows a catastrophic failure of any of the sides of the mirror. If this happens, the video feed will be briefly interrupted until the master switch is toggled. These switches are alien to this project, but they are commercially available.

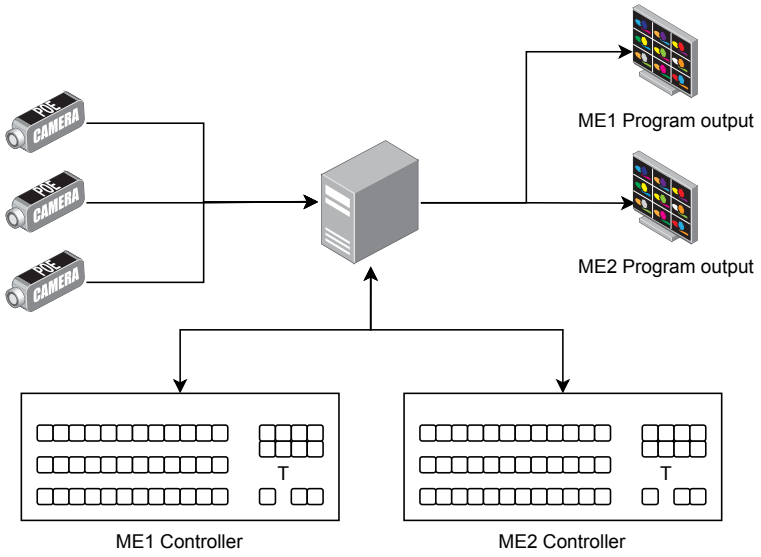


Figure 4.16: Multi M/E mixer control configuration

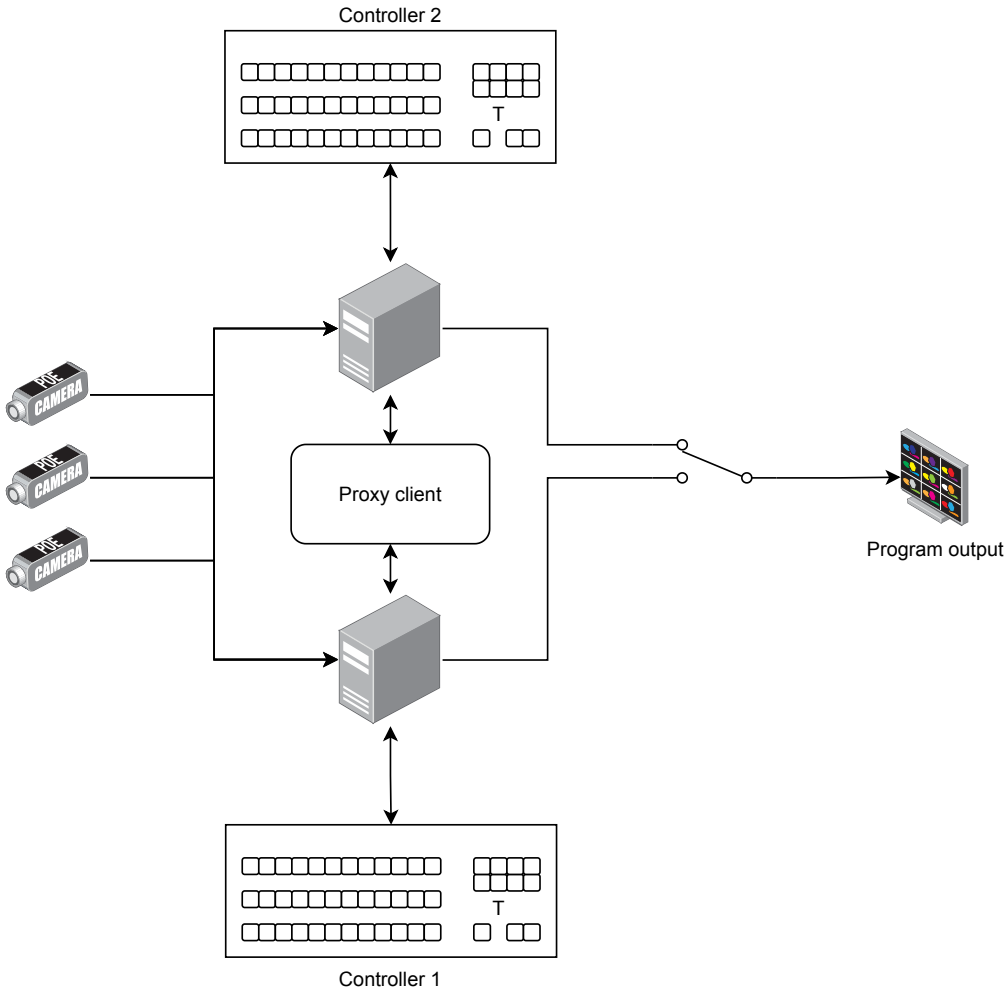


Figure 4.17: Redundant mixer control configuration

4.3 Web application for control

The main client implemented for controlling this mixer is a web application. Web applications are executed by browsers and generally do not require installation, as they are downloaded from the internet on demand. Moreover, they use interpreted programming languages, so they can be executed on any desktop platform, or even mobiles and tablets. This makes them a very versatile multi-platform development tool, and the number of web applications has sky-rocketed in the last decade. In fact, this report has been written in a web application.

Web applications come with their own drawbacks. In general terms, they are not as efficient as compiled languages, making them unsuitable for compute intensive applications. However, in this case, all the heavy lifting is done in the server and the task of controlling it is very lightweight, so this concern is negligible. Another inconvenience of web applications is that they require an internet connection to work. This can be solved using *Electron*, a framework to bundle web applications with an instance of *Chromium* (FOSS version of *Google Chrome*) which serves as a standalone package of the application that works offline.

This controller has been built using *Vue.js* and *Bootstrap* frameworks. The former one relates to the development of the user interface (UI) itself. The later one is a CSS library to obtain a consistent Material Design (MD) look. Additionally *vue-router* has been used to create multiple pages and *vuex* for state management.

To communicate with the server, the CLI has been implemented in JS. This commands are carried from and to the mixer using WebSockets, as this is one of the few real-time protocols widely supported by browsers. This implementation of the CLI is wrapped around a custom *vuex* plugin that synchronizes the state stored in *vuex* with the mixer's status.

GitHub-pages has been selected to deploy the web application, as it is free, easy to use and integrates well with the *Git* version control used to develop this application. However, if desired, the resulting web page can be easily hosted in any other hosting service.

Chapter 5

Results

To evaluate the degree of success in the execution of this project, several measurements have been carried out. These measurements evaluate several aspects that directly influence the usability of the final product.

5.1 Reliability

As vision mixers are a central element in video production facilities, it is crucial to keep them in operation, as any downtime compromises the broadcast. The reliability of the implemented mixer has not been objectively measured, but all crashes have been logged and investigated.

Subjectively speaking, the mixer has proven to be pretty reliable. Although a few crashes have occurred, most of them where when performing preoperative settings and not when using it. Some examples of these preoperative settings include operations like adding M/Es or keyers, loading invalid clips, etc... This is logical, as many of these operations are made up of a lot of complex tasks.

5.2 Latency

The latency is the time delay from the occurrence of a stimulus to the response of that same stimulus. Several estimations and calculations can be done to infer the latency of some of the systems related to this project. The latencies are going to be measured for a typical set-up where the video comes from a NDI camera and once processed, it is outputted to a display attached to the computer. A general idea of the equipment involved in the test is illustrated in the figure 5.1. Note that many latency sources such as the speed of light have been neglected.

Two latencies are going to be measured. The first one will be the time that it takes for a stimulus originated in front of the camera to take effect in the monitor, this is, the latency of the whole pipeline. The second latency is related to the time delay from an event in the web application to its reaction in the monitor.

5.2.1 End-to-end latency

The latency from the camera to the display will be called end-to-end latency. To measure it, the camera will be pointed at the display itself, so that the event generation is tied to the event perception. Using half of the monitor to generate the events and the other half to show the response, the latency can be accurately measured. In this case, the event generator will be a chronometer.

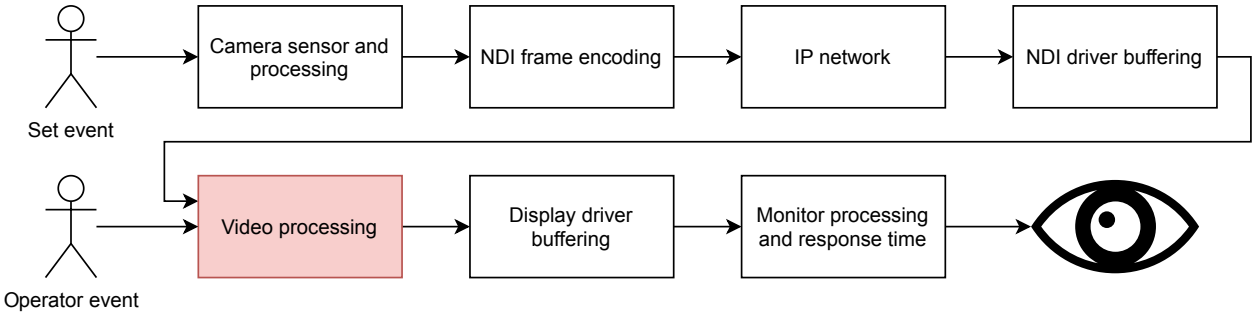


Figure 5.1: Latency sources

An example of this measurement is displayed in the figure 5.2.

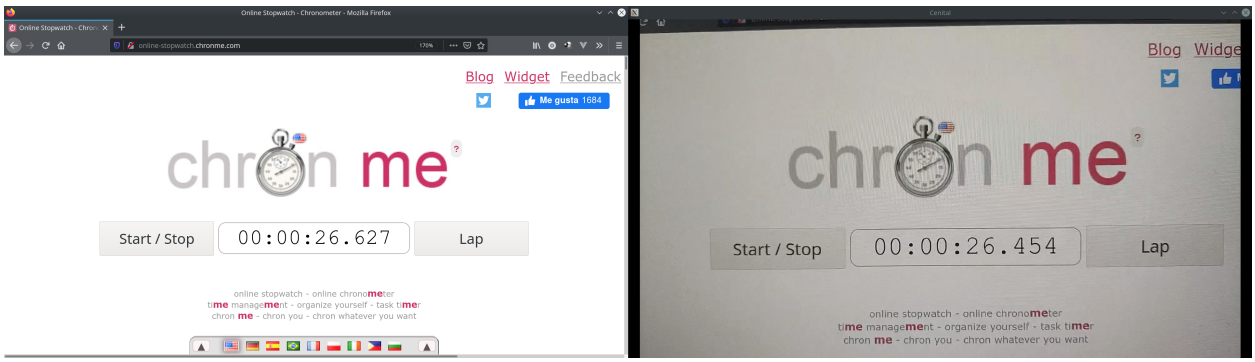


Figure 5.2: End-to-end latency measurement example

10 samples have been taken, obtaining an average latency of $\Delta t = 173.4$ms with a standard deviation of $\sigma^2 = 1.1$ms. These values are represented in the box plot 5.3. During these measurements, the system was working with a framerate of 30Hz. Therefore, this latency is equivalent to $0.173\text{s} \cdot 30\text{frames/s} = 5.19\text{frames}$. In chapter 2 it was mentioned that commercial solutions have a latency of 2 frames, but this is regarding the mixer itself.

Although these numbers are high and noticeable by the human brain, it must be considered that the measured latency is for the whole pipeline and not only for the mixer implemented in this project. Moreover, the NDI signal was being transmitted across WiFi, which is not ideal. As frames are processed in a sequential manner, the delay that is attributed to the video processing itself on the mixer is less than one frame. However, this is only true when the mixer is not overloaded. As a positive sidenote, the jitter in the latency is very low.

5.2.2 Control latency

This latency relates to the time between a button-press in the web application and the corresponding action in the display. The measurement of this delay is different to the previous one. In this case, the monitor was filmed with a high speed camera at 240Hz and frames between the appearing of the *glow* around the button and the response were counted. This method does not account for the latency introduced by the monitor, as both starting and ending events carry it, so they cancel each other when subtracting.

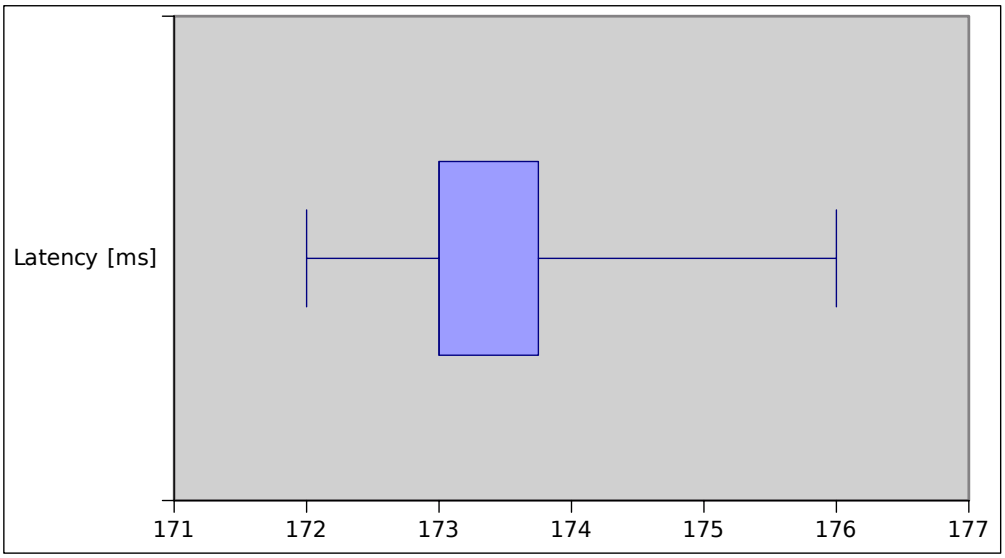


Figure 5.3: End-to-end latency measurement distribution

This latency has turned out to be less than one frame of the mixer, probably due to the fact that the web control software was running in the same computer as the server. More precise measurements can not be carried out with this equipment.

5.3 Chroma key quality

One of the most criticized aspects of commercial vision mixers is the quality of the chroma key effect. To test the results of the implemented mixer, two tests have been carried out. They were performed with the green screen of the *Laboratorio de Vídeo de la Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación (ETIST)-Universidad Politécnica de Madrid (UPM).*

The first test uses a teddy bear with white, black and red features, as shown in the figure 5.4. The green screen encompasses the whole field of view of the camera, so a simple chroma key is enough. Several spots of the green screen have deep shadows, which is a worst case scenario. Moreover, the green screen casts a greenish reflection on the white parts of the teddy bear, which increments the chances of rejection of these parts. In spite of all these inconveniences, the result is very acceptable, despite having some defects around some shadows in the lower part of the image and the right side of the bear. In addition, some white spots on the toes of the bear are also wrong, as they are transparent when they should not.

The second test is more demanding than the previous one. The new foreground is not clean, as it includes parts of the floor that are not green. Moreover, some parrots in which we are not interested have entered into the shot, as shown in the figure 5.5. To solve this, an elliptical mask has been designed with Bézier curves. This mask is used to trash everything but the subject and the green surroundings. After applying it, a chroma-keyable image is left. However, this is still a very demanding task, as the picture is full of shadows and the silver color of the tripod reflects its surrounding green environment. After applying the chroma key, the shadows are successfully deleted, but one of the tripod's legs is also amputated due to the previously mentioned green reflection. Moreover, the green accent on the camera lens is also deleted.

(a) Background image

(b) Foreground image

(c) Composited image

Figure 5.4: First chroma keyed image

(a) Background image

(b) Foreground image

(c) Masked foreground

(d) Composited image

Figure 5.5: Second chroma keyed image

5.4 Performance and throughput

The computer hardware primarily used by this project is shown in the figure 5.6. Depending on the use case of the mixer and the nature of each one of the components, bottlenecks may appear. These bottlenecks will prevent the mixer from functioning in real-time. Therefore, it is worth mentioning them to properly scale the hardware to the needs. The points described hereafter relate each of the mixer's parameters to the components that it may saturate.

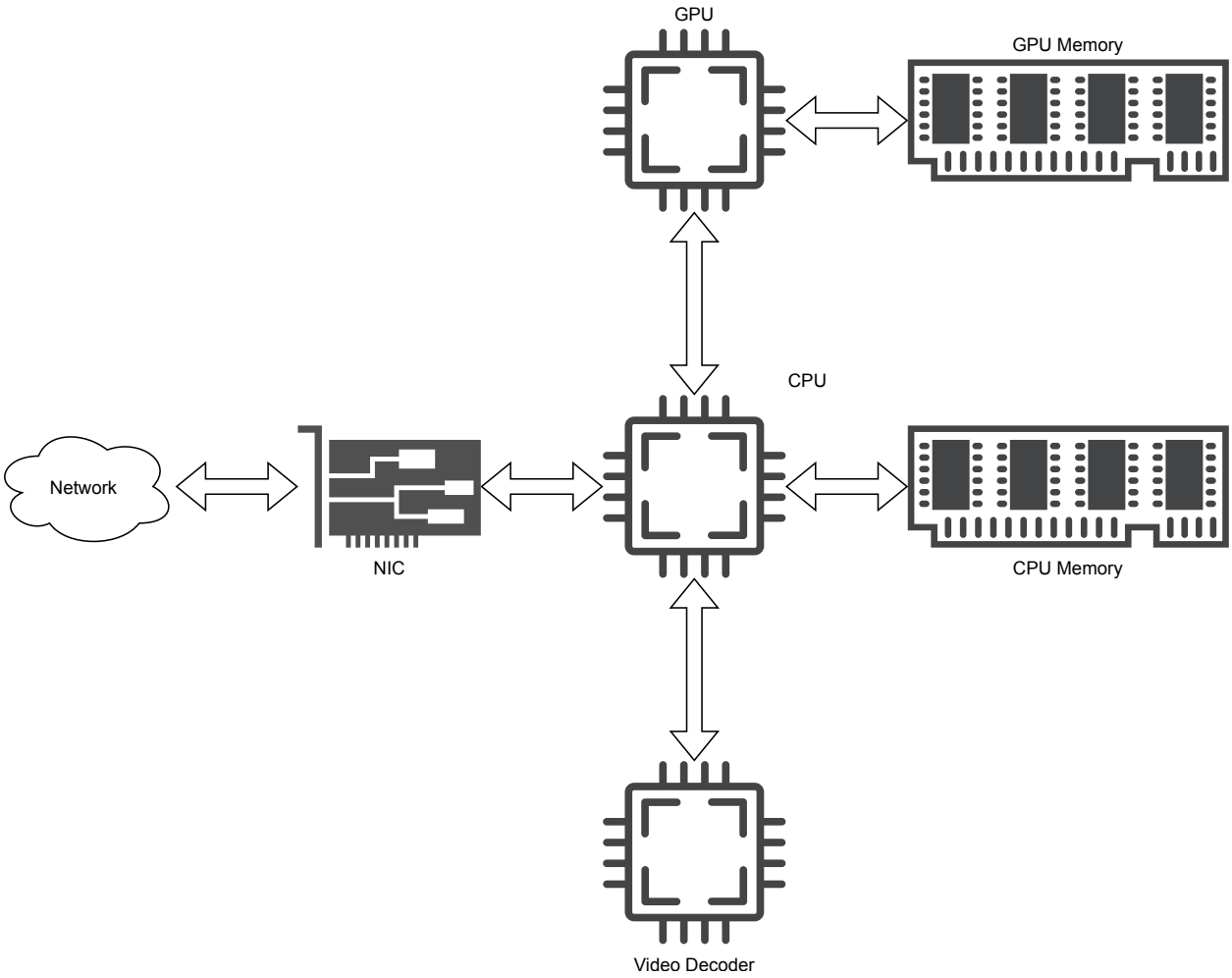


Figure 5.6: Hardware used by the mixer

NDI sources

NDI sources primarily relate to the available network bandwidth. The network interface card (NIC) of the host computer and the rest of the network must be able to handle the bandwidth generated by NDI traffic. Moreover, it is advisable to leave plenty of headroom to reduce latency and damp traffic spikes. Due to a lack of NDI sources, testing could not be done, so the table 5.1 has been elaborated according to the official data[35].

Modern computers have 1Gbit/s NICs which can easily fit multiple FHD streams. However, when using multiple UHD streams, several NICs or 10Gbit/s NICs may be required.

NDI uses the CPU for decoding video. This is currently one of the few CPU-bound tasks of the

Video mode	Bandwidth [Mbit/s]
480i60	20
720p60	90
1080i60	100
1080p60	150
2160i60	250
2160p60	400
1080p30 (HX)	8
2160p30 (HX)	20

Note: HX means that H.264 is used instead of the NDI's own codec.

Table 5.1: Network bandwidth used by NDI

mixer. According to the tests performed, 1 CPU core per NDI stream should be enough in modern processors, even when using UHD signals.

Video decoding

If possible, the mixer takes advantage of video decoding hardware to decode video clips. However, this hardware only supports a limited number of concurrent video streams. Currently, there is no mechanism to offload this work to the CPU when the dedicated hardware gets saturated, so the video decoder must be able to handle the requested workload. Video decoders usually come bundled in GPUs and CPUs, so specifications of these components should be referred to decide if they can accommodate this task. To put numbers in perspective, 8th-gen *Intel* processors (2018) can decode more than 16 concurrent FHD streams[36].

If no decoding hardware is available, the previous rule of 1 CPU core per stream applies. Obviously, this number of CPU cores must be added to the previous count.

Input count

Regardless of the nature of the external inputs, all frames lie at CPU memory before being processed. According to the previous chapter, this uncompressed frame is uploaded to the GPU for further processing. Therefore, the CPU-GPU bandwidth must be able to accommodate these uncompressed video transfers. For system on a chip (SOC)-like architectures, this bandwidth is virtually infinite. However, if a discrete GPU is used, this bandwidth is crucial. The standard bitrates for PCIe are shown in the table 5.2.

Bandwidth [Gbit/s]	x1	x2	x4	x8	x16
1.0	2	4	8	16	32
2.0	4	8	16	32	64
3.0	7.9	15.8	31.6	63.2	126.4
4.0	15.8	31.6	63.2	126.4	252.8

Table 5.2: PCIe bandwidths

Most modern GPUs use PCIe 3.0 x16. Even though the actual bitrates are very high, the uncompressed frames need to be and processed in less than a frame period, so using the link at its full potential is not viable. A good rule of thumb can be to approximate a SDI bandwidth to the used video settings and ensure that no more than the 25% of the available PCIe bandwidth

would be used. Unlike the previous rules, this rule only applies to the inputs that are actually being outputted in some manner.

GPU's computational power

GPUs are made of several highly specialized logic, so quantifying their computational capabilities and comparing them to the mixer's performance is not an easy task. The approach described here consists in measuring the video manipulation capabilities of two computers and comparing them to publicly available benchmark scores. There are several factors that directly impact GPU usage, such as resolution and framerate. However, these tests are focused on determining the amount of usable keyers and M/E banks. The other parameters are assumed to scale linearly, although this is not strictly true.

The tests consist in increasing the keyer and M/E count until 75% of GPU usage is reached. This percentage leaves plenty of room for errors in the measurement procedure, as this neglects many variables. The amount of keyers and M/E banks are increased orthogonally, this is, the 75% is reached twice, once increasing the keyer count and once increasing the M/E count. Then, these numbers can be compared with the scores provided by PassMark[1] and a slope can be assigned to each of the parameters. All tests will be performed in FHD 30Hz and with the keyers configured as chroma keyers (Chroma keying is the costliest keying type).

During the tests, it has been observed that the relation between GPU usage and resource count is not linear. The influence of adding the first element is much higher comparing it to the second, and so on. Therefore, a logarithmic ponderation will be used when considering the element count.

To calculate the M/E slop, the GPU's benchmark score is divided by the logarithm of the number of M/Es, as illustrated by the equation (5.1). Similarly, the slope of the keyer's influence is calculated dividing the score by the logarithm of the keyer count. However, the cost of having one M/Es is subtracted, as at least one M/E must exist to test keyers.

$$m_{me} = \frac{s}{log(1 + n_{me})} \tag{5.1}$$

$$m_{key} = \frac{s - m_{me} \cdot log(2)}{log(1 + n_{key})} \tag{5.2}$$

As observed in the table 5.3, the slopes obtained for both test computers are very different. This is probably due to the fact that both test computers have completely different architectures. The first one integrates the GPU inside the CPU and both share memory. As opposed to this, the second one resembles the architecture previously shown in the figure 5.6.

GPU model	Intel UHD 620	NVidia GTX 980
PassMark score	888	11282
Max. M/Es	2	5
Max. keyers	4	36
M/E slope	1860	14500
Keyer slope	470	4400

Table 5.3: Saturation thresholds for both test computers

[1] https://www.videocardbenchmark.net/gpu_list.php

Considering all this information, a rough estimation for the required GPU power has been formulated in the equations (5.3) and (5.4). The result is the required minimum PassMark score for a given configuration. Indeed, this has been elaborated using the available information, which is very reduced, so the estimation is very imprecise. As the results obtained for both computers are very disparate, two equations are described, one for SOC-like architectures (5.3) and the other for discrete GPUs (5.4). The input variables for these formulas are: resolution ($w \times h$), framerate (f), M/E count (n_{me}) and the total keyer count (n_{key}).

$$s_{soc} = [1860 \log(1 + n_{me}) + 470 \log(1 + n_{key})] \cdot \frac{w}{1920} \cdot \frac{h}{1080} \cdot \frac{f}{30} \tag{5.3}$$

$$s_{disc} = [14500 \log(1 + n_{me}) + 4400 \log(1 + n_{key})] \cdot \frac{w}{1920} \cdot \frac{h}{1080} \cdot \frac{f}{30} \tag{5.4}$$

Chapter 6

Budget

The cost of developing this project boils down to the engineer's salary and the prize of some materials. Additionally, some software licenses were required, although most of the used software was FOSS. The table 6.1 accounts for all these spendings.

Concept	Description	Qty.	Unit cost	Total
Dell precision Tower 5820	Desktop computer for development	1	2.366,27 €	2.366,27 €
Dell UltraSharp U2722D	Monitor for the desktop computer	2	510,00 €	1.020,00 €
Xiaomi Mi 10T Lite	Android smartphone used for testing NDI	1	329,00 €	329,00 €
NDI HX Android app	Application used to broadcast video from a smartphone	1	9,99 €	9,99 €
GitHub Pro subscription	Version control for development. Monthly subscription	6	4,00 €	24,00 €
Engineer salary	Developer. Prize per hour	310	19,41 €	6.017,10 €
			Total	9.766,36 €

Table 6.1: Project budget

Chapter 7

Future work

This project provides a good baseline for developing related projects on top. These projects can be classified in two groups: The expansion of the capabilities of the mixer itself and the development of controllers for the mixer.

7.1 Expansion and improvement of the mixer

As described earlier, the architecture of the mixer allows easy expansion by inheriting from the provided base classes. The most important base classes that have a high potential of expansion are `ZuazoBase`, `TransitionBase` and `OverlayBase`.

7.1.1 Adding new video processing elements

Currently, the fan of available video processors is very narrow, as there are only four classes that perform such a task (the media player, the NDI source, the window output and the M/E). However, inheriting from the `ZuazoBase` class, new elements can be added. Here are some useful examples of them:

Mosaic viewer

All modern mixers, regardless of their implementation, integrate a component that allows to simultaneously view multiple sources at the same time. This component is commonly referred as the mosaic generator or the multi-viewer. Moreover, they usually show useful information such as a signal identifier and a tally light. Adding a plain mosaic viewer is an easy task using the *Zuazo* library. In fact, at the current state, a M/E could be configured to accomplish this task. However, adding the tally light and especially the text requires a bit more of effort.

Colour-bar generators

As the name suggests, it would be useful to have a component that procedurally generates standard color bars. These color bars can be used to test many aspects of the mixer.

NDI output

Currently, only NDI input is implemented. However, this protocol also allows outputting video, which might be useful for many production environments. Implementing this feature would involve

expanding the *zuazo-ndi* module and wrapping it with a mixer component.

SDI I/O through Black Magick Design's DeckLink cards

Similarly to the previous improvement, this one also involves expanding the I/O capabilities of the mixer. In this case, the new I/O would be performed using SDI through Black Magick's DeckLink cards. These cards offer a good cross-platform C++ programming API.

7.1.2 Improvements to existing video processing elements

Some existing elements have plenty of room for improvement. For instance, the control of the media players is very rudimentary. Important features like entry points, playlists and synchronization are missing.

7.1.3 Adding new transitions

Similarly to the expansion of video processing elements, new transitions can be added by inheriting from `TransitionBase`. Currently, wipe style transitions have not been implemented, so there is definitely room for improvement. Moreover, the existing transitions can be further improved by adding more variations.

7.2 Development and improvement of controllers

As mentioned in previous chapters, the mixer itself is agnostic of the UI. This allows to implement different controllers that focus on different usages of a production switcher. Some examples of controllers are described hereafter.

7.2.1 Physical control panel

Although in the Appendix B a prototype of a physical panel is described, this is a mere proof of concept. Moreover, the physical control panel lacks some of the most crucial aspects such as the *t-bar*. However this prototype proofs the viability of implementing a control panel that interacts with this mixer.

There are two strategies to accomplish this task. The first one is to design everything from scratch. The benefit of doing so is that everything is under the control of the designer, but it requires a lot of work. The other approach is to reverse engineer a decommissioned commercial control panel to adapt it to the new mixer.

7.2.2 Improvements of the web application based controller

Although the implemented web application for controlling the mixer covers most aspects of it, there is a lot of room for improvement.

7.2.3 Show sequencer

Another interesting controller would be a sequencer. Unlike all previous controllers, this does not require a human operator. Instead, the camera shots are time-coded. In essence, linear TV is produced as if it were nonlinear. Currently, the only application that allows to do this is CuePilot, which is shown in the figure 7.1. The multi-client nature of this mixer is ideal for this type of control, as a human controller can override the automated controls.

Figure 7.1: CuePilot being used at the Eurovision Song Contest 2017

Chapter 8

Conclusions

Despite the proliferation of many video-on-demand streaming services, live video remains unbeatable for many applications. Therefore, as long as the linear TV continues to exist, so will the vision mixers, regardless of their nature. This does not mean that there is no room for innovation on these systems, as they have to keep adapting to new multimedia trends.

In this project, a usable software-based vision mixer has been implemented. This mixer makes efficient use of the available hardware resources and allows to perform relatively complex compositions on modest hardware. Moreover, all operations are performed with colour accuracy in mind, as it implements all widespread industry-standards.

Compared to the rest of commercial mixers, it introduces the ability to use arbitrary Bézier contours to mask parts of a keyer. This is useful to trash undesired parts of an image when performing a chroma key. In addition, it can be used for more creative applications. The major limitation of this mixer is the lack of available I/O, as it only supports NDI inputs and window outputs. However, the architecture allows easy expansion, so this problem can be addressed in the future.

Considering that it has proven to be relatively reliable, it could be used in some select production environments, as long as more extensive stability testings are carried out. Additionally, it could be used for educational purposes, as it allows to have one mixer per student.

Bibliography

[1] (2020). "The a to b of early video mixer technology", Broadcast Engineering Conservarion Group, [Online]. Available: https://becg.org.uk/2020/10/18/the-a-to-b-of-early-video-mixer-technology/ (visited on 05/28/2021).

[2] P. Ward, *Studio and outside broadcast camerawork a guide to multi-camerawork production*, eng, 2nd ed., ser. Media manuals. Oxford ; Boston, MA: Focal Press, 2001, ISBN: 1-136-05482-0.

[3] J.-J. Peters, "A history of television", EBU, Tech. Rep. [Online]. Available: http://arantxa.ii.uam.es/~jms/tvd/tv_history.pdf (visited on 05/16/2021).

[4] "Historia de los medios técnicos de la televisión. manipuladores de señal i: Mezcladores de vídeo", *TM Broadcast*, 2016, (Spanish). [Online]. Available: https://tmbroadcast.es/index.php/manipuladores-de-senal-tv/ (visited on 05/18/2021).

[5] (2015). "The evolution of vision mixers", kitplus, [Online]. Available: https://www.kitplus.com/articles/The_evolution_of_vision_mixers/1354.html (visited on 05/28/2021).

[6] J. Owens, *Television production*, eng, Seventeenth edition. 2020, ISBN: 0-429-02758-3.

[7] A. Utterback, *Studio Television Production and Directing: Concepts, Equipment, and Procedures*, eng. Oxford: Routledge, 2015, ISBN: 9781138193970.

[8] L. I. O. Berenguer and J. L. R. Vázquez, *Ingeniería de vídeo en entornos broadcast*. Fundación General de la Universidad Politécnica de Madrid, 2018, ISBN: 9788416397686.

[9] "Reference electro-optical transfer function for flat panel displays used in HDTV studio production", International Telecommunications Union, Standard ITU-R BT.1886-0, Mar. 2011.

[10] *ATEM production studio switchers manual*, Black Magick Design, 2020. [Online]. Available: https://www.blackmagicdesign.com/developer/product/atem (visited on 06/24/2021).

[11] *User's guide for Sony XVS-9000/6000/7000/6000 multi format switchers*, ver. 3.5, Sony. [Online]. Available: https://pro.sony/ue_US/support-resources/xvs-9000/manual (visited on 06/24/2021).

[12] Wikipedia. (2021). "Software vision mixer", [Online]. Available: http://en.wikipedia.org/w/index.php?title=Software%5C%20vision%5C%20mixer&oldid=1023821024 (visited on 05/28/2021).

[13] L. Pardo, "Video toaster: El editor de vídeo que revolucionó la cultura de los '90", Neoteo, Tech. Rep., 2019, (Spanish). [Online]. Available: https://www.neoteo.com/video-toaster-editor-de-video/ (visited on 06/24/2021).

[14] *Ross Graphite user manual*, ver. 5.0, Ross Video. [Online]. Available: https://www.rossvideo.com/products-services/acquisition-production/production-switchers/graphite/ (visited on 06/24/2021).

[15] (2021). "Open Broadcaster Software (OBS) Wiki", OBS Project, [Online]. Available: https://obsproject.com/wiki/ (visited on 06/24/2021).

[16] (2021). "CasparCG Wiki", SVT, [Online]. Available: https://github.com/CasparCG/help/wiki (visited on 06/24/2021).

[17] *NewTek TriCaster manual*, 2021st ed., NewTek. [Online]. Available: https://downloads.newtek.com/LiveProductionSystems/VMC1/TCXD.pdf (visited on 06/24/2021).

[18] *VMix user's guide manual*, vMix. [Online]. Available: https://www.vmix.com/help24/vMixUserGuide.pdf (visited on 06/24/2021).

[19] "RTVE Cataluña evoluciona su producción al HD y a la tecnología IP", *TM Broadcast*, 2021, (Spanish). [Online]. Available: https://tmbroadcast.es/index.php/rtve-sant-cugat-hd-ip/ (visited on 06/24/2021).

[20] "Señales de producción en red", *TM Broadcast*, 2017, (Spanish). [Online]. Available: https://tmbroadcast.es/index.php/senales-de-produccion-en-red/ (visited on 05/19/2021).

[21] (2021). "SMPTE-2110 FAQ", SMPTE, [Online]. Available: https://www.smpte.org/smpte-st-2110-faq (visited on 05/18/2021).

[22] "Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios", International Telecommunications Union, Standard ITU-R BT.601-7, Mar. 2011.

[23] "Parameter values for the HDTV standards for production and international programme exchange", International Telecommunications Union, Standard ITU-R BT.709-6, Jun. 2015.

[24] "Parameter values for ultra-high definition television systems for production and international programme exchange", International Telecommunications Union, Standard ITU-R BT.2020-2, Oct. 2015.

[25] "Signal parameters for 1125-line high-definition production systems", Society of Motion Picture and Television Engineers, Standard SMPTE 240M, 1995.

[26] "D-Cinema quality - screen luminance level, chromaticity and uniformity", Society of Motion Picture and Television Engineers, Standard SMPTE 431, 2006.

[27] "Digital source processing - color processing for D-Cinema", Society of Motion Picture and Television Engineers, Standard SMPTE 432, 2006.

[28] "High dynamic range electro-optical transfer function of mastering reference display", Society of Motion Picture and Television Engineers, Standard SMPTE ST 2084, 2014.

[29] "Parameter values for the hybrid log-gamma (HLG) high dynamic range television (HDR-TV) system for programme production", Association of Radio Industries and Businesses, Standard ARIB STD-B67, 2015.

[30] "Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space - sRGB", International Electrotechnical Commission, Standard IEC61966-2-1, 1999.

[31] "Multimedia systems and equipment - Colour measurement and management - Part 2-4: Colour management - Extended-gamut YCC colour space for video applications - xvYCC", International Electrotechnical Commission, Standard IEC61966-2-4, 2006.

[32] "IEEE standard for floating-point arithmetic", Institute of Electrical and Electronics Engineers, Standard IEEE 754, 1985.

[33] *Efficient bicubic filtering code in glsl?* [Online]. Available: https://stackoverflow.com/questions/13501081/efficient-bicubic-filtering-code-in-glsl (visited on 06/28/2021).

[34] *Fast branchless RGB to HSV conversion in GLSL.* [Online]. Available: http://lolengine.net/blog/2013/07/27/rgb-to-hsv-in-glsl (visited on 06/28/2021).

[35] (2021). "NDI network bandwidth", NewTek, [Online]. Available: https://support.newtek.com/hc/en-us/articles/217662708-NDI-Network-Bandwidth (visited on 05/28/2021).

[36] "Technical Resources: Intel Core Processors", Intel, Documentation, 2021.

Appendices

Appendix A

GPU based Bézier contour rasterization

A.1 Introduction

Bézier curves are a type of parametric curves invented by Pierre Bézier in the 1960s. They are widely used in graphic design software and computer aided design (CAD) tools, as they provide an intuitive control over the shape of the curve. This intuitiveness is achieved using control points, which in colloquial terms, attract the curve towards them. The number of control points that are used define the degree of the curve[1]. Nowadays, most programs use cubic Bézier curves, which are defined by 2 control points plus the two extreme points, as shown in the figure A.1.

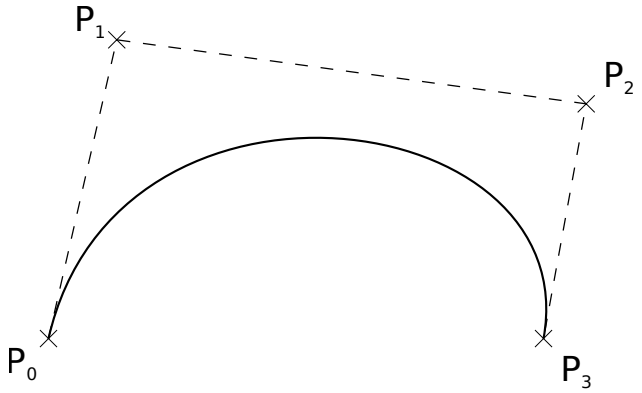


Figure A.1: Cubic Bézier example

Modern GPUs are highly specialized in rendering triangles. Therefore, it may seem that rendering smooth Bézier curves is an impossible task for this hardware. However, in the last 2 decades, several approaches have been developed to solve this problem.

Some of them rely on segmenting the curves in very short lines, so that it appears to be quasi-smooth. This approach involves creating lots of vertices and thin triangles, which might hurt performance and memory availability. Moreover, unless external anti-aliasing is used, the edges will appear to be jaggy.

Some other approaches are based on uploading the contour data to the GPU and processing it in a fancy fragment shader. They usually feature a signed distance field (SDF) based anti-aliasing mechanism, so the edges appear to be smooth. However, for the sake of optimization, they might not obtain pixel-perfect results, while having a very heavy impact on the performance, as the curves

need to be processed once per frame. This may make them a good candidate for mutating curves, but do not suit more static scenarios.

Here is where Loop-Blinn's solution comes into place. This solution consists in describing the Bézier curve as a set of implicit curve equations and assigning its value to each of the rendered pixels[2]. This approach is highly efficient due to its linearity, although it requires a bit of preliminary *number crunching* in the CPU.

This last approach has been chosen to crop video frames with an arbitrary shape. In particular, cubic curves will be used, as they have become an industry standard.

A.2 Mathematical definition of Bézier curves

The analytical form of the Bézier curves is based on polynomials of arbitrary degree. For a N-degree Bézier curve, $N+1$ control points are needed, where two of them will be the start and end points. Therefore, a cubic curve is defined by 4 control points, as shown in the figure A.1. The generic form for a N-degree Bézier curve is displayed in the equation (A.1) [1], [3]. Setting $N = 3$ for cubic curves, the equation (A.3) is obtained.

$$B(t) = \sum_{i=0}^{N} P_i \cdot b_{N,i}(t) \qquad t \in [0,1] \tag{A.1}$$

where $b_{N,i}$ is the N-th degree Bernstein basis polynomial's i-th term, which is defined as (A.2).

$$b_{N,i}(t) = \binom{N}{i}(1-t)^{N-i}t^i \qquad t \in [0,1] \tag{A.2}$$

$$B_3(t) = P_0(1-t)^3 + 3P_1(1-t)^2t + 3P_2(1-t)t^2 + P_3t^3 \qquad t \in [0,1] \tag{A.3}$$

Multiple Bézier curves can be chained one after the other so that the end-point of one curve is the beginning of the next one. This is commonly known as a polycurve. If both ends of the curve meet, a closed contour is formed. These closed contours will be the input data for the algorithm.

Any Bézier curve can be expressed in terms of a higher order curve, although the reverse operation is not generally possible. For instance, cubic curves can be used to draw quadratic curves or lines, as lines can be thought of as first degree curves [1]. This property will become handy in later stages where the implementation of the algorithm is described. Moreover, a Bézier curve can be subdivided in many shorter segments of same degree.

The derivative of a Bézier curve will be another Bézier curve of one degree lower. As all terms are multiplied by t^N, N must be multiplied by the derivative, due to the polynomial derivation rules. The new control points will be the differences between consecutive pairs of points in the original curve, as displayed in the equation (A.4).

$$\frac{d}{dt}B(t) = N\sum_{i=0}^{N-1}(P_{i+1} - P_i) \cdot b_{N-1,i} \qquad t \in [0,1] \tag{A.4}$$

If $t = 0$ and $t = 1$ are substituted in the expression (A.1), the equation simplifies down to P_0 and P_N respectively, this is, the extreme points of the curve. If the same values are substituted in the derivative, the expression is simplified to the difference between the extreme points and its neighbor. These four trivial cases are displayed in the equation (A.5). Although they might seem

useless at first glance, they play a key role on the intuitiveness of the Bézier curves. It can be easily inferred that the curve in the extreme points will be tangent to the line joining the extreme point with its neighboring control point. This can be confirmed observing the figure A.1, where the curve in P_0 is tangent to $\overrightarrow{P_0P_1}$ and tangent to $\overrightarrow{P_2P_3}$ in P_3.

$$B(0) = P_0 \quad B(1) = P_N \quad \frac{d}{dt}B(0) = N(P_1 - P_0) \quad \frac{d}{dt}B(1) = N(P_{N-1} - P_N) \tag{A.5}$$

This statement has also an impact on the polycurves. A polycurve will only be derivable at the transition point between the segments if the following condition is met: The penultimate control point of a segment must lie on the line joining the first and second control points of the next segment -remember that due to continuity, the first control point of the next segment is also the last control point of the current segment-. This derivability condition is known as C1 continuity. Although it does not have to be met, it does have an artistic meaning: The absence or presence of corners.

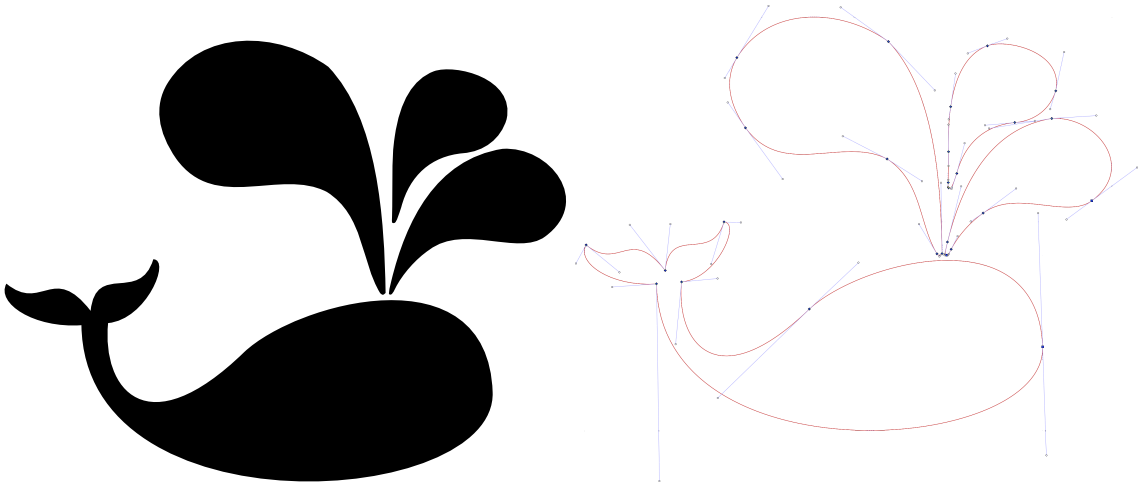
A.3 Loop Blinn's algorithm implementation

Loop Blinn's Bézier curve rasterization algorithm was developed by Charles Loop and Jim Blinn in 2005. For the purposes of this project, a variant of this algorithm has been implemented to be able to crop a video frame with an arbitrary shape. The original algorithm has two variants, the first one being for quadratic curves and the other one for cubic curves. In this case, the later one has been chosen, as quadratic curves can be easily converted into cubic curves.

A.3.1 The test image

This algorithm must handle some extreme cases. Therefore, to showcase how all those situations are handled, the test image must feature them. For this reason, the vectorization of a whale shown in the figure A.2 has been chosen, as it provides the following special cases:

- No holes, as currently they are not supported.
- Multiple independent contours. In this test image, 4 bodies can be distinguished: The whale itself and 3 drops of water.
- Highly convex shape.
- Star-like sections: The tail forms a 3 tipped star.
- Straight lines, which can be spotted in the root of the center droplet.
- Quadratic curves
- S shaped cubic curves, which form the whale's tail fins.
- U shaped cubic curves, which are the most common ones.
- Curves that come very close, as it does on the tail.



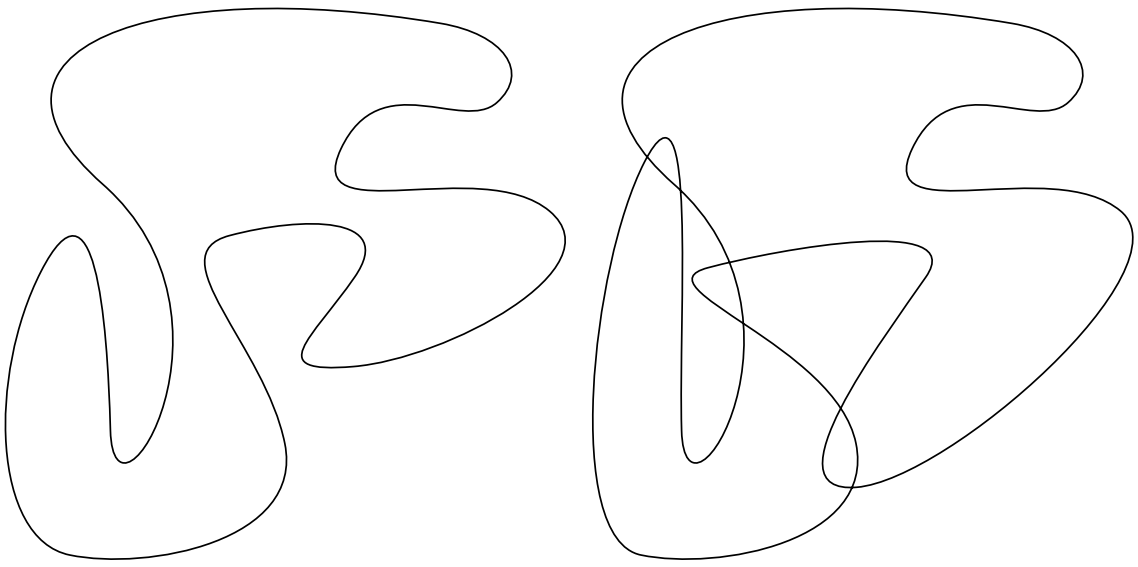
(a) Filled (b) Outline with control points

Figure A.2: Vectorization of a cartoon whale

A.3.2 Procedure

As mentioned before, the input data for the algorithm is a collection of contours which in turn are formed by multiple cubic Bézier curves. As the contour must be continuous and closed, two consecutive segments share a single extreme point, so there is no need to store it twice. Therefore, for a given contour formed by M N-degree segments, $M \cdot N$ points will be used to store the contour. Note that each of the individual segments is defined by $N + 1$ control points.

Although the algorithm aims to have as few as possible input restrictions, there is one condition that the input must meet: The contours must not self-intersect. An example of a valid and invalid contour is displayed in the figure A.3



(a) Valid (b) Invalid

Figure A.3: Valid and invalid input data

Most of the operations described in the following steps will be performed repeatedly on each of the contours. Therefore, only the body of the whale will be used as an example for those steps.

Contour sanitation

In the context of computer graphics, contours and polygons are usually described in counter clockwise ↺ (CCW) order. In fact, if a contour is defined as clockwise ↻ (CW), it is usually to signal that it is a hole inside another bigger contour. This convention is illustrated in the figure A.4. However, as mentioned earlier, this algorithm does not currently support holes, so all CW contours are turned into CCW by reversing the order of its points. Therefore, from now on, all contours can be assumed to be CCW.

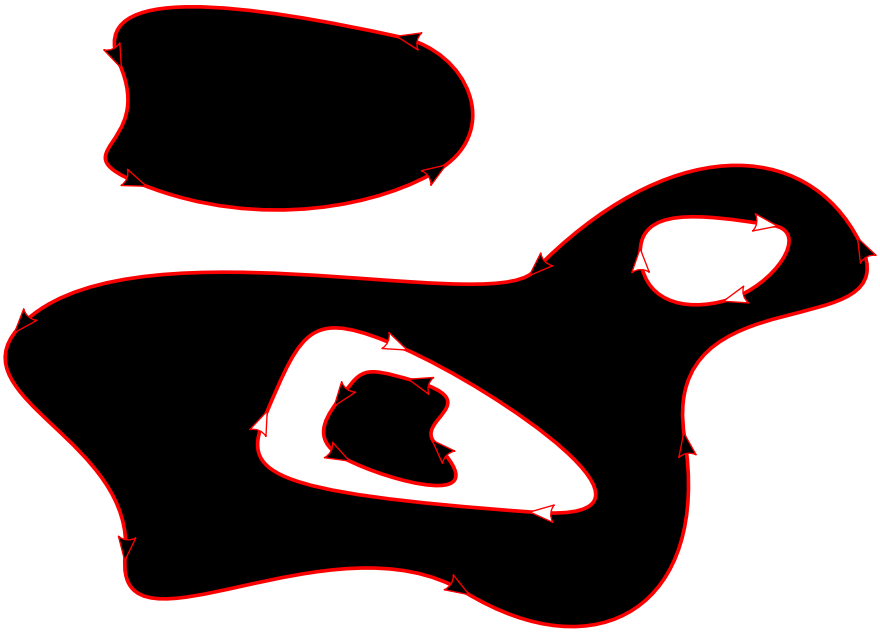
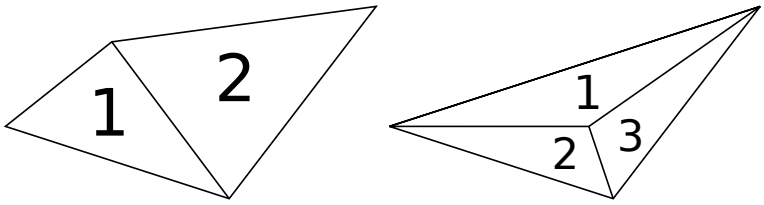


Figure A.4: Winding convention

Outer-hull triangulation

As mentioned before, cubic segments are defined by four control points. Due to the curve tangency properties, which were described earlier, it can be ensured that a cubic Bézier curve will not escape the convex polygon formed by its four control points. This is a strong property, as all calculations can be narrowed down to this area. As these computations will be performed in the fragment shader of the GPU's graphics pipeline, this convex polygon needs to be triangulated. The collection of convex polygons regarding each of the segments, will be called the outer hull. Each of the pixels rastered by the outer hull will have to be individually evaluated to determine whether it is inside or outside the contour.

Note that until this point the term "convex polygon" has been vaguely used. This is because the nature of it is unknown. Therefore, the first step is to determine the convex shape formed by the control points. This can be boiled down to two cases, assuming that the control points are not colinear nor the same point. The first case is the trivial one, the four points form a convex quadrilateral. The other one is the tricky case, which occurs when they form a concave quadrilateral, as shown in the figure A.5. In this situation, a triangle -which is always convex- will be formed, containing one of the points inside.



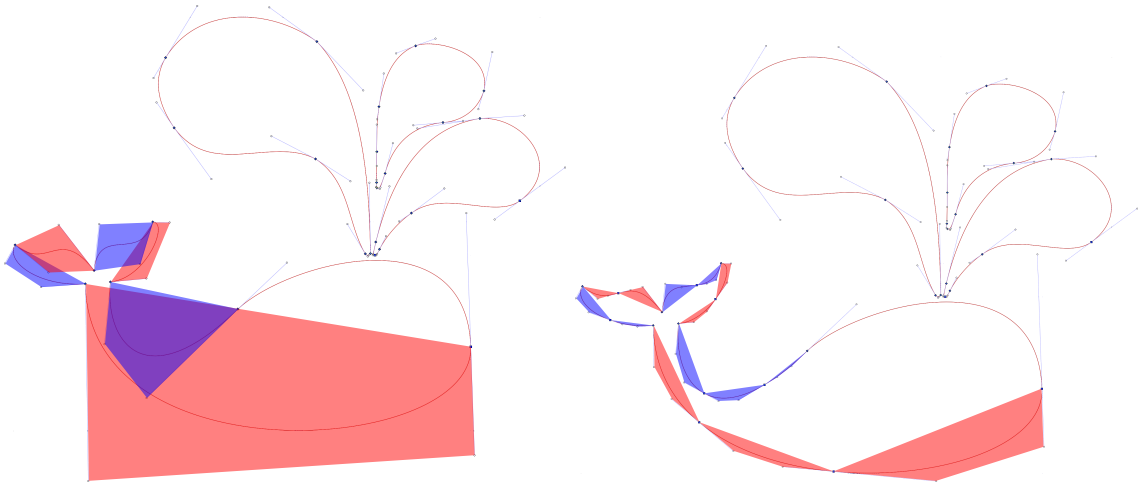
(a) Convex quadrilateral

(b) Concave quadrilateral turned into a triangle

Figure A.5: Triangulation of the outer-hull

In the first case, the triangulation is performed by splitting the quad on its shortest diagonal. The benefit of splitting it on its shortest diagonal is that repeated pixel calculations are minimized. In the second case, three triangles will be formed, all having the middle vertex in common. For the sake of avoiding repeated vertex shader executions, both triangulations are made in a triangle-strip compatible manner and using a index buffer. This adds extra complexity to the algorithm, but it may be beneficial for complex contours.

If two curves come very close, their bounding convex polygons may collide. This can lead to rendering artifacts, so it needs to be addressed. The solution adopted here is fairly crude, as this is not a common case. The solution consists of splitting both colliding segments in halves, using the method described earlier. All cases that occur on the whale's image have been highlighted in the figure A.6. There is another potential case that requires splitting a segment in two. This other situation will be mentioned later.



(a) Intersections

(b) After splitting

Figure A.6: Intersection removal example

Inner-hull extraction and triangulation

After defining the outer hull, a big unfilled void remains in the middle of the contour. Luckily, it can be ensured that this arbitrarily shaped polygon is fully inside the contour. Therefore, the only task regarding it is to triangulate. The algorithm selected for this task is based on the source code of PolyPartition[4], although it has been optimized to avoid unnecessary calculations and heap allocations.

The first step is to determine which pairs of vertices are mutually visible. Assuming that the polygon does not self-intersect, it can be ensured that consecutive points are mutually visible. However, this may not be true for other pairs of vertices if the polygon is concave. For the sake of testing their visibility, the intersection between the line joining them and all edges of the polygon is computed. Moreover, this line must lie between the two edges associated with the departing vertex. If no intersection is found and the line does not exit the polygon, it can be considered a diagonal.

The only remaining step is to choose the appropriate diagonals to triangulate the polygon. To begin with, a random edge of the polygon is placed into a queue, which was previously empty. Then, a line is extracted from this queue -obviously, the first time, this line will be the newly placed edge-. For this line, the closest vertex to which diagonals or edges can be traced is chosen. These diagonals -not the edges- are pushed into the queue, so that they are the starting point for future iterations. This process is repeated indefinitely until the queue is empty. On each iteration, zero, one, or two diagonals will be added to the queue, while a single one will be removed. A possible result of the execution of this algorithm is shown in the figure A.7.

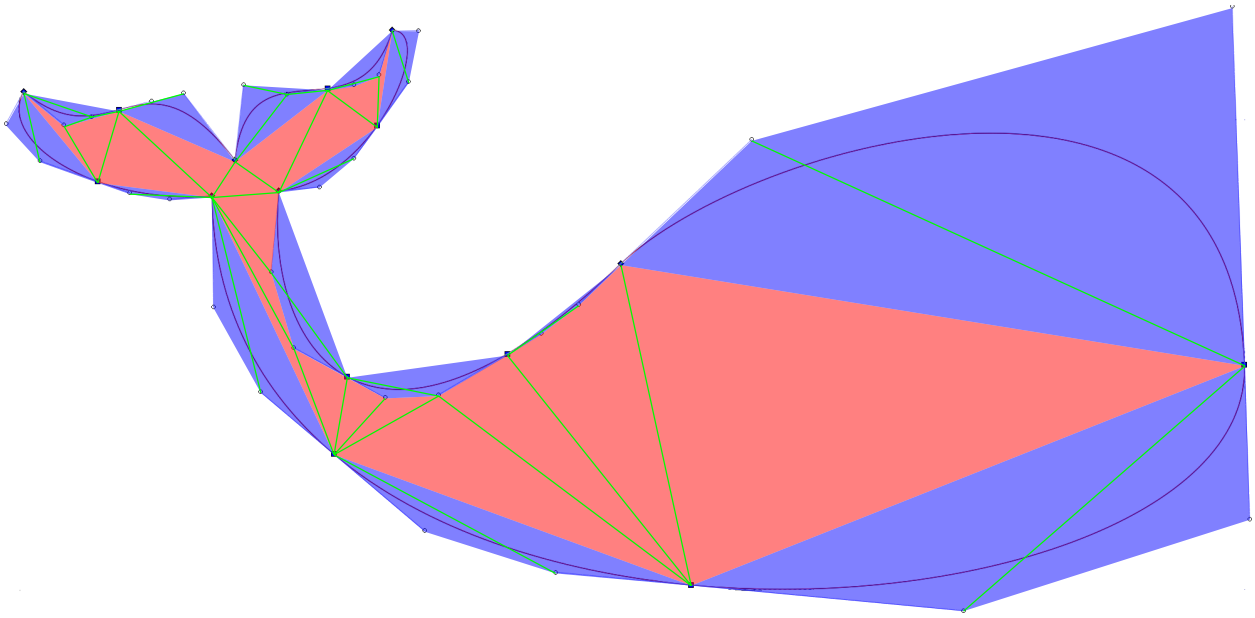


Figure A.7: Triangulation of the inner-hull and outer-hull

The only reason for not supporting holes is that the chosen triangulation algorithm does not support them. Its not even an efficient algorithm, as it has $O(n^3)$ complexity, but it provides the best triangulation in all situations. Definitively, there is a lot of room for improvement in this section. Therefore, for future releases, a Constrained Delaunay triangulation algorithm will be considered. This is the algorithm used by Charles Loop and Jim Blinn in their paper[2]. This other algorithm has $O(n \log n)$ complexity[5].

Homogeneous implicit line coordinate calculation

Until this point, C. Loop and J. Blinn's algorithm has been followed *a grosso modo*, adding the author's own criteria. However, this part of the algorithm has been implemented as a mere copy of the equations described in their article[2] and their chapter in GPU Gems 3[6]. As a recap, until this point, we have separated a cubic Bézier contour in two parts. The first one, the outer hull, corresponds to the parts of the shape that are going to be conditionally filled. The other one,

the inner hull, will be fully filled. Therefore, the next logical step is to implement a method to determine which pixels of the outer hull correspond to the inside of the contour and which ones to the outside, as only the ones that are inside will have to be filled.

As stated by C. Loop and J. Blinn, "A simple implicit equation for a parametric curve is found in a space that can be thought of as an analog to texture space"[2]. This means that any Bézier curve can be mapped to those implicit equations which will be linearly interpolated. Therefore, the main goal of this section is to calculate the implicit equations that belong to each of the control points.

This implicit equation will be (A.6) for cubic curves. Luckily, n always equals to 1 as the line that it represents is always at infinity. Therefore, the actual equation will be (A.7). This implies that a $[k, l, m]$ vector will have to be assigned to each of the control points in order to map the $c(x, y)$ implicit function to space.

$$c(x, y) = k^3 - lmn \tag{A.6}$$

$$c(x, y) = k^3 - lm \tag{A.7}$$

The input for this section will be each of the individual segments that form the outer hull. According to Jim Blinn's Corner[7], a cubic segment can be classified as one of the following topologies:

- Serpentine
- Loop
- Cusp: The transition point between serpentine and loop
- Quadratic
- Line
- Point

As the equations vary from topology to topology, the first step is to classify the segments. For this task the $\delta_1, \delta_2, \delta_3$ parameters and the discriminant are calculated according to the equations (A.8) (A.9) (A.10). Note that a affine space is used, so that $P_i' = [P_i, 1]$. This is how the mixed product of 2D points is defined. As a small caveat, the result of the mixed product may get very large, easily overwhelming 32bit float-s. Luckily, multiplying the result by a constant does not alter the outcome, so the a values should be normalized to avoid computing the δ-s with large numbers, potentially introducing floating-point errors.

$$a_1 = P_0 \cdot P_3 \times P_2 \qquad a_2 = P_1 \cdot P_0 \times P_3 \qquad a_3 = P_2 \cdot P_1 \times P_0 \tag{A.8}$$

$$\delta_1 = a_1 - 2a_2 + 3a_3 \qquad \delta_2 = -a_2 - 3a_3 \qquad \delta_3 = 3a_3 \tag{A.9}$$

$$disc = \delta_1^2 \cdot (3\delta_2^2 - 4\delta_1\delta_3) \tag{A.10}$$

The next step is to obtain the homogeneous implicit line coordinates according to the curve's topology. Each of the topologies is characterized by a particular set of conditions regarding the δ values and the discriminant. The P_i control point's klm vector will be denoted as F_i:

- **Line or point**: $\delta_1 = 0, \delta_2 = 0, \delta_3 = 0$
  As lines and points can be considered 1D or 0D, they do not form an area to fill, so they can be simply discarded.

- **Quadratic**: $\delta_1 = 0, \delta_2 = 0, \delta_3 \neq 0$

	k	l	m
F_0	0	0	0
F_1	$1/3$	0	$1/3$
F_1	$2/3$	$1/3$	$2/3$
F_0	1	1	1

Table A.1: Homogeneous implicit line equation coefficients for quadratic curves

- **Serpentine**: $\delta_1 \neq 0, \delta_2 \neq 0, disc > 0$

	k	l	m
F_0	$l_s m_s$	l_s^3	m_s^3
F_1	$l_s m_s - 1/3 l_s m_t - 1/3 l_t m_s$	$l_s^2(l_s - l_t)$	$m_s^2(m_s - m_t)$
F_2	$l_s m_s - 2/3 l_s m_t - 2/3 l_t m_s - 1/3 l_t m_t$	$l_s(l_s - l_t)^2$	$m_s(m_s - m_t)^2$
F_3	$(l_s - l_t)(m_s - m_t)$	$(l_s - l_t)^3$	$(m_s - m_t)^3$

Table A.2: Homogeneous implicit line equation coefficients for serpentines

where l_s, l_t, m_s and m_t are obtained as displayed in the equation (A.11)

$$l_s = 3\delta_2 - \sqrt{3(3\delta_2^2 - 4\delta_1\delta_3)} \qquad m_s = 3\delta_2 + \sqrt{3(3\delta_2^2 - 4\delta_1\delta_3)} \qquad l_t = m_t = 6\delta_1 \tag{A.11}$$

- **Cusp**: $\delta_2 \neq 0, disc = 0$

	k	l	m
F_0	l_s	l_s^3	1
F_1	$l_s - 1/3 l_t$	$l_s^2(l_s - l_t)$	1
F_2	$l_s - 2/3 l_t$	$l_s(l_s - l_t)^2$	1
F_0	$(l_s - l_t)$	$(l_s - l_t)^3$	1

Table A.3: Homogeneous implicit line equation coefficients for cusps

where l_s, and l_t are obtained as displayed in the equation (A.12)

$$l_s = \delta_3 \qquad l_t = 2\delta_1 \tag{A.12}$$

- **Loop**: $\delta_1 \neq 0, \delta_2 \neq 0, disc < 0$

	k	l	m
F_0	$l_s m_s$	$l_s^2 m_s$	$l_s m_s^2$
F_1	$l_s m_s - 1/3 l_s m_t - 1/3 l_t m_s$	$l_s^2 m_s - 2/3 l_s l_t m_s - 1/3 l_s^2 m_t$	$l_s m_s^2 - 2/3 l_s m_s m_t - 1/3 l_t m_s^2$
F_2	$l_s m_s - 2/3 l_s m_t - 2/3 l_t m_s - 1/3 l_t m_t$	$(l_s - l_t)(l_s m_s - 2/3 l_s m_t - 1/3 l_t m_s)$	$(m_s - m_t)(l_s m_s - 2/3 l_t m_s - 1/3 l_s m_t)$
F_3	$(l_s - l_t)(m_s - m_t)$	$(l_s - l_t)^2(m_s - m_t)$	$(l_s - l_t)(m_s - m_t)^2$

Table A.4: Homogeneous implicit line equation coefficients for loops

where l_s, l_t, m_s and m_t are obtained as displayed in the equation (A.13)

$$l_s = \delta_2 - \sqrt{4\delta_1\delta_3 - 3\delta_2^2} \qquad m_s = \delta_2 + \sqrt{4\delta_1\delta_3 - 3\delta_2^2} \qquad l_t = m_t = 2\delta_1 \tag{A.13}$$

Once these coordinates are assigned to the control point vertices, they are uploaded to the GPU's local memory. When a draw call is issued, their values will be linearly interpolated at the rasterization stage, supplying to each of the executions of the fragment shader its own set of klm values. Then the fragment shader evaluates the equation (A.7). If it turns out to be greater than zero, that particular pixel is outside the contour and it can be discarded.

Similarly to as proposed by Antonio J. Rueda, Juan Ruiz de Miras, and Francisco R. Feito in their paper [8], many parts of this later part of the algorithm could be executed by the geometry shader of the GPU, instead of doing so in the CPU. However, this last optimization has not been considered as geometry shader support is not mandatory in Vulkan, so a fallback method that computes the klm values in the CPU should be still available, greatly increasing the complexity and reducing maintainability.

Signed distance estimation

Considering all previous steps, the purposes of this chapter are already achieved. However, the edges appear to be jaggy due to the aliasing effect. With the aim of solving this, a widespread practice is to obtain the signed distance to the curve, this is, the distance from a point to the closest point on the curve with a different sign depending on the side of the contour on which the point lies. The convention is to use positive values for the outside while using negative ones for the inside.

The $c(x, y)$ function displayed earlier has this particular tendency in points close to the contour, but with a small caveat: Its rate of change is not known. In order to implement a SDF based anti-aliasing strategy, the distance to the border in pixels must be known, so that subpixel distances are used to modulate the intensity of frontier pixels. This metric can be obtained dividing $c(x, y)$ by the length of its gradient, as shown in the equation (A.14). This works because the gradient is approximately square to the border of its proximities. As a consequence, the $||\nabla sd(x, y)|| \approx 1$, which is one of the requirements for the signed distance functions.

$$sd(x, y) \approx \frac{c(x, y)}{||\nabla c(x, y)||} = \frac{c(x, y)}{\sqrt{[\frac{\partial}{\partial x}c(x, y)]^2 + [\frac{\partial}{\partial y}c(x, y)]^2}} \tag{A.14}$$

The gradient of the c function can be easily obtained by applying the chain rule. This is displayed in the equation (A.15) using the Jacobian matrix.

$$\nabla c(x, y) = \begin{bmatrix} \frac{\partial c(x,y)}{\partial x} \\ \frac{\partial c(x,y)}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial k}{\partial x} & \frac{\partial l}{\partial x} & \frac{\partial m}{\partial x} \\ \frac{\partial k}{\partial y} & \frac{\partial l}{\partial y} & \frac{\partial y}{\partial x} \end{bmatrix} \cdot \begin{bmatrix} 3k^2 \\ -m \\ -l \end{bmatrix} \tag{A.15}$$

Modern GPUs can approximate partial derivatives, as their processors are based on 2x2 or 4x4 single instruction multiple data (SIMD) architectures. This means that at any given time, a 2x2 -or 4x4- group of pixels is being concurrently processed, allowing comparisons among them. In GLSL, the functions to obtain partial derivatives are named as `dFdx` and `dFdy` for $\frac{\partial}{\partial x}F(x, y)$ and $\frac{\partial}{\partial y}F(x, y)$, respectively.

After all this process, the image shown in the figure A.8 is obtained. Due to the nature of vector graphics, any transformation performed to this shape prior to the rasterization stage will not affect

the sharpness of the rendered image.

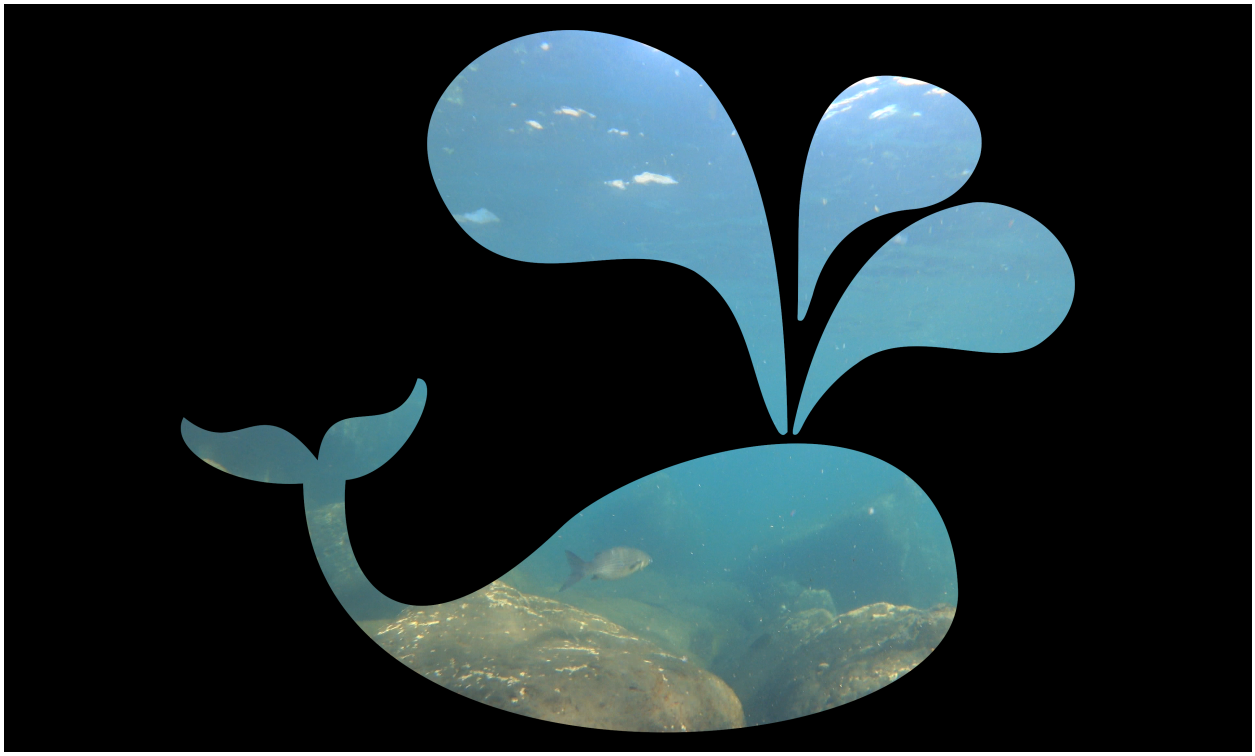


Figure A.8: Result of cropping an image with multiple Bézier contours

Bibliography

[1] (2021). "A primer on bézier curves", [Online]. Available: https://pomax.github.io/bezierinfo/ (visited on 03/23/2021).

[2] C. Loop and J. Blinn, "Resolution independent curve rendering using programmable graphics hardware", *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 1000–1009, 2005. DOI: https://dx.doi.org/10.1145/1073204.1073303.

[3] Wikipedia. (2021). "Bézier curve — Wikipedia, the free encyclopedia", [Online]. Available: http://en.wikipedia.org/w/index.php?title=B%5C%C3%5C%A9zier%5C%20curve&oldid=1012838219 (visited on 03/23/2021).

[4] ivanfratric, *Polypartition*. [Online]. Available: https://github.com/ivanfratric/polypartition (visited on 03/23/2021).

[5] Wikipedia. (2021). "Constrained Delaunay triangulation — Wikipedia, the free encyclopedia", [Online]. Available: https://en.wikipedia.org/wiki/Constrained_Delaunay_triangulation (visited on 04/16/2021).

[6] C. Loop and J. Blinn, "Rendering vector art on the gpu", in *GPU Gems 3*, H. Nguyen, Ed., Addison-Wesley Professional, 2007, ch. 25, ISBN: 9780321545428.

[7] J. Blinn, *Jim Blinn's Corner: Notation, Notation, Notation*. Morgan Kaufmann, 2002, ISBN: 1558608605.

[8] A. J. Rueda, J. R. de Miras, and F. R. Feito, "Gpu-based rendering of curved polygons using simplicial coverings", *Computers & Graphics*, vol. 32, no. 5, pp. 581–588, 2008. DOI: https://doi.org/10.1016/j.cag.2008.07.005.

Appendix B

Control panel prototype

In April 2021 *Delegación de Alumnos ETSIST-UPM* organized a microcontroller competition. The author of this project participated with a very basic physical control panel for this vision mixer. Due to its tight relation with this project, a short report about the prototype is enclosed.

B.1 Overview

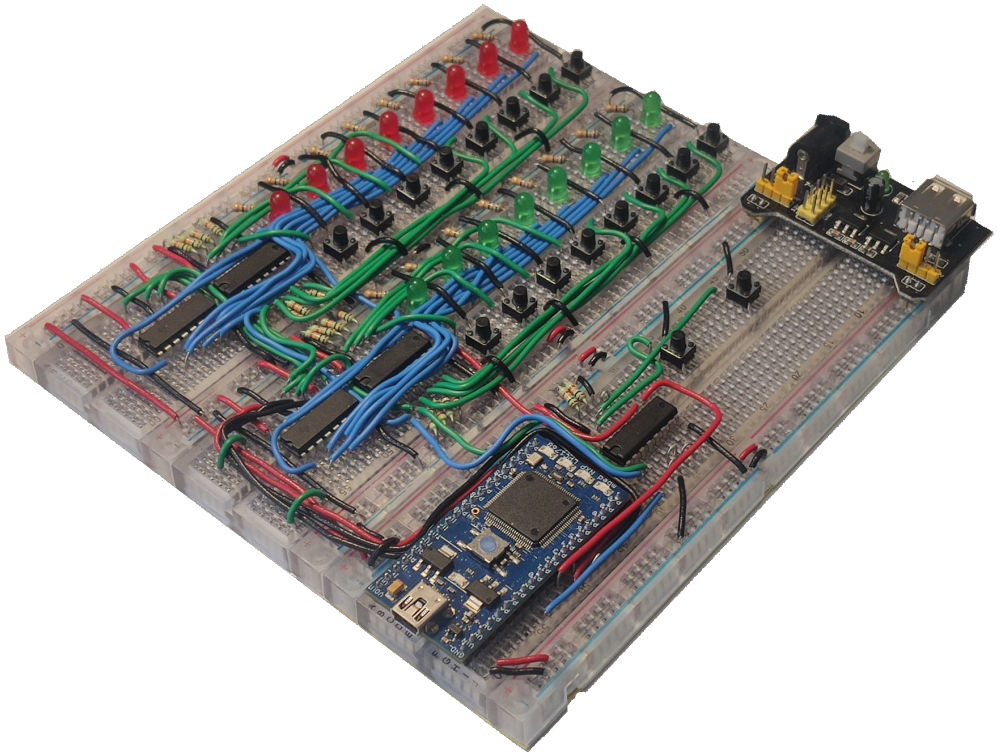
In the context of linear TV production equipment, a physical panel is a must for medium to large size productions, as it allows rapid interaction with the crucial parts of the system. Vision mixers are not an exception to this rule, so a prototype of a physical control panel has been developed that allows controlling the mixer described in this end-of-degree project.

Most commercial control panels feature dedicated sections for the cross-point and the transitions. More often than not, they also allow performing configurations related to the keyers. However, the prototype displayed in the figure B.1 does not enable the user to perform very complex tasks. It only features a 8 channel crosspoint and two buttons to perform transitions and cuts, with no *t-bar*. However, this comparison is not fair, as the bill of materials (BOM) of it costs around 100€, including the micro-controller, which corresponds to the 70% of the budget. As opposed to this, full-featured commercial solutions may cost orders of magnitude more.

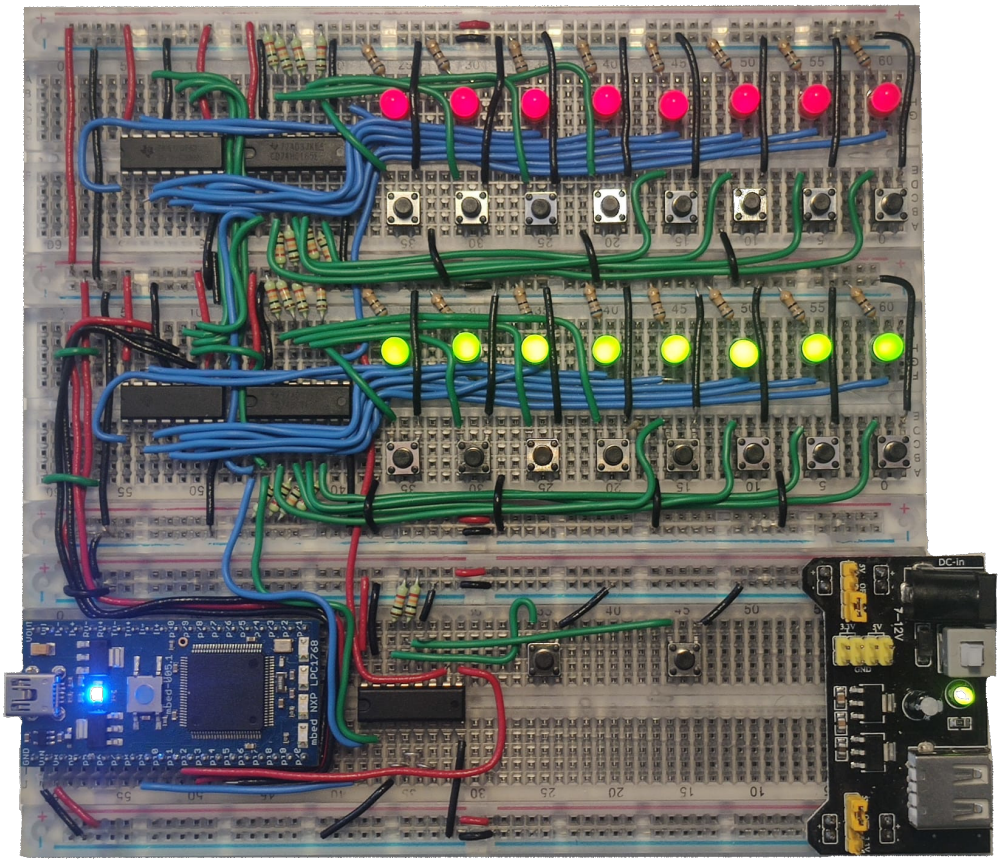
Due to the multi-client nature of the mixer, the absence of some controls on this panel is not a major problem, as the rest of the configuration is still available in the web control software. Therefore, the most used controls are easily reachable in a physical control panel, while the rest of the configuration may be done through the web-app.

B.2 Design

I/O requirements for a control panel may easily overwhelm the pin count of any micro-controller. Therefore, several serial in parallel out (SIPO) and parallel in serial out (PISO) registers have been used to reduce the pin count to 4 outputs and 1 input. Moreover, using a discrete not logic gate, one of those outputs could be reused elsewhere. In total, 2 74HC595 8 bit SIPO registers are used to handle 16 light emitting diodes (LEDs), whilst 3 74HC165 8 bit PISO registers are used for the 18 push-buttons. The implemented circuit diagram is shown in the figure B.2.



(a) Chopped shot



(b) Cenital shot

Figure B.1: Prototype of a physical control panel

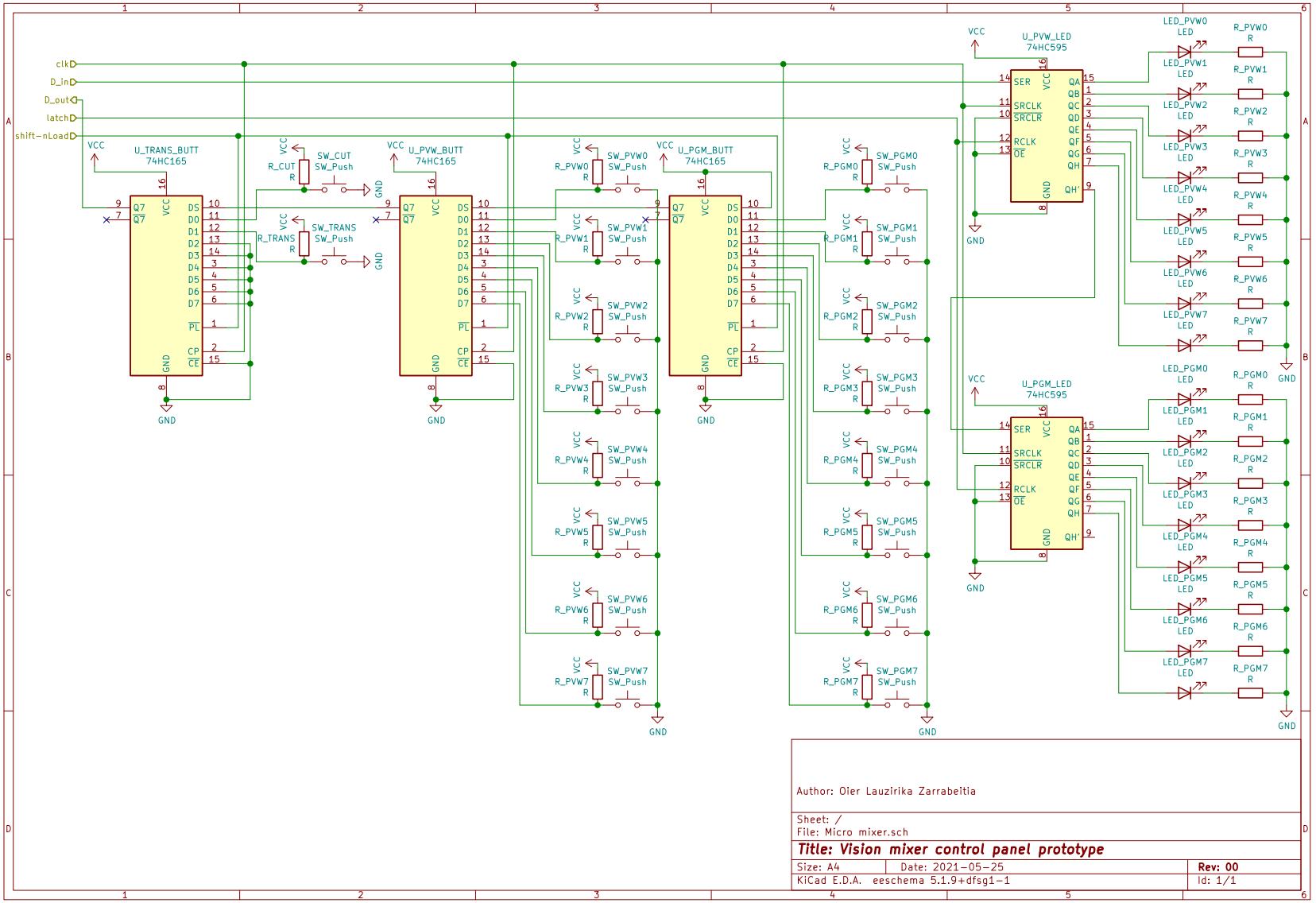


Figure B.2: Digital circuit blueprints for the control panel prototype

This control panel has been designed according to the competition's requirement of using a NXP LPC1768 microcontroller. However, in another context, this microcontroller should not be considered, as it is costly for the computational requirements of this control panel. The microcontroller only acts as a translation layer between the shift registers and the mixer. To communicate with the mixer, it uses the standard CLI protocol described in its corresponding appendix. However, these commands are transmitted using Universal Synchronous/Asynchronous Receiver Transmitter (USART) over Universal Serial Bus (USB). As the mixer does not support serial communications, a proxy must be set up, which forwards serial messages to a TCP socket and vice versa.

Appendix C

Manual

C.1 Introduction

Cenital is an open-source vision mixer. It has been developed considering a server-client architecture so that all the video processing is performed on the server and clients send commands to control its behaviour. This manual describes the installation and execution of the server and provides a reference of the available commands.

C.2 Installation

This software can be installed in two ways: Release and development. The first manner is advised for normal use. The second one should only be used when adding new features to the source code.

C.2.1 Release installation

The first step is to install all the dependencies. On Debian based systems this can be done using the following command:

```
sudo apt install\
libvulkan1 vulkan-validationlayers\
libglfw3\
libavutil56 libavformat58 libavcodec58 libswscale5
```

Copy all the shared libraries to the `/usr/local/lib/` directory and update registries:

```
sudo cp /path/to/provided/files/lib/* /usr/local/lib/
sudo ldconfig
```

Copy the executable to the `/usr/local/bin/` directory:

```
sudo cp /path/to/provided/files/bin/cenital /usr/local/bin/
```

C.2.2 Development installation

Firstly, install all the development tools required to build the binaries. In this example, the GNU C compiler (GCC) is used, but the reader may choose an alternative compiler such as Clang.

```
sudo apt install g++ cmake
```

Then install all the dependencies using the following command:

```
sudo apt install\
libvulkan-dev vulkan-validationlayers-dev glslang-dev\
libglfw3-dev\
libavutil-dev libavformat-dev libavcodec-dev libswscale-dev
```

Now proceed to build and install all the *Zuazo* modules. This can be automated using the following script:

```
cd /path/to/provided/files/src
for d in zuazo*/; do
    cd $d
    mkdir build
    cd build
    cmake ../
    make -j16
    sudo make install
    cd ../..
done
```

When a line of code inside one of these libraries is changed, it needs to be rebuilt using the following commands. Moreover, if a file is added or deleted, `cmake ../` must be executed after `cd`.

```
cd /path/to/provided/files/src/module/build
make -j16
sudo make install
```

Finally, build *Cenital* itself:

```
cd /path/to/provided/files/src/cenital
mkdir build
cd build
cmake ../
make -j16
sudo make install
```

C.3 Execution

If *Cenital* was installed inside one of the directories referenced by `$PATH` (as is the case when following the previous installation process) simply type the `cenital` command. If this is not the case, `cd` to the directory of the binary and type `./cenital`. Sometimes the firewall will reject listening to port 80 (the default port for WebSocket). The solution to this problem is explained hereafter.

C.3.1 Command line arguments

Similarly to other Unix commands, this accepts command line arguments to tweak the behaviour of the application. These arguments are the following:

- **-t** or **–tcp-port** is used to establish the TCP port for communications. The default value is 9600 and setting it to 0 disables it.
- **-w** or **–web-socket-port** is used to establish the WebSocket port for communications. The default value is 80 and setting it to 0 disables it. If the 80 port is rejected by the firewall, try changing its value, for instance to 9601.
- **-v** or **–verbose** is used to display debug information

C.4 Command reference

The *Cenital* mixer exposes a command line interface (CLI) which allows performing any supported action. Currently, this is the only way of controlling the mixer. This CLI has been implemented considering a multi-client architecture, this is, all the modifications performed by a client are broadcasted to the others, so that they can keep track of the state of the machine. The purpose of this section is to be a reference cheat-sheet for either manually setting up the mixer or developing applications that interact with it.

As mentioned earlier, currently this CLI is the only way to control the mixer. However, it is exposed using two different protocols. One of them is Web Sockets, which is a thin layer on top of HTTP that allows interaction with web applications. See Mozilla's documentation[1] for further references. A possible implementation in JavaScript is provided with the Web Control application's source code.

The other protocol is plain TCP, which pretends to be used by the rest of the applications and human managers. For developing applications that use it, refer to the chosen language's specific TCP socket library. For instance, for C++, Boost's ASIO is a good starting point. The simplest way for a human to connect to the mixer from a Unix machine is using the `nc` command:

```
$ nc <address> <port>
```

This will allow typing the commands referenced in this document and sending them to the mixer to perform manual changes. All the responses and state updates will also be displayed here.

C.4.1 Conventions

The commands described in this section follow some conventions. Therefore, to avoid repetitive explanations of these conventions, they will be explained once for all. The commands used here are pet examples and do not represent the real operation of the mixer, they just highlight a particular usage of the syntax. In all examples, a "+" at the beginning is to indicate that the message was sent to the mixer whilst a "-" signals that the message was received from the mixer.

This CLI is case sensitive. This means that the user must pay attention to the casing of the words. All commands are *kebab-case*, but a client may decide to name elements with any casing. In addition, enumerations have all their letters *camelCase*. Non ASCII characters such as tildes and "ñ"s have not been extensively tested. Avoid using them for increased stability.

Token separation

A command can be considered as a collection of tokens. A token can be thought of as a word, this is, a single distinct meaningful element. However, the term word will not be used as a synonym of token, as later their meanings will diverge. Similarly to other human readable languages, tokens are separated using spaces:

```
token1 token2 token3 token4
```

A token may be formed by multiple words. This is because the elements can be named arbitrarily. However, this presents the following dilemma, how to distinguish between two tokens and a two-worded token? The answer is that multi-worded tokens have an escaped space, this is, the character "\" is used to signal that the following space is not a token separator but a true space. However, this presents a new problem. What if that space is actually a token separator and the

[1] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

previous token ends with a "\" -quite a cumbersome case-. The solution to this problem is that the escape character should be in turn escaped.

```
This has four tokens
This\ is\ a\ single\ token
This\ token\ has\ a\ back\\slash
This\ token\ ends\ with\ a\ backslash\\ And\ this\ is\ another\ token
```

As a recap, to use a space inside a token, a "\␣" is inserted. Analogously, to use a backslash inside a token, a "\\" is used. This may seem familiar to Unix users, as it is the same strategy implemented on its command prompt.

From a logical perspective, tokens form a decision tree. Its root is on the left and each of the tokens specifies which branch to follow. This means that the correctness of a token and the number of remaining tokens is dependant on the preceding tokens.

Acknowledgment

If the first character of the first token starts with a #, it is considered to be an acknowledgment (ACK) identifier (ID). This means that the first token of the response will be this same token. The rest of the operation is the same but one token shifted to the right. Due to this convention, commands must not begin with a #.

```
+hello
-how are you?
+#1234 hello
-#1234 how are you?
+#multiple\ words\ as\ ACK\ ID hello
-#multiple\ words\ as\ ACK\ ID how are you?
```

Request responses

Requests can be one of two types: Queries and mutations. The difference is that the queries expect an answer back from the mixer. Meanwhile, the mutations broadcast themselves to all the clients in case they are successful. In any case, a response will be received from the mixer, indicating if it was successful (OK) or not (FAIL). The ACK will not be sent with the broadcast.

```
+light Living\ Room state get
-OK true
+#99 light Living\ Room state get
-#99 OK true
+light Living\ Room state set false
-light Living\ Room state set false
-OK
+#120 light Living\ Room state set false
-light Living\ Room state set false
-#120 OK
+light Living\ Room state set apple
-FAIL
+#123 light Living\ Room state set apple
-#123 FAIL
```

Special tokens

For any given position, there are three reserved tokens: `help`, `type name` and `ping`. The former one lists all the supported values for the proceeding token. `type` is used to obtain the class of the referred element and `name` returns its name. The later one is used to check the validity of the preceding tokens and returns `pong`.

```
+Sammy ping (There is no cat named as Sammy)
-FAIL
+Snow\ Ball ping (Snow Ball is the name of the referred cat)
-OK pong
+Snow\ Ball type
-OK cat
+Snow\ Ball name
-OK Snow\ Ball
+cat Snow\ Ball help
-OK help type ping meow purr eat sleep ignore-human
```

Attributes

An attribute is a variable which represents the state of a particular feature. Depending on its nature, a particular set of `set`, `get`, `enum`, `unset`, `add`, `rm` and `config` tokens will be available.

- `set`: Modifies the value of the variable. It usually requires a single proceeding token, which is the new value. As it is a mutation, if successful, the mixer will broadcast it with no ACK. It will also send a OK response with the ACK if present.

  ```
  +day set monday
  -day set monday
  -OK
  +day set july
  -FAIL
  ```

- `get`: Obtains the value of the variable. It usually requires to be the last token.

  ```
  +day get
  -OK monday
  +day get tuesday
  -FAIL
  ```

- `enum`: Obtains the possible values for this variable. It usually requires to be the last token.

  ```
  +day enum
  -OK monday tuesday wednesday thrusday friday saturday sunday
  +day enum tuesday
  -FAIL
  ```

- `unset`: Sets a null value for the variable. It usually requires to be the last token. If successful, the mixer will broadcast it with no ACK to all clients. It will also send a OK response with the ACK if present.

  ```
  +day unset
  -day unset
  -OK
  +day unset tuesday
  -FAIL
  ```

- `add`: If the options are dynamic, this command is used to add new elements. It is usually followed by the construction arguments.

```
+day add someday
-day add someday
-OK
```

- `rm`: If the options are dynamic, this command is used to remove an option. It is usually followed by an identifier of the element to remove.

```
+day rm tuesday
-day rm tuesday
-OK
```

- `config`: It is used to mutate the state of one of the enumerated types.

```
+day config monday festive set true
-day config monday festive set true
-OK
```

Attribute hierarchy

Some attributes may present a hierarchy. This is signaled using ":". From the token point of view, it will simply be a distinct single token, but it is worth pointing out what is its logical meaning.

```
+lamp set on
-lamp set on
-OK
+lamp:color set blue
-lamp:color set blue
-OK
+lamp:color:hue set 50
-lamp:color:hue set 50
-OK
+lamp:strobe set true
-lamp:strobe set true
-OK
```

Indices

All indices are zero based unless the contrary is specified.

```
+floor:count get
-OK 10
+floor set 0
-floor set 0
-OK
+floor set 10
-FAIL
```

Data types

Some tokens may require to be a particular data type. In such a case, the requirement will be pointed out in this documentation. These are the most used data types:

- `boolean`: The token must be either `true` or `false`.
- `integer`: The token must be a base-10 number without decimal places (.0 is not allowed). In case it needs to fit into a small bit count, it will be mentioned explicitly. Otherwise, consider it to be at least 32bits wide.
- `unsigned integer`: Same as above but negative numbers are not allowed.
- `number`: Any real number. Dot is used as a decimal separator.
- `duration`: Any duration in seconds (must end with "s")
- `rational`: `numerator/denominator`, where `numerator` and `denominator` are `integer`s. It does not need to be provided as a reduced fraction, although it will always be returned as such.
- `resolution`: `width×height`, where `width` and `height` are `unsigned integer`s
- `N-vector of type`: A comma separated array of `N` elements of `type`. A space after the comma is allowed, but remember that it needs to be escaped to be a single token.
- `quaternion`. Same as a `4-vector of numbers`. WXYZ ordering.
- `color`. Same as a `4-vector of numbers`. Normalized RGBA.
- `enumerated`: Only one of the values returned by the respective `enum` command. If the `enum` command always returns the same values, a hint will be provided.

C.4.2 Command list

Once the basic conventions have been introduced, it is time to showcase a list of all the real commands that can be ordered to the mixer.

Element management

The mixer can be thought of as a collection of elements that do something with video signals. To add a new element to the mixer the `add` command is used. The first token relates to the type of the element to be added. The possible values will be addressed in its corresponding sections. The next token is the name of the element, which needs to be unique, as it will be used to reference it. Depending on the type of the element, some additional tokens may be required. The keywords for each element type are shown in the table C.1

```
+add <element_type> <element_name> <optional_construction_tokens>
-add <element_type> <element_name> <optional_construction_tokens>
-OK
```

Type	Keyword
Output window	output-window
NDI Input	input-ndi
Media player	input-media-player
Mix effect	mix-effect

Table C.1: Keywords for element types

The `enum` lists all the added elements, regardless of their type.

```
+enum
–OK <element_name0> <element_name1> <element_name2> ...
```

To remove an existing element, `rm` command is used. It is followed by the name of the element to remove.

```
+rm <element_name>
–rm <element_name>
–OK
```

Meanwhile, the `config` command is used to manage the state of an object. It is followed by the name of the element to configure and the configuration tokens. These last tokens are type dependant, so they will be addressed in the corresponding sections.

```
+config <element_name> <configuration_tokens>
```

Signal routing

Each of the elements that constitute the mixer can be chained arbitrarily. Some elements will only offer inputs, some others will only offer outputs and some others may offer both. These I/O ports can be thought of as the physical connectors of a hardware device. An output may feed several inputs but an input can only be fed by a single output. Moreover, it can also be disconnected.

Inputs can be queried with the command `connection:dst enum` whilst outputs can be gathered with `connection:src enum`:

```
+connection:src enum <element_name>
–OK <output_port0> <output_port1> <output_port2> ...

+connection:dst enum <element_name>
–OK <input_port0> <input_port1> <input_port2> ...
```

To route an output to an input the `connection set` command is used:

```
+connection set <dst_element_name> <dst_port> <src_element_name> <src_port>
–connection set <dst_element_name> <dst_port> <src_element_name> <src_port>
–OK
```

To stop feeding an input, `connection unset` command is used:

```
+connection unset <dst_element_name> <dst_port>
–connection unset <dst_element_name> <dst_port>
–OK
```

Finally, the current connection can be queried with `connection get`

```
+connection get <dst_element_name> <dst_port>
–OK <src_element_name> <src_port>
–OK (If no feeding signal is set)
```

NDI Input

NDI inputs are used to listen to a remote NDI source and present it as a normal input to the rest of elements. The only configuration allowed is the NDI device to listen to. Moreover, video parameters can be queried, as illustrated by the table C.2. Naturally, all these commands are preceded by `config <ndi_input_name>`.

Output window

Output windows are the only way of outputting video from the mixer. However, they offer great flexibility so that they can be used in many circumstances. The commands offered for configuring them are shown in the table C.3. Clearly, those commands are preceded by `config <output_window_name>`.

Mix effect

Mix effects are used to manipulate video signals inside the mixer. The available commands are enumerated in the table C.4. Similarly to other elements, all these commands are preceded by `config <mix_effect_name>`. In this table, the keyer configuration has been obviated. This is because the upstream and downstream keyers are configured equally using the commands `config <mix_effect_name> us-overlay config <index>` and `config <mix_effect_name> ds-overlay config <index>` respectively. The particular configuration commands of the keyers are displayed in the table C.5.

Attribute/Command	Type	Attribute commands	Desription
source	enumerated	get	Remote NDI device used as a source.
video-mode:frame-rate	rational	get	Frame rate in frames per second.
video-mode:pixel-aspect-ratio	rational	get	Ratio between the physical width and height of pixels.
video-mode:resolution	resolution	get	Size of the resulting frames in pixels.
video-mode:color-primaries	enumerated	get	Color primaries used when compositing. Currently ignored.
video-mode:color-model	enumerated	get	Color model of the frames.
video-mode:color-transfer-function	enumerated	get	Gamma correction standard of the frames.
video-mode:color-subsampling	enumerated	get	Subsampling scheme of the resulting frame.
video-mode:color-range	enumerated	get	Range encompassed by component values.
video-mode:color-format	enumerated	get	Pixel storage format. It defines the color depth.

Table C.2: NDI input commands

Attribute/Command	Type	Attribute commands	Desription
video-scaling:filter	enumerated	get/set/enum	Filter used to scale the frames. Possible values: nearest, linear and cubic.
video-scaling:mode	enumerated	get/set/enum	Adjusting done on the frame when scaling. Possible values: stretch, box, crop, clampVertically and clampHorizontally.
position	2-vector of integers	get/set	Size of the monitor in screen coordinates (may not equal pixels).
size	2-vector of integers	get/set	Position of the monitor in screen coordinates (may not equal pixels).
title	text	get/set	Title shown at the heading of the window and task bar
resizable	boolean	get/set	Enable the user to resize the window.
decorated	boolean	get/set	Enable the decoration around the window.
opacity	number	get/set	Opacity used by the window manager when embedding this window in its environment. The value must lie in 0 to 1
monitor	enumerated	get/set/unset/enum	Monitor used to show this window full-screen. When set, the window is displayed full-screen on that monitor. Otherwise, the window behaves normally.
video-mode:frame-rate	rational	get/set	Frame rate in frames per second.

video-mode:pixel-aspect-ratio	rational	get	Ratio between the physical width and height of pixels.
video-mode:resolution	resolution	get	Size of the resulting frames in pixels.
video-mode:color-primaries	enumerated	get	Color primaries used when compositing. Currently ignored.
video-mode:color-model	enumerated	get	Color model of the frames.
video-mode:color-transfer-function	enumerated	get/set/enum	Gamma correction standard of the frames.
video-mode:color-subsampling	enumerated	get	Subsampling scheme of the resulting frame.
video-mode:color-range	enumerated	get	Range encompassed by component values.
video-mode:color-format	enumerated	get/set/enum	Pixel storage format. It defines the color depth.

Table C.3: Window output commands

Attribute/Command	Type	Attribute commands	Desription
video-scaling:filter	enumerated	get/set/enum	Filter used to scale the frames in the background. Possible values: nearest, linear and cubic.
video-scaling:mode	enumerated	get/set/enum	Adjusting done on the frame when scaling. Possible values: stretch, box, crop, clampVertically and clampHorizontally
video-mode:pixel-aspect-ratio	rational	get/set	Ratio between the physical width and height of pixels.
video-mode:resolution	resolution	get/set	Size of the resulting frame in pixels.
video-mode:color-primaries	enumerated	get/set/enum	Color primaries used when compositing. Currently ignored.
video-mode:color-model	enumerated	get	Color model used when compositing.
video-mode:color-transfer-function	enumerated	get/set/enum	Gamma correction standard of the result.
video-mode:color-subsampling	enumerated	get	Subsampling scheme of the resulting frame.
video-mode:color-range	enumerated	get	Range encompassed by component values.
video-mode:color-format	enumerated	get/set/enum	Pixel storage format. It defines the color depth.
us-overlay:count	integer	get/set	Upstream keyer count
us-overlay:ena	boolean	get/set	Visibility of a upstream keyer. The first token is the index of the keyer. Then the value.
us-overlay:transition	boolean	get/set	Transition of an upstream keyer. The first token is the index of the keyer. Then the value.
us-overlay:feed	integer	get/set/unset	Signal source of an upstream keyer. The first token is the index of the keyer. Then, the name of the input port (fillIn/keyIn). Finally, the input index.

ds-overlay:count	integer	get/set	Downstream keyer count
ds-overlay:ena	boolean	get/set	Visibility of a downstream keyer. The first token is the index of the keyer. Then the value.
ds-overlay:transition	boolean	get/set	Transition of a downstream keyer. The first token is the index of the keyer. Then the value.
ds-overlay:feed	integer	get/set/unset	Signal source of an downstream keyer. The first token is the index of the keyer. Then, the name of the input port (fillIn/keyIn). Finally, the input index.
input:count	integer	get/set	Input count for this Mix Effect unit.
pgm	integer	get/set/unset	Input index for program's background.
pvw	integer	get/set/unset	Input index for preview's background.
cut			This command swaps the contents of the program and preview buses.
transition			This command executes an automatic transition.
transition:pvw	boolean	get/set	If enabled, the transition occurs in the preview bus.
transition:bar	number	get/set	Manually set the progress of the transition.
transition:duration	duration	get/set	Duration of the transition when executed automatically .
transition:effect	enumerated	get/set/enum/config	Effect used in the transition. Possible values: Mix and DVE.

Table C.4: Mix effect commands

Attribute/Command	Type	Attribute commands	Desription
video-scaling:filter	enumerated	get/set/enum	Filter used to scale the frames. Possible values: nearest, linear and cubic.
video-scaling:mode	enumerated	get/set/enum	Adjusting done on the frame when scaling. Possible values: stretch, box, crop, clampVertically and clampHorizontally.
size	2-vector of numbers	get/set	Area encompassed by the keyer.
shape	See description	get/set	Cubic Bézier contours used to crop the frame. Each of the contours is a distinct token. The contours are defined as a semi-colon separated list of 2-vector of numbers. The continuity of the curve is ensured skipping the last point of each section. Therefore, the length of the list must be a multiple of 3.
transform:pos	3-vector of numbers	get/set	Position of the keyer.
transform:rot	quaternion	get/set	Rotation of the keyer.

transform:scale	3-vector of numbers	get/set	Scale of the keyer.
blending:mode	enumerated	get/set/enum	Blending mode used to mix the background with the contents of the keyer. Possible values: write, opacity, difference, differenceInv, lighten, darken, multiply and screen.
blending:opacity	number	get/set	Gain applied to the alpha channel.
linear-key:ena	boolean	get/set	Enable of the linear key.
linear-key:inv	boolean	get/set	Complementation of the linear key.
linear-key:ch	enumerated	get/set/enum	Source component of the linear key. Possible values: fillR, fillG, fillB, fillA, fillY, keyR, keyG, keyB, keyA and keyY.
luma-key:ena	boolean	get/set	Enable of the luma key.
luma-key:inv	boolean	get/set	Complementation of the luma key
luma-key:min	number	get/set	Lower threshold of the luma key.
luma-key:max	number	get/set	Upper threshold of the luma key.
chroma-key:ena	boolean	get/set	Enable of the chroma key
chroma-key:hue:center	number	get/set	Hue value in degrees of the colour to delete using the chroma key .
chroma-key:hue:span	number	get/set	Width of the interval of hues to delete using the chroma key.
chroma-key:hue:smooth	number	get/set	Width of the interval of hues to partially delete using the chroma key.
chroma-key:sat:min	number	get/set	Colour saturation threshold for the chroma key.
chroma-key:sat:smooth	number	get/set	Colour saturation threshold's transition interval for the chroma key.
chroma-key:val:min	number	get/set	Brightness threshold for the chroma key.
chroma-key:val:smooth	number	get/set	Brightness threshold's transition interval for the chroma key.

Table C.5: Mix effect commands

Appendix D

Web Control Manual

D.1 Introduction

Cenital Web Control is a web application that controls the *Cenital* vision mixer. This application exposes most of the mixer's features through an easy to use graphical user interface. This manual serves as a guide to familiarize with the previously mentioned graphical user interface

D.2 Installation

As this is a web application, there is no need to install any piece of software. A publicly accessible instance is available at https://oierlauzi.github.io/cenital-web-control/. However, to modify and deploy a modified version, some additional steps must be carried out.

D.2.1 Development and deployment of the application

The building process of the application is managed by `npm`. Therefore, this needs to be installed before proceding[1]. Once ready, go to the project directory (a folder named as `cenital-web-control` in the `src` directory) and install all the dependencies using the following command:

```
npm install
```

A development server may be started using the next command. This server is useful to rapidly prototype new features, but should not be used for real deployments.

```
npm run serve
```

The last step is to deploy the website. This can be done using the following command, which outputs a standard website structure to the `dist` directory. These contents should be placed on a web server, which can be self-hosted or in the cloud.

```
npm run build
```

D.3 User interface guide

When the web page is accessed, the site should look like in the figure D.1. Here, the URL of the server should be introduced, using the same URL notation. The port number in the URL can be ignored only if the server is exposing the WebSocket in port 80. Once connected to the mixer, elements can be added using the tabs in the navigation bar. The dialogs used to give an initial configuration to the elements are self-explanatory. However, keep in mind that the elements must

[1] https://docs.npmjs.com/cli/v7/configuring-npm/install

have a unique name.

D.3.1 NDI input

NDI inputs serve the purpose of ingesting live NDI video. Each element can be used to read a single NDI source. This source can be selected using the user interface displayed in the figure D.2.

Although the video parameters of NDI are controlled by the remote source, they can be read using this same graphical user interface. Some interesting metrics include the resolution, frame rate, YCbCr standard, color subsampling, etc...

D.3.2 Media player

Media players allow loading multiple video clips or still images. However, only one of those loaded clips can be outputted simultaneously. The clip on the output is selected using the drop-down shown in the figure D.3. When a clip is selected, there are many available controls for that clip, such as a play/pause selector, repeat mode, play speed and a modifiable progress bar.

D.3.3 Output window

Windows are currently the only way for outputting video from the mixer. The nature of the windows may vary significantly. As illustrated by the figure D.4, there are many settings that modify the appearance and behaviour of the windows, such as their size, position, head title, resizability, decorations, opacity, etc... Moreover, a window can be configured to be full-screen on a monitor attached to the host computer.

D.3.4 Mix effect

Mix effects allow compositing among many sources. They are based on the program and preview buses, which as their names suggest, they are used as the definitive and previsualization outputs. Each of the composition buses is formed by a background layer and many keyers. The contents of the background layers can be easily selected using the respective rows of the crosspoint shown in the figure D.5. Moreover, this crosspoint also serves the purpose of selecting the input signals routed to this M/E. To select the video feed of a keyer, the `AUX` delegation button must be selected in the respective control tab of that keyer, as displayed in the figure D.9. Then, the `AUX` row of the crosspoint can be used to feed the selected keyer input.

Many basic composition parameters can be configured in the settings tab shown in the figure D.6. Some of these parameters control the properties of the resulting frame, such as the resolution, color depth, etc.... Some others relate to the capabilities of the mixer, as is the case of the input, upstream keyer and downstream keyer counts. Finally, the video scaling parameters are used to configure the background layers' behaviours.

The `Cut` button can be used to instantaneously swap between the contents of the preview and program buses. In addition, a smooth transition can also be used, which can be manually executed using the *T-Bar* or in a predetermined amount of time using the `Auto` button. The nature of the transition can be selected using the transition effect type selector, which can be further configured on the respective accordion tab, as shown in the figures D.7 and D.8. Moreover, the configured transition can be executed in the preview bus by selecting the `Prev` button.

Each of the keyers' visibilities in the program bus can be manually toggled with the `Visible` button of the corresponding keyer control tab. Additionally, this toggle can be queued for the next cut or transition using the `Transition` button. This will instantaneously affect the preview bus.

Keyers can be freely positioned in space using the DVE transform parameters. Specifically, the position in pixels, the rotation in degrees and the scale can be manually set up. Moreover, a global opacity can be configured to make the overlay translucent. The effect used to achieve this transparency can also be configured using the `Blending mode` drop-down. Lastly, each of the keying techniques can be configured in their respective section. The meaning of each of these controls is explained in depth in the Description chapter of the report.

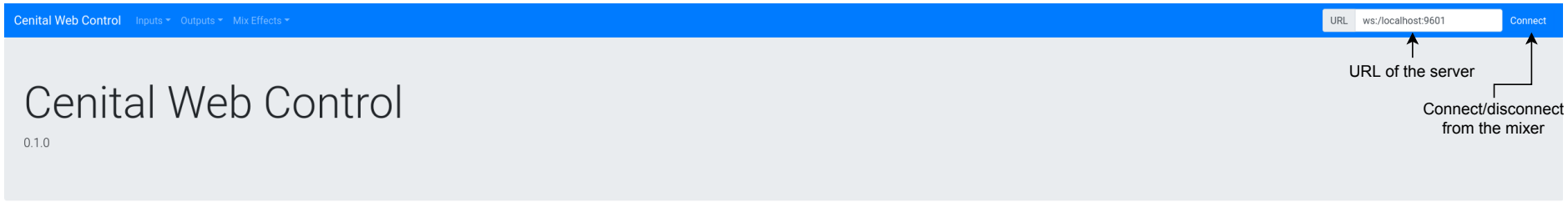


Figure D.1: Home page

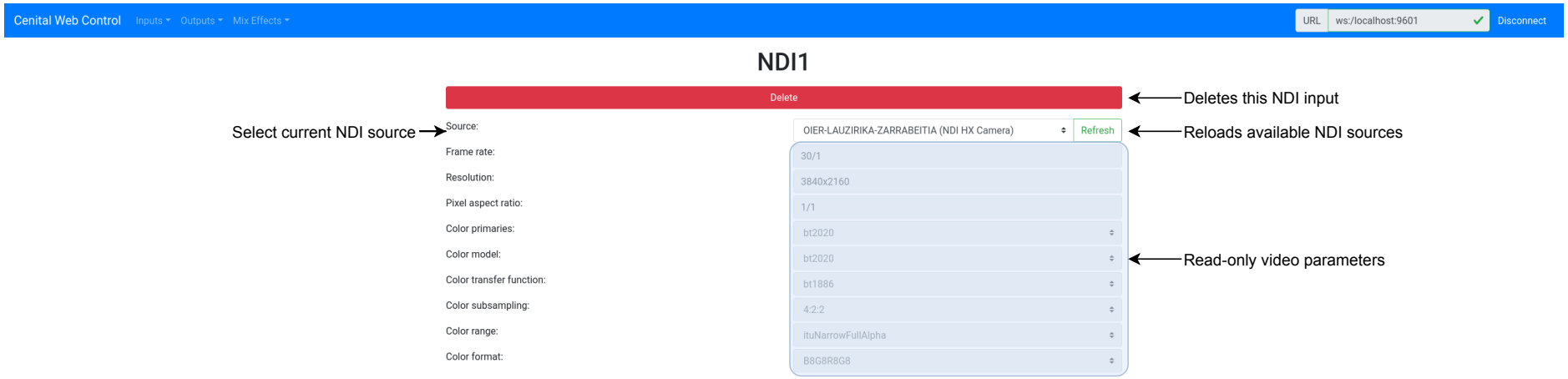


Figure D.2: NDI input settings

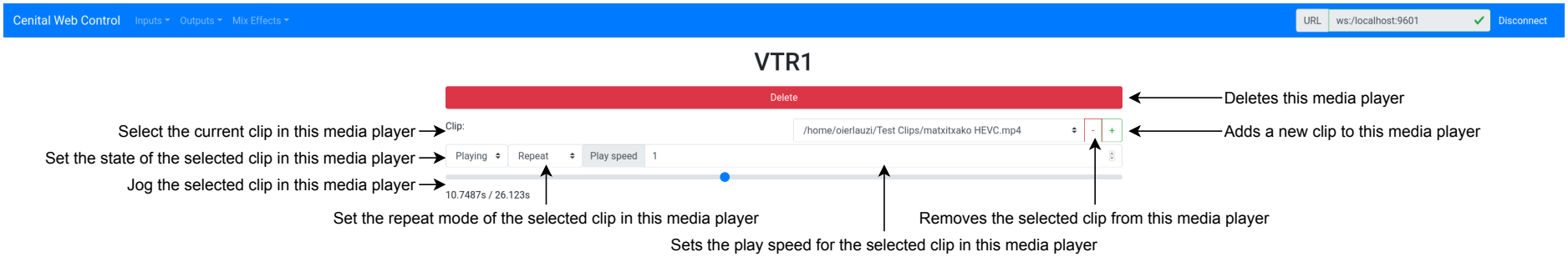


Figure D.3: Media player settings

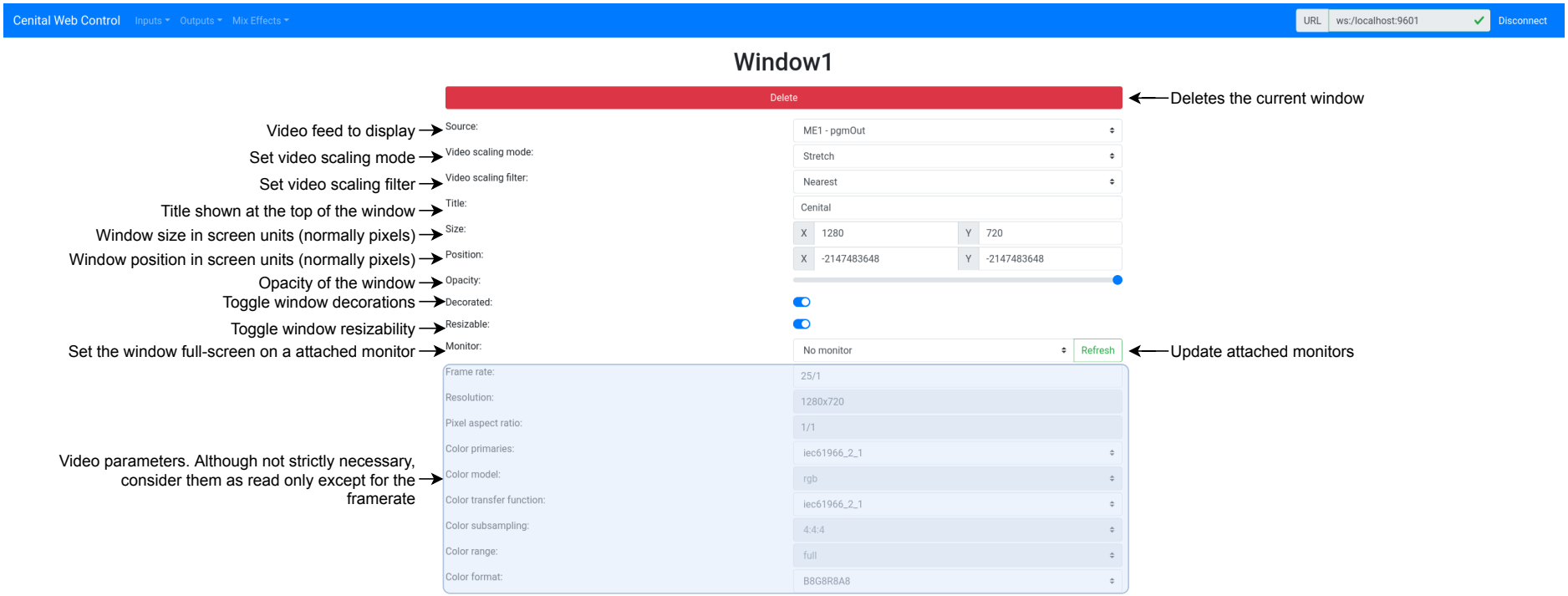


Figure D.4: Output window settings

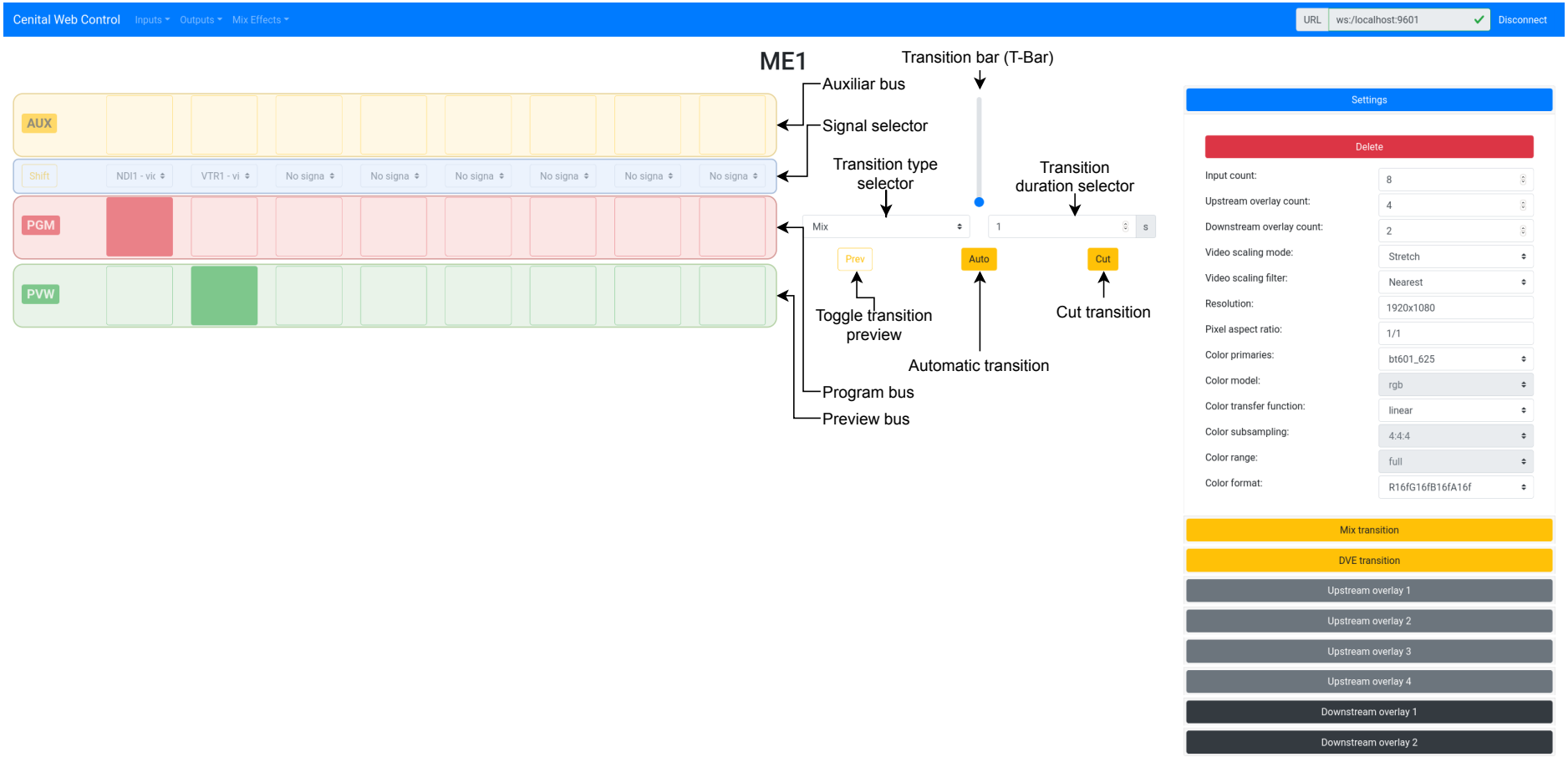


Figure D.5: Crosspoint settings

Figure D.6: Mix effect settings

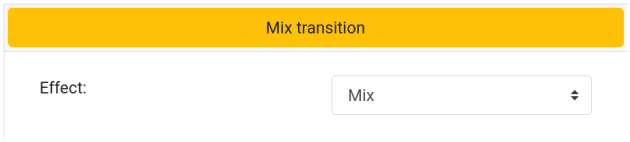


Figure D.7: Mix transition settings

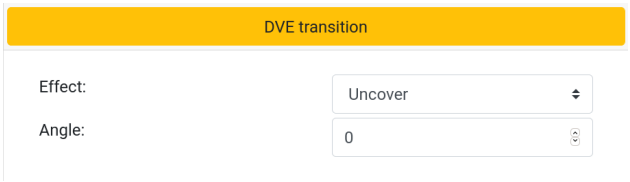


Figure D.8: DVE transition settings

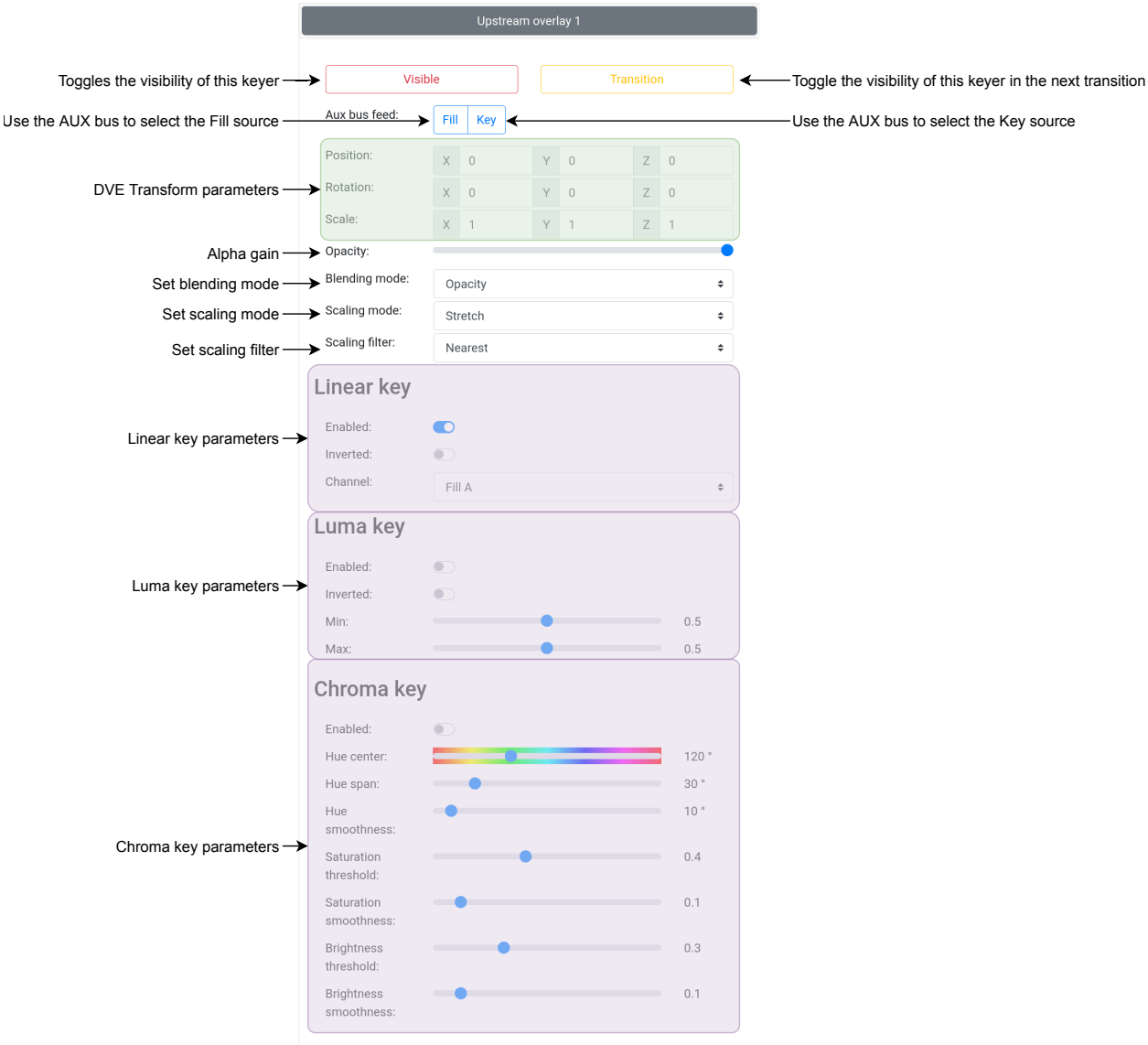


Figure D.9: Keyer settings