UiO : Department of Informatics
University of Oslo

A Real-Time Video Retargeting Plugin for GStreamer

Haakon Wilhelm Ravik

Master's Thesis Autumn 2016

Abstract

Video on demand and live streaming services have recently become a ubiquitous form of entertainment delivery. These services provide media content for a wide array of devices that differ in aspects such as processing power and screen size. Ensuring a consistent viewing experience across all devices is challenging, but adaptation mechanisms can assist with alleviating issues that stem from disparate viewing platforms. In particular, video retargeting can adapt video content for arbitrary screen sizes. While many exist, the majority of video retargeting techniques are not suited to adapt streamed video on demand or live content concurrently with presentation, and very few are readily available for consumers or programmers. This thesis focuses on altering one offline video retargeting algorithm to an online algorithm and implementing it as a part of a multimedia framework.

We present the design, implementation and evaluation of an on-the-fly video retargeting solution for video on demand and live streaming. Our work is motivated by making advanced real-time video retargeting readily available in a framework that is easy to use for both consumers and application programmers. This solution is as such implemented as a plugin in the GStreamer framework to provides video retargeting for arbitrary pipelines. We have compared and analysed two offline state-of-the-art video retargeting algorithms with respect to performance, memory usage and their ability to be altered for online retargeting. Based on this analysis, we have chosen the algorithm which is best suited to be utilized for our system - *Parallelized SeamCrop*.

To convert the offline algorithm to an online retargeting technique, we perform the retargeting process on predefined segment lengths called *frame windows* instead of processing the entire video in one session. This segmentation confounds the global camera optimization that the algorithm performs on the video, but is necessary to reduce memory usage and latency to acceptable levels for streaming. To deal with the resulting presentation disparities between two frame windows as a consequence of this separation, we perform a gradual transitional smoothing between their views over a subset of the frames in the newest frame window. This alteration allows the algorithm to be used for online retargeting at the cost of an additional computational step in the smoothing and more frequent virtual camera

movement when transitioning between frame windows.

We perform the evaluation of our plugin by using measurements on the performance of our implementation in a real adaptive streaming scenario. Measurements are done with different adaptive bitrate streaming techniques to demonstrate compatability with these techniques.

Our evaluation shows that our plugin is capable of performing video retargeting at 25 FPS with different retargeting severities for several different resolutions without exceeding a 2000 millisecond initial latency. We also show that longer video segment sizes equate to higher retargeting rates when compared to smaller video segment sizes, and that the computational time required by our additional smoothing step is negligible. We have managed to reduce the width of a 640x360 resolution video with up to as much as 44% in real-time. Through these experiments, we have demonstrated that our plugin is compatible with both DASH and HLS within the context of the GStreamer framework and consequently that it is usable for real-time adaptation of video on demand and live streaming content with these adaptive bitrate techniques.

Acknowledgements

I would like to express my gratitude to my supervisors, Thomas Plagemann and Francisco Javier Velazquez, whose guidance and advice have been indispensable for the completion of this thesis.

I want to thank my fellow students and friends that have taken the time to proofread and discuss the thesis with me.

I am also grateful to Stephan Kopf, Professor at the Mannheim University, for providing me with the source code for the Parallelized SeamCrop algorithm. Without it this thesis would not have been feasible to complete within the given time frame.

I would like to thank my family for their continued support and words of encouragement, especially in times where the tasks at hand seemed insurmountable.

Finally, I am grateful to Mia Cecilia Sandstrøm, whose patience, support and encouragement during my writing has been a continuous source of motivation.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In this chapter, we look at the background and motivation for this master thesis. Section 1.1 provides background information for the thesis topic and Section 1.2 details the motivations for the thesis work. In Section 1.3 we present the requirements for the output of our work, while Section 1.4 describes our approach and methods. Finally, Section 1.5 outlines the thesis in its entirety.

1.1 Background

In today's world, video on demand and live streaming services are rapidly becoming a ubiquitous form of entertainment delivery with an ever increasing growth. Services such as Netflix and YouTube provide multimedia content to millions of users daily through a multitude of heterogeneous devices such as smart phones, desktop computers, laptops and tablets. The content provided by these services is typically available in multiple formats, allowing a video on demand or live streaming application to dynamically adapt to the presentation environment through a set of adaptation mechanisms. In this context, adaptation mechanisms are processes that alter one or more aspects of the video stream in response to environmental characteristics such as bandwidth fluctuations or device properties.

Currently, the majority of adaptation mechanisms used for both video on demand and live streaming are concerned with content delivery to ensure a consistent viewing experience. As most modern devices differ in capabilities such as screen size and processing power however, the quality of experience may suffer from factors that are unrelated to delivery, such as video content that is ill suited for presentation on the device. This can for example occur when there is an aspect ratio disparity between the device screen and the video, or that the device screen is simply too small to adequately present the video. The traditional approach to this issue in a video on demand or live streaming context is to either crop the con-

tent so that it conforms to the devices capabilities or present it in a letterboxed format. Both of these approaches detract from the viewing experience by either removing portions of the content or minimizing the presentation. The field of video adaptation, specifically video retargeting, has been extensively researched over the last decades and multiple different sophisticated approaches for adapting content to different screen sizes without removing or obscuring important content exist. However, few of these retargeting techniques have as of yet been widely employed for either video on demand or live streaming.

1.2 Motivation

The majority of current video retargeting techniques are offline algorithms. In contrast to an online algorithm, an offline algorithm requires the entire input from the beginning to produce output, while an online algorithm can process input piece by piece [8]. This effectively means that an offline video retargeting technique requires the entire video to produce adapted output. In a video on demand context, such techniques can be used to adapt video content for arbitrary screen sizes on a server prior to delivery, but are of limited use for client side adaptation due to the input requirement. Additionally, many of these techniques are not able to provide concurrent adaptation and presentation, precluding them from on-the-fly content adaptation.

While offline techniques can be used in this manner to adapt video content for different screen sizes in advance, they are incompatible with live streaming. Since live streamed content is continuously generated and delivered in real-time from a source media, the video stream does not have a fixed length and cannot be adapted by such algorithms. The vast majority of retargeting techniques are as such of limited viability for adapting video on demand content and incapable of adapting live streamed content.

Although some online video retargeting techniques exist, only a few of these approaches are feasible for real-time retargeting of video on demand and live streaming due to either technical or design limitations which negatively impact the viewing experience. The main limiting factors are either the quality of the output or an inability to retarget video in real-time, i.e., producing adapted output at a rate equal to or higher than the frame rate of the video. In addition to this, utilizing them as a consumer or programmer is difficult due to the complexities of each method and the lack of readily available implementations.

Developing an adaptation mechanism that can perform on-the-fly online video retargeting is as such beneficial to provide a good quality of experience regardless of device screen size for both video on demand and live streaming content. Making such a mechanism widely available would be valuable to bridge the gap in

viewing experience that stems from aspect ratio differences between the content and the viewing platform when consuming these forms of streaming media.

As multimedia applications have recently surged in popularity, we implement this adaptation mechanism as a part of a multimedia framework to provide an easily accessible video retargeting system to both consumers and application programmers. We have converted an offline video retargeting technique capable of real-time video retargeting to an online algorithm and incorporated it into the GStreamer framework as a plugin. This video retargeting plugin allows users to adapt video streams for arbitrary screen sizes without discarding salient content. This facilitates real-time video retargeting for both video on demand and live streaming content, on either server or client, along with the comprehensive tools of the GStreamer framework. We claim that this adaptation mechanism enhances the user experience when streaming video to devices with a different screen aspect ratio than the video being streamed.

1.3 Requirements

Our principal goal is to determine if our system is capable of performing real-time retargeting of video content in an internet streaming context. The following requirements apply for our system:

1. **Performance** Our plugin should be able to perform on-the-fly retargeting of continuously streamed video content from a remote server. It needs to retarget videos at the same or higher rate than the frame rate of the video being streamed, to not impact the viewing experience.

   Videos produced for television or cinema typically operate at a frame rate of at least 25 frames per second [9]. In order for our plugin to provide retargeting support for such videos, it must be able to match this frame rate for one or more resolutions used in video on demand streaming.

2. **Latency** The plugin must not incur significant latency between retargeting initiation and the retargeted output of the video being streamed. Viewers grow impatient and are more likely to abandon a video if the startup delay exceeds 2 seconds [10], so the incurred latency should not exceed this threshold. However, this study also shows that users are more tolerant of latency at the onset of streaming as opposed to buffering mid stream. Therefore our requirement is slightly flexible given an acceptable trade off between the two, but a startup latency below 2 seconds is preferable.

3. **Resource usage** The machine resource consumption of the plugin should not be so severe as to prevent the machine from regular usage alongside plugin operation.

4. **Format independence** Our plugin should be able to retarget incoming videos regardless of their format.

5. **Adaptive bitrate streaming compatibility** Our plugin must be able to handle adaptive bitrate streams that dynamically vary in bit rate, frame rate and resolution. As a majority of popular video on demand and live streaming services such as Youtube and Netflix utilize such approaches, this plugin must be able to retarget streams received through at least one such standard.

1.4 Methods

The methods used in this thesis are design, implementation and evaluation. As this thesis is practical in nature, the design is subject to small changes over the course of implementation as unexpected difficulties might emerge. We investigate and compare several state-of-the-art retargeting algorithms, choosing the algorithm best suited for conversion to online retargeting as well as the requirements listed in Section 1.3. We design an alteration of an offline retargeting system to facilitate online retargeting and implement it within a plugin for a widely available multimedia framework. The results are gathered from experiments performed with the plugin and are evaluated by comparing average values against the requirements for each respective metric presented in Section 5.1.

1.5 Outline

This thesis is organized as follows: Chapter 2 provides an introduction to the retargeting algorithms we considered to use in our system as well as the multimedia framework and streaming techniques we utilize. Chapter 3 describes the design and goals of our system while Chapter 4 presents the implementation of our design. The implementation is evaluated in Chapter 5 based on the requirements presented in this chapter. Chapter 6 concludes our work and discusses issues that can be solved in future work.

Appendix A lists the Git repository containing the source code of the GStreamer plugin, measurement applications and the original source code of the retargeting algorithm along with information about how to compile and deploy the plugin.

Chapter 2

Background and Related Work

In the first part of this chapter, we analyse and detail retargeting methods that are used for adaptation of video content, presenting two different approaches in Sections 2.1 and 2.2. Section 2.3 details the GStreamer multimedia framework we utilize in this thesis. Finally, Section 2.4 presents techniques used for streaming media content.

Over the past decades, many different retargeting methods have been proposed and developed. These methods tend to differ in multiple aspects, from their core assumptions to the techniques utilized to perform the retargeting. Despite the pervasive heterogeneity, most retargeting methods are variations on a set of basic operations; either removing pixels from an image, merging the pixels of an image or a combination of the two. These methods are classified as either *discrete*, *continuous* or *hybrid* methods, respectively [9]. Figure 2.1 visualizes typical approaches associated with each of these methods.

The majority of modern retargeting techniques are complex approaches to these methods. These techniques can roughly be classified as either cropping, seam carving, warping or multioperator methods [11]. Cropping and seam carving are discrete methods as they remove pixels from an image, warping is a continuous method by merging pixels while the latter is equivalent to higher level hybrid methods. An in-depth account of these methods is outside the scope of this thesis [12].

Currently, there is no retargeting approach which excels in every quantifiable aspect [13]; each approach has its own relative strengths and weaknesses. In [14], Krähenbühl et al. present an efficient system which achieves high quality real-time video retargeting for multiple resolutions through a combination of multiple algorithms. However, their approach relies on interactive annotation of the video to maintain coherence in scenes with several salient objects, otherwise degenerating to linear scaling. This annotation involves manually specifying regions of the

Figure 2.1: Different retargeting methods. Left: Cropping (*discrete*). Center: Scaling (*continuous*). Right: A combination of both methods (*hybrid*). The faded borders illustrate the original image prior to the application of a method.

video frame that the system should regard as important; a degree of interactivity which precludes it from use in an on-the-fly video retargeting context. To realize real-time, online retargeting in a multimedia framework plugin, we require a retargeting method which is fast, produces high quality output and does not rely on significant user interaction during the retargeting process apart from initial parameter specifications.

While there are several offline techniques that fit these criteria, converting them to provide online retargeting poses a set of nontrivial difficulties. Many of these approaches rely on a global optimization for the entire video to produce a palatable result [3][15], which is incompatible with the continuous provision and presentation of content in either video on demand or live streaming. Additionally, they can be quite memory intensive due to these global optimizations, requiring considerable resources even for small videos, which is problematic for long video sequences. Such approaches are classified as "video cube based algorithms", i.e, algorithms operating on a large number of frames simultaneously [16]. These techniques generally yield high quality results, but must be altered in some way to be used for online video retargeting in a multimedia streaming context.

Based on a preliminary comparison of available retargeting algorithms, we present and analyse two such state-of-the-art methods that provide real-time retargeting for multiple resolutions and can feasibly be altered to provide online video retargeting: *Parallelized SeamCrop* [2] and *Scalable and Coherent Video Resizing with Per-Frame Optimization* [1]. These methods are described in detail in the following sections.

2.1 Scalable and Coherent Video Resizing with Per-Frame Optimization

In [1], Wang et al. present a novel content-aware video retargeting method utilizing a combination of cropping and warping, developed with OpenMP to benefit from CPU-based parallel processing. It emphasizes a balance of motion consistency and shape preservation as well as scalability for high resolution videos without compromising the temporal coherence.

This method can be separated into three sequential steps:

1. Spatial frame resizing
2. Motion trajectory analysis
3. Per-frame retargeting

2.1.1 Spatial Frame Resizing

In the first step, an individual frame is resized with the content-aware scale-and-stretch method from [17] to preserve the per-pixel correspondence of salient objects between the original and resized frame. Preserving the pixel correspondence is essential to maintain consistent spatial shapes of objects between the frames, which is required for optimizing their motion pathlines in the next step. Gradient magnitude of pixel colors, optical flow vectors and face detection is utilized to compute the saliency map used to guide the resizing. This step only accounts for a portion of the total retargeting as the resulting frame is transformed to a size which may be larger than the desired width.

2.1.2 Motion Trajectory Analysis

While the previous step preserves the spatial shapes of objects, the motion information may be distorted by either stretching or compressing as each frame is resized independently. This step corrects the motion pathlines by optimizing the offset deformation between neighboring pathlines, encouraging the optimization towards constant scaling. This is performed as a global optimization for the entire video. The optimization is a combination of temporal coherence and the spatial shape preservation from the previous step. Essentially, it attempts to strike a balance between where the object shapes *are* in the resized frames and where they *should be* based on the original frames. The motion pathlines are visualized in Figure 2.2.

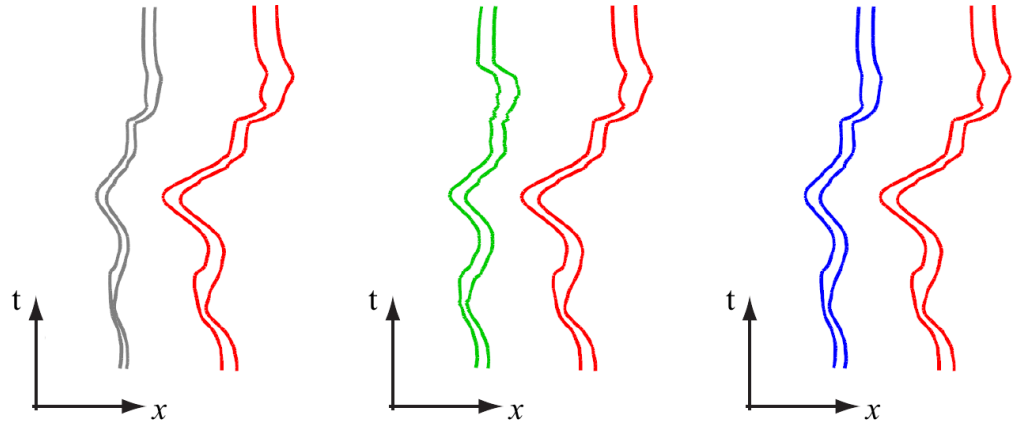


Figure 2.2: Optimization of motion pathlines across frames (t) and neighboring pathlines (x). Red: original Gray: linearly scaled Green: per-frame resized Blue: optimal. The horizontal offsets are reduced in both linear scaling as well as the optimized pathlines [1].

2.1.3 Per-Frame Retargeting

The final step of the method consolidates the optimized pathlines into one coherent video by repeating the content aware retargeting performed in step 1. In this repeated retargeting, the locations of the pathlines for each frame are added to the warping energy of each frame *t*.

These three steps constitute the primary retargeting technique in this approach. For frames which are densely populated with prominent objects, use of this technique degenerates to linear scaling. In such cases, this approach resorts to utilizing a cropping technique based on the one presented in [18] to perform the retargeting. This technique warps the frame to a natural width, pans the virtual camera to include the most critical region of the frame and crops the edges outside of this critical region. The retargeting process for this technique is illustrated in Figure 2.3.

The algorithm is performed with variable grid mesh sizes, which are quad grids overlaid on each video frame used to seed motion pathlines. A small grid mesh size results in more precise retargeting at the cost of increased memory usage and processing time. Conversely, a large size results in a more coarse retargeting result but is faster and less memory intensive.

Performance The algorithm produces a consistent output rate ranging between 80-100 FPS on a PC with a Core i5 2.66 GHz CPU and 8GB of RAM. The author does not disclose the resolution of the test sequence or the grid

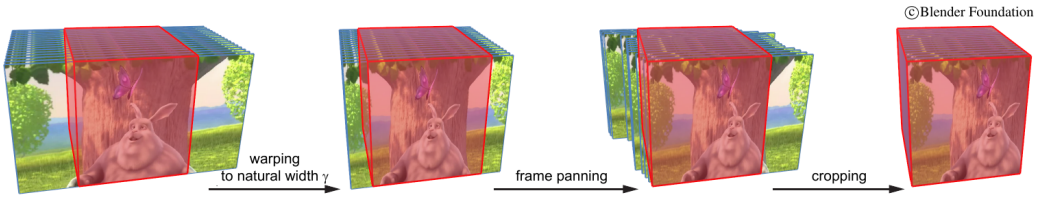


Figure 2.3: The cropping and warping process. The target video cube is depicted in pink [1].

mesh size used for this experiment, but as they state that their technique scales linearly with resolution this rate is likely to linearly decrease as the resolution increases.

Alterability This algorithm relies on globally optimizing the motion pathlines for the entire video to ensure consistent retargeting. This optimization approach must be altered if it is to be compatible with both video on demand and live streaming.

Memory consumption Memory usage is directly tied to the grid mesh size as well as the pixel resolution. Each individual grid mesh size exhibits consistent memory consumption for a configuration, but the memory utilized increases considerably along with the grid mesh size. For a video with a resolution of 688x288 pixels and 224 frames, a grid mesh size of 20x20 reports a peak memory usage of 22 Mb while a 3x3 grid mesh consumes 1.2 Gb for the same case.

2.2 Parallelized SeamCrop

Parallelized SeamCrop is a retargeting algorithm aimed at real-time adaptation of video streams, using a combination of cropping and seam carving. It builds upon the efficient SeamCrop [3] technique, utilizing the GPU to significantly enhance the performance of the algorithm. The algorithm reduces the dimensions of a video cube from $m \times n$ to $m' \times n$, removing the least salient content from each individual frame. Salient content in this context refers to the regions of the image that the importance function assigns a high energy value.

The algorithm itself can be separated into three sequential phases:

1. Energy calculation
2. Cropping window path computation
3. Seam carving

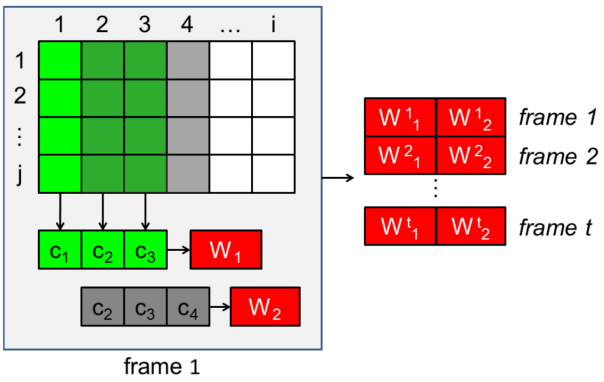


Figure 2.4: Energy values in each column are summed up first. The column cost values are then summed up for each cropping window position before combining them into a 2D array [2].

Energy calculation and seam carving are operations executed on the GPU, while the cropping window computation is performed by the CPU. The algorithm performs two passes on the video; once for the energy calculation and once for removing the identified seams.

2.2.1 Energy Calculation

The algorithm begins by searching for an optimal cropping window path of a target size *m'* over the course of the whole video. The relative energy of each pixel is measured by way of motion saliency and gradient magnitude. Motion saliency detects moving objects whose motion is discontinuous to the background [19]. It is measured by comparing the difference of the pixel value between the preceding, current and following frame. Gradient magnitude indicates how quickly and in which direction the image is changing most rapidly [20], measured by normalizing the length of its' gradient to [0..1]. The energy of an individual frame is computed through a weighted combination of these measurements, favoring the motion saliency.

These energy values effectively map the image into a two-dimensional matrix where each pixel has an associated energy value. The values of each column in this matrix are summed up to create an associated column cost c for each column $i = 1, .., m$. These column costs are then used to determine the total energy W_i contained within each possible cropping window position $i = 1, .., (m - m' + 1)$ for each frame. This results in a two dimensional array containing the total energy for each possible position of the cropping window for each frame of the video. Figure 2.4 illustrates the interactions between these computations.

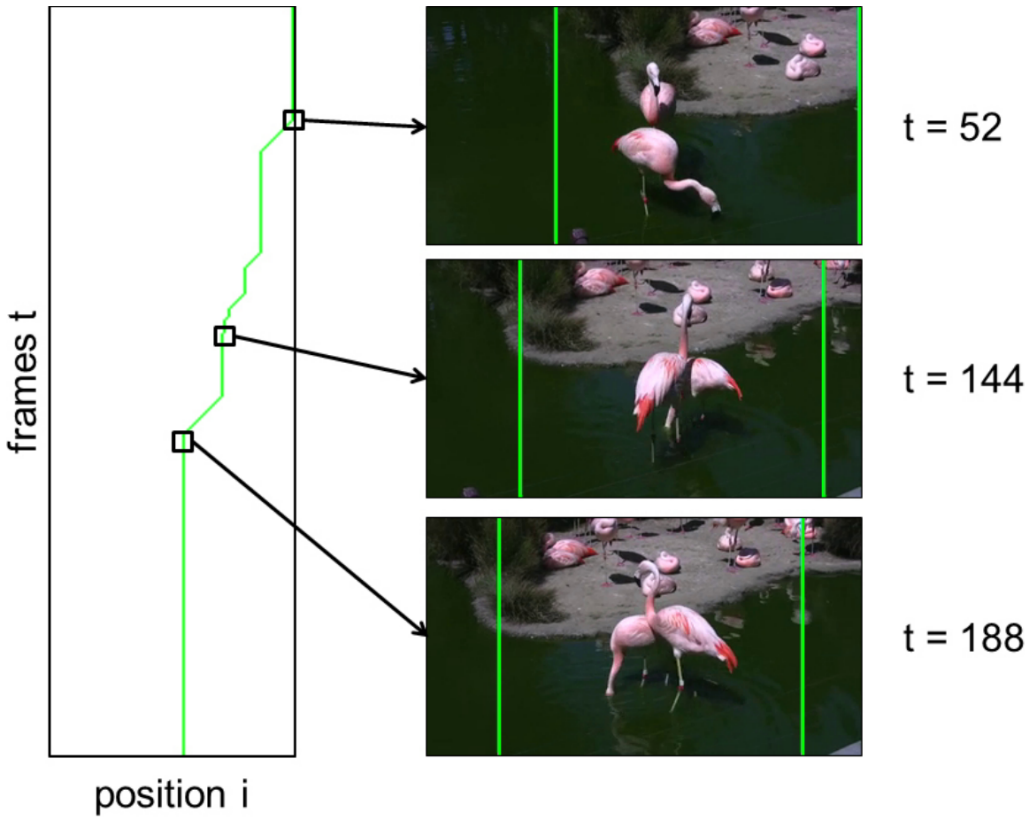


Figure 2.5: Path of the cropping window over time. Each point represents the position of the cropping window in the corresponding frame [3].

2.2.2 Cropping Window Path Computation

A cropping window in this context is the effective video window of target size *m'* that the retargeting will produce. This window can be visualized as a virtual camera that pans over the source video, globally optimized to contain and follow as much of the salient content as possible throughout the length of the video. The frame-by-frame position of this window is decided based on the energy map W described in the previous section.

The path is searched for and identified by using dynamic programming on the 2D array with similar restrictions as in the seam carving algorithm in order to find the path with the maximum energy [21]. Details regarding this search are explained in the next section. The energy of a path is the sum of the energy values in all path positions from frame $t = 0, .., T$. It is determined by traversing the W array along the time axis t, starting from and continually choosing the maximum adjacent energy value. When the last row is reached ($t = T$), it has

Figure 2.6: Left: Original image. Center: Seams found. Right: Image after removal of seams [3].

found a path with the lowest energy cost. Backtracking from the cheapest position in the last row yields the optimal window path through the video sequence. Figure 2.5 illustrates a possible cropping window path for a video sequence.

While the paths found by this algorithm are temporally connected, they are not impervious to inter-frame jitter. This typically occurs when the maximum energy values of adjacent cropping window positions oscillate back and forth, causing the window to jump between these few positions. To mitigate the effects of this jitter, the computed positions are smoothed with a Gaussian filter.

2.2.3 Seam Carving

Seam Carving is a technique that can be described as a form of dynamic cropping, removing pixels of an image to obtain an image with reduced resolution. In contrast to cropping however, the pixel removal is not performed indiscriminately on the borders of an image. Pixel seams, either horizontal or vertical, depending on the desired aspect ratio, are removed from within the image itself. This removal is based on an energy mapping of the image, such as the one described in the previous section. In the original approach, seams with lowest cumulative energy values for each pixel are removed in ascending order until the desired image resolution is reached. Figure 2.6 illustrates the application of Seam Carving on an image.

Each individual seam is computed by initially choosing a pixel at either the top row or leftmost column of an energy map spanning the image. The starting position is determined by the direction of the seam to remove and the energy of the pixel. The pixel with the lowest value of its row or column is typically chosen, as it marks the end of an optimal seam. This example will focus on a seam extending from top to bottom.

From the identified starting position, the pixel with the lowest energy value of the three adjacent pixels in the next row is chosen. This step is continually performed until the bottom row of the image is reached, at which time a seam has been found, and this process is repeated until enough seams have been found.

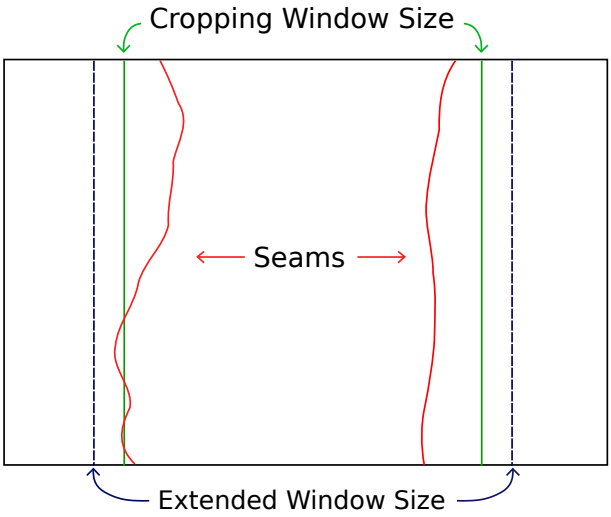


Figure 2.7: Interaction between seam carving, the cropping window and the extended window.

Note that a seam can move between columns, but cannot move more than once per traversed row as this would introduce discontinuous seams.

2.2.4 Seam Carving in Parallelized SeamCrop

Parallelized SeamCrop utilizes a modified approach to the selection of seams. When performing seam carving on video, seams have to be chosen with a temporal concern in mind. As frames in a sequence need to be temporally coherent, each seam depends upon the seams previously calculated in the current frame *t* as well as the corresponding seam from the preceding frame $t-1$. A temporal coherence cost is used to favour seams that are in close proximity to the relevant seam in the previous frame, linearly increasing outward from the position of said seam. The intent is to utilize seams that are more temporally coherent rather than the optimal ones. Choosing seams in this manner limits the introduction of artifacts within a sequence of frames, ensuring consistent representation of objects across them.

The calculated cropping window position for a frame might remove the edges of important visual objects despite being in the optimal position. To circumvent this issue, the borders of the cropping window are extended by a predetermined factor during the identification of seams. The extended width, illustrated in Figure 2.7, is expressed as $m' + ((m - m') * extendfactor)$. This allows the search for seams to include as much of the salient objects as possible into the cropping window.

Before the search for a seam commences, the energy map of the frame is re-

calculated for the extended cropping window. This calculation is only performed once per frame. Each search for a new seam on the same frame utilizes the same map, preventing duplicate use of pixels by artificially increasing the energy values of all pixels utilized in the previous seam. Each column is assigned its own thread so that each pixel in a row can be computed independently. Since the rows depend on each other, every row has to wait until the previous one has finished. Once finished, the cheapest value in the last row is identified as it marks the end of the optimal seam.

During the search, each thread concurrently calculates seams for a given frame, waiting until the seams they require from the preceding frame are identified. For instance, if a seam *i* should be calculated for frame *t*, the thread yields until a thread calculating seams for frame $t - 1$ has at least identified seam $i + 1$ before resuming. This process repeats for each frame until enough seams have been found.

The algorithm is performed by executing each of the previously described phases once. The degree of retargeting is determined by a retargeting factor, expressed as a float value between 0 and 1. The target width m' is obtained by multiplying the original width with this factor, expressed as $m' = m * retargetfactor$.

Performance The output rate of this algorithm is highly dependent on two factors: video resolution and retargeting factor. The experiments in this paper were performed on a PC with the following specifications: Intel i7-3770 CPU with four cores at 3.4 GHz, 16 GB DDR3 RAM and a NVIDIA GeForce GTX 650 TI with 1024 MB memory and 768 CUDA cores. With a resolution of 480x270 and a retargeting factor of 25%, a retargeting rate of 52 frames per second is achieved. This rate decreases as both resolution and retargeting factor increases. Assuming 25 frames per second, the algorithm achieves real-time retargeting up until a resolution of 720x405 pixels with a retargeting factor of 25%.

Alterability The algorithm performs a global optimization of the cropping window path across the entire video. In order to be compatible with video on demand and live streaming, this optimization must be altered or replaced.

Memory consumption Memory usage is not reported in the paper, but as it requires the video in its entirety to perform the retargeting, the memory consumption is likely to increase linearly with the resolution of the video and its length.

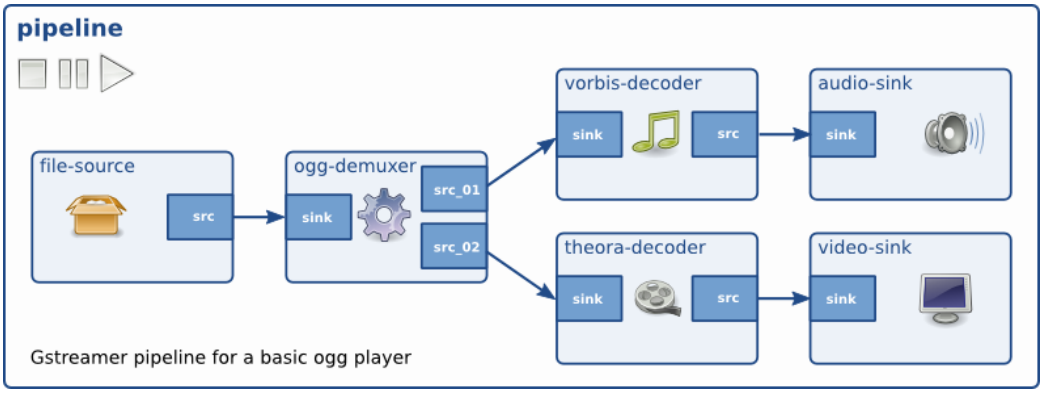


Figure 2.8: Example GStreamer pipeline [4].

2.3 GStreamer

GStreamer is a pipeline based open-source multimedia framework that can be used to easily create multimedia applications for a vast variety of purposes. The core principle of GStreamers design is the creation of pipelines which define, produce, or otherwise modify a data flow. These pipelines are formed by linking various elements together, creating a chain of modules which operate on the data. Elements are either included in the framework itself or provided by third party programmers. The framework manages these elements by directing data flow and negotiating formats between them. Despite being labeled as a multimedia framework, it can handle any type of data stream.

2.3.1 Elements

Elements are the building blocks of a Gstreamer pipeline. Each element has a specific behavioral pattern, where the cumulative patterns of the pipeline define the application behaviour. In general, there are three types of elements: sources, filters and sinks, as illustrated in Figure 2.8. Each of these types exhibit a set of defining characteristics.

Sources

Sources are elements concerned with producing data, typically set as the first element in the pipeline. There is a wide array of different sources for different types of data, but their role is the same. Examples of such elements are *videotestsrc*, which can produce video data in a variety of formats, *audiotestsrc*, which generates audio on a specified frequency and *filesrc* which reads data from a file.

Filters

Filters are elements that alter received data, such as applying a visual effect to an incoming video stream or modify the volume of an audio stream. The role of these elements is typically to either modify the data for consumption or act as an intermediary between elements that do not have any formats in common. Decoders, muxers and protocol handling elements fall under this banner. An example of a simple filter is the *videoscale* element, which scales the dimensions of a video stream from an input resolution to a different output resolution.

Unique to filter elements is that they are linked to multiple other elements, both upstream and downstream. This allows elements which split data streams to forward separate streams to differing receivers. An example is the *avidemux* element, which separates an AVI stream into a video stream and an audio stream. These two streams can later be combined into a single stream with a muxer such as the complementary *avimux* if desired. The *ogg-demuxer* element in Figure 2.8 is one such element.

Sinks

Sinks are the principal consumers of the Gstreamer pipeline. Data sent from a source element upstream will eventually arrive at a sink element, which typically outputs data to either a video display, sound card or harddrive. Sink elements cannot produce data, and are as such always pipeline endpoints.

In addition to these element archetypes, we have an ancillary element type which acts as a container for other elements.

Bins

Bins are elements used to encapsulate a set of linked elements into one logical entity. When multiple elements are combined in this way, they are referenced as a single unit rather than being handled individually. This is used to provide an abstraction for a logical segment of a pipeline, such as consolidating all elements related to a video stream into a single element. The pipeline itself is a specialized type of bin which manages synchronization and bus messages between the contained elements.

Each pipeline must contain at least one source element and a sink element - producer and consumer. When constructing a pipeline with these elements, it defines the behaviour of the enveloping multimedia application.

2.3.2 Plugins

Plugins are preprogrammed sets of one or more elements that can be used in a pipeline. An element cannot be utilized in a pipeline without first being encapsulated in a plugin, so a plugin is essentially a shipped version of one or more elements. They typically encompass a specific sequence of execution with a set of elements, but may also just contain a single basic element. As such, plugins vary greatly in content and complexity. Plugins are distributed as dynamically linked libraries or shared object files which can be directly plugged into a pipeline.

2.3.3 Pads

Pads are the I/O interfaces of an element, used to communicate and transfer data between entities in the pipeline. They manage negotiation of the formats between the elements, restricting the types of data one can produce or receive. There are two types of pads, *sourcepads* and *sinkpads*, which fulfill the same roles as their names indicate: sending and receiving. While elements typically have one of each, some may have multiple of either, as in the case of *filter* elements. However, an element must have *at least* one pad which either produces or receives.

2.3.4 Properties

The vast majority of GStreamer elements have customizable properties. These properties are unique to each element, where they typically either alter its behaviour or determine its internal state. This functionality is common for all GStreamer elements. Each element derives from a *GObject*, which provides the necessary functions for setting properties via *g_object_set* or *g_object_get* in an application.

To illustrate, consider a small scale pipeline consisting of two elements: *videotestsrc* and *autovideosink*. *videotestsrc* produces a video stream mainly used for testing purposes which can be customized through various properties, while *autovideosink* is a plugin which automatically chooses an appropriate sink element for viewing a received video stream.

In Figure 2.9 we can see two different pipelines. For the leftmost picture, we have set the *pattern* property of the *videotestsrc* element to *circular*, making it produce a circular black and white pattern. In the second picture, we set the same property to *smpte100*, where it produces a color bar pattern for color testing.

2.3.5 Communication

GStreamer provides a handful of mechanisms for exchanging data and general communication between pipeline and application as well as from one element to

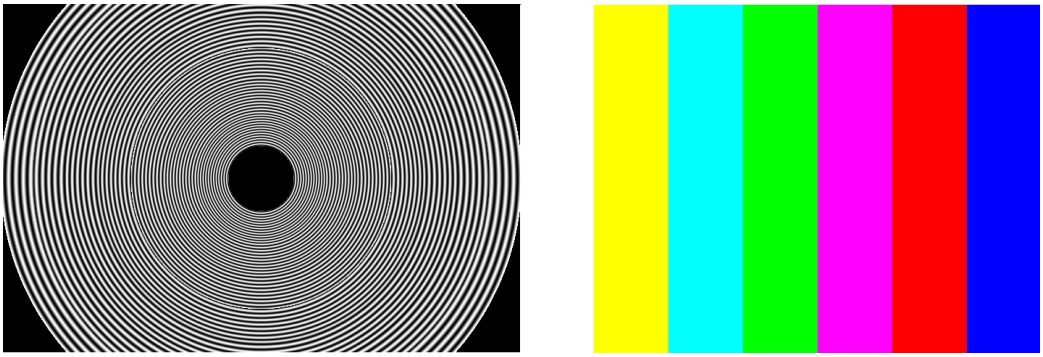


Figure 2.9: Output of two GStreamer pipelines with differing *pattern* properties for *videotestsrc*.

neighboring elements in the pipeline. Figure 2.10 illustrates the different communication flows in a Gstreamer pipeline. A GStreamer application is an application which creates elements, packs them together in a pipeline and plays the pipeline. The application can influence the operation of the pipeline by sending events and queries to elements. All communication is facilitated through the following mechanisms: events, queries, messages and buffers.

Events

Events are objects used for general communication between both elements and an application and its elements. These objects can travel both upstream and downstream, depending on the actual event that occurred. Downstream elements might need to notify an upstream element of some event, such as a user seeking through a video, while upstream elements will have to notify downstream with events such as the end of a stream. Events that travel upstream always travel out-of-band, i.e., travelling through the pipeline instantly, while events that travel downstream, such as format changes, can be synchronized with the data flow.

Queries

Similar to events, queries are used to communicate between upstream and downstream elements. However, instead of providing information, they are used to ask an element for specific information, such as the state of an element, the duration of a video stream or the capabilities of the receiving element. Queries are always synchronously answered due to the possibly time sensitive nature of the queried information, such as querying the current position of the stream. If an element cannot handle a query, it is sent further up/downstream until it reaches an element that can handle it.

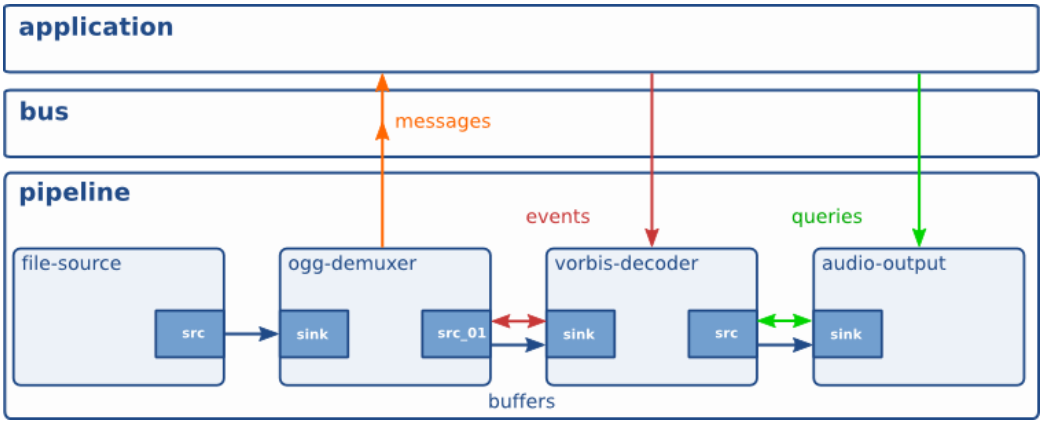


Figure 2.10: Communication flows between elements in a GStreamer pipeline and from an application to the elements in a pipeline [5].

Buffers

In GStreamer, buffers are objects used to exchange streaming data between elements. A buffer is an allocated space in memory whose location is exchanged through pointer passing. These objects exclusively travel downstream, as a data stream never flows from a sink element to a source element.

2.3.6 Capabilities

As previously mentioned, GStreamer can be used to stream any form of data. While there are no restrictions imposed by the framework itself, each element is required to describe what types of data it can handle. This information is stored in a *GstCaps* structure, where 'caps' is shorthand for capabilities. *GstCaps* are contained in the pads of an element, and individual pads of an element can differ in what types of data they support. A filter element for instance can transform the data between its sinkpad and sourcepad, requiring separate input and output caps. These capabilities are either statically set in the pad template or dynamically negotiated between one element and another.

2.3.7 Negotiation

Capabilities negotiation is the process of identifying media formats that two neighboring elements have in common and deciding which format to use. In some cases there is only one common format, while other times there might be a plethora of different options. GStreamer differentiates between three separate types of negotiation.

Fixed Negotiation

This type of negotiation is used when an element has only one format it can receive and produce. Usually, this format is statically fixed in the code itself, which prevents any possibility of renegotiation. The peer elements are forced to accept this format if the pipeline is to function at all. An example of fixed negotiation is the presence of a source element which can only produce a certain media format.

Transform Negotiation

This technique imposes a transformation between the input caps and the output caps, where the output format depends on the input format and alternatively some element properties. Negotiation of this type is typically used to convert between a fixed upstream format and a fixed downstream format. Elements like decoders or encoders are common transform negotiators that do this, while elements that only inspect the data stream without changing the format also utilize this technique. Such elements operate in *passthrough* mode, which intuitively passes the format from the input caps to the outgoing ones.

Dynamic Negotiation

Dynamic negotiation is the most powerful negotiation technique, but entails greater complexity. Elements utilizing this technique can convert fixed input caps received on its sinkpad into multiple formats that can be negotiated with the next downstream element. The sourcepad of the element will have to choose an output format from a range of possibilities, usually one that the element downstream can accept. Queries are used to discover the capabilities of the downstream element, and the format is chosen thereafter. Ideally, the format that requires the least effort to produce is chosen. Examples of elements that use dynamic negotiation are *videoconvert* and *audioresample*.

2.3.8 Negotiation Process

In Gstreamer, pads are always the driving force behind the data flow of the pipeline. There are two separate scheduling modes that the pads can operate in: Push mode and pull mode.

Pads operating in pull mode request data from the upstream element preceding it, while pads in push mode directly send buffers downstream. Only sinkpads can operate in pull mode, and conversely, only sourcepads can operate in push mode. Data cannot flow downstream without a producing elements sourcepad pushing data or a receiving elements sinkpad requesting data. There are advantages and

disadvantages with both modes, but the mode to use is largely determined by the needs of either an element or the application. Push mode is useful for situations where data needs to be transmitted as fast as it can be produced, pull mode for when a receiving element can't utilize data as quickly as it is produced, requiring control of the input flow.

The negotiation process for each of these scheduling modes is similar, where pull mode negotiation is slightly more complex. We will focus on push mode negotiation, as it is most prevalent.

Push mode negotiation operates as follows, illustrated in Figure 2.11:

1. Element 1 queries downstream Element 2 about the formats it can receive. As the elements might have differing capabilities, a common format must be identified.
2. Element 2 responds with a list of formats that it can handle.
3. Element 1 compares the formats in the returned list with its own internal list, choosing a format that is suitable for both elements.
4. Element 1 asks Element 2 whether the chosen format is acceptable.
5. Element 2 responds with either a confirmation or rejection.
   (a) If Element 2 rejects the format, Element 1 returns to step 3 to choose a different format
6. If Element 2 accepts the format, Element 1 instructs Element 2 to prepare to receive the chosen format.
7. Element 1 begins transmitting data.

Renegotiation

Renegotiation is initiated by a downstream element that wishes to receive a different format from a negotiated pipeline. The catalyst for renegotiation is typically a result of changes on the pipeline sink which cannot be handled by that particular element. An example can be that the size of the window presenting a video changes, where the presenting element is not capable of performing the scaling itself. A request for a new format that matches the dimensions of the new window is sent upstream, using a RECONFIGURE event. Depending on the active negotiation mode in the upstream elements, they will react differently to this event.

An element operating under *fixed negotiation* will drop the event, as the element is not able to alter its own format. Furthermore, since its output caps do

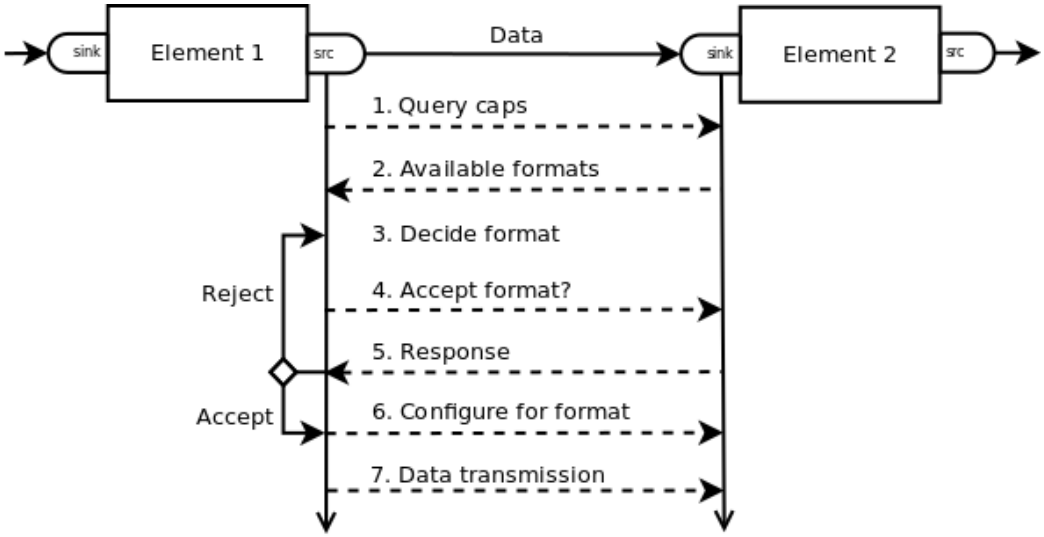


Figure 2.11: Push mode capability negotiation between two pipeline elements.

not depend on any caps upstream, the event can safely be dropped. Elements operating under *transform negotiation* will forward forward the event upstream, as their output caps depend on the caps further upstream. When the request arrives at an upstream element operating under *dynamic negotiation*, the element checks whether it can produce the new format. If so, it restarts the negotiation process detailed in Figure 2.11.

2.3.9 Element Structure

This section describes the basic structure of a filter element from a plugin writer's point of view. Most plugins are created as a subclass of a GStreamer base class that provides supporting functionality to simplify development. The internal functions of an element depends on the base class, but the most important functions are listed in Table 2.1.

Capabilities

The capabilities of an element are statically assigned in a structure called a *pad template*. This template defines the formats that the pads of the element can accept or produce, the direction of the pad and the name of the pad. Each pad has an associated template, i.e., source or sink, as the element might be able to accept different formats than it can produce. These templates are set by the plugin writer based on the restrictions of the element.

Function	Description
plugin_init	registers the plugin with the GStreamer framework
gst_class_init	one time initialization of the class
gst_init	initializes an instance of the class
get_property	gets a property of the element
set_property	sets a property in the element
event	handles incoming events
query	handles incoming queries
chain	handles incoming data buffers
generate_output	generates output to be inserted into the pipeline

Table 2.1: Internal functions of a GStreamer filter element.

Metadata

Each element is described by additional metadata that is statically assigned during plugin initialization. This metadata provides extra information about the element and is comprised of four fields:

- the name of the element,
- the element type,
- a brief description of the element,
- name of the author and email address.

2.4 Streaming Techniques

There are multiple different approaches to media streaming. Traditional streaming generally uses a stateful protocol such as the Real-Time Streaming Protocol (RTSP) [6]. When a client connects to the streaming server with this protocol, the server keeps track of the client's state until it disconnects. Once a session has been established, media is sent as a continuous stream of packets over either TCP or UDP. This typically entails frequent communication between the client and server.

In contrast to traditional streaming techniques, modern streaming techniques utilize HTTP to stream media. An important benefit of HTTP streaming is that HTTP packets rarely have issues with passing through routers and firewalls in comparison to traditional techniques. HTTP is stateless, where each request issued by the client is handled as a standalone one-time transaction. Utilizing HTTP for streaming is not a new concept, where HTTP progressive download is widely used for media delivery from standard HTTP servers. However, there are multiple disadvantages with this approach, such as lack of support for live media services, poor bandwidth utilization and absent bit rate adaptation.

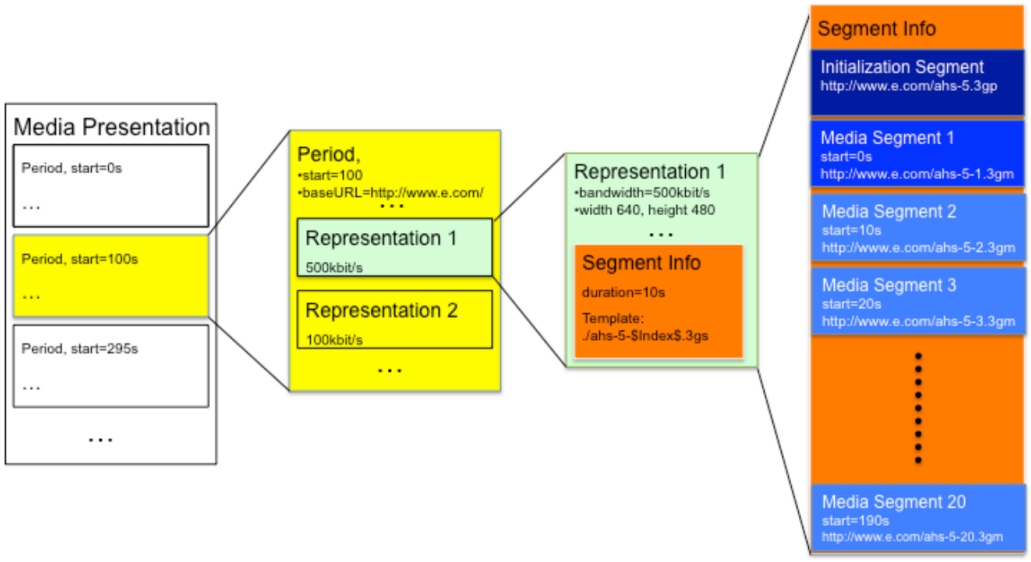


Figure 2.12: Structure of a Media Presentation in DASH [6].

Newer approaches such as Dynamic Adaptive Streaming over HTTP [22] or HTTP Live Streaming [23] provide stateless communication in addition to addressing the aforementioned disadvantages of HTTP progressive download [24]. Other approaches such as Adobe HTTP Dynamic Streaming [25] and Microsoft Smooth Streaming [26] exist, but as they are proprietary they are not discussed in this section.

2.4.1 Dynamic Adaptive Streaming over HTTP

Dynamic Adaptive Streaming over HTTP, henceforth referred to as DASH, is a HTTP streaming protocol standard aimed at solving the complexities of media delivery to multiple heterogeneous devices. DASH is a fairly new adaptive bitrate streaming technique and the first of its kind to be ratified as an international standard. It supports both live and video on demand content and is currently adopted by services such as YouTube and Netflix.

The technique breaks down media content into small file segments, alternatively referred to as chunks, each containing a short interval of playback time. These segments are made available in a variety of different encoded bit rates and resolutions which are stored along with a media presentation description (MPD). The MPD is metadata describing the relation of the segments and how they form a media presentation. The media presentation referenced by the MPD is a structured collection of encoded data of some media content, illustrated in Figure 2.12.

In DASH, the client is in full control of the streaming session, managing on-

time requests and ensuring smooth playout of the segment sequence. Streaming is initiated by downloading the MPD, through which the client requests segments with HTTP GET or other partial GET methods. Each chosen segment is typically the highest quality option that can be downloaded in time for playback without incurring re-buffering or stalling. This permits seamless adaptation under fluctuating bandwidth conditions at the client's discretion, ensuring the highest quality playback possible.

2.4.2 HTTP Live Streaming

HTTP Live Streaming, also known as HLS, is a HTTP streaming protocol similar to DASH. As it was developed prior to DASH, it has served as one of the progenitors to the technique. While it lacks some features such as HTML5 support and codec agnosticism when compared to DASH, it is still employed despite being less feature rich than its more recent counterpart.

HLS consists of three parts: the server component, the distribution component and the client software, as illustrated in Figure 2.13. The server component is responsible for segmenting input streams into different encodings and encapsulating them in a suitable format. The distribution component consists of standard web servers that are responsible for accepting client requests and delivering the media. The client software is responsible for requesting the appropriate media, downloading resources and reassembling them for presentation.

As with DASH, HLS separates media content into segments of predefined length, typically containing 10 seconds of playback content. The metadata for these segments are stored in an m3u index file, sometimes also referred to as a playlist file, which is functionally identical to the MPD in the DASH approach. It specifies the location of each segment with either an absolute path name, relative path name or a URL. Streaming is initiated by downloading this m3u manifest, through which the client subsequently requests individual segments via HTTP GET methods.

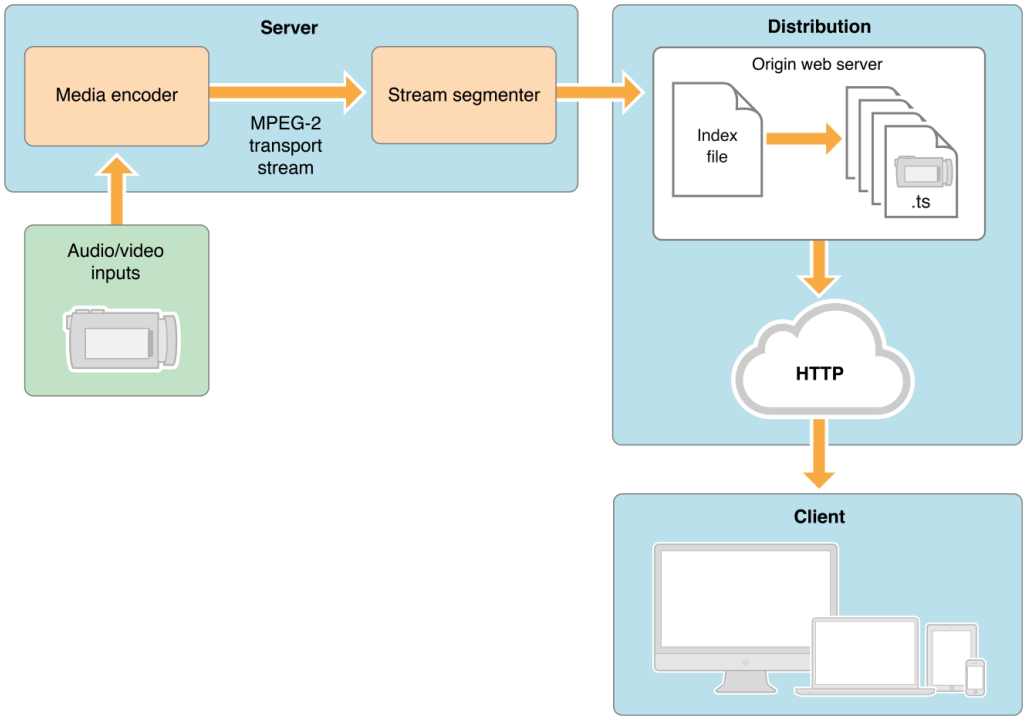


Figure 2.13: Architecture of HLS [7].

Chapter 3

Design

In this chapter, we present the design of our video retargeting plugin which receives a stream of raw video frames and produces as output retargeted frames with altered aspect ratio and size. In Section 3.1 we describe our goals as a product of the requirements we outlined in Chapter 1. Section 3.2 details the reasoning behind the choice of retargeting algorithm. In Section 3.3, we combine the retargeting algorithm and the GStreamer framework by presenting the design of our plugin.

3.1 Goals

Our principal goal is to make a plugin that can be directly inserted into a GStreamer pipeline to perform on-the-fly adaptation of video on demand and live streaming content. Through this plugin, a user is able to view adapted video content on a screen that is smaller than the intended viewing platform of the original video.

Our first goal is to choose a retargeting algorithm that is best suited for our requirements. The next goal is to implement and wrap the chosen retargeting algorithm within a GStreamer plugin.

The outcome of our work is a video retargeting plugin. No knowledge other than that of the GStreamer framework and the parameters to the plugin itself should be required to utilize it in a pipeline or use it in future application development.

3.2 Retargeting Algorithm

One of the tasks is to choose a suitable retargeting algorithm to perform the video adaptation. Since our plugin should function as a wrapper for the algorithm, it inevitably inherits the traits of the approach. As properties such as performance

or memory usage can either make or break the plugin, careful consideration is required.

In Chapter 2, we introduce and analyse retargeting algorithms with respect to the requirements we presented in Chapter 1. In this section, we compare the two algorithms with particular emphasis on alterability. We have chosen the retargeting algorithm which best fulfills these requirements.

Performance *Scalable and Coherent Video Resizing with Per-Frame Optimization*, henceforth referred to as *SCVR*, exhibits the best performance of the two algorithms. In the paper, the technique achieves an output rate averaging approximately 80 FPS. While the resolution of the test sequence or the quad grid size for this experiment is not reported, the rate is higher than any of the rates achieved by the *Parallelized SeamCrop* approach and scales linearly with resolution. *Parallelized SeamCrop* achieves 52 FPS for a 480x270 resolution test sequence and at most 25 FPS for a resolution of 720x405. Despite the difference, both approaches are viable in terms of performance as our requirement is real-time retargeting at 25 FPS for at least one resolution.

Alterability Neither of the analysed techniques are able to provide online retargeting without adjustments. Both approaches rely on global optimizations for the entire video to produce the intended output, which must be altered in order to be viable. The least invasive adjustment to these algorithms is to perform the optimization on predefined segment lengths as opposed to the entire video. In this respect, *Parallelized SeamCrop* is less problematic to customize than *SCVR* since it utilizes the same retargeting approach regardless of video content. *SCVR* alternates between a warping and cropping technique depending on the presence of multiple salient objects in a frame, which can cause inconsistencies in the frame composition if the global optimization is not performed for the entire video. In contrast, the optimization performed in *Parallelized SeamCrop* is easier to segment as it only determines the cropping window path. Preserving the frame composition between segments is much simpler, as the final position of the cropping window and its seams in the previous segment can be used to adjust the path of the next segment, since they are guaranteed to use the same retargeting approach.

Memory consumption Without alterations to the algorithm, *SCVR* is more memory efficient than *Parallelized SeamCrop* as the memory use is constant regardless of video length [1]. The space required depends on the quad grid size, and the memory utilization for all quad sizes are manageable. The memory usage of *Parallelized SeamCrop* scales linearly along with the video length due to the need to store each individual frame until the global optimization is complete, which rapidly becomes an issue for longer video sequences. However, if the optimiza-

Algorithm	Performance	Alterability	Memory Consumption
SCVR	✓	(✓)	✓
Parallelized SeamCrop	✓	✓	(✗)

Table 3.1: Comparison of the retargeting algorithms.

tion is performed on segments as explained in the previous paragraph, the memory utilization is reduced to the space required to store a segments length of frames. Depending on the length of a segment and the size of each individual frame, this alteration makes *Parallelized SeamCrop* viable from a memory usage perspective.

Table 3.1 displays the viability of both algorithms with respect to the properties we discuss in this section. The most important aspect to consider when choosing which algorithm to select is whether or not it can be customized for use for a video streaming from a remote server. If the algorithm can not be altered to provide online retargeting, all other properties are irrelevant. While both approaches can be modified for this purpose, *SCVR* is much more difficult to customize than *Parallelized SeamCrop*. Due to this, guaranteeing output consistency between segments with *SCVR* is problematic.

As *Parallelized SeamCrop* exhibits satisfactory performance and its memory usage concern can be addressed by the algorithm alteration, we choose to utilize this retargeting algorithm for our plugin.

3.3 Detailed Design

In the preceding section, we chose *Parallelized SeamCrop* as our retargeting algorithm of choice. This section presents the design details of our GStreamer plugin, how we integrate the algorithm with the GStreamer framework as well as the interaction between the two. The following subsections focus on the plugin's relation to other GStreamer elements, the inner workings of each individual component and ultimately the flow of the plugin as a cohesive unit. Since most of the complexity is contained within the plugin itself, the majority of these sections are concerned with the internal structure.

3.3.1 SeamCrop Plugin

As described in Section 2.3.1, extensions to the GStreamer framework are implemented as elements that can be inserted into the pipeline. These elements, either multiple or single, are combined together into plugins which can then be loaded by the GStreamer framework.

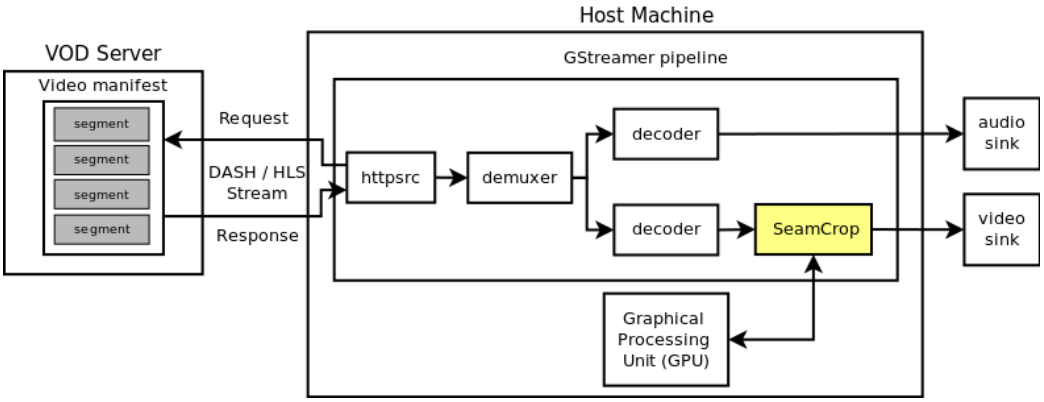


Figure 3.1: Architecture of our system.

The output of our work is a GStreamer plugin containing a single element, **GstSeamCrop**. This plugin is a loadable block of code, shipped as a dynamically linked library which can be inserted into a pipeline at runtime. *GstSeamCrop* is a filter element, neither sink nor source, which consumes video data sent from upstream and produces an adapted output stream.

3.3.2 Architecture Overview

In order to retarget video content using our plugin, the host machine is required to have a CUDA capable GPU with a compute capability higher than 2.0 as well as the GStreamer framework installed. A CUDA capable GPU is required to utilize the parallelization property of the retargeting algorithm, and the plugin will consequently not work without it. Apart from this, our plugin only depends upon the GStreamer framework and its libraries.

Figure 3.1 illustrates the usage of our plugin. In this figure, our plugin is used as part of a GStreamer pipeline which utilizes either DASH or HLS to stream content from a video on demand server. The httpsrc element requests and receives combined video/audio segments from the server via either the MPD or the m3u manifest depending on the streaming technique. The demuxer separates the incoming segments into individual streams. The decoder element decodes the video stream into individual frames which are passed on to our plugin. Our plugin sequentially retargets the video frames via the GPU and passes them onwards to the video sink for presentation.

3.3.3 Supporting GStreamer Elements

As the plugin should be able to retarget any type of video stream, it has to be format independent. The retargeting algorithm itself performs the video adaptation on decoded and uncompressed video frames. Unless format conversions are performed at either end of our element, raw video frames are all that it can accept and produce. Only accepting raw frames is not particularly desirable, and there are a couple of approaches to deal with this issue.

One way is to implement decoding and encoding within the element. This makes the plugin more comprehensive and feature filled, but due to the vast array of available formats, it is not feasible to provide this functionality for all format possibilities within the confines of our project. The sheer amount of time required for an implementation places it outside the scope of this thesis.

A second approach is to implement support for only a few encoded formats. The workload in this approach is less comprehensive and easier to manage, but the resulting functionality is too limited to be of considerable value. The plugin is no longer format independent and is only usable when these formats can be guaranteed. We consider this approach to be too inhibiting as it specializes the functionality as opposed to generalizing it.

The last approach is to utilize decoding and encoding elements from the GStreamer framework. As outlined in Section 2.3, GStreamer is a comprehensive and powerful framework which provides a plethora of elements that we can use to provide this functionality. Plugins such as *decodebin* and *uridecodebin* can be used to automatically choose suitable demuxer and decoder elements for an incoming video stream out of the plugins available in the GStreamer installation. Similarly, plugins for muxing and encoding are also great supply, providing an application programmer with a substantial amount of flexibility when it comes to what format to produce. By surrounding our element with both a decoder and an encoder element in the pipeline, any format concerns are effectively eliminated.

The benefit of this approach is that incoming format is no longer an issue, as the frame is decoded upstream to a format that is readable by our plugin prior to its arrival. Additionally, the format to produce can be chosen at will by the programmer through the downstream encoding element, and none of this functionality needs to be implemented in our element. While this approach necessitates the use of both a decoder and an encoder in the pipeline, it is both the easiest to utilize and provides the most functionality. As the benefits outweigh the detriments, we claim that limiting the element to accept and produce raw frames is reasonable.

3.3.4 Caps Negotiation

In Section 2.3 we present and outline the different capability negotiation techniques that are available in GStreamer. We need to determine and decide upon which negotiation technique that best suits our needs. Considering that our plugin is limited to receive and produce raw video frames, certain restrictions are imposed on which negotiation technique we can utilize.

The following paragraphs compare the different negotiation techniques and how they align with our plugin.

Dynamic Negotiation is the most powerful and flexible negotiation type. Since our element is not able to produce an array of formats or to change the format between its sink and source pads, we have no need for the complexity dynamic negotiation provides.

Transform Negotiation is used to either impose a fixed format transformation between the input and output caps, depending on the input format, or simply pass data through the plugin in *passthrough* mode. Again, as we do not alter the capabilities, we do not need the fixed transformation, and since our plugin is not able to pass on arbitrary streams, passthrough is not feasible. Consequently, neither of these use cases are a good fit.

Fixed Negotiation is the only negotiation technique that suits our plugin. Our element only accepts and produces one format, which means it can not alter its capabilities to either suit a downstream element or an upstream element. As a result, renegotiation with our element is not possible; the other elements in the pipeline will have to accommodate for the format it requires.

3.3.5 Internal Plugin Design

In the preceding sections, we presented how the plugin relates to and communicates with other plugins in the GStreamer framework. This and the following sections present the details of the internal structure, where there are multiple components that cooperate. The structure of the plugin is separated into three main components:

Plugin component The part of the plugin that is exclusively concerned with the GStreamer framework.

Algorithm wrapper Code that acts as an intermediary between the plugin component and the retargeting module.

Retargeting module Code that performs the resizing of the received video frames, using the *Parallelized SeamCrop* algorithm.

The division of the plugin into these components is based on the *separation of concerns* design principle, where each component is designed to address separate concerns [27].

The plugin component interacts and negotiates with other GStreamer elements present in the pipeline. It passes messages and pointers to buffers sent from upstream to other elements downstream as well as the wrapper where necessary. The wrapper relays the content contained in each buffer to the retargeting module in a suitable format as well as relevant messages like end of stream and flushing events. Additionally, it passes the data structures and video stream properties necessary to initialize the retargeting module environment. These properties are the height, width and frame rate of the video to be processed. The retargeting module processes incoming frames to produce outgoing retargeted frames and is essentially controlled by the plugin component through the wrapper.

The plugin component needs to have full control of the retargeting module to adapt and respond to changes occurring in the pipeline. In contrast, the retargeting module only requires a buffer space to store frames, otherwise solely reacting to directives from the plugin component. Similarly, the wrapper does not need to know the inner workings of either the plugin component or retargeting module, only requiring access to the module itself. Consequently, we isolate the as much of the internal functionality of these components as possible. Our principal reasoning behind this choice is that we wish to maintain a plugin structure that is modular. In addition to aligning with the overall design paradigm of the GStreamer framework, this approach enables future development of similar plugins to make use of the plugin component by substituting the wrapper and the retargeting module with minimal refactoring required.

3.3.6 Component Communication

To facilitate the transfer of both buffers and messages in this modular architecture, we outline a set of functions for communication between the components. The main operations we require functions for are initialization, frame transfer and termination. These operations are handled by the functions described in this section, summarized in Tables 3.2 and 3.3.

initialize

Function used to initialize the environments of the wrapper and the retargeting module. It is invoked by the plugin component once the element capabilities have been negotiated and the width, height and frame rate of the incoming stream has been determined. The wrapper and retargeting module are subsequently configured for these properties, allocating space for necessary structures.

pass_buffers

Passes two GStreamer buffer pointers from the plugin component to the wrapper. This function is required to transfer incoming frames from the plugin component to the wrapper. One of the buffers points to an input frame of the negotiated width and height while the other points to an allocated space where the processed frame should be stored. The input frame is passed on from the wrapper to the add_frame function of the retargeting module and the output pointer is stored until the finished frame can be inserted into it.

signal_end

Notifies the wrapper that an end of stream signal has been received from the pipeline and that the pipeline is shutting down. The wrapper invokes the corresponding signal_end function of the retargeting module to tear down the retargeting environment. The retargeting module finishes retargeting any pending video frames before freeing the allocated space and returning, at which point the wrapper tears down its own environment.

flush

Notifies the wrapper that a flush signal has been received from the pipeline. Similar to signal_end, this function invokes the corresponding retargeting module function flush. However, instead of retargeting the remaining frames, the retargeting module immediately tears down its environment. This function is necessary to quickly react to changes in the pipeline that require a rapid response, such as seeking in the video.

run

Invoked from the wrapper to initiate the retargeting process upon reception of the first frame. A separate thread is dispatched to execute this function, as an additional thread is required to manage the retargeting module. This function is only called once. The reason we require a separate thread to manage the retargeting module is explained in Section 3.3.8.

add_frame

Invoked by the wrapper to pass an incoming frame to the retargeting module, inserting it into the internal buffer of the retargeting module. The frame is retargeted once all previously received frames have been processed. This function is necessary to transfer incoming frames from the wrapper to the retargeting module.

Plugin ⇒ Wrapper	
Function	**Description**
initialize	Initializes the wrapper and retargeting module environments.
pass_buffers	Hands over an input and output buffer to the wrapper.
signal_end	Signals the end of the stream and tears down the wrapper environment.
flush	Flushes session specific buffers and information from the wrapper environment.

Table 3.2: Functions for communication between the plugin component and the wrapper.

Wrapper ⇒ Retargeting module	
Function	**Description**
run	Starts the retargeting module.
add_frame	Adds an image to the retargeting module's internal buffer.
signal_end	Signals the end of the stream and tears down the retargeting module environment.
flush	Flushes session specific buffers and structures in the retargeting module environment.

Table 3.3: Functions for communication between the wrapper and the retargeting module.

3.3.7 Plugin Component

The GStreamer related portion of our plugin needs to facilitate use of the retargeting algorithm, avoid superfluous memory copying as well as provide configurable adaptation. As performing memory copy operations are computationally expensive, reducing the amount of copies is beneficial for the overall efficiency of the plugin. The adaptation required in a particular scenario depends on the desired aspect ratio of the retargeted frames. To allow the user to control the degree of retargeting, this must be configurable in the element. These requirements shape the design of this component as a whole, and the manner in which each issue is addressed is detailed in this section, along with additional operations that need to be performed on behalf of the other components.

Properties

In Section 2.3.4, we described GStreamer properties and how they are used by elements. Given that the main goal of this plugin is to provide customizable adaptation for both video on demand and live streams, it is crucial to be able to adapt for different purposes. As such, we require several customizable properties in order to alter the behavior of the plugin to achieve the required adaptation, such as the degree of width reduction.

The following paragraphs name and briefly detail each property we employ in our plugin.

Retargeting factor is the degree to which the width of the video is to be reduced. It is represented as a number between 0 and 1, where 1 subtracted with this factor gives the reduction in width. For example, a retargeting factor of 0.60 gives us a $1 - 0.60 = 0.40$ reduction in width, which can be expressed as a width reduction of 40%. With a width of 640 pixels, the width is reduced to $640 * 0.6 = 384$.

Frame window size indicates how many frames the retargeting module should process at a time, represented as an integer.

Extend window factor is by how much the cropping window should be extended in the search for seams during the second pass of the retargeting algorithm. As we introduce in Section 2.2.2, the cropping window is the target frame size the that the retargeting produces. This factor extends the cropping window size during seam carving to include edges of salient objects that would

otherwise be cut from the final frame. It is represented as a number between 0 and 1 and has the corresponding semantics as the retargeting factor.

These properties are used to configure the retargeting module that relies on information not included in either the pipeline or the stream. While video features such as width and height can be obtained from the pipeline stream, these characteristics are plugin specific properties that need to be explicitly stated. They are set when the plugin is initialized, either to a default value or a value provided on the command line, but can also be set by an application programmer with the line of code shown in Listing 3.1. These property values are passed to the wrapper when its environment is initialized.

Listing 3.1: Setting a property on the seamcrop element.

```
1 g_object_set(gst_seamcrop, ''retargeting-factor'', value, NULL);
```

Choosing the Video Format

As outlined in Section 3.3.3, we limit our plugin to receive and produce raw video frames. There are many different raw formats, where most of them are derived from the pixel formats/color spaces *YUV* and *RGB*. Which color space we utilize is not important; they are both viable and generally indistinguishable from one another[28]. Based on this and that an in depth comparison of the different formats within these color spaces is outside the scope of this thesis, we choose a format that is practical for our purposes.

By utilizing a decoding and demuxing element such as *decodebin*, we are able to receive most of the different formats that are available. The retargeting module makes use of a 24-bit *RGB* pixel format (3 bytes per pixel) internally when representing images, which intuitively makes this format a suitable choice for our element.

However, during preliminary testing with this format, we were not able to make it function properly. Caps negotiation would repeatedly fail regardless of the *RGB* format requested, even with a *videoconvert* element placed between our element and the *decodebin* element. Despite close inspection of the debug logs, we were unable to identify the exact reason, and hence not able to find a solution. Consequently, we chose to utilize a different format.

One of the more ubiquitous raw video formats utilized both within and outside of GStreamer is the *YUV420P/I420* format. While it is not the exact format that the retargeting module requires, it has the advantage of being considerably smaller in size compared to the raw *RGB* counterparts. As such, we have chosen to use this format as the one we request and produce, performing the necessary conversion between *YUV420P* and *RGB24* within our element at both ends. This conversion is described in Section 3.3.8.

Parsing Stream Properties

The specific characteristics of a video stream are not available until the pipeline upstream has been negotiated. Before the negotiation is finished, properties such as width, height and frame rate are specified as ranges that the element will accept. When upstream negotiation is finalized, these properties are set and manually extracted from the caps set in the final stage of the negotiation. The dimensions of the output video we produce are then computed by multiplying the width we extracted with the retargeting factor set during plugin initialization. Our output capabilities are then updated to inform the next element of the video size it should expect and downstream negotiation continues.

Since both the wrapper and the retargeting module require these properties, their environments are not initialized until the entire pipeline is negotiated. This is a conscious choice to postpone the reservation of resources for the retargeting module until the pipeline is ready to transfer data, in the event that downstream negotiation fails. As a result, their initialization takes place when the plugin receives a GST_EVENT_SEGMENT event on its sinkpad, signalling the arrival of a data segment.

Allocating Buffers

Allocating new buffers for each single data segment to be sent is both cumbersome and expensive. Since the size of the frame contained in each buffer is known after negotiation, it is beneficial to reuse the allocated buffer space to eliminate the overhead associated with continually allocating new ones. To limit the allocation overhead incurred by the plugin and in the pipeline overall, a buffer pool is allocated, from which buffers are requested when needed. This pool can be local to a single element or shared across several elements in the pipeline, where sharing a pool has the additional advantage of zero-copy memory transfer between the elements involved. Clearly, sharing a pool is desirable.

Before a shareable pool can be reserved, the element needs to negotiate the properties of the pool with the next element downstream. This is separate from the format negotiation, taking place after the format has been agreed upon. Pool allocation negotiation is always initiated upstream; an element negotiates with the neighboring downstream element, which in turn does the same with the next element. Figure 3.2 visualizes the pool negotiation process between two elements.

Negotiation is initiated by transmitting an ALLOCATION query from the source pad, where the initiator decides whether it provides or requests a pool. Depending on the response from the peer element, we either configure the properties of the proposed pool or allocate a new pool with the properties that were suggested. If the peer element is not able to share a pool, we reserve a pool locally.

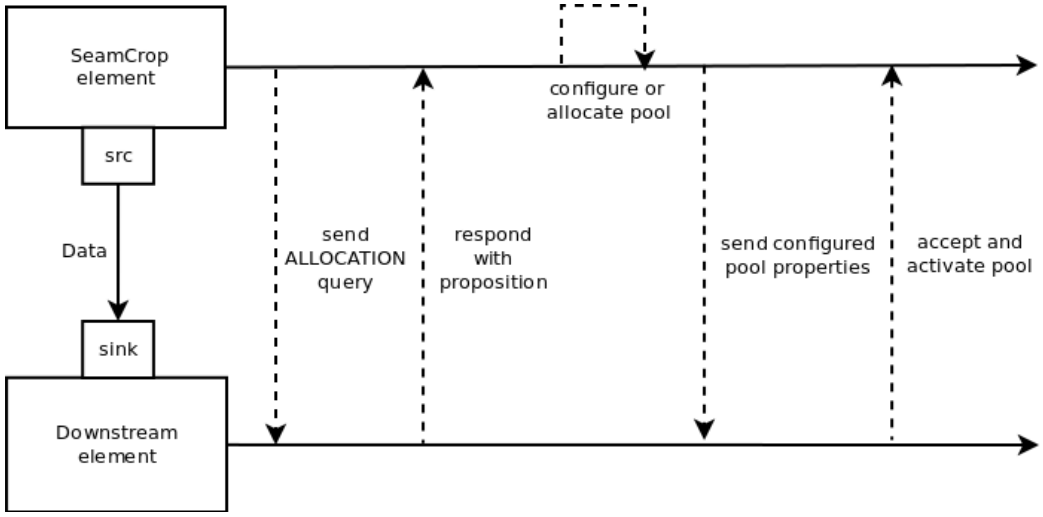


Figure 3.2: Negotiation of a buffer pool.

While not optimal, it saves considerable overhead compared to allocating buffers individually over a prolonged streaming session. Once the pool is allocated by either element, it is activated and ready to be used.

Processing Frames

When all negotiations are finished, a SEGMENT event is transmitted downstream from the source element in the pipeline to signal that data transmission has begun. Once this event is received, the wrapper and retargeting module are initialized. Upon reception of a pointer to a buffer, the element claims ownership of the buffer to ensure that the data within can be consumed. Each incoming buffer pointer contains a frame to process. This essentially means that the preceding element removes the reference count for that particular buffer, whether a pool was negotiated or not. The buffer contents are copied when it is passed to the retargeting module, so it is put back into the pool once the retargeting module begins processing the data contained within.

After the relevant bookkeeping such as timestamp verification is done, the buffer pointer needs to be passed to the wrapper. Due to the discrepancy between the size of the space allocated for the incoming frame and the one to produce, the input buffer space cannot be repurposed to contain the retargeted frame. Ideally, separate buffer pools are negotiated with both upstream and downstream elements. This omits the need to allocate new buffer space for the outgoing frame as well as freeing the space associated with the received buffer pointer. Whether these pools are negotiated or not, a pointer to an available buffer space from the negotiated or internally allocated pool is requested. Both the input and output buffer pointers

are subsequently passed to the wrapper.

In a typical GStreamer plugin, the thread which receives a buffer pointer on the sink pad is also responsible for transmitting the output buffer pointer. It consumes the input buffer data, produces the output and transmits it in one sequence. This is particularly evident in the design of all the current GStreamer base classes [29]. Due to the design of our chosen retargeting algorithm, we are not able to produce an output frame for each received frame at runtime. The underlying reasons for this are detailed in Section 3.3.9, but in short it requires multiple frames to perform the intended retargeting.

Waiting for the retargeting module to produce the output frame with this thread effectively blocks the thread executing the code of the element. This frame will never be finished unless new frames are continually passed to the module, since the retargeting module is not able to iteratively produce output for each incoming frame. This results in a circular dependency where new input depends on finished output, which itself depends on new input and so forth; an instance of the classical producer consumer problem. There are a few possible solutions to this, of variable feasibility:

1. Redesign the algorithm to linearly produce output frames for each input frame.

2. Transfer frame to retargeting module, yield thread and transmit the output frame when it is produced.

The first option is a considerable undertaking which requires major alterations to the retargeting algorithm. In addition to being outside the scope of this thesis, the efficiency of the algorithm can no longer be guaranteed should we depart significantly from the design presented in the original paper. As such, this solution is too comprehensive in nature to be feasible.

Conversely, the second solution is much more manageable. The benefits of this solution is that it allows us to preserve the core algorithm design while simultaneously incorporating it into a GStreamer plugin. This ensures that the efficiency of the algorithm bears a certain resemblance to what was achieved in [2].

However, both approaches introduce a baseline of latency to the pipeline. This latency is unavoidable as it stems from the entire process of receiving a frame and producing an output frame, but the approaches differ in how this latency is distributed. The first solution incurs latency on a frame by frame basis, while latency in the second approach is the cumulative time required to process a set of frames. Essentially, the time elapsed between the transmission of a frame from the source element to its presentation in the sink element is shorter in the linear solution.

While the first is the faster option, inter-frame jitter is more likely to occur, which is exacerbated further should the performance be too poor. The second solution incurs a higher initial latency with less likelihood of inter-frame jitter due to internal buffering. As users are more tolerant of initial latency as opposed to recurring buffering when streaming a video [10], the viability of the second solution is substantiated further. Based on this reasoning, we employ this approach and defer transmission until the threads internal to the retargeting module have processed and finished the frame.

When the buffers have been successfully given to the wrapper, the managing thread returns and proceeds as if the buffer pointer was transmitted, subsequently taking care of other incoming frames. This is done by returning a $\mathrm{GST\_FLOW\_OK}$[30] value internally in the element once the frame within the buffer is extracted. This allows the plugin component to continually supply new input frames while the retargeting module independently processes them as soon as they are submitted. The result is an asynchronous data flow where the output of the plugin is bounded by the speed at which the retargeting module produces the output.

To solve this asynchronicity, we maintain a shared asynchronous queue in the plugin component. This queue acts as a container for finished frames, where each entry is a buffer pointer. The wrapper is informed of the address to this queue on initialization, through which the retargeting module will be able to continually insert finished buffers. However, since the receiving thread that would originally transmit the output buffer pointer has long since terminated at this time, we need a different thread to transmit the output.

Dispatching Thread

Our solution is to explicitly dispatch a thread whose sole purpose is to transmit buffer pointers from the asynchronous queue, sending them downstream as soon as they become available. This thread is created at the same time that the wrapper and retargeting module environments are initialized. It continually transmits buffer pointers while they are available, blocking until new ones arrive while the queue is empty. When a buffer pointer arrives, it is removed from the queue and transmitted immediately. As this thread is blocked while the queue is empty, the performance impact of managing this additional thread is minimal.

3.3.8 Algorithm Wrapper

In order to make use of the algorithm in our plugin, we utilize a wrapper that the GStreamer component and the retargeting module communicate and exchange frames through. One of the primary reasons this component is necessary is that

the GStreamer component and the retargeting module are written in different languages, i.e., they are not able to communicate through usual means. Additionally, the wrapper needs to translate the incoming raw video frames into the internal types the retargeting module requires. This wrapper is as such a fairly simple component which acts as an intermediary between the plugin component and the retargeting module.

To encapsulate the internal functionality of the wrapper from the plugin component, only a handful of functions are exposed, as presented in Section 3.3.6 and Table 3.2. Most of these functions directly lead to their counterparts for the retargeting module, which are listed in Table 3.3. However, the threads in the retargeting module must have access to the queue where they insert finished frames without exposing it to the internal functions of the wrapper. To this end, an additional wrapper component is introduced.

This additional component is analogous to the reverse operation of the original wrapper component, solely used to reconvert frames and insert them in the reserved output buffers. This component has only one publicly available function, $\mathrm{putImage}$, which performs this operation. By separating the wrapper operations in this manner, the retargeting module does not have access to the functions used to manage it.

Adding Frames

When the input and output buffer pointers arrive from the plugin component, the output buffer pointer is stored for later use and the raw video frame is extracted from the input buffer space. The extracted video frame is converted from YUV420P to RGB24 format using the FFmpeg framework [31]. This format is required since the retargeting module uses the $\mathrm{IplImage}$ structure from the OpenCV framework to represent an image, which natively uses RGB [32]. After this conversion, each color channel for each pixel in the image is sequentially copied into the $\mathrm{IplImage}$ structure, which is added to the internal buffer of the retargeting module through the $\mathrm{add\_frame}$ function once it is finished. This approach incurs a small performance penalty, but we have not found a solution to circumvent copying the data in this step. At this time, the input buffer is no longer needed and is dereferenced to return it to the buffer pool. Not to be confused with the input/output buffers, the internal buffer in the retargeting module is of fixed length, capable of storing a *frame window size* amount of frames. This internal buffer is explained in Section 3.3.9.

Once the frame has been passed to the retargeting module, the module itself is started. As the module must run independently of the rest of the plugin to achieve maximum efficiency, we dispatch a new thread which manages the retargeting portion of the plugin. The newly created thread calls the run function of the

retargeting module, starting the retargeting process. This is only done once after the first frame has been passed to the module.

If the internal buffer of the retargeting module is full, the thread that wants to add a frame continually attempts this operation until space is made available. The internal buffer is full when the current frame number is equal to the value indicated by the *frame-window-size* property set at plugin initialization. We choose to keep this thread waiting to prevent the plugin component from accepting too many buffers from the preceding element, as this would most likely result in buffer overflow since the input production is generally faster than the output. Busy waiting is utilized as opposed to blocking to prioritize that the retargeting module always has frames available. This behavior and the significance of the *frame window size* property is further explained in Section 3.3.9.

Writing Frames

When a thread in the retargeting module has produced a frame, it enters the additional wrapper component to add it to the asynchronous queue. The oldest stored output buffer pointer is extracted from the queue and in reverse fashion of adding a frame, the retargeted RGB24 frame is converted into a YUV420P frame. This frame is subsequently written into the memory referenced by the output buffer pointer which is then pushed onto the asynchronous queue for immediate transmission by the dispatching thread.

3.3.9 Retargeting Module

In Section 2.2, we present the *Parallelized SeamCrop* algorithm and how it functions. In this section, we detail how this algorithm is incorporated into our plugin and explain a modification we make to the algorithm that necessary to provide online retargeting. Since this module closely follows the flow of the original algorithm, we briefly summarize it.

The *Parallelized SeamCrop* algorithm is split into two passes with one intermediary step between them. The first pass performs an energy calculation on each individual frame of the video, mapping the energy of each individual pixel as well as calculating the total energy of each possible cropping window position. These calculations are then used in the intermediary step to find the optimal path for the cropping window throughout the video. Once this has been established, the second pass sequentially crops each frame, identifying and removing a set amount of seams from them to obtain retargeted frames of the desired size. This flow is illustrated in Figure 3.3 for a video of 500 frames.

As detailed in the previous section, the module receives input from the wrapper and is initiated after the arrival of the first frame. For each pass, a predefined

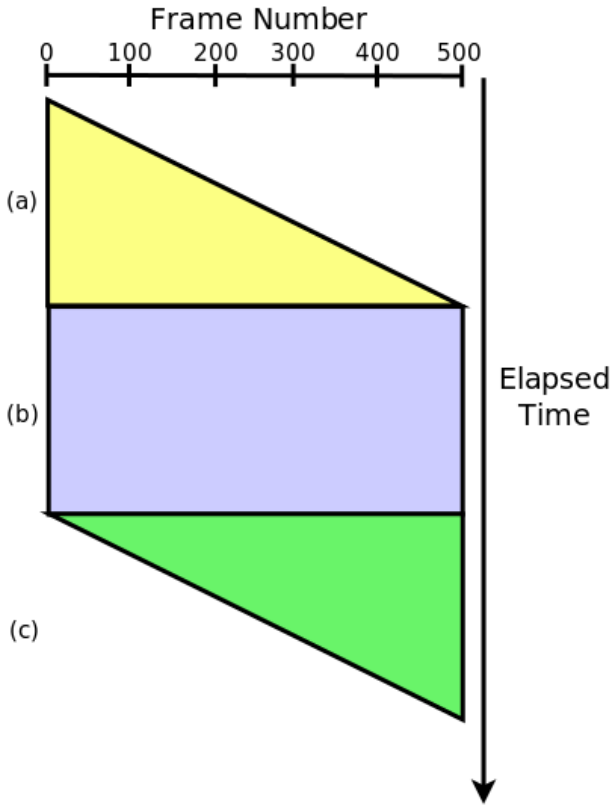


Figure 3.3: Flow of the original algorithm. (a) Energy and cropping window path calculation (b) seam carving (c) presentation.

number of worker threads are dispatched to process the individual frames, uploading them to the GPU to perform the computations in parallel. Between the first and second pass, these threads are terminated and the cropping window path computation is performed on the CPU, as this calculation is not parallelizable, before dispatching the threads again for the second pass. A more detailed description of the threads, their operation and synchronization is included in Section 3.3.10. The retargeting performed on each frame is determined by the *retargeting factor* and the *extend window factor* properties described in Section 3.3.7. Finished frames are pushed onto the asynchronous output queue provided on initialization.

In the original design of this algorithm, each pass is performed only once and each pass is performed on the entirety of the video. Consequently, the algorithm requires the entire video to be available on either disk or in memory throughout its execution. For concurrent on-the-fly retargeting and presenting of a live or video on demand stream, this is not a feasible approach, which ultimately presents a number of issues:

Availability When receiving a video from a remote location, we do not have access to the video in its entirety at the start of execution.

Latency Retargeted frames are not produced until the first pass and the intermediary step are finished, and can as such not be presented until a considerable amount of time has passed.

Resource Usage When retargeting comprehensive videos, memory usage as a result of buffering quickly inflates beyond acceptable boundaries. In the original algorithm, this issue was circumvented by reading the same video file twice, discarding input frames as soon as they had been energy mapped or retargeted. This is possible when downloading, but exceedingly impractical, as the video would either need to be downloaded and stored on disk in its entirety or downloaded twice.

Evidently, some alterations to the algorithm are necessary if it is to be used for concurrent retargeting and presentation of video on demand or live streaming content. If none are made, the presentation cannot begin until the video has been buffered in memory or stored on disk in its entirety and the seconds pass of the algorithm has begun, which would never occur in a live streaming scenario. Our claim is that this solution is not acceptable from either a latency perspective or resource usage standpoint should this algorithm be utilized for simultaneous retargeting and presentation of both video on demand and live streaming content.

To illustrate, imagine you wished to use this system to watch the newest Star Wars film on a mobile device using a video on demand streaming service. For the sake of simplicity, assume that the video has a resolution of 854x480 pixels, a file

size of 700 MB (5600 Mbit) and that it is downloaded at a rate of 24 Mbit/s. In this scenario, it would take $5600/24 = 233.3$ seconds or alternatively 3 minutes and 53.3 seconds to download the file in its entirety. Regardless of retargeting rate, this is the earliest point at which the presentation can begin, which is unacceptable from a startup latency standpoint. Despite the surmountable resource usage in this example, it can quickly become insurmountable, as it always requires at least as much memory or storage space as the size of the file, which in turn also exacerbates the startup latency further.

As for compatibility with live streaming, the video is simply never presented as the algorithm would continuously buffer incoming frames until the live stream ends or storage space runs out.

Based on the identified issues, we present an alteration of the algorithm that attempts to balance each of these concerns.

Algorithm Alteration

The core dilemma in each of the issues mentioned previously is the need to perform each pass on the entire length of the video. This need arises from calculating the optimal cropping window path throughout the video, which requires all frames in the video sequence to be energy mapped. While this requirement is somewhat integral to the overall efficiency of the algorithm, we circumvent this limitation by segmenting the received video into windows as indicated by the discussion in Section 3.2.

As opposed to performing each pass on the total length of the video, we perform the algorithm on predetermined segments of frames that we refer to as *frame windows*. Rather than applying the algorithm on the total amount of frames T for the entire video, we perform it on t frames at a time over T/t consecutive frame windows. In this case, t corresponds to the last property we described in Section 3.3.7, *frame window size*. This behavior is illustrated in Figure 3.4, where a video of 500 frames is sequentially split into windows of 100 frames, performing the algorithm on each frame window individually.

To facilitate this, we store the incoming frames in a circular buffer. This circular buffer is the internal buffer of the retargeting module referred to over the previous sections. The size of this buffer is determined by the *frame window size* property, where the integer it represents corresponds to the amount of slots in the buffer. When this buffer is filled, the algorithm proceeds to compute the cropping window path and carving out seams. Once a frame is finished during the second pass, it is copied into a reserved buffer from the GStreamer buffer pool which passed on to the wrapper for transmission. The slot it occupies is then marked as available and a new frame can be inserted into the buffer. Thus, the buffer is continuously refilled as retargeted frames are produced during the seam carving

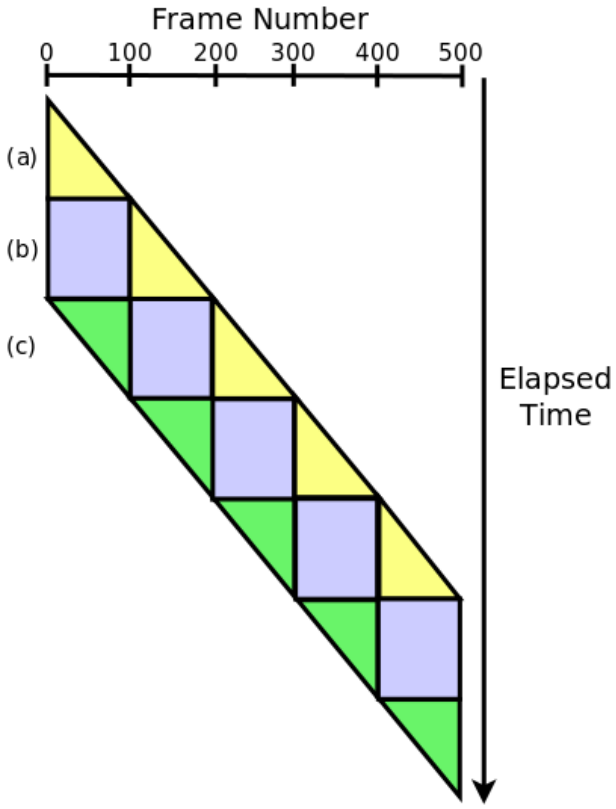


Figure 3.4: Flow of the altered algorithm. (a) Buffering, energy and cropping window path calculation (b) seam carving (c) presentation.

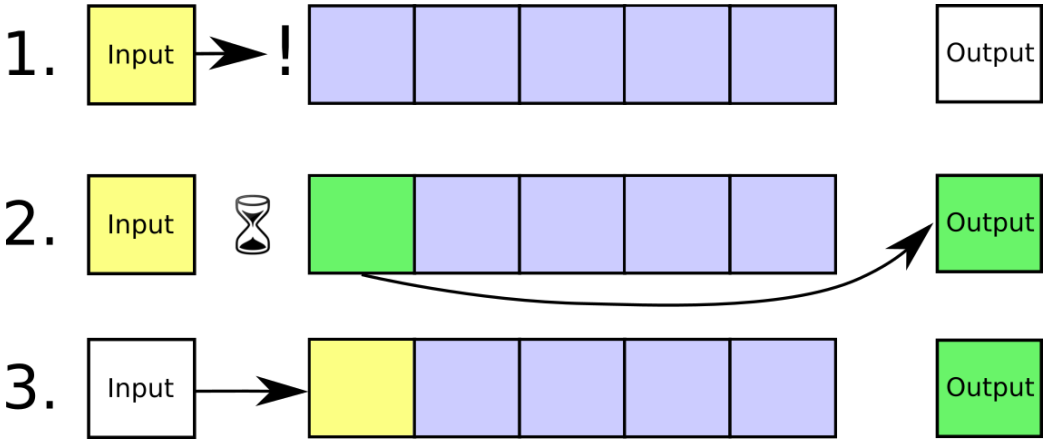


Figure 3.5: Buffering new frames during the second pass of the algorithm. (Yellow) New frame (Blue) Frames being retargeted (Green) Finished frame. (1) Input tries to add new frame, but the space is occupied; (2) frame is finished, new frame waits until space is marked as available; (3) space is now available, new frame is inserted.

portion of the algorithm, as illustrated in Figure 3.5. It is important to note that the new frames that are inserted belong to the next frame window. This eliminates the need for the entire video to be available throughout execution, allowing the algorithm to receive and process frames concurrently as well as outputting retargeted frames at a much earlier stage than in the original approach. The effects of altering the algorithm in this manner is explored in Section 3.3.12.

3.3.10 Threading

As mentioned in the previous sections, we utilize multiple threads to handle and manage the different components of the plugin. Figure 3.6 shows a summarized and simplified overview on how the threads interact as well as in which components of the plugin they operate. As no threads apart from the ones originating in the retargeting module overlap in their workflow, we avoid issues related to thread synchronization. In contrast, the threads inside the retargeting module require considerable synchronization.

Threading in the Retargeting Module

The individual threads in the retargeting module are managed by the thread dispatched from the wrapper upon submission of the first video frame. They concurrently perform the retargeting on separate frames within the confines of the two parallelizable passes of the algorithm. The amount of threads that are used in the

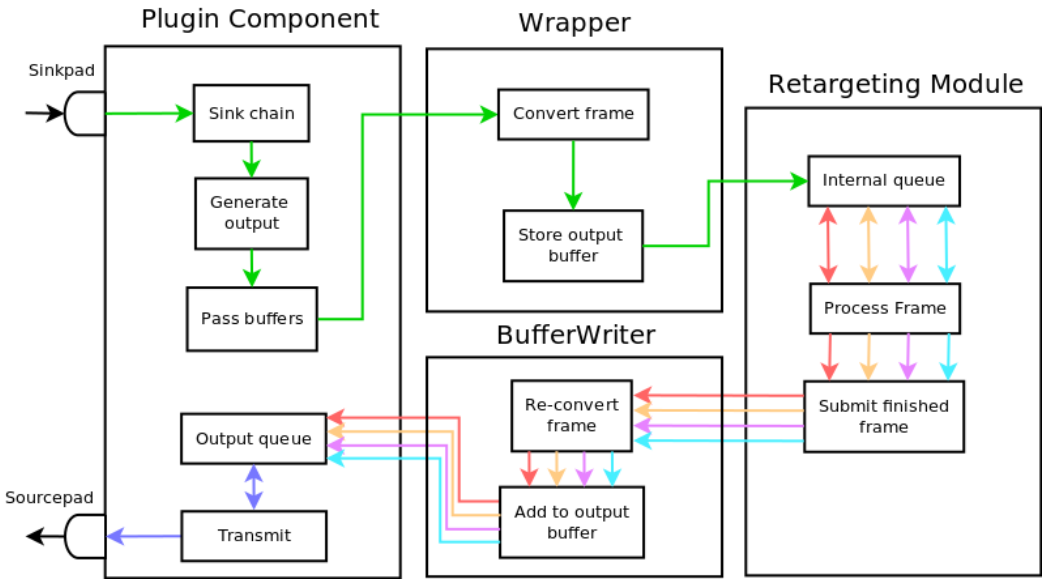


Figure 3.6: Simplified thread flow in the plugin. Each color represents the execution of a separate thread.

passes is configurable, but must be at least 2, as a single thread will wait forever for the next frame, and at most 8. The managing thread determines when they run and performs the cropping window computation as well as the smoothing between the passes. Algorithm 1 presents the pseudocode for the operation of the managing thread.

For both passes, the managing thread dispatches the worker threads and blocks until one of them finishes their operation, as shown in Algorithm 2. The managing thread is notified of a finished pass via a condition variable, flagged by the first thread that exits its internal processing loop. The processing loop of a working thread executes until there are no more frames to process in the circular buffer, i.e, an attempt to acquire a frame from the circular buffer yields a frame number that exceeds *frame window size*. When a worker thread finishes, the last frame of the frame window has begun processing for the current pass, which lets the managing thread start joining the threads to finish the pass. The variable denoting the next frame number to process is reset between passes.

The worker threads require synchronization at multiple points of their execution. The principal blocking points occur when a thread either attempts to acquire a new frame from the circular buffer or in the second pass when it requires seams from a previous frame which have not been found yet. Since the first pass of the algorithm is fully parallelizable, no synchronization apart from acquiring frames and uploading them to the GPU is required. To prevent any race conditions, the thread locks the the circular buffer as well as the counter that indicates the frame

```
1: while True do
2:     run the first pass
3:     calculate the cropping window costs
4:     calculate the maximum energy path
5:     if first_run ≠ 1 then
6:         smooth cropping window transition between frame windows
7:     end if
8:     smooth the cropping window path
9:     run the second pass
10:    if end_of_stream and pending_frames = 0 then
11:        break
12:    end if
13:    first_run = 0
14: end while
```

Algorithm 1: Managing thread pseudocode

```
1: for i ← 0 to number of threads do
2:     DISPATCH(thread[i])
3: end for
4: WAIT(conditional variable)
5: for i ← 0 to number of threads do
6:     JOIN(thread[i])
7: end for
```

Algorithm 2: Dispatching and joining threads pre/post each pass

to process next when retrieving a new frame. This is fairly straightforward. The counter for the current frame is incremented before the lock is released and the shared variables denoting whether a frame is available is updated after the GPU synchronizes its internal stream.

During the search for seams in the second pass, each thread must verify that the seam it requires from the previous frame has been found. The amount of seams currently found for the previous frame is maintained in a shared variable which is only updated when a seam has been identified. Worker threads are not able to proceed to the next seam until the GPU has finished all seam computations in the current CUDA stream, facilitated by the $\mathrm{cudaStreamSynchronize}$ function present in the CUDA API [33]. This ensures that the threads are synchronized for each computed seam, omitting the need for any additional mutual exclusion.

3.3.11 Event Handling

In the preceding sections, we have presented the design of our plugin in great detail. In this section, we describe the different events that affect our element as well how the data flow changes in response to them. Not all events significantly impact the element; only a few directly affect the internal data flow and we describe them in the following paragraphs.

Segment

A $\mathrm{GST\_EVENT\_SEGMENT}$ event is sent from the source element in the pipeline to signal that data is ready to be transmitted. This event always occurs before any data can be sent downstream. When it arrives on the sink pad, the wrapper and retargeting module environments are initialized and the dispatching thread is created.

Buffer Reception

Reception of a buffer pointer is not defined as an explicit event in the GStreamer framework, but always occurs after a $\mathrm{GST\_EVENT\_SEGMENT}$. The data flow associated with this event is visualized in Figure 3.7. When input arrives on the sink pad, the plugin claims ownership of the incoming buffer pointer to ensure that it is consumed in this element. The thread dispatched to handle the sink chain requests an output buffer from the negotiated pool and passes both buffer pointers onwards to the wrapper. Upon reception of the buffers, the wrapper stores the output buffer pointer and proceeds to convert the newly arrived frame from *YUV420P* to *RGB24*. Should the circular buffer of the retargeting module be full,

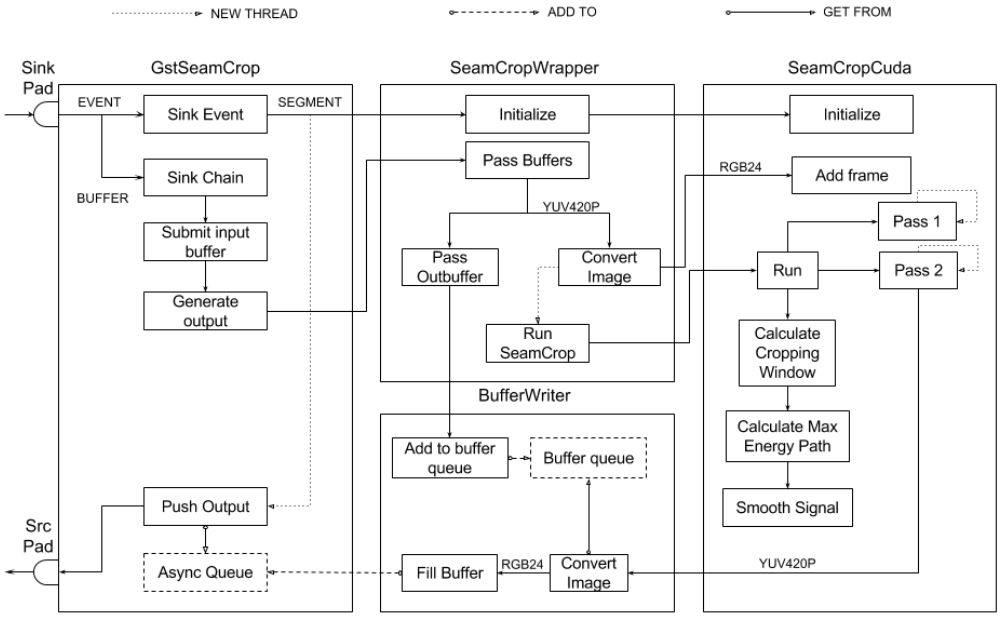


Figure 3.7: Internal data flow in the element for initialization and buffer reception.

it waits until space is available, otherwise it inserts the frame and returns to handle the next incoming frame.

The retargeting module threads continually perform the first pass on incoming frames, preventing further frames from being added once a *frame window size* amount of frames is received, at which point the first pass is finished. Cropping window computations and the smoothing is subsequently performed before the initiating the second pass. During this pass, the external thread that receives new frames from the pipeline replaces the finished frames concurrently alongside the retargeting process, as illustrated in Figure 3.5. When a worker thread has fully processed a frame, it enters the second wrapper component, retrieves the oldest stored output buffer and converts the frame back to its original format, subsequently adding it to the asynchronous output queue. The oldest output buffer is always chosen as the retargeting module produces retargeted frames in sequential order, eliminating the need for any extraneous bookkeeping with regards to order. At this time, the thread responsible for transmitting output buffers wakes up, updates the timestamps of the outgoing buffer and transmits it via the sourcepad before returning to wait for more buffers.

End of Stream

When a GST_EVENT_EOS is received on the sink pad from the pipeline, the element initiates the teardown of the internal components. However, as there might still be outstanding frames that need to be processed, the plugin component must wait until the retargeting module has processed all remaining frames. The plugin component notifies the wrapper that the stream has ended through the signal_end function. The wrapper informs the retargeting module of the last frame to process through the corresponding function of the same name, after which the thread waits for the module to finish. Once it finishes the remaining second pass, the thread managing the module frees all allocated space in the retargeting environment and terminates. The wrapper does the same and the element is torn down.

Flushing

Over the lifetime of the pipeline, flushing might occur via the GST_EVENT_FLUSH event. The GST_EVENT_FLUSH event differs from GST_EVENT_EOS in that the element must adapt to a change in the pipeline as fast as possible. There are many different reasons for flushing, where the typical reasons are renegotiations upstream or downstream as a result of a new incoming format or seeking performed by the client downstream. Regardless, the element must free all the memory associated with the retargeting environment to prepare for a possible new format, as the signal does not differentiate between the reasons for flushing.

In contrast to the GST_EVENT_EOS case, the plugin component does not wait until all pending frames are processed. Since it must adapt as fast as possible, the retargeting module terminates as soon as it receives the signal, discarding all currently processing frames. While this results in lost frames, the catalyst for flushing is usually an event that favors rapid adaptation as opposed to consistent presentation.

3.3.12 Consequences of Algorithm Alteration

In Section 3.3.9, we concluded our description of the retargeting module with a modification to the algorithm necessary to facilitate online retargeting. Altering the algorithm in this manner does not come without certain ramifications. In the following paragraphs, we examine and describe these effects, how they affect both the algorithm and the plugin operation as well as how we deal with the consequences that emerge.

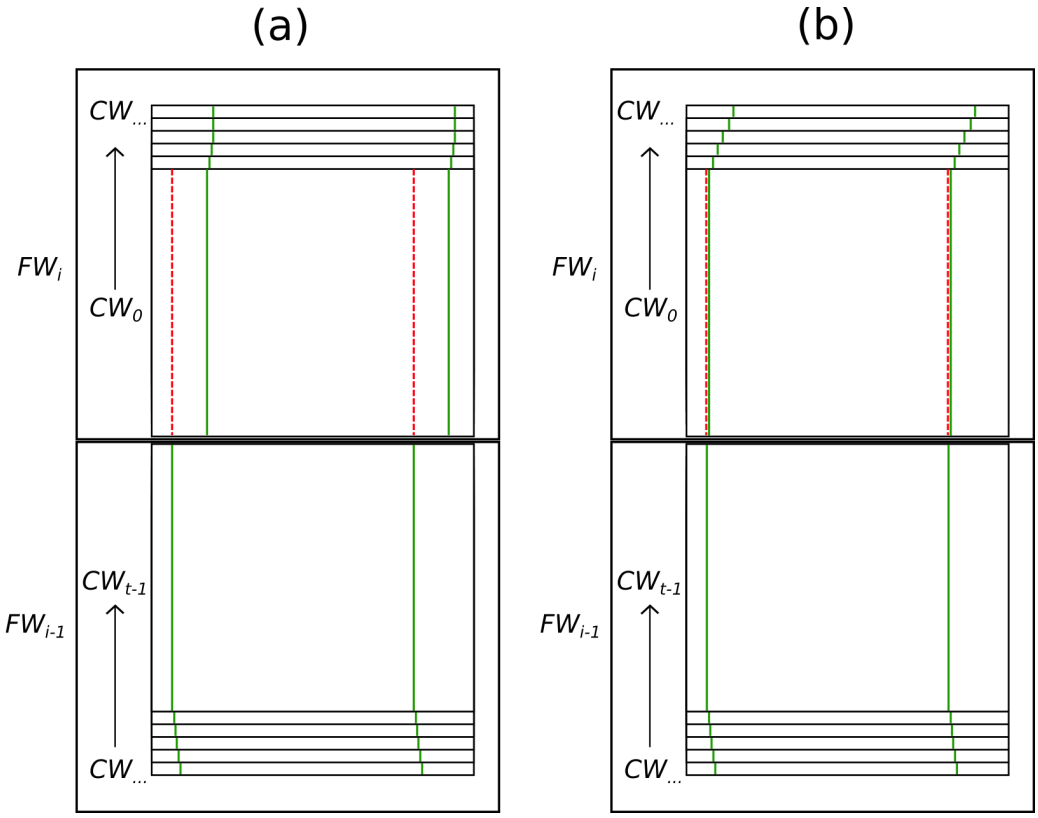


Figure 3.8: Cropping window differences between frame windows. (a) Without transitional smoothing (b) With transitional smoothing.

Cropping Window Discrepancy

When segmenting the entire video into frame windows, we essentially divide the video into many smaller videos. This inevitably affects the produced video as the global optimization of the cropping window is no longer one contiguous path. The path computed for one frame window will not necessarily end in the same position as the first cropping window position computed for the next window, resulting in a dissonant transition between them. This behavior, illustrated in Figure 3.8, occurs since the relative importance value assigned to a pixel fluctuates between frame windows due to either object motion or the introduction of new ones. Each frame window is independent without any inherent knowledge of the preceding frames. Short of redesigning the manner in which the cropping window path is determined, this is an unavoidable consequence of our modification that needs to be addressed.

To alleviate this issue, we introduce an additional smoothing step between the two passes. Consider the example we presented in Figure 3.8. Before smoothing

the computed cropping window path, the position of the last cropping window CW_{t-1} in the previous frame window FW_{i-1} is taken into account. We do this by smoothing the discrepancy between the two windows, gradually transitioning the last cropping window position CW_{t-1} in the previous frame window FW_{i-1} over to the new path over a subset of the initial frames in the next window FW_i. By utilizing a weight which initially prioritizes the position of the previous window, we gradually transition over to the new cropping window path. Essentially, this means that the cropping window path of the new frame window FW_i is a partial compromise between the last cropping window CW_{t-1} and the current path.

The product of this algorithm modification is a more pronounced virtual pan across the video. Depending on the severity of the performed retargeting and the size of a frame window, it is either imperceptible or distinct when compared with the output of the original algorithm. The source material used also bears significance, where the output of sequences with scarce movement in either camera or motives do not differ considerably. Conversely, this also entails that the opposite case results in a more accentuated disparity.

Aside from the dissimilarity between the output of the modified algorithm and the original, the added virtual pan does yield a potential benefit. The increased frequency of importance evaluation and cropping window movement allows it to adapt to new occurrences in the content in a more expedient fashion. Admittedly, this benefit comes at the cost of more sporadic camera movement. Depending on the nature of the video content, this can be either advantageous or detrimental.

Regardless, a comparison of the qualitative results of the original approach and ours is not the prime mover of this thesis; developing a GStreamer plugin capable of performing real-time, online video retargeting for both video on demand and live streaming is. The modification to the algorithm is as such ancillary to the primary goal, and we claim that the presented alteration is a necessity to achieve this with the *Parallelized SeamCrop* algorithm, based on the issues and arguments we presented in Section 3.3.9.

Latency

In the context of the retargeting module, latency refers to the time elapsed from when an input frame is received to when the retargeted counterpart is produced. The latency introduced to the pipeline by the retargeting module can roughly be separated into the separate passes of the algorithm, with one addition:

- buffering incoming frames,
- analyzing frames,
- computing cropping window path,

- retargeting frames.

Assuming the input rate of frames is consistent and as least as fast as the algorithm, the latency incurred by buffering is minimal. For the first frame window, the energy calculation is performed concurrently alongside the buffering of incoming frames. For all following frame windows, buffering and seam carving occurs simultaneously since new frames are added to the circular buffer once an output frame is produced, as illustrated in Figure 3.5. As a result, the buffering latency in the retargeting module is virtually eliminated, provided the processing rate of the algorithm does not exceed the output rate of the producing element, which we claim is widely feasible. The bulk of the latency is as such to be expected from the processing time exhibited by the algorithm itself.

The latency induced by the algorithm is highly dependent on the resolution of the incoming frames as well as the severity of the desired retargeting. In broad terms, a high resolution incurs higher processing time in each pass while a higher retargeting factor will mainly affect the seam carving pass of the algorithm. Their effect on the speed of the individual passes is thoroughly explored in the original article, and as we do not alter the either of the computationally heavy passes, we do not detail them here to any significant degree [2]. Instead, we focus on the latency that performing the algorithm on multiple segments introduces to the presentation.

Since finished frames are continually produced and passed on during the second pass of the algorithm, a video being retargeted at the same rate as the original frame rate can be presented concurrently. Note that the retargeting rate in this context refers to the amount of frames produced per second by the algorithm as a whole, not solely the second pass. To enunciate, a retargeting rate equal to the frame rate of the video entails that the second pass actually produces frames at a higher rate, compensating for the preceding passes. Figure 3.9 illustrates this, where the total retargeting rate is 25 frames per second while the rate of the second pass is slightly higher at 27 frames per second. An eventual receiver will have buffered enough frames at the end of the second pass to not be affected by the gap in frame production introduced by swapping frame windows.

However, despite a congruent video and retargeting rate, no frames are produced when the algorithm is not operating in the second pass. This means that the retargeting module will invariably introduce a guaranteed baseline of latency at the start of execution. Thus, assuming the video is retargeted in real-time, the latency introduced to the pipeline by the retargeting module boils down to the time it takes to buffer, perform the energy calculation and cropping window path computation for the first frame window. Compared with the original algorithm, the initial latency incurred by our approach is much lower.

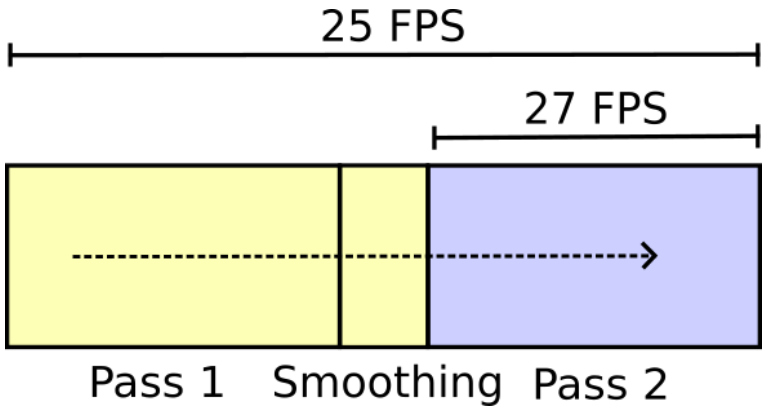


Figure 3.9: Visual representation of retargeting rate for the second pass versus the algorithm as a whole.

Resource Usage

The resource usage required to operate the retargeting module is greatly decreased by the proposed alteration when compared with the original. As storing the entire video is not necessary, only a frame window of space is required to store frames throughout the duration of a stream. When comparing the examples we presented in Figures 3.3 and 3.4, the memory required by our approach is $1/5$ of the original.

Frame Window Size

In the exploration of the effects of the algorithm modification in this section, a direct relationship between the frame window size, latency and resource usage emerges. The effects of this relationship and their interactions with the other properties are explored in Chapter 5.

3.4 Summary

The plugin is separated into 3 components: the plugin component, wrapper component and retargeting module. The plugin component handles all aspects related to the GStreamer framework such as negotiation, data reception, transmission and signals. The wrapper component translates data between the plugin component and the retargeting module. The retargeting module adapts received frames through dynamic image retargeting.

When a frame arrives in the element, the plugin component reserves buffer space for the output and passes both buffer pointers on to the wrapper component. The wrapper component converts the frame contained in the input buffer space

from *YUV420P* to *RGB* and inserts the converted frame in the circular buffer of the retargeting module. The retargeting module performs the first pass of the *Parallelized SeamCrop* algorithm on incoming frames, advancing to the next stage of the algorithm once a *frame window size* amount of frames have been processed. Once the cropping window transition is smoothed, frames are sequentially passed back through the wrapper component for data transmission as soon as they have been retargeted. The wrapper component re-converts the frames to *YUV420P*, inserts each individual frame into a previously reserved output buffer space and places the buffer pointer onto the output queue. The dispatching thread in the plugin component extracts each buffer pointer, verifies the timestamp information and subsequently transmits it downstream as soon as possible.

Chapter 4

Implementation

In this chapter, we describe the implementation of our GStreamer plugin. Section 4.1 outlines which parts of the implementation that are presented in this chapter. Sections 4.2, 4.3 and 4.4 describe the implementations of the plugin component, wrapper and retargeting module.

The CUDA and C++ implementation of the *Parallelized SeamCrop* algorithm was provided by Stephan Kopf, Professor at the Mannheim University. It is the implementation that was used for testing and evaluation in [2], and much of the original source code remains unchanged. In order to utilize this implementation in our plugin however, many alterations to the C++ code were required, such as the modification of the algorithm presented in Section 3.3.9. These alterations are described in Section 4.4 of this chapter. The original source code can be viewed in the Git repository referenced in Appendix A.

Our project has been implemented as a GStreamer plugin in the C language, which internally wraps a retargeting algorithm implemented in C++ and CUDA. This GStreamer plugin contains one element - *seamcrop*, closely following the architecture we presented in Figure 3.7. The plugin has been tested on a Linux platform, Ubuntu 15.04. In order to use the plugin in a GStreamer pipeline, the programmer utilizes a shared library with the seamcrop filter element.

4.1 Implementation Details

Over the following sections, we present the implementation details of the individual components of the plugin. The most important aspects of our design are implemented in C and C++. The presented implementation includes some additional functionality not listed here, which can be viewed from our Git repository referenced in Appendix A.

In order to present the essentials of our plugin in a fastidious but succinct manner, we present the most vital subsets of the four main files we have written. Functions that have no direct relation to the design aspects are omitted. The file *gstseamcrop.c* embodies the source code for the GStreamer element in our plugin, written in C. Files *seamcropwrapper.cpp* and *bufferwriter.cpp* both contain the wrapper functionality, while *seamcropcuda.cpp* contains the retargeting module, all of which are implemented in C++. Apart from *seamcropwrapper.cpp* and *bufferwriter.cpp*, the implementation of each file is described separately.

4.2 Plugin Implementation

In Section 3.3.7, we describe the design of our plugin component operating in the GStreamer framework. This section describes the problems we encountered during implementation and how the most important aspects are implemented.

Initially, our plugin utilized one of the GStreamer Base Classes that are available for plugin development, specifically *GstBaseTransform*. The purpose of base classes is to simplify plugin development by handling the aspects of the GStreamer framework not directly related to the specific functionality of the plugin, such as state changes, buffer allocation, caps negotiation and so forth by making the plugin implementation a subclass of the base class. The subclass can override much of the functionality in the base class, making this a quite flexible option. On paper, the description of *GstBaseTransform* aligns perfectly with our goals for the plugin [34]:

> *" This base class is for filter elements that process data. Elements that are suitable for implementation using GstBaseTransform are ones where the size and caps of the output is known entirely from the input caps and buffer sizes. These include elements that directly transform one buffer into another, modify the contents of a buffer in-place, as well as elements that collate multiple input buffers into one output buffer, or that expand one input buffer into multiple output buffers. "*

However, there is one important issue with this base class. While *GstBaseTransform* allows the programmer to override a subset of its functions, it does not support overriding the transmission of data. More specifically for our case, separating the buffer submission to the wrapper and the transmission of the output is not possible as the function chain that receives and transmits a buffer pointer is hidden from the subclass. Without this option, the deadlock we describe in Section 3.3.7 is inevitable. To remedy the situation we would need to alter the source code of the base class itself. As the base classes are prepackaged in a GStreamer library, altering the source code is a nontrivial task which would require repackaging the

library into a custom library after changing the source code. Changing aspects of a framework is generally not desirable, as it can lead to inconsistencies with future updates of the framework. Consequently, we abandoned this approach in favor of a more general solution.

Since none of the higher level base classes permit this separation, we base our plugin on the most general of the GStreamer element classes, *GstElement*. As a result, the codebase for the plugin component is extensive to include the functionality initially provided by *GstBaseTransform* which is absent from *GstElement*. Most of these functions are not critical to the design and are as a result omitted. Several functions such as sink_event, sink_chain and push_output are also left to Appendix A as they are too comprehensive to be listed in this chapter.

Initialization When initializing our element, we need to perform some additional actions which are usually handled by an underlying base class. This includes manually setting all function pointers for our pads and overriding the property functions as well as the function to clean up our element from the *GstElement*. As our element only receives and consumes one format, we define this format in our static *GstCaps* template, shown in Listing 4.1. Since the capabilities are identical for both pads, only one is presented.

Listing 4.1: The capabilities of our element.

```
1  static GstStaticPadTemplate gst_seamcrop_src_template =
2  GST_STATIC_PAD_TEMPLATE ("src",
3  GST_PAD_SRC,
4  GST_PAD_ALWAYS,
5  GST_STATIC_CAPS (
6  "video/x-raw, "
7  "format = (string) I420, "
8  "framerate = (fraction) [ 0/1, 2147483647/1 ], "
9  "width = (int) [ 1, 2147483647 ], "
10 "height = (int) [ 1, 2147483647 ]")
11 );
```

plugin_init and gst_seamcrop_class_init functions are called to initialize function pointers, metadata, properties and register the plugin with GStreamer as shown in Listing 4.2. We set the pad templates in lines 9–12 and plugin meta data on line 14. The function pointers for altering the properties are initialized in lines 19–21 and the properties are installed with their respective limits in lines 23–36.

Listing 4.2: Initializing function pointers, meta data and properties.

```
1  static void
2  gst_seamcrop_class_init (GstSeamCropClass * klass)
3  {
```

```
4     GObjectClass *gobject_class = G_OBJECT_CLASS (klass);
5     GstElementClass *element_class = GST_ELEMENT_CLASS (klass);
6
7     parent_class = g_type_class_peek_parent (klass);
8
9     gst_element_class_add_pad_template (element_class,
10      gst_static_pad_template_get (&gst_seamcrop_sink_template));
11    gst_element_class_add_pad_template (element_class,
12      gst_static_pad_template_get (&gst_seamcrop_src_template));
13
14    gst_element_class_set_static_metadata (GST_ELEMENT_CLASS(klass
        ),
15      "GStreamer SeamCrop element", "GStreamer SeamCrop Plugin",
16      "Retargets video frames using the SeamCrop algorithm.",
17      "Haakon Wilhelm Ravik <haakonwr@ifi.uio.no>");
18
19    gobject_class->set_property = gst_seamcrop_set_property;
20    gobject_class->get_property = gst_seamcrop_get_property;
21    gobject_class->finalize = gst_seamcrop_finalize;
22
23    g_object_class_install_property(gobject_class, PROP_RETARGET,
24      g_param_spec_float("retargeting-factor", "Retargeting Factor
          ", "Factor for how "
25      "much the video should be reduced in width.",
26      0.10f, 0.99f, 0.75f, G_PARAM_READWRITE));
27
28    g_object_class_install_property(gobject_class, PROP_EXTEND,
29      g_param_spec_float("extend-border", "Extend Border Factor",
          "How much the border "
30      "of the video should be extended with during seam carving.",
31      0.10f, 0.99f, 0.25f, G_PARAM_READWRITE));
32
33    g_object_class_install_property(gobject_class,
        PROP_WINDOW_SIZE,
34      g_param_spec_int("frame-window-size", "Frame Window Size",
35      "How many frames to retarget at a time from an incoming
          stream.",
36      10, 500, 100, G_PARAM_READWRITE));
37  }
38  static gboolean
39  plugin_init (GstPlugin * plugin)
40  {
41    return gst_element_register (plugin, "seamcrop", GST_RANK_NONE
        ,
42    GST_TYPE_SEAMCROP);
43  }
```

`gst_seamcrop_init` initializes a new instance of the seamcrop filter element. As shown in Listing 4.3, we start by initializing the sink pad and setting all

the required functions before adding the pad to the element on line 4. A similar sequence of events is performed for the source pad in lines 12–17. The source pad is not assigned a chain function as it never receives data from a downstream element. Lines 20–22 set the default property values, which can later be altered by the GStreamer framework through an application or on the command line, while lines 24–37 perform general value initializations. Finally, we acquire an output queue in which to store retargeted frames on line 40.

Listing 4.3: Initialization of a new seamcrop instance.

```
1  static void
2  gst_seamcrop_init (GstSeamCrop *seamcrop)
3  {
4    seamcrop->sinkpad = gst_pad_new_from_static_template (&
         gst_seamcrop_sink_template, "sink");
5
6    gst_pad_set_event_function(seamcrop->sinkpad,
         GST_DEBUG_FUNCPTR(gst_seamcrop_sink_event));
7    gst_pad_set_chain_function(seamcrop->sinkpad,
         GST_DEBUG_FUNCPTR(gst_seamcrop_sink_chain));
8    gst_pad_set_activatemode_function(seamcrop->sinkpad,
         GST_DEBUG_FUNCPTR(gst_seamcrop_sink_activate_mode));
9    gst_pad_set_query_function(seamcrop->sinkpad,
         GST_DEBUG_FUNCPTR(gst_seamcrop_query));
10   gst_element_add_pad(GST_ELEMENT(seamcrop), seamcrop->sinkpad);
11
12   seamcrop->srcpad = gst_pad_new_from_static_template (&
         gst_seamcrop_src_template, "src");
13
14   gst_pad_set_event_function(seamcrop->srcpad, GST_DEBUG_FUNCPTR
         (gst_seamcrop_src_event));
15   gst_pad_set_activatemode_function(seamcrop->srcpad,
         GST_DEBUG_FUNCPTR(gst_seamcrop_src_activate_mode));
16   gst_pad_set_query_function(seamcrop->srcpad, GST_DEBUG_FUNCPTR
         (gst_seamcrop_query));
17   gst_element_add_pad(GST_ELEMENT(seamcrop), seamcrop->srcpad);
18
19   /* General value initializations */
20   seamcrop->extend_border_factor = DEFAULT_PROP_EXTEND;
21   seamcrop->retargeting_factor = DEFAULT_PROP_RETARGET;
22   seamcrop->frame_window_size = DEFAULT_PROP_WINDOW_SIZE;
23
24   seamcrop->cache_caps1 = NULL;
25   seamcrop->cache_caps2 = NULL;
26   seamcrop->pad_mode = GST_PAD_MODE_NONE;
27   seamcrop->gap_aware = FALSE;
28
29   seamcrop->processed = 0;
```

```
30    seamcrop->dropped = 0;
31    seamcrop->cur_frame = 0;
32    seamcrop->prev_width = 0;
33    seamcrop->prev_height = 0;
34    seamcrop->input_width = 0;
35    seamcrop->input_height = 0;
36    seamcrop->started = FALSE;
37    seamcrop->reinitialize_module = FALSE;
38
39    /* Queue in which retargeted frames will be placed. */
40    seamcrop->output_queue = g_async_queue_new();
41  }
```

read_video_props extracts the video properties of the stream by parsing the *GstCaps* that was negotiated with the neighboring upstream element, as shown in Listing 4.4. This is done prior to finishing negotiations with the downstream element to compute the width of the frames that our element will produce. The new frame dimensions are compared with the old on line 21 to determine whether the capabilities differ. If they do, the retargeting module must be reinitialized for a new frame resolution. The output width is obtained on line 27 by multiplying the original width with the retargeting factor. Note that we enforce the computed output width to be divisible by two on line 28. This is required by the *GstVideoFrame* structure, which internally determines the size of each buffer based on width, height and format. Frames which do not adhere to this property are not recognized as valid frames by the GStreamer framework.

Listing 4.4: Parsing video stream properties.

```
1  static void
2  read_video_props(GstSeamCrop *seamcrop, GstCaps *caps)
3  {
4    gint denom, framerate;
5    const GstStructure *str;
6
7    seamcrop->prev_width = seamcrop->input_width;
8    seamcrop->prev_height = seamcrop->input_height;
9
10   str = gst_caps_get_structure (caps, 0);
11
12   if
13   (!gst_structure_get_int (str, "width", &seamcrop->input_width)
         ||
14   !gst_structure_get_int (str, "height", &seamcrop->input_height
       ) ||
15   !gst_structure_get_fraction (str, "framerate", &framerate, &
       denom))
16   {
17     GST_LOG("read_video_props: No relevant properties available"
```

```
        );
18      return;
19    }
20
21    if((seamcrop->prev_width != seamcrop->input_width) ||
22      (seamcrop->prev_height != seamcrop->input_height))
23      seamcrop->reinitialize_module = TRUE;
24
25    seamcrop->input_framerate = ((float)framerate / (float)denom);
26
27    seamcrop->output_width = seamcrop->input_width * seamcrop->
          retargeting_factor;
28    if(seamcrop->output_width % 2)
29      seamcrop->output_width -= 1;
30  }
```

gst_seamcrop_do_bufferpool attempts to negotiate a buffer pool with the downstream element, presented in Listing 4.5. As described in Section 3.3.7, we query our downstream peer for a pool, performed on line 11. The query result is passed to gst_seamcrop_decide_allocation, where we determine whether the proposed pool is satisfactory or allocate our own pool should it not be suitable. When either of these are performed, the pool and allocator are stored for use in data transmission via the gst_seamcrop_set_allocation function. As these functions are quite comprehensive they are not included in the listing.

Listing 4.5: Allocating a buffer pool.

```
1  static gboolean
2  gst_seamcrop_do_bufferpool (GstSeamCrop * seamcrop, GstCaps *
       outcaps)
3  {
4    GstQuery *query;
5    gboolean result = TRUE;
6    GstBufferPool *pool = NULL;
7    GstAllocator *allocator;
8    GstAllocationParams params;
9
10   /* find a pool for the negotiated caps */
11   query = gst_query_new_allocation (outcaps, TRUE);
12
13   result = gst_seamcrop_decide_allocation(seamcrop, query);
14
15   if (!result)
16     goto no_decide_allocation;
17
18   /* we got configuration from our peer or the decide_allocation
         method, parse them */
19   if (gst_query_get_n_allocation_params (query) > 0) {
20     gst_query_parse_nth_allocation_param (query, 0, &allocator,
```

```
        &params);
21    } else {
22      allocator = NULL;
23      gst_allocation_params_init (&params);
24    }
25
26    if (gst_query_get_n_allocation_pools (query) > 0)
27      gst_query_parse_nth_allocation_pool (query, 0, &pool, NULL,
          NULL, NULL);
28
29    /* now store */
30    result = gst_seamcrop_set_allocation (seamcrop, pool,
        allocator, &params, query);
31
32    return result;
33
34    no_decide_allocation:
35    {
36      GST_WARNING_OBJECT (seamcrop, "Failed to decide allocation")
          ;
37      gst_query_unref (query);
38
39      return result;
40    }
41  }
```

4.3 Wrapper Implementation

In Section 3.3.8, we describe the operational design of our wrapper for the retargeting module. In this section, we explain how the functionality is implemented.

The wrapper functionality is divided into two separate files to mask the internal operation of the wrapper from the retargeting module. The *seamcropwrapper.cpp* file concerns the communication between the plugin component and the retargeting module, while *bufferwriter.cpp* solely acts as the interface through which the retargeting module re-converts and pushes buffers onto the output queue. Henceforth, they are referred to as the BufferWriter and the SeamCropWrapper, respectively.

SeamCropWrapper

To encapsulate the inner wrapper functionality from the plugin component as described in Section 3.3.5, the C interpretation of *seamcropwrapper.h* is limited to the function calls listed in Table 3.2. In the transition from design to implementation, the names of some of the functions listed in this table have been altered

Design name	Implementation name
initialize	initSeamCrop
pass_buffers	passBuffers
signal_end	signalEndOfStream
flush	flushCurrentInstance

Table 4.1: The updated names of the functions exposed to the plugin component by the wrapper.

Design name	Implementation name
run	run
add_frame	addFrame
signal_end	endOfStreamSignal
flush	stopExecution

Table 4.2: The updated names of the functions exposed to the wrapper by the retargeting module.

to be more verbose. The new names of each function is listed in Table 4.1. The C++ interpretation includes all of these, a class SeamCropWrapper as well as the functions listed in Table 3.3 by including the *seamCropCuda.h* file. Some of these functions also have updated names, which are listed in Table 4.2. A global object of class SeamCropWrapper is maintained. This object contains all variables and objects related to the operation of the wrapper, omitting the need for the plugin component to reference or store these variables directly. While it is possible to directly wrap and reference C++ objects in C, we claim that this approach is unnecessarily cumbersome and would intertwine the different components.

Initialization Initialization of the SeamCropWrapper object is performed through the initSeamCrop function, invoked by the plugin component. The initialization functions are presented in Listing 4.6. Within, we allocate space for three separate picture objects: two AVFrame[35] structures, one for *YUV420P* and the other for *RBG24*, as well as one Image8U structure which has a native color space of *RGB24*. The AVFrame objects are used for the format conversion, copying the converted frame into the Image8U object which is inserted into the retargeting module. Additionally, we initialize a BufferWriter object and the retargeting module object, SeamCropCuda. A pointer to the writer is maintained in the wrapper, passed to the retargeting module through the setWriter function. The relationship between the writer and the wrapper is a unidirectional one, where the SeamCropWrapper has access to the BufferWriter, but not vice versa. This

denies the retargeting module access to the SeamCropWrapper. The image conversion context required by passFrame is initialized on line 48.

Listing 4.6: Initialization of the SeamCropWrapper.

```
1
2  // Global variables.
3  SeamCropWrapper handler;
4  boost::thread seamExec;
5
6  gint initSeamCrop(GAsyncQueue *output_queue, gint width,
7  gint height, double framerate, float retargetingFactor,
8  float extendBorderFactor, gint frameWindowSize,
9  guint64 duration)
10 {
11   handler = SeamCropWrapper(width, height, framerate,
12   retargetingFactor, extendBorderFactor, frameWindowSize,
13   duration, output_queue);
14
15   return handler.getTargetFrameSize();
16 }
17
18 SeamCropWrapper::SeamCropWrapper(gint inWidth, gint inHeight,
19 double inFramerate, float retargetFactor,
20 float inExtendBorderFactor, gint inFrameWindowSize,
21 guint64 inDuration, GAsyncQueue *output_queue)
22 : width(inWidth), height(inHeight), frameRate(inFramerate),
23 retargetingFactor(retargetFactor),
24 extendBorderFactor(inExtendBorderFactor),
25 frameWindowSize(inFrameWindowSize), started(false), currentFrame
       (0), currentMaxFrame(0)
26 {
27   targetWidth     = width * retargetingFactor;
28   targetFrameSize = av_image_get_buffer_size(AV_PIX_FMT_YUV420P,
         targetWidth, height, YUV420P_ALIGNMENT);
29
30   // Allocate picture frames.
31   pFrameYUV = av_frame_alloc();
32   pFrameBGR = av_frame_alloc();
33
34   // Allocate buffer of appropriate size for pFrameBGR.
35   av_image_alloc(pFrameBGR->data,pFrameBGR->linesize, width,
         height, AV_PIX_FMT_RGB24, RGB24_ALIGNMENT);
36
37   // Allocate an image to fill the received frames into.
38   image = boost::shared_ptr<Image8U>(new Image8U(width, height,
         3));
39
```

```
40    seamCropCuda = SeamCrop(width, height, frameWindowSize,
          retargetingFactor, extendBorderFactor);
41    writer = boost::shared_ptr<BufferWriter>(new BufferWriter(
          output_queue, targetWidth, height));
42
43    // Pass writer to seamCropCuda.
44    writer->start();
45    seamCropCuda.setWriter(writer);
46
47    // Set up conversion context.
48    img_convert_ctx = sws_getContext(width, height,
          AV_PIX_FMT_YUV420P, width, height, AV_PIX_FMT_BGR24,
          SWS_BICUBIC | SWS_CPU_CAPS_MMX, NULL, NULL, NULL);
49
50    if(!img_convert_ctx)
51      BOOST_THROW_EXCEPTION(IOException(std::string("Cannot␣
            initialize␣conversion␣context.")));
52  }
```

Operation The main contributor to the overall execution of the SeamCropWrapper is the passFrame function. It is supported by a handful of other functions, i.e., passBuffers, passOutbuf and signalEndOfStream, that handle the organisational aspects of the SeamCropWrapper.

passBuffers and passFrame functions are used in the transfer of video frames from the plugin component to the retargeting module. These functions are presented in Listing 4.7 passBuffers is invoked by the plugin component to transfer buffers to the wrapper component. It initiates the input buffer parsing by calling passFrame and gives the output buffer to the writer through the passOutbuf function.

The passFrame function converts the video frame format and inserts the frame in the internal queue of the retargeting module. To perform the format conversion, we utilize av_image_fill_arrays[36] at line 15 to parse the received frame into the allocated AVFrame and sws_scale[37] at line 20 to perform the format conversion. The converted frame is copied pixel by pixel into the allocated Image8U structure. This image is added to the internal queue by invoking the addFrame function, shown in Listing 4.11. The retargeting module is started on line 46 by dispatching a thread to invoke the startSeamCrop function. This function is minimal, but required as the thread needs a function handle or callable object to be dispatched[38].

Listing 4.7: Frame conversion, buffer and frame passing.

```
1  void passBuffers(GstBuffer *inbuf, GstBuffer *outbuf)
2  {
3    handler.passFrame(inbuf);
```

```
4     handler.passOutbuf(outbuf);
5   }
6
7   void SeamCropWrapper::passFrame(GstBuffer *inbuf)
8   {
9     bool seamReturn;
10    GstMapInfo info;
11    // Extract contents of buffer.
12    gst_buffer_map(inbuf, &info, GST_MAP_READ);
13
14    // Fill picture frame with the raw YUV image (info.data).
15    av_image_fill_arrays(pFrameYUV->data,pFrameYUV->linesize,
16      info.data,AV_PIX_FMT_YUV420P,width,height,
17      YUV420P_ALIGNMENT);
18
19    // Convert image from YUV420P to RGB24.
20    int ret = sws_scale(img_convert_ctx,pFrameYUV->data,
21      pFrameYUV->linesize,0,height,pFrameBGR->data,
22      pFrameBGR->linesize);
23
24    if(ret <= 0)
25      BOOST_THROW_EXCEPTION(IOException(std::string("Input␣frame␣
          conversion␣failed.")));
26
27    // Set pointer to beginning of imagedata.
28    uint8_t* dataPointer = pFrameBGR->data[0];
29    Image8U& img = *image;
30
31    // Copy contents into the allocated Image8U frame.
32    for(uint32 y = 0; y < height; y++)
33      for(uint32 x = 0; x < width; x++)
34      {
35        img(x,y,0) = *dataPointer++;
36        img(x,y,1) = *dataPointer++;
37        img(x,y,2) = *dataPointer++;
38      }
39
40    do {
41      seamReturn = seamCropCuda.addFrame(currentMaxFrame, image);
42    } while(!seamReturn);
43
44    // Start SeamCropCuda when the first frame has been added.
45    if(!started) {
46      seamExec = boost::thread(startSeamCrop);
47      started = true;
48    }
49
50    // Update the frame counter. Will reset to 0 when
        frameWindowSize is reached.
```

```
51    currentMaxFrame = (currentMaxFrame + 1) % frameWindowSize;
52
53    gst_buffer_unmap(inbuf, &info);
54  }
55
56  void SeamCropWrapper::passOutbuf(GstBuffer *outbuf)
57  {
58    GstMapInfo info;
59    gst_buffer_map(outbuf, &info, GST_MAP_READ);
60    writer->addBuffer(outbuf);
61    gst_buffer_unmap(outbuf, &info);
62  }
```

BufferWriter

The BufferWriter is much simpler than the SeamCropWrapper as it is only used by the retargeting module to write to the output queue. It is initialized and started from the SeamCropWrapper, allocating two AVFrame objects for the frame conversion. It has only one important function, putImage, presented in Listing 4.8. This function converts the frame back to the received format, inserts it into an allocated output buffer and pushes it onto the output queue, analogous to the inverse operation of the passFrame function in the SeamCropWrapper.

Listing 4.8: Outputting a frame.

```
1
2  void BufferWriter::addBuffer(GstBuffer *outbuf)
3  {
4    outBuffers.push(outbuf);
5  }
6
7  void BufferWriter::putImage(Image8U const& image)
8  {
9    GstBuffer *outbuf;
10   GstMapInfo info;
11
12   /* Retrieve output buffer. */
13   outbuf = outBuffers.front();
14   outBuffers.pop();
15
16   /* Retrieve buffer info. */
17   gst_buffer_map(outbuf, &info, GST_MAP_WRITE);
18
19   /* Write correct alignment for the YUV420P image to info.data.
         */
20   av_image_fill_arrays(pFrameYUV->data,pFrameYUV->linesize,
21     info.data,AV_PIX_FMT_YUV420P,width,height,YUV420P_ALIGNMENT)
           ;
```

```
22
23    /* Fill pFrameBGR with data from the provided image. */
24    uint8_t* dataPointer = pFrameBGR->data[0];
25    for(uint32 y=0;y<height;y++)
26      for(uint32 x=0;x<width;x++)
27        for(uint32 z=0;z<3;z++)
28          *dataPointer++ = image(x, y, z);
29
30
31    /* Convert the BGR24 image to YUV420P. */
32    int ret = sws_scale(img_convert_ctx,pFrameBGR->data,
33      pFrameBGR->linesize,0,height,pFrameYUV->data,
34      pFrameYUV->linesize);
35
36    if(ret <= 0)
37      BOOST_THROW_EXCEPTION(IOException(std::string("Output␣frame␣
          conversion␣failed.")));
38
39    /* Insert output buffer into queue. */
40    g_async_queue_push(outputQueue, outbuf);
41    currentFrame++;
42
43    gst_buffer_unmap(outbuf, &info);
44  }
```

4.4 Retargeting Module Implementation

In Sections 3.3.9 and 3.3.10, we describe the design of the retargeting module with regards to flow, threading and alteration of the algorithm. In this section, we present how this is implemented.

Retargeting is driven by objects of the two main classes SeamCrop and SeamCropPipeline. The former manages the latter, where the SeamCrop object is managed by the thread dispatched from the SeamCropWrapper. Objects of class SeamCropPipeline are worker threads, operating solely within a shared environment.

As mentioned previously, several alterations and additions were required to modify the original implementation for online retargeting purposes. These changes and additions can be summarized as follows:

- replacement of input and output functionality,
- memory storage of video frames,
- frame window approach to retargeting process,

- cropping window path smoothing between frame windows,
- accounting for seams and temporal cost between frame windows.

The original implementation reads frames from a video file and outputs them to a separate file through the *FFmpeg* framework. Since it relies on files as a video source, no intermediate storage in memory is required between the passes as a file can simply be read twice. Once a frame has been read and uploaded to the GPU it is discarded from main memory. This is not possible for when streaming video from a remote location, where each frame only arrives once and must be stored until the retargeting process has finished.

The wrapper classes SeamCropWrapper and BufferWriter replace the original input and output functionality. Frames received from the SeamCropWrapper are stored in a shared Image8U array with a size equal to the *frame-window-size* property set at plugin initialization. Worker threads retrieve frames from this shared array. A frame is removed from this array once it has been uploaded to the GPU in the second pass of the algorithm and is subsequently replaced with a frame from the next frame window.

The remaining alterations are described throughout this section.

Initialization SeamCrop is instantiated from within the SeamCropWrapper object constructor. The SeamCrop constructor, presented in Listing 4.9, computes all necessary data to perform retargeting, storing this information in the static structure FrameInfo. This structure is made available to SeamCropPipeline threads through the setFrameInfo function. It initializes the SeamCropPipeline worker threads and allocates space for the calculation of the cropping windows and column costs for the energy mapping. The size of a frame window is given with the numFrames argument.

Listing 4.9: Initializing the SeamCrop object.

```
1  SeamCrop::SeamCrop(uint32 videoWidth, uint32 videoHeight, uint32
       numFrames, float retargetingFactor, float extendBorderFactor
     )
2  {
3    uint32 w = videoWidth;  // width
4    uint32 h = videoHeight; // height
5    uint32 c = 3;           // image channels (R,G,B)
6    uint32 fc = numFrames;  // total framecount
7
8    // Set frameinfo.
9    fi.vWidth = w;
10   fi.tWidth = w * retargetingFactor;
11   fi.eWidth = fi.tWidth + ((w - fi.tWidth) * extendBorderFactor)
       ;
```

```
12      fi.oWidth = w - fi.tWidth + 1;
13      fi.numSeams = fi.eWidth - fi.tWidth;
14      fi.height = h;
15      fi.channels = c;
16      fi.frameCount = fc;
17
18      SeamCropPipeline::setFrameInfo(fi);
19      SeamCropPipeline::originalVideo.resize(fi.frameCount);
20
21      for(uint32 i = 0; i < NUM_THREADS; ++i)
22      scp[i] = boost::shared_ptr<SeamCropPipeline>(new
            SeamCropPipeline(i));
23
24      columnCost = boost::shared_ptr<CudaImage32FHandle>(new
            CudaImage32FHandle);
25      columnCost->allocate(fi.vWidth, fi.frameCount, 1, 1);
26
27      croppingWindowCost = boost::shared_ptr<CudaImage32FHandle>(new
            CudaImage32FHandle);
28      croppingWindowCost->allocate(fi.oWidth, fc, 1, 1);
29
30      predecessors = boost::shared_ptr<CudaImage32FHandle>(new
            CudaImage32FHandle);
31      predecessors->allocate(fi.oWidth, fc, 1, 1);
32
33      cropLeftGPU = boost::shared_ptr<CudaVectorHandle<unsigned int>
            >(new CudaVectorHandle<unsigned int>);
34      cropLeftGPU->resize(fc);
35
36      cropLeft.resize(fc);
37
38      prevCropLeft = 0;
39      firstPreSmoothCropLeft = 0;
40      totalRetargetedFrames = 0;
41      pendingFrames = fc;
42
43      SeamCropPipeline::setColumnCost(columnCost);
44
45      endOfStream = false;
46      SeamCropPipeline::haltExecution = false;
47  }
```

setFrameInfo sets up the shared variables used to ensure consistent operation and synchronization between SeamCropPipeline threads, shown in Listing 4.10. It initializes six arrays whose importance merit further explanation.

- originalVideo contains all raw frames received from the SeamCropWrapper.
- imgPresent indicates whether a frame has been added to the internal queue.

- imgAvailable denotes whether a frame has been uploaded to the GPU in this pass.
- imgDone specifies whether a frame has finished processing for a pass.
- seamsDone contains the number of seams currently identified for each frame.
- seams contains the seams identified for each frame.

Listing 4.10: Setting the frame info.

```
1  void SeamCropPipeline::setFrameInfo(FrameInfo const& fi)
2  {
3    SeamCropPipeline::fi = fi;
4    firstPass = true;
5    wraparound = false;
6    lastFrameNumber = 0;
7    lastFrameOfVideo = fi.frameCount;
8    lastFrameOfStream = fi.frameCount+2;
9
10   imgBuffer.resize(fi.frameCount+2);
11   imgAvailable.resize(fi.frameCount+2);
12   imgDone.resize(fi.frameCount+2);
13
14   imgPresent.resize(fi.frameCount+2);
15
16   seams.resize(fi.frameCount+2);
17   seamsDone.resize(fi.frameCount+2);
18
19   imgPresent[0]   = false;
20   imgAvailable[0] = true;
21   imgDone[0] = true;
22
23   seams[0] = boost::shared_ptr<CudaImage32FHandle>(new
         CudaImage32FHandle);
24   seams[0]->allocate(1, 1, 1, 1);
25   seamsDone[0] = fi.numSeams;
26
27   for(uint32 i = 1; i <= fi.frameCount; ++i)
28   {
29     imgAvailable[i] = false;
30     imgPresent[i] = false;
31     imgDone[i] = false;
32     seamsDone[i] = 0;
33   }
34
35   imgAvailable[fi.frameCount+1] = true;
36   imgDone[fi.frameCount+1] = true;
37   seamsDone[fi.frameCount+1] = fi.numSeams;
38 }
```

The SeamCropPipeline threads allocate space for their own operation since all shared variables are instantiated in the setFrameInfo function. As this initialization has not been altered it is not presented here.

Operation The retargeting process entails ample use of a wide array of functions. As this number is quite comprehensive, we limit the presentation to the most important functions we have altered. These are described in the order they are invoked through the data flow.

addFrame adds a frame to the originalVideo frame array as shown in Listing 4.11. It is invoked before the retargeting module begins retargeting, as the module has no need to begin until a frame has been added. If the module is flushing, i.e., discarding all current frames and tearing down the environment, the incoming frame is immediately discarded. To avoid overwriting an unfinished frame with a new one, we refer to the imgPresent value for the frame number of the particular frame. The value is set after the frame has been added and unset once it has finished processing. These two operations account for the synchronization required between the SeamCropWrapper and the retargeting module.

Listing 4.11: Adding a frame to the internal queue.

```
1  bool SeamCrop::addFrame(unsigned int frameNum, const boost::
       shared_ptr<Image8U> &image)
2  {
3    if(SeamCropPipeline::haltExecution)
4      return true; // Module is flushing.
5
6    if(SeamCropPipeline::imgPresent[frameNum])
7      return false; // The frame cannot be added yet.
8
9    if(!SeamCropPipeline::wraparound)
10     SeamCropPipeline::originalVideo[frameNum] = boost::
           shared_ptr<Image8U>(new Image8U(fi.vWidth, fi.height, fi.
           channels));
11
12   // Adds image to originalVideo[].
13   Filter::resize(*image, *SeamCropPipeline::originalVideo[
         frameNum]);
14   SeamCropPipeline::imgPresent[frameNum] = true;
15
16   return true;
17 }
```

run begins the retargeting process. This function is presented in Listing 4.12. It is the implementation of the algorithm we presented in Algorithm 1, dispatching threads for each pass and determining the cropping window for each

frame window. In the original implementation, this function is only executed once for the entire retargeting process. In this altered version, each loop iteration corresponds to retargeting one frame window. The loop within this function executes until either all frames have been processed and endOfStream is set by the SeamCropWrapper, or if haltExecution is set. endOfStream is set when the EOS signal is received in the plugin component, while haltExecution is set upon reception of the FLUSH signal. Since the majority of structures are allocated as shared pointers, dereferencing them is sufficient to deallocate the reserved space [39]. These structures are dereferenced in lines 49-52 by assigning the shared pointers a new, empty pointer.

Listing 4.12: Main loop of the retargeting module.

```
1  void SeamCrop::run()
2  {
3    SeamCropPipeline::setCropLeft(&cropLeft);
4    SeamCropPipeline::setPreSmoothCropLeft(&firstPreSmoothCropLeft
       );
5
6    while(true){
7      if(SeamCropPipeline::haltExecution)
8        break;
9      totalRetargetedFrames += pendingFrames;
10
11     run_pass1();
12
13     // Calculate cropping window path.
14     SeamCropCuda::calculateCostCroppingWindowTime(*columnCost->
         getDataDescPtr(),
15       *croppingWindowCost->getDataDescPtr(), fi.tWidth);
16     SeamCropCuda::calculateMaxEnergyPath(*croppingWindowCost->
         getDataDescPtr(),
17       *predecessors->getDataDescPtr(), *cropLeftGPU->
           getDataDescPtr());
18     cropLeftGPU->getData(cropLeft);
19
20     // Smooth transition between frame windows.
21     if(SeamCropPipeline::wraparound)
22       smoothTransition();
23
24     // Smooth the cropping window path.
25     smoothSignal();
26
27     // Switch operation from first to second pass.
28     SeamCropPipeline::nextPass(PASS_TWO);
29
30     run_pass2();
31
```

```
32      if((endOfStream && pendingFrames == 0) || SeamCropPipeline::
            haltExecution)
33        break;
34
35      SeamCropPipeline::nextPass(PASS_ONE);
36    }
37
38    // Finished. Reset all structures for a potential new run.
39    for(int i = 0; i < fi.frameCount; i++)
40    {
41      if(i < NUM_THREADS)
42        scp[i] = boost::shared_ptr<SeamCropPipeline>();
43
44      SeamCropPipeline::originalVideo[i] = boost::shared_ptr<
            Image8U>();
45      SeamCropPipeline::imgPresent[i] = false;
46      SeamCropPipeline::wraparound = false;
47    }
48
49    columnCost = boost::shared_ptr<CudaImage32FHandle>();
50    croppingWindowCost = boost::shared_ptr<CudaImage32FHandle>();
51    predecessors = boost::shared_ptr<CudaImage32FHandle>();
52    cropLeftGPU = boost::shared_ptr<CudaVectorHandle<unsigned int>
          >();
53  }
```

doStuff is the arbitrarily named operational function of the SeamCropPipeline threads. It loops execution until there are no remaining frames to process in the current frame window. To prevent the acquisition of a frame that has not been added yet, readNextFrame is continually called until one is available. On success, the thread proceeds; on failure, it yields. This function is described in Listing 4.14. By calling cudaStreamSynchronize after a frame has begun uploading to the GPU, we ensure that the frame upload is complete before a thread can operate on it. As a frame cannot be processed until both the preceding and succeeding frame is present on the GPU, each thread must yield until this has come to pass. A frame is removed from the GPU once the previous, current and next frame are finished. The first frame to exit the loop notifies the managing SeamCrop object that the pass is nearing completion.

Listing 4.13: Operational loop of a SeamCropPipeline thread.

```
1   void SeamCropPipeline::doStuff()
2   {
3     if(lastFrameOfVideo < fi.frameCount &&
4       !imgDone[lastFrameOfVideo+1])
5     {
6       /* When processing the last frame window,
7       sets the last actual frame on stream end.
```

```
8       Otherwise, the last thread will wait forever
9       for a frame that never arrives. */
10      imgAvailable[lastFrameOfVideo+1] = true;
11      imgDone[lastFrameOfVideo+1] = true;
12    }
13
14    if(imgDone[1] || haltExecution)
15    {
16      cond.notify_one();
17      boost::this_thread::yield();
18      return;
19    }
20
21    // Processing loop.
22    while(true)
23    {
24      int32_t t = readNextFrame();
25
26      // Attempt to read next frame until it is available.
27      while(t < 0) {
28        boost::this_thread::yield();
29        t = readNextFrame();
30        if(haltExecution)
31          t = 0;
32      }
33      if(t == 0 || haltExecution)
34        break;  // No more frames to process.
35
36      // Upload frame to the GPU.
37      imgBuffer[t] = boost::shared_ptr<CudaImage8UHandle>(new
            CudaImage8UHandle);
38      imgBuffer[t]->put(*readFrameCPU, stream);
39
40      // Ensure that the frame is uploaded before proceeding.
41      cudaStreamSynchronize(stream);
42      imgAvailable[t] = true;
43
44      // Allow the frame to be overwritten.
45      if(!firstPass)
46        imgPresent[t-1] = false;
47
48      while(!(imgAvailable[t-1] && imgAvailable[t] && imgAvailable
            [t+1])){
49        boost::this_thread::yield();
50        if(haltExecution)
51          break;
52      }
53
54      if(firstPass)
```

```
55          processImage_pass1((uint32)t);
56        else
57          processImage_pass2((uint32)t);
58
59        imgDone[t] = true;
60
61        // See if a frame can be removed from the GPU.
62        for(uint32 i = 1; i <= lastFrameOfVideo; ++i)
63          if(imgDone[i-1] && imgDone[i] && imgDone[i+1])
64            imgBuffer[i] = boost::shared_ptr<CudaImage8UHandle>();
65    }
66  }
```

readNextFrame attempts to retrieve a frame from the originalVideo array. This is one of the few functions in which an explicit mutex is utilized to maintain synchronization. A boost::unique_lock ensures singular access to the function. If the desired frame is absent, the process returns and subsequently yields until it becomes available. Should the counter exceed the frame window size, there are no more frames to process and the thread exits the processing loop.

Listing 4.14: Retrieving a frame from the internal queue.

```
1  int32_t SeamCropPipeline::readNextFrame()
2  {
3    boost::unique_lock<boost::mutex> lock(mutex);
4
5    if(lastFrameNumber >= lastFrameOfVideo)
6      return 0;
7    else if(!imgPresent[lastFrameNumber])
8      return -1;
9
10   // Frame is available.
11   readFrameCPU = originalVideo[lastFrameNumber];
12
13   lastFrameNumber += 1;
14   return lastFrameNumber;
15 }
```

processImage_pass1 and calculateEnergy functions are the core functions in the first pass of the algorithm. They are fairly sparse as the actual calculations are performed by the CUDA implementation. Energy values for the preceding and following frames are included in the energy calculation provided the frame is neither the first nor the last frame of the frame window.

Listing 4.15: Energy calculation.

```
1  void SeamCropPipeline::processImage_pass1(uint32 t)
2  {
3    calculateEnergy(t);
```

```
4     cuda->addColumnCost(*totalSaliency->getDataDescPtr(), *
          columnCost->getDataDescPtr(), t-1);
5   }
6
7   void SeamCropPipeline::calculateEnergy(uint32 t)
8   {
9     CudaImage8UDataDescriptor const* prevFrame = imgBuffer[t]->
          getDataDescPtr();
10    cuda->computeGradient(*prevFrame, *gradient->getDataDescPtr())
          ;
11
12    CudaImage8UDataDescriptor const* nextFrame = prevFrame;
13
14    if(t < lastFrameOfVideo)
15      nextFrame = imgBuffer[t+1]->getDataDescPtr();
16    if(t > 1)
17      prevFrame = imgBuffer[t-1]->getDataDescPtr();
18
19    cuda->findMotionSaliency(*prevFrame, *nextFrame, *
          motionSaliency->getDataDescPtr());
20    cuda->smooth(*motionSaliency->getDataDescPtr());
21
22    cuda->mergeSaliency(*motionSaliency->getDataDescPtr(), *
          gradient->getDataDescPtr(), *totalSaliency->getDataDescPtr
          ());
23  }
```

smoothTransition performs the transitional smoothing between cropping windows of two different frame windows as visualized in Figure 3.8, executed after the first pass is finished. This function, presented in Listing 4.16, is crucial to maintain presentation consistency between frame windows. The position of the last cropping window is stored in prevCropLeft by the managing SeamCrop object upon completion of the smoothSignal function. This position is compared with the first cropping window position calculated for the next frame window.

The cropLeft value for each cropping window is smoothed by utilizing the importanceWeight variable which initially favors the previous window position. This importance gradually abates until the cropping window path eventually follows the one that was originally computed. This smoothing is only performed if the disparity between the left cropping position of both windows exceeds a distance of two pixels. More pronounced disparities result in a jagged transition between the two frame windows.

Listing 4.16: Transitional smoothing.

```
1   void SeamCrop::smoothTransition()
2   {
3     int diff = prevCropLeft - cropLeft[0];
```

```
4     if(diff <= 2 && diff >= -2)
5     {
6       // Difference is negligible. No need to smooth the
            transition.
7       firstPreSmoothCropLeft = prevCropLeft;
8       return;
9     }
10
11    float weightedPrevCrop;
12    float weightedCurCrop;
13    float importanceWeight = (1 / (float)SeamCropPipeline::
          lastFrameOfVideo);
14
15    for(uint32 i = 0; i < SeamCropPipeline::lastFrameOfVideo; ++i)
16    {
17      // Gradient smoothing between the cropping window positions.
18      weightedPrevCrop = prevCropLeft - (prevCropLeft * (i *
            importanceWeight));
19      weightedCurCrop = cropLeft[i] * (i * importanceWeight);
20      cropLeft[i] = weightedPrevCrop + weightedCurCrop;
21    }
22    firstPreSmoothCropLeft = cropLeft[0];
23  }
```

smoothSignal and defineBorders functions smooth the cropping window differences within the transitional path we computed. smoothSignal performs repeated smoothing on the path to make the perceived camera motion palatable, while defineBorders determines whether the calculated position exceeds the edges of the original video frame. Should the cropping window position for a frame be too far to one side, additional seams are removed from the opposing side equal to the amount of pixels outside the boundary. Both of these functions are presented in Listing 4.17.

Listing 4.17: Intra-path smoothing and border definition.

```
1   void SeamCrop::smoothSignal()
2   {
3     uint32 const fc = SeamCropPipeline::lastFrameOfVideo;
4
5     float *next = new float[fc];
6     float *prev = new float[fc];
7
8     if(next == NULL || prev == NULL)
9       BOOST_THROW_EXCEPTION(RuntimeException("Out␣of␣memory."));
10
11    for(uint32 i = 0; i < fc; ++i)
12      prev[i] = cropLeft[i];
13
14    // gauss-based average, repeated;
```

```
15    for(uint32 repeat = 0; repeat < 100; repeat++)
16    {
17      for(uint32 i = 0; i < fc; ++i)
18        if(i == 0)
19          if(!SeamCropPipeline::wraparound)
20            next[i] = (prev[i] + prev[i+1])/2.0f; // (i + (i+1))/2
21          else
22            next[i] = prev[i];
23          else if (i < fc-1)
24            next[i] = 0.25f*prev[i-1] + 0.5f*prev[i] + 0.25f*prev[
                  i+1]; //(0.25 + 0.5 + 0.25)
25        else
26          next[i] = (prev[i] + prev[i-1])/2.0f;
27
28      float* tmp = prev;
29      prev = next;
30      next = tmp;
31    }
32
33    float smoothWeight = (1 / (float)SeamCropPipeline::
          lastFrameOfVideo);
34
35    for(uint32 i = 0; i < fc; ++i)
36      cropLeft[i] = defineBorders((uint32)(prev[i] + 0.5f),i,
            smoothWeight);
37
38    // Remove small one-pixel jitters.
39    for(uint32 i = 1; i < fc; ++i)
40      if(cropLeft[i-1] == cropLeft[i+1] && cropLeft[i] != cropLeft
            [i+1])
41        cropLeft[i] = cropLeft[i-1];
42
43    // Store the last position to transition between frame windows
          .
44    prevCropLeft = cropLeft[fc-1];
45
46    delete prev;
47    delete next;
48  }
49
50  uint32 SeamCrop::defineBorders(uint32 cropLeft, uint32 curFrame,
        float smoothWeight)
51  {
52    uint32 const w = fi.vWidth;
53    uint32 const ew = fi.eWidth;
54    uint32 const tw = fi.tWidth;
55
56    int extraSpace = (ew - tw)/2;
57
```

```
58    if(SeamCropPipeline::wraparound)
59    {
60      // Transitioning between windows.
61      int adjustedSpace = extraSpace * (curFrame * smoothWeight);
62
63      if(((int) cropLeft) - adjustedSpace <=0)
64        cropLeft = 0;
65      else if((cropLeft - adjustedSpace) > (w - ew - 1))
66        cropLeft = w - ew - 1;
67      else
68        cropLeft = cropLeft - adjustedSpace;
69    } else
70    {
71      if(((int)cropLeft) - extraSpace <= 0)
72        cropLeft = 0;
73      else if((cropLeft + tw + extraSpace) >= w-1)
74        cropLeft = w - ew - 1;
75      else
76        cropLeft = cropLeft - extraSpace;
77    }
78    return cropLeft;
79  }
```

nextPass function alters and resets the shared variables and structures necessary to operate in the next pass, shown in Listing 4.18. Both passes need to clear the imgAvailable and imgDone shared variables to proceed, while the first pass additionally needs to reset seamsDone for a new frame window. The value of firstPass determines which pass to execute in the doStuff function while wraparound indicates the switch to a new frame window.

Listing 4.18: Switching between passes.

```
1   void SeamCropPipeline::nextPass(int passMode)
2   {
3     switch(passMode)
4     {
5       case PASS_ONE:
6         firstPass = true;
7         wraparound = true;
8         seamsDone[0] = fi.numSeams;
9         break;
10      case PASS_TWO:
11        firstPass = false;
12        break;
13    }
14    lastFrameNumber = 0;
15
16    for(uint32 i = 1; i <= fi.frameCount; ++i)
17    {
```

```
18        imgAvailable[i] = false;
19        imgDone[i] = false;
20        if(passMode == PASS_ONE)
21          seamsDone[i] = 0;
22      }
23  }
```

processImage_pass2 is the final and most important stage of retargeting where the actual cropping and seam carving is performed. This function is presented in Listing 4.19. On the first frame window, this function proceeds normally as it would in the original implementation. When transitioning between frame windows, it takes the seams computed for the final frame in the previous frame window into account, as shown on line 24. These seams are stored in prevSeams on line 74, to be used for the initial seam computation of the next frame window. Without accounting for these seams, the temporal consistency between the seams computed for each frame are lost, resulting in rapid expansion or compression of objects or scenery when transitioning between frame windows. When a frame is finishied, it is passed to the putImage function of the BufferWriter on line 79.

Listing 4.19: Cropping and carving seams.

```
1  void SeamCropPipeline::processImage_pass2(uint32 t)
2  {
3    uint32 const numSeams = fi.numSeams;
4
5    seams[t] = boost::shared_ptr<CudaImage32FHandle>(new
         CudaImage32FHandle);
6    seams[t]->allocate(fi.numSeams, fi.height, 1, 1);
7
8    CudaImage8UDataDescriptor const& origFrame = *imgBuffer[t]->
         getDataDescPtr();
9    CudaImage8UDataDescriptor const& frameData = *frame->
         getDataDescPtr();
10   CudaImage32FDataDescriptor const& energyData = *energy->
         getDataDescPtr();
11   CudaImage32FDataDescriptor const& tmpEnergyData = *tmpEnergy->
         getDataDescPtr();
12   CudaImage32FDataDescriptor const& fwdEnergyData = *fwdEnergy->
         getDataDescPtr();
13   CudaImage32FDataDescriptor const& optimalCostData = *
         optimalCost->getDataDescPtr();
14   CudaImage32FDataDescriptor const& predecessorsData = *
         predecessors->getDataDescPtr();
15   CudaImage32FDataDescriptor const& seamsData = *seams[t]->
         getDataDescPtr();
16
17   while(seamsDone[t-1] < numSeams)
18   {
```

```
19      boost::this_thread::yield();
20      if(haltExecution)
21        return;
22    }
23
24    CudaImage32FDataDescriptor const& prevSeamsData = (wraparound
        && t == 1) ? *prevSeams->getDataDescPtr() : *seams[t-1]->
        getDataDescPtr();
25
26    calculateEnergy(t);
27
28    unsigned int& curCropLeft = (*cropLeft)[t-1];
29
30    cuda->cropImage8U(origFrame, frameData, curCropLeft);
31    cuda->cropImage32F(*totalSaliency->getDataDescPtr(),
        energyData, curCropLeft);
32
33    cuda->computeForwardEnergy(frameData, fwdEnergyData);
34
35    int32 cropOffset = 0;
36    if(t > 1)
37      cropOffset = ((*cropLeft)[t-2] - curCropLeft);
38    else if(wraparound)
39      cropOffset = ((*preSmoothCropLeft) - curCropLeft);
40
41    for(uint32 seamID = 0; seamID < numSeams; ++seamID)
42    {
43      bool copyEnergy = true;
44
45      while(seamsDone[t-1] <= seamID)
46      {
47        boost::this_thread::yield();
48        if(haltExecution)
49          return;
50      }
51
52      if(t > 1 || wraparound)
53        copyEnergy = cuda->addTemporalCoherenceCost(energyData,
             tmpEnergyData, prevSeamsData, seamID, cropOffset);
54
55      if(copyEnergy)
56        *tmpEnergy = *energy;
57
58      cuda->computeCostWidth(tmpEnergyData, fwdEnergyData,
           optimalCostData, predecessorsData);
59      cuda->markSeamWidth(optimalCostData, energyData,
           predecessorsData, seamsData, seamID);
60      // necessary to prevent the next thread from reading the
           seam info too soon.
```

```
61        cudaStreamSynchronize(stream);
62
63        seamsDone[t] = seamID+1;
64      }
65
66      cuda->removeSeams(frameData, *finalFrame->getDataDescPtr(),
            seamsData);
67      finalFrame->getImage8UData(*writeFrameCPU, stream);
68
69      while(!imgDone[t-1])
70        boost::this_thread::yield();
71
72      // Maintain a pointer to the seam data of the final frame in
            this retargeting window.
73      if(t == lastFrameOfVideo)
74        prevSeams = boost::shared_ptr<CudaImage32FHandle>(seams[t]);
75
76      seams[t-1] = boost::shared_ptr<CudaImage32FHandle>();
77
78      // Pass the finished image to the writer.
79      writer->putImage(*writeFrameCPU);
80    }
```

Chapter 5

Evaluation

In this chapter, we evaluate the performance and usage of our video retargeting plugin. Section 5.1 presents the goals of our evaluation. In Section 5.2, we discuss the evaluation techniques we employ to evaluate the technical aspects. The metrics we use for the evaluation are described in Section 5.3. Parameters and factors that influence or affect the operation and results are presented in Section 5.4. Our evaluation setups are presented in Section 5.5 and Section 5.6 presents the results of the experiments. Finally, in Section 5.7 we discuss the results presented in Section 5.6.

5.1 Goals

The primary goals for our evaluation is to demonstrate that our plugin can be used for real-time, online video retargeting of video content provided by an adaptive streaming technique. We evaluate the performance of our video retargeting plugin in relation to the requirements we presented in Section 1.3.

- **Goal 1:** To determine if our plugin can be used for real-time retargeting of video content, we analyze the frame rate produced by the element for various input streams. As the frame rate of a video determines the playback rate, the output rate of our element is required to match or exceed the frame rate of the video for seamless concurrent playback. The minimal requirement is to support real-time retargeting of a video presented at 25 frames per second for a resolution of at least 640x360 pixels with a retargeting severity of 25%. Additionally, we investigate the impact of the step we introduced to the algorithm in Section 3.3.9, comparing the total time spent performing the step with the total running time of the algorithm.

- **Goal 2:** Ramesh et. al. show that a startup latency exceeding 2 seconds

when streaming results in increased viewer abandonment [10]. We investigate whether a startup latency less than 2 seconds is feasible for multiple resolutions and retargeting factors. As the delay introduced by our element is added to the overall startup latency, it should ideally be lower. However, as our element is intended for deliberate use, we claim that an added latency of this magnitude is tenable. The minimal requirement is as such to satisfy this latency requirement for at least one evaluation factor configuration which satisfies the previous goal. We use the results of this investigation and the previous to determine whether the implemented solution is useful for multiple configurations.

- **Goal 3:** We evaluate the CPU and memory load of the GStreamer plugin to determine the consumption of available machine resources. The CPU load should be considerable due to the threading aspect of our element, but should not on average consume more resources than the equivalent of 6 cores operating at full capacity. The memory load should be consistent for a specific configuration of the evaluation factors, and we examine whether this is the case to identify possible memory leaks.

- **Goal 4:** We investigate whether the plugin is compatible and efficient with adaptive bitrate streaming systems. As these streaming techniques are widely utilized, our plugin must be able to receive and retarget video streams from applications using them for both versatility and general compatibility. We evaluate this by retargeting separate streams utilizing these techniques, examining whether our element properly handles dynamic capability changes.

5.2 Evaluation Approach

When performing an evaluation of the solution to a problem, there are three different approaches that need to be considered. These approaches are *modeling*, *simulation* and *measurement* [40].

Modeling is a theoretical approach which presents a mathematical model and calculates the performance for a specific solution. This approach is viable for simple solutions, but as the complexity of the solution increases, so does the complexity of the model.

Simulation is the approach of testing a solution within a controlled, simulated environment. When performing simulations, factors within the environment that have an impact on the simulation are controlled and accounted for. If executed correctly, repeated simulations with the same parameters will always yield the same results. Simulation is viable if a solution relies on a small number of factors

and is simple to implement in a simulation environment. Conversely, accurate simulations are difficult to achieve if they depend on many different factors.

Should simulation be too difficult due to an excess of uncontrollable factors and dependencies, a final option is to perform measurements. Measurements are performed on an implementation of the solution in a real environment. As a result of this environment, measurement results can exhibit a great deal of variability due to the presence of uncontrollable factors. Unlike simulations, reproducing the same results can be difficult. While the impact of these factors can be reduced to a minimum, they will inevitably affect the measurements made.

Each of these approaches have their own respective benefits and detriments and can be chosen based on an array of factors, such as the metrics being analyzed or the complexity of the evaluation. Modeling or simulating the CPU load, for example, is problematic due to many uncontrollable factors. Similarly, memory load shares these difficulties as factors such as excessive buffering can lead to inflated memory usage. As such, we utilize the measurement approach when evaluating both CPU usage and memory load.

Since DASH and HLS streaming is inherently complex and prone to frequent changes, it is difficult to simulate due to the sheer variety of factors involved. Accounting for all these factors is problematic, making it difficult to ascertain the generalizability of any results obtained from simulations. For the same reasons, utilizing the modeling approach is likely to result in an exceedingly complex model of dubious accuracy.

Due to these difficulties, we also utilize the measurement approach when evaluating the retargeting rate and latency. However, to ensure comparable evaluations, we require the ability to control all the evaluation factors involved in this process. This is problematic, as support for DASH and HLS in the GStreamer framework is still in its relative infancy when it comes to managing the incoming stream. Plugins for parsing DASH/HLS streams such as $\mathrm{dashdemux}$ and $\mathrm{hlsdemux}$ do not offer functionality for either requesting an input resolution or limiting the available bandwidth to trigger this resolution change. As a result, the resolution of an incoming DASH/HLS stream can not be controlled through these elements.

In an attempt to circumvent this limitation, we tried to utilize a separate tool, *wondershaper*, to limit the bandwidth on our network interface of choice. Since stream resolution is associated with varying degrees of available bandwidth, where more bandwidth generally correlates to a better transmitted video resolution, the intent was to trigger resolution adaptation by throttling or increasing the available bandwidth. This assumption is correct, but provoking specific resolutions in this manner proved to be unreliable at best, rendering this approach inadequate.

Consequently, we make use of two additional elements in the GStreamer pipeline of the experiments. These elements are used to control the incoming video resolu-

tion without interfering with the DASH/HLS stream itself, allowing us to perform measurements without resorting to limited simulations or modeling. These elements as well as how the experiments are performed are detailed in Section 5.5.

5.3 Evaluation Metrics

We evaluate the GstSeamCrop plugin against an assortment of separate performance metrics. These metrics are *CPU-load*, *memory load*, *retargeting rate* and *latency*. Each metric and its accompanying analysis approach is described in the following sections.

5.3.1 CPU Load

CPU load is measured by utilizing the *top* command in UNIX. This command measures CPU load in percentage of the available CPU time. By using *top*, we measure both the *seamcrop* element as well as the presentation of the video. It is important to note that top measures the CPU load of an application by reporting the percentage it uses each core, where each core can be utilized up to 100%. For multi-threaded applications, the reported number is as such the total percentage sum of the cores used, which on a CPU with multiple cores can exceed 100%. In the case of a quad-core CPU, a multithreaded application can yield up to 400% CPU usage.

When measuring the CPU load using top, we need to ensure that we obtain a correct average. To do this, we start the top tool at the same time as we initialize the streaming GStreamer pipeline we wish to measure. As it is not possible to separate the impact our plugin in the pipeline from the pipeline itself in the same measurement, the CPU load is measured as the load of the application running the pipeline in its entirety. To approximate the impact of our plugin, we compare the measured CPU load with the CPU load of an identical pipeline without the presence of our plugin. The GStreamer pipeline is run for a set duration, after which top is stopped. The beginning and end of the measurement includes the CPU load for setting up and tearing down the GStreamer pipeline. As these measurements do not reflect the actual average CPU load of the application, we omit them by ignoring the 10 first and last seconds of the measurement time. This prevents these outlying measurements from impacting the calculation of the average CPU load.

5.3.2 Memory Load

Similar to CPU load, memory load is measured by utilizing the *top* tool, as it also reports the memory load percentage of an application. This measurement reports

the percentagewise share of physical memory used by a process [41]. The measurement is initialized and performed by the same measuring process described in the previous section.

5.3.3 Retargeting Rate

There are multiple ways we can measure the retargeting rate. One possibility is to isolate and measure the operation of the retargeting module alone. This omits the impact of both the plugin component as well as the algorithm wrapper, producing the tightest measurement of the retargeting rate exhibited by the retargeting module.

The other option is to measure the buffer transmission rate on the source pad of the element. As we described in Section 3.3.9, the output of the element is bounded by the rate at which the retargeting module can produce it. Since each buffer contains one retargeted frame, counting the amount of buffers sent per second is analogous to the retargeting rate of the element in its entirety.

We choose to utilize the first option, as it is the most viable measurement of the retargeting rate of our element. While the second option gives a realistic reflection of the retargeting rate, it can be deceptive as it is susceptible to extraneous factors impacting the input rate. Additionally, the GStreamer framework synchronizes the transmission of buffers against the pipeline clock and the rate of the video itself, which can restrict the element to output a fixed frame rate. This makes it a less viable approach, as factors outside of our control can exhibit an artificial impact on the reported rate.

There is an argument to be made for including the operation of the plugin component as well as the wrapper when examining this rate. However, we claim that their impact is so minute as to be insignificant for the overall retargeting rate of the element as they only directly affect the retargeting rate during the first frame window of a session.

When measuring the retargeting rate, we utilize 5 separate measurement timers: one for each of the separate operations and one for the algorithm as a whole. These operations are the first pass, transitional smoothing, cropping window computation and the second pass. Each of the timers encapsulate the operations, starting when the task begins, pausing when it is finished and ending when the final frame window is processed. This way, we can observe how much of the time spent retargeting is attributed to each of the different tasks in addition to the overall processing time.

Since the first frame window is required to buffer incoming frames during the first pass, the operation during this frame window is slower than all subsequent frame windows. Including the measurements from this frame window confounds

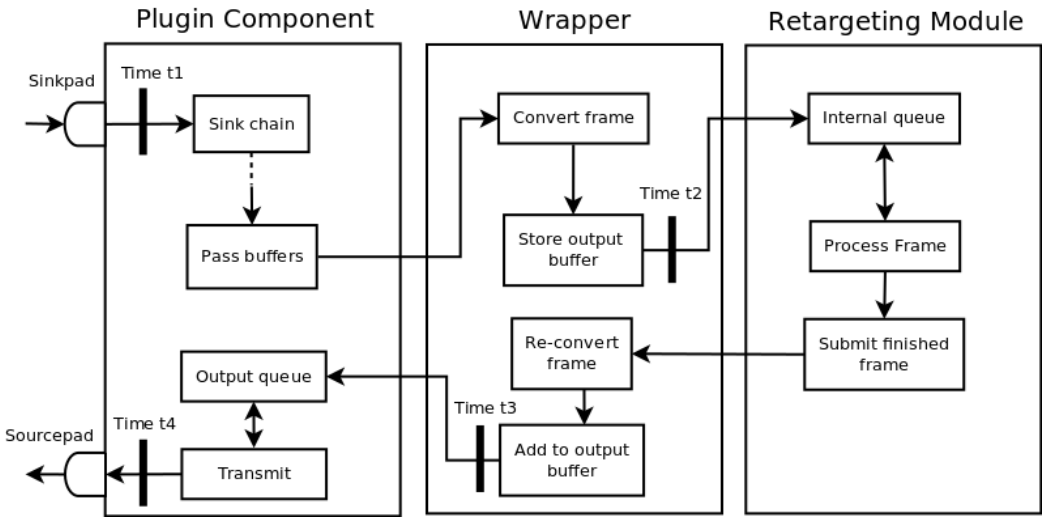


Figure 5.1: Measuring internal element latency.

the retargeting rate, and is as such omitted from the final calculation. The retargeting rate is therefore computed as $RetargetingRate = \frac{Frames - FrameWindowSize}{RetargetingTime}$.

5.3.4 Latency

In a GStreamer pipeline, latency is defined as the time elapsed between buffer transmission from the source element and buffer reception in the sink element. We differentiate between two types of latency: *internal* and *external*. External latency stems from extraneous factors such as network bandwidth and internal latency originates within each element by virtue of processing the input stream. To investigate the delay our element introduces to the pipeline, we measure the internal latency of our element.

The internal latency is effectively the cumulative processing time required to adapt an incoming frame to an altered output frame. We define this latency as the time difference between reception and transmission of each individual frame. This approach is beneficial as it mitigates the effects that the external latency exhibits on the plugin by computing the internal latency on a per-buffer basis as opposed to a total running time. To do this, we utilize the pipeline clock, obtaining time stamps through the *gst_clock_get_time* function at the points shown in Figure 5.1.

The most vital aspect of the internal latency to measure is the initial latency induced by the element. This latency is the cumulative time from reception of the first frame of the first window to the transmission of the first frame in said window. We differentiate between initial latency and start-up latency, as the former is the latency induced by our element while the latter is a measure of the

total time between the initiation request and beginning of the presentation. The internal latency is important because this is the only time during plugin operation where output frames are not produced as well as the only time that the operation of the plugin component/wrapper directly impedes the retargeting process. As retargeting begins immediately upon submission of the first frame, the retargeting module must wait for the following frames to arrive until it can proceed. For all subsequent frame windows, buffering is performed concurrently with frame production. Due to this, we measure the latency incurred for the first frame window to examine the latency our element adds to the pipeline. As the internal latency for all subsequent frame windows never exceeds the initial latency due to continuous frame production, the initial latency is the effective latency exhibited by the element.

Another important latency aspect of our element is the reinitialization time of the retargeting module. As each resolution change occurring in the pipeline results in a reinitialization of the retargeting environment, this operation invariably results in additional latency. Such resolution changes can be quite frequent when streaming under variable bandwidth conditions, and as each change is associated with this added time, it is pertinent to investigate.

These two aspects of the element latency are measured in the following manner:

Initial latency is measured as $t4_0 - t1_0$ for the first frame window. The measurements $t2$ and $t3$ are used to investigate the latency distribution during this window, where $t2_n - t1_n$ and $t4_n - t3_n$ account for the time spent by the plugin component/wrapper for each frame.

Reinitialization latency is measured as $t1_n - t4_{n-1}$ where n is the first frame of the first window after reinitialization and $n - 1$ the last frame of the last window prior to reinitialization.

5.4 Evaluation Factors

During the evaluation of $\mathrm{GstSeamCrop}$ we modify a set of factors to investigate the effect changing the factor has on the results. The factors we employ are *frame window size*, *retargeting factor* and *resolution*, as detailed in Table 5.1. Each of these factors are described over the following sections.

Retargeting factor	15%	25%	50%
Frame window size	50	100	200
Resolution	640x360	854x480	1280x720

Table 5.1: Factor variations during the evaluation.

5.4.1 Frame Window Size

As presented in Section 3.3.9, *frame window size* is the amount of frames the plugin stores and processes concurrently. We vary between a small, medium and large frame window size to observe the impact this factor has on the memory load, latency and retargeting rate of the element. This size is not principally restricted by anything other than available memory and initial latency, but to avoid intentionally exceeding the startup latency threshold presented in Section 5.1, we do not utilize a window size larger than 200 frames.

5.4.2 Retargeting Factor

Similar to the frame window size factor, we use three different configurations for the retargeting factor, ranging from slight to severe. By varying this parameter, we examine the retargeting rates that our element is capable of for various input resolution configurations to determine if real-time retargeting is possible for that particular configuration. Two of these factors are taken from Rubenstein et al. [42], where factors of 25% and 50% are regarded as considerable resizing. The configurations we utilize correspond to situations where there is a slight, medium or significant discrepancy between the incoming aspect ratio and the ratio desired for the presentation. We examine the effect this factor has on latency and retargeting rate.

5.4.3 Resolution

We utilize multiple instances of the same input stream with different resolutions. We vary this parameter to examine its impact on latency, memory load and retargeting rate. Since we utilize a variation of the *Parallelized SeamCrop* algorithm, we do not explicitly investigate resolutions exceeding 1280x720, as these are unlikely to be retargeted in real-time [2]. The resolutions used for this evaluation factor are chosen based on their ubiquity within VoD contexts; all have an inherent aspect ratio of 16:9 and are widely utilized. Testing them should provide enough information about the impact of resolution on the retargeting rate.

Operating system	Ubuntu 15.10 64 bit
CPU	Intel Core i7-2600 CPU @ 3.40GHz x 8 Cores
GPU	NVIDIA GeForce GTX 750 Ti, 2048 MB memory, 640 CUDA cores
Memory	8 GB

Table 5.2: Technical specifications of the testing node.

5.5 Evaluation Setups

We utilize four evaluation setups to analyze the performance of our plugin in relation to the evaluation factors and goals. These setups are *DASH streaming*, *Initial latency*, *Reinitialization latency* and *HLS streaming*, each measuring different aspects of the evaluation metrics. The experiments in the setups described in this section are performed on a node with the hardware specifications listed in Table 5.2. For all experiments, we utilize 4 worker threads in each pass of the algorithm, as this offered the most consistent performance during preliminary testing. Additionally, since we do not vary the *extend window factor*, we utilize a window extension of 20% for all experiments as the same value was used in [3].

Three of the setups detailed in this section utilize a DASH stream providing a video with a resolution of 1280x720. We utilize a DASH stream as opposed to a HLS stream for these setups, as preliminary testing with the dashdemux and hlsdemux elements revealed that the latter was prone to more errors causing the pipeline to crash, regardless of our elements presence. To evaluate that our element is compatible with HLS, the last setup performs experiments on one HLS test sequence. Each setup and how the evaluation is performed is detailed in the following sections.

5.5.1 DASH Streaming

In this setup, we utilize our element in a pipeline during a real video streaming scenario. Figure 5.2 presents the flow of the experiments we perform. The experiments in this setup are used to measure the following metrics:

- CPU load,
- memory load,
- retargeting rate.

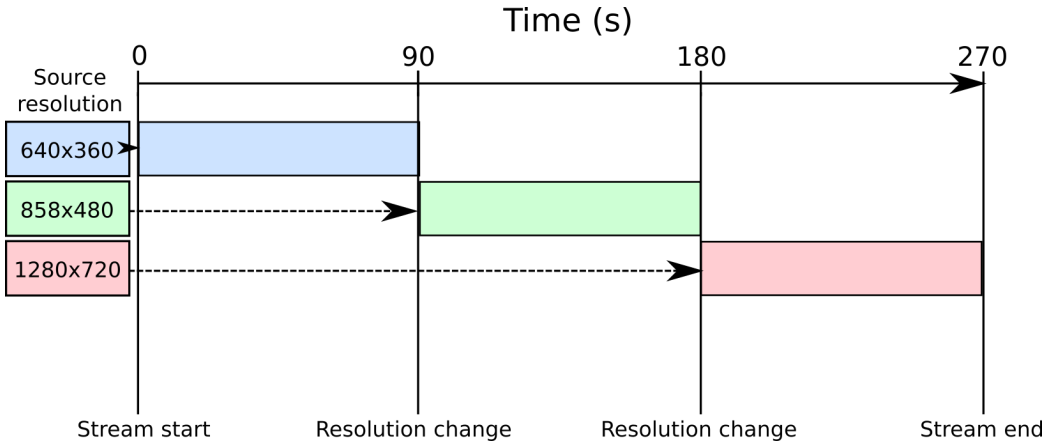


Figure 5.2: Visual representation of each experiment performed in setup 5.5.1.

Each video resolution is streamed for a total of 90 seconds, at which time a resolution change is initiated. This results in a stream which dynamically alters the resolution during runtime without shutting down the pipeline. We choose to stream each resolution for 90 seconds to obtain long term averages for each of the evaluation metrics.

For each of the experiments, we vary the resolution, retargeting and frame window size factors between the values presented in Table 5.1. Each experiment corresponds to one permutation of a retargeting factor and a frame window size performed on all resolutions, and every configuration is measured. The retargeting rate measurements are performed independently for each resolution to obtain the performance metrics for each individual combination of the evaluation factors.

The CPU load and memory load are measured over the course of each experiment with a one-second sampling rate. As the top tool is started at the same time as the GStreamer pipeline, the individual load reports can be directly attributed to each configuration. To obtain the CPU load as well as memory load averages, we omit the first and last 10 seconds of each configuration to eliminate the effects of reinitialization resulting from the resolution changes. Omitting 10 seconds is sufficient to avoid confounding the averages while still providing ample measurements for a representative average.

As explained in Section 5.2, we are not able to directly influence the resolutions we receive from the DASH stream. We work around this issue by forcing specific capabilities in the pipeline prior to our element. To do this, we employ two extra elements in the pipeline: videoscale and capsfilter. The resulting pipeline is presented in Listing 5.1.

Listing 5.1: The core pipeline used for the DASH streaming experiments.

```
1 gst-launch-1.0 uridecodebin ! videoscale ! capsfilter ! queue !
```

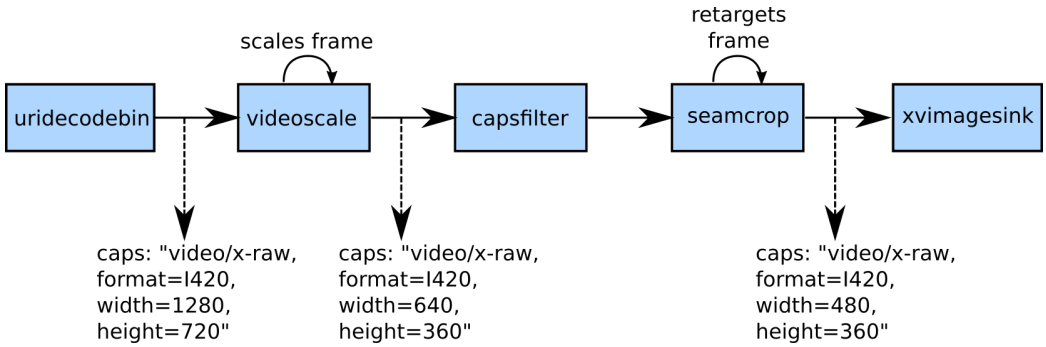


Figure 5.3: Scaling incoming frames prior to the seamcrop element.

```
seamcrop ! queue ! xvimagesink
```

At each predetermined resolution change time, we alter the caps property of the capsfilter element. This forces a new capability negotiation through the CAPS event, where the result is that the videoscale element scales the incoming frames to the resolution specified in the property we set. These events are identical to the CAPS events sent by the dashdemux element when the resolution of the source stream changes. It is important to note that this event does not alter the resolution of the actual stream received from the remote location, but changes each individual frame locally to adhere to the capabilities we require for our experiments. Figure 5.3 illustrates a snapshot of the pipeline with the videoscale and capsfilter elements.

Performing this resolution change through the CAPS event does not interfere with the pipeline events originating from the dashdemux element present in the uridecodebin. All events are forwarded downstream to our element, including upstream CAPS events that are rendered ineffectual by the capsfilter element. By doing it in this manner, we can ensure that a specific set of resolutions are passed to our element at predetermined points of operation, facilitating the comparability of the results while simultaneously demonstrating compatibility with the DASH technique within the GStreamer framework.

5.5.2 Reinitialization Latency

In this setup, we perform frequent input resolution changes to measure the reinitialization latency exhibited by the element. As each resolution change prompts a reinitialization of the retargeting environment, we measure this latency to determine the impact it delay it incurs on resolution changes. This latency is measured as the time elapsed between the transmission of the last frame of the previous resolution to the reception of the first frame for the new resolution.

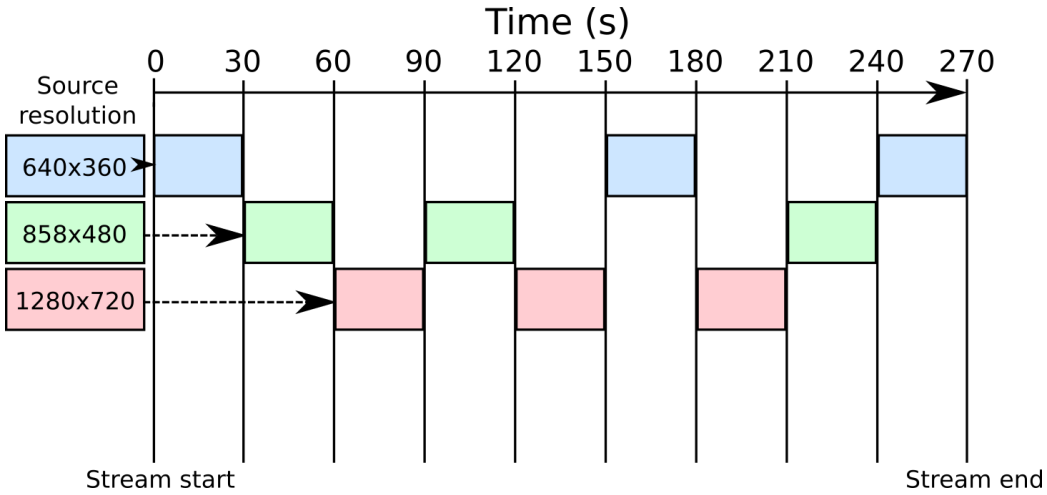


Figure 5.4: Resolution change flow of the experiments performed in 5.5.2.

The experiments are performed in the same manner as the in previous setup with the same evaluation factor variations. Figure 5.4 visualizes the sequence of these alterations. Each resolution segment is streamed for 30 seconds, where the order of the resolution changes varies over the course of the stream. The stream is run for a total of 270 seconds to alter between each resolution 3 times. Altering the resolution frequently also serves to examine the operation of our plugin for recurring dynamic pipeline changes.

5.5.3 Initial Latency

In this setup, we perform an initialization of each configuration 10 times to obtain the average initial latency exhibited by the individual evaluation factor configurations. This is performed to see what effect the evaluation factors have on the initialization time. Unlike the previous setups, each experiment initializes a new stream for each resolution. The measurements are presented in Section 5.3.4 .

In addition to this measurement, we also investigate the internal distribution of this latency to see how it can be attributed to each of the components involved. This is done to identify the main contributing factors to this latency. The initial latency stems from the cumulative time required to perform the following operations:

- receive and add frames to the retargeting module,
- perform the first pass on the frame window,
- compute and smooth cropping window,

- seam carve and send the first frame.

To determine the impact of the plugin component, wrapper and retargeting module, we measure the total time spent for each one in these experiments. For receiving and adding a frame, we utilize the time stamp measurements $t1$ and $t2$ as presented in Figure 5.1, adding together the individual time stamp differences. The first measurement of $t4 - t3$ is also included. This accounts for the time spent by the plugin component and the wrapper. For the remaining operations in the retargeting module, we measure the internal time from when the worker threads are started until the first frame has left the retargeting module. These measurements are averaged for all experiments performed for each configuration. Comparing these averages with the latency measurement yields the average latency distribution.

To determine how much of the processing time exhibited by the retargeting module can be attributed to buffering and frame allocation, we compare this initial retargeting average with the corresponding average without buffering for the same factor configurations, obtained from the experiments detailed in Section 5.5.1.

5.5.4 HLS Streaming

In this setup we perform real streaming scenarios similar to those described in Section 5.5.1, utilizing the HLS technique. The purpose of these experiments is to verify that our element is compatible with the HLS streaming technique for the same configurations we use to evaluate the DASH streaming scenario. As such, the pipeline employed for these experiments is identical to the one presented in the previous sections, barring the substitution of the $\mathrm{dashdemux}$ element for $\mathrm{hlsdemux}$.

Each experimental configuration is run for the same duration as their DASH equivalent. We perform measurements of latency and retargeting rate in order to compare performance with the DASH streaming experiments. We do not measure CPU and memory load in these experiments as the choice of streaming technique has no bearing on these measurements.

The resolution of the frames provided by the HLS stream is 1024x448 as we were unable to locate a HLS stream providing a resolution equal to 1280x720 that could be utilized for our experiments. However, as we scale the incoming frames to match the resolutions we require for the evaluation of our element, the results are comparable and should not be significantly dissimilar.

	Configuration								
Factor	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Frame window size	50	100	200	50	100	200	50	100	200
Retargeting factor	0.85	0.85	0.85	0.75	0.75	0.75	0.50	0.50	0.50

Table 5.3: Evaluation configurations used for the retargeting and frame window size factors.

5.6 Evaluation Results

This section presents the results of the experiments described in Section 5.5, performed on the testing node listed in Table 5.2. To perform the internal measurements, we utilize modified versions of *gstseamcrop.c*, *seamcropwrapper.cpp* and *seamcropcuda.cpp*, initializing the $\mathrm{GstSeamCrop}$ element with a new property, *measurement*. We have also implemented a configurable GStreamer application, *evaluationseamcrop.c*, which performs the resolution changes. These files can viewed in the Git repository referenced in Appendix A.

To simplify the presentation of the results, each configuration of frame window size and retargeting factor is attributed a number. The configurations and their corresponding number assignment can be viewed in Table 5.3. The following sections examine the results for each evaluation metric in turn.

5.6.1 Retargeting Rate

In this section, we investigate how varying the resolution, retargeting factor and frame window size affects the retargeting rate exhibited by the element.

DASH Experiments

Table 5.4 presents the results obtained from the experiments described in Section 5.5.1.

When examining the rates in the *Initial* row with the *Overall* row, performance during the first frame window is consistently lower than for subsequent frame windows in all configurations. This confirms the prediction we made in Section 3.3.12 that the retargeting rate for the first window is slower than the rest as the first pass needs to wait for new frames to arrive. This impacts the retargeting rate as the processing rate of the first pass is higher than the input rate.

We can see that both retargeting factor and frame window size impact the output rate of the element, where the retargeting severity has a directly adverse effect

	Configuration								
Resolution	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
640x360									
- *Initial*	47.31	49.94	48.33	33.57	31.72	27.64	18.84	18.88	18.67
- *Overall*	52.60	51.15	53.29	34.95	34.51	35.32	19.48	20.60	21.68
854x480									
- *Initial*	24.85	25.31	25.60	17.17	16.45	16.41	9.56	9.59	9.61
- *Overall*	29.74	29.37	31.17	19.79	19.85	21.02	10.69	11.71	13.01
1280x720									
- *Initial*	10.99	10.83	11.19	7.49	7.45	7.26	4.01	4.32	4.05
- *Overall*	13.31	13.92	15.26	8.87	9.15	10.83	5.07	5.81	7.60

Table 5.4: Average frames per second (FPS) generated by the element for each resolution and configuration in the DASH experiments. The *Initial* row shows the average rate for the first frame window while *Overall* denotes the average for all subsequent frame windows.

on the frame output production. This stems from the fact that a higher retargeting factor requires more processing due to an increase in the number of seams to remove. This effect is exacerbated further by increasing the frame resolution, where configurations with both the highest resolution and retargeting severity, i.e. 1280x720 resolution and 50% pixel removal, yield the poorest performance.

Contrary to the retargeting factor, an increase in the frame window size appears to have a beneficial impact on the retargeting rate. This is particularly evident when comparing the configurations with the highest frame window size, i.e., 3, 6 and 9, with their smallest counterparts, 1, 4 and 7. Every configuration with a frame window size of 200 unequivocally outperforms all configurations with a size of 50, regardless of retargeting factor or frame resolution.

Evidently, it appears that as the size of the computations increases, a higher frame window size results in a higher retargeting rate. This is explained by better overall utilization of the GPU and the parallelization property of the algorithm. When the frame window size is large, each pass of the algorithm can run for a longer time and thus process more frames in parallel, yielding a higher performance benefit. Additionally, longer passes equate to less overhead associated with performing the algorithm multiple times in succession, however miniscule.

This pattern is however not present for all configurations, as the performance of configurations with a low resolution and retargeting factor do not linearly increase alongside the frame window size. For resolutions of size 640x360, a frame window size of 50 fares better than a frame window size of 100 when the retargeting severity is 15%, and to a lesser extent, 25%. This pattern can also be seen

	Configuration								
Resolution	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
640x360	55.15	56.63	58.40	36.45	36.49	39.08	20.25	20.78	22.38
854x480	30.01	30.43	32.61	19.74	20.28	21.63	11.28	11.86	13.05
1280x720	13.17	13.79	15.01	8.82	9.34	10.55	5.08	5.64	6.71

Table 5.5: Average frames per second (FPS) generated by the element for each resolution and configuration during the HLS experiments.

for a resolution of 854x480, but only for the case with a retargeting percentage of 15%. The performance increase of utilizing the largest window size for these configurations is also less pronounced.

There are a few potential explanations for this behavior. As low resolutions with a lesser degree of width reduction exhibit less computational load, it is possible that the benefit of a higher frame window size does not manifest until the size exceeds a threshold after which more available frames equates to a higher retargeting rate. This is unlikely, but offers a probable explanation for this behavior.

Alternatively it can stem from inherent variance either as a result of inefficient computations for the frame window size of 100 or a volatile input rate. These options are more plausible, where fluctuations in bandwidth impacting the input rate offers the most credible explanation. Investigating the performance trade off between frame window size and computational load is interesting but is left for future work due to time constraints.

HLS Experiments

Table 5.5 shows the results obtained from the experiments detailed in Section 5.5.4.

The results of these experiments are similar to those obtained from their DASH equivalents. There are minor differences in the reported retargeting rates, but the majority of these fluctuations are not so significant as to indicate an appreciable performance difference between the streaming techniques. However, the results of the HLS experiments do exhibit a higher retargeting rate for most of the configurations, particularly for configurations 1, 2 & 3 with a resolution of 640x360. This pattern is most prevalent for smaller combinations of retargeting factor and resolution, diminishing as both increase. Since the retargeting rates for these 3 configurations are over twice as fast as the presentation rate in both setups, it is conceivable that the actual output rate exceeds the input rate for the DASH experiments. As the DASH/HLS experiments utilize different input streams, the reported disparities are likely to stem from the input rate differences.

When comparing the rates for the individual configurations, it is interesting to note that in these experiments, an increase in the frame window size uniformly yields a higher retargeting rate. This result corroborates the explanation that the input rate is the most contributing factor to the observed variance in the retargeting rates.

The experiments indicate that an increase in both retargeting factor and resolution results in a slower retargeting rate, while increasing the frame window size produces a faster retargeting rate overall. Across all experiments, the impact of the transitional smoothing step introduced to the algorithm never exceeds 0.02ms for any configuration. This added computational impact is so minute as to be insignificant to the overall retargeting rate of the algorithm. A frame window size of 200 appears to yield a performance increase between $\sim$0.5–3.25 frames per second for all configurations when compared with their resolution and retargeting factor equivalents that utilize a size of 50. This increase is also present between frame window sizes of 50 and 100, albeit less prominent.

Regardless of this performance increase, our element is not able to retarget at 25 frames per second for any configurations with a retargeting factor of 0.50 in either of the streaming setups. For a retargeting factor of 0.75, only the smallest resolution achieves acceptable output rates, but does so for all three frame window sizes. With the lowest retargeting factor, 0.85, the two smallest resolutions yield acceptable retargeting rates across all frame window sizes.

5.6.2 Latency

In this section, we examine how the evaluation factors impact the initial latency and reinitialization latency induced by the element.

Initial Latency

Table 5.6 shows the average initial latency incurred by each configuration, measured as described in Section 5.5.3.

From this table it is evident that as the frame window size increases, so does the initial latency. Between the configurations, the frame window size is doubled from either 50 to 100 or 100 to 200. The latency increase is most pronounced between configurations where the frame window size is increased by 100, i.e., 2-3, 5-6 and 8-9, which is to be expected as the amount of frames added is higher. This is particularly evident between configurations 5-6 as well as 8-9 for the 1280x720 resolution. However, while an increase in frame window size leads to higher overall initial latency, doubling the window size for a resolution does not linearly

	Configuration								
Resolution	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
640x360	222.6	514.6	1016.9	236.7	436.1	898.7	243.5	453.4	978.7
854x480	416.2	836.4	1545.6	493.4	818.0	1642.4	448.3	804.4	1404.8
1280x720	860.5	1222.4	2654.9	865.3	1419.8	3515.8	901.7	1435.9	3274.8

Table 5.6: Average initial latency in milliseconds induced by the element for each evaluation factor configuration. Each measurement is the elapsed time from reception of the first frame to its transmission during the first frame window.

	Configuration								
Resolution	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
640x360									
- *DASH*	180	292	633	172	366	1657	211	383	1027
- *HLS*	181	319	952	182	598	1123	885	361	987
854x480									
- *DASH*	398	841	1345	470	966	1879	672	1177	2293
- *HLS*	288	736	1806	353	794	1585	507	1619	1099
1280x720									
- *DASH*	913	1934	2891	939	1427	3541	1360	1396	4021
- *HLS*	891	1369	2731	621	1787	3550	743	1746	4333

Table 5.7: Initial latency in milliseconds induced by the element for each evaluation factor configuration during the HLS and DASH experiments.

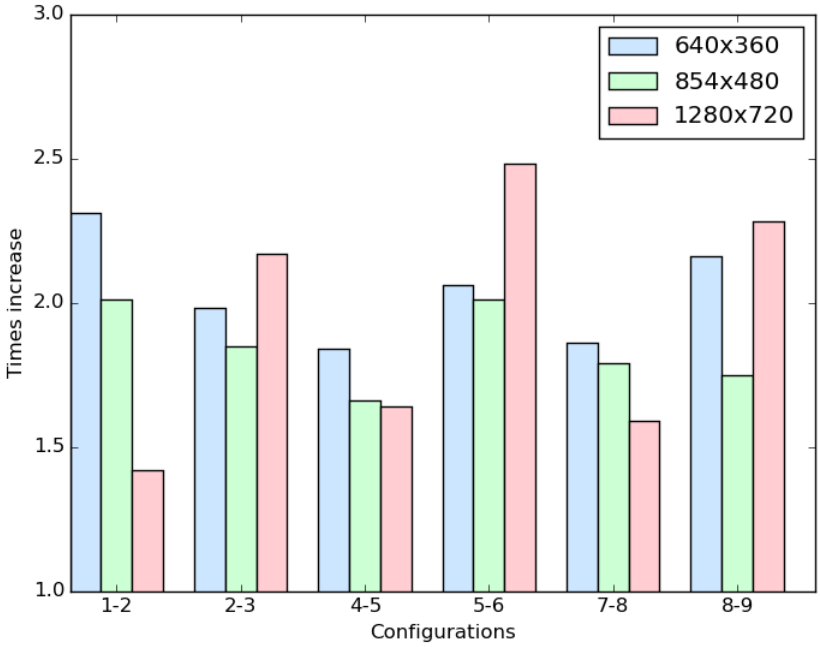


Figure 5.5: Relative latency increase between configurations with different frame window sizes, based on average initial latency.

lead to a doubling of the initial latency. The pattern roughly correlates to a ∼1.5–2.5 times increase in the initial delay for each doubling of the frame window size, as shown in Figure 5.5. It is likely that this variance is a side effect of a volatile input rate.

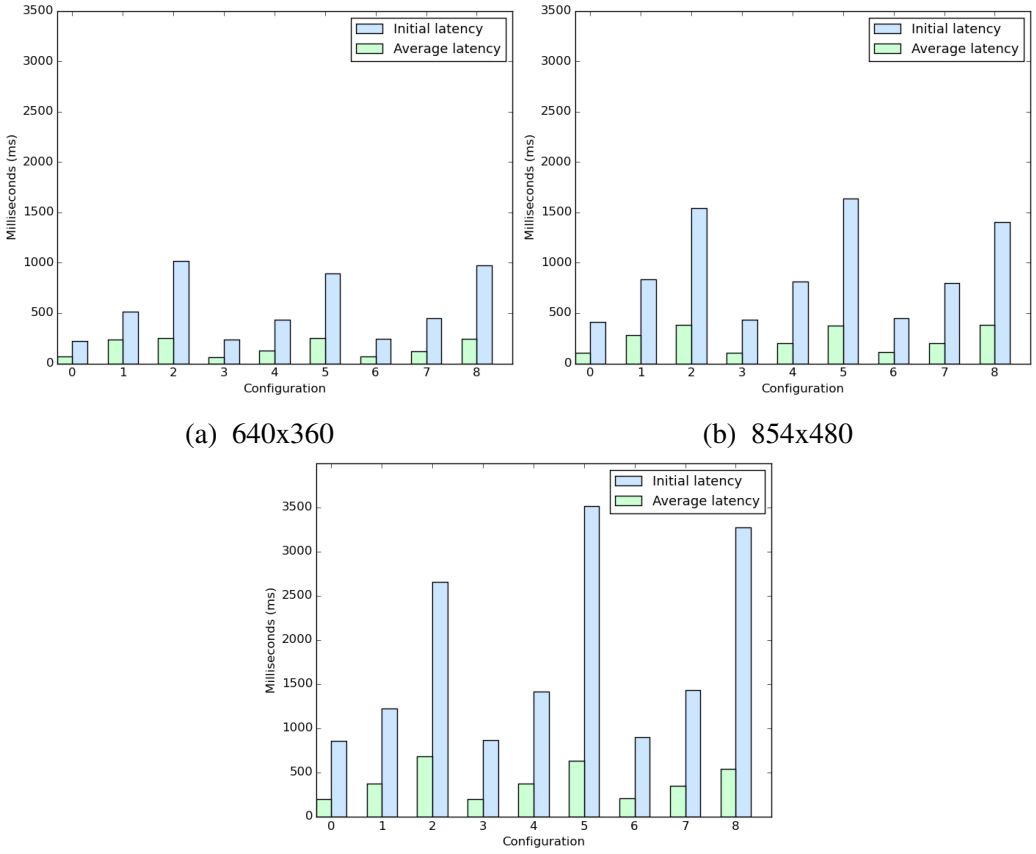
An overall increase in latency also present when increasing the resolution within each configuration. When comparing the results of the different resolutions within a configuration, the relative latency difference is similar to that of increasing the frame window size.

Interestingly, among configurations with identical retargeting factor, a lower resolution with a large frame window size performs comparably to a higher resolution with a smaller window size in the previous configuration. The latency difference among these cases can be as low as 5 ms or as high as ∼300 ms, but mostly range between 30–150 ms. While not identical, the measurements suggest that increasing either the resolution or the frame window size results in a similar increase in the overall initial latency.

Table 5.7 presents the initial latencies reported by the DASH and HLS experiments. When compared with the averages presented in Table 5.6, there is a great deal of variability as they are single measurements as opposed to averages, but for most configurations they are within expected ranges.

Initial Latency Distribution

Figure 5.6 shows comparisons of the average processing time for the first pass, cropping window computation/smoothing and seam carving one frame with the average initial retargeting time exhibited by the retargeting module. From these figures, it becomes apparent that the vast majority of the time elapsed is spent waiting for new frames to arrive.



(a) 640x360

(b) 854x480

(c) 1280x720

Figure 5.6: Average retargeting latencies for producing the first frame of a window. Shows both the average latency overall (green) and initial latency (blue).

Apart from the considerable time differences between the separate resolutions, the ratio between the average time spent and the initial time spent is quite consistent for each retargeting factor configuration. This is because the retargeting factor does not significantly impact this measurement. It is most relevant during the seam carving portion of the algorithm, which is only performed once for each individual measurement of the initial latency. In every measurement, the impact

	Frame window size		
Resolution	**50**	**100**	**200**
640x360	70.83%	75.43%	77.83%
854x480	65.54%	72.15%	72.97%
1280x720	73.80%	74.85%	80.06%

Table 5.8: Average percent of the initial latency spent waiting for incoming frames. This percentage is obtained by comparing the average retargeting time in the absence of buffering and frame allocation with the retargeting time spent during the initial latency.

of adding and sending a frame, i.e., the total effective time spent by the plugin component, is consistently <1% of the total initial latency. This impact is negligible, further indicating that the input rate and the frame window size are the main contributing factors to the initial latency exhibited by the element.

Table 5.8 shows the average percentage time spent buffering and allocating frames during the initial latency for each resolution and frame window size. It is interesting to note that for the 854x480 resolution, the buffering impact for each frame window size configuration is lower than for the other resolutions. We do not have a clear explanation for why this is occurs, as variance in the most prominent external factor, bandwidth, would result in worse performance.

Reinitialization Latency

As the retargeting environment is required to be reinitialized on each resolution change, it invariably introduces some latency in addition to the initial latency exhibited by the element. Figure 5.7 presents the average reinitialization time for each resolution. Each of these measurements are calculated as the time difference between the transmission of the last frame from the previous resolution setup to the reception of the first frame in the next. Since the frame window size did not exert any apparent effect on the reinitialization time, the average is calculated from all measurements for each resolution.

The reinitialization latency is quite small for all resolutions, increasing in time taken along with an increase in the resolution. This increase is explained by the need to allocate more space for larger resolutions. While the apparent latency induced by the reinitialization is not significant, it can have an appreciable impact depending on the new configuration. Consider the first configuration for a resolution of 640x360 presented in Table 5.6. In this case, since the initial latency is quite small, the reinitialization latency does add $1/10$ additional time to the

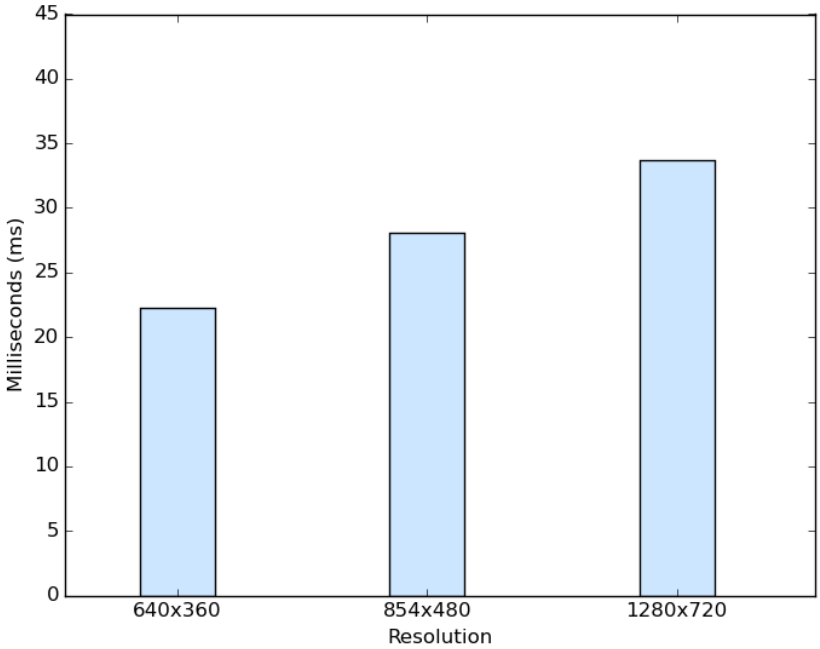


Figure 5.7: Average reinitialization time per resolution.

overall latency. However as the resolution or retargeting severity increases, the proportional impact of reinitialization decreases significantly.

5.6.3 CPU load

Figure 5.8 shows the CPU load of the application running the pipeline for the separate configurations.

Section 5.3.1 describes that a multithreaded application is able to utilize >100% of the total CPU resources due to the presence of multiple cores, where each core can be utilized 100%. This is evident in the results. We observe that the CPU usage of the application is fairly consistent, ranging between 496% at the least and 529% at the most. The application running the pipeline consistently utilizes the total capacity of 5 cores or more, with the exception of configuration 8 for resolution 1280x720 which uses less than the sum resource total of 5 cores. Whether each core is used to their total capacity is not known as top calculates this percentage from the individual utilization of each core. Distributions such as either 100% utilization of 5 cores or 65% of 8 cores or similar are possible.

The CPU of our testing node contains 8 cores, which corresponds to a maximum usage of 800%. Considering this, the application consumed between 62-66.125% of the total CPU resources when running the experiment pipelines. This

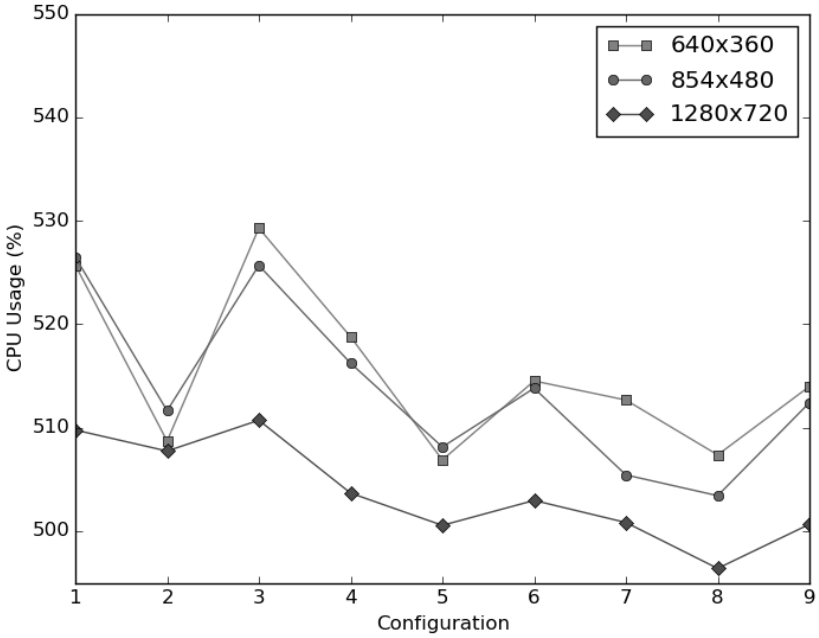


Figure 5.8: Average percentage (%) of total CPU resources utilized by each resolution for each configuration.

resource consumption is explained by the use of multiple threads in our element, 6 of which are explicitly dispatched.

Since four threads are dispatched to perform retargeting during each of the two passes of the algorithm, it is very likely that these threads account for most of the resources consumed. Each of these threads can utilize up to 100% of one core, with a combined resource consumption of up to 400% in total. The retargeting process performed by each thread is very CPU intensive, as each frame is copied between main memory and the GPU once per pass. Due to this, it is reasonable to assume that these threads frequently utilize one core each to its fullest during their operation.

For the rest of the application, it is likely that the remaining CPU utilization stems from the thread dispatching outgoing frames and the thread receiving incoming frames. Each received frame is copied from its GStreamer buffer into the memory allocated in the retargeting module environment, which is an expensive operation in terms of processing.

As for the impact of the evaluation factors, we can observe that as the degree of retargeting increases, the overall CPU usage for each frame window size configuration tends to decrease. This can be explained by less efficient parallelization during the second pass of the algorithm when the retargeting severity is high, since each thread spends more time waiting for seams to be computed on the GPU. Each

thread yields while waiting for the required seam to be identified which results in less overall use of the CPU.

From this, we would expect that the CPU utilization when increasing the frame window size within the retargeting factor configurations would also gradually decrease. However, the results defy this expectation. The CPU load exhibited by configurations with a frame window size of 100 is significantly lower than their counterparts that utilize the same retargeting factor, regardless of video resolution. The load differences between frame window sizes of 50 and 200 also exert a degree of variability, either increasing or decreasing, but both are consistently higher than for a frame window size of 100.

We do not know the cause of this behavior. This may be a result of an external factor influencing the element, but this is unlikely as the pattern appears to be consistent across all of the experiments. Alternatively it is possible that it is a result of inefficient threading in our implementation. This offers an explanation for the apparent decrease in performance observed for configurations 2 & 5 with a resolution of 640x360 in Table 5.4. However, as the pattern persists for the higher resolutions and retargeting factors that do not exhibit this behavior as well, it is difficult to identify the cause. Investigating the cause for this performance discrepancy is left for future work.

To ascertain how much of the CPU load can be attributed to our element, we performed three additional experiments of the CPU load on the pipeline presented in Listing 5.1, removing our element. Each experiment scaled the incoming stream to the different resolutions as done in the rest of our experiments. The resulting average is a CPU utilization of 19.15%, meaning that our element on average accounts for $\sim$96% of the total CPU resources consumed by the application.

5.6.4 Memory Load

Figure 5.9 presents the average memory load exhibited by the application for each configuration.

In this figure we can see that there is a clear relationship between the frame window size, resolution and the induced memory load, where an increase in either results in higher memory load. This is as expected, since both storing more frames as well as frames of higher resolution requires more storage space. The amount of resources required for each resolution escalates proportionally along with the frame window size.

While varying the retargeting factor does not have a significant impact on these measurements, it does appear to result in decreased average memory usage when increased. This decrease is exaggerated further when the frame window size is increased along with it. To illustrate, the difference in average memory

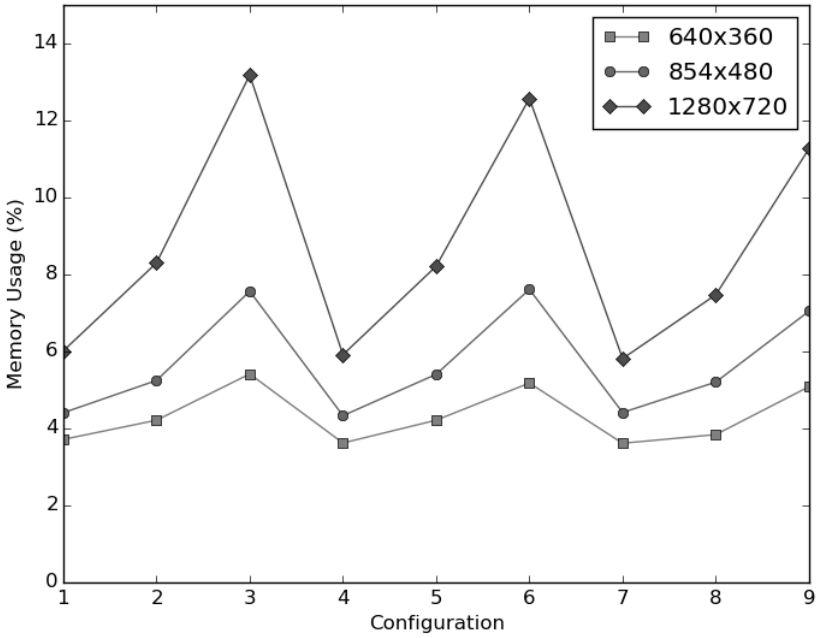


Figure 5.9: Average percentage (%) of physical memory utilized by each resolution for each configuration.

load between configurations 9 & 3 for a resolution of 1280x720 is 1.9%. This pattern is also present for the smaller resolutions, but to a lesser degree.

This is explained by the decreased size of the frames produced by the element. In configuration 9 with a resolution of 1280x720, an individual raw YUV420P frame is 1.3824 MB in size prior to retargeting. The size reduction of the retargeting is directly correlated with the retargeting factor. When retargeting with a severity of 50%, the size of each retargeted frame is effectively halved, reducing this size to 0.6912 MB. Compared with the produced frame size of 1.17504 MB for configuration 3, which has a retargeting severity of 15%, the difference is clearly visible. The memory required to store the incoming frames is identical, but the memory required to store generated output frames is decreased by 35%, explaining the apparent reduction in memory load.

This pattern is prevalent between all retargeting factor configurations regardless of resolution or frame window size, but is less pronounced for lower resolutions with low frame window sizes. This behavior, coupled with the fact that less output frames are concurrently present in configurations with a higher retargeting factor due to a slower retargeting process, results in less overall memory usage.

5.7 Discussion

In this section, we demonstrate that the results presented in Section 5.6 meet the requirements of the evaluation goals presented in Section 5.1. We also discuss the usefulness and limitations of our implementation. Each evaluation goal is examined to demonstrate that we have fulfilled it.

Our first goal is to demonstrate that $\mathrm{GstSeamCrop}$ can be used to retarget a video stream with a resolution of at least 640x360, a frame rate of 25 frames per second and a retargeting factor of 0.75 in real-time. Section 5.6.1 shows that with a retargeting factor of 0.75, our element is able to retarget incoming video frames with a resolution of 640x360 at a rate of $\sim$35 frames per second. This satisfies the minimum requirement of 25 frames per second for this configuration, surpassing the requirement with 10 additional frames retargeted per second. As the performance exceeds this requirement, it is possible to satisfy the presentation rate of 25 frames per second for even greater resolutions or retargeting factors, but it is restricted to retargeting severities below 25% should the resolution exceed 854x480. The impact of the additional smoothing step we introduce to the algorithm is negligible, never exceeding a processing time beyond 0.02ms for any configuration.

Our second goal is that the initial latency induced by our element is required to be less than 2000ms. Section 5.6.2 shows that the average initial latency for most of the configurations incur acceptable delays. Only configurations with a frame window size of 200 and a resolution of 1280x720 exceed this threshold on average, whose retargeting rates are not otherwise satisfactory. The evaluation factor configurations which satisfy the previous goal are guaranteed to satisfy the initial latency requirement as long as the frame window size does not exceed 200. For a resolution of 640x360 with a frame window size of either 50, 100 or 200, the average initial latencies are within acceptable margins, ranging between 236-898ms. As these values are well below the threshold, higher frame window sizes are also likely to yield acceptable latencies and possibly higher retargeting rates.

Our third goal is to investigate the CPU load of our element and verify that the memory consumption is consistent across differing configurations. Section 5.6.3 shows that our element utilizes between $\sim$62-66% of the total resources available on the 8-core CPU of the testing node, depending on the evaluation factor configuration. On an 8-core CPU, 6 cores operating at full capacity corresponds to 75% of the total processor resources. The load induced by our element does not surpass this threshold.

Section 5.6.4 shows that the memory load induced by our element is consistent between the differing configurations. The experiments yielding these results were run for a considerable duration, during which no unexplanatory increases in memory consumption emerged.

	Configuration		Results	
Resolution	**Retargeting factor**	**Frame window size**	**Initial latency**	**Output rate**
640x360	0.56	300	1797ms	25.01 fps
854x480	0.79	280	1834ms	25.17 fps
1280x720	0.94	190	1975ms	25.16 fps

Table 5.9: Highest evaluation factor configurations that yield a retargeting rate of at least 25 frames per second that do not exceed an initial latency of 2000ms.

Our last goal is to establish that our plugin is compatible with the DASH/HLS streaming techniques. The experiments described in Sections 5.5.1–5.5.3 have been performed using the DASH technique through use of the dashdemux element. Section 5.5.4 performed streaming experiments utilizing the HLS streaming technique through the hlsdemux element. Results of both experimental setups yielded similar results. Our element properly adapted to pipeline events originating from either of the elements during these experiments and no issues or incompatibilities surfaced during the evaluation. We consider these results sufficient to satisfy the goal.

5.7.1 Optimal Factor Configurations

In the previous paragraphs, we demonstrated that our plugin is usable for real-time retargeting of video content provided by an adaptive streaming technique, with respect to the requirements presented in Section 5.1. In this section, we present the optimal factor configurations that can be achieved for the three separate resolutions on our testing node and discuss the viability of our element for these resolutions. Table 5.9 presents the highest retargeting severity we were able to achieve for each of the resolutions that satisfy a presentation rate of 25 frames per second and an initial latency below 2000ms.

In these additional experiments we attempt to identify an optimal trade off between frame window size, retargeting factor and resolution that satisfies Goal 1 and Goal 2. As described in Section 5.6, a larger frame window size is generally correlated with a higher retargeting rate at the cost of increased initial latency. Since most configurations in our experiments yield an initial latency far below the requirement, we increase the frame window size until the initial latency threshold is reached and adjust the retargeting factor until the output rate hovers just above 25 frames per second.

From this table we can see that our element provides ample adaptation for frames with a resolution of 640x360, able to reduce the width by a maximum of

44%. This amount is quite considerable, and in the majority of cases a retargeting factor in this range is adequate to provide a presentation in the desired aspect ratio. However, the usefulness diminishes as the resolution increases. For a resolution of 854x480, the reduction is limited to a maximum of 21%. This is less universally viable, but nonetheless useful for lesser adaptations. For the highest resolution, only a 6% reduction in width is feasible to provide real-time retargeting. This amount is so small as to be relatively imperceptible when compared with an unaltered presentation, making the element considerably less useful for resolutions of this size.

The reason for this rapid decrease in viability for higher resolutions is a combination of an increase in the amount of pixels to inspect per frame and the amount of seams to remove. From lowest to highest resolution, each frame consists of either 230400, 409920 or 921600 pixels. Between 640x360 and 854x480, there are approximately 1.78 times more pixels to process per frame, and from 854x480 to 1280x720 there are approximately 2.25 times more. A frame with a resolution of 1280x720 has in total 4 times as many pixels as a frame of 640x360.

To illustrate further, a frame with a width of 1280 pixels needs to remove 320 seams with a retargeting factor of 0.75. Compared to a frame with a width of 640 pixels, which only has to remove 160 seams for the same retargeting factor, it has to remove double the seams while also inspecting 4 times as many pixels. As a result, it takes a considerable amount of time to find and remove seams from frames with a high resolution for the same retargeting factor.

Consequently, the usefulness of our plugin depends on the degree of retargeting required and the resolution of the video. If considerable adaptation is required, the resolution must be small. In the opposite case, the resolution requirement subsides, but the benefit of the retargeting also diminishes.

5.7.2 Limitations

As we have investigated in this chapter, our element is able to provide satisfactory retargeting rates for a small variety of configurations, constrained by the resolution of the incoming frames and the degree of retargeting. Providing ample retargeting in real-time for resolutions higher than 854x480 is not feasible on the hardware of our testing node. Additionally, each resolution change incurs both a reinitialization latency as well as the initial latency for that particular evaluation factor configuration. This added latency is detrimental to the viewing experience, especially when streaming under variable bandwidth conditions. These are the two primary limitations of our implementation.

An immediate solution to deal with both limitations is to enforce a specific resolution in the pipeline prior to our element that is guaranteed to yield a retargeting rate that is greater than or equal to the presentation rate for the chosen retarget-

ing factor. While the scaled frames will not necessarily match the resolution of the incoming stream, it does ensure that our element is able to retarget efficiently. This can be done in the same manner as in our experiments by including both a videoscale and a capsfilter element in the pipeline.

The benefit of this solution is that changes in resolution as a result of bandwidth fluctuations do not incur a reinitialization of the retargeting environment. The drawback is that it requires the programmer to have prior knowledge of the plugin and hardware limitations as well as rely on additional elements in the pipeline.

A permanent solution would be to extend the the plugin to automatically choose a suitable resolution that results in an acceptable output rate, which would overcome both limitations. This is however left for future work.

Chapter 6

Conclusion

In this chapter, we conclude our work and discuss the product of this master thesis. Section 6.1 presents our contributions. In Section 6.2, we discuss improvements that can be explored in future work.

6.1 Contributions

We have designed and implemented a video retargeting system that provides online retargeting of video in real-time. The main aim of our work was to realize advanced video retargeting for video on demand and live streaming content within a widely available multimedia framework for use in a ubiquitous computing paradigm. We have analysed two state-of-the-art offline retargeting techniques and picked the one best suited for our system based on the requirements presented in Section 1.3. We have chosen the *Parallelized SeamCrop* algorithm as the retargeting technique for adapting video frames and wrapped its functionality in a plugin for the GStreamer multimedia framework.

Our principal contribution is the GStreamer plugin implementation of the *Parallelized SeamCrop* algorithm which can be used to adapt an incoming video stream to match the screen aspect ratio of a given device in real-time. The GStreamer plugin design allows the element to be inserted in arbitrary video streaming pipelines to provide customizable video resolution adaptation. By implementing the retargeting system as a part of the GStreamer framework, any video format that has a decoding element available in the framework can be adapted while also making the system available for a large community of users. As the design of our plugin is modular, future retargeting algorithms can easily be implemented in the GStreamer framework with our plugin code by replacing the modules concerned with the specific retargeting algorithm.

We have presented an alteration of the *Parallelized SeamCrop* algorithm which

allows it to be used for online retargeting. This alteration separates retargeting of a video into smaller segments, smoothing the differences between them with a virtual camera pan, allowing the video to be presented and retargeted concurrently once the first segment has been processed. Additionally, the memory requirement of the algorithm is also reduced, utilizing a fixed amount of memory as opposed to linearly increasing along with the video length.

We have implemented this design and evaluated it in Chapter 5. In this evaluation, we have demonstrated that the plugin can be used for real-time video adaptation with two modern streaming techniques, but is limited to resolutions below 854x480 should extensive adaptation be required. This limitation however depends on the underlying hardware, where hardware more powerful than our testing node is likely to achieve better results. Since the evaluation has been performed in a real streaming environment, the results are applicable to real world scenarios.

6.2 Future Work

We separate potential future work into two categories. The first category relates to the implementation of the retargeting plugin itself. The second category is concerned with the *Parallelized SeamCrop* algorithm.

6.2.1 SeamCrop Plugin

Our plugin implementation is fairly basic, and there are multiple ways in which the functionality could be extended or improved upon. In Section 5.7.2 we mentioned a few approaches that would be beneficial. As the plugin is restricted to certain resolutions for real-time retargeting, extending the plugin to automatically choose an appropriate resolution based on its limitations would solve several issues. This would simplify the usability of the plugin by removing its dependence on other GStreamer elements as well as guaranteeing real-time retargeting for any video stream. Additionally, this would prevent reinitialization of the retargeting environment as a result of resolution changes, eliminating the associated latency.

Plugin performance could be improved by reducing the amount of both memory copy operations and format conversions. Each frame that arrives is converted from *YUV420P* to *RGB*, copied, retargeted, copied and subsequently converted again. Identifying a solution which eliminates one or both of these necessities would improve the performance of the plugin overall.

Other functionality that would improve usability would be to add properties for either aspect ratio or output resolution to the element. Retargeting in the current implementation is performed based on a retargeting factor specified at initial-

ization. This factor determines the output resolution based on the input resolution, performing a percentage reduction in width. Supplementing this option by allowing direct assignment of either aspect ratio or output resolution would simplify the use of the plugin.

6.2.2 Parallelized SeamCrop

Performing a qualitative comparison of output from the altered and the original algorithm would be interesting to ascertain the impact of our alteration on the produced video. As the alteration of the algorithm was a secondary concern in this thesis, we do not explore the disparities between them. Investigating the qualitative difference introduced by separating the global optimization into bounded windows would be beneficial to determine its effect on the algorithm's efficiency with regards to output quality.

The transitional smoothing we perform between the separate frame windows can also be improved. Our approach performs a gradual transition which statically pans the virtual camera between the frame windows. This movement can at times seem unnatural, especially when the distance between the cropping window views is considerable. Improving this transition by introducing a more dynamic camera pan based on the pixel distance and frame window size would be beneficial to make the perceived camera movement more natural.

Exploring other approaches to separating the video could potentially yield worthwhile improvements. Segmenting the video based on scene detection rather than a predetermined window size could omit the impact of segmenting the global optimization and reduce camera movement. This could be achieved by analyzing the energy maps of two adjacent frames computed during the first pass of the algorithm. As compositionally differing scenes do not require temporal consistency between seams or cropping windows when transitioning, retargeting each individual scene separately could feasibly result in higher quality output. Optimizing the cropping window path for individual scene compositions would yield the optimal path for each particular scene rather than a weighted compromise between several. Maintaining an upper bound to scene length would counteract scenes that would otherwise either introduce significant mid-stream latency or consume too much memory.

Bibliography

[1] Yu-Shuen Wang, Jen-Hung Hsiao, Olga Sorkine, and Tong-Yee Lee. Scalable and coherent video resizing with per-frame optimization. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 88:1–88:8, New York, NY, USA, 2011. ACM.

[2] Johannes Kiess, Daniel Gritzner, Benjamin Guthier, Stephan Kopf, and Wolfgang Effelsberg. Gpu video retargeting with parallelized seamcrop. In *Proceedings of the 5th ACM Multimedia Systems Conference*, MMSys '14, pages 139–147, New York, NY, USA, 2014. ACM.

[3] Johannes Kiess, Benjamin Guthier, Stephan Kopf, and Wolfgang Effelsberg. Seamcrop: Changing the size and aspect ratio of videos. In *Proceedings of the 4th Workshop on Mobile Video*, MoVid '12, pages 13–18, New York, NY, USA, 2012. ACM.

[4] Basic tutorial 3: Dynamic pipelines. `http://docs.gstreamer.com/display/GstSDK/Basic+tutorial+3%3A+Dynamic+pipelines`. [Accessed 27-May-2016].

[5] Gstreamer - communication. `https://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/section-intro-basics-communication.html`. [Accessed 28-July-2016].

[6] Thomas Stockhammer. Dynamic adaptive streaming over http – standards and design principles. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pages 133–144, New York, NY, USA, 2011. ACM.

[7] Http live streaming overview. `https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/Introduction/Introduction.html`. [Accessed 9-May-2016].

[8] Richard M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92, Volume 1 - Volume I*, pages 416–429, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[9] Stephan Kopf, Thomas Haenselmann, Jonathan Kiess, Benjamin Guthier, and Wolfgang Effelsberg. Algorithms for video retargeting. In *Multimedia Tools Applications*, volume 51, pages 819–861, 2011.

[10] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 211–224, New York, NY, USA, 2012. ACM.

[11] Yongwei Nie, Qing Zhang, Renfang Wang, and Chunxia Xiao. Video retargeting combining warping and summarizing optimization. *The Visual Computer*, 29(6):785–794, 2013.

[12] Stephan Kopf. *Algorithms for Image and Video Processing*. Habilitation, Faculty of Business Informatics and Mathematics, University of Mannheim, Germany, 2011.

[13] Michael Rubinstein, Diego Gutierrez, Olga Sorkine, and Ariel Shamir. A comparative study of image retargeting. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 160:1–160:10, New York, NY, USA, 2010. ACM.

[14] Philipp Krähenbühl, Manuel Lang, Alexander Hornung, and Markus Gross. A system for retargeting of streaming video. *ACM Trans. Graph.*, 28(5):126:1–126:10, December 2009.

[15] Yu-Shuen Wang, Hongbo Fu, Olga Sorkine, Tong-Yee Lee, and Hans-Peter Seidel. Motion-aware temporal coherence for video resizing. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 127:1–127:10, New York, NY, USA, 2009. ACM.

[16] Pierre Greisen, Manuel Lang, Simon Heinzle, and Aljosa Smolic. Algorithm and vlsi architecture for real-time 1080p60 video retargeting. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 57–66, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[17] Yu-Shuen Wang, Chiew-Lan Tai, Olga Sorkine, and Tong-Yee Lee. Optimized scale-and-stretch for image resizing. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 118:1–118:8, New York, NY, USA, 2008. ACM.

[18] Yu-Shuen Wang, Hui-Chih Lin, Olga Sorkine, and Tong-Yee Lee. Motion-based video retargeting with optimized crop-and-warp. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 90:1–90:9, New York, NY, USA, 2010. ACM.

[19] Md. Haidar Sharif, Jean Martinet, and Chabane Djeraba. *Encyclopedia of Multimedia*, chapter Motion Saliency, pages 442–444. Springer US, Boston, MA, 2008.

[20] David Jacobs. Image gradients. *Class Notes for CMSC*, 426, 2005.

[21] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. *ACM Trans. Graph.*, 26(3), July 2007.

[22] Dynamic adaptive streaming over http. `http://mpeg.chiariglione.org/standards/mpeg-dash`. [Accessed 28-July-2016].

[23] Http live streaming internet-draft. `https://tools.ietf.org/html/draft-pantos-http-live-streaming-19`. [Accessed 8-May-2016].

[24] Christopher Müller and Christian Timmerer. A vlc media player plugin enabling dynamic adaptive streaming over http. In *Proceedings of the 19th ACM International Conference on Multimedia*, MM '11, pages 723–726, New York, NY, USA, 2011. ACM.

[25] Adobe http dynamic streaming. `http://www.adobe.com/products/hds-dynamic-streaming.html`. [Accessed 28-July-2016].

[26] Microsoft smooth streaming. `http://www.iis.net/downloads/microsoft/smooth-streaming`. [Accessed 28-July-2016].

[27] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, 1995.

[28] Michal Podpora, Grzegorz Paweł Korbas, and Aleksandra Kawala-Janik. Yuv vs rgb–choosing a color space for human-machine interaction. In *Position Papers of the 2014 Federated Conference on Computer Science and Information Systems*, page 29. Citeseer, 2014.

[29] Gstreamer base and utility classes. https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-libs/html/gstreamer-base.html. [Accessed 18-March-2016].

[30] Gstreamer - gstpad reference. https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/GstPad.html#GstFlowReturn. [Accessed 28-March-2016].

[31] Ffmpeg. https://ffmpeg.org/. [Accessed 29-July-2016].

[32] Opencv iplimage struct reference. http://docs.opencv.org/trunk/d6/d5b/structIplImage.html. [Accessed 27-March-2016].

[33] Nvidia cuda library reference: cudastreamsynchronize. https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/group__CUDART__STREAM_geb3b2f88b7c1cff8b67a998a3a41c179.html. [Accessed 28-March-2016].

[34] Gstbasetransform - base class for simple transform filters. https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-libs/html/gstreamer-base.html. [Accessed 2-April-2016].

[35] Ffmpeg: Avframe struct reference. https://ffmpeg.org/doxygen/2.7/structAVFrame.html. [Accessed 1-April-2016].

[36] Ffmpeg: Image related. https://ffmpeg.org/doxygen/2.4/group__lavu__picture.html#ga5b6ead346a70342ae8a303c16d2b3629. [Accessed 1-April-2016].

[37] Ffmpeg: Libswscale. https://ffmpeg.org/doxygen/2.1/group__lsws.html#gae531c9754c9205d90ad6800015046d74. [Accessed 1-April-2016].

[38] Boost: Thread management. http://www.boost.org/doc/libs/1_55_0/doc/html/thread/thread_management.html. [Accessed 1-April-2016].

[39] Boost: Shared pointers. http://www.boost.org/doc/libs/1_59_0/libs/smart_ptr/shared_ptr.htm. [Accessed 2-April-2016].

[40] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[41] The linux man-pages project: Top(1) man page. http://linux.die.net/man/1/top. [Accessed 5-April-2016].

[42] Michael Rubinstein, Ariel Shamir, and Shai Avidan. Multi-operator media retargeting. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 23:1–23:11, New York, NY, USA, 2009. ACM.

Appendix A

Source Code

The source code of the GstSeamCrop plugin can be downloaded from the Git repository at http://github.com/HaakonRav/GstSeamCrop. This repository includes the GstSeamCrop plugin, measurement applications, experiment results and the original source code of the *Parallelized SeamCrop* algorithm. The source code is released under the GNU General Public License 3.

The measurement applications are included to allow to facilitate effortless replication of the experiments with the same metrics and factor configurations. This permits comparing other results with the results presented in this thesis.

The repository includes a README file which gives instructions on compiling and running the plugin as a part of a GStreamer pipeline