Extensibility, Modularity, and Quality-Adaptive Streaming towards Collaborative Video Authoring

by

Jean-Sébastien Légaré

B. Sc. Computer Science, McGill University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

March 2010

Abstract

Video capture devices and online video viewing sites are proliferating. Content can be produced more easily than ever but the tasks required to compose and produce interesting videos are still very involving. Unfortunately, very little support is given for groups of amateurs to meet and collaborate to creation of new media. Existing video sharing sites do have some support for collaboration, but their best-effort mode of content delivery makes it impossible to support many of the desirable features usually available in local editors, such as advanced navigation support and fast-startup.

Quality-adaptive streaming is interesting as it allows content to be distributed, and allows clients of varying capabilities to view the same encoded video source, the so-called "Encode once, stream anywhere". This becomes even more important as the gap between low-end and high-end devices widens. In previous work we presented a Quality Adaptive Streaming system called QStream which has none of these limitations, but lacks the editing features. There are several media frameworks on the desktop that can provide the modules and pipelines necessary to build an editor, but they too are non-adaptive, and have multiple incompatibilities with QStream. This thesis presents *Qinematic*, a content creation framework that is quality-adaptive, and that borrows concepts from popular media frameworks for extensibility and modularity.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Glossary

This is the list of acronyms that are used throughout the thesis, sorted alphabetically by their short form. The full text of the acronym is only printed next to the very first occurrence of the acronym in the text.

API	Application Programming Interface
DOM	Document Object Model
GUI	Graphical User Interface. In the case of Qinematic and Qvid, the GUI consists of the user controls for playback of video and the video area.
JSON	Javascript Object Notation, a lightweight data-interchange format, easy for humans to read and write, easy for machines to parse and generate, and based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 (see http://www.json.org)
MF	Microsoft Media Foundation
NPAPI	Netscape Plugin Application Programming Interface
PTS	Presentation TimeStamp, the time at which a video image is to be presented on the screen.
SAX	Simple API for XML, SAX provides a mechanism for reading data from an XML document. It is a popular alternative to the Document Object Model (DOM). SAX Parsers functions as stream parsers that receive

	events that are fired when various features are encountered in the document (attributes, element start tag, element end tag, etc.).
SNR	Signal-to-Noise ratio, a measure of how clear a signal is received. SNR scalability lets you vary how much data of the original signal is lost.
XFADE	Cross-Fade, a transition from one audio or video source to another. Also called a dissolve.
XML	Extensible Markup Language

Acknowledgements

First and foremost, I am infinitely grateful to my supervisor, Buck Krasic, who has guided me wisely through my research. This project would not have been possible without all of his research on quality-adaptive streaming and his investments in the Qvid project. I thank my second reader also, Norm Hutchinson, for his constructive comments and feedback. I also want to give my gratitude to Noémie and my family, for their loving moral support and patience. Thanks to my many good friends from the department for their help, and finally to all others who have provided feedback on my work, and encouraged me (in no particular order): thank you Aiman, Andy, Brendan, Debo, Kan, Karol, Mayukh, and Patrick. Last but not least, I want to thank NSERC and UBC Graduate studies for their financial support.

Chapter 1

Introduction

The amount and popularity of user generated video on the Internet is exploding. For instance, in January 2009 alone, it has been estimated that users in the United States have viewed over 14.8 billion videos online [3], a 4% increase from the number of views the month before. Responsible factors undoubtedly include the increasing ubiquity of devices capable of video capture and manipulation, such as personal computers, digital cameras, camcorders and small phones, as well as the proliferation of Internet sites to aid in distribution and sharing of these videos such as YouTube[1], MySpace, etc..

We believe commoditization of video tools and the rise of the Internet as a distribution vehicle bears striking similarity to evolution of open source software over the years. As personal computers became commoditized, and Internet access commonplace, amateur software developers increasingly adopted methods and tools to collaboratively develop software projects, and as a result these developers could participate in projects of greater ambition and quality than they could hope to achieve on their own. With respect to user generated video, devices and tools for video generation and authoring are proliferating, but as is the case for software, the human tasks involved in video production can range tremendously in scale.

The teams behind professional film and television productions easily rival those behind the largest software projects. As with software, we believe that amateur video enthusiasts will embrace tools which will allow them to collaborate with like-

[1]From the number of views in December, this web site's share of the pie is 91%.

minded partners and through teamwork produce more elaborate video projects.

In effect, the combination of video with the instant accessibility of the web has led to the formation of an entirely new category of websites to create, edit, and share video creations. Popular examples include Kaltura [7], MediaSilo [8], but there are many others. Unfortunately, these solutions suffer from a common problem, namely that their mode of delivery is significantly limited with respect to reliability, quality and navigation features when compared to other modes of video delivery, namely downloads or portable storage such as DVDs. Furthermore, most of them are locked-down into using proprietary technology, often based on Adobe Flash, to play and stream their content.

To have a streaming editor that supports the same advanced navigation features that the other modes provide, such as fast-forward, rewind, scrubbing, and instant start-up we soon find the best-effort model of the above platforms to be insufficient. The need for these navigation features, and the need to support both high-end and low-end video-enabled devices, requires the streaming server to *adapt* its quality of service to match the capabilities of the client. This is very challenging, considering that none of the system resources needed, i.e., storage network and CPU/processing, provide such quality guarantees.

In our previous work we have presented a quality-adaptive streaming system, QStream that narrows the gap between streaming and the other modes of delivery [14]. The performance of our implementation, comprising navigation functionality, robustness, and quality are to our knowledge unrivalled by any other available streaming system, and is comparable even to that of the best local video applications. Our goal is to bring the innovative features of QStream into a platform which would reconcile video-editing and quality-adaptation.

We have reached the point where we can apply the quality-adaptation techniques of QStream to video-editing. Unfortunately, the initial QStream architecture was oriented towards the case of video playback only. Online video editors do a lot more than just playing video. They allow cutting videos, combining them, and applying effects to those videos to create more interesting videos.

As of this writing, there exist several open-source and proprietary frameworks that can be used to create sophisticated video pipelines with effects, transitions, and transcoding operations. Examples include GStreamer, DirectShow, and AviSynth

[2, 5, 6]. Those frameworks could drive all the necessary construction blocks to build a video editing platform. Their strength lies in the abundance of elements that they offer: decoders, encoders, (de)muxers, container format readers, etc., and their flexibility to connect these elements together. Unfortunately, they suffer from the same problem as the online video editors cited earlier, namely that they lack the capacity to adapt to changes in the availability of storage and network bandwidth, as well as processing resources.

This thesis is about the design and implementation of our system Qinematic. Qinematic is work in progress, and so this thesis reports on the initial decisions and experiences taken to reach our ultimate goal of creating a collaborative video authoring platform. The main contributions of this thesis are threefold:

- The introduction of *Qinematic* itself and its implementation, a new open-source prototype framework built to serve as the basis for a quality-adaptive streaming editor.
- An extension of the static pipeline model of QStream to match the flexibility and extensibility of existing media frameworks, without sacrificing quality adaptation.
- A demo application that demonstrates the framework in action, and that can be used to view new edited content.

Qinematic is a prototype framework written in C that provides the scaffolding necessary to build a quality-adaptive distributed video editor. Qinematic offers a set of editing operations on video, which are applied to video files according to a pipeline specification. The end-result of the pipeline can be visualized in a friendly Graphical User Interface (GUI) offering advanced navigation controls over the video: videos can be played in forward and reverse at speeds ranging between 1 and 128 times the nominal playback speed, or in "slow-motion" (0.5x). Videos can also be browsed by scrubbing or moving the playback point back and forth in time. Furthermore, unlike typical editors Qinematic is designed to work from remote content streamed on the Internet.

1.1 Goals

Qinematic is the first step towards an extensible and modular quality-adaptive video editing platform. Our future hope is to see Qinematic evolve into an elaborate editor for distributed video content. Once this is achieved all the ingredients will be in place to add collaboration in the picture. For the moment we shall discuss the challenges we have encountered in building our editing framework. The collaboration aspect is outside the scope of this thesis, and is considered future work, however we feel that what we have constructed will lend itself well with related work in this particular area, for tree based approaches like [17], or on collaborative Extensible Markup Language (XML) editing approaches.

1.1.1 High Level Overview of System Features

We now present a more technical overview of Qinematic, which can serve as a summary for the next chapters.

A main feature of Qinematic is that it is *Quality Adaptive*. That is, Qinematic can adapt the quality of its output on the client based on the availability of network and CPU resources which vary over time. On the server, quality adaptation techniques are also used to adjust the stream with respect to the available storage bandwidth using an optimized on-disk layout of the video files and some priority information. This will be detailed later in Chapter 2.

Qinematic is an extensible framework written in C and Python. The framework borrows design ideas from other popular frameworks, namely GStreamer, AviSynth, and DirectShow. The difference with those frameworks lies the thread model used, and the semantics for processing data to conserve quality adaptation. Those will be detailed in Chapter 3.

Qinematic also uses a descriptive meta-language (using XML) to specify the content and how it is modified. The edits are specified in this language, and at the same time read in to be visualized in the Qinematic player. The descriptor language borrows concepts from SMIL [19] to describe content and timing. For extensibility, we rely on naming units of the meta-language with registered names in Qinematic. Names of modules in the description files are mapped to elements of the C application framework using introspection features[2]. The language and its uses in Qinematic will be explained in Section 3.4.

During our investigation, we have also built other components that are not involved directly in the editing, but that are worth mentioning. First, we have constructed a web browser plug-in for Mozilla Firefox using Netscape Plugin Application Programming Interface (NPAPI) [10] to embed Qinematic in web applications. We have also created an OpenGL output driver to speed up certain rendering operations for Qinematic. More information will be provided during discussions later in Chapter 5.

The rest of the thesis is organized as follows. In Chapter 2, we explain the concepts behind quality adaptation and how the architecture of QStream influenced Qinematic's design. Then we shall provide details on the current implementation in Chapter 3. Chapter 4 provides a description of related work and shows their differences with Qinematic. Finally, we will conclude the thesis with discussions and conclusions in Chapter 5. Also, for readers that would like to play with the framework, Appendix A contains instructions to download, compile and run Qinematic.

[2]We rely on GObject reflection to inspect C elements.

Chapter 2

Design

Qinematic strives to create a platform with which teams can collaborate to author media. The platform must be intuitive and flexible to allow complex media creations, yet it must be efficient and manage available computer resources wisely when driving these creations. We believe Qinematic reconciles these two goals by first adopting architectural concepts like pipelining and pluggable components, concepts that are widely adopted by other media frameworks. Then, Qinematic adapts to resource fluctuations by taking these concepts and uniting them with QStream's event-driven model paradigms. This chapter gives an overview of how these two aspects have influenced the design of Qinematic.

2.1 Quality Adaptation

Quality adaptation is the process by which QStream balances the quality of the video stream to match resource availability. QStream is mainly concerned with storage, network, and CPU resources to read video from disk, send it over the network, decode it, and display it to the screen.

Because the availability of all these resources vary in time, for instance because the bit-rate of the video can change drastically from one scene to another or because these resources are shared with competing applications, QStream aims to gracefully adapt to situations where resources would be insufficient to maintain the maximum quality level.

Quality adaptation has several benefits. Firstly, it allows working with the same high-quality encoded video stream on a wide range of devices varying in capabilities from portable players to high-end desktops.

Furthermore, quality adaptation absorbs sudden variations in resource availability, which could otherwise result in interruptions of service. This also results in a viewing experience with fewer perceptible changes in visual quality, and graceful degradation in worst-case scenarios.

Quality adaptive streaming relies on the application to be written in a specific way, following 3 basic principles outlined in the next section.

2.1.1 Requirements

Each of the adaptations that QStream provides, i.e., network, CPU, and storage, rely on the following three general principles.

First, a quality adaptive system should be designed to produce incremental results, that is, results whose quality can be ameliorated by successive iterations of work. This provides greater control over the quality of the final results.

The second principle assumes that quality adaptation can be done across multiple quality dimensions. For example, SPEG [15], the streaming-friendly video compression format used throughout QStream, allows the quality of the video stream to be adjusted along two axes of scalability: temporal quality, and spatial quality (Signal-to-Noise ratio (SNR)). QStream provides ways to prioritize data across the multiple dimensions with a component called the *mapper*. The mapper takes the multidimensional coordinates of each media unit to process and maps those to a scalar priority value.

The third and last principle is *priority-progress* execution, which prescribes the use of timestamps and the priorities obtained from the mapper to perform adaptation. The timestamps are used to subdivide all the data to process in what we call *adaptation windows*, which we shall cover in Subsection 2.2.2. Those windows consist of stages where data are prioritized, and low-priority data can be dropped.

Entailment on Program Structure

QStream follows the principles enumerated previously by adopting a programming model inspired by the paradigms of reactive programming:

Event-loop At the core of QStream we find a single-threaded event loop which executes prioritized events. The loop can juggle various types of events, namely application specific events, and external events such as arrival of data on the network.

Timer and best-effort Events are categorized in two classes, timer-based, and best-effort. Adaptive operations will generally make use of a timeline to ensure that they are done by a certain time and will thus create timer events, which become eligible to run only after a certain release time is past. Best-effort events are used by operations that should run as soon as possible. Events of this type are always eligible to run, given that there are no outstanding timer events and no best-effort events with higher priority.

No event preemption Events should be short-lived, and cannot be preempted. This avoids unpredictable timing that can be caused by preemption, and helps avoid the need for locks and synchronization primitives. Long running operations have to be separated in smaller steps, either by changing code, or using co-routines. Fine-grained events can complete faster and be prioritized better.

Partial completion of events Allow for certain operations to be partly completed. The more time allocated for a task, the better the result. This cascades from the incremental results principle.

Cancellation of events As time passes and higher priority events arrive, it may be necessary to cancel execution of events that have expired. Many of the algorithms in QStream rely on this principle.

Qvid is a quality-adaptive streaming video player which is part of the QStream framework. Using the concepts enumerated above, Qvid is capable of adapting video quality according to available CPU, disk, and network resources. It is a complex application written in C, composed of approximately 90'000 lines of code.

We have leveraged this code to construct the extensible video platform that is Qinematic.

2.2 Qinematic Targeted Features

Our goal was to provide a good subset of features that are provided by local authoring applications, while still allowing streaming over the Internet and maintaining quality adaptation features. In order to have interesting results, we settled for a realistic list of editing features commonly found in other timeline based editors:

1. Defining clips. It should be possible to split long movies into shorter segments. Clips are defined as a consecutive series of frames from a video source.

2. Chaining Clips sequentially. We want to allow forming movies by concatenating clips together. Ideally, we would like different clips to come from the same or different video sources.

3. Adding effects to clips (intra-clip). We want to allow visual effects for every image inside a clip, such as converting their colours and/or image formats.

4. Adding effects between clips to provide transitions. The most common instance of inter-clip effects is the cross-fade, but in the general sense this extends to any effect that combines or superimposes in some way frames from two clips, such as picture-in-picture or overlays for example.

2.2.1 Call for an Extensible Pipeline

The Qvid video player project features a simple video pipeline, which we used as our code base to create Qinematic. In Qvid, data elements flow through a fixed number of stages that process the data. The video player is divided into two monolithic applications, the server and the client which each hold their half of the pipeline. The server side is responsible for retrieval of the media from disk and sending it over the Internet, while the client side receives, decodes and displays the video to the user.

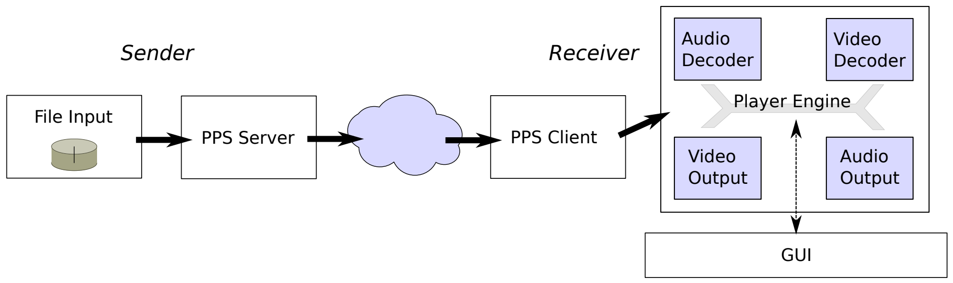


Figure 2.1: Static pipeline organization in the Qvid video player. Each rectangle represents a code module driving one piece of the pipeline. The pipeline is static, i.e., it cannot be reconfigured for different movies. The thick arrows indicate the direction of travel for the frames in the pipeline.

In the Qvid architecture, the media pipeline arrangement is static: it cannot be reconfigured to play different types of movies. The overall organization of Qvid is schematized in Figure 2.1.

On the server, chunks of the video are first read from disk in the file input module, prioritized and sent over the Internet to the client using the Priority-Progress algorithm, which will be covered later in this chapter. At the client, video chunks arrive to be decoded and then shown to the screen. The decoding and the display of the decoded data is orchestrated by logic inside an engine called the "player". Also, the user can control the flow of the video using the GUI using several action buttons, and a progress bar that can be used to "scrub" the video. These controls allow the user to change the speed, direction, and position of the playback. The GUI also indicates the state of the engine to the user to provide feedback.

The Qvid player was designed to play a single movie only, from the beginning to the end. The targeted features for Qinematic call for a more extensible model that Qvid's initial model could not provide.

Therefore, in order to implement clips and effects, we exchanged Qvid's static model for a dynamic one. In Qinematic, the pipeline is configured dynamically to reflect the composition of the movie to play, that is the arrangement and sequence of clips and effects to apply on the video source(s).

2.2.2 Qvid Timeline and Priority-Progress

The biggest strength of Qvid is that it can adapt to extreme streaming conditions and still provide fast start-up times and smooth uninterrupted playback with good visual quality. This is due to the way Qvid prioritizes its data: it employs a general algorithm called *Priority-Progress*, based on a sliding window approach. The algorithm is presented in previous work on media-streaming [15, 16] but will be reviewed briefly in this section.

At a given time the algorithm defines a series of frames eligible for transmission, denoted as an “adaptation” window. The constituent chunks of video frames enter and leave the window as it slides forward according to time. The chunks are prioritized along two dimensions, as discussed earlier, based on their degree of refinement (spatial layer), and on the temporal importance of the containing frame (adjusted for current playback speed). Thus, chunks will enter and leave the window according to time, but they will be enqueued within according to priority, and transmission will proceed in priority order amongst the eligible chunks. Also, the algorithm works by priority-drop: when the window position moves forward and a frame exits the transmission window, any chunks of that frame that are still unsent are dropped. We say that those dropped chunks are “expired”.

A big aspect of the algorithm lies in the timing of these adaptation windows. The adaptation window size is expressed in units of time, defined by a left and a right boundary. The contents of the window are the video data (video frames layered spatially) with logical timestamps falling within the window boundaries. The core insight of the algorithm is that the size of an adaptation window directly relates to a trade-off between responsiveness to interactive navigation actions (start-up times) and flexibility to smooth transient fluctuations in resource availability.

The priority order processing of the adaptation windows means that for a player to decode and display any particular frame, the player must wait first for the window position to reach a point where it knows that no more data will arrive for this frame. Therefore the size of the windows, i.e., the time between when a frame is first eligible for transmission (chunks of the frame enter the window), and when the frame expires (chunks exit the window), will have an impact over the start-up times. A small window will allow shorter start-up times.

On the other hand, small windows are more sensitive to variations as frames will expire faster, possibly causing frames to be dropped and quality to be uneven. Remember that chunks in the adaptation windows are processed first in priority order and then by timestamp. This implies that the algorithm will try to transmit all frames at an equal level of quality for the duration of a window. Hence, large window sizes generally increase start-up times but will be better in terms of adapting to resource fluctuations due to the greater smoothing effects they have on video quality.

Dynamic Window Scaling

To balance between the goals of shorter start-up times and better quality and robustness, the Priority-Progress algorithm dynamically adjusts the window size, starting small, and growing it as the stream proceeds. When a window grows, its position moves faster than the playback speed, without stalling the receiver.

The algorithm employs a technique called bandwidth skimming [13] to grow the adaptation windows. It borrows a small fraction of the client's immediate available bandwidth to build up a buffer of frames (the "skim") that will be needed later. In other words, bandwidth skimming allows sacrificing a small fraction of the immediate video quality to smooth out future resource fluctuations.

Because the adaptation window sizes are defined in units of time, the algorithm can control the amount of bandwidth skimming without knowing anything about the rate or throughput of the input and the resource. For example, if the window position is advanced at a rate of 10% faster than the nominal playback speed, then the window size can grow precisely 10% time units larger, and the reduction in quality/bit-rate of the video will be 10% (relative to whatever bandwidth was used).

Qvid uses Priority-Progress to execute a skimming plan that works in the face of arbitrary fluctuations of resource requirements and availability. When the player engine is requested to start playing a video, Qvid will calculate a timeline that will use small windows at first to provide quick start-up, and with bandwidth skimming grow the windows to provide increasingly smooth and robust video quality.

Adaptation Timeline

In Qvid (and QStream), the delivery of video will adapt as necessary to network [15], processor [16], and storage resources [14] due to the adaptation timeline.

The adaptation timeline is a divided in multiple consecutive stages, where each stage consists of a dynamic time window that grows as the video plays. The size of each time window will determine how much time the data spends in the corresponding stage before entering the next.

In Qvid, the sequence of stages in the timeline through which a frame passes is illustrated in Figure 2.2. A video frame first begins processing in the *setup* stage, which is when the metadata for the frame is first retrieved from storage. The algorithm uses the setup stage to smooth out delays of storage access, and to match the rate of storage requests to recent transmission rates. This avoids consuming chunks that will be dropped because they expire in the following stages [14].

The following stage is the network transmit stage, or *xmit* stage, where the frame is in the Priority-Progress adaptation window. Following transmission, the decode stage is the time during which frames can be decoded at the receiver (client).

Both the transmit stage and the decode stage have mechanisms to prevent initiating transmission or decoding of chunks too late. They do so by reserving a small jitter window at the end of the stage which acts as an expiry interval. Work towards the next stage can only be initiated before the chunk reaches the jitter window. QStream allows partially sent chunks to be cancelled, and so chunks whose transfer or decoding is incomplete by the end of the respective window will be cancelled, but that incurs an overhead (wasted bandwidth or CPU). The expiry windows will prevent this problem, given that they are longer than the time taken to complete the work (transmission or decoding of the chunk, resp.).

The timeline finishes with the end of the decode stage, when decoded frames are enqueued to be presented to the screen (presentation timestamp). This also corresponds with the end of the adaptation in the lifetime of frames. After that, the frames are either shown or are skipped, because they cannot be shown in part or incrementally (at least, not efficiently).

The timeline in Qvid is used to provide a trade-off between fast start-up, and quality and robustness by controlling its size (duration). To achieve the goal of

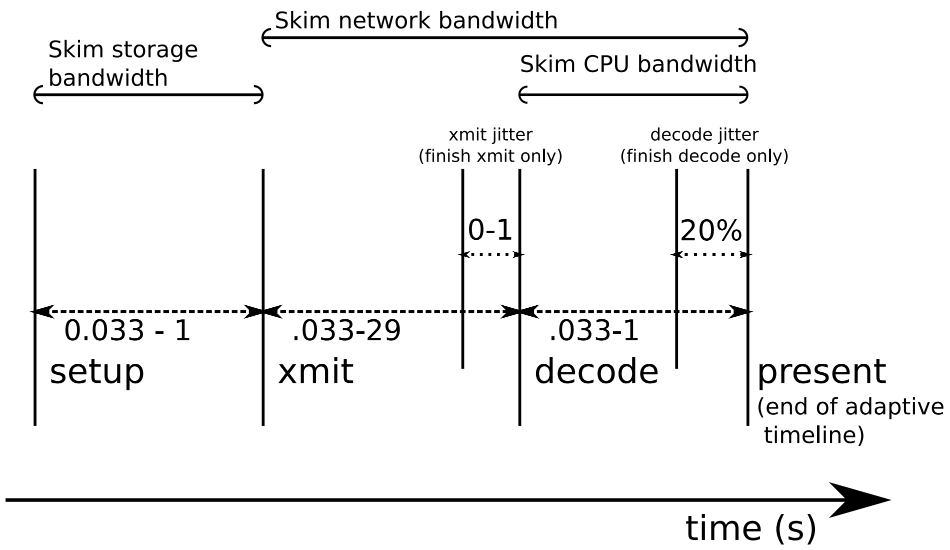


Figure 2.2: The Qvid pipeline controls its flow of data using the above timeline. The figure shows the sequence of windows/stages in the timeline, as well as each stage's initial and final size (in seconds). Data flows from left to right.

perceptually instantaneous start-up time, the pipeline stages use the minimum durations feasible for the first frame of the stream, which by default is the nominal duration of a single video frame[1]. The reasoning behind this choice is that if the server and client have the basic capacity to process the stream in real-time without interruption, then the transmission and decoding stages must be capable of performing their work as fast as the frames will be presented. From their initial durations, the windows grow over periods of up to a few minutes until they attain sizes in the order of a few seconds, with the notable exception of the transmit window which is configured to grow until it reaches around 30 seconds in size.

2.2.3 Leveraging the Timeline

As we have seen earlier, the timeline is what allows Qvid to schedule processing of the constituent chunks of the video during streaming. It is what regulates the

[1]At 30 frames per second, this corresponds approximately to 0.033 second

times at which the various data enters each stage of the pipeline. The timeline is also resized during playback to allow fast start-up, and also provide resilience to resource variations.

In Qinematic, the video pipeline construction is dynamic to allow for various filters and custom processing of the data units. This is different from Qvid which has a static, linear pipeline from the server to the client. As we have seen in the earlier sections, the timeline also contains a static set of stages, closely reflecting the organization of the processing modules/stages in the Qvid.

In order for Qinematic to benefit from the advantages provided by the timeline, the timeline must be used in a different way. We will explain in the next chapter how we proceeded.

2.3 Summary

In this chapter, we have revisited some of the previous work on quality adaptation to clarify how it influenced the design of Qinematic, and to provide background for later explanations on the implementation. Qinematic adopts architectural concepts like pipelining and pluggable components, concepts widely adopted by other media frameworks, and extends the static pipeline model of Qvid. Furthermore, Qinematic still embraces the QStream model and the Qvid timeline by reusing the timeline to orchestrate the processing of the video chunks and preserve their quality adaptation capabilities.

Chapter 3

Implementation

The application model of Qinematic is designed to be simple, yet be very flexible and general enough to provide useful and professional looking video manipulation elements. Qinematic is inspired by other mature frameworks, notably the GStreamer framework [6], and AviSynth [2], which also provide flexibility and extensibility in a general fashion.

We believe QStream proved to be an elegant and efficient solution to the problem of achieving quality-adaptive streaming. For building Qinematic, we decided to approach the problem by augmenting our existing system with extensibility and flexibility, rather than from the other end, i.e., extending or porting an existing framework to support quality adaptation. The quantity of changes required would have otherwise been impractical, given the number of differences between QStream and GStreamer or AviSynth: event loops, threading models, pull vs. push streaming, and closure formats, etc.

3.1 Program Structure

An application using Qinematic has to adopt a specific structure to ensure proper functioning. The core of Qinematic is built upon QStream's QSF library, which provides all the scaffolding necessary to build a quality-adaptive framework, but imposes a set of ground rules on the layout of Qinematic applications. For example, as discussed in Subsection 2.1.1, Qinematic's mechanisms all revolve around a global event loop which executes timer events and prioritized best-effort events. Fortunately, applications already built with QStream will require very little modification to benefit from Qinematic's features.

Furthermore, because Qinematic is built atop QStream, those applications should also possess the characteristic traits of quality-adaptive applications, e.g., work units encapsulated in short non-preemptible events and asynchronous-IO operations.

Here, we present a global picture of the life cycle of a Qinematic application by listing some of the application requirements. Some of the steps listed in this section will be described in further detail later.

1. The Qinematic application first initializes the Qinematic subsystem (via a single call to `qinematic_init()`)
2. Creation of a Qinematic context (and event loop) (via a simple call to `qine_context_new()`)
3. Construction of an element pipeline graph (see Subsection 3.4.1)
4. Pipeline activation and user interaction.
5. Tear-down of the pipeline, and event-loop halt (see Subsection 3.4.2)
6. Context deletion via call to `qine_context_delete()`

The function `qinematic_init()` initializes all Qinematic dependencies, loads the Python module (see Subsection 3.4.1), and invokes any registered initialization functions. Those registration hook functions are in turn responsible to introduce new element types to the system. Those hook functions are themselves registered at load time using constructor functions[1], hence allowing a very flexible plug-in infrastructure for static and dynamic libraries.

In item 2, the function `qine_context_new()` creates the main event loop (though the details of its initialization are abstracted away from the user) that will drive the application and returns a context pointer, which essentially constitutes a handle to that loop. An attachment point is created on the loop, so that the

[1]See gcc's `__attribute__ ((constructor))` attribute.

Qinematic application can add arbitrary information to the loop and context. Given the event loop handle (available in thread local storage), the qinematic context and anything attached to it can also be retrieved easily and quickly.

The application logic all happens in item 3 where elements are assembled together to form the streaming media pipeline. The system does not limit in any way the number of active pipelines that can coexist, but it should be more common to find a single, possibly very complex, pipeline. At this point, the event loop is running and media can start flowing in the application in the form of buffers and events.

The last 2 items are executed when the application reaches its conclusion. Their task is to shut down Qinematic activity and reclaim borrowed resources. They have the reverse effects of the first items.

3.2 Structured Flow of Media

In this section we shall delve into the organization of the media pipelines in Qinematic applications, and into a description of the data running through them.

Qinematic features a flexible platform to build complex pipelines for audiovisual streaming applications. Readers familiar with other media frameworks should recognize many of the design similarities in common with Qinematic. At a very high level, a Qinematic pipeline could be defined as a graph of communicating *elements* or stages which process data units that flow through them. They do so from an element designated as a *source*, through any number of other intermediary elements, until they reach a *sink* element.

The data units are classified into two categories, *buffers* and *events*, that are distinguishable by their direction of travel in the pipeline as well as their payload. For now, a brief description of the two will suffice to get a global view of the system.

The buffers contain data to be processed by the elements and always travel downstream, i.e., from source to sink. Once an element has finished processing a buffer, it *pushes* that buffer to a neighbouring downstream element. Also, Qinematic focuses only on image buffers for video data and audio frames for audio data, but buffer contents can be arbitrary.

Events represent control units in the pipeline, and can travel in one direction or the other, depending on the nature of the event in question. As their name implies, events are used to indicate some sudden change in the flow of the pipeline. Examples of events are the *SEEK* event to indicate a change of position in the stream, or the *EOF* event, to notify downstream that the end of a stream/file has been reached. Elements will generally take various actions on receipt of events, actions which will usually end by passing the event to appropriate neighbour elements. We shall defer the in-depth discussion on the pipeline creation/destruction as well as event and buffer propagation to Section 3.4.

3.2.1 Elements and Pads

The pipelines are formed of *elements* that process multimedia data flowing between them, and *pads* that provide communication channels between elements that connect to them. Pads abstract away the differences between neighbouring elements by allowing them to communicate through a single and very simple programming interface.

Elements pass processed output to their downstream neighbours, but the nature of the processing itself is variable. Generally they will be used to alter data, but they can also act as monitors or profilers of other elements. This concept is well known, and is being used extensively in many other media frameworks, such as GStreamer [6], AviSynth [2], and to some degree Microsoft Media Foundation (MF), which provide statistics gathering elements.

Pads have a direction: they are either *source* or *sink* pads. The former type pushes buffers out, while the latter receives them. Pads are also connected to one other pad (at most), denoted as a *peer*. The diagram in Figure 3.1 depicts a portion of a pipeline containing 3 elements and pads.

Pads are normally attached to elements, so that everything the pad receives is handed out to the element. An element attached to a pad is said to *own* the pad. The element-to-pad relationship is one-to-many, so elements can own as many pads as they need (but a pad can only have one owner).

Different pads attached to an element can receive data from different neighbours or data of different types (e.g., audio and video). The element on the right

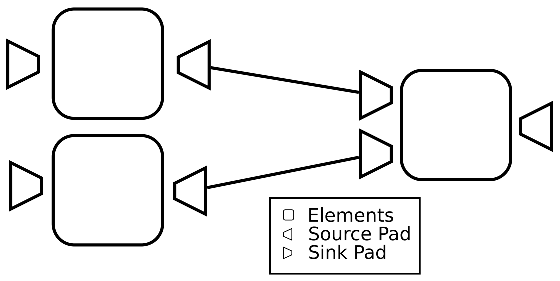


Figure 3.1: A section of a Qinematic Pipeline with 3 elements and their pads. Buffers flow from left to right.

hand side of Figure 3.1, for instance, owns 3 pads, two of which are sink pads. Pads also have a name which lets elements connect together along named paths.

All of the data processing logic is placed in the code of the elements. The pads don't participate in the logic, but they are used as bridges connecting the various elements and thus determine where buffers get processed.

3.3 Qinematic Elements

As of this writing, several basic elements are provided by Qinematic to allow the creation of complex video authoring projects. Firstly, movies can be decoded and played by decoder and output elements. Secondly, *clips*, or sub-movies can be created out of other movies, recursively, and can be organized sequentially or overlapped in time with control elements. Lastly, Qinematic provides intra-clip transformations, such as filter elements, to apply certain post-processing and/or artistic effects to video. We will dedicate the rest of this section to describe in further details these basic elements.

3.3.1 The Clip Element

The Clip element (or CLIP) lets the author define a new shorter movie from a continuous section of a longer one. It can be used for instance to extract a scene or shot from longer footage.

The CLIP is given two numeric parameters at construction time, a *start-frame*

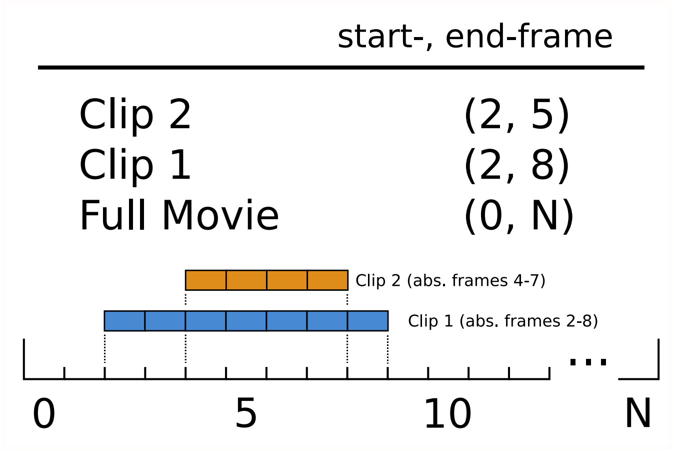


Figure 3.2: A setup of nested clips. Frames of Clip2 are a subset of those in Clip1. When Clip2 is played, only frames 4 to 7 (incl.) from the full movie should be shown.

and *end-frame* which define its position and length. Clips can be cut out of the complete movie or from other clips, as many times as needed. In the latter case, the clip position (start, end) is relative to the bounds of its source. This makes it very easy to define new movies out of existing ones.

When a clip is cut out of a longer segment, like Clip2 in Figure 3.2, it will be placed downstream in the pipeline from that segment (one step further away from the original content). In Qinematic, it is the `seek` event that specifies the start and end points for stream playback. The seek events originate from the sink of the pipeline, the GUI, and propagate back towards the source. Every time playback is stopped or started, new start and end bounds are calculated. When a clip receives a `seek` event, it translates the playback start and end positions into the coordinate system of the element upstream[2], so that the final positions at the source are relative to the start and end of the whole video.

The clip element is perhaps the simplest of all existing elements provided by Qinematic. The clip element intervenes only in the processing of events `seek`,

[2]We increment the `seek`'s start position by the clip's offset within the upstream element. A similar approach is taken when playing backwards.

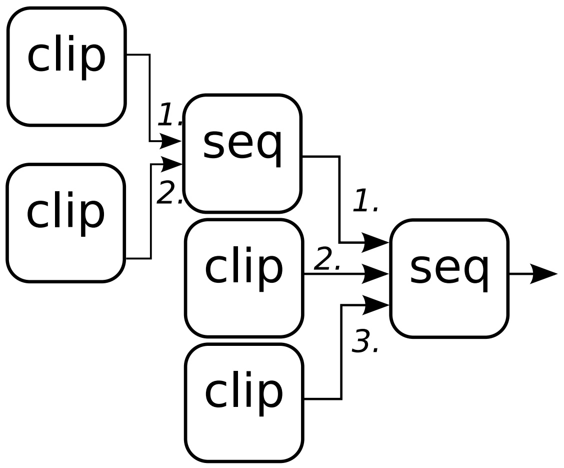


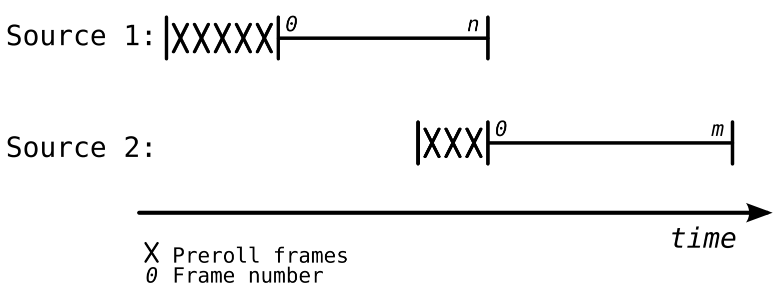
Figure 3.3: Section of a pipeline with multiple clips and sequence elements. The sequence elements play their sources in the order they were connected (illustrated by the numbers).

`open_done`, and `seek_done`. On open, the stream duration is set to the length of the clip, and for `seek`and `seek_done`, seek position offsets are updated to reflect the bounds of the clip. Buffers arriving at the element, on the other hand, are left untouched and are simply forwarded downstream.

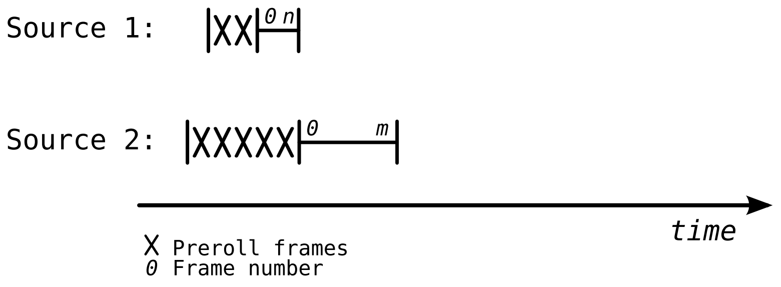
3.3.2 The Sequence Element

The Sequence element (or SEQ) lets the user combine various sources into a continuous sequence. When played, each source in the sequence will play in turn, with no perceptible gap nor overlap. In the simplest cases the SEQ is usually connected to a series of CLIP source elements, but it is also compatible with all other types of elements that can be used as sources, even other sequence elements. To illustrate this, Figure 3.3 shows a typical pipeline arrangement for a Seq element.

The challenge with the SEQ consists in orchestrating the start times of the contained clips so that their respective first frames be ready in time on the client. Each source requires a certain amount of processing before its first frame is to be shown on screen, mostly because of the time required to process the pre-roll frames and



(a) Case 1



(b) Case 2

Figure 3.4: Possible arrangements when sequencing multiple sources. In the first case, the second source is started shortly before the first one finishes. For case 2 however, the second source has to be started first to process the long train of pre-roll frames.

delays to get the start messages across from the client to the servers.

In order to get a video frame to show up on screen, several actions must be taken on the original encoded frame, including fetching, decoding, sending it to the display, etc. The time taken to complete some of these steps, for instance decoding, will vary greatly depending on the number of dependent frames. Because playback is allowed to start and stop at any frame in Qinematic, the nature (e.g., I,B,P) and number of dependent frames of the very first frame that is to be shown in a clip will have great effect on delay between the time the clip is "started" (data being served) and the time that first frame can be shown to screen. This section presents the logic used in the SEQ element to correctly determine when the timelines of the individual sources should be started, given the time at which the first frame should appear on screen.

From the Qvid timeline, the SEQ element infers how much time it should take for the first frame to be displayed for each of its sources, based on the duration of that source (in frames), the frame-rate, and the number of pre-roll frames. The algorithm deals with different forms of interleavings of two sources, as illustrated in Figure 3.4, or of more than two sources. Algorithm 3.1 orchestrates the start time for each source.

It is worth mentioning that the current approach composes clips using the same timeline strategy that is employed for the single clip case. That is, the timelines work the same for all clips regardless of their interleavings in time: fast start-ups are provided by small windows at first, which then grow as the clip advances to provide better robustness and fewer changes in quality. It would not be strictly necessary to start all the clips with the same timeline strategy, as ideally, the timeline of one clip could continue from the state of the previous clip. This is an optimization that is left to future work.

3.3.3 XFade, the Cross-Fade Element (Lap-Dissolve)

Cross-fading two clips consists in playing the media from two clips and having the first clip fade out when the second clip fades in, by superimposing the last few frames at the end of the first clip with the first few frames of the beginning of the second clip. Cross-fades are used mostly to create mood and smooth out transitions between scenes.

It is often the case that cross-fades implicate one movie clip consisting of a solid colour, such as black or white, blended with the end or the beginning of a regular video clip. Those types of cross-fades are commonly designated as *fade-ins* or *fade-outs*. In Qinematic, those cross-fades are not treated specially, except for the fact that the solid colour frames can be generated on the fly.

Qinematic allows creation of cross-fades via the Cross-Fade (XFADE) element. The XFADE receives data from two possibly overlapping input streams and is configured at the time of its creation with the duration of the overlap. In the degenerate case where the duration of the overlap duration is nil, the element behaves like a SEQelement configured with the same input streams.

The element will pair frames from the two input streams during the overlapping

Input:

speed:	The playback speed (e.g. 0.5, 1.0, 2.0, ...)
NUM:	The number of clips in the sequence.
vid_rate:	The frame-rate of the video (same for every clip).
$num\_frames[i]$:	The number of frames in the i'th clip.
$num\_depframes[i]$:	The number of dependencies for the 1st frame of the i'th clip.

Output:

$start\_offset[i]$:	Time difference between the time the SEQ starts playing, and the time the i'th clip starts playing.
$output\_offset[i]$:	Time difference between the time the SEQ starts playing and the output time of the first frame (not counting dep. frames) of the i'th clip of the sequence.

```
$frame\_duration \leftarrow 1/(vid\_rate * speed)$
$min\_start\_time \leftarrow$ +Infinity
Timeline $T \leftarrow$ timeline_init($clip\_duration, frame\_duration$)
for $i \leftarrow 0$ to $NUM - 1$ do
    $clip\_duration \leftarrow num\_frames[i] * frame\_duration$
    #ndt is the instant in time we want
    Position $p \leftarrow$ pos_init(
    $T, ndt = num\_depframes[i] * frame\_duration$)
    if $i == 0$ then
        $output\_offset[i] \leftarrow p.output$
        $start\_offset[i] \leftarrow 0.0$
    else
        $output\_offset[i] \leftarrow output\_offset[i-1] +$
        $(num\_frames[i-1] * frame\_duration)$
        $start\_offset[i] \leftarrow output\_offset[i] - p.output$
    end
    $min\_start\_time \leftarrow$ MIN($min\_start\_time, start\_offset[i]$)
end
#At this point, min_start_time is
#either 0.0, or negative
for $i \leftarrow 0$ to $NUM - 1$ do
    $start\_offset[i] \leftarrow start\_offset[i] - min\_start\_time$
    $output\_offset[i] \leftarrow output\_offset[i] - min\_start\_time$
end
```

Algorithm 3.1: Scheduling Start Times for the SEQ Element

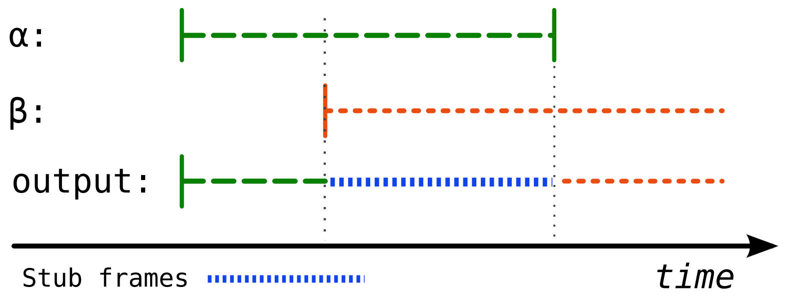


Figure 3.5: The frames output by XFADE are either taken from α or β directly, or "stub" frames that represent the combination of one frame from each input stream

region and blend them so as to output a single frame from each pair formed. Currently, the blending operation applied on the pairs is alpha blending, with an alpha parameter that is a function of the pair's time distance from the commencement of the overlap. This specific choice of blending function will result in a smooth visual transition from the frames of the first to those of the second stream.

For the remainder of Subsection 3.3.3, the following assumptions and definitions shall hold:

- We denote the first stream of the cross-fade α, and the second stream β.
- Streams α and β have the same frame-rate, and their presentation timestamps coincide (same "phase")[3]. Thus, a one-to-one correspondence can be established in the overlap region between any frame in one stream to another frame in the other stream.
- For a given configuration of some XFADE element, let function `dur`(s) and $|s|$ represent the time duration and number of frames of the stream s, respectively. Argument s could be one of the full input streams, or just a portion of a stream.
- For simplicity, but without loss of generality, we assume that the first frame of stream α is displayed at time 0.

[3]Internally, we use frame numbers to derive presentation timestamps

XFADE Output

Even though the XFADE element takes its input from two different streams, to the elements downstream it appears as a single continuous stream whose length is as long as α and β concatenated minus the duration of their overlap. Also, the output frame-rate of the XFADE element is the same as its input streams frame rate. Generally speaking, this means that each time a frame from either α or β is received by the XFADE element, it will also output one frame (or one frame for each *pair* of frames during the overlap).

XFADE Delayed Execution

The XFADE element differs greatly from the other element types seen so far because of its ability to output data coming from a mix of multiple streams, each with different scheduling and resource constraints. In order to avoid wasting resources, and to allow better prioritization of other parallel tasks, the expensive blending operations are delayed for as long as possible, using Qinematic's `QineImage` work closure features (see Subsection 3.4.4).

Frames outside the overlapping region will be output by the XFADE element unmodified, whereas special *stub* frames will be output for frames that will use the delayed processing feature. Unlike regular video frames, stub frames do not carry image data. Rather, they carry the integer offsets of the two frames of the pair that will be alpha-blended.

Each stub frame S_i is assigned a number i when it is created, a number from which the corresponding frame numbers from α and β can be calculated. We will denote stub frame S_i to represent the pair (α_i, β_i), each taken from their respective streams at position i, relative from the first frame output by the XFADE element. When the time comes to output frame S_i, the corresponding frames from the pair will be retrieved and blended. The main reason for delaying the retrieval of the peer frames is that each of the XFADE element's input streams is subject to its own scheduling constraints.

No guarantees can be made regarding the arrival of frames from either α or β. This implies that it will be possible, in poor streaming conditions for instance, that there exists a stub S_i for which α_i, β_i, or both are missing when it comes time to produce the blended frame. Rather than dropping stubs whose member frames cannot be retrieved (e.g., dropped because of adaptation decisions), Qinematic covers for the loss by substituting the missing peer frame with one that is identified as the "most sensible" at the time the stub frame is blended. In case no suitable replacement candidate can be found, a solid-black frame can be used instead. In good conditions, frames α_i and β_i should have arrived by the time S_i needs to be displayed, in which case then the blending operation simply proceeds by picking those two.

The candidate frame for substituting a missing frame is selected using the following criteria:

1. Comes from the same input stream as the missing frame.
2. Comes earlier in presentation order than the missing frame.
3. Presentation timestamp must be as close as possible to missing frame's own timestamp.

Finding replacements that match these criteria will reproduce the visual impressions left on the viewer by missing or dropped frames during playback of a single video stream. In the single video case, a missing frame causes the previously displayed frame to stay longer on the screen. If a viewer were to superimpose two video sources with varying alpha channels, and if each source handled missing frames the same way as the single-source case does, she would see exactly what the XFADE element outputs. That is, at any point in time during playback of the overlap period of the XFADE element, the viewer will see a blend of the most recent information available at that instant.

XFADE Element's Coordination of Start Times

Because XFADE's input streams are subject to independent constraints, and use independent adaptation timelines, the element must have a certain level of control over the arrival times of buffers. The XFADE element has to coordinate the two input streams to address the following issues:

1. Both input streams must be started at adequate times. Starting too soon may cause unnecessary memory usage or sacrifice some quality (some other operations could have been given more processing time). On the other hand, starting too late may cause drops in quality or interruptions in video playback.

2. Frames from the two input streams may never arrive, or may arrive in different orders. This means that certain frames may not be paired up with their ideal one-to-one peer.

In a manner similar presented in the algorithm for scheduling start times of input streams of SEQelements, streams α and β can be coordinated in such a way that the two frames used to compose the first frame of the overlap have arrived at the XFADE by the time the first frame of the overlap is to be output. This is done by adjusting timeline origins so that the output of the first overlap frames of α and β match. This simple approach allow timelines from both streams to work identically, regardless of their arrangement in time. The main drawback of this approach is that the windows of the timeline of β will start small, while those of α will have grown by the start of the overlap. This will unnecessarily cause the frames of β to arrive at a higher rate than those of α. Like we mentioned in the description of the SEQ element algorithm, one optimization over this arrangement of the timelines would be to start the timeline of the second stream with window sizes matching the first stream.

Let `o_offset` denote a relative time value at which the first frame of one of the input streams should be displayed, then XFADE uses the following rules to set the time offsets to apply to α and β.

```
o_offset_alpha = position.output for first frame of alpha;
o_offset_beta  = o_offset_alpha + duration(alpha) - duration(overlap);
```

XFADE Element Core Mapping Algorithm

This section describes the algorithm used by XFADE to map stub frames to received buffer frames from the two input streams when forming pairs. The XFADE mapping algorithm is composed of a few subroutines. First, when image buffers (containing a video frame) are received from the element's input pads, procedure `XFADE_RECV_BUFFER` is called. Frames are then added to a data structure that allows the frames to be later retrieved, and this operation is performed by procedure `XFADE_REMEMBER_FRAME`. A counterpart to this is invoked by the algorithm to determine the moment at which it is safe for frames to be "released" from the same data structure: `XFADE_SET_FORGET_DEADLINE`. Lastly, another procedure is invoked on the image buffers when they are ready to be output: `XFADE_READY_IMAGE`.

Algorithm 3.2: XFADE_RECV_BUFFER()

```
   input  : from (Origin of image buffer, either $\alpha$ or $\beta$ ).
            buffer (The image buffer received).
 1 func XFADE_RECV_BUFFER (from, buffer)
 2     # offset from 1st frame
 3     $i \leftarrow$ get_buffer_pos (from, buffer)
 4     XFADE_REMEMBER_FRAME (from, i, buffer)
 5     if outside_overlap (i) then
 6         push_buffer (buffer)
 7     else if not peer_has_arrived (from, i) then
 8         stub $\leftarrow$ create_stub ($pts = i$)
 9         push_buffer (stub)
10     end
11 end
```

Algorithm 3.2 lists the actions taken when frames arrive at the XFADE element. The procedure first finds the offset of the buffer relative to the first frame of α. The input frame is remembered so that it may possibly be used when blending frames later on. It is worth pointing out that frames falling outside the overlap are also remembered, because they could become substitutes for missing frames inside the overlap.

On line 5, we determine if the frame is inside the overlap or not, by comparing the presentation order of the frame with the known boundaries of the overlapping section. Frames that fall into the non-overlapping portions of the XFADE element's schedule can then simply be passed downstream on reception. Line 7 deals with buffers that are part of the overlap and filters out cases where the corresponding peer in the other stream has already arrived, in which case a stub frame should already have been pushed out for that frame. In this case, no further actions are needed. Given the presentation timestamp of a frame from α, we determine the corresponding frame from β that it should be paired with (and vice-versa) by following simple additions and subtractions.

Algorithm 3.3: XFADE_REMEMBER_FRAME()

```
input : from (origin of image buffer, either $\alpha$ or $\beta$ ).
        i (offset from 1st frame).
        buffer (The image buffer received).

func XFADE_REMEMBER_FRAME (from, i, buffer)
    lookup ← get_frame_lookup (from)
    handle ← lookup_insert (lookup, key = i, buffer)
    predecessor ← handle.prev
    successor ← handle.next
    if predecessor $\neq$ Nil then
        output_time ← get_display_time (from, i)
        XFADE_SET_FORGET_DEADLINE (from, predecessor,
        output_time)
    end
    if successor $\neq$ Nil then
        output_time ← get_display_time (from,
        get_buffer_pos (from, successor) )
        XFADE_SET_FORGET_DEADLINE (from, buffer, output_time)
    end
end
```

In an adaptive streaming scenario, we must allow for frames to arrive out of order, because of inter-frame dependencies, and varying temporal importances. We also must anticipate that some frames may arrive only partially, or that they may never arrive in time. Moreover, in the current implementation, the priority (or utility) of a given frame does not take into account the dependencies it may have outside its own stream. This means that frames from one stream labelled as high-priority (or with high utility) may very well end-up paired with low-utility frames from the other stream because their presentation timestamps coincide.

Algorithm 3.3 stores the frames that arrive in a search data structure that al-

Algorithm 3.4: XFADE_SET_FORGET_DEADLINE()

```
input  : from (origin of image buffer, either $\alpha$ or $\beta$).
         buffer (offset from 1st frame).
         deadline (The time at which buffer is to be forgotten).

func XFADE_SET_FORGET_DEADLINE (from, buffer, deadline)
    $forget\_event \leftarrow buffer.forget\_event$
    if $forget\_event \neq Nil$ then
        qsf_event_cancel (forget_event)
        free (forget_event)
    end
    def forget_closure (from, i)
        $lookup \leftarrow$ get_frame_lookup (from)
        delete_from_lookup (lookup, $key = i$)
    end
    $forget\_event \leftarrow$ create_event (forget_closure, deadline)
    $forget\_event.arg[0] \leftarrow from$
    $forget\_event.arg[1] \leftarrow i$
    $buffer.forget\_event \leftarrow forget\_event$
    qsf_event_submit (forget_event)
end
```

lows fast retrieval of frames keyed by their position (presentation order) within the XFADE element. The lookup must allow finding data "closest" to the key given, must provide sorted iteration of values in presentation order, and must provide fast deletion and insertion. Off-the-shelf balanced search tree implementations[4] that have been augmented to allow iteration over their values are good candidates for that purpose. If we also consider that the frames arrive at the XFADE element partially sorted in order of Presentation TimeStamp (PTS) because of the priority order in which they are received, we might instead use a skip list [18] in the future, which might prove to be more efficient under the circumstances.

Frames are kept around in the XFADE at least as long as they can be needed. The XFADE maintains references to a buffer with PTS i until any frame in the same stream with PTS j, with $j > i$, passes its presentation time. The algorithm XFADE_REMEMBER_FRAME() determines the precise times at which buffers can

[4]Balanced tree implementations are widely available in multiple languages.

be discarded, based on buffer display times. This time value is calculated by the helper function `get_display_time()`. This function uses the QStream timeline of a stream and the buffer's PTS within that stream to perform the calculation.

When the XFADE element receives a new buffer, lines 7 and 11 in Algorithm 3.3 update the "forget time" for both the new frame and previous frame (in presentation order) in the lookup at the time of reception. For example, if the lookup were to hypothetically contain two frames A and C with respective PTS x and $x+2$ at the time another frame B with PTS $x+1$ was fed to Algorithm 3.3, the algorithm would setup events to forget A at the time B is displayed (line 7), and B at the time C is displayed (line 11). Algorithm 3.4 is the function that is responsible for registering the events with the QSF event loop. It defines a closure that will remove the forgotten buffer from the lookup. Because this function may be called several times on the same input stream and buffer, it takes care of cancelling any existing event before registering a new event.

Algorithm 3.5: XFADE_READY_IMAGE()

```
input  : stub (the image stub to blend).
output: The blended image.

func XFADE_READY_IMAGE(stub)
    $i \leftarrow stub.pts$
    $lookup_\alpha \leftarrow$ get_frame_lookup($\alpha$)
    $lookup_\beta \leftarrow$ get_frame_lookup($\beta$)
    $img_\alpha \leftarrow$ lookup_closest_match($lookup_\alpha$, $i$)
    $img_\beta \leftarrow$ lookup_closest_match($lookup_\beta$, $i$)
    $transparency \leftarrow (i - overlap\_start)/overlap\_duration$
    $stub.data \leftarrow$ alpha_blend($img_\alpha$,$img_\beta$,$transparency$)
    $stub.ready \leftarrow true$
    return stub
end
```

When a `QineImage` must be displayed, its `transform` method is called. The stub frames created by `XFADE_RECV_BUFFER` are setup to invoke function `XFADE_READY_IMAGE`, which retrieves the pair of images from the two input streams and performs the alpha blending operation. Once `XFADE_READY_IMAGE` returns, the image is completely transformed and can be displayed.

3.4 Pipeline Assembly

An application's behaviour in Qinematic is completely driven the manner in which its pads and elements are chained together. The structure of the graph of elements and the nature of each element determines how data is moved and how it is processed. This arrangement of elements and pads is called a *pipeline* in Qinematic, like in most of the other frameworks. This section intends to shed light on the tools Qinematic provides to construct, modify, and destroy the pipeline elements, as well as provide a description of the established data model.

The elements drive the application in two possible ways. First, they can act on incoming data *immediately* by dropping it, pushing it downstream, or transforming it before passing it down. The links formed between elements in the pipeline will determine *where* image and audio buffers are transported. Also, the logic inside elements can enclose additional units of work to be processed inside the passing data. These additional processing stages can be stacked on buffers to allow operations to be performed in an order that would be difficult to program from the arrangement of the elements in the pipeline graph alone. This will be covered later in the chapter when we discuss operations on buffers (see Subsection 3.4.4)

In Qinematic, a pipeline is fully described by an XML file. Using a simple and transparent descriptive language such as XML allows the Qinematic tools to infer a lot of information statically from the description file.

Several factors have been considered in choosing the language:

1. Manipulating the language from a web application.

2. Reading, Querying, and Generating the pipeline description files.

3. Extending language features.

XML presents an ideal solution to all these factors and blends well in the Web picture. The wide availability of tools to parse and verify XML also made it easy for us to develop tools outside the browser, such as on the desktop.

We have considered other simpler languages such as Javascript Object Notation (JSON) to describe the pipelines, which provides several of the desirable features that XML provides. Even though the language is simpler to manipulate and allows literal translation in JavaScript, we found that that the available tools for manipulating JSON were not as mature and feature rich as those available for XML with regards to error-reporting. The initial parser for constructing Qinematic pipeline used JSON as the source language, and we found it more difficult to report meaningful errors. We have found also that working with XML will generally result in more maintainable pipeline definition files.

3.4.1 Construction

Qinematic provides a set of primitives for assembling pipelines. Elements are first constructed from a set of named attributes and values, and then linked together along named paths.

Constructing elements requires an element type, an element name, and a list of *name-value* pairs for any properties that are essential at construction time. The permitted names for the types of elements (e.g., "seq", "xfade", etc.) are determined at the time the element types are registered with Qinematic. The list of permitted attributes for each type can be introspected at run-time (using reflection) from hook functions on the element class objects registered.

Linking two elements requires their names (or a pointer), as well as names of their respective pads that should be joined. If the names of the pads are not specified, then Qinematic defaults to using the first available pad of each element. The only restriction to the application of this rule is that the pad directions are compatible (i.e., sink and source).

Qinematic provides a basic C Application Programming Interface (API) for constructing elements, specifying their attributes, manipulating their pads and the connections of those pads. The core API is very simple, but is not suitable for manually creating complex pipelines with hundreds of elements.

For that purpose, Qinematic provides a higher level C API that leverages the core API, as well as Python bindings to this API and a pipeline builder tool. The builder can create and play a pipeline from a static XML pipeline description file containing element definitions and their links. An example valid input for this utility is listed in Figure 3.6.

The description files are enclosed in a <pipeline> tag and contain a set of <root>

```
<pipeline>
 <root rid="firstroot">
  <clip eid="clip100">
   <prop name="start-frame" type="int">1000</prop>
   <prop name="end-frame"   type="int">1099</prop>
   <vdec>
    <client eid="fi3kdb">
     <prop name="filename">3000.db</prop>
     <prop name="addr"    >localhost</prop>
     <prop name="port"    >5000</prop>
    </client>
   </vdec>
  </clip>
 </root>
<pipeline>
```

Figure 3.6: Example pipeline definition file featuring 3 elements: a client, a decoder, and a clip.

elements, which identify entry points into the pipeline. The <root> elements are listed at the top level of the file, and act as sink elements in the static pipeline. When the pipeline is instantiated however, the pipeline builder will connect the static sink element from the XML definition to a QvidPlayer element. The player element will allow the user to view the video data flowing through the pipeline. The same set of operations that can be performed on a Qvid video can be performed on a Qinematic element regardless of the complexity of the static pipeline. That is, playing in forward and reverse directions at fast and slow speeds, and scrubbing.

Tags in the XML file will be instantiated into Qinematic elements, and connections between the elements will reflect closely the organization of the tags in the file. The current XML format reserves names *root*, *pipeline*, *prop*, and *ref* for special XML elements, but all other names are mapped to actual programmed Qinematic elements (e.g., seq, xfade, clip). The other special elements not discussed so far are:

- The <prop> tag defines a named property on the enclosing element. The value of this attribute is contained inside the <prop> element itself, and the type of that value is specified in the *type* attribute, as in type="int". Specifying

```
<vdec eid="movie_decoder">
  <client>
    <prop name="addr">a.example.com</prop>
    <prop name="filename">movie.db</prop>
    <prop name="port">5000</prop>
  </client>
</vdec>
<clip eid="clip1">
  <prop name="start-frame" type="int">30</prop>
  <prop name="end-frame" type="int">90</prop>
  <ref eid="movie_decoder" />
</clip>
<root rid="1">
  <seq eid="clip1_twice">
    <ref eid="clip1" />
    <ref eid="clip1" />
    ...
  </seq>
</root>
```

Figure 3.7: Example Usage of the <ref> Element

the type in the XML file allows the pipeline constructor to do type checking and type coercion with the corresponding attribute of the Qinematic object representing that element in the pipeline.

- The <ref> acts as a symbolic reference to another element placed elsewhere in the XML document. Because it is very common to reuse the same elements in many of the sub-trees of the graph, the <ref> element is very convenient. When the pipeline construction algorithm encounters a <ref> element, it replaces it with the element it refers to, based on the *eid* specified as its attribute. For instance, in Figure 3.7 we use the <ref> element to reuse a component identified as rat2 that was previously defined. <ref> nodes are not allowed to have children elements in the static XML representation.

Construction Process

The pipeline builder zips through the tree from one of the <root> elements (specified prior to the construction) and towards the source(s) (e.g., the < client > node in Figure 3.7). Each element encountered is added to a lookup data structure during the process, indexed by an element unique identifier <eid> so that the targets of <ref> nodes can be identified quickly.

To build the dynamic pipeline, the pipeline builder makes multiple traversals over the graph:

Parsing The static pipeline (XML document) is first parsed using a Simple API for XML (SAX) Parser, and the first intermediate representation of the graph is constructed. Nodes (vertices) are stored in various symbol tables indexed by rid for root elements, and eid for elements. Also, each node holds a list of element ids to which it is connected. The anonymous elements (those with no ids) are assigned unique ids implicitly in the process. At the end of this step, all the possible edges and vertices of the graph are known, and the static pipeline information is represented in memory as a forest.

Reference Resolution This step resolves elements referenced by <ref> nodes (verifies existence of elements) and detects cycles that may have been introduced (cyclic pipelines are rejected). All the edge endpoints that are <ref> nodes are replaced with their respective target, using the eid of the reference.

Qinematic Conversion The last stage of the process converts the intermediate representation of nodes and edges into the corresponding Qinematic elements and pads. The names of the XML elements translate to the names of the Qinematic element classes, and the appropriate element constructors are invoked with the attributes present in the XML elements. Each edge will become a connection between 2 pads of neighbouring elements. All values assigned are type checked, and appropriate error messages are returned to the user if necessary.

The current builder, like any prototype, has some limitations. Most notably is that it makes certain types of elements very awkward, if not impossible, to represent. The conversion phase of the construction will create a new Qinematic element for each visit of a vertex during the traversal, without pruning. As a result, vertices reachable from the root by more than one path like the targets of reference nodes will be cloned for each path. For example, the element with `eid="clip1"` in Figure 3.7 is defined only once but can be reached in two ways from `<root rid="1">`. It will therefore be represented twice in the instantiated pipeline, similarly with all the elements downstream from that clip. Elements such as demuxers should have the ability to output data on more than one pad, and therefore cannot be described currently. The Qinematic element-pad system however has no such restriction, so this is really a limitation of the pipeline builder only.

To resolve these issues, the prototype would need to be changed in the following ways:

- The edges should specify the names of the pads that can be linked. This would allow multiple edges between 2 given elements (i.e., multigraphs).
- The current `<ref>` nodes are useful because they allow sub-trees to be repeated in the pipeline, but it is also desirable to be able to link elements without duplicating them. A new syntax, perhaps using another tag, should allow "referencing" without duplication. At the same time this would allow directed acyclic graphs to be represented, not just trees.

3.4.2 Destruction

The destruction of the pipeline is conceptually much simpler than the construction, but is equally challenging. Once the pipeline is constructed and data has started flowing, a graceful arrest of the system requires coordination between the various elements.

Shutdown of a Qinematic pipeline is ordinarily initiated by the user (e.g., the user closes a window, or completes a key sequence). The following sequence of events will bring Qinematic to a full stop:

1. The builder releases the references to the elements it has created in the pipeline.
2. Any element whose last reference goes away will initiate its own shutdown sequence. First, it releases its own references. This usually means that it

drops auxiliary data it keeps around in memory, and it loses references to its own pads.

3. The elements are then individually finalized and their memory is reclaimed. If memory for certain elements cannot be reclaimed right away, for instance because the element still maintains network connections open, then they will be deleted asynchronously once the resources have been released.

4. The final step of the pipeline construction consists in stopping the event loop. The event loop will stop iterating once its last reference is dropped.

3.4.3 Qinematic Events

The most interesting stage in the life of a Qinematic application, the main stage, takes place between the construction and destruction. During that stage, pipelines react to various types of application defined events. In this section we shall describe the life-cycle of a Qinematic application in terms of those events.

Each Qinematic element can be in one of many possible states at any time, and the Qinematic events are the objects that cause transitions between element states to occur. The set of possible states of an element in Qinematic closely follows the QStream/Qvid state model.

CLOSED This is the state of the pipeline elements just after their construction. The elements are present in memory, they are connected, their properties are set, but connections to the server, database, or files on disk have not yet been opened.

OPENED This is the state of the pipeline elements after the files and network connections have been opened. Once opened, the length of the input stream, the number of streams, and the stream configuration (i.e., frame-rate, or codecs used) is known. Each element in the pipeline knows about its source.

SEEKING In this state, each element has set a playback position on its source. This happens when the pipeline is instructed to start playing, for instance when the "play" button is pressed on the GUI. Elements also use this state

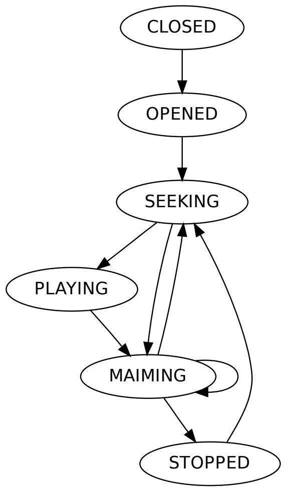


Figure 3.8: Relations Between Element States

to calculate the amount of time that will be needed to use the first frame after the start message, by consulting the timeline (see Subsection 2.2.2, and Algorithm 3.1 for more details).

PLAYING In this state, the elements are processing data according to their individual timelines.

MAIMING Elements in this state cease to process new data. Elements go in this state for two reasons: they are either preparing for a full stop, or preparing to receive a new seek position.

STOPPED Elements go in this state when the maiming operation is complete and no new seek position has been ordered for the element. The difference with the **CLOSED** state is that resources remain opened in the stopped state, and the element positions can be set again anytime with a seek.

The set of allowed events in Qinematic is closely related to those states. For each of those states, Qinematic generally provides a pair of events that will bring an element in and out of that state: a request event and its reply (response).

Here is a list of the current Qinematic events and the effect they should have on the elements pipeline. When applicable, each entry describes both the request and the reply (suffixed with "_DONE").

OPEN & OPEN_DONE Asks the element upstream to open a stream. On success, the response will contain a description of the stream that was opened (i.e. codec, length, and dimensions). This moves the element from a closed state to an opened state.

SEEK & SEEK_DONE Asks the element upstream to stop what it is doing and set new playback parameters (position, speed, duration, and direction). To minimize costly round trips between client and server, the operations that stop and restart are always pipelined. The response will contain information about the new playback position[5]. When received, an element moves to state MAIMING if it was PLAYING and makes the new seek position effective. When the new seek position is set, it sends SEEK_DONE and waits for a START event (unless it has already arrived by the time SEEK_DONE is sent).

MAIM & MAIM_DONE Sent upstream to request that no more data be sent. This is generally used to stop the pipeline temporarily until a new SEEK request comes, or just before the application comes to a permanent stop (e.g., before quitting). The element receiving this message moves into the MAIMING state. Elements in the maiming state can have one operation pending, again, to minimize round trips.

EOF Sent downstream to notify that there is no more data to send according to the last SEEK event.

START The start event allows delaying the time at which upstream should start sending data. The START is sent after a SEEK message, either after or before the SEEK_DONE response comes back. When an element wishes

[5]Depending on which playback mode is enabled (e.g., frame-accurate, or key-frame only), the position in the response may be different than the one requested.

to receive the data as soon as it is ready, this request is pipelined with the SEEK. Qinematic uses the delay feature only to synchronize arrival of data between streams.

CLIENT_CONSUME This event is sent upstream to indicate that a given amount of data (buffers) has been processed completely downstream. This is only used when streams are playing in a mode we call “best-effort”, where dropping data along the pipeline is not allowed, but data can arrive late. Best-effort mode would be used for instance when video is saved to disk (and quality needs to be at its maximum) instead of played on screen. This event throttles the arrival of data on the client to prevent overallocation of memory resources.

This relatively small set of events has been sufficient in Qinematic to convert all of the existing Qvid modules into Qinematic elements.

Event Propagation

Qinematic follows conventions to orchestrate elements in the pipeline. These conventions define where and how events can travel between elements in the pipeline. What becomes of the events once they enter an element is not, and cannot, be monitored by Qinematic.

Firstly, care must be taken to never block the event loop thread when receiving an event, or when receiving a reply to an event. Secondly the elements must expect to receive and send out events in accordance to the event-type and state rules described earlier. For instance, elements must allow multiple seeks in a row to be received.

Elements register callbacks on the pads they own to receive request events, and use methods on the qine_pad to reply. When an event is received or generated, the element performs one of the following actions:

1. Reception: The element can ignore an event, and simply *push* it out.

2. Reception: The element can *absorb* the event if the event is a downstream event such as EOF by choosing to not pass it.

3. Reception: The element can *subscribe* to a request event to signal an interest in the response. This is done by placing a hook (closure) on the request event, which will be invoked later when the response for that request is generated. For instance, placing a hook on a SEEK event will allow an element to also receive the corresponding SEEK_DONE event

4. Reception: The element can hold on to a received event until some condition is satisfied. This will delay the propagation of the event (or a response to it).

5. Generation: The element can create a response event, attach it to an existing request, and *publish* that response event. Qinematic will invoke the closure(s) attached to the request object. These closures are invoked in the opposite order they were registered, which gives room for a lot of flexibility in terms of controlling the moment at which other elements are notified.

6. Generation: The element can generate a request event, *subscribe* to it so that it receives the response, and *push* it out. This also allows creating complex chains of events between elements. For instance, an element with two upstream neighbours and one downstream element could receive a SEEK from downstream, create two new SEEK events to forward upstream, and then wait for both of the upstream SEEKs to finish before responding to the original.

The act of *pushing* an event on a pad always sends that event to the corresponding peer on that pad[6], for both downstream and upstream events. This is the default mode of travel of all request events.

Reply events on the other hand travel solely from one element to the other via closures that have been registered. Hence, during a *publish* operation, reply events will spring into only the elements that have shown interest in the corresponding request, i.e., the *subscribers*.

In this section we have covered the various types of events that Qinematic offers, and the manner in which those events are propagated between elements. In the next section we shall give an overview of the life-cycle of buffers in Qinematic.

[6]If the pad is not connected, then the event is silently garbage collected.

```
struct QineBuffer {
        /*< public >*/
        QineBufferType   type;
        QineStructure   *structure; /* type specific extra info */
        gpointer         data;      /* type specific data pointer */

        /*< private >*/
        QsfCallback      destroy;
        QsfCallback      consume;
        gboolean         consumed;
        volatile gint    refcount;
};
```

Figure 3.9: The QineBuffer structure.

3.4.4 Qinematic Buffers

In addition to events that direct the flow of the pipeline, Qinematic provides *buffers* to encapsulate data. Buffers always flow downstream and are composed of a header and payload.

The header contains type information on the payload, a reference count, as well as consumption and destruction callbacks.

The destroy callback is invoked when the last reference to the buffer is dropped, and was setup by the creator of the buffer. It is responsible for freeing memory associated with the buffer. Small buffers are common, and their data is often part of the same allocation as the buffer header itself.

The consume callback is similar to the destroy callback, but is only used in best-effort mode. It can be used to notify upstream that the buffer has been completely processed. It is separated from the destruction operation to allow destroying buffers without sending the notification (for instance this would be used during a maim operation).

Two main types of buffers are currently in use in Qinematic: image and audio buffers. Each of those can be divided further in subtypes for each of the various decoding stages traversed. The rest of this section will focus on a description of the image buffers.

```
struct QineImage {
        /* Memory management */
        gint                    ref_count;
        QsfCallback             release;

        /* Transformation context */
        GQuark                  transform_id;
        QineStructure          *transform_ctx;
        GList                  *deps; /* input images */
        QsfCallback             transform;

        /* Final image */
        gboolean                ready;
        QineImgFmt              fmt;
        guint                   w;
        guint                   h;
        gdouble                 aspect;
        QvidImage               data;
};
```

Figure 3.10: A QineImage transformation context. It contains information about the transformation operation to apply and the input images.

More than Images: QineImages

Qinematic pipelines are able to represent transformations on video using special objects called QineImages. QineImages travel inside data buffers, but provide a lot more than mere data. They are effectively closures that will generate video images on demand.

Computationally expensive operations can be deferred inside QineImages until their result needs to be displayed or must take part in the computation of another image that must be displayed, etc.

Transformations accumulate on the QineImage as it traverses the pipeline. At the end of the traversal, when the QineImage reaches a sink, it has encapsulated all the work that must be performed to produce its final representation. That is, the image that the viewer will see.

This is a great feature to possess in a quality-adaptive scenario, as it allows work to be prioritized better. Firstly, effort can be spent only on QineImages that reach the end of the pipeline. Secondly, deferring the computational work until the last moment may allow better informed decisions to be taken than if they had been taken on arrival of the buffer at the element.

QineImages contain a transformation id that uniquely identifies the transformation by name, as well as a transformation function pointer. Each element that applies effects to QineImages can define its own identifier.

The structure also holds a list of references to other QineImages to be used as input to the transformation. If extra state is needed, the QineImage conveniently also carries a QineStructure which can hold typed key-value pairs. The images are reference counted, and can be included inside dependency trees of images. However, cyclic dependencies are not prevented by Qinematic so elements should exert extra caution.

A QineImage must always provide information about the final image that will result if the transformation is applied (this is specified in the last block of Figure 3.10). This is necessary for checking that transformation operations are compatible.

Unfortunately, the current transformation functions are currently not allowed to complete asynchronously. The convention is that once invoked, they should perform their computation, and toggle the "ready" flag on the image. They have been designed for video effect operations that are non-adaptive, such as colour conversions, cropping, and blending. If elements wish to perform transformations whose quality will vary depending on time and available resources, they should be integrated as part of the Qvid timeline.

3.5 Summary

Qinematic is a framework to process video in sophisticated pipelines: elements encapsulate the processing code and are connected together using pads. This processing abstraction is simple and is employed by most current media frameworks. It is simple, but is generic enough to easily model common video editor features such as separating movies into clips, sequencing clips, adding effects, and cross-fades. By leveraging Qvid's timeline and the execution model of the QStream framework, Qinematic also preserves the ability to adapt its output quality during streaming. The timeline code assigns timestamps to the various buffers, which allows their prioritization. Qinematic pipelines make use of one timeline per input stream, and will combine the information of all timelines to control the overall execution of the pipeline. That is, given a start point to play a video, Qinematic schedules the arrival of buffers in the various stages of the pipeline and can thus coordinate the overall flow of data. Regarding the creation of pipelines, Qinematic provides tools that will parse XML files containing pipeline definitions, will assemble the pipeline, and then render the output of the pipeline in a GUI. The GUI supports all the advanced navigation features commonly found in local editors, namely fast forward and rewind (in speeds that range from 0.5x to 128x), single-step, and last but not least scrubbing. The Qinematic framework is work in progress, and is still missing many desirable features found in today's professional tools. However, our ability to extend Qvid enough to what we currently support constitutes a strong indication that Qinematic is moving in the right direction.

Chapter 4

Related Work

Qinematic combines quality adaptation aspects and extensibility to create a platform with which amateurs can collaborate in editing video. Qinematic embraces the Internet video streaming model as it allows distributed teams to work on common distributed content. This is in contrast with most current professional editing utilities which usually manipulate video content stored locally or on LAN network storage.

GStreamer [6] is a popular open-source multimedia framework with a plug-in based architecture running on a variety of platforms including Linux, Windows, and OS X. It is a very successful framework that is used inside many audio and video applications. GStreamer offers a rich panoply of plug-ins (over 150) which contain elements that process multimedia data in various ways. Examples of elements are video and audio codecs, container file format readers and writers (.avi, .mkv, etc.), and various drivers for audio and video (v4l, alsa, x11, etc.). GStreamer also provides mechanisms for plug-ins to communicate and negotiate common supported formats, and provides a pipeline architecture. Users can write complex application pipelines using the various plug-ins, and programmers can easily integrate legacy code (e.g., codecs) into GStreamer, by following their API, or by using one of the higher level language interfaces.

We have considered augmenting the GStreamer model to support adaptation. This would allow us to leverage all of the pipeline construction features, a wealth of plug-ins, and the interoperability between formats. Unfortunately, there are many fundamental differences between the QStream and GStreamer code bases that made the integration of the two frameworks impractical. The end result is that Qinematic is a unique blend between extensibility provided by GStreamer-like pipeline structures (which GStreamer in turn borrows from DirectShow), and quality-adaptation oriented execution.

Firstly, the thread models are different. GStreamer uses a multithreaded architecture and offers two modes of operation to displace data in the pipeline: pull-based or push-based scheduling. Pulling means that elements run a thread to control the flow of their input, and pushing means that an element is driven completely by its upstream elements. Different sections of the pipeline can use different modes in different threads, provided that they communicate via message queues or a shared bus. Qinematic on the other hand is single threaded and event-driven, and has no immediate need for the pull-based mode, given that it is the timeline that controls the arrival of data in the elements.

GStreamer is designed to be embedded inside an application (like an mp3 or movie player), not the opposite. It is also a framework, which means that the application is written *with it*, not *around it*.

More importantly, GStreamer is very limited in terms of quality adaptation. For controlling the rate in the pipeline, we must rely on QoS events, which record statistics on differences between timestamps of buffers on arrival and the clock. The claim is that those events could be used to provide some form of feedback loop with elements upstream to adjust their processing. However, the application of feedback loops is of dubious utility in video streaming settings as they require complex resource estimation. In order to be efficient, the adaptation mechanisms must be able to estimate the CPU time required to process the video data, and estimate the CPU time that will be available, and then derive control decisions that will lead to the best final video quality [16]. Because Qinematic uses Priority-Progress, such complex estimates are not needed to adapt output quality.

It is important to mention that GStreamer also has many auto-configuration features that Qinematic lacks, such as video format auto-detection, grouping of sets of elements into bins for easier pipeline management, and special meta-elements such as generic decoders that can transform into specific decoder elements based on the sources and sinks connected to them. Qinematic is in its early stages, and we simply reached for the simplest solution, due to time constraints.

Pitivi [11] is a new project which combines existing GStreamer elements with GStreamer's python bindings, to allow non-linear editing in a friendly desktop user interface. Being written on top of GStreamer, the project also suffers from the same deficiencies with regards to quality adaptation. In the hope to simplify writing and prototyping of new elements, Qinematic also provides Python bindings to prototype the necessary elements to build interesting video editing software.

Kaltura [7] is a new platform that shares our goal of providing an open source video platform. The Kaltura advanced editor (KAE) provides a web-based timeline-based video editing and sharing platform. The platform itself is open-source, but uses proprietary Adobe Flash technology to stream videos and show interfaces. Similarly to Kaltura we aim to expose Qinematic using a web interface.

Qinematic also borrows ideas from the popular image processing solution Core Image [4] for lazy processing of image buffers. Core Image lets developers define a series of image filters, called *Image Units* as image convolution kernels, compositions, or deformations to apply on source images. Core Image dynamically computes the processing pipeline for each pixel and then recombines the results in the final image. The platform has the capability to compile the image units into shader language assembly to take advantage of available accelerated graphics hardware. The images that Qinematic provides, QineImages, have similar properties, as pipelines of transformations can also be applied to them, and these pipelines can be introspected to perform optimizations (e.g., for doing algebra on image transformations). This optimizer, as well as the compiler are currently missing from Qinematic, and are part of future work.

Aside from multimedia frameworks, it is also worth mentioning how our approach compares to the most popular open-source video players, MPlayer [9] and VLC [1], which take a different view on the need for quality adaptation. They are both highly optimized applications that provide large sets of features, codecs, and output drivers written by talented and devoted teams of developers. However, even there, adaptation goes only as far as adjusting frame rate. Their philosophy is that you don't need adaptation if everything is done quicker. In practice however, this works well only until you play very high quality content, or multiple videos simultaneously [16]. Qinematic, on the other hand, doesn't have as many features, but reaps the benefits of adaptation for CPU, network and storage.

In this chapter we have reviewed existing multimedia frameworks similar to Qinematic, and compared their features in terms of extensibility and quality adaptation. The Qinematic platform is unique in its genre as it combines the plug-in/pipeline models of popular frameworks and state of the art quality adaptation techniques.

Chapter 5

Conclusions

We have built Qinematic, a quality-adaptive video editing framework. Qinematic adopts the event-driven model of QStream necessary for quality-adaptation, and leverages the video functionalities of Qvid, a video player also built with the same model. The client in Qinematic uses the Qvid timeline to prioritize data flowing in its pipeline, which consequently allows QStream's event loop to do the orchestration of the processing in all elements at once.

Our hope with Qinematic is to see it being used in a distributed video-authoring platform, to help teams of amateurs elevate the quality of their collective work. We are targeting a small set of features that are common to many editors we find today: clips, effects, and transitions. In order to do that, we had to extend Qvid's static model to something more dynamic which would allow complex video processing pipeline configurations. We have looked for inspiration in existing media frameworks, and found that they couldn't be used directly because of large differences in design and implementation. These frameworks have elegantly solved the problem of modularity and flexibility, and we believe that QStream and Qvid solve the problem of adaptation. Qinematic stands in the middle: extensibility and quality-adaptation.

Qinematic was constructed over a very short period of time, and time constraints have forced us to abandon features part of the initial plan, unfortunately. From the initial three goals of providing support for clips, intra-clip effects, and inter-clip effects (transitions), only the first two are supported in the implementation as of this writing. The third is documented in this thesis in the form of algorithms only. Implementation for XFADE has started but its completion is part of the future work.

Several smaller side projects related to Qinematic have been completed during the course of the masters and not been included in this thesis but are still worth mentioning. Originally, we had decided that Qinematic would be built using the rich Internet application model. Hence, one of the early experiments we conducted was to see how well the existing Qvid player could be adapted to work from a web interface. At the time, the video-editing on-line services we studied (Kaltura, JumpCut, YouTube Remixer), were offering very similar flash based services, locked down inside an applet which took only a small portion of the available viewing space. For the experiment, we have built a fully compliant NPAPI to provide access to video content. Shortcomings of the API, namely the inability for plug-ins to freely access information in the Document Object Model (DOM) of the enclosing page at will, would have severely affected the user experience. Furthermore, noticing that some of these sites were losing popularity, e.g. JumpCut, we decided to focus on different aspects of Qinematic.

Another point worth mentioning is that Qinematic's default video output driver uses OpenGL, to allow close visual inspection of the output produced with zoom-ins, and zoom-outs, and also out of convenience to avoid extra conversions when working with RGB image formats. Generally speaking, GL makes many operations very simple to implement, like linear transformations and overlays of text and image.

We concede that Qinematic is still quite far from having all the features we find today in professional video editing tools produced by Apple, Adobe, or Ulead. Video editors are hard, especially when they are distributed. Many improvements are still needed, but we are convinced that what we have in Qinematic is the first step forward towards a collaborative video editing platform.

Bibliography

[1] VLC: The cross-platform open-source multimedia framework, player and server. Homepage: http://www.videolan.org/vlc/.

[2] AviSynth. Homepage: http://avisynth.org/.

[3] YouTube Surpasses 100 Million US Viewers. http://www.comscore.com/Press_Events/Press_Releases/2009/3/YouTube_Surpasses_100_Million_US_Viewers, March 2009.

[4] Developing with Core Image. http://developer.apple.com/macosx/coreimage.html.

[5] Microsoft DirectShow API. http://msdn.microsoft.com/en-us/library/dd375454(VS.85).aspx.

[6] GStreamer: Open Source Multimedia Framework. Homepage: http://gstreamer.freedesktop.org/.

[7] Understanding Kaltura - Open Source Online Video Platform. http://www.kaltura.org/understanding-kaltura-open-source-video-online-platform.

[8] MediaSilo. Homepage: http://www.mediasilo.com.

[9] MPlayer. Homepage: http://www.mplayerhq.hu/.

[10] Gecko Plugin API reference. https://developer.mozilla.org/en/Gecko_Plugin_API_Reference.

[11] PiTiVi: An open-source video editor for Linux. Homepage: http://www.pitivi.org/.

[12] QStream: Quality-Adaptive Media Streaming. Homepage: http://www.qstream.org.

[13] D. L. Eager, M. K. Vernon, and J. Zahorjan. Bandwidth Skimming: A Technique for Cost-Effective Video-on-Demand. In *Proceedings of IS&T/SPIE Conference on Multimedia Computing and Networking (MMCN)*, pages 206–215. SPIE, 2000.

[14] C. Krasic and J.-S. Légaré. Interactivity and scalability enhancements for quality-adaptive streaming. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 753–756, New York, NY, USA, 2008. ACM.

[15] C. Krasic, J. Walpole, and W.-c. Feng. Quality-adaptive media streaming by priority drop. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 112–121, New York, NY, USA, 2003. ACM.

[16] C. Krasic, A. Sinha, and L. Kirsh. Priority-progress CPU adaptation for elastic real-time applications. volume 6504, pages 65040G.1–65040G.12. SPIE, 2007.

[17] B. Novikov and O. Proskurnin. Towards Collaborative Video Authoring. In *Advances in Databases and Information Systems, 7th East European Conference (ADBIS 2003)*, pages 370–380, 2003.

[18] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[19] L. Rutledge. Smil 2.0: Xml for web multimedia. *IEEE Internet Computing*, 5(5):78–84, 2001. ISSN 1089-7801.

Appendix A: Qinematic in Action

This appendix will provide examples and instructions to compile and run Qinematic. It also details how the code for the project is divided, and describes the purpose of each of the project's components.

A.1 Obtaining the Code

The source code for Qinematic is packaged with the rest of the QStream framework in the Qvid component. The homepage of the QStream project [12] has details on how to checkout and install a copy of the code. The essential installation can be reproduced by executing a program equivalent to Figure A.1. Assuming you have the necessary dependency libraries, the Qinematic folder should be built along with all of Qvid.

A.1.1 File Organization

The core of the Qinematic code is located in the Qvid component of QStream, in qvid/src/qinematic/. And some files shared by Qinematic and the rest of Qvid have been placed in a different directory to avoid shortcomings of autotools to track dependencies across directories for generated header files (i.e. files listed under BUILT_SOURCES variables). Those files contain declarations for QineImage constants, macros used to define new objects, as well as declarations for basic enums. They are located in qvid/src/common/. All files from the Qinematic project are prefixed with "qine_".

The Qinematic source tree is subdivided into components, each in their subfolder.

```
#!/bin/bash
set -e
mkdir -p ~/src/
cd ~/src/
if [[ ! -d "$BRANCHNAME" ]]; then
  HPAGE="http://svn.qstream.org/qstream/branches/"
  svn checkout "$HPAGE"/"$BRANCHNAME"
else
  cd "$BRANCHNAME"
  ./build-all.sh libpcl-qstream qsf qsf-example \
                 xvidcore-qstream gscope qmon2 \
                 mxtraf2 qps qggen qvid
fi
```

Figure A.1: Essential steps to download and compile QStream and Qinematic. The environment variable BRANCHNAME should be set to the most recent branch of QStream.

core/ Contains the set of base classes of Qinematic, for elements, buffers, and pads and associated header files.

elements/ Contains the Qinematic elements written in C. Some of these elements are wrappers around Qvid components so that they can be used in Qinematic's pipeline, while others are purely for video editing operations or effects.

defs/ For a structure/object from Qinematic to be exposed in Python, a definitions file must be created. An excerpt of a definitions file for an object called a QinePipeInfo object is shown in listing Figure A.2.

python/ This directory contains files that get compiled into the Qinematic Python qine module. The files that compose that module are created from the files in the defs/ folder. A special source-to-source compiler from the python-gobject library is used to produce C code for Python extension modules only from the information in a definitions file.

override/ In some corner cases, the above compiler cannot produce correct code. To cover for these cases, the compiler allows reading in special files that

```
(define-pointer PipeInfo
  (in-module "Qine")
  (c-name "QinePipeInfo")
  (gtype-id "QINE_TYPE_PIPE_INFO")
  (fields
  )
)

(define-method add_root
  (of-object "QinePipeInfo")
  (c-name "qine_pipe_info_add_root")
  (return-type "none")
  (parameters
    '("const-gchar*" "root_id")
    '("const-gchar*" "root_elem_id")
  )
)

(define-method destroy
  (of-object "QinePipeInfo")
  (c-name "qine_pipe_info_destroy")
  (return-type "none")
)
```

Figure A.2: A portion of a sample definitions file for a QinePipeInfo object. The definitions define the mapping between symbol names, and types between the language of the bindings and the C implementation.

will replace the compiler's output of functions specified. This folder contains these files. In most cases, the overridden methods are very close to the original output from the compiler.

tests/ A collection of sample XML pipeline definitions to use for testing of Qinematic features.

A.2 Building Qinematic

Certain steps in Qinematic's build system require some explanation. Qinematic's build is complicated because of the bindings it packages for Python, and its layer of GObject code: some small extra steps need to be taken, notably code generation steps.

Firstly, most header and C files in Qinematic are parsed with tools such as glib−mkenums from the `glib2.0` library. These tools look for enum or struct definitions inside header files, and generate corresponding enum type declarations that can be introspected at run-time. For example, if we take the enum declaration for a QinePad's direction:

```
typedef enum {
        QINE_PAD_SRC,
        QINE_PAD_SINK,
} QinePadDirection;
```

When Qinematic runs glib−mkenums on that declaration, the following accessors are automatically created:

```
...
GType qine_pad_direction_get_type (void);
#define QINE_TYPE_PAD_DIRECTION (qine_pad_direction_get_type ())

/* ... AND ... */

GType
qine_pad_direction_get_type(void) {
  static GType type = 0;
  if (type == 0) {

    static const GEnumValue QinePadDirection_values[] = {
      { QINE_PAD_SRC , "QINE_PAD_SRC", "src" },
      { QINE_PAD_SINK , "QINE_PAD_SINK", "sink" },
      { 0, NULL, NULL } /* sentinel */
    }; /* GEnumValue QinePadDirection_values */
    type = g_enum_register_static("QinePadDirection",
                                  QinePadDirection_values);
  } /* if */
  return type;
}
/* qine_pad_direction_get_type */
```

Qinematic uses another pass of code generation for its Python bindings code. Definitions files, such as the ones located in the defs/ sub-directory of the source tree are generated by a compiler called h2def.py from the python-gobject library. It parses C and header files to find structures and GObject definitions, which it identifies because of the naming conventions used in GObject code. It then creates language-neutral definitions files, such as the one in listing Figure A.2. The output is most often correct, but sometimes a minor amount of manual adjustment is needed for instance to add or hide methods from C in the higher level language.

The definitions files are passed as input to another compiler that will generate C code to be compiled in a Python extension module for Qinematic. This is a simple compiler, called codegen.py that is also provided by the python−gobject library. The one used in Qinematic is slightly different. The Qinematic codegen.py generates code for setters in addition to getter functions. This reduces the amount of code that has to be overridden to access (get and set) fields of simple structs from Python.

Each definitions file maps to a single Python extension module file. Once all definitions files have been converted, their symbols are imported into a single module called qine, which is initialized via the following function:

```
static gboolean
_qine_python_module_init (void)
{
        PyObject *m;
        PyObject *d;

        load_pygobject();

        combine_function_tables(&qine_module_functions,
                                &py_qine_pipeline_functions,
                                &py_qine_object_functions,
                                &py_qine_structure_functions,
                                &py_qine_buffer_functions,
                                &py_qine_event_functions,
                                &py_qine_pad_functions,
                                &py_qine_element_functions,
                                &py_qine_element_factory_functions,
                                NULL);

        m = Py_InitModule ( "qine", qine_module_functions );
        d = PyModule_GetDict (m); /* borrowed */

        py_qine_buffer_add_constants         (m, "QINE_");
        py_qine_event_add_constants          (m, "QINE_");
        py_qine_pad_add_constants            (m, "QINE_");

        py_qine_pipeline_register_classes    (d);
        py_qine_object_register_classes      (d);
29          py_qine_structure_register_classes   (d);
30          py_qine_buffer_register_classes      (d);
31          py_qine_event_register_classes       (d);
32          py_qine_pad_register_classes         (d);
33          py_qine_element_register_classes     (d);
34          py_qine_element_factory_register_classes(d);
35
36          _PyQineModule = m;
37
38          if (PyRun_SimpleString("import␣sys\nsys.path.insert(0,'')")) {
39                  PyErr_Print();
40                  return FALSE;
41          } /* if */
42
43          helper.module = PyImport_ImportModule("qine_helper");
44          if (!helper.module) {
45                  PyErr_Print();
46                  return FALSE;
47          } /* fi */
48
49          /* borrowed ref */
50          helper.module_dict = PyModule_GetDict(helper.module);
51
52          if (PyErr_Occurred()) {
53                  PyErr_Print();
54                  return FALSE;
55          } /* if */
56
57          if (!fill_helper_functions()) return FALSE;
58
59          return TRUE;
60  }
61  /* _qine_python_module_init */
```

Lines 9 to 20 load the symbols generated by the compiler and use them to populate the qine module. Python classes are registered with the Python GObject type system in lines 27 to 34, and then on line 43, a helper Python module is executed. The file qine_helper is a Python script used to bootstrap additional classes, which we will cover in the next section.

A.3 Running Qinematic

The Qinematic framework provides one application that allows the user to specify a pipeline in an XML file and connect that pipeline to a graphical player to visualize the processed information. In this chapter we will describe the functions of this application through the pipeline shown in Figure A.3.

The main application from Qinematic is called “qinematic” and can be found inside a working copy of the repository under the directory <reporoot>/qvid/src/qinematic/. To play the pipeline from Figure A.3, you can invoke qinematic it as follows:

```
./qinematic tests/test_suite.xml –R [rid]
```

The first argument is the path to the pipeline definition file. This file will be read and processed according to the rules explained in Subsection 3.4.1. The −R option argument to qinematic allows selecting which of the root elements from the pipeline will be selected for viewing in the GUI. This means that for this file, roots “localmovie”, “localclip”, “bwclip”, “remoteclip”, and “3clips” can be specified. If the −R option is omitted, then qinematic defaults to the first root element it finds, starting from the top of the XML document. Also, the list of roots that can be selected can be conveniently retrieved with the −−list−roots command line option.

If we start a QStream server on host “somehost” to listen for connections on port 5555, we should be able to connect to it using qinematic and the “remoteclip” root id. This will ask the server at somehost:5555 to serve a video source file called “movie.db” to a pps client. Because the element “remoteclip100” is configured with `"start-frame"`==2000, and `"end-frame==2099"`, the full movie that qinematic will show will contain 100 frames in total.

The application will check the validity of the pipeline and command line arguments and should start a player window similar to the one shown in Figure A.4. The window shows the first frame of the movie, and waits for user input. The Qinematic player will respond to the keys listed in Table A.1.

For additional instructions on using and debugging the programs from QStream and Qinematic, you should refer to the READMEs, and the API documentation inside the various components of the source tree.

```
<pipeline>
  <vdec eid="local"><fi><prop name="filename">movie.db</prop></fi></vdec>
  <vdec eid="remote">
    <client>
      <prop name="filename">movie.db</prop>
      <prop name="addr"    >host.example.org</prop>
      <prop name="port"    >5555</prop>
    </client>
  </vdec>
  <clip eid="localclip100">
    <prop name="start-frame" type="int">2000</prop>
    <prop name="end-frame"   type="int">2099</prop>
    <ref eid="local" />
  </clip>
  <bw eid="bwlocalclip100">
    <ref eid="localclip100" />
  </bw>
  <clip eid="remoteclip100">
    <prop name="start-frame" type="int">2000</prop>
    <prop name="end-frame"   type="int">2099</prop>
    <ref eid="remote" />
  </clip>
  <seq eid="3clips">
    <ref eid="remoteclip100" />
    <ref eid="bwlocalclip100" />
    <ref eid="localclip100" />
  </seq>

  <!-- Play a local movie (like qvid_play) -->
  <root rid="localmovie"><ref eid="local" /></root>

  <!-- Play a clip of a local movie -->
  <root rid="localclip"><ref eid="localclip100" /></root>

  <!-- Play a BW version of localclip100 -->
  <root rid="bwclip"><ref eid="bwlocalclip100" /></root>

  <!-- Play a clip of a movie stored remotely -->
  <root rid="remoteclip"><ref eid="remoteclip100" /></root>

  <!-- Play 3 clips: local bw and remote, in sequence -->
  <root rid="3clips"><ref eid="3clips" /></root>
</pipeline>
```

Figure A.3: This pipeline contains various root elements that can each be viewed inside the Qinematic player. We will use this pipeline as an example to demonstrate how the Qinematic application works. You can find a copy of that file in <reporoot>/qvid/src/qinematic/tests/test_suite.xml.

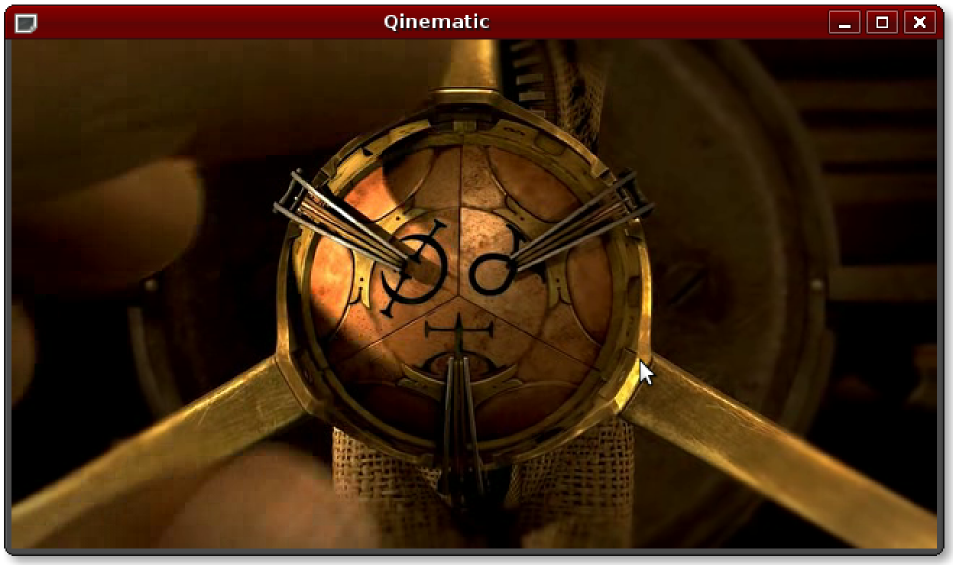


Figure A.4: The Qinematic GUI's Player Window after starting qinematic on root "remoteclip".

Key(s)	Action
Up and Down	Changes playback speed from "frame accurate", to "GOP step mode" (key-frame only), to speeds 0.5x, 1.0x, 2.0x, 4.0x, . . . , up to 128x.
Left and Right	Reverses playback direction.
F	Toggles full-screen mode.
+ and -	Zooms in and out on the video.
Y,G,H, and J	Translates the visible portion of the video while zoomed in.
0	Resets the zoom level and centers the view.
Space	Pauses/Starts playback. In step mode, steps by one frame in current direction.
Page Up/Down	Advances by one minute forward or backward in time.

Table A.1: List of key to action mappings used in the Qinematic player window.