State of the Art Study

António Ferreira #45356
Francisco Cunha #45412

ABSTRACT

Building intuitive multimedia exploration and visualization systems isn't a simple task. Not only are we faced with the challenge of building a fast, efficient and reliable querying engine, but also with the HCI side of the spectrum – building easily accessible, quickly learnable and highly scalable user interfaces.

There have been many studies showcasing different approaches to all of these problems - particularly in academia – and so we'll start this short paper by summarizing what we found. Afterwards, we'll propose the initial specification for the project we have in hands.

Keywords

Video Interface; Metadata; Querying Engine; Resource Overview.

1. INTRODUCTION

The goal of our project is to semi-automatically build a video timeline that can be used for playing or further editing videos. Like every video player we will have to implement some common features like: a video search engine, a video player and a video archive, all distributed across a pleasant interface. In order to achieve this, we will discuss in this paper the state of the art in these topics, which will help us understand and improve our knowledge to build this kind of systems.

2. POWERING UP THE QUERY ENGINE

Finding the best ways to power up browsing engines in the context of multimedia systems is the subject of a lot of research. To design and implement a fast, flexible and reliable retrieval system that may be consulted with high speed is a real challenge, because ultimately, achieving fine-grained results **[1]** without requesting too much effort from the user is difficult, especially when we want to reach high levels of accuracy. In this section, we'll briefly discuss how both metadata-based and feature extraction approaches can contribute to better querying results.

2.1. Using Metadata

Metadata has many different definitions, ranging from elaborate predicates *("(…) information about data that enables intelligent, efficient access and management of data"*) **[5]** to simple ones *("(…) is data about data"*). Much of the previous work on multimedia-based retrieval systems focuses on complex structuring (*e.g.* organized, hierarchical metadata) and/or on different types of metadata (attributes, annotations and automatic classifications).

2.1.1. Hierarchical Metadata: Case Study

In their paper **[3]**, the authors propose the usage of hierarchical organized metadata, in order provide a fast and efficient way to search and view videos from a video library. To achieve this, they focus on a video metadata cataloging system which would provide "*various query methods such as query-by-feature, query-by-definition, query-by-keyword and structural-browsing*". With this proposal, the hierarchical metadata will be stored in a relational database, which will facilitate the usage of these queries in the search engine.

2.1.2. Attributes, Annotations and Automatic Classifications

One of the most widely used techniques used in dealing with multimedia-base searching is using attributes. Whilst common attributes such as time and location can be helpful for information retrieval and categorization, they do not describe the content of the data itself. Thus, content-based browsing is pretty much impossible when relying simply on attributes.

Annotating data is a different approach that can actually turn out to be quite effective in terms of data browsing and retrieval. However, because it is essentially a manual injection of metadata, doing it for large data sets would be a rather daunting task.

In an attempt to make the former task a little more bearable, automatic classification was born. This is the process of generating annotations automatically. Whilst a great idea in theory, often coarse-grained classes are generated, making this technique not so appropriate when we need finer-grained separation.

2.2. Feature Extraction for Content-Based Similarity

Feature extraction presents a different way to power up multimedia querying systems. The implementation details will, of course, vary from author to author, but the high-level process is roughly equal: segment the available multimedia content into small chunks and have each chunk be represented by a distinct set of video and/or audio (depends on the content) features. This is a pre-processing step that is present in most content-based similarity search engines and is useful because it allows us to further cluster the segments based on their similarities, using techniques such as the Rock algorithm or the so popular K-Means. The clustering step itself may also be optimized. As seen in the research paper "Video Exploration: From Multimedia Content Analysis to Interactive Visualization", the authors improve their clustering generation and classification step with supervised learning techniques. This is not the case for VFerret, the search tool for continuous archived video which we will analyze now.

2.2.1. VFerret: Case Study

In this section, we'll go through a high-level walkthrough of the feature extraction process that happens behind the scenes in VFerret. As we stated early, it's essentially a search tool for continuous archived video, built it's important to notice that it was built with very specific goals in mind:

- Combine content-based similarity search with the above-mentioned annotation/attribute-based search in retrieval tasks.

- Use as little metadata as possible, in order to make the multimedia storage scalable and lightweight.

The whole tool is composed of two major sub-systems, the input processing engine and the query processing engine.

2.1.1.1. Input Processing Engine

This is the engine responsible for running important data pre-processing tasks. After the initial video segmentation phase, each resulting segment will go through an extraction phase.

The first important task executed by the input processing engine is attribute extraction. Why? Because typically, the querying (explained later on) starts with an attribute-based search. Usually this attribute will be a timestamp or time range, as people naturally anchor events with time. When a certain event has a very precise date, then this timeline-based search will prove to be very precise and effective. Otherwise, a bounded range is still reasonable.

Alongside, for each segment, the system extracts a set of visual features that better represent it. How is this done? For each clip, one frame is extracted for every X seconds, where X is some constant. Then, that frame's RGB is converted to HSV (which is better for measuring human perceptual similarity). Finally, the images are compared for similarity, based on central moments of color distribution. A similar approach is applied to the audio features.

Now, every clip's visual/audio features are merged and stored, generating a single feature vector for each clip. Having done this, the input processing step of the VFerret engine is complete.

2.1.1.2. Query Processing Engine

The query processing engine is triggered by user input and uses attributes and features extracted from the input processing phase to perform a similarity search, and ultimately get back to the user with its best results. As stated above, this step will typically start with an attribute-based search.

Resulting clips from this search are then clustered (using the k-means algorithm), based on their similarity for the merged features. These clusters are formed based on feature vector comparison.

To follow, a representative video clip of each cluster is displayed, so that the user can quickly scan through them. He/she will then proceed to select a cluster whose representative video appears to have similarities to the clip he/she is looking for. The selected clip shall be called the query clip.

Finally, the system will use the provided query clip to perform a distance-based search (using the k-NN algorithm) to find all the clips that are similar to it, returning a ranked list of results.

3. USER INTERFACES FOR DYNAMIC VIDEO BROWSING – EXPLORATION AND COMMON PROBLEMS

We'll now talk briefly about user interfaces in the context of multimedia systems, namely how to effectively display this type of content and what a very common UI pitfall is.

3.1 Interfaces for Video Content Resource Overview

The paper from Viaud *et al.* **[2]** shows different how different interfaces can achieve practical layouts for video content exploration. To start, the Stream Explorer is showcased, an interface with two spirals that represent the timeline. Those spirals are also segmented by color, which separates the entire video stream. It's also presented to us the Collection Explorer, a graph-model-based document gatherer that gives users an overview of the size and richness of the resources available in real-time. It achieves a funky but intuitive layout thanks to an energy force model algorithm that is backing it up.

3.2 Getting Your Sliders Right

It's a known problem that sliders and scrollbars do not scale to large document sizes nor to continuously smaller screen sizes. This is because:

- Even the smallest unit to move the slider might be too big.

- Moving the slider continuously results in a jerky visual feedback.

- For videos there are no comparable, fixed units (like there are, for instance, pages in books).

The state of the art in this area is exciting, as many attempts at solving these problems have been proposed. It gets even more complicated when we dive into the realm of mobile interaction design, where the use of additional on-screen widgets (e.g. to control distance granularity) becomes overwhelming for a user in such a small screen.

Focusing on desktop software, research **[4]** showcases many approaches to solving this problem - many of which you've seen in popular video editing software. For instance, the *ZoomSlider* allows the user to change the slider scale even while skimming through the data, which is rather practical. There's also the *NLslider*, that solves the scaling problem by using a non-linear slider scale that always stays within the application window. Other complex approaches such as the position & velocity based *PVslider* or the *Elastic Skimming* are also viable, depending on a project's particular requirements and needs.

4. REFERENCES

[1] Zhe Wang, Matthew D. Hoffman, Perry R. Cook, Kai Li (2006) – VFerret: Content-Based Similarity Search Tool for Continuous Archived Video. Last access: 02/04/2018.

[2] ML Viaud, O Buisson, A Saulnier, C Guenais (2010) – Video Exploration: From Multimedia Content Analysis to Interactive Visualization. Last access: 03/04/2018.

[3] Henry C. Wang, David D. Feng and Jesse S. Jin (2001) – Cataloging and Search Engine for Video Library. Last access: 03/04/2018.

[4] Wolfgang Hürst, George Götz, Philipp Jarvers (2004) – Advanced User Interfaces for Dynamic Video Browsing. Last access: 03/04/2018.

[5] P. Merialdo, G. Sindoni (1996) – Using Metadata to Enhance Multimedia Query Semantics. Last access: 03/04/2018.

[6] https://computer-vision-talks.com/2011-07-13-comparison-of-the-opencv-feature-detection-algorithms/

5. DESIGN GOALS

After the methodologies described above, we've learned about several different techniques used to build intuitive multimedia exploration and visualization systems. We will now showcase the design goals and the UI composition of our project: (Semi) Automatic Timeline, in other words, what we intend to achieve with our proposed design.

5.1. Easy Repository Overview

One thing that immediately stands out from the UI-related articles we read is the importance of having a layout that can display a proper overview of the multimedia resources we have available. Thus, it's important for us to keep our content easily viewable at all times.

5.2. Immersive Playing Mode

Following the line of thought from the previous goal, the same way we want to achieve a good overview layout we also prioritize a clean and distraction-free interface for when the actual consumption content is being displayed.

5.3. Using Sliders with Caution

Based on the sliders/scrollbar usage pitfalls we've encountered in previous articles, one of our design goals is to use those sliders only when dealing with scales of fixed granularity (e.g. 0-100%).

5.4. Combining Metadata and Content-Based Similarity

Mixing the more traditional metadata-based approaches with content-based querying, our application will focus on providing an extensive range of filters, hopefully achieving highly accurate results. Like the VFerret engine (mentioned above), we will base our content-based searching on feature extraction. However, we likely won't introduce K-means clustering nor any type of supervised learning techniques in the process.

6. UI COMPOSITION

Our interface is divided in two components: Main Overview and Video Overview (**Figure 1** and **Figure 2**). Both of these components fill the entire application area.

6.1. Main Overview

In this section, as we can observe in **Figure 1**, the main overview is composed by three inner sections: **A**, **B** and **C**. **Section A** displays information about the selected video metadata or the video search filters according with the current selected tab. **Section B** displays all the available clips in the library or the current clips that are in the timeline, also according with the current selected tab. Finally, **Section C** displays the video timeline.

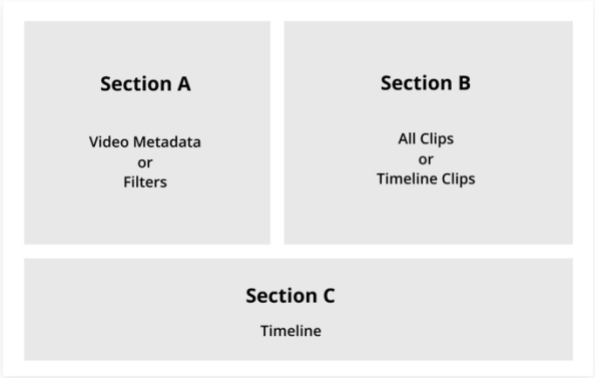

Figure 1 –Main Overview (High-Level)

6.2. Video Overview

In this section, as we can observe in **Figure 2**, the video overview is composed by **Section D**. In this section, both the timeline or a single video, can be displayed.

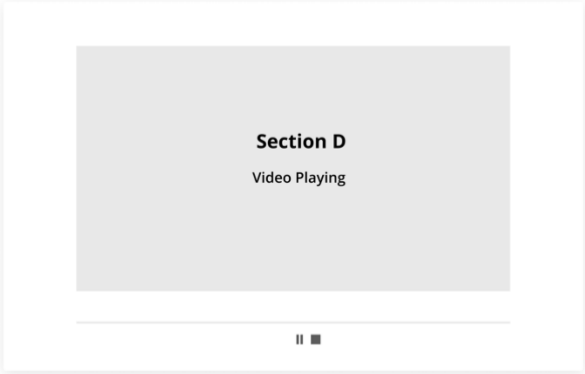

Figure 2 – Video Overview (High-Level)

6.3. Behavior & Actions

In this section we will discuss some important behavioral aspects and actions of our design, namely the high-level flow and important functionality details. Looking at **Figure 3,** in the top left corner of **Section B**, the little folder icon lets you import video clips, that will be shown in the "All Clips" section. Another important aspect to notice is that, when hovered, the video clips' moving icons are played. When a video clip is clicked once the "Video Settings" tab opens (on the left section, **Section A**), showing that video's metadata. If, on the other hand, the video is double clicked, the video will start playing in the "Video Playing" section (**Section D**).

In **Section C**, there's a small play button on the top left corner. If you press it, the timeline will start playing in **Section D**, exactly like playing a single video. Lastly, in the "Video Playing" section, the **Stop** button, when pressed, will bring you back to the Main Overview.

7. ALGORITHMS AND TECHNIQUES

To start off, we will talk about the algorithms we used during the video processing phase. A bunch of metadata was extracted so that it could further be used to perform content-based similarity queries.

7.1. Luminance

To calculate a video's average luminance, we calculated this value for each analyzed frame (of that video). In order to do this, we calculated each pixel's lightness, effectively mapping the three-color dimensions (R, G, B) onto a single-dimension (R/G/B average in grayscale, so in the zero to 255 interval). Whilst according to color theory this is not the only thing to consider for luminance calculations, it gave as a good estimate.

7.2. First Color Moment

The first color moment in an image is that image's average color (*i.e.* the mean color). Because openFrameworks grants us access to each pixel's individual color channels, we decided it would be more flexible (in terms of user input) to calculate the first color moment for each color component (R, G and B). Each of those would once again fall into the zero to 255 range. Thus, similarly to how we did for luminance, we calculated this value for all the frames of the video and then averaged it.

7.3. Face Detection

In order to implement face detection in our application, we used OpenCV's Haar Cascade trainer and detector, internally implemented using an object detection algorithm based on the usage of Haar feature-based cascade classifiers.

This detection algorithm is a machine learning based approach, that goes through a learning phase first. Specifically, a supervised learning phase takes place in the very beginning, where the classifier is trained from a lot of positive and negative images (in our case, images with and without faces, respectively). In order to extract features from the training set, simple Haar features are used (*e.g.* line features, edge features ...). The problem with this approach is the huge amount of computation it requires. Thus, in order to fight that, two different techniques are applied afterwards: Ada-Boosting and Classifier Cascading. Without going into detail, we should just point that both of them are used to cut down on irrelevant checking, making the algorithm's run-time much more bearable without sacrificing accuracy too much.

Now, whilst this algorithm can give us the number of faces appearing in a single image (*i.e.* video frame), it cannot *identify* faces, meaning that, in our application, what our metadata extraction algorithm actually outputs is the maximum number of faces found *in a single frame* throughout the whole video.

7.4. Rhythm

To calculate a video's approximate rhythm, we had multiple options, similar to the ones used for scene cut detection (*e.g.* intensity differences, simple difference of histograms, square of the difference of histograms, ...). We ended up adopting intensity-based difference checking as our go-to technique. This was mostly an implementation option; generating grayscale histograms would be unnecessary, given that we had already calculated luminance values before. Thus, we could save those same values and further calculate the absolute intensity differences between every single video frame we analyzed. The video's effective rhythm would be the average of those intensity differences, taking a value anywhere from zero (no intensity differences at all throughout the whole video) to 255 (maximum intensity differences between every frame f and its following frame $f + 1$). Because we're checking every single frame, the *average* rhythm is very low (even though there might be some high difference "spikes" between neighbor frames).

7.5. Scene Cut Detection

To detect which frames should be a video's representative frames, we take the frame-by-frame differences previously obtained during the rhythm extraction step and pick the top five frames that represent the biggest intensity "spikes" (*i.e.* differences) from the previous frame. We save those five frames as image files, so that this process does not need to be re-done in runtime - for every video - every time the application starts.

7.6. Nº of Times a Specific Object Appears

To detect the number of times a specific object appears in a video, we studied different flavors of keypoint-based image matching algorithms. Thus, we needed to choose a keypoint detector, a descriptor extractor and finally a descriptor matcher.

7.6.1. Keypoint Detection

For our possible keypoint detectors (*i.e.* detectors of points of interest), we analyzed a few different algorithms. SIFT was the first one, but it was quickly discarded in favor of SURF, which is essentially an improved, speeded-up version of the former. Furthermore, we had to pick between SURF and FAST. Even though FAST has an impressively low average detection time [**6**] in comparison to the other algorithms checked, its non-robustness to noise made it unfeasible to use. Thus, we ended up picking SURF as our detector.

7.6.2. Descriptor Extractor

After detecting keypoints in the previous phase, we needed to pick a descriptor extractor. The goal of a descriptor is to provide a unique and robust description of an image's features (*e.g.* by describing the intensity distribution if the pixels within the neighborhood of a point of interest). Descriptors are usually computed in a local manner; hence a description is obtained for every point of interest identified before.

SIFT and SURF include a detector and a descriptor. For the same reason as above, we picked SURF as our descriptor extractor. We also looked into OpenCV's homemade "free" alternative to SIFT/SURF, called ORB. However, we couldn't find enough information to make us go with that.

7.6.3. Descriptor Matching

By comparing the descriptors obtained from different images, we can find matching pairs, and OpenCV offers a few different descriptor matchers. From the two we tried (FlanBasedMatcher and BruteForceMatcher), the BruteForceMatcher with L2 distance norm and cross-checking seemed to give use the best results.

7.7. Texture Characteristics

To evaluate the texture of a video's first frame, we used the recommended Gabor filter, supported by OpenCV. We created multiple Gabor kernels - all of size five (5x5 block) and with fixed gamma and sigma, but with varying orientations and wavelengths, in order to obtain multiple results. Specifically, we created six filters with varying wavelength and four filters with varying orientation, obtaining a total of 10 different filters. After applying these 10 filters to the first frame of each video, we averaged the 10 different means. Lastly, we compared the resultant mean from the application of the 10 filters with three threshold values. If the mean is "<= 5", then we have an image with a lower different textures count, *e.g.* an image with few different colors. If the mean is "> 5" and "<= 10" then we have an image with medium texture count, *e.g.* an image of a person. Last but not least, if the mean is "> 10", then we have an image with high texture count, *e.g.* an image of a landscape. Thus, each video can be described as having a Low, Medium or High texture count, based on the method described.

7.8. Edge Distribution

To detect a video's average edge distribution, we first analyzed each frame's individual edge distribution. We decided to use the OpenCV implementation of the Scharr operator, as it seemed to be *the* overall best for this type of analysis (relatively inexpensive to apply the 3x3 kernels and more accurate than Sobel-Feldman). Because this technique is based on the computation of intensity differences, we needed to transform each frame into grayscale to further process it. And so, we did, right after applying a Gaussian Blur, in an attempt to remove some noise. Under the hood, the Scharr operator applies both the horizontal and vertical 3x3 kernels to every pixel, obtaining two values which define how strong the gradient at that point is (for each axis). Typically, we would then apply Pythagoras' Theorem to get the overall gradient magnitude at that point. But because we were simply interested in knowing the edge distribution of the frames (*i.e.* horizontal to vertical ratio), we skipped that step and instead counted and averaged the results obtained previously, using an "acceptance threshold" of 50.

8. APPLICATION IMPLEMENTATION – DECISIONS/RELEVANT DETAILS

In this section, we'll talk about the implementation of our application. Namely, we'll talk about what addons we used, the top-level architecture and how everything communicated.

8.1. Addons

On top of openFrameworks itself, we used four addons. To ease the loading/saving of XML files, we relied on ofxXmlSettings. We also found ofxGui to be a little restrictive in terms of customization options; thus, we used ofxDatGui for most of the UI instead. Finally, OpenCV was used for their implementations of computer vision algorithms and utilities, and ofxCv for their very friendly and convenient ways of bridging openFrameworks with OpenCV.

8.2. High-Level Architecture

We will now go through some aspects that we think are relevant to better understand how our project is laid out. The purpose of all individual classes/files will be discussed in the next section.

8.2.1 States and Sections

The structure of our application is based on two core architectural building blocks: states and sections. To keep the highest-level files tidy (main.cpp and ofApp.cpp), we opted to create two individual states (MainState and PlayerState) as the foundation of our project. Each state is composed of sections, themselves composed by custom logic and UI components.

8.2.2 Components

We used a composition of the elements offered by the ofxDatGui addon to build some custom components. In order to follow the design previously planned, we created components such as named tabs and image-based buttons. Alongside, we built abstractions for our video items and respective containers (*e.g.* video grid and video row), encapsulating their functionality and making them very straight-forward to deal with anywhere in the application.

8.2.3 Processing

Processing is another very important part of our project, covering everything related to video metadata representation, extraction and comparison.

8.2.3.1 In-Depth: Metadata Extraction

As soon as a clip is imported (anywhere from the file system) onto the application, it is copied to our app's resources folder. We then run our metadata extraction algorithms (explained in the previous section) on top of it, and, when done, save the results to an XML file with the same name as the video (*i.e.* for every imported video - say, myvideo.mp4, - there is a respective myvideo.xml). The video's representative frames are also obtained during this process, and they too are saved as PNGs, so that they're instantly available in next usages. Next time a previously-processed clip is imported, we simply need to load its metadata from an XML file and construct a representation of that same information that can be used inside our app. For that, we built the custom type Metadata, associated with every video item (3.2.2).

To make the process as performant as possible, the majority of the extraction phase is done inside the same loop. At the same time, we check every frame, one-by-one, in an attempt to get the most accurate values we can.

8.2.3.2 In-Depth: Object Detection in Runtime

Detecting the number of times an object appears on the video is the only information that does *not* get extracted when a clip is imported. This is because the image that is passed as the input (*i.e.* the image with the object we want to check our videos against) could be any image in the file system. Thus, if the user has that "filter" enabled, we will match the target image against the available videos. However, because this introduces quite a runtime overhead, we crafted this process carefully. It is the only place in the application where we're doing frame skipping, and we also take advantage of short-circuiting to minimize the evaluations needed.

8.2.4 Events

A centralized and globally-accessible event system was our backbone for communication, as it glued our project together. We made extensive use of openFrameworks' publish-subscribe implementation, registering listeners and broadcasting events between all the elements of our application.

8.2.5 Utility

It would be cumbersome to deal with the "raw" form of ofxXmlSettings everywhere in the app, and so we encapsulated file I/O into a specialized class. Miscellaneous convenience operations and data holders can also be found inside our utility folder, such as time formatters and global color constants.

9. CLASS DESCRIPTIONS

We will now briefly describe the purpose of each of our classes/relevant files. We recommend reading this alongside the class diagram - and maybe even the User Manual - presented as an appendix.

State

State: Abstract class that represents a base State. Concrete State implementations will, of course, extend this class.

MainState: The main state (or screen) of our application, *i.e.* the container of the high-level sections that the user sees when the app is first loaded.

PlayerState: The playback-mode state (or screen), shown when the user is watching a single clip or the whole timeline.

Section

Section: Abstract class that represents a base Section. Concrete Section implementations will, of course, extend this class.

PlayerSection: Section where either the video or the timeline is going to be played.

TimelineSection: Displays all the clips that are currently part of the timeline, with the option to reorder them.

PropertiesSection: Section that will display either a single video metadata or the available filters.

FiltersTab: Displays all the available filters to select and apply when the timeline is generated.

VideoSettingsTab: Displays the selected video properties (video's metadata).

VideoSection: Section where all the videos loaded in the system are displayed.

AllClipsTab: Displays all the videos in the system.

TimelineClipsTab: Displays only the videos added to the timeline, with the option to delete them from the timeline.

Component

Tab: Generic tab navigation button.

ImageButton: Generic, reusable button with an image.

LoadVideoButton: The button component used to import clips onto the application. Uses an ImageButton internally.

VideoGridItem: Represents an item inside a video grid.

VideoRowItem: Represents an item inside a video row.

VideoGrid: Configurable grid (2D) that holds video grid items.

VideoRow: Configurable row (1D) that holds video row items.

Processing

Metadata: Data class (getters/setters) that represents a video's metadata inside the application. Every VideoGrid/RowItem has a metadata field of this type.

Filters: Represents the values of all filters applied in a given moment in time after the "Apply Filters" button is clicked. We built this class in order to easily check which videos were "timeline-ready" for the given combination of filters applied.

FilterItem: Individual UI input element that the user can manipulate. It combines a toggle button (so that it can be enabled/disabled) and a generic ofxDatGuiComponent (*e.g.* text input, value slider, dropdown...).

ImportableFilterItem: A special kind of FilterItem that, on top of the latter's composition, has a file import input. We use this for the "Number of Times an Object Appears in the Video" input filter.

ProcessVideoAgent: This is where the processing algorithms for video metadata extraction are implemented. For each processed video, the results are saved onto an XML file and returned as a Metadata object that can be used inside the application.

Events

Events: Our events are saved into a globally-accessible namespace, as explained in section 3.3.3.

Utility

XMLFileAgent: Class responsible for handling all file I/O operations, such as reading from/writing to XML or deleting all the information about a specific video (*e.g.* the video file itself, the respective XML file and the representative PNG images).

Bounds: Many of our components need X/Y/Width/Height variables. Thus, we grouped these commonly used variables into a struct called Bounds. Thus, instead of passing four individual variables, we pass an instance of a Bounds struct.

Colors: Namespace holding colors that we use throughout the app.

Dimensions: Like we did with Colors, this file has a namespace that holds globally-accessible, high-level dimensions for different elements of our application. We use these to define the positions/sizes of the elements that are lower in the project hierarchy, making them "automatically resize" very easily.

TimeFormatter: Miscellaneous time conversion and formatting utilities.

10. CONCLUSION

In summary, building this project, gave us an insight on *how* multimedia and video management/editing applications work. More specifically, it taught us how to use all the algorithms and techniques described in **Section 7** that were lectured in our class.

We can see the impact of the multimedia subject in current applications. For instance, we can take for guaranteed the facial recognition feature we have on our phone's camera, and with this work, it lead us to understand all the processes that are behind this kind of algorithm and how it works.