Computer Science Department

Final Report– Spring 2024

Title of the project:	"Web-based Video Editor"
Team Members:	Nurtau Toganbay Ayazhan Abdirakhym Rakhat Myrzakhan
Project Advisor/Co-Advisors	Askar Boranbayev

Executive Summary

The project's goal is to produce a web-based video editor that runs only on the client side. The project's primary characteristics are:

- The video player displays real-time editing results and has player controls for pause, play, and seek functionalities.
- Video frame changers: altering opacity, color, saturation, brightness, and applying blur effects to video frames.
- Uploading several videos to edit.
- Video manipulations include trimming and removing video tracks.
- Caching client-side files which allow offline mode operation.
- Exporting films of various ratios and resolutions.
- Simple user interface with easy navigation and controls.

The link to the project codebase: https://github.com/Nurtau/video-editor

Introduction

Problem definition and motivation:

The problem this project aims to solve is the need for a fully functional web-based video editor that operates without the need for server-side processing. Many existing web-based video editors rely on server-based processing, which can be resource-intensive, slow, and costly due to the large size of video files. By creating a serverless video editor, this project addresses the following motivations:

- **Reduced bandwidth and cost:** eliminating the need for server-side processing reduces the demand for network bandwidth, making it more cost-effective for both users and the application provider.
- **High performance:** handling video processing on the client side can boost the performance several times. This approach not only accelerates processing times but also ensures that users with slower internet connections are not adversely affected.
- **Enhanced privacy and security:** processing video on the client side improves user privacy and data security since video files don't need to be uploaded to external servers.
- **Offline access:** enabling offline access through the Service Worker API ensures that users can edit their videos even when they are not connected to the internet, increasing the utility of the application.

The solution:

Our serverless web-based video editor uses browser technologies like the WebCodecs API for video processing and the Service Worker API for offline editing. These technologies will address the issues stated above by decreasing resource usage and thus enhancing efficiency, strengthening user privacy and security, and providing offline access for ongoing editing.

The report will be organized as follows:

1. **The executive summary:** provides a brief description of the project's key components.
2. **The introduction:** describes the problem, motivation, and proposed solutions.
3. **Background and related work:** examines the development of web-based video editing tools, prospective technologies for application and provides relevant works by emphasizing key technologies and methodologies.
4. **Project approach:** provides a full explanation of the project's implementation using diagrams.
5. **Project execution:** depicts adjustments made throughout implementation, what went wrong and how these issues were overcome, as well as the project's initial and improved current designs.
6. **Evaluation:** describes how we evaluate the project and whether it solved the challenge outlined in the introduction or not.
7. **Conclusion:** summarizes the project report and offers ideas for further work.
8. **References:** includes the list of used material.

Background and Related Work

The introduction of the WebCodecs API into web browsers like Chrome and Safari allowed developers to create powerful video editing tools which run directly within the browser environment (W3C, 2023).

This technology uses the hardware capabilities of the user's device which allows to reach fast performance of video frame decoding and encoding processes as in the native desktop applications.

Before the existence of the WebCodecs API, web-based video editing faced significant challenges. Developers were often constrained by the limitations of web technologies and had to rely on WebAssembly to process video. However, these solutions mainly use the CPU for decoding and encoding tasks and as a result, provide slow video editing experiences for users relative to hardware solutions in native applications.

Another alternative is server-side processing. This approach involves sending video data to remote servers for processing, which introduces several drawbacks. Since video files can be quite large, they require substantial network bandwidth to transmit them to the server. As a consequence, the process is time-consuming, leading to significant delays in video editing workflows.

Since WebCodecs API is pretty new, there's a lack of comprehensive articles and documentation. Instead, we found two GitHub repositories that have emerged as invaluable resources for our project. The first one is vjeux/mp4-h264-re-encode (**https://github.com/vjeux/mp4-h264-re-encode**), which provides information about the entire process of video re-encoding. This repository is particularly useful in understanding how to effectively use the WebCodecs API with muxing libraries which are needed to extract information and video samples from a video file. The second repository, MattiasBuelens/baby-video (**https://github.com/MattiasBuelens/baby-video**), is beneficial in demonstrating how to construct a video player using the WebCodecs API. Since building a video player is a complex process, this repository is used as a reference.

In addition to the WebCodecs API, an understanding of video codecs, frame types, and video container formats is needed for the development of a video editor application. Lee, J. B., and Kalva, H. provide valuable information about video codecs, particularly illustrating how these codecs achieve significant video compression [4]. Meanwhile, Krishna Rao Vijayanagar introduces video frame types (I, P, and B-frames) and explains their differences and interdependencies [2, 3]. On the topic of video containers, Avaro, O. et al offer an in-depth exploration of MPEG-4 structure by pointing to the importance of each field [1].

Project Approach

Video editor flow mainly consists of three main parts:

1. Video uploading
2. Playback and manipulation of video
3. Video exporting

To start with video uploading, Figure 1 illustrates how different components interact with each other when a user uploads a video through PlayerUI. FileReader parses an array of bytes from an uploaded video file and then passes this raw data to VideoDemuxer and Big Storage. In the next step, by using the mp4box library VideoDemuxer extracts chunks of video and audio tracks, video and audio decoding configs. These data are then bundled into the VideoBox class, which has a convenient API to work with. Meanwhile, Big Storage saves an array of bytes in browser memory by leveraging IndexDB API. IndexDB API efficiently handles huge files and has bigger memory limits compared to other storage features, like localStorage. The purpose of Big Storage will be explained at the end of the Project Approach.

In the above-mentioned video uploading flow, it is essential to correctly parse a video file and extract needed fields for future uses.

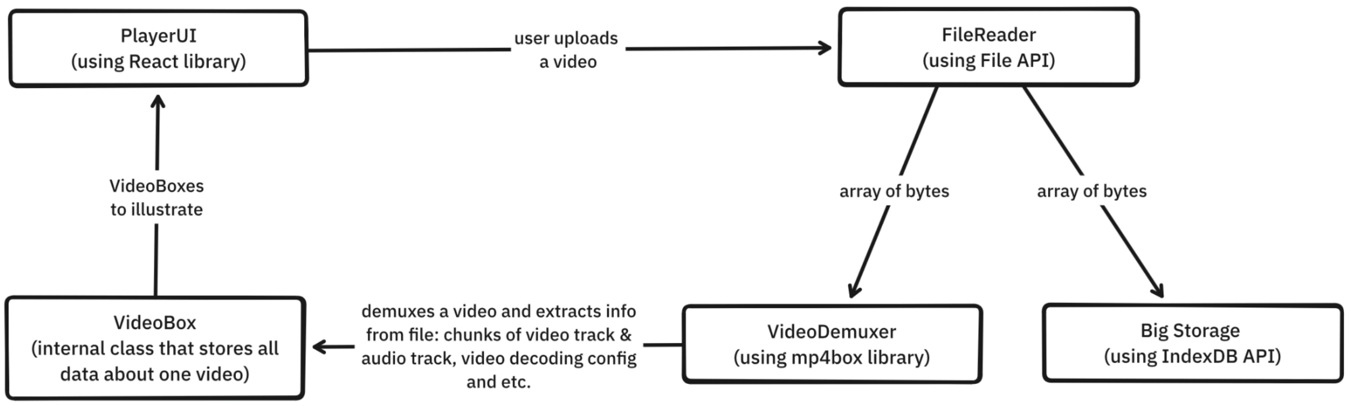


Figure 1. Video upload flow diagram

The logic behind the second part "Playback and manipulation of video" is quite big and complex. Figure 2 depicts what happens when a user manipulates a video. VideoController is a core of logic that orchestrates many other classes, like VideoDecoder, AudioDecoder, VideoFrameChanger and others, to play and manipulate a video. To play a video, video and audio chunks need to be decoded by VideoDecoder and AudioDecoder respectively. To note, VideoDecoder and AudioDecoder are thin wrappers around WebCodecs API. Then decoded video frames and audio data are pushed to their corresponding queue. Since decoded frames and data are big in memory, the size of the queue should be kept to a minimum to optimize memory usage. Therefore, the sizes of these queues are not bigger than the specified threshold.

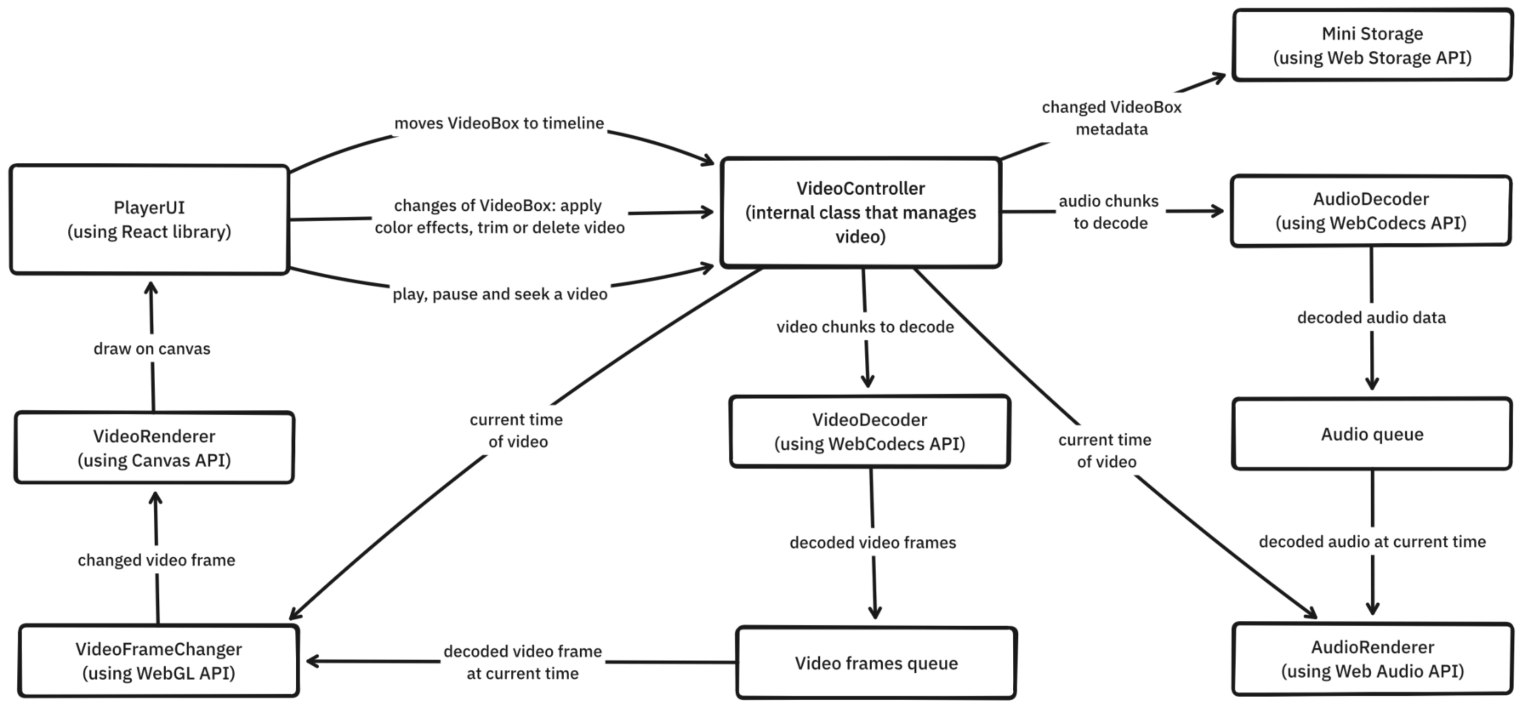

Mini Storage (using Web Storage API)
changed VideoBox metadata
moves VideoBox to timeline
PlayerUI (using React library)
changes of VideoBox: apply color effects, trim or delete video
VideoController (internal class that manages video)
audio chunks to decode
AudioDecoder (using WebCodecs API)
play, pause and seek a video
draw on canvas
video chunks to decode
decoded audio data
VideoRenderer (using Canvas API)
current time of video
VideoDecoder (using WebCodecs API)
current time of video
Audio queue
changed video frame
decoded video frames
decoded audio at current time
VideoFrameChanger (using WebGL API)
decoded video frame at current time
Video frames queue
AudioRenderer (using Web Audio API)

Figure 2. Video playback and manipulation diagram

Additionally, there is quite an interesting fact about video chunks. Video chunks might be dependent on future chunks, in other words, a chunk that should be presented at 1:00 cannot be decoded without first decoding a chunk at 1:01. These moments are carefully managed and handled by the VideoBox class, which exposes `getVideoChunksDependencies`, `getNextVideoChunks`, `getAudioChunksDependencies` and `getNextAudioChunks` methods.

To correctly update video frames on canvas and play the needed part of audio, VideoController calls `advanceCurrentTime` on video play. This method is recursive and calls itself on each browser paint, so if a computer has a 60 Hz monitor, then `advanceCurrentTime` will be called 60 times per second. The implementation of this function is shown below:

```javascript
JavaScript
private advanceCurrentTime(now: number) {
    this.currentTimeInS = this.getCurrentVideoTime(now);
    this.lastAdvanceTimeInMs = now;
    let reachedEnd = false;
    if (this.currentTimeInS >= this.totalDurationInS) {
      reachedEnd = true;
      this.currentTimeInS = this.totalDurationInS;
    }

    eventsBus.dispatch("currentTime", this.currentTimeInS);
    this.decodeVideoFrames();
    this.decodeAudio();
    this.renderAudio();
    this.renderVideoFrame();

    if (reachedEnd) {
      this.pause();
    } else {
      this.advanceLoopId = requestAnimationFrame((now) =>
        this.advanceCurrentTime(now),
      );
    }
  }
```

On `advanceCurrentTime`, "currentTime" event is dispatched to update ui components. After that decoding methods are called. If the sizes of video frame queue and audio queue are already bigger than specified threshold, then nothing will be decoded, otherwise some chunks will be decoded to fill these queues until threshold. On this.renderAudio(), specific decoded audio data, which playing time lies in the current time of video, is extracted from the audio queue and passed to the AudioRenderer class, which in turn plays audio by using Web Audio API. Meanwhile, on this.renderVideoFrame() a found decoded video frame from the video frame queue is processed by VideoFrameChanger before rendering it on canvas by VideoRenderer. VideoFrameChanger changes a frame in two steps:

```
JavaScript
processFrame = (frame: VideoFrame, effects: VideoBoxEffects) => {
    const init = {
      codedHeight: frame.codedHeight,
      codedWidth: frame.codedWidth,
      displayWidth: frame.displayWidth,
      displayHeight: frame.codedHeight,
      duration: frame.duration ?? undefined,
      timestamp: frame.timestamp,
      format: frame.format!,
    };

    const processedByPixelCanvas = this.perPixelProcessor.processTexture(
      frame,
      effects,
    );
    const processedCanvas = this.spatialConvolutionProcessor.processTexture(
      processedByPixelCanvas,
      effects,
    );

    return new VideoFrame(processedCanvas, init);
  };
```

Firstly, frames are passed through a pixel changer, where effects, like brightness, and saturation, are applied. Secondly, it applies spatial effects, like blur. To note, on spatial effects, the resulting color values of pixels depend on themselves and their neighbors. This two-step approach helps to reduce the number of calculations, thus to boost performance.

To store changed VideoBox metadata, like the range of video and applied video effects, Web Storage API, specifically localStorage is used. The component is called "Mini Storage" because localStorage is good at storing little data compared to IndexDB. The purpose of Mini Storage will be explained at the end of the Project Approach.

Thirdly, the logic of the video exporting part is quite similar to the previous part. But, as Figure 3 shows during video export there is no need to track video and audio queue because each decoded video frame is instantly processed. After processing each frame by VideoFrameChanger, VideoEncoder encodes video frames and outputs video chunks. These video chunks along with audio chunks are passed to VideoMuxer, where an mp4 file is created and filled with video and audio tracks. It is worth noting that audio chunks have not been decoded. Since their data are not changed compared to video frames, there is no need to decode audio chunks.

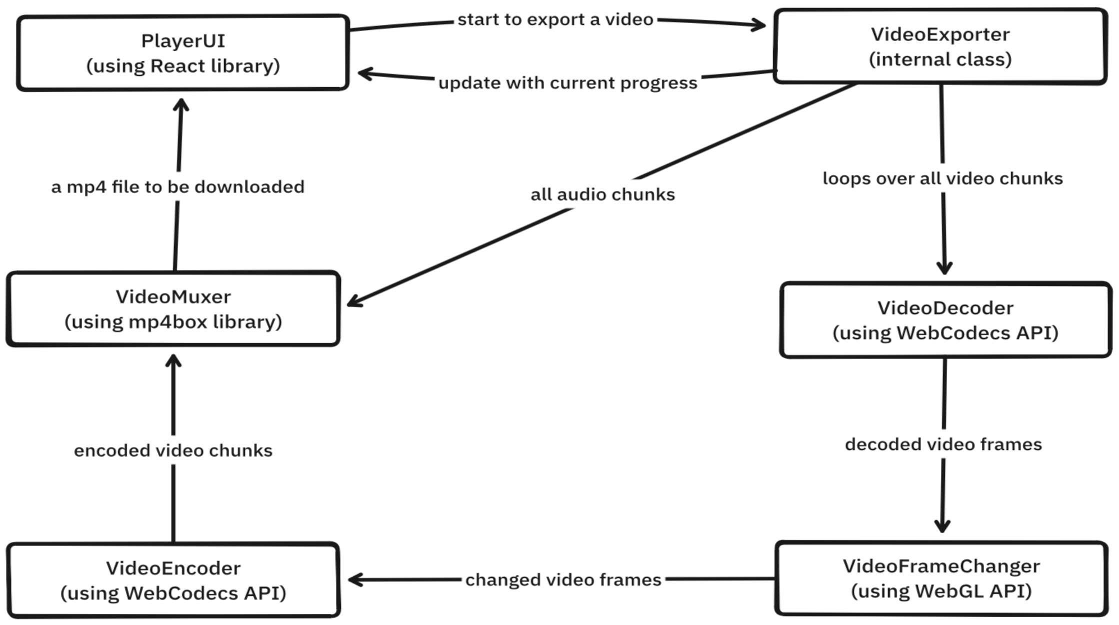


Figure 3. Video export flow diagram

As previously mentioned, the purposes of Big Storage and Mini Storage are going to be explained. The system caches almost everything from big video files to small video metadata. This cached data is used on visiting the video editor to recreate the previous state of the editor, in other words, on page refresh nothing will be deleted and changed. This greatly improves the user experience. Figure 4 illustrates a high-level overview of how VideoBoxes are populated from storage during the visit and subsequent refreshes of the page.

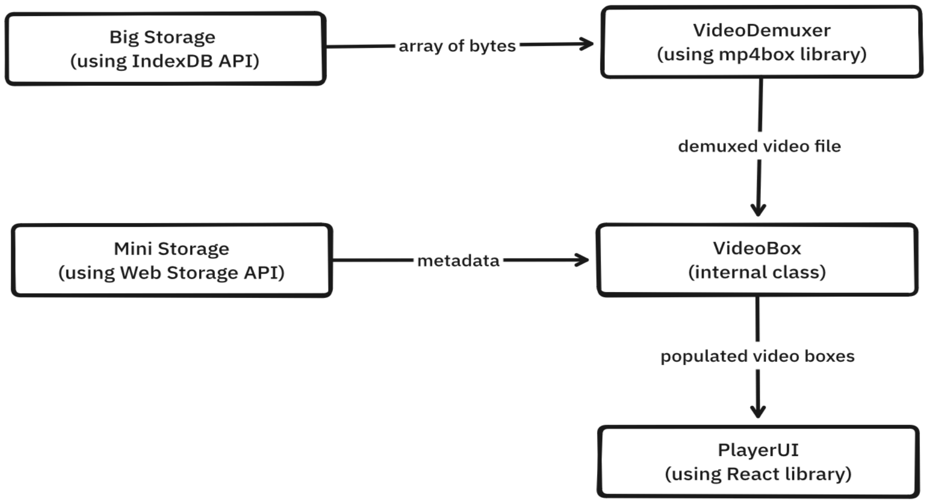

Big Storage (using IndexDB API)
array of bytes
VideoDemuxer (using mp4box library)
demuxed video file
Mini Storage (using Web Storage API)
metadata
VideoBox (internal class)
populated video boxes
PlayerUI (using React library)

Figure 4. VideoBox population from storage

Since the project does not need a server to process video files, then it is possible to implement a cache layer to allow offline mode access. The cache layer will be built using Service Worker API to intercept each request of the browser and cache responses in memory. Cached responses can be used in the case of an offline network.

Figure 5 illustrates how client-side files (HTML, CSS, JS) could be fetched when there is no internet connection. Firstly, the user visits a website, then the browser requests needed client-side files. The request is intercepted by the cache layer and redirected to the server that serves these files. When a request cannot reach that server, which is a usual case caused by offline internet access, the cache layer returns cached files, that have been saved during previous successful visits, to the browser. Thus, the project could support offline mode access which, in turn, enhances user experiences.

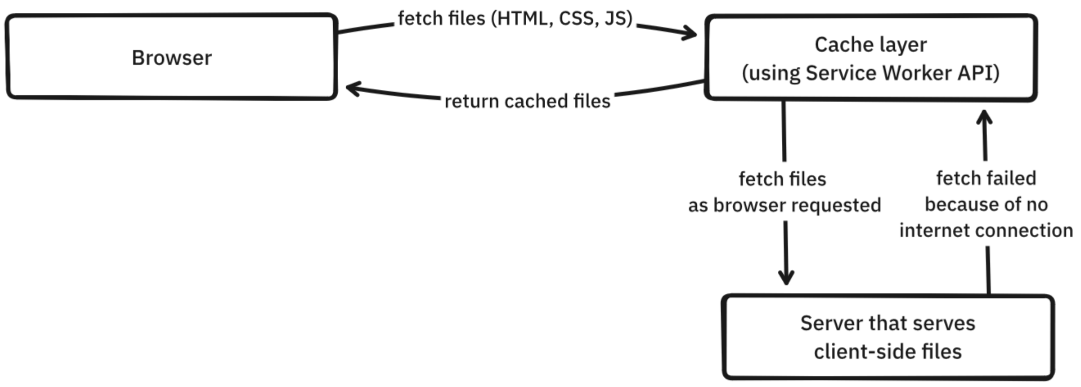

Browser
fetch files (HTML, CSS, JS)
Cache layer (using Service Worker API)
return cached files
fetch files as browser requested
fetch failed because of no internet connection
Server that serves client-side files

Figure 5. Offline mode diagram

Project Execution

The video editor logic is complex, therefore we have encountered some unexpected problems and made wrong design decisions during project execution.

Firstly, on the first-semester logic flow, there was a tight coupling between components of all parts: video uploading, playback, manipulation and export. It was hard to debug and add new features because of tight coupling. Therefore, in the second semester, it was decided to rewrite components to remove coupling at all. As it is visible in Figures 1-3, we successfully completed this big refactoring task.

Secondly, we thought that we would use WebAssembly libraries to apply effects on video frames because WebAssembly is fast and can easily parallelize work by leveraging Web Workers API. However, we have encountered a huge performance issue. Decoded video frames were stored in GPU memory. So, to apply effects by WebAssembly we need to copy video frame data from GPU to CPU. Then process video frames by WebAssembly (CPU), and finally copy processed video frame data from the CPU to the GPU. Firstly, the video frame data size is big, therefore transferring this data from CPU to GPU and vice versa is slow, about 5 ms per video frame. In addition to this, processing video frames by CPU is also slow, 8 ms per video frame. So, in total processing one video frame by WebAssembly takes 18ms (two copies and one processing). 18 ms is slow and laggy. For comparison, to support 60 FPS video each video frame should be decoded and processed in 16 ms. Therefore, we analyzed alternatives and found that we can avoid copying fully by processing video frames on GPU. In addition to this, GPU is well suited to process hundreds of pixels at the same time. After migrating from WebAssembly (CPU) to WebGL (GPU) processing, we found that processing time takes only 1 ms per video frame (0 copies and one processing). To note, all the above measurements are taken in MacBook Pro 16.2 M1 Pro.

Finally, we improved the initial UI/UX design of the editor from the last semester:

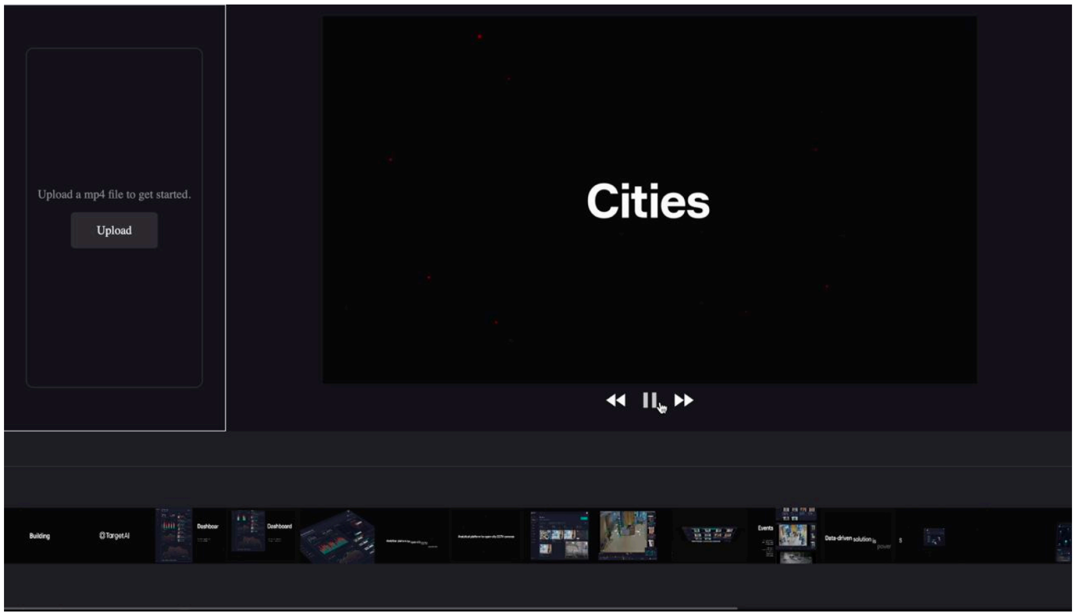


Figure 6. Initial design of the Video Editor

There were several problems with the initial design, which is why we needed to upgrade the UI/UX of the video editor:

1. In the first design, you can really spot that the elements are fighting for attention rather than leading the user's eye to the key areas. There is really no pinpoint of focus.
2. It is not clear with respect to the functionality that every part of the interface may have, and hence, a little bit of confusion here and there may be inevitable. For example, there is a lack of clarity in the function of the 'Upload' button—will it make a new upload or just finalize the existing selection?
3. The only controls for navigation are slim, and it is entirely unclear how one might shuttle back and forth between these different planes of the editing process. There is no visual cue, much less a label, to guide the user.
4. The first impression reveals a not-too-good use of space, particularly in the aspect of the timeline. It is really squeezed quite tight, meaning that accurate editing may be somewhat difficult.
5. The use of different styles of buttons for 'Upload' and the play controls is almost jarring to the eye, causing disharmony across the interface and inconsistency in the UI.
6. It appears that the design has not been proportional, thus it may bring about a problem for those using different screens with different resolutions.

To tackle these issues, the following improvements were implemented:

1. The layout was changed to be more focused and clean, and a timeline was added to allow the users to be more precise during editing. Also, we enabled thumbnails for the uploaded videos in our left-top panel that will help users identify their videos at a glance.
2. Our UI was enhanced by exploiting the explanatory power of usability heuristics. By enabling direct manipulation in the canvas, the editor shows the results immediately to the user who has visible access to all the objects (Nielsen, 1994).
3. We applied some changes to this sidebar on the left side by adding larger icons with more space between them. Thanks to this, we could add new sections and tools, and ease the navigation between them since the icons reduce the visual cognitive load, and UX should be improved (McGuffin & Balakrishnan, 2005).
4. A visual hierarchy was built through contrasts and spacing in the area around the main video preview window. This naturally draws users' attention to the most important part of the application - the video they are editing. (Vlasenko, 2022)
5. Consistency in design elements, like color and typography was needed for our design. We could achieve it using dark tones and a sans serif font, thereby helping the eyes of the user not to strain, especially for those spending long hours editing videos.
6. We added zooming in and out buttons to the timeline to increase preciseness and editing speed for users, thus we can reach a wider audience covering users that work with long-form and/or short-form content.

As a result, we could achieve this upgraded UI with better, more user-friendly UX (more interfaces can be found in an appendix of this report):

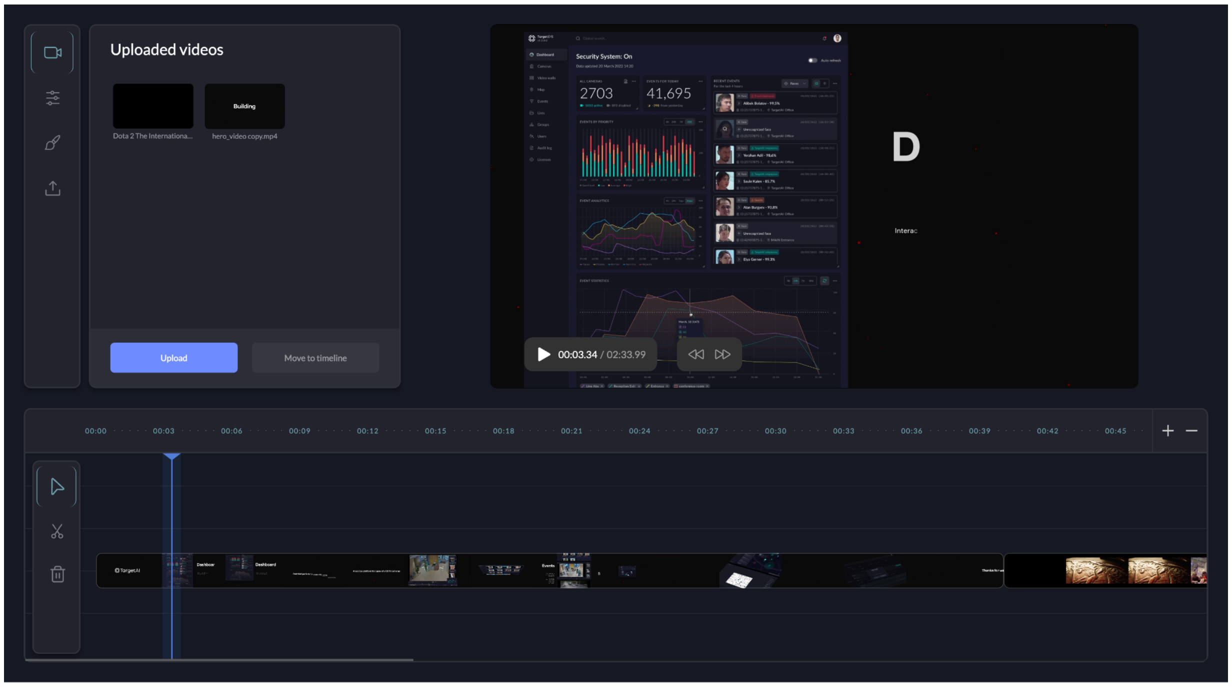


Figure 7. Upgraded UI/UX design of the video editor

Evaluation

The problem description:

- The problem this project aims to solve is the need for a fully functional web-based video editor that operates without the need for server-side processing. Many existing web-based video editors rely on server-based processing, which can be resource-intensive, slow, and costly due to the large size of video files.

Our solution solves these issues by including the following critical features:

- Client-side processing considerably improves performance, resulting in quicker processing times.
- Offline editing enables users with slow internet connections to have a pleasant editing experience.

Experiments:

We evaluated our solution by comparing it to other enterprise level web-based video editors. The experiment involves importing and exporting the same video file across three platforms: our app (serverless), a server-side video editor, and a client-side video editor. The video file used was in MP4 format, 38.3 MB in size, and with a resolution of 1080p. We focused on two metrics: export time (the time it takes to export a video file) and output video size.

According to Table 1, our video editor greatly outperformed the server-side one in export time, 7.25 sec compared to 23.68 sec. Such a great difference is due to the fact that in server-side video editor video files should be downloaded from the server, which takes some time and highly depends on internet speed. However, the server-side video editor produced a lesser export file size of 5.6 MB than our project's 9.2 MB file size. This suggests that server-side processing can be more efficient in terms of file size optimization, possibly due to better compression algorithms and optimization techniques performed on the server-side.

In contrast, another client-side video editor completed the export in 6.21 seconds and with a file size of 32.7 MB. Our project has a little longer export time of 7.25 seconds, but a much smaller file size of 9.2 MB. These results show that our video-editor is performant and produces video files in an optimized way.

To summarize, the experiment evaluates export performance and associated file sizes across three video editors, confirming the project's efficiency in terms of both speed and file size optimization during video export.

Name	Input size (1080p)	Export time	Export size (720p)
Our project	38.3 MB	7.25 sec	9.2 MB
Another server-side video editor		23.68 sec	5.6 MB
Another client-side video editor		6.21 sec	32.7 MB

Table 1. Experiment results

Conclusion and possible future work

While developing a web-based, client-sided video editor, our team significantly advanced in video processing technologies that optimally utilize the capabilities of client-sided systems. Thanks to using modern APIs like WebCodecs and Service Worker, we could solve the problems that arise while working with web-based video editors such as costly and slowed performance, privacy and accessibility.

Our project's goal was to address the initial problem description provided in the introductory section, and we believe that we succeeded. However, by including the following future work items, the web-based video editor might evolve into a comprehensive and feature-rich platform for video production and editing, addressing a wide range of user requests while increasing overall functionality and usability.

Future work include:

- **Separate audio addition and manipulation:**
  - Implement the ability to add and change audio tracks separately from video, allowing users to enhance their videos with background music, voiceovers, or sound effects.
  - Include features like volume adjusting, audio track cutting, and multi-track mixing.
- **Animations:**
  - Use animation capabilities to add dynamic elements to videos, including transitions, text, and motion graphics.

- **AI integration:**
  - Auto-Captions: Integrate AI algorithms to automatically generate and synchronize video captions/subtitles, hence boosting accessibility and user experience.
- **Text addition:**
  - Allow users to put text overlays on their videos, including titles, subtitles, captions, and notes.
  - Users should be able to adjust the font, size, alignment and color of the text.

Presentation at Conference [05.09.2023]:

We also had a chance to present our “Web-based Video Editor” project in the international scientific and practical conference “Industrial development: technologies for people and services in the era of innovation”, dedicated to the memory of the founder of the university, academician Zulharnay Aldamzhar. More about the conference can be found here: https://drive.google.com/file/d/11t0mDhZ2urRlcKN6SG9GQT5-Felxh5PA/view?usp=sharing

References

[1] Avaro, O., Eleftheriadis, A., Herpel, C., Rajan, G., & Ward, L. (2000). MPEG-4 systems: overview. *Signal Processing: Image Communication*, *15*(4-5), 281-298. Available: https://www.sciencedirect.com/science/article/pii/S0923596599000508

[2] Krishna Rao Vijayanagar. "Closed GOP and Open GOP – Simplified Explanation." Ottverse, December 17, 2020. Available: https://ottverse.com/closed-gop-open-gop-idr/

[3] Krishna Rao Vijayanagar. "I, P, and B-Frames – Differences and Use Cases Made Easy." Ottverse, December 14, 2020. Available: https://ottverse.com/i-p-b-frames-idr-keyframes-differences-usecases/

[4] Lee, J. B., & Kalva, H. (2006, July). An efficient algorithm for VC-1 to H. 264 video transcoding in progressive compression. In *2006 IEEE International Conference on Multimedia and Expo* (pp. 53-56). IEEE. Available: https://ieeexplore.ieee.org/abstract/document/4036534

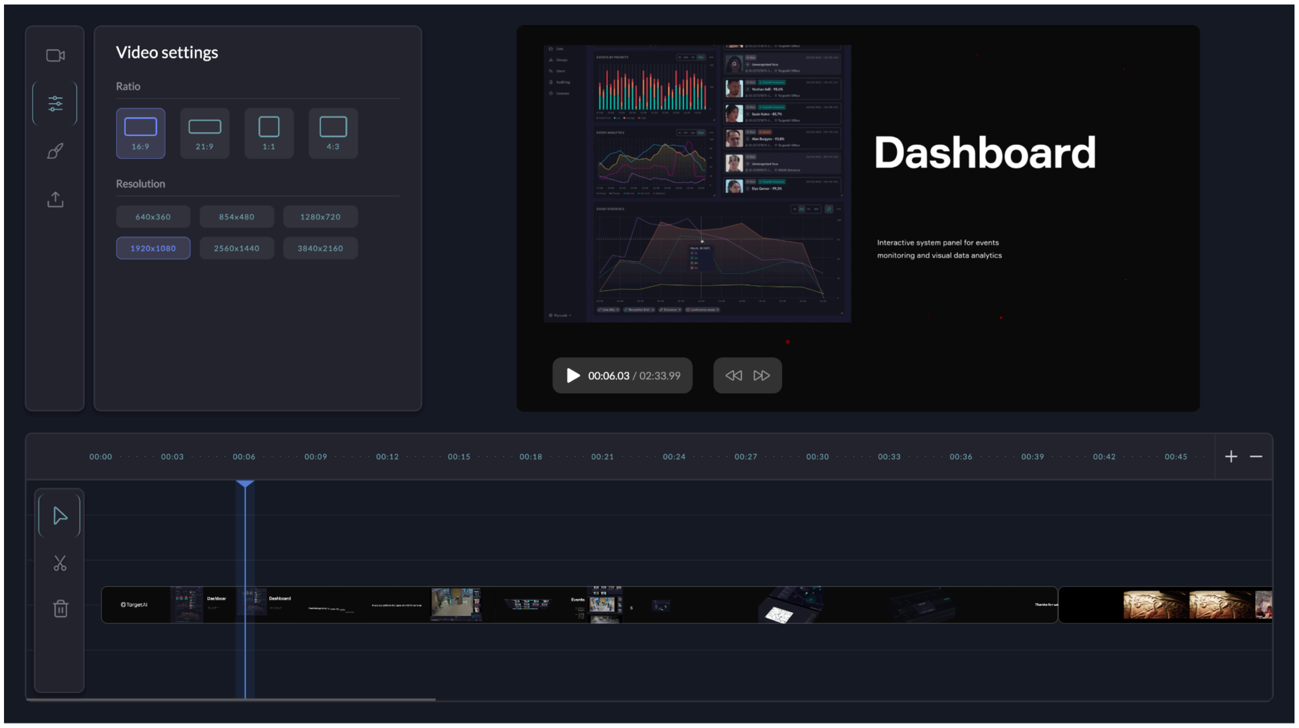
[5] McGuffin, M. J., & Balakrishnan, R. (2005). Fitts' law and expanding targets: Experimental studies and designs for user interfaces. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *12*(4), 388-422. Available: https://dl.acm.org/doi/pdf/10.1145/1121112.1121115

[6] Nielsen, J. (1994, April). Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (pp. 152-158). Available: https://dl.acm.org/doi/pdf/10.1145/191666.191729

[7] Vlasenko, K. V., Lovianova, I. V., Volkov, S. V., Sitak, I. V., Chumak, O. O., Krasnoshchok, A. V., ... & Semerikov, S. O. (2022, March). UI/UX design of educational on-line courses. In CTE Workshop Proceedings (Vol. 9, pp. 184-199). https://acnsci.org/journal/index.php/cte/article/view/114

[8] W3C. (2023). WebCodecs API. World Wide Web Consortium (W3C). Available: https://www.w3.org/TR/webcodecs/

Appendix


Video settings

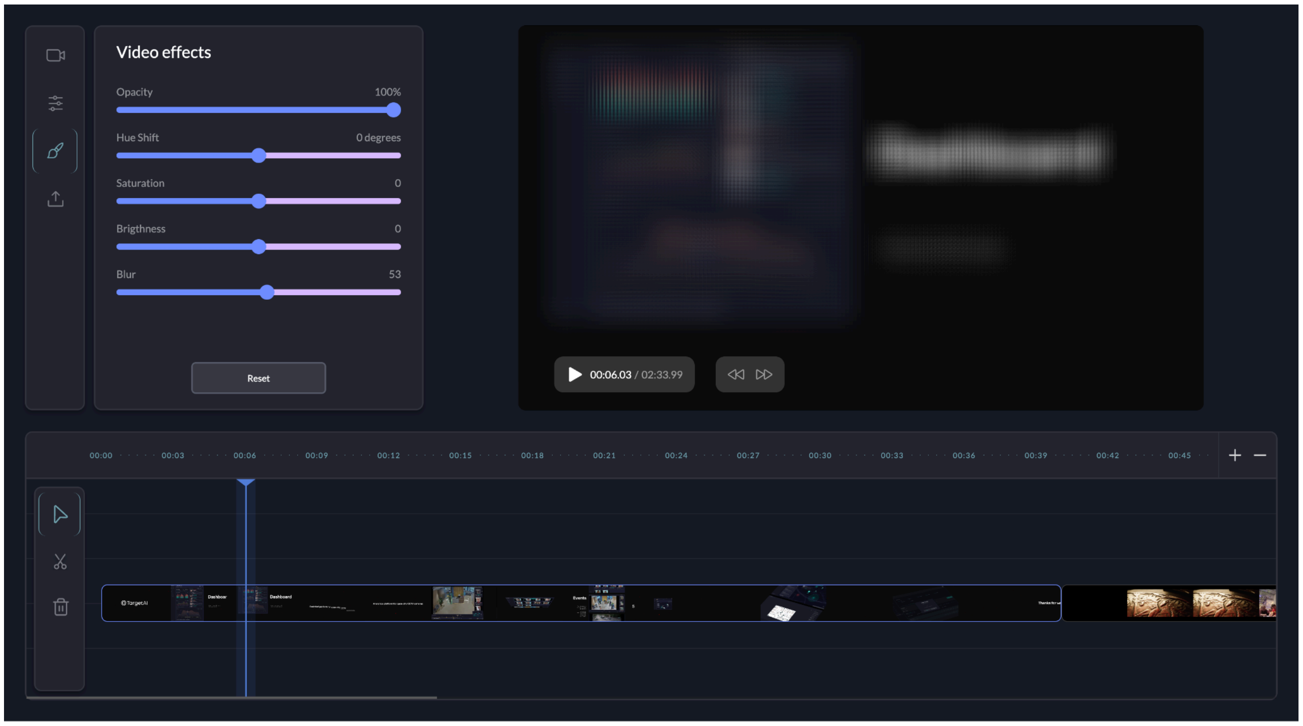
Ratio

[✓ 16:9] 21:9 1:1 4:3

Resolution

640x360 854x480 1280x720 [✓ 1920x1080] 2560x1440 3840x2160

Figure 8. UI of Video Settings panel: ratio and resolution


Video effects

Opacity 100%

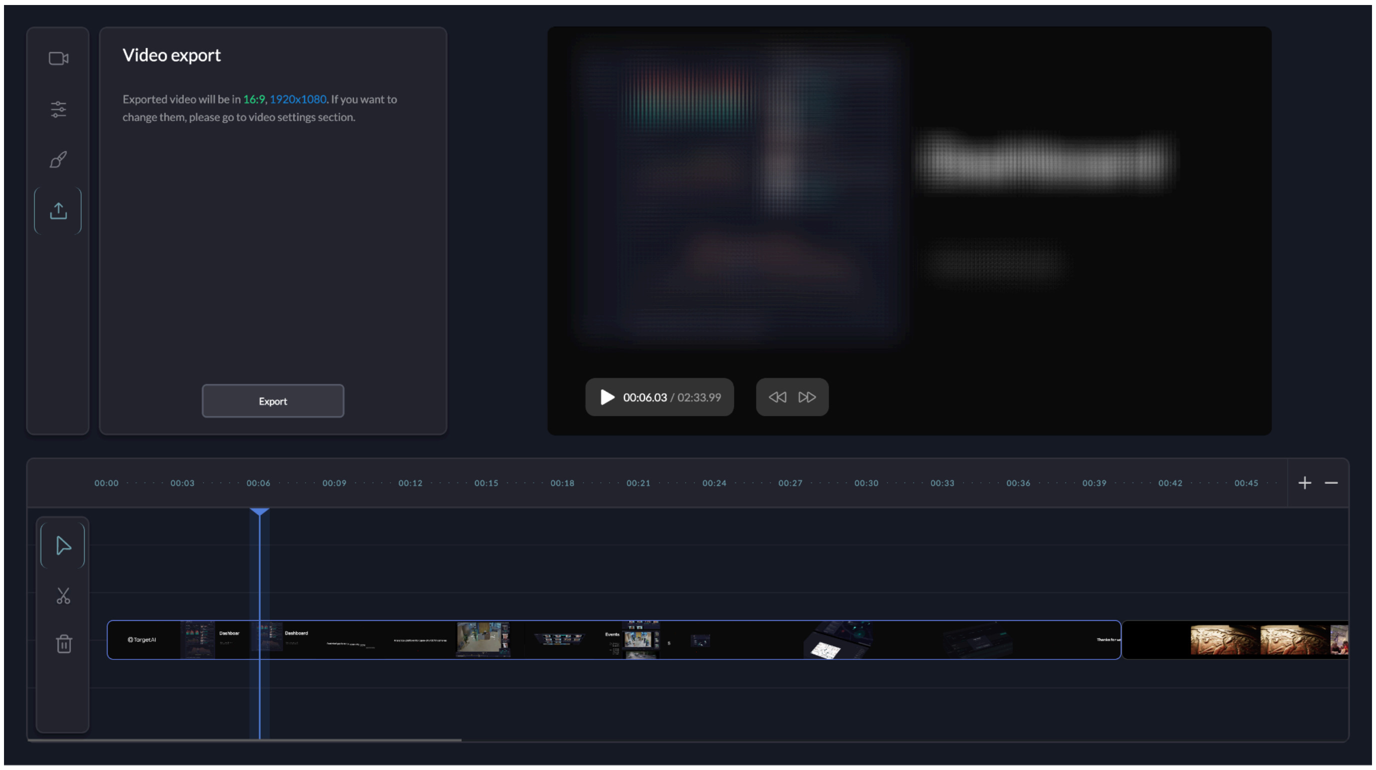
Hue Shift 0 degrees

Saturation 0

Brigthness 0

Blur 53

Figure 9. UI of Video Effects panel (video was blurred to show the effect of blur)



Exported video will be in 16:9, 1920x1080. If you want to change them, please go to video settings section.

Figure 10. UI of Video export panel that shows the settings chosen by user

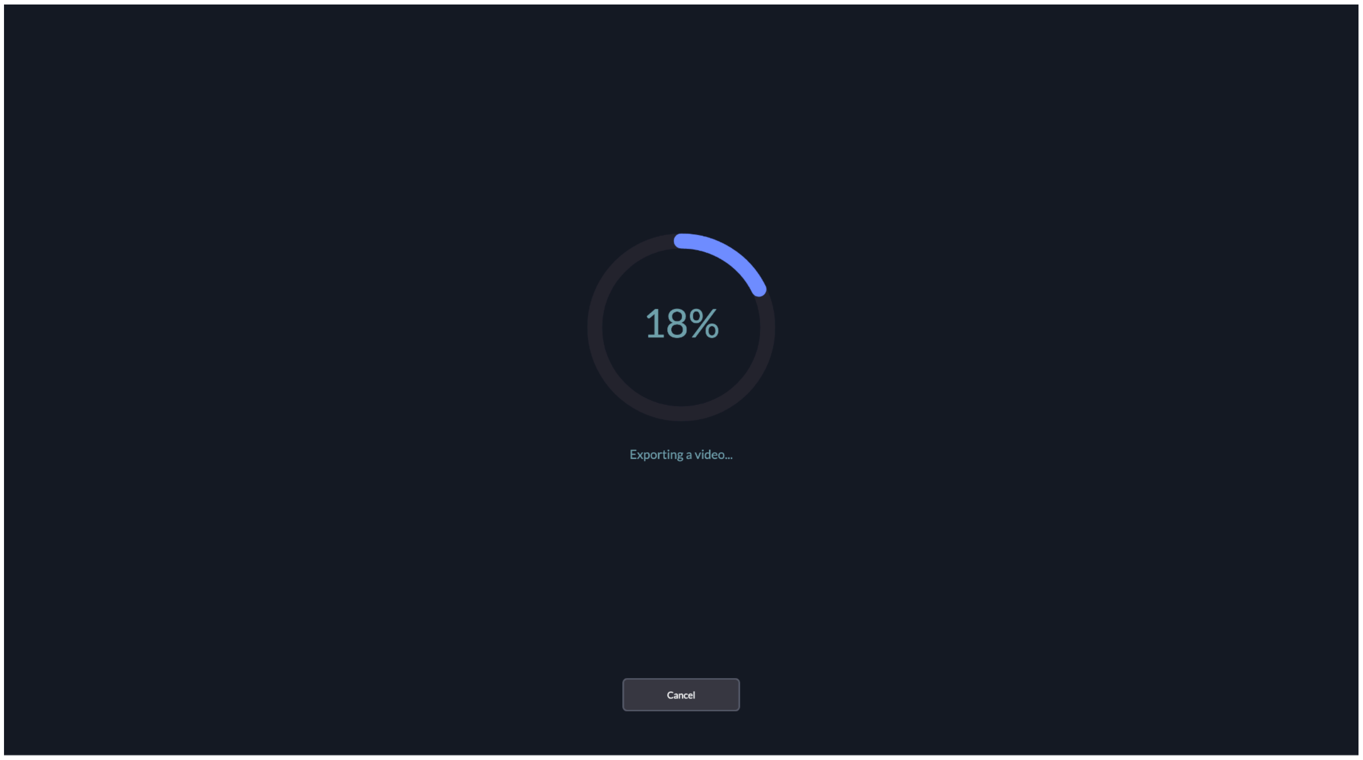
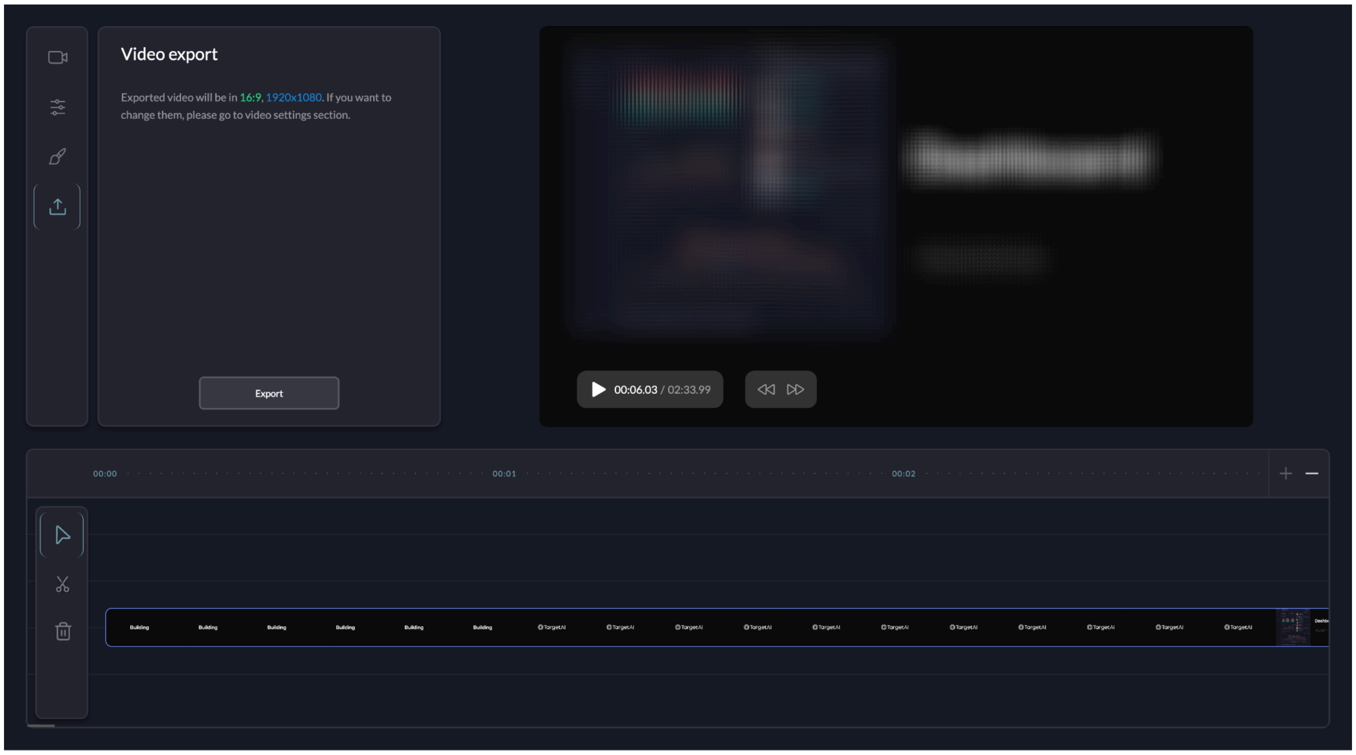
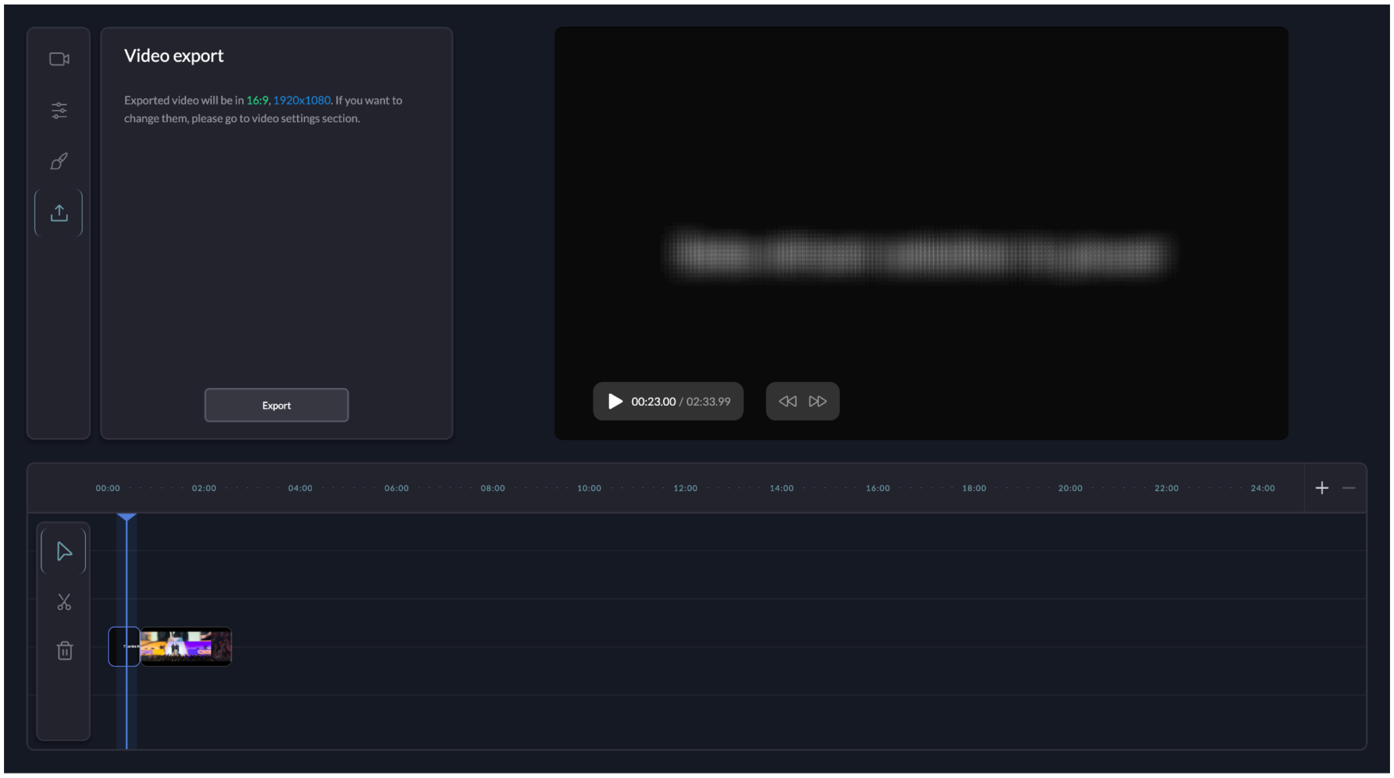


Figure 11. UI of video exporting



Exported video will be in 16:9, 1920x1080. If you want to change them, please go to video settings section.

Figure 12. Maximum zoomed timeline for higher precision



Exported video will be in 16:9, 1920x1080. If you want to change them, please go to video settings section.

Figure 13. Maximum zoomed-out timeline for higher coverage