A web-based video editor for PCIT-VR

Alistair Geesing

July 26, 2019

Supervisor(s): Robert Belleman
Abstract

In Parent-Child Interaction Therapy (PCIT) a parent is taught skills that will reinforce appropriate behaviour of their child with behavioural problems. A virtual reality app has been developed for smartphones to allow parents to train these skills at home. The virtual environment is created using 360-degree video and interactive elements that appear at key moments in the video. This research has produced a video editor that allows therapists to easily create these virtual environments. The video editor is a web-application using the client/server model as a result of the restrictions imposed by the environment in which it will run. Being reliant on the network for data access imposes some restrictions on the editor. In contrast with native video editing applications which can freely use full-fidelity video for their user-facing parts, those of the PCIT-VR editor can only access videos with a lower bitrate so as to provide a smooth user experience. Nonetheless, the editor can be used for easy creation of interactive virtual environments.
# Contents

1 Introduction .................................................................................................................. 7
   1.1 Parent-Child Interaction Therapy ............................................................................. 7
   1.2 PCIT-VR .................................................................................................................. 7
   1.3 Research goal .......................................................................................................... 7

2 Design ................................................................................................................................ 9
   2.1 Functional requirements ............................................................................................ 9
   2.2 Design considerations ................................................................................................ 10
      2.2.1 Organizational restrictions ................................................................................. 10
      2.2.2 Editing requirements ......................................................................................... 11
      2.2.3 User friendliness ............................................................................................... 13
      2.2.4 Responsiveness .................................................................................................. 13
   2.3 Application design .................................................................................................... 14

3 Implementation: Editor .................................................................................................... 15
   3.1 UI layout and styling .................................................................................................. 15
      3.1.1 HTML: principles .............................................................................................. 15
      3.1.2 CSS: Grid layout ............................................................................................... 16
   3.2 Scripting the editor .................................................................................................... 18
      3.2.1 TypeScript .......................................................................................................... 18
      3.2.2 Architecture ...................................................................................................... 19
      3.2.3 Server interaction ............................................................................................. 19

4 Implementation: Server ................................................................................................... 23
   4.1 FFmpeg ....................................................................................................................... 23
   4.2 NGINX ....................................................................................................................... 24
   4.3 Remote procedure calls ............................................................................................. 25
   4.4 Jobs .............................................................................................................................. 25
   4.5 Events ......................................................................................................................... 25
   4.6 Database .................................................................................................................... 25

5 Results ................................................................................................................................ 27

6 Conclusion and future work ........................................................................................... 29
   6.1 Conclusion ................................................................................................................. 29
   6.2 Limitations and future work ..................................................................................... 29

References ............................................................................................................................. 31

Appendices ............................................................................................................................ 33

A HTML and CSS used for layout ...................................................................................... 35
   A.1 HTML ....................................................................................................................... 35
   A.2 CSS .......................................................................................................................... 35
1.1 Parent-Child Interaction Therapy

Parent-Child Interaction Therapy (PCIT) can be used to improve counter-productive parent-child dynamics [1]. For parents of a child with behavioral problems it can be challenging to respond to their child in a way that is conducive to improving the child’s behavior. Disruptive behavior disorders such as oppositional defiant disorder (ODD) and conduct disorder (CD) cause uncooperative behavior. ODD is characterized by persistently defiant, disobedient and hostile behavior; an aspect of CD is the persistent violation of the rights of others. These behavioral problems can increase in severity if they are not treated and increase the risk of developing psychiatric disorders and unemployment in adulthood.

The behavioral disorders and their progression have a number of causes such as genetics, the child’s environment and parenting. A feedback loop exists between the behavior of a parent towards a child and the child’s disruptive behavior: increased parenting stress leads to increased disruptive behavior of the child, which in turn leads to increased parenting stress.

To help break this vicious cycle PCIT can be used. The therapy consists of multiple sessions that are split into two stages: a play phase and a management phase. In the play phase parents are taught how to reinforce appropriate behavior by using a number of responses: praising, reflecting, imitating, describing and enjoying. In this phase only desired behavior is acknowledged. During the management phase the parent continues using the techniques of the play phase and is taught how to effectively set limits and how to decrease undesired behavior.

1.2 PCIT-VR

A virtual reality application which runs on smartphones was developed that makes it possible for parents to train the skills taught in PCIT [20].

Users are situated in an environment created with 360-degree video. Over the course of the video, the environment is frozen at key points and the user is asked how to respond to the current situation. A number of options is presented from which the user picks the response that they think is most appropriate. After an answer has been chosen the correct answer is shown and the video continues playing.

1.3 Research goal

To create the interactive PCIT-VR videos, video editing software is used to edit source material from a 360-degree camera into a single video, after which interactive elements are manually added. The final video must be in a specific format at a specific bitrate, must be packaged in a particular way, and has to be made available over the web to users of the VR application.
Adaptive bitrate streaming is a technology which dynamically switches between videos of different bitrates, choosing the highest possible bitrate that the network can support at the time. To use adaptive bitrate streaming, it is necessary to create a number of final videos in a range of bitrates.

The goal of this research is to develop a video editing application which lets therapists easily create interactive 360-degree video for use in PCIT-VR, without them having to worry about these technical details.
The 360-degree video used to create the PCIT-VR environment is recorded by the therapist during the parent’s PCIT training. The recorded video must be edited to cut out irrelevant sections and to add interactive elements at key points, where the video is paused and the parent is asked questions about the current situation. The therapist and a technical assistant work together to edit the video and add the interactive elements, which results in an interactive 360-degree video of the therapy session. The parent views the interactive 360-degree video with a VR player to re-evaluate the therapy session in order to reflect on and to improve their behaviour toward their child.

The basic steps for the editing of interactive 360-degree video are displayed in figure 2.1. Currently, a commercial video editor and a handwritten XML file are used to produce the interactive video. The goal of the research project described in this thesis is the creation of an editor with a lower barrier to entry, to let therapists produce PCIT-VR videos without assistance. The following sections elaborate on the requirements and design of the editor.

2.1 Functional requirements

The following list specifies the requirements of the video editing application:

1. Organizational restrictions — In many clinical settings, because of patient privacy concerns, installation of unapproved software on the organization’s computing environments is not allowed. Therefore, the editor must be made available to the therapist without running into this restriction.

2. Editing requirements — The editor must have the following abilities:

   • Arrange video — The recording of a therapy session may produce multiple video files. The therapist must be able to arrange these videos in a sequence.

   • Cut out video — Some parts of the recorded session will not be of interest. The editor must provide functionality to allow cutting out these sections of video.

   • Adjust volume — The audio of some parts of video may be too soft or too loud. Playing the interactive video will be annoying to the parent if the volume of these parts is not adjusted during editing. The therapist has to be able to adjust the volume of parts of audio that are not at the correct volume.
• Add interactive elements — The therapist must be able to specify when the video is paused and the parent is shown menus from which they pick the most appropriate response to the situation at that point in the video.

• Preview video — It is necessary to see what is being edited and to view intermediate results. Because the preview shows a projection of the 360-degree video on a flat screen, analogous to a projection of the earth on a map, it can happen that an area of interest is divided and displayed at opposite edges of the preview. Therefore, the user must be able to pan the video to be able to center areas of interest in the preview.

• Export — When editing is complete, the interactive 360-degree video must be made available to the parent. The exporting process consists of combining the individual assets appropriately and producing video files in a range of bitrates, packaging the video files with the file that describes the interactive elements, and making this package available on the web.

3. User friendliness — The therapists that will use this editor have no training in video editing. The editor must therefore be easy to use and have no unexpected behaviour or unnecessary knobs to turn. Ideally, when the video is fully edited, the only thing that the therapist has to do to perform all the export steps is press a single button.

4. Responsiveness — The user interface must reflect the intended effect of the user’s actions as soon as possible, preferably immediately. The time limit for giving the user the feeling that their action has an instantaneous reaction is approximately 0.1 second [17]. Waiting longer than that will cause the user to feel that they are not operating directly on the data.

2.2 Design considerations

The functional requirements listed above are mutually dependent: design choices made to satisfy one requirement influence what choices can be made to satisfy other requirements. In this section these choices and their implications are analyzed.

2.2.1 Organizational restrictions

Computing environments in many clinical settings do not allow installation of unapproved software. A number of possible solutions exist to allow the editor to be used with this restriction:

Approval

By getting approval to install the editor it can be freely used. In order to be approved, however, the editor and all of its dependencies need to be audited. This is an economically costly endeavour because any video processing tool used by the editor would be a relatively large piece of software; auditing it requires many expensive (w)man-hours. It is safe to say that it is cheaper to pay someone to assist with production of PCIT-VR videos than it is to have the editor audited. Therefore, getting the editor approved for installation is a non-starter.

Portable application

To get around the approval for installation it is possible to use a so-called portable application. A portable application is completely self-contained and does not require the installation of files: they can be stored on a USB flash drive or in the user’s home directory, from which the application is run directly. However, using a portable app requires that the organization has not implemented a policy that only allows execution of applications when they are located in a set of privileged locations — to which external drives and home directories do not belong. Because day-to-day operation does not require the execution of portable applications it is reasonable to assume that this policy has been implemented to prevent malware. Therefore, implementation as a portable application does not allow use of the editor either.
Dedicated PC

A dedicated PC that is allowed to run native code can be used. However, the therapist’s employer would have to provide the device, for which budget must be allocated. Additionally, the machine must be set up in a physical location where it can run undisturbed during the export process. The device also needs to be sufficiently separated from the network to which regular workstations connect to ensure that the required level of security is maintained. Satisfying these requirements takes significant investment on the part of the organization. Using a dedicated PC is therefore not feasible either.

Web application

A web browser is installed on the workstations available to the therapist. By implementing the editor as a web application, no provisions have to be made on the part of the therapist or their organization. With this design, though, the way that video is processed must be considered: processing can either be done locally or remotely. For both local and remote processing, access to the video files is required. The web platform provides the application with a way to access the user’s file system by requiring the user to manually select files \[15\]. However, access to the files does not persist between sessions, i.e. files on the user’s file system that were accessible before the application is closed are not accessible when the application is opened again. An API is currently being designed to allow persistent access to files on the user’s file system \[22\]. As the API is still being designed we cannot use it for the editor. The implications of file access for local and remote processing are described next.

For local processing, because file access is not persistent, the user needs to select the same files every time the editor is opened if no accommodations are made to prevent this. Requiring the user to manually select the same files over and over is unacceptable. Therefore, a way to gain persistence of the data contained in the files is required. Persistence can be achieved with a number of APIs that allow data to be stored. However, regardless of which API is used, when the amount of stored data exceeds a certain limit, the browser will start deleting data until the limit is no longer exceeded \[19\]. On some browsers the storage limit is correlated with the amount of free disk space. As a result, while the limit cannot be exceeded at the moment at which data is stored, it can be exceeded when free disk space decreases and the limit is adjusted accordingly. In the long run, as the number of stored videos increases, it is likely that the the storage limit is reached. Then, when for some reason free disk space decreases, the videos are removed, resulting in unacceptable data loss. The only way to do local processing on the web platform is by using APIs that are subject to this behaviour, therefore local processing is not a viable option.

With remote processing, storage and access of files is straightforward. To store local files, after being made available by the user, the application uploads them to a remote server. The application can then, for example display the video using only a URL. Remote processing has a few implications for user friendliness related to for example uploading and exporting. These implications are described in the relevant section.

2.2.2 Editing requirements

The editor must enable the user to perform all the necessary tasks for editing PCIT-VR video. The user interface of our editor is inspired by the one used by iMovie, shown in Figure 2.2 Many people consider iMovie easy to use, and it is possible that the therapist is already familiar with it. Three basic user interface elements are required to satisfy all the editing requirements: an asset library, a timeline, and a video preview. The rest of this subsection describes how these user interface elements satisfy the editing requirements and what considerations must be made to satisfy other requirements.
Figure 2.2: The interface design is inspired by that of iMovie, which many people consider easy to use.

Asset library

The asset library contains all the assets on the remote server. An asset is a video file which is the result of recording; it is shown in the asset library with a thumbnail and a name to be able to identify it. The asset library enables the addition of clips to the timeline.

Timeline

The timeline is the key user interface element in allowing all of the editing requirements to be satisfied. It has a cursor to specify the current time and contains a sequence of clips — sections of an asset.

To arrange videos in a sequence, clips are added to the timeline by dragging assets from the asset library onto the timeline.

In order to cut out sections of video, the beginning and end of a clip can be trimmed to specify when in the video the clip should start and end.

Three options exist for adjusting the volume of sections of the final video: manually adjusting the volume of sections of a clip or that of an entire clip; or using dynamic range compression. Adjusting the volume of sections of a clip requires using a graph to adjust the volume over time. Specifying which section of the graph needs to be adjusted requires relatively complex interactions such as adding and also removing the points between which the volume is adjusted at the correct time. Adjusting the volume of entire clips requires cutting up clips at the points in time where the volume should be adjusted. Both of these options are not user friendly: the correct times must first be determined after which the user must match the volume with that of surrounding clips. A solution which requires no user interaction is dynamic range compression [8]. Dynamic range compression processes the audio signal to make the volume at each point in time lie within a specified range, reducing the volume of loud sounds and amplifying soft sounds. Research has been done on parameter automation in a dynamic range compressor [7] which can be used to automatically determine the correct parameters for the final result. Compression can be performed for the intermediate result; however it is possible that different parameters are used for different assets, leading to differences in volume when viewing the intermediate result. Automated dynamic range compression does not require any user interaction, therefore this method will be used for volume adjustment.
To add interactive elements, the timeline cursor is dragged to the correct point in the video and a button is pressed to add an interactive menu. An icon is added above the clip at the current time and a form is displayed in which the contents of the interactive menu are filled in.

Video preview

To see what is being edited and to view intermediate results, a video preview is used. A few controls appear below the preview: a play/pause button and skip backwards and forwards button. The play/pause button causes the preview to start playing or to pause the intermediate result. The preview displays the video frame corresponding to the time to which the timeline cursor points. When the cursor is dragged to a different time, the video in the preview seeks to that time and displays the video frame.

As a result of the remote processing described in the previous subsection, responsiveness is impacted by the need to seek based on user input. This will be described in the corresponding section.

2.2.3 User friendliness

The impact on user friendliness comes from the decision to do remote processing combined with the video preview, and automatic volume adjustment.

Remote processing affects user friendliness both negatively and positively, however the net effect is not negative. For remote processing, video files must be uploaded to the server before they can be used, which means that the user can not immediately start editing new assets. However, during exporting, the editor does not need to be kept open as all the necessary data is available on the server, which is not the case for local processing. A very user friendly consequence of remote processing is that the editor is agnostic to on which workstation it is being run, i.e. if the therapist uses the editor in office A on day 1 they can continue working on the same video in office B on day 2 or even from home on day 3.

As will be explained in the next subsection, the video shown in the preview has a lower bitrate than the original video. The transcoding process required to produce the lower bitrate video adds to the time before a new asset can be used and thus has a negative impact on user friendliness. The lower bitrate video preview itself does not affect user friendliness, however, as long as the user is not annoyed by the lower-quality video shown in the preview.

The automatic volume adjustment described in the previous subsection benefits user friendliness as the user does not need to perform any actions to ensure that volume is even throughout the video. Side note: the user does need to be aware of the fact that the volumes heard in the intermediate result may differ from those heard in the final result though.

Overall, the choices that have been made do not negatively impact user friendliness.

2.2.4 Responsiveness

Given the decision to create a web application that uses remote processing, we need to consider the fact that the responsiveness of the editor suffers as a result of remote file access. The requirement that the intended action of the user is performed within 0.1 second must be met.

Because the video file must be transferred over the network, moving the timeline cursor will not cause the video preview to update instantaneously. The time before the video preview updates after the user has moved the timeline cursor depends on the performance characteristics of the network and the bitrate of the video that is used for the preview. While the performance characteristics of the network are out of the control of the editor, it is possible to lower the bitrate of the video. To achieve the responsiveness requirements, the original video is transcoded to a lower bitrate video, to be used for the preview. An appropriate bitrate must be chosen depending on the network conditions to be able to update the video preview within 0.1 second.

It is important to emphasise that the lower bitrate video is only used for the video preview. The original, high-bitrate videos are used to create the final product.
2.3 Application design

The analysis of the design choices in the previous section have resulted in the following design decisions. The editor will be implemented as a web application using the client/server model. Figure 2.3 shows the expanded data flow of the editor. Assets are uploaded to the server, after which they are transcoded into videos of the appropriate bitrate for use in the video preview. Editing is done with an asset library and a timeline containing a sequence of clips, to arrange and trim clips. Adding interactive elements can be done at any time in the editing process using the timeline. Volume adjustment happens entirely automatically during export, without user input or predefined parameters. During export, the final results are generated, packaged, and made available on the web. Only simple interaction is required to perform any of these tasks.

Figure 2.3: Expanded data flow of the PCIT-VR editor. Red components are client-side. Blue components are server-side.
CHAPTER 3

Implementation: Editor

The editor is a single-page web application written in HTML5, SCSS, and TypeScript. A goal for the implementation of the editor was writing code that is as clean and easy to understand as possible.

3.1 UI layout and styling

The HTML is not very complex: all of the necessary markup is contained in a single file of 174 lines (including whitespace). The rules in the SCSS files define the layout and style of the editor. The CSS file is more elaborate than the HTML file with 690 lines (including whitespace). To achieve clean and easy to understandable code the layout and style were written in SCSS and no scripting was used for layout and styling. The SCSS language is a superset of CSS and is compiled to it. Writing large stylesheets is less cumbersome in SCSS because it has features such as modules and variables. No scripting is used for layout and styling except to provide user interactions that require dragging.

3.1.1 HTML: principles

The relatively simple markup is achieved by adhering (with some exceptions) to the principles of semantic HTML \[13\] and the separation of content and presentation \[3\] which respectively dictate that by looking at the markup alone it should be clear what information the page conveys and that the markup should not be concerned with layout and how it is styled.

Take for example the following snippet of code and how it is displayed after the button has been clicked in Figure 3.1:

```html
1 <div class="asset-browser menu">
2  <div class="title">Assets</div>
3    <div class="left">imaginary</div>
4      <div class="left">imaginary</div>
5    <div class="set add-asset">
6      <div class="button"><i class="icon">add</i></div>
7        <span class="add-asset-name-label">Name:</span>
8          <input type="text" value="Upload" />
9      </div>
10 </div></div>
```

15
For most elements it is clear what it represents: line 1 corresponds with the entire top segment — the menu —, line 7 with the button on the right, line 8 with the popover which appears on the lower right part of the image, etc.

**Violating the principles**

In some cases it is necessary to violate the principles to produce better code. The title (line 2) and buttons (line 3) HTML elements must remain in the same order while they have no ordering, semantically. Requiring this specific ordering is arguably a violation of separation of content and presentation. However, allowing both orderings will complicate the stylesheet, therefore, only allowing one ordering seems a reasonable tradeoff.

In a similar vein, the set element (line 6) can be argued to be a violation of both principles. Semantically, it would be more correct if the hierarchy at line 6 were replaced with the following:

```html
<div class="button">
  <i class="icon">add</i>
  <div class="popover">
    ...
  </div>
</div>
```

However, this markup requires scripting to be involved for layout of the popup. Scripting layout is laborious and easily leads to bugs as the correct positioning must be manually determined and updated when elements resize. In this case the stylesheet is easily adapted to account for the two cases of a button on its own and it being contained in a set:

- **Single case:**
  ```css
  .menu > .buttons > div > .button
  ```
- **Both cases:**
  ```css
  .menu > .buttons > div .button
  ```

The chosen solution is a violation of separation of content and presentation, leading to the violation of the principles of semantic HTML. This is a small price to pay in order to keep layout unscripted. Overall, the principles result in code that is easier to reason about, however dogmatically adhering to them results in more complicated code (and more bugs).

**3.1.2 CSS: Grid layout**

CSS grid layout enables the editor to be built using the compact HTML which adheres to the principles described previously. Grid layout places elements in a 2-dimensional grid. The markup contains only semantically relevant elements, as opposed to them being contained in e.g. a table, and is agnostic to the layout. An example of the layout produced by such HTML and CSS can be seen in Figure 3.2. A wireframe showing the layout of the user interface of the editor is shown in Figure 3.3. The code which is used to produce the layout of the user interface of the editor can be found in appendix A.
Figure 3.2: Example of layout created using CSS grid layout. The code on the left produces the layout on the right.

Figure 3.3: Wireframe of the editor user interface
Figure 3.4: (a) Desired behaviour (b) Single grid behaviour

The types of layouts that can be created by using a single grid is limited. For example, it is not possible to create the layout in figure 3.4 (a) using a single grid. Consider shrinking the layout in figure 3.4 (b) with two cells of fixed height A1 and B2: rows 1 and 3 will not resize, as A1 and B2 have fixed heights; when the grid shrinks, the only row that can shrink with it is row 2; when B2 is almost at the same position as A2, row 2 will have shrunk to its minimum height and the grid will not be able to shrink any further.

To get the layout in figure 3.4 (a) it is necessary to introduce a second grid element which contains either A1 and A2 or B1 and B2:

```
<div id="container">
  <div id="a1">A1</div>
  <div id="a2">A2</div>
  <div id="b">
    <div id="b1">B1</div>
    <div id="b2">B2</div>
  </div>
  <div id="c">C</div>
</div>
```

3.2 Scripting the editor

A design goal for the implementation of the editor is reducing the likelihood of bugs occurring.

3.2.1 TypeScript

The editor is scripted with the TypeScript language which help prevent certain classes of bugs. The language adds a statically checked type system, including generics, to JavaScript. The advanced type system allows developers to be more certain of the correctness of code. For example, the language helps ensure that errors are handled. Performing an RPC can return an error value or the desired value. Before the desired value can be used for further processing, the code must check if the value is an error and handle it. The type system can statically check whether the error has been handled. A compile time error occurs if the error has not been handled. Similarly, access of object properties is checked at compile time too: it is not possible to access non-existent properties. Accessing such a property results in the value undefined. Performing operations on undefined results in an exception being thrown. When the value is used in close proximity to where it was introduced, the exception is useful in determining where the bug is. However, it can be hard to track down the bug when the value has been passed through the entire system. The compiler can make the developer aware of where such bugs occur, provided that type signatures are correct. TypeScript ensures that type-related bugs do not occur, which gives the developer more confidence in their code.
3.2.2 Architecture

The PCIT-VR editor has been organised according to the model-view-controller pattern.

The model-view-controller (MVC) pattern guides how code dealing with user interfaces is organised [6]. As the name suggests, code is separated into three parts: models, views and controllers. The model is responsible for managing the data. The view represents the user-facing part of the system and perform actions on the UI to display the data stored in the models. The model and view need to be consistent with each other. This is handled by the controller, which mediates updates of the view in response to changes in the model, and vice-versa.

The code of the PCIT-VR editor client is organized according to the architecture in Figure 3.5. Conceptually, the models, views and controllers shown in the figure can respectively be combined into one. However for ease of development they have been split up into smaller components.

Dynamic content

In the editor, user interface elements such as assets and clips are dynamically created. Dynamic content can provide an attack vector for hackers, therefore care must be taken when dealing with it, especially when the application is run in an environment that deals with sensitive data.

Some methods for dynamically creating a hierarchy of HTML elements, such as the createElement() function when combined with the innerHTML property, can be vulnerable to injection attacks [11]. The template element [14], combined with the textContent property [12] can be used to avoid vulnerability to injection attacks.

In the editor, the template element and the textContent property are used for all dynamic content. While security is not a big concern for the editor currently — the only dynamic content is based on input provided by the therapist — in the future functionality may be added whereby a parent can provide input which is read by the therapist, thereby allowing injection attacks to be attempted. That is not to say that a parent will have malicious intent, however it is possible that the parent’s device or credentials are compromised as a result of a targeted attack.

3.2.3 Server interaction

The editor must interact with the server in order to affect state and be notified of changes in state. Two mechanisms are used for this: remote procedure calls and events.
Remote Procedure Calls

When the user performs an operation which cannot be done locally, the editor delegates the operation to the server with a remote procedure call (RPC). All editing operations such as adding a clip and trimming it can be done without contacting the server. Loading or modifying the asset library, accessing the assets themselves, and loading, saving and exporting a project require delegation to the server.

RPCs are performed by using a RESTful API (on top of the HTTP protocol), using JSON to transfer data in the body of a request or response. The body of requests is fully dependent on the RPC. The response body follows a common format, however:

```
{ "result": "ok", "payload": <data> }
```
or

```
{ "result": "err", "reason": <error> }
```

Because RPC responses use a common format, the code that implements RPCs can be simplified with an abstraction that is capable of handling all RPCs. Each RPC is implemented using the same code that handles parsing and dealing with errors, including recursive RPCs.

The following code is sufficient to implement the RPC that returns a list of all projects:

```typescript
function parse(json: JSONValue<{ id?: JSONValue; name?: JSONValue; }>): Model.Project | undefined {
    let obj = JSONValue.getObject(json);
    if (obj === undefined) return;
    let id = JSONValue.getNumber(obj.id);
    let name = JSONValue.getString(obj.name);
    if (id === undefined || name === undefined) return;
    return new Model.Project(id, name);
}

export class List extends ResEndpoint<Model.Project[]> implements Endpoint {
    readonly url = "projects";
    readonly parse = parseArray(parse);
}
```

By choosing the right abstraction, code duplication and the number of special cases can be minimized or even eliminated. This allows reducing code to its essence, thereby becoming more understandable, which in turn decreases the likelihood of bugs being present in the code.

Events

In order to keep the state of clients consistent with the state on the server, a mechanism for keeping the state of the client up to date is required. We will call a change of the server state an event. Examples of events that occur are the addition of an asset to the library or creation or progress updates of a job. We will consider two methods for receiving events: polling and pushing.

When polling, the client periodically requests information on the resources that it is interested in. To maintain a sense of immediacy to the user, the period between requests must be sufficiently short. When no events have occurred since the last time the server was polled a certain amount of network traffic is wasted.

The simplest way to implement polling is by requesting all the information on relevant resources and extracting events from that. This strategy will cause a relatively large amount of wasted network traffic, especially when there are no updates. This causes a delay before the user will be able to become aware of events. Additionally, processing the data to extract events will complicate the client code, increasing the likelihood of bugs being present.

To solve these problems, a different strategy is to let the client explicitly request the events that occurred after it previously polled the server. This vastly reduces the amount of wasted traffic and simplifies the client code. However, the server code will be more complicated, as the
server must maintain a list of events and implement logic to purge events from the list once they have been received by all interested clients. Consequently, a reliable list of active clients must be maintained, which is hard to do with HTTP connections.

Instead, events are pushed over a WebSocket connection. A WebSocket connection is bidirectional and remains open for the duration of the session of the client. Therefore, the server can send events to connected clients the moment they happen. This means that network traffic is only caused by events being sent, instead of occurring at a fixed period, often being wasteful. The code on both the client and the server is a lot simpler when events are pushed over a WebSocket connection: the client does not need to perform complicated processing — it simply receives events when they occur; the server does not need to keep track of events — it simply sends them to connected clients when they occur.
The server is composed of several components, as shown in figure 4.1. The backend is written in the Rust, a memory- and thread-safe language with a rich type system \[19\]. As will be shown in this chapter, the type system allows powerful abstractions to be used, resulting in clear and concise code. The memory- and thread-safety guarantees allow the developer to use multi-threading without fear of data races occurring, or undefined behaviour being present. The thread-safety guarantees do not, however, protect against deadlocks.

The following sections describe the components used in the server.

4.1 FFmpeg

The raison d’être of the server is to process video. FFmpeg \[9\] is the tool used to do all video processing tasks: combining assets into the finished video files, and transcoding assets to be...
used in the video preview of the editor. FFmpeg can be used as a library or a stand-alone executable. For ease of implementation the backend invokes FFmpeg as an executable when any kind of video processing needs to happen. To transcode video, the tool only needs to be provided with an input file, the output resolution, and the desired bitrate. To combine assets, FFmpeg needs to know how in what order the input files must be combined, what sections of the video need to be used, and how the result should be encoded. FFmpeg can output multiple files, each with different parameters. Therefore, it is possible to generate all the files required for adaptive bitrate streaming with one invocation of the tool. Writing the arguments that are passed to FFmpeg to combine video is straightforward: the start time and duration is specified for each input file using two flags; a so called filter is used to tell FFmpeg the ordering of the input files and to first create a stream that concatenates the relevant sections of the input files, duplicate the stream into \( N \) copies — where \( N \) is the number of output files — and then scale the video of each stream down to the required resolution; finally parameters, e.g. bitrate, and the output file are specified. The following is an example of such an invocation of FFmpeg:

```
ffmpeg
    -ss 160 -t 30 -i a.mp4
    -ss 180 -t 400 -i a.mp4
    -ss 0 -t 600 -i b.mp4
    -ss 400 -t 300 -i c.mp4
    filter_complex
        [0:v][1:v][1:a][2:v][2:a][3:v][3:a]
        concat=n=4:v=1:a=1
        [vconcat][aconcat];
        [vconcat]split=2[vsplit0][vsplit1];
        [aconcat]split=2[asplit0][asplit1];
        [vsplit0]
            scale=1280:720:
            force_original_aspect_ratio=increase
            [vscale0];
        [vsplit1]
            scale=854:480:
            force_original_aspect_ratio=increase
            [vscale1];
    -map [vscale0] -map [asplit0]
    -b:v 3000000 -b:a 128000
    -hls_time 4 -hls_playlist_type vod
    -hls_segment_type fmp4 -hls_fmp4_init_filename o1-init.mp4
    o1.m3u8
    -map [vscale1] -map [asplit1]
    -b:v 1200000 -b:a 96000
    -hls_time 4 -hls_playlist_type vod
    -hls_segment_type fmp4 -hls_fmp4_init_filename o2-init.mp4
    o2.m3u8
```

### 4.2 NGINX

In the PCIT-VR server, NGINX [16] is used for serving static files such as assets and HTML, and to pass on RPC calls and WebSocket connections to the backend. All connections to the server go through port 80 and are handled by NGINX. Depending on the path specified in the HTTP request, NGINX forwards the request to the RPC server, the WebSocket server, or serves the appropriate static file.

While the WebSocket protocol is different to the HTTP protocol, they share the same handshake [21]. This allows NGINX to pass an incoming WebSocket connection on to the WebSocket server. Once the connection has been passed on, NGINX is agnostic to the data which is transferred between the client and WebSocket server.

NGINX allows the RPC server to specify a static file in its response which NGINX will then serve. This means that the RPC server does not need to be concerned with efficiently serving large files.
4.3 Remote procedure calls

RPCs are handled by a REST API, built with the Rocket framework [18]. The framework relies on various features of Rust to make it possible to compactly specify how requests should be handled.

For example:

```rust
 #[get("/projects")]
 fn get_projects(conn: db::Conn) -> Res<Vec<Project>> {
     use crate::schema::projects::dsl::*;
     projects.order_by(name).load::<Project>(&*conn).into()
 }
```

This request handler is automatically called when a request is made for "/projects". When the handler runs a database connection is automatically opened. The return type causes the list of projects to be serialized into JSON according to the RPC protocol.

This declarative style makes it easy to write server code and requires no boilerplate.

4.4 Jobs

The tasks that FFmpeg performs are represented by jobs. When the client has uploaded an asset or requests exporting to start, respectively, a transcode or export job is placed in a FIFO queue. When a job finishes running, the next job in the queue is started. Jobs are executed by a separate thread that monitors the progress of FFmpeg. Progress updates are sent to the client as events over the WebSocket connection.

4.5 Events

When the client must be made aware of a change in the server’s state, an event is sent. The event dispatcher takes care of broadcasting the event to all the connected clients on its own thread. Code that wishes to send an event, uses a channel to send the event to the dispatcher for broadcasting.

4.6 Database

Interaction with the database is done using the Diesel framework, an object-relational model (ORM) [4]. The developer writes models that describe and contain data; the framework uses these models to generate queries, send them to the server, and return the result.

The burden of painstakingly binding values to handwritten queries and extracting values from results is lifted from the developer. The framework ensures that the correct types are used when binding and extracting values. The intent of code that interacts with the database through the ORM is immediately clear:

```rust
diesel::insert_into(projects)
 .values(&*project)
 .get_result::<Project>(&*conn)
```
CHAPTER 5

Results

Figure 5.1 shows the PCIT-VR editor. Most of the functional requirements in section 2.1 have been satisfied. The editor has been implemented as a web application, using a client/server model. The user can upload assets, arrange videos in a sequence, cut out sections of video, add interactive elements, preview video and export the finished video. The appearance and user interactions are comparable to those of iMovie, on which they were based, though lacking its polish.

Figure 5.1: The PCIT-VR editor. The visual design is heavily borrowed from Apple iMovie.

The requirement of responsiveness, which depends on being able to update the video preview sufficiently fast in response to user input, is satisfied by adjusting the bitrate of the video preview when the editor is deployed, depending on the network conditions.

A few features that are needed to satisfy the remaining requirements have not been implemented; however, this is not because the features cannot be implemented, but rather due to a
lack of time. Three features that contribute to user friendliness still need to be implemented: clip thumbnails, scaling the timeline, and showing the timecode. Implementing these features is for the most part a matter of writing some more code for the UI, especially for the last two cases: the underlying support for scaling the timeline is already present, the functionality just needs to be exposed by a control; showing the timecode only requires a text element to reflect the time to which the timeline cursor points. Two features related to the editing requirements have not been satisfied either: the ability to pan the 360-degree video in the preview and automatic volume adjustment. In general, it is possible to pan 360-degree video in a browser, as evidenced by the A-Frame VR framework [2]. However it will require some implementation work to support panning in the editor. The code for automatic dynamic range compression is available, however it is implemented as a VST plugin, which FFmpeg does not support. Still, using the VST plugin is possible; it just requires a bit of plumbing.
CHAPTER 6

Conclusion and future work

6.1 Conclusion

In this thesis, we evaluated the considerations that must be made when designing and implementing an easy-to-use video editor for PCIT-VR. The result of the research project that is described in this thesis is a web-based video editor for PCIT-VR which can be used by therapists.

The web-based user interface achieves a very similar appearance and method of interaction as a native application. However, accessing files fast enough to provide the same responsiveness as a native application is not possible. A tradeoff must be made between responsiveness and quality of the video preview.

It was possible to satisfy all the design requirements for a PCIT-VR editor using a web-based application with a client/server model. The video editor that was created as part of this project lets therapists easily produce interactive 360-degree videos for use in PCIT-VR.

6.2 Limitations and future work

Instead of using a fixed bitrate for the video preview, it may be possible to use adaptive bitrate streaming. Adaptive bitrate streaming is a technology which dynamically switches between videos of different bitrates, depending on the network conditions. This technology could enable deployment of the editor without first having to measure network conditions to determine the appropriate bitrate for the video preview.

The design of the editor delegates all processing tasks to the server. An example of a limitation of this design is described in subsection 2.2.2: the volume adjustment of the intermediate result may differ from that of the final result. By implementing volume adjustment in the client as well as on the server, the results will be the same. Because volume adjustment is automatic, this limitation does not have a big impact. However, video processing features that require manual adjustment can not be used if they are not implemented by the client too. To complicate matters even more, the results of both implementations must be exactly the same.

When the writable files API [22] is available, it will be possible to gain persistent access to files on the user’s file system. By using this API it is feasible to do local processing if future WebAssembly features like multi-threading and SIMD are sufficient for achieving reasonable performance.

Security was not an explicit design goal for the editor, as shown by the following example. It is safe to assume that arbitrary code execution (ACE) vulnerabilities exist in FFmpeg, as it is a large piece of software which deals with complex file formats and is not written in a memory-safe language. An (evil parent) attack is thinkable, where a maliciously crafted file is added to the SD card of the camera used to record therapy sessions, after which the file is uploaded and processed by FFmpeg. Once the attacker has control of the server, they can start exploiting vulnerabilities down the chain. Isolating FFmpeg from the rest of the system is not a watertight solution: the exploit can produce a malicious video that can be used to exploit a vulnerability in the browser.
To ensure that the user adds the asset to the timeline, which causes the malicious file to be loaded in the video preview, an enticing thumbnail is created. Replacing FFmpeg with a video processing tool written in a memory-safe language is, again, not a water tight solution. While memory-safe languages do vastly decrease the likelihood of ACE vulnerabilities, there is still the possibility that one exists. Similar to the possibility that the attack described above is actually performed. More reasonable attacks using different vulnerabilities may be possible though.
References


Appendices
APPENDIX A

HTML and CSS used for layout

The code snippets in this appendix are used in the editor; they have been edited for brevity.

A.1 HTML

The following HTML specifies the sections of the user interface:

```html
<div id="base">
  <div id="main-menu" class="menu"></div>
  <div id="view-container">
    <div class="editor view">
      <div class="asset-browser menu"></div>
      <div class="asset-browser container"></div>
      <div class="player">
        <div class="content"></div>
        <div class="controls"></div>
      </div>
      <div class="composer toolbar menu"></div>
      <div class="composer container"></div>
    </div>
  </div>
</div>
```

A.2 CSS

The following CSS specifies the layout of the sections:

```css
body, html {
  height: 100%;
  margin: 0;
  padding: 0;
}

#base {
  width: 100%;
  height: 100%;
  display: grid;
  grid-template-rows: min-content 1fr;
}

#main-menu {
  grid-row: 1/2;
}

#view-container {
  grid-row: 2/3;
}
```