Internship Report

Jonas Gann (Matrikel-Nr.: 3367576)

September 2021

Contents

1 Introduction

This is a report about the advanced internship "Frontendentwicklung in React" (Frontend Development in React). The purpose of the internship was to gain experience in developing applications with the frontend development framework for web-apps called React. As part of the internship, the development of an application was given as a task. The application is a common tool in video editing software: A visualization of marker-movements through rendering of corresponding tracks.

The report introduces the topic by presenting an already existing software solution, describes details about the implementation of the new application and explains the reason for software design decisions. The functionality of the application in regard to its main use case is described. The report closes with an outlook on further improvements.

1.1 Related Work

Powerful video editing software like Blender contain tools for processing movement of features in a video. This can be useful to achieve special effects such as video stabilisation [1], centering of moving objects [5] or insertion of 3D object in a scene [4].

However, to process movement in a video, the software requires markers which indicate the position of features in each frame. The resulting marker-tracks describe the motion of the objects in the video or the motion of the camera.

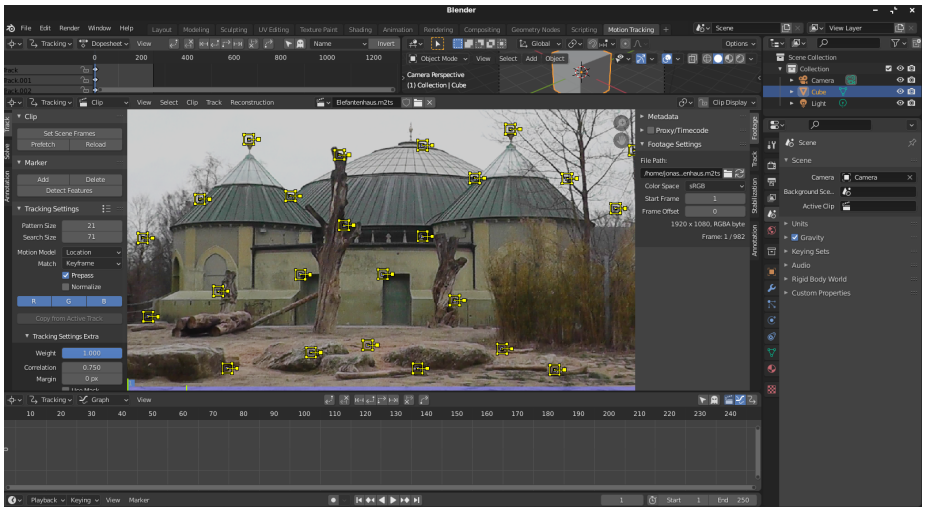

Figure 1: Marker Selection in Blender

It is important to select suitable markers which can be tracked throughout the video and which stand out from the background. Blender enables its users

to automatically generate an initial set of markers based on specifications about the preferred number of markers and their distance to each other. A resulting selection can be seen in figure 1.

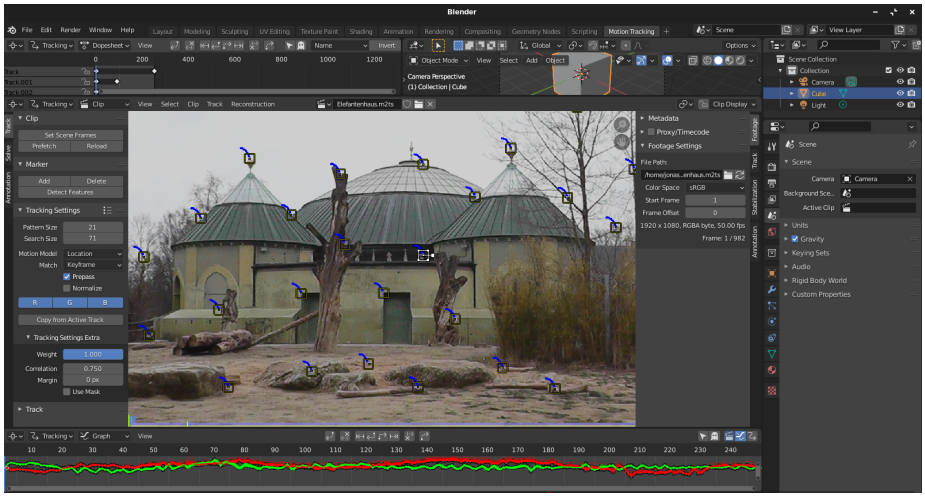

Figure 2: Marker Tracks in Blender

Instead of manually placing each marker for each image, Blender can automatically create marker-tracks as shown in figure 2. The previously described special effects rely on a high quality motion tracking. It therefore is important to verify the quality of the marker-tracks and when necessary adjust marker positions. In figure 2, for each marker, Blender visualizes its marker-track. If the features in a video are static, uniform movements of the markers and thus similar tracks can be expected when the camera is panned. Deviating tracks indicate errors in the underlying motion capture and can be detected and corrected by the user.

As comparing marker tracks distributed across the image is not ideal, Blender additionally displays a graph at the bottom, visualizing the speed of all trackers while the green line is vertical movement and the red line is horizontal movement [2]. Figure 3 shows the graph in more detail.

By looking at this graph, the user can immediately see the quality of the tracking by the density of the red and green lines. The closer green lines are together (and red lines respectively), the more similar marker movements are. Deviating marker movements such as on time index 60 can be quickly discovered and manually corrected.

1.2 Project Description

The focus of the application which should be developed as part of the internship is the visualization of marker movements through marker tracks, as shown in

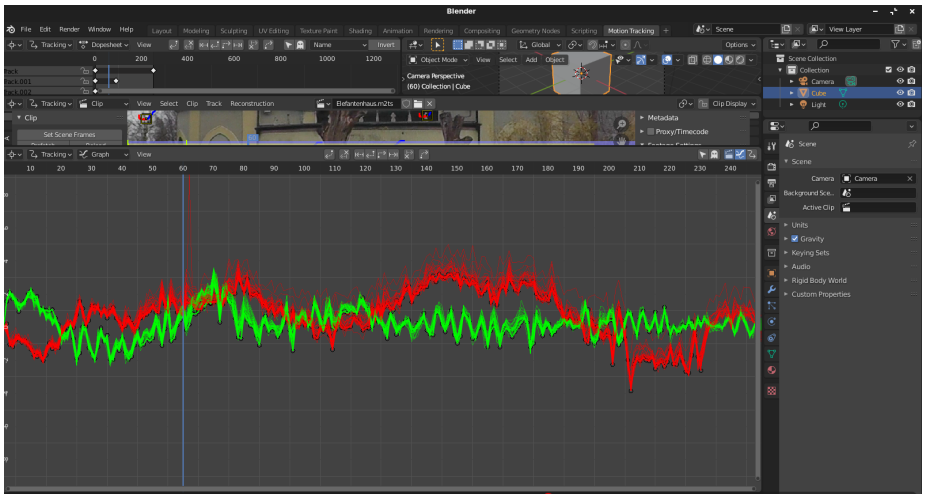

Figure 3: Marker Speed Graph in Blender

figure 2, and the modification of markers to correct possible errors. The application should be able to receive as input a video in the form of an image sequence as well as a motion capture which contains a list of markers and their positions in the image sequence. As shown in subsection 1.1, the application should visualize markers and marker movements through marker-tracks. The image sequence, along with markers and marker-tracks, should be played back as a video. This requires options for starting and pausing the playback. The application should enable the user to modify the position of existing markers and to add or delete markers. It should be possible to download modified markers as a file for later usage.

2 Implementation

The development of the application can be separated into seven key challenges. Each of which will be covered in this section. Each subsection describes a challenge and how it was solved, as well as reasons for design decisions.

2.1 User Interface

Challenge The first challenge I faced when developing the application was to design a suitable user interface for the features mentioned in the project description. The UI is not only important to make the application easy to operate by the user, but it also determines the possibilities on how to separate the application into components.

I decided to design the application with the look and feel of a desktop app. The reason is, that applications for video editing usually are executed locally

on desktop computer and not on mobile phones or through the web. The main difference between typical web pages and a desktop application is the full utilization of the available width and height of the window. This means, the user interface should automatically align to the full window size while displaying all UI elements without the need for scrolling.

Implementation As shown in Figure 4, the user interface consists of a canvas which plays back the loaded video and marker tracks and two menus, one on the bottom to control the behavior of the playback and one on the side to load, save and edit relevant data.

Figure 4: User Interface of the Application

To implement this layout, I utilized CSS grids: A grid separates the left (canvas and bottom menu) and the right (files and images menu) side of the user interface. Each half of the grid, again, separates its upper half from its lower half through a grid. The grid layout was configured to adapted to resizing of the window by resizing the canvas and leaving the menu size constant. The content of the canvas therefore had to be updated to automatically fit the new canvas size. This will be described in more detail in a following subsection. CSS grids were a suitable choice for this kind of layout, as they simplify the utilization of the whole width and height of a window instead of extending content through scrolling as it is the case in conventional web pages. Separating the UI elements with grids also fitted the notion of separating the UI into React components, as described in the following subsection.

2.2 Components

Challenge Based on the previously described user interface, the application has to be separated into React components. Thoughtful definition of components improves performance and maintainability.

Implementation The structure of the grid layout was used to structure the React components. Figure 5 shows how the user interface is separated into components, giving an overview of which components exist and what their purpose is. Besides the underlying "app" component, a "canvas" and "editor" component exist. The "canvas" component contains a "Konva" (red) [3] component - a library for drawing images and simple geometry - and a "canvas control" (purple) component. The "editor" component contains a "load files" (blue) component and a "display files" (turquoise) component. The "display files" component contains multiple instances of the "file entry" (green) component, which contains multiple instances of the "marker entry" (yellow) component.

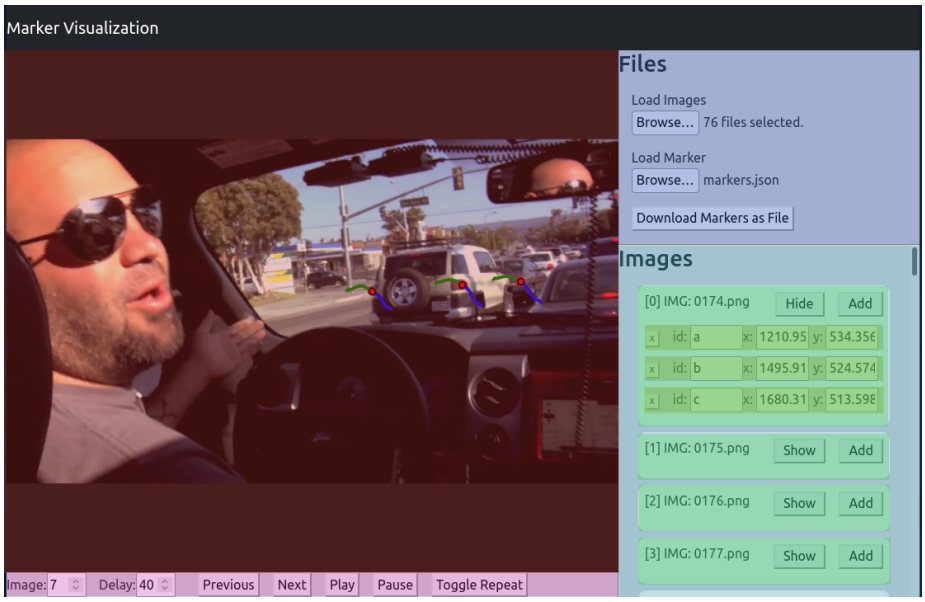

Figure 5: Visualization of React Components

The "canvas" component is responsible for playing back the image sequence as a video, as well as selecting markers connected to the viewed image and calculating marker-tracks. Through props, the "Konva" component is instructed by the "canvas" component to do the actual drawing. Through a callback function, the "canvas control" component instructs the "canvas" component which image and therefore which markers and marker-tracks to display. The "editor" component is responsible for managing the application data. Through callback functions, the "load files" component transmits loaded data to the

"app" component. Through props, it receives the current state of marker data and provides it for download as a file. The "display files" component as well as its children receive the current state of loaded images and marker as well as marker modification functions as props, combining them to provide marker modification capabilities. Each component and its implementation will be covered in the following subsections, in more detail.

2.3 Data Loading

Challenge The application should be able to load and processes two types of data: images and markers. Through a user interface, the user should be able to select images and markers to load into the application. Loaded resources should be made available to all application components. Upon changing images or markers, the application should render dependent components again.

Implementation The application provides a "Files" menu to the user (see figure 4) which is located on the upper right side. Images and markers can be loaded from the file system of the device, and markers can be downloaded to the file system by pressing the corresponding button. An HTML input tag, as shown in figure 6, is used to open the file system dialog of the operating system. Through the "onInput" callback, files selected by the user are submitted to the "handleLoadImages" function.

```
<input
    type="file"
    multiple
    accept="image/png, image/jpeg"
    onInput={(e) => handleLoadImages(e.target.files)}
/>
```

Figure 6: Input Images Code Snippet

As shown in figure 7, the function asynchronously loads all selected images as a URL. The reason for loading files as URL is that in the next step an Image object is created for each loaded file and the Image object expects its image source to be a URL or file system path. As through the browser, due to security restrictions, the application is not able to retrieve the file system path of a loaded file, the URL option has to be chosen. An Image object is created, as it is necessary in order to display it on the canvas. As creating the Image object turned out to be an expensive operation, it should only be done once after loading the image files. To utilize the loaded images as an image sequence, as a last step, the application sorts them based on their file name. The process for loading the marker file is similar, however somewhat easier, as it is only a single text file.

```
const handleLoadImages = async (files) => {
    let awaitList = [];
    let nameList = [];
    // read all selected files as an URL
    for (var i = 0; i < files.length; i++) {
        // asynchronously load image file
        awaitList.push(readFileAsyncURL(files[i]));
        // in same order as images, store image name
        nameList.push(files[i].name);
    }
    // wait for all load operations to resolve
    const urlList = await Promise.all(awaitList);
    awaitList = [];
    const imageList = [];
    // asynchronously compile all loaded images to the Image object
    for (var e = 0; e < urlList.length; e++) {
        let image = new Image();
        imageList.push(image);
        awaitList.push(
            new Promise((resolve, reject) => {
                image.onload = () => resolve();
            })
        );
        image.src = urlList[e];
    }
    // wait for all operations to resolve
    await Promise.all(awaitList);
    // add image name to images
    imageList.forEach((image, index) => {
        imageList[index].name = nameList[index];
    });
    // sort images by name
    imageList.sort();
    // transmit images to "app" component through callback
    handleNewImages(imageList);
};
```

Figure 7: Load Images Handler Code Snippet

As most application components require access to the loaded data, the "load files" component hands the loaded data over to the "app" component through a callback function, from where it passes both objects to child components as props. The "app" component stores an images list and a markers object as its state. Modifying the markers object, therefore, results in a new render of dependent components.

2.4 Markers

Challenge Markers are used to track the position of image features, which can for example be the red backlight of a car, through a sequence of images. A suitable data format and data model to store, load and process this data has to

be defined.

Implementation The purpose of the data format is to enable the application to easily understand the structure of the data. As it is common in JavaScript application, the JSON format is used, as it provides good integration with the JavaScript language. For the data model the main design factor were four observations:

1. Markers require a unique ID to distinguish between them
2. Markers describe a point in two-dimensional space and therefore require an X and Y property
3. Each marker can occur in at least one or more images (usually in all images, never in no image)
4. Each image can contain multiple markers

As a result of the first two observation, an individual marker, as shown in figure 8 contains an ID and an x and y property.

```
const marker1 = {
    id: "a",
    x: 100,
    y: 100
}
```

Figure 8: Marker Code Snippet

To connect markers and images, image objects as shown in figure 9 can be created which store connected markers in a "markers" property.

```
const image1 = {
    name: "image1.png",
    markers: [
        "a": {
            x: 100,
            y: 100,
        }
    ]
}
```

Figure 9: Image Option 1 Code Snippet

Another option would be to further expand the marker object by adding an "images" property, as shown in figure 10.

```
const marker1 = {
    id: "a",
    images: {
        "image1.png": {
            x: 100,
            y: 100
        }
    }
}
```

Figure 10: Image Option 2 Code Snippet

I decided to use the first option, as it increases the performance when playing back the image sequence as a video. In contrast to the second option, when displaying an image, the connected markers are easily accessible through the "markers" property and do not have to be searched. To improve the random access of markers for a selected image, image objects are stored in an object with the image name as key, as shown in figure 11. The indirection through the "markers" property can be removed.

```
const markers = {
    "image1.png": {
        "a": {
            x: 100,
            y: 100,
        }
    }
}
```

Figure 11: Markers Code Snippet

The application accesses markers in two ways: To play back the image sequence and to modify individual markers. When playing back the image sequence, the canvas iterates the list of images and displays each image. For the displayed image, the connected markers can be easily accessed through:

```
const markerList = markers[image.title]
```

A marker can be easily accessed if given the image title and the marker ID:

```
const marker = markers[image.title][markerID]
```

2.5 Data Modification

Challenge To add new marker tracks and to correct errors in the motion capture, markers have to be able to be created, deleted and modified. Upon modification of markers, components which rely on the modified data should render again.

Implementation Markers can be modified in two ways: The first and most intuitive way is to drag and drop existing markers to a new position on the canvas. This will be described in more detail in the following subsections. The second way is to change marker data in the "Images" menu on the right side. Here, the user can display and change properties of all markers, delete markers and create new markers. The "app" component defines methods, implementing marker operations (addMarker, setMarker, removeMarker) for changing its marker state.

As shown in figure 12, the "setMarker" function, for example, can modify an existing marker by either changing its ID or its x and y properties. By calling the "setMarker" function at the end, the state of the "app" component is updated.

```
1  // "markers" object as the state of the "app" component
2  const [markers, setMarkers] = useState({});
3
4  const setMarker = (imageTitle, markerID, x, y, newID) => {
5      // Create copy of the markers object
6      const markerCopy = { ...markers };
7      if (!newID) {
8          // Change the values for the existing marker
9          let marker = markerCopy[imageTitle][markerID];
10         if (x) marker.x = x;
11         if (y) marker.y = y;
12     } else if (newID) {
13         // Rename the marker by creating a new marker and
14         // deleting the old one
15         delete markerCopy[imageTitle][markerID];
16         markerCopy[imageTitle][newID] = {
17             x: x,
18             y: y,
19         };
20     }
21     setMarkers(markerCopy);
22 };
```

Figure 12: setMarker Code Snippet

Through props, these methods are passed down to components, enabling them to modify the marker data of the "app" component. As shown in figure 13, "setMarker" can be used in the "onChange" callback of an input tag to

change the "x" value of a marker.

```
const MarkerEntry = ({ marker, markerId, image, operations }) => {
    return (
        <div>
            <p>x: </p>
            <input
                type="text"
                // Display current x value of the marker
                value={marker.x}
                // When input field changes, update the  marker
                onChange={(e) => {
                    operations.setMarker(
                        image.name,
                        markerId,
                        // Set the changed input value as x value
                        e.target.value,
                        marker.y
                    );
                }}
            />
        </div>
    )
}
```

Figure 13: MarkerEntry Code Snippet

As shown in figure 13, the component also receives the marker object from the "app" component as prop. Updating the marker object through the "setMarker" function therefore results in a new render of the component with the updated values.

By defining marker operations on the "app" component which modify its state and by utilizing them in child components, child components can modify marker data and change the state of the "app" component. Therefore, whenever a child component changes the value of a marker through the provided functions, the "app" component and all child components which receive the marker data through props are rendered again.

2.6 Visualization

Challenge As described until now, the application is capable of loading images and markers as well as modifying marker data. However, the focus of the application is the visualization of the data. It should be able to play back the loaded images and, for each image, display markers and marker tracks. In addition to that, the size of the canvas along with its content should adapt to changes of the window size.

Implementation The "canvas" component is responsible for visualizing the data. It utilizes Konva [3] to draw images and basic shapes such as circles and lines. As shown in figure 14, using Konva, the application draws the loaded images. By calling "renderMarkerLine" and "renderMarkers", the component calculates Konva lines and Konva circles to be drawn for visualizing marker tracks and current markers. Resulting marker tracks can be seen in figure 4.

```
return (
    <div className="canvas">
        <div ref={canvasContainer} className="canvas-container">
            <Stage className="canvas" width={size.width} height={
    size.height}>
                <Layer>
                    // view image
                    <Image
                        x={0 + offsetX}
                        y={0 + offsetY}
                        width={width}
                        height={height}
                        image={img}
                    ></Image>
                    // calculate and view marker tracks
                    {renderMarkerLine(10, "green")}
                    // calculate and view markers
                    {renderMarkers(10, "red", true)}
                </Layer>
            </Stage>
        </div>
        <div>
            <CanvasControl
                setCurrentImage={setCurrentImage}
                imagesLength={images.length}
                markerOperations={markerOperations}
            ></CanvasControl>
        </div>
    </div>
);
```

Figure 14: Canvas Code Snippet

To properly scale the canvas, the effect shown in figure 15 is used. Through "useRef" the effect accesses the parent of the canvas component to retrieve its current size. This size is used to set the initial size of the canvas and to update it on each "resize" event. The canvas size is used as the state of the component. Updating the size therefore results in a new render of the component.

```
1  // set size as state
2  const [size, setSize] = useState({ width: 0, height: 0 });
3  // access canvasContainer div
4  const canvasContainer = useRef(null);
5
6  useEffect(() => {
7      // retrieve size of canvasContainer
8      const { clientWidth, clientHeight } = canvasContainer.current;
9      const newSize = { width: clientWidth, height: clientHeight };
10     // Set initial canvas size
11     setSize(newSize);
12
13     const handleResize = () => {
14         // retrieve size of canvasContainer
15         const { clientWidth, clientHeight } = canvasContainer.
       current;
16         const newSize = {
17             width: clientWidth,
18             height: clientHeight,
19         };
20         // Set new canvas size
21         setSize(newSize);
22     };
23     // Update canvas size on each resize event of the window
24     window.addEventListener("resize", handleResize);
25     return () => window.removeEventListener("resize", handleResize)
       ;
26 }, []);
```

Figure 15: Effect Code Snippet

As shown in figure 16, on each render, the canvas component calculates the scale and offset of the image to be displayed correctly for the new canvas size. First, the required scale for resizing the image width and height to the canvas width and height respectively is calculated (line 2 and 3). To make sure, that the image always fits within the canvas and the image is not distorted, the smallest scale is selected (line 4). Based on this scale, the new image width and height are calculated (line 5 and 7). As the image keeps its aspect ratio, either image width or image height do not match the width or height of the canvas. Therefore, an x and y offset are calculated as the difference in height or with between the image and the canvas. To position the image in the middle, half the size difference is added as offset to the x and y value, respectively (line 6 and 8).

```
const img = images[currentImage];
const scaleWidth = size.width / img.width;
const scaleHeight = size.height / img.height;
const scale = scaleWidth < scaleHeight ? scaleWidth : scaleHeight;
const width = img.width * scale;
const offsetX = (size.width - width) / 2;
const height = img.height * scale;
const offsetY = (size.height - height) / 2;
```

Figure 16: Scaling Code Snippet

As shown in figure 17, the calculations are used to determine width, height and offset of images.

```
<Image
    x={0 + offsetX}
    y={0 + offsetY}
    width={width}
    height={height}
    image={img}
></Image>
```

Figure 17: Image Visualization Code Snippet

For every displayed image, the connected markers have to be retrieved and drawn onto the image. Using the marker operation "getMarkers" defined in the "app" component, markers for a given image name can be retrieved as shown in figure 18.

```
const currentMarkers = markerOperations.getMarkers(img.name);
```

Figure 18: getMarkers Code Snippet

Just like images, markers have to be scaled according to the canvas size. The scale and offset values, earlier used for the image, are now also used for the markers, as shown in figure 19. The radius of the circle is also scaled to keep the same size in relation to the image.

```
<Circle
    x={x * scale + offsetX}
    y={y * scale + offsetY}
    radius={10 * scale}
></Circle>
```

Figure 19: Marker Visualization Code Snippet

To enable the drag and drop feature, as shown in figure 20, a callback function is attached to the Konva shape. This function is executed each time the shape is dragged. Using the same marker operations as introduced previously, dragging the marker results in a "setMarker" operation. As in most cases, the marker position retrieved by the drag event is relative to the scaled canvas, the scaling operations have to be undone before updating the marker value.

```
<Circle
    x={x * scale + offsetX}
    y={y * scale + offsetY}
    radius={10 * scale}
    onDragEnd={(e) =>
        // change marker position
        markerOperations.setMarker(
            img.name,
            id,
            // undo marker scaling
            e.target.x() / scale - offsetX / scale,
            e.target.y() / scale - offsetY / scale
        )
    }
></Circle>
```

Figure 20: Marker Drag and Drop Code Snippet

To display marker tracks, for each marker connected to the current image, the x and y values of the marker of past (or future) images are collected in a list as shown in figure 21.

```
// get IDs of the current markers
return Object.keys(currentMarkers).map((id) => {
    let points = [];
    // iterate through past images
    for (let cnt = currentImage - size; cnt <= currentImage; cnt++)
     {
        // select image of current iteration
        const img = images[cnt];
        // calculate markers of selected image
        const currentMarkers = markerOperations.getMarkers(img &&
    img.name);
        // select the marker with the marker ID of the current
    iteration
        const currentMarker = currentMarkers[id];
        if (currentMarker) {
            // push x andy value of the marker to the array
            points.push(currentMarker["x"] * scale + offsetX);
            points.push(currentMarker["y"] * scale + offsetY);
        }
    }
})
```

Figure 21: Marker Track Code Snippet

The list contains the x and y values of markers as an ordered list: [x1, y1, x2, y2, ...]. This list is then visualized as a line by Konva as shown in figure 22.

```
<Line points={points}></Line>
```

Figure 22: Marker Track Visualization Code Snippet

2.7 Playback

Challenge The application is now able to visualize all necessary data. However, some additional features are required to enable the user to utilize the image sequence as a video. This includes a play and a pause button. Buttons for jumping to the next and previous image. And an indicator of the currently displayed image.

Implementation To control the image the canvas displays, a "canvas control" component was added. Through props, it receives a callback function to change the current image (as an index value) of the canvas. The canvas saves the current image as part of its state. Changing the current image therefore results in a render, calculating new scale values and drawing the new image, markers and maker tracks. To implement the playback feature, the "canvas control"

component defines the effect shown in figure 23. This effect increments the current image after a determined interval called "speed". If the interval reaches the last image, the image index is set back to zero.

```
const [imageIndex, setImageIndex] = useState(0);
const [play, setPlay] = useState(false);
const [repeat, setRepeat] = useState(true);
const [speed, setSpeed] = useState(40);

useEffect(() => {
    const intervalID = window.setInterval(() => {
        if (imagesLength) {
            if (play && imagesLength > imageIndex) {
                setImageIndex((oldIndex) => oldIndex + 1);
            } else if (imagesLength === imageIndex && repeat) {
                setImageIndex(() => 0);
            }
        }
    }, speed);
    return () => window.clearInterval(intervalID);
});
```

Figure 23: Playback Code Snippet

Another effect, shown in figure 24, updates the image index of the canvas through the "setCurrentImage" callback function each time the "canvas "control" component changes its "imageIndex" state.

```
useEffect(() => setCurrentImage(imageIndex), [imageIndex]);
```

Figure 24: Set Current Image Code Snippet

Using various input methods, the component enables the user to influence the described playback by pausing the playback, changing the speed value, stopping the automatic playback and specifying the "imageIndex" value.

3 Experimental

In this section, the functionality of the application in regard to its main use cases is presented. The main use case of the application is the detection and correction of errors in a motion capture.

To load motion capture resources into the application, they have to modified to use the data format defined by the application. This requirement greatly decreases usability of the application. However, there are ways to fix this problem,

as discussed in the following outlook section. Once image and marker files are loaded, the user sees a UI similar to figure **??**. A quick look at the motion of the marker tracks shows, that the leftmost track describes a different path than the other tracks. This leaves only two options: Either the object in the image sequence changed position, or the motion capture is faulty. As the leftmost and middle marker describe the position of the headlights of the same car, it is very unlikely, that the difference in marker track motion is the result of a moving object. It can be assumed, that the motion capture is faulty.

Figure 25: Marker Track Errors [4]

The user sees, that the motion of the blue part of the marker track does not match the other marker tracks. As the blue track indicates the motion of the marker in past images, the user presses the "Previous" button, to inspect the previous positions of the marker. And indeed, in previous images, the marker changes its position from one backlight of the car to another, as shown in figure 26. This proves the assumption, that the motion capture contains errors.

Figure 26: Marker Track Errors 2 [4]

To correct the errors, the user can iterate through the images using the "Next" button and drag the marker to the correct position in the image. Once finished, the user can press the "Play" button, to view the image sequence as a video and make sure, that all errors have been corrected. To download and reuse the result, the user can press the "Download Markers as File" button and store the markers on the file system.

4 Related Work

5 Outlook

In this section, possible improvements and new features are described.

5.1 Compatibility

As mentioned in the previous section, an important issue of the application is the "unique" data format it uses. Tools for utilizing motion captures created by other applications would greatly increase usability of the application. The application could for example recognize the data format of widely used motion capture applications and provide automatic formatting. To expand on this feature, the application could enable developers to add new formatting tools through a plugin API.

5.2 Loading Files

To improve the user experience of loading files, the application should display loading indicators, whenever its is occupied with I/O operations.

An additional way of loading image and marker files could be through drag and drop of files onto the application window.

5.3 Error Handling

As of now, the application does not handle errors. A system for catching errors and displaying non-technical error information (error codes, etc.) could be useful.

5.4 Testing

To guarantee correct operation of the application, automated tests should be added.

5.5 Customization

An additional page for customization could be added to the application. Here, the user could optimize the behavior of the application to his workflow.

5.6 Performance

The current method of visualization is based on displaying individual images. Switching between images, each time rendering markers and marker tracks using Konva, uses many resources. To improve performance, the application could distinguish between playback and editing modes: When the video pauses, the application behaves as it is now, enabling the user to drag and edit markers. But when playing back, the application compiles the current version of images and markers to a video file and displays it.

5.7 Canvas Menu

The current method of adding markers is not intuitive. A right-click-menu on the canvas can be added to enable the creation of new markers at the selected point.

5.8 Electron

To improve upon the design as a desktop application, a framework like Electron can be used to create an installable desktop application.

References

[1] *Blender 2.8 Tutorial: How to stabilize shaky/blurry videos*. Website. URL: https://www.youtube.com/watch?v=982RL4a899g.

[2] *Graph View*. Website. URL: https://docs.blender.org/manual/en/latest/movie_clip/tracking/graph.html.

[3] *Konva.js - HTML5 2d canvas js library for desktop and mobile applications*. Website. URL: https://konvajs.org/.

[4] *Master Blender Camera Tracking in 6 Minutes!* Website. URL: https://www.youtube.com/watch?v=kzym73lhmD4.

[5] *MOTION TRACK OBJECTS in Premiere Pro 2021 Quickly Explained — Keep Objects Centered*. Website. URL: https://www.youtube.com/watch?v=rZY5yq9eJ9c.