FACULTY OF DIGITAL ENGINEERING
Computer Graphics Systems Group

Design and Implementation of a Framework for Automated Generation of Content-Aware Video Effects on Mobile Devices

Entwurf und Implementierung eines Frameworks zur automatisierten Erzeugung inhaltsbezogener Videoeffekte auf mobilen Endgeräten

Master's Thesis
in partial fulfillment for the academic degree
"Master of Science"
(M.Sc.)
in IT Systems Engineering

Hasso Plattner Institute // Faculty of Digital Engineering // University of Potsdam

submitted by
Philipp Trenz

Supervision:
Prof. Dr. Jürgen Döllner
Sebastian Pasewaldt

Potsdam,
August 22, 2022

Contents

List of Figures v

List of Tables xi

Listings xiii

Abstract xv

Kurzfassung xvii

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Requirements . . . 3
  - 1.2.1. Functional Requirements . . . 3
  - 1.2.2. Non-Functional Requirements . . . 4

2. State of the Art 7
- 2.1. Human Motion Visualization . . . 8
- 2.2. Multidimensional Video Data . . . 11
- 2.3. Multidimensional Video Processing . . . 12

3. Concept 15
- 3.1. Silhouette-Based Motion Visualization Effects . . . 15
  - 3.1.1. From Videos to Motion Visualization Effects . . . 15
  - 3.1.2. Effect Specification and Design . . . 16
- 3.2. Multidimensional Video Processing Framework . . . 22
  - 3.2.1. Graph-Based Processing Concept . . . 22
  - 3.2.2. Reactivity of Node Ports . . . 23
  - 3.2.3. Graph Nodes for Specialized Processing Tasks . . . 23
  - 3.2.4. Adaptability of Processing Graphs . . . 25
- 3.3. App and User Experience design . . . 27
  - 3.3.1. The App View Hierarchy . . . 28
  - 3.3.2. Intuitive App Interface for Video Effects . . . 30
  - 3.3.3. Timeline-Based Video Effect Creation . . . 31
  - 3.3.4. Customizing Effect Representations . . . 34

4. Implementation **39**
4.1. System Architecture 39
4.2. Implementation of the Multidimensional Video Processing Framework . . 41
4.3. Interactive Parameter Adjustment 51

5. Discussion and Future Work **59**
5.1. Introduction to the MotionViz App 59
5.1.1. Capturing and Storing of Multidimensional Video Data 59
5.1.2. Creating Content-Aware Video Effects 61
5.1.3. Video and Project Export 62
5.2. Evaluation of the Multidimensional Video Processing Framework 65
5.2.1. Effect Evaluation 65
5.2.2. Performance Evaluation 69
5.2.3. Developers' Feedback 71
5.3. Future Work 73

6. Conclusion **77**

Bibliography **81**

A. Developers' Interview **87**

List of Figures

1.1. Video frame from the music video for the song “Wait for you” by Tom Walker with in a video editing program manually drawn motion lines. . . 2

2.1. Video excerpts from the music video of the song “Wait for You” by Tom Walker, illustrating different possibilities of human motion visualization. In Figure 2.1(a), motion in the video is annotated by action lines. Figure 2.1(b) shows a basic silhouette line around the dancer. In Figure 2.1(c), silhouette lines are drawn only around individual body parts. A detailed drawing of the woman depicted in the video is shown in Figure 2.1(d). Figure 2.1(e) shows four copies of the same human silhouette annotation and in Figure 2.1(f) multiple silhouettes are drawn using a brush-like texture. The video frame in Figure 2.1(g) is illustrated with white chalk-like dashes and Figure 2.1(h) uses various graphical elements to illustrate multiple enlarged paths of the dancer’s silhouette. 9

2.2. An example video frame containing a person (2.2(a)) and a human segmentation matte computed on this data (2.2(b)). 11

3.1. Illustration of the conceptual flow on how to create a video enriched by motion visualization effects. For each individual image in a video in which one or multiple human beings are depicted, 1. the outline of each person is traced, 2. the traced outlines get stylized and 3. the stylized outlines are blended onto the original video frame. This, together with the other sequentially processed frames, forms the stylized output video. 16

3.2. Sample illustration of the **Halo Effect** from the music video “Wait for You” by Tom Walker . It is distinguished by several graphic elements based on the dancer’s silhouette at different scaling levels, which continuously follow the dance movements. 18

3.3. Sample illustration of the **Silhouette Effect** from the music video “Wait for You” by Tom Walker . It is distinguished by only one graphic element outlining the dancer’s silhouette, which continuously follows the dance movements. 18

3.4. Sample illustration of the **Twins Effect** from the music video "Wait for You" by Tom Walker . It is distinguished by multiple graphic elements outlining the dancer's silhouette, which are horizontally shifted and continuously follow the dance movements. The silhouettes shown next to the dancer give the impression of stylized copies of himself. 19

3.5. Sample illustration of the **Freeze Effect** from the music video "Wait for You" by Tom Walker . It is distinguished by a single graphical element, which outlines the dancer's silhouette only at the beginning of the video effect and is then not changed while the video of the dance performance continues. 19

3.6. A video effect is temporally divided into **Intro Stage**, **Main Stage** and **Outro Stage**. Keyframes K_0 to K_3 are used to define the start and end of an effect and the transitions between effect stages. The total duration of an effect is the time difference between K_0 and K_3. Animations of video effects can be created by interpolating visual variables between keyframes. 20

3.7. The illustrated processing graph shows a selection of processing nodes used to generate Motion Visualization Effects. Processing nodes can be connected using typed input and output ports with the same data type to create complex data processing flows for creating video effects. 24

3.8. Processing nodes that affect the visual appearance of video effects can be extended with input ports to adjust visual variables while processing. Visual variables can either be adjusted by user input or in an automated manner, e.g. for the creation of effect animations. 24

3.9. The reader node is used as a root node of a processing graph and reads multidimensional data from a multidimensional video file or multidimensional camera feed and sequentially provides the read multidimensional data to subsequent nodes in a synchronized manner. Typical data types in multidimensional videos include, among others, RGB data, human segmentation data, and Lidar depth data. 25

3.10. The depicted processing graph consists of a configuration of processing nodes for creating **Silhouette Effects**. Video frames from an RGB video are read by a *RGB Frame Reader Node*. The *Human Segmentation Node* masks people in the video frame, this mask information is used by an *Contour Detection Node* to generate silhouette outlines. The resulting bezier paths are styled by a *Path Stylization Node*. The stylized bezier path and the original RGB video frame are blended into a single video frame using the *Path Blending Node*, which finally gets written to a video file using a *RGB Frame Writer Node*. 26

3.11. By reading precomputed human segmentation matte data together with RGB data from a multidimensional video file, the complexity of the processing graph in Figure 3.10 can be reduced and the processing performance increased. 27

3.12. The *Path Duplication Node* scales and duplicates stylized bezier paths. The number of duplicates and a fixed value of scaling between duplicates can be set via the nodes' input ports. 28

3.13. The structure of the mobile app is divided into several app views, which provide the user with the necessary functionality for video creation. The user is able to navigate between app views by buttons, indicated by the shown lines between views. The Main View is the entry point of the app. The user is able to capture multidimensional videos using the Capture View. Captured videos can be enhanced by silhouette-based video effects within the Edit View. Applied effects are fine-tuned using the Parameter View. After editing, the resulting video is created and saved or shared via the Export View. Additionally, the user is able to save the current state of editing as a project to continue editing at a later point in time using the Projects View. 29

3.14. Timeline sliders are a typical interaction element on touch-enabled devices for navigating and manipulating videos. The screenshots show well-known app examples for the Apple iOS operating system. 33

3.15. From the apps Edit View the user reviews the selected video footage and creates, adjusts, and deletes silhouette-based video effects. By tapping an effect *Preset* button, the effect can be previewed while the effect gets placed in the timeline. The user adjusts the position and length of an effect by well-known tap-and-drag gestures within the timeline slider. Tapping an effect marker within the timeline allows to customize the effect representation via the Parameter View as well as to delete an effect. . . . 35

3.16. The app's Parameter View allows the user to customize effect appearance. Therefore, four generic parameter interfaces are provided: Switch (Figure 3.16(a)), Slider (Figure 3.16(b)), Stepper (Figure 3.16(c)) and Color Picker (Figure 3.16(d)). 37

4.1. The component diagram shows the five main components of the app: The *user interface* component implements the graphical representation of the app and allows the user to interact with the app functionalities. The *pipeline* component implements the multidimensional video processing framework and, thus, represents the core component of the app. The *managers* component contains a set of state managers that act as interfaces between *user interface*, *pipeline* and *IO* component. The latter provides data access to sensors, local memory, and the media library of the device. Additionally, custom-developed processing routines are bundled in the *Processing* component. 40

4.2. Overview of the classes used for providing interactive parameter adjustment. The property wrappers `SwitchValue`, `SliderValue`, `PickerValue` and `ColorValue`, which hold the current parameter value and provide a SwiftUI view for the user interface, inherit from the generic `Viewable Parameter` class. The `Viewable` protocol ensures that SwiftUI views of type `AnyIterableView` can be created for `ViewableParameter` instances. The `Parameter` parent class implements the reactive data storage mechanism of the `value` property, which allows propagating value changes from the user interface up to the node parameters. While the `ViewableParameter` subclasses annotate parameter properties on effect-level, an additional `views` array holds the user interface views, provided by the `buildView()` method, to be displayed to the user. 56

5.1. The figure shows screenshots of the developed MotionViz app, including the start screen providing navigation options to the functionalities of the app, the capture screen while recording a multidimensional video, the trim screen for trimming a captured video to the desired length as well as the loading screen for loading captured multidimensional video files. 60

5.2. The figure provides screenshots of the MotionViz apps edit screen. Figure 5.2(a) shows the edit screen with its four effect buttons and no applied effect. At Figure 5.2(b), a Silhouette Effect gets applied by tap-and-holding the Silhouette Effect button. Figure 5.2(c) shows, that multiple effects can be applied in different sections of the video. In Figure 5.2(d), by tapping an effect the tooltip is shown for adjusting effect parameters and for deleting the effect. 61

5.3. The figure shows screenshots of the parameter screen of an applied Silhouette Effect. Parameters are listed for the main stage. In Figure 5.3(a), the *Twins Right* parameter is selected, showing a picker interface for parameter adjustment. *Twin Distance* can be adjusted by the use of a slider interface in Figure 5.3(b). The *Line Color* parameter was adjusted to blue in Figure 5.3(c) using a color interface. And in Figure 5.3(d) the *Chalky* mode is enabled by the use of a toggle interface. 63

5.4. The figure shows screenshots of the export screen. In Figure 5.4(a), the progress of the current video export is shown together with the current rendered video frame. Figure 5.4(b) shows the rendered video together with the share button for saving and sharing the video. Also, by clicking the *Done* button the project save modal opened up to save the effects and parameter configurations. 64

5.5. This screenshot of the MotionViz app depicts the loading screen for loading saved projects into the edit screen. 64

5.6. The figure compares visualizations of a Halo Effect from the music video “Wait for You” by Tom Walker (Figure 5.6(a)) and the produced effect by the MotionViz app (Figure 5.6(b)). 66

5.7. The figure compares visualizations of a Silhouette Effect from the music video “Wait for You” by Tom Walker (Figure 5.7(a)) and the produced effect by the MotionViz app (Figure 5.7(b)). 67
5.8. The figure compares visualizations of a Twins Effect from the music video “Wait for You” by Tom Walker (Figure 5.8(a)) and the produced effect by the MotionViz app (Figure 5.8(b)). 68
5.9. The figure compares visualizations of a Freeze Effect from the music video “Wait for You” by Tom Walker (Figure 5.9(a)) and the produced effect by the MotionViz app (Figure 5.9(b)). 68
5.10. Screenshot of the Xcode Memory Report screen. To measure memory usage, several effects were created, previewed, their parameters adjusted, and resulting videos exported. The typical memory consumption of the MotionViz app is between about 140 and 200 MB, with memory peaks of up to 258.2 MB measured. The measurement is based on a recorded 30-second multidimensional video with RGB, segmentation, depth, and audio track (1440 by 1080 pixels, 50 frames per second) and was performed on an iPad Pro (11-inch, 2nd generation with A12Z GPU and 6 GB shared memory), running iPad OS 15.4.1. 72
5.11. Shown is an excerpt from the interim presentation by Rosmarie Debski and Ole Schmitt in the seminar *Algorithms for Processing Visual Media (APVM)* at the Hasso Plattner Institute, in which the students extended the MotionViz app by a *GlowStick Effect*. 73
5.12. The shown video frame from a video created with the MotionViz app shows segmentation artifacts for the applied Silhouette Effect. These come from errors in the person segmentation data for this video frame. First, the dancer’s arms were not segmented correctly. In addition, a bush in the background was incorrectly segmented. Due to the segmentation artifacts, the displayed contour lines do not correctly trace the dancer, and in addition, a silhouette is drawn around the bush in the background. 74
5.13. The displayed video frame from the music video “Wait for You” by Tom Walker shows a motion visualization that, in addition to the dancer’s contour line, distinguishes other details by means of a contour line. The dancer’s right forearm is traced, and further, another contour line distinguishes between the left arm and the left foot in the background. 75

List of Tables

1.1. Definition of functional requirements for a video effects app, which enables the automatic generation of human silhouette-based motion visualizations. 3
1.2. Definition of non-functional requirements for a video effects app, which enables the automatic generation of human silhouette-based motion visualizations. 4

2.1. Classification of different approaches to the representation of motion in static images with respect to their efficacy after Cutting [15]. 7

3.1. List of possible visual variables for the silhouette-based motion visualization effects **Halo**, **Silhouette**, **Twins**, and **Freeze**. Visual variables are used by the effect designer to define the appearance of video effects and can be immutable, animatable, or user-adjustable. 17

5.1. The table shows performance measurement results for the rendering of the video preview and video export for the four effect types Halo, Silhouette, Twins, and Freeze. For the video preview, the average adaptive frame rate during video playback was measured in frames per second (fps). The performance of the export rendering was determined based on the total duration of the rendering process and divided by the total number of video frames to get the fps metric. Measurements are based on a 30-second multidimensional video with RGB, segmentation, depth, and audio track with a resolution of 1440 by 1080 pixels per video track at 50.6 fps (Original Res.). For comparison, additional measurements were performed on the same video with a reduced resolution of 1280 by 960 pixels (Low Res.). Video effects are applied to the entire video length in each case and additional measurements without applied effects were taken for reference. All measurements were performed on an iPad Pro (11-inch, 2nd generation with A12Z GPU and 6 GB shared memory), running iPad OS 15.4.1. . . 70

Listings

4.1. Simplified except of the `Frame` class from `Frame.swift`. It wraps processing data by assigning it to its `value` property and can be adapted to a new `Frame` instance of differently typed `value` property while passing its meta data, the `timestamp` property, via the `adapt<NewType>(for newValue: NewType)`. 42

4.2. The `SingleInputNode` class is one of four generic base classes for processing node implementations. It receives a single value publisher compatible with the `Input` type. Within the constructor, `setupPipeline(input:)` is called, which subscribes to values published by the publisher. For each received value, the data is passed to the `process(input:)` method. Subclasses overwrite `process(input:)` to implement the actual data processing. . . . 44

4.3. By using type aliases, the generic node classes, `SingleInputNode`, `DualInputNode`, `TripleInputNode` and `QuadInputNode`, are specified for processing data wrapped in a `Frame` instance. A processing node that inherits from these nodes therefore only needs to specify the generic type `Input`, which determines the data type of the data contained in a `Frame` instance. 44

4.4. Shown is a simplified except of the `ContourNode` class from `ContourNode.swift`. Due to the functionality inherited from the `SingleFrameNode`, the implementation mainly consists of the `process(input:)` method for extracting an `UIBezierPath` from an `Rgb` frame. The `@Output` property wrapper provides the created `UIBezierPath` data via a publisher to subsequent nodes. 45

4.5. Simplified except of the `Output` property wrapper from `Node.swift`. It provides a convenient and clean way to create publishers for distributing data via the Combine Framework. 46

4.6. Simplified except of the `OutlineEffect` class from `OutlineEffect.swift`. It combines multiple processing nodes for creating a silhouette-based video effect from multidimensional data input. 48

4.7. By the definition of type aliases, the allowed data types for the input and output publishers of an `TimedEffect<Input: TimedData, Output: TimedData>` class get specified. 49

4.8. Simplified except of the `Pipeline` class from `Pipeline.swift`. It receives incoming data via the `processAsync(input:fallback:)` method and routes it to the first effect whose `timeRange` property matches the `timestamp` of incoming data. 50
4.9. Excerpt from the `Effect` base class in `Effect.swift` showing the *manualDownstream* mechanism. By a combination of `PassthroughSubject` and the `manualDownstream` computed property, single data frames can be pushed to an `Effect` via its `send(_:)` method. 51
4.10. Simplified excerpt of the `BlurNode` from `BlurNode.swift`. It applies a gaussian blur via its `process(input:)` method. The resulting blurred frames can be subscribed via the annotated `blurredFrame` property. The blur intensity is controlled by the `sigma` property, which either provide the assigned static value of `10` or an externally controlled value, assigned via a binding at the `NodeParameter` property wrapper. 52
4.11. Simplified excerpt of the `NodeParameter` property wrapper from `Node.swift`. It is used to annotate parameter properties at the node level and enable the change of parameter values via the user interface. If a binding is assigned at the `projectedValue` property, the data provided via the binding is used. Otherwise the `defaultValue`, which was assigned to the annotated property at initiation, is returned as a `wrappedValue`. 53
4.12. Simplified exerpt of the `BlurEffect` from `BlurEffect.swift`. It defines `name`, `color` to be displayed in the user interface. Further, an `allowed Durations` defines the minimum and maximum duration of the effect. The effect consists of a single `BlurNode`, which blurs an RGB video frame. The blur intensity is defined by the property `sigma`. It is annotated with the `SliderValue` property wrapper, which automatically displays a slider interface within the Parameter View of the effect. The `sigma` parameter is identified with the set `name` property on the property wrapper and changes via the user interface are restricted to a value range between `0` and `20`. It further defines the effect stage to which the parameter is applied. 55
4.13. Simplified excerpt of the `SwitchValue` property wrapper from `Switch.swift`. It inherits from the generic `ViewableParameter` class and overwrites the `buildView()` method, which constructs an `AnyIterableView` from the included `structs IconView` and `ControlView`, which define SwiftUI view layouts. The SwiftUI views contain the `SwitchValue` instance itself as `@ObservedObject` for binding its boolean value property to the user interface reactively. Via the `projectedValue` property of `SwitchValue`, a binding for the `NodeParameter` property wrapper is provided for binding node parameters to the `value` property of `SwitchValue`. 58

Abstract

Mobile short-form videos represent one of the most influential media types today and are consumed and shared daily by millions of active users on social media platforms. The platforms themselves and third-party providers offer a variety of video effects that users and content creators use to edit and enhance their videos. These effects range from preconfigured color grading filters to content-aware video effects, which provide stylizations that incorporate the video content. To date, however, video effects mostly either are "one-click" options and don't offer customizations at all or require domain knowledge for the use of complex effects settings.

This thesis presents a software framework and a mobile app that allows users to create and interact with content-aware video effects in an automated manner. The proposed framework uses a graph-based processing approach that enables the processing of multiple video streams as well as data from modern hardware features such as Lidar sensors and neural processing engines. Through a flexible processing approach, video data and intrinsic semantic information, such as human segmentation and pose, are processed together into content-aware video effects.

The capabilities of the framework are showcased in a mobile app, which enables the creation of motion-focused effects in an automated manner and thereby eliminates the time-consuming manual generation of required contextual information and graphics. An intuitive timeline-based interface allows the accurate art direction of manifold silhouette-based effects, including different styles and complex animations. These can be interactively applied and animated to express dynamics with ease, e.g., of dancers, athletes, and other performers.

The processing framework, as well as the app for motion visualization, is evaluated quantitatively and qualitatively to underline its interactive performance as well as general applicability. The capabilities of the processing framework together with the user experience of the mobile app make complex customizable motion effects accessible to a wide audience and lead to time savings for professionals.

Finally, an outlook on emerging opportunities is provided, for further development of the presented framework and app and new possibilities that arise as a result.

Kurzfassung

Kurzvideos gehören heute zu den einflussreichsten Medientypen und werden täglich von Millionen aktiver Nutzer*innen auf Social-Media-Plattformen konsumiert und geteilt. Die Plattformen selbst und Drittanbieter*innen bieten eine Vielzahl von Videoeffekten an, mit denen Nutzer*innen und Inhaltsersteller*innen ihre Videos bearbeiten und verbessern können. Diese Effekte reichen von vorkonfigurierten Filtern für Farbanpassungen bis hin zu inhaltsbezogenen Videoeffekten, die den Videoinhalt stilisieren. Bislang handelt es sich bei den meisten Videoeffekten jedoch entweder um „Ein-Klick"-Optionen, die keinerlei Anpassungsmöglichkeiten bieten, oder sie erfordern Fachkenntnisse zur Bedienung komplexer Effekteinstellungen.

In dieser Arbeit werden ein Software-Framework und eine mobile App vorgestellt, mit denen Nutzer*innen inhaltsbezogene Videoeffekte auf automatisierte Weise erstellen und mit ihnen interagieren können. Das vorgeschlagene Framework verwendet einen Graphenbasierten Verarbeitungsansatz, der die Verarbeitung mehrerer Videoströme sowie die Verarbeitung von Daten moderner Hardware-Ressourcen, wie Lidar-Sensoren und neuronale Prozessierungseinheiten, ermöglicht. Durch einen flexiblen Verarbeitungsansatz werden Videodaten und intrinsische semantische Informationen, wie zum Beispiel menschliche Segmentierung und Pose, gemeinsam zu inhaltsbezogenen Videoeffekten verarbeitet.

Die Fähigkeiten des Frameworks werden in einer mobilen App vorgestellt, die die automatisierte Erstellung von bewegungsorientierten Effekten ermöglicht und damit die zeitaufwändige manuelle Generierung von erforderlichen Kontextinformationen und Grafiken überflüssig macht. Ein intuitives, zeitleistenbasiertes Interface ermöglicht die präzise Steuerung vielfältiger Silhouetten-basierter Effekte, einschließlich verschiedener Stile und komplexer Animationen. Diese können interaktiv angewandt und animiert werden, um die Dynamik von Tänzer*innen, Sportler*innen und anderer Darsteller*innen herauszustellen.

Sowohl das Prozessierungsframework als auch die App zur Bewegungsvisualisierung werden quantitativ und qualitativ evaluiert, um ihre interaktive Leistungsfähigkeit sowie die allgemeine Anwendbarkeit zu unterstreichen. Die Fähigkeiten des Frameworks zusammen mit der Nutzerfreundlichkeit der mobilen App machen komplexe anpassbare Bewegungseffekte einem breiten Publikum zugänglich und führen zu Zeitersparnis für professionelle Motion Artists.

Abschließend wird ein Ausblick auf sich abzeichnende Chancen für die Weiterentwicklung des vorgestellten Frameworks und der App gegeben, sowie auf neue Möglichkeiten, die sich aus dieser Arbeit ergeben.

Chapter 1

Introduction

With the rising photo and video capturing capabilities of mobile devices, creating high-quality photos and videos gets accessible to nearly anyone. At the same time, user-generated content is very popular on social media platforms. Applying photo and video filters to self-captured media enjoys high popularity on social media platforms such as TikTok, Snapchat, and Instagram.

People who share photos on social networks often want to give them an individual look. A wide range of photo and video filters and effects are available for this purpose. Most camera and photo library apps bring filters for adjusting aspects like contrast, brilliance, or brightness. More advanced filters can be found in social media apps, which change the overall color appearance of a photo or video. Apps like Clip2Comic [11] allow the user to convert videos to a comic style. BeCasso [32] creates realistic oil or sketch paintings.

Also popular on social media platforms are animated video effects applied to user-generated content. Some of them build on a contextual understanding of video content. For example, the *Dubbing* or *Mouth Sync filter* transfers a person's mouth movements to other people or animals in the video. Also, Animojis are animated 3D rendered faces or heads that are superimposed over a person's head in a video. Thereby, the Animoji mimics the facial expressions and head movements of the person in the video.

All these applications are made possible by steadily increasing processing capabilities, including on mobile devices. As a result, operators of social media platforms and providers of stylization and filter apps can use increasingly sophisticated photo and video processing methods, like machine learning algorithms and 3D renderings, in combination with high-quality image and video recordings and enable users to customize their content in new ways.

In the context of video filters and effects, and with a view to current social media trends, this paper presents a processing concept and, based on it, a concrete effects app for mobile devices that goes beyond the usual processing of recorded videos.

1.1. Motivation

One of the most popular types of videos on social media channels, especially on TikTok, are dance videos [43]. Influencers like Charli D'Amelio, for example, use the hashtag *#distancedance* to call for dance challenges with musical accompaniment and receives

Figure 1.1.: *Video frame from the music video for the song "Wait for you" by Tom Walker with in a video editing program manually drawn motion lines.*

thousands of video reactions [28]. These video reactions are usually recorded and edited on a smartphone. Thanks to the algorithms of modern social media platforms, this type of non-professional, but user-generated content can reach and entertain thousands.

The popularity of this user-generated content format has already extended beyond social media platforms. Tom Walker's music video for the song "Wait for You" [48], released in June 2020, taps into the existing popularity. "Similar to the song itself, which was recorded under [COVID-19] lockdown, the video was also shot under lockdown and finds 34 dancers from 13 different countries performing a series of self-choreographed routines." [45] The style of the self-shot videos is combined with professionally created motion effects with a sketchy, hand-drawn look. The types of drawn motion effects used are diverse: From motion lines, as can be seen in Figure 1.1, to diverse variations of silhouettes to abstract representations of people. This involves working with solid, dotted, or even patterned lines of different line widths. Partly solid colors, partly a chalk-like line style is used. The video achieves over 2.7 million views on the YouTube video platform[1].

To date, adding such motion effects to dance videos with a hand-drawn look requires a laborious manual process. In professional video editing software such as Adobe After Effects, video frame by video frame must be drawn onto the captured video by hand, just as in stop motion animations. This manual process, therefore, requires both knowledges of professional video editing software and a lot of time to implement. It is therefore unsuitable for user-generated content, which is typically edited by non-professionals directly on mobile devices. To open up this style of video effects to the user-generated content domain, we present MotionViz: A video effects app that automatically creates artistic hand-drawn looking, silhouette-based annotations for human motion in videos.

Being able to generate motion effects automatically opens up numerous technical

[1] Accessed 2022-07-19.

ID	Requirements	Description
F-01	Capturing	• Capturing of multidimensional videos • Captured RGB video streams have 30 fps or more
F-02	Saving and Loading	• Multidimensional videos can be persistently stored and loaded
F-03	Processing	• Silhouette-based video effect can be created
F-04	User Interface	• App provides User Interface to create, edit and delete effects • App provides live preview while video editing • Effect parameters can be adjusted from the user interface
F-05	Projects	• App supports saving work states during effect creation • Work states can be restored and edited at a later time

Table 1.1.: *Definition of functional requirements for a video effects app, which enables the automatic generation of human silhouette-based motion visualizations.*

challenges: Humans must be computationally segmented from the RGB video, leading to another image stream beside the generic RGB image stream. As a consequence, the app is required to generate additional information and has to process multiple image data streams at once. We also refer to these multiple video streams as multidimensional video.

From these multidimensional data, silhouette-based vector paths have to be generated afterward. Vector paths can then be offset to different line styles or assets can be placed on the path to create dotted or other patterns. Finally, the created lines must be rendered upon the original video frames.

1.2. Requirements

In the following, the non-functional as well as functional requirements are listed, which are required by a mobile app for creating motion visualization effects.

1.2.1. Functional Requirements

As described in Table 1.1, the system to be developed must meet functional requirements regarding the recording of videos (F-01), the storage and loading of these videos (F-02), the processing (F-03), and the user interface (F-04).

For video recording, this means that the app has to record video data from a camera embedded in the mobile device, along with other sensor data such as depth information. To achieve a smooth multidimensional video, at least 30 frames per second for the RGB stream are required.

ID	Requirements	Description
NF-01	Usability	• Error-free • Intuitive to use • No need for professional knowledge of video editing • Fun to use
NF-02	Reliability	• Effects can be created, modified, and deleted as desired • Effects can be consistently reproduced
NF-03	Maintainability	• Adaptable for other data and use cases • Extendable with new effects and additional parameters
NF-04	Effect Quality	• Fluent • Appealing • Artistic

Table 1.2.: *Definition of non-functional requirements for a video effects app, which enables the automatic generation of human silhouette-based motion visualizations.*

For recorded multidimensional videos, it must be possible to store them persistently for reading them at a later time. For this purpose, an existing data format is adapted or a new data format can be developed.

The central goal of the app is to allow the user to process multidimensional videos, in which one or more silhouette-based video effects can be applied to a video.

In addition to the technical functionality of recording, storing, loading, and processing videos, a key requirement is the ability of the user to interact with the app.

For this, the app is supposed to allow users to create, edit and remove video effects on a video. The user should thereby be presented with a live preview of the created video effects.

1.2.2. Non-Functional Requirements

Table 1.2 defines non-functional requirements regarding usability (NF-01), reliability (NF-02), maintainability (NF-03) and the quality of video effects (NF-04).

In terms of usability, the app to be created has to be error-free. In addition, creating videos, applying video effects, and exporting the resulting video should be easy to learn. Video editing must be designed in such a way that professional video editing knowledge is not necessary. This ensures that the app is also applicable for user-generated content. Since the mobile app is aimed at social media and leisure activities, the fun of using the app is also an important requirement.

The fact that effects can be created, modified, and deleted as well as consistently reproduced ensures the reliability of the application.

To be able to respond to new requirements in the future, the application is to be

designed in such a way that it can be easily adapted to new data and usage scenarios. Also, new effects and configuration parameters must be easily customizable.

To ensure the visual quality of the video effects, they have to fit smoothly into the edited video, are visually appealing, and have a certain artistic expression.

Chapter 2

State of the Art

James E. Cutting states that "Representing motion in a picture is a challenge to artists, scientists, and all other imagemakers [sic!]. Moreover, it presents a problem that will not go away with electronic and digital media, because often the pedagogical purpose of the representation of motion is more important than the motion itself."[15]

Cutting identifies four visualization modes to evoke motion in static images:

1. Dynamic balance.
2. Usage of multiple stroboscopic images.
3. Affine shear or forward lean.
4. Photographic blur.
5. Usage of image and action lines.

He further points out that each of the visualization modes is a compromise in the accuracy of the motion represented. Table 2.1 shows Cuttings classification of the efficiency of the representation based on these four criteria *evocativeness*, *clarity of object*, *direction of motion* and *precision of motion*.

While Cutting's focus lies on an artistic transfer of human perception of motion into a still image, Semmo et al. in their paper "ViVid: Depicting Dynamics in Stylized Live

Distinct solutions	Evocativeness	Clarity of object	Direction of motion	Precision of motion
Dynamic balance	✓	✓	(✓)	✗
Multiple stroboscopic images	✓	✓	✗	✓
Affine shear / forward lean	✓	✓	✓	✗
Photographic blur	✓	✗	✗	✗
Image and action lines	✓	✓	✓	✓

Table 2.1.: *Classification of different approaches to the representation of motion in static images with respect to their efficacy after Cutting [15].*

Photos" [39] present an automated approach to compose motion in videos into static stylized images. Therefore, Apple Live Photos are used as a data basis, which consists of a high-resolution photo as well as a short video sequence simultaneously captured with the photo. Motion analysis of the video is performed by a convolutional neural network. The original image is enhanced by a stylization filter and superimposed by motion visualizations of the motion analysis. Three different motion representations are presented: Two of them are using action lines. The third type of representation is named *ghosting* and fits into Cutting's definition of stroboscopic images. The composition of stylized images and motion visualization results in appealing and effective static graphics that illustrate the movements of the recorded scene.

In contrast to pictures, where motion is worth explaining due to the static presentation, videos often already give a good impression of motion due to the presentation of many successive frames of a scene. In addition, the phenomenon of motion blur, which is the corresponding counterpart to photographic blur for videos, supports the perception of motion in videos but also occurs with the same drawback as described by Cutting, as it "typically sacrifice[s] the accuracy of the motion represented" [15]. Namely in terms of clarity of objects and precision of motion, since the direction of motion is apparent from the sequence of images in the video. We, therefore, consider in the following the visualization of movement in videos as a special emphasis or as a stylistic device of exaggeration with the aim of visually enhancing the presentation. And with regard to user-generated content on social platforms, we especially focus on human movements in videos.

2.1. Human Motion Visualization

As introduced in section 1.1, the music video "Wait for You" by Tom Walker offers multiple appealing animated graphical annotations, which we classify as motion visualizations in the course of this work. These video effects in the following will be considered prototypically for human motion visualizations.

Comparing the different shapes of graphical annotations used within the music video, two main groups can be identified: Action lines, as can be seen within Figure 2.1(a), and path annotations based on the human silhouette. For the silhouette-based annotations, three subtypes can be specified: The basic silhouette, as shown in Figure 2.1(b). Silhouettes, which only cover individual body parts, as shown in Figure 2.1(c). And human drawings, which, in comparison to the first two subtypes, contain a higher level of detail in representing a person (cf. Figure 2.1(c)).

Regarding the path style of the graphical annotations, it can be noted that all of them have an opaque, hand-drawn look. No straight paths or perfect curves are used. In more detail, four visual variables can be identified with respect to path style:

(a) *Chalky action lines.*

(b) *Solid adjacent silhouette.*

(c) *Dotted illustration of body parts.*

(d) *Human drawing.*

(e) *Multiplied silhouettes.*

(f) *Brushstroke silhouette.*

(g) *Dotted static silhouette.*

(h) *Exploding silhouette of various styles.*

Figure 2.1.: *Video excerpts from the music video of the song "Wait for You" by Tom Walker, illustrating different possibilities of human motion visualization. In Figure 2.1(a), motion in the video is annotated by action lines. Figure 2.1(b) shows a basic silhouette line around the dancer. In Figure 2.1(c), silhouette lines are drawn only around individual body parts. A detailed drawing of the woman depicted in the video is shown in Figure 2.1(d). Figure 2.1(e) shows four copies of the same human silhouette annotation and in Figure 2.1(f) multiple silhouettes are drawn using a brush-like texture. The video frame in Figure 2.1(g) is illustrated with white chalk-like dashes and Figure 2.1(h) uses various graphical elements to illustrate multiple enlarged paths of the dancer's silhouette.*

1. Graphical Elements: Lines, dashes, dots, circles, crosses
2. Color: White, black
3. Texture: Solid, chalk-like, brush
4. Transparency: Opaque

Figure 2.1(g) shows a path illustrated with white chalk-like dashes, while for example Figure 2.1(b) has a solid line style and Figure 2.1(f) uses a brush-like texture. While some annotations are based on a single illustrated path, the position and number of human silhouette-based paths are varied in other annotations. Thus, Figure 2.1(h) uses six paths based on the dancer's silhouette, which is scaled and smoothed to varying degrees. The path styles are illustrated using circles, crosses, and tangentially arranged dashes as graphical elements. In Figure 2.1(d), the solid lined human drawing is enlarged and offset to the left. Figure 2.1(e) shows four copies of the same annotation, which were shifted horizontally.

Finally, a look at the animation styles of the graphical annotations in the music video is taken, as they contribute significantly to the visual appearance of the video. All of the drawn annotations have in common that they are not animated with equal fluidity across video frames as the base video itself. The animations have rather a jerky character, as known from stop motion movies. This is probably largely due to the fact that manual annotation is very time-consuming, especially when applied to every frame. At the same time, however, this is certainly a deliberately used stylistic device, whereby the graphical visualizations should stand out in their animation from the source video.

Looking at the animations, three different animation types can be identified: First, the annotation is static over several frames. The animation shown in Figure 2.1(g) is an example of this variation. Second the annotation follows the movements of the dancer (e.g. Figure 2.1(a), Figure 2.1(b), and Figure 2.1(d)). These two types are partly combined with an input or output animation, or both, in that the annotation is built up or broken down over several frames. For example, by varying stroke thickness or dot patterning of the line. The last type consists of a throughout changing animation, with no clear input or output animation. Examples of this are the video sections referenced in Figure 2.1(f) and Figure 2.1(h).

It can be concluded that human motion visualizations within the music video are created based on four identified visual variables. It is clear to see that, except from Figure 2.1(a) and Figure 2.1(d), all paths are taken from the silhouette of dancers or parts of their bodies. The paths then have been used as is or been enlarged, moved, or duplicated. Regarding the referenced animations, the paths are created based on the person's silhouette from the same or previous frames.

Based on this findings, we define the term *Motion Visualization*, or short *MotionViz*, as animated human silhouette-based annotations in videos. The human silhouette-based path design follows a finite defined choice of visual variables. Further visual variation is

(a) *Original video frame.* (b) *Generated human segmentation matte.*

Figure 2.2.: *An example video frame containing a person (2.2(a)) and a human segmentation matte computed on this data (2.2(b)).*

created by multiplying and translating the paths, and by changing the path design and position across multiple frames of the video. The overall goal of motion visualizations is to emphasize and highlight the movements of people depicted in a video in an artistic and entertaining way.

2.2. Multidimensional Video Data

Tracing and abstracting human silhouettes from video frames, as shown in the previous chapter, is an artistic, time-consuming but not very demanding activity for humans. It is therefore an obvious choice to automate this manual process. Creating such an interpretation of image content in an automated way requires advanced processing algorithms and methods that compute contextual information from the RGB color information of video frames.

The task of identifying which pixel of an image belongs to a representation of a person in the image is called human segmentation. With the help of convolutional neural networks, a so-called human segmentation matte is calculated based on an RGB input image. The segmentation matte indicates for each pixel of the output image whether it belongs to a person or not. Figure 2.2 shows an example of a video frame and a human segmentation matte calculated from it. Well-known implementations include MODNet by Ke [23] and the proprietary segmentation network, which is provided by the Apple Vision Framework on the iOS and iPad OS operating system since version 15 [47].

After the process of human segmentation, silhouette-based paths can be calculated based on segmentation mattes per individual frame of the video.

The computational complexity of human segmentation methods is high, so the generation of highly detailed segmentation mattes generally cannot be achieved in real-time. And in order to be able to generate motion visualizations, both original RGB video data and segmentation information must be processed simultaneously. To be able to process both data streams simultaneously while allowing for real-time processing speeds, the contextual image information can be precomputed and stored together with the input video. As a result, once the segmentation data has been generated, the two data

streams can be efficiently processed simultaneously. When video data and contextual data are stored together in one file, we refer to it as *Multidimensional Video Data* or *Multidimensional Video*.

Multidimensionality is already used in the literature in various contexts with video data: In *Multidimensional Processing of Video Signals* [40], Sicuranza et al. describe multidimensionality in the processing of video signals in the context of color processing. By defining a color value as a three-dimensional vector in a given color space, versus a one-dimensional brightness value, according to their definition, a color video is already considered to be a multidimensional video.

Woods in *Multidimensional Signal, Image, and Video Processing and Coding* [50] considers the processing of spatiotemporal data to be three-dimensional processing. Thereby any type of video processing that takes the temporal dimension of data into account could be considered to be the processing of the multidimensional video.

However, in the context of this work, we use the term *Multidimensional Video Data* to refer to a set of spatiotemporal data which goes beyond a single video and transports not only an RGB video stream but also other time-dependent contextual information. Thus, in a multidimensional video, for each video frame of the RGB video, there may also be an associated segmentation matte containing the position of people in the respective video frame. But the definition of multidimensional video may not be limited to segmentation information.

Modern mobile devices are increasingly equipped not only with single cameras but with multi-camera arrays and additional sensors. Typical examples are camera arrays with different focal lengths, allowing the user to choose from multiple fields of view. In addition, optical sensor arrays enable stereoscopy procedures, with which depth information can be calculated [21]. Furthermore, the manufacturer Apple, for example, equips top-of-the-line devices with a so-called *Lidar sensor* to bring modern depth-sensing technology to mobile devices [2]. And with machine learning algorithms, such as *Pose Estimation* [38] or *Optical Flow* [51], further contextual information can be extracted from RGB data.

All these different types of data can be considered as extending data dimensions in terms of multidimensional videos, which together may open up new video processing and stylization possibilities.

2.3. Multidimensional Video Processing

Multidimensional Video Processing extends the sequential processing of RGB video frames, by including additional data dimensions from multidimensional video data, such as scene depth or segmentation data. Multidimensional video processing can be conceptually compared to parallel processing of multiple video streams, as it is common in video editing. Unlike video processing of multiple video streams, e.g. by applying crossfades or chroma keying, the different streams from multidimensional data are typically not in the same file format or temporal resolution.

However, the core of multidimensional video processing is the processing of multiple dimensions of spatiotemporal and other time-dependent data from different sources, which inevitably leads to differences in format and resolution. For reading the data, for each data format, a special decoding routine is required. Pre-processing of data for homogenization before the actual processing step may also be necessary. Afterward, as with multiple video tracks, this data can be synchronized and processed in parallel based on temporal resolution or using time stamps.

Therefore, a flexible processing framework is required to prepare and unify the different data representations and to process the individual data streams into one output format. In the context of this work, the requirements and limitations of such a framework on the hardware have in particular to be considered. Therefore, an Apple iPad Pro (11-inch, 2nd generation with A12Z GPU and 6 GB shared memory) is defined as reference hardware.

In 2009, Hawe et al. introduce *MutanT*, which stands for *Multi-Sensor Data Processing Tool*. "Due to its modular design concept the framework is not limited to specific tasks like image- or speech-processing but can be used for any data processing application." The framework has a graphical user interface (short: GUI) and consists of so-called filter modules. These modules contain encapsulated functionalities and can be linked to each other via the GUI without knowledge about the concrete implementation. Each module has $n \geq 0$ input and $m \geq 0$ output ports, via which data can be exchanged with connected modules. The framework uses the C++ programming language, was released as open source, and, according to the authors, can be easily extended with additional filter modules. Due to the graphical user interface, where parameters can be changed by the user even during runtime, it is said to be particularly well suited for rapid prototyping. The authors showcase their framework with the processing of a video stream from a video camera, an infrared camera, and a 1D Lidar scanner into a pseudo stereoscopic image. [19]

Due to its modular approach with encapsulated functionalities, the framework presented by Hawe et al. offers a flexible system for sequentially processing data of different data formats. However, the work does not make any concrete statements concerning performance and the handling of hardware resources. The provision of a GUI for the creation and linking of modules and the possibility of interactively adjusting variables makes it a good tool for explorative prototyping. Nevertheless, it requires a high level of expertise to combine the individual modules into a pipeline that processes the input data into the desired output format.

The implementation of the framework in the C++ programming language potentially enables portability to different target systems. However, with regard to the above-specified reference hardware, it is more purposeful to rely on the natively provided languages, such as the Swift programming language for Apple iOS operating system. Regarding the particular challenges of limited hardware resources and efficiency in battery-powered systems, using the native languages and frameworks enables the best possible optimizations and access to specialized resources, like unified memory, graphics, and machine learning processors. Further, in the context of this work, the GUI is not relevant

to either the effects designer or the end user. With the focus on user-generated content, a data-driven user interface is intended for this work, which abstracts the technical parameters and makes them configurable for the user. This allows the effect designer to have full control over all technical parameters in the design phase of the development process. For the end user, only individual parameters are then made available to enable effective use that is abstracted from the technical implementation of the data processing.

Since the framework presented above does not meet the requirements for the technical realization of a Motion Visualization App, a novel processing framework for multidimensional video data is presented in the following chapters. For the best possible hardware integration and optimization as well as power efficiency regarding the specified reference hardware of an Apple iPad Pro, the framework relies on Apple iOS native frameworks. This includes, among others, the Combine framework for processing data over time according to the Publisher-Subscriber pattern and the Vision framework for machine learning-based human segmentation. In the following chapter, the technical concept of the Motion Visualization App, as well as the processing framework, will be discussed.

Chapter 3

Concept

The following chapter is devoted to the conceptual challenges of enhancing videos with silhouette-based video effects. Therefore, the basic approach for an automated generation of silhouette lines is explained. Furthermore, based on the music video "Wait for You", four exemplary video effects are specified according to their visual variables and their animation. A multidimensional video processing framework is conceptualized that ensures the generation of video effects while incorporating non-functional and functional requirements. By compiling exemplary processing graphs, the graph-based processing approach underlying the framework is thereby illustrated. Finally, the user experience design of an app is presented, which allows the enhancement of videos with motion visualization effects beyond professional users.

3.1. Silhouette-Based Motion Visualization Effects

This section focuses on the conceptional foundation needed to create silhouette-based motion effects for videos in an automated computational manner. In reference to section 2.1, we base our concept on the graphical annotations of basic silhouettes. First, the procedure for creating human silhouette outlines from video frames is described. Further, four motion visualization effects are presented, which are generated on the basis of such outlines: Halo, Silhouette, Twins, and Freeze. For each effect, the visual variables that determine the visual appearance of the effects are explained. In addition, a concept for animating these effects across multiple video frames is presented.

3.1.1. From Videos to Motion Visualization Effects

For the automated creation of motion visualization effects, we propose a processing concept, which works with multidimensional video on a per-frame basis. Therefore, the multidimensional video contains for each RGB video frame a corresponding human segmentation frame to distinguish image regions with and without persons. Based on the RGB and segmentation data, one or more graphical annotations can be created for each frame of the video. These are finally blended with the original RGB video frame.

In detail, this results in the following workflow for each video frame of a motion visualization effect (cf. Figure 3.1): For each individual image in a video in which one or multiple human beings are depicted,

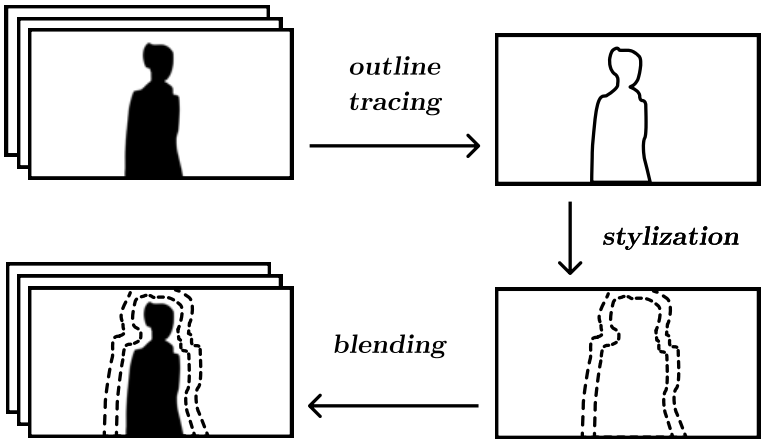


Figure 3.1.: *Illustration of the conceptual flow on how to create a video enriched by motion visualization effects. For each individual image in a video in which one or multiple human beings are depicted, 1. the outline of each person is traced, 2. the traced outlines get stylized and 3. the stylized outlines are blended onto the original video frame. This, together with the other sequentially processed frames, forms the stylized output video.*

1. the silhouette outline of each person is traced,
2. the traced outlines get stylized and
3. the stylized outlines are blended onto the original video frame.

Performing this process for several subsequent video frames creates a video segment with a motion visualization effect.

Since the first and last steps, i.e. tracing of the outline and blending onto the original video frame, are first and foremost technical tasks and do not essentially contribute to the effect design, these steps will not be explored in detail in the following and for the technical realization, we refer to chapter 4. In contrast, the stylization step of silhouette outlines is a central component of effect design, since it determines the visual appearance of the effect. We, therefore, refer to the person who develops the effects as an effects designer. The following section specifies different types of appearances for effects and possible parameterizations, which are available to an effect designer for the stylization process.

3.1.2. Effect Specification and Design

The visual representation of graphical annotations within an effect is based on visual variables, also known as semiotic variables [14]. The form of these variables, such as the color and thickness of a line, determines the visual appearance of the effect. Defining these is therefore the main task of effect design. Variables and their form relevant to

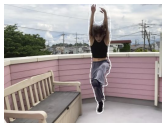
				
Effect name	**Halo**	**Silhouette**	**Twins**	**Freeze**
# Annotations	3	1	3	1
Stroke type	Solid line	Solid line	Solid line	Dashed
Stroke spacing	-	-	-	Medium
Stroke style	Regular	Sketchy	Regular	Chalk-like
Stroke weight	Medium	Medium	Medium	Thick
Outline size	Enlarged (multiple)	Enlarged	Original	Original
Color	Black	White	White	White
Transparency	Opaque	Opaque	Opaque	Opaque
Location	Original	Original	Shifted	Original
Orientation	Tangential	Tangential	Tangential	Tangential

Table 3.1.: *List of possible visual variables for the silhouette-based motion visualization effects* ***Halo***, ***Silhouette***, ***Twins***, *and* ***Freeze****. Visual variables are used by the effect designer to define the appearance of video effects and can be immutable, animatable, or user-adjustable.*

this work are listed in Table 3.1. These were determined based on excerpts from [48] (cf. Figure 3.2 to Figure 3.5) and are explained in the following:

The visual effect shown in Figure 3.2 consists of several silhouette outlines of the depicted person in different sizes. In the video, these graphical annotations follow the movements of the dancer with each video frame, accentuating the dance movements. Regarding the form of the visual variables for this effect, the following can be observed from the selected video frame: The video frame is annotated by three silhouette outlines, which use a *solid line stroke*. The lines are at the *original location* of the persons' silhouette with different levels of *enlargement*. They are drawn as *opaque* lines with a *black stroke color* and a *medium stroke weight*. As the silhouette outline radiates around the depicted person, we refer to this effect as **Halo Effect**.

In comparison to the previous example, in Figure 3.3 only a single *solid line* aligned to the dancer is shown. The line style can be described as *sketchy*, as the line, although solid, looks frayed as if the silhouette has been sketched several times. The chosen color for the line is *white*. As the graphical annotation encloses the silhouette of the depicted person, we denote this effect in the following as **Silhouette Effect**.

In Figure 3.4, the dancer's movements are accompanied by silhouettes of himself to his left and right. For this representation, three silhouette outlines were shifted in their location on the horizontal axis. The positioning of the drawn *silhouette outline* with a solid, white line stroke and the continuous update with each video frame creates the impression of stylized duplications of the person. This effect type is therefore denoted as **Twins Effect**.

Figure 3.2.: *Sample illustration of the **Halo Effect** from the music video "Wait for You" by Tom Walker . It is distinguished by several graphic elements based on the dancer's silhouette at different scaling levels, which continuously follow the dance movements.*

Figure 3.3.: *Sample illustration of the **Silhouette Effect** from the music video "Wait for You" by Tom Walker . It is distinguished by only one graphic element outlining the dancer's silhouette, which continuously follows the dance movements.*

Figure 3.4.: *Sample illustration of the **Twins Effect** from the music video "Wait for You" by Tom Walker . It is distinguished by multiple graphic elements outlining the dancer's silhouette, which are horizontally shifted and continuously follow the dance movements. The silhouettes shown next to the dancer give the impression of stylized copies of himself.*

Figure 3.5.: *Sample illustration of the **Freeze Effect** from the music video "Wait for You" by Tom Walker . It is distinguished by a single graphical element, which outlines the dancer's silhouette only at the beginning of the video effect and is then not changed while the video of the dance performance continues.*

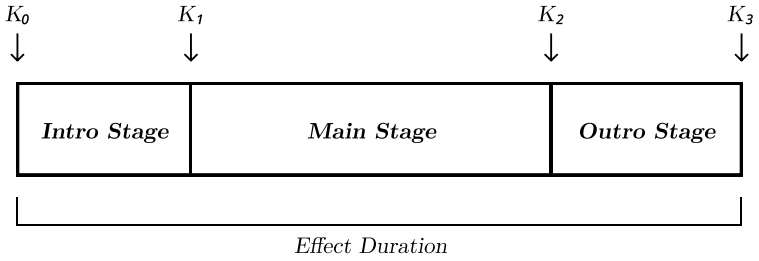


Figure 3.6.: *A video effect is temporally divided into **Intro Stage**, **Main Stage** and **Outro Stage**. Keyframes K_0 to K_3 are used to define the start and end of an effect and the transitions between effect stages. The total duration of an effect is the time difference between K_0 and K_3. Animations of video effects can be created by interpolating visual variables between keyframes.*

While the graphical annotations in Figure 3.2, Figure 3.3, and Figure 3.4 use a solid line stroke type, in Figure 3.5 a *dashed line* is used. In order to uniquely define the visual appearance, the distance between dashes is defined via an additional visual variable *stroke spacing*. With regard to the stroke style, the dashes resemble a chalk line on a blackboard, it is therefore defined as *chalk-like*. In the previous examples, the graphical annotations are always derived from the silhouette outline of the person within the current video frame. In contrast, in Figure 3.5 the dashed line does not correspond to the silhouette of the dancer within each shown frame. In this effect type, the dancer's silhouette outline gets derived once at beginning of the effect and remains unchanged over the length of the effect, while the video continues. This results in a "frozen" appearance of the annotation, therefore this effect is named **Freeze Effect**.

While Halo, Silhouette, and Twins are created based on the current frame only, the Freeze Effect results in a time dependency on an earlier video frame. So while the scene in the video changes, the graphical annotation remains unchanged, making them stand out. To achieve a similar effect for Halo, Silhouette, and Twins Effect so-called effect animations are used. For this purpose, an effect is divided into three sections, so-called animation stages:

- Intro Stage: The Intro Stage can be used to control the start of an effect (i.e. fade-in, the successive appearance of graphic elements, ...).
- Main Stage: The Main stage controls the visual appearance during the main phase of an effect (i.e. pulsing, shifting, freezing).
- Outro Stage: The end of an effect can be animated via the Outro Stage (i.e. fade-out, successive reduction of the graphic elements, ...).

As illustrated in Figure 3.6, each stage refers to a temporal section of an effect and thereby enables the effect designer for a dramaturgical division. Thereby, they give control to the effect designer over how the effect is introduced (Intro Stage), how the

graphical annotations appear within the main part of an effect, and how the effect is broken down within the Outro Stage.

For each effect stage, animations are achieved by defining differing values for visual variables at the beginning and the end of an effect stage. As a result, the appearance of the graphical annotations changes successively over the duration of the effect stage. This concept is similar to the concept of keyframes known from character animation [44] or video editing [17]. Therefore, for our concept, we define a keyframe K_i as a tuple of a timestamp t and a tuple of values of visual variables v (e.g. number of graphical annotations, stroke type, color) and is defined as follows:

$$K_i = (t, (v_0, ..., v_n)), i \in \mathbb{N}_0, t \in \mathbb{R} \tag{3.1}$$

Between two keyframes K_i and K_{i+1} effect values of visual variables are interpolated. The tripartition of an effect on the temporal axis results in four keyframes, K_0 to K_3, at which the effect designer defines values for visual variables. The length of an effect is determined by $t_{K_3} - t_{K_0}$. The ratio between Intro, Main, and Outro Stage can be adjusted by varying t for K_1 and K_2. Furthermore, the use of Intro and Outro Stage is optional for the effect designer, as they may not be desired for some effects (e.g. Freeze Effect).

3.2. Multidimensional Video Processing Framework

The silhouette-based motion visualization effects described in the previous section can basically be implemented as independent sequential routines, as shown in Figure 3.1. However, the following requirements, formulated in subsection 1.2.1 and subsection 1.2.2, indicate to use a more flexible processing framework:

- App provides live preview while video editing (cf. F-04).
- Effect parameters can be adjusted from the user interface (cf. F-04).
- Adaptable for other data and use cases (cf. NF-03).
- Extendable with new effects and additional parameters (cf. NF-03).

Meeting these requirements not only requires the sequential creation of effect frames, including outline tracing, stylization, and blending steps. Also, a stream processing workflow is required, which provides interactive performance and thus enables a live preview. To allow interactive adaptability of effect parameters, visual variables must be editable by the user while the video gets presented within the live preview. Finally, extending a processing flow by additional effect types or data types and formats requires a high flexibility of the processing concept to avoid extensive new implementations.

In the following a concept for a framework is proposed that is able to process a multitude of data types and formats with interactive performance while allowing a video preview, parameterization of effects at render time as well as easy maintainability and expandability for new effect and data types.

3.2.1. Graph-Based Processing Concept

In contrast to Figure 3.1, which illustrates a high-level abstraction of the processing steps necessary to generate a motion visualization effects, the single steps (**outline tracing**, **stylization** and **blending**) in fact can be divided into a multitude of successive processing routines. Following the paradigm “Divide and Conquer” [8, p. 209], a decomposition of video processing and effect creation into small sub-steps leads to easier manageability. Such small, self-contained processing steps can then be combined into a so-called processing pipeline. In addition, implementations of sub-steps can thereby easily be reused for similar or new processing pipelines and video effects, as required by NF-03.

We represent the processing pipeline as a directed acyclic graph, with individual processing steps represented as nodes and data flowing between graphs as edges (Figure 3.7). Each node has a set of typed input ports and output ports, which for the same data type can be connected to build the graph, so that data of a certain type can flow between two nodes.

3.2.2. Reactivity of Node Ports

To design data flow between nodes consistently, we propose to implement the input and output ports of nodes in a *reactive* manner (cf. reactive programming, [6]). Reactivity means that the processing of a node is triggered by a change of value at its input ports. Since processing results are immediately passed on via the output ports to the following node, a cascade effect is triggered, which leads to sequential processing of nodes within the processing graph. The processing of graph branches can be omitted if values at the input ports of a node have not changed, which reduces the processing effort to a necessary minimum.

To increase performance, complex processing operations for image data typically get shifted from the Central Processing Unit (CPU) to specialized hardware components, such as Graphical Processing Unit (GPU) or Neural Processing Units (NPU). In order to free up processing resources on the CPU while data is processed on GPU or NPU, the asynchronous programming paradigm is used. By this, other computational tasks can be executed on the CPU while waiting for results from GPU or NPU. [26] The reactive manner of node ports ensures a sequential data flow, as calculation results are passed on to the subsequent nodes as soon as this data is available. By the asynchronous processing algorithms, enabled by the reactive concept of node ports, flexibility in effect development and optimization of performance is provided.

Nodes can provide more than one input port. If so, it is necessary to specify when a node starts processing. Options are whenever new data is provided:

1. On any input port.
2. On a specific input port.
3. On a group of specific input ports since the last processing.
4. On all input ports since the last processing.

Which of these options a processing node should use is up to the effect developer. By implementing a caching mechanism on output ports, the performance of the processing graph can be further optimized, as it allows a calculated processing result to be retrieved again.

3.2.3. Graph Nodes for Specialized Processing Tasks

In the following, the flexibility of the graph-based processing concept is explained by means of an example. The goal is to create a stylized Bezier path using an RGB frame (cf. Figure 3.7). This can be realized by chaining the following processing nodes:

1. A *Human Segmentation Node*: It receives an RGB video frame on its input port, transforms the data into a human segmentation matte, and returns it via its output port.

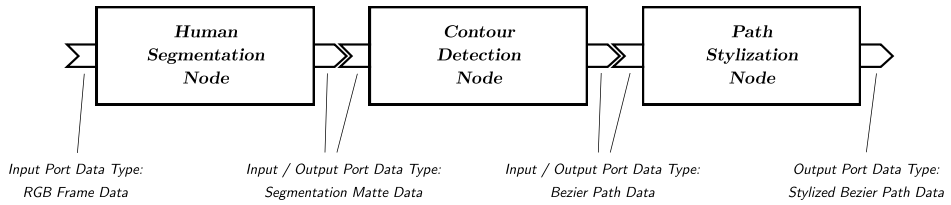


Figure 3.7.: *The illustrated processing graph shows a selection of processing nodes used to generate Motion Visualization Effects. Processing nodes can be connected using typed input and output ports with the same data type to create complex data processing flows for creating video effects.*

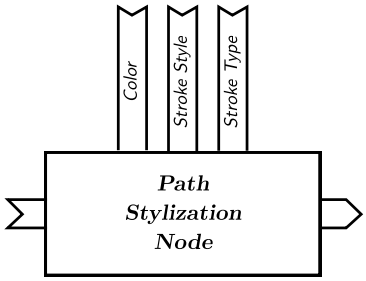


Figure 3.8.: *Processing nodes that affect the visual appearance of video effects can be extended with input ports to adjust visual variables while processing. Visual variables can either be adjusted by user input or in an automated manner, e.g. for the creation of effect animations.*

2. A *Contour Detection Node*: Receiving a human segmentation matte, applies a contour detection algorithm on it, and returns the resulting bezier path.

3. A *Path Stylization Node*: It receives a bezier path on its input port, stylizes the path by provided variables (e.g. stroke type, stroke style, and color), and returns a stylized bezier path.

To customize the appearance of a bezier path, the *Path Stylization Node* can be extended by additional input ports (cf. Figure 3.8). Due to the reactive implementation of input ports, changing these parameters, e.g. by user input (F-04), leads to an immediate update within the processing graph.

In order to fetch input data for the processing graph and write its result to a video file, reader and writer nodes are proposed. Specialized reader nodes read multidimensional videos frame-by-frame and provide the individual data (e.g. RGB video data, human segmentation data, ...) as typed frame data to their subsequent nodes in a synchronized manner (cf. Figure 3.9).

Due to the reactive approach, with each multidimensional set of frame data, a

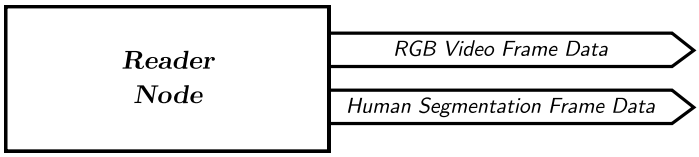


Figure 3.9.: *The reader node is used as a root node of a processing graph and reads multidimensional data from a multidimensional video file or multidimensional camera feed and sequentially provides the read multidimensional data to subsequent nodes in a synchronized manner. Typical data types in multidimensional videos include, among others, RGB data, human segmentation data, and Lidar depth data.*

processing pass is triggered. Thus, passing the data to the following nodes, the reader node determines the frequency of processing passes. If processing results are to be displayed in a preview with interactive performance, the processing frequency also corresponds to the frame rate of the preview video. To achieve the desired frame rate while respecting the utilization of hardware resources (CPU, GPU, NPU, read and write speeds of file storage), we propose to equip reader nodes with a scheduling routine, which is able to control the frequency of processing passes. Since the exact design of such a scheduling routine depends on multiple factors, like application scenarios, hardware configurations, and data resolutions, it will not be further discussed further at this point.

3.2.4. Adaptability of Processing Graphs

A complete processing graph for generating **Silhouette Effects** is shown in Figure 3.10. It extends the exemplary graph in Figure 3.7 by reader and writer nodes as well as a *Path Blending Node*, which blends a stylized bezier paths with the original RGB frame to obtain the resulting video frames enhanced by the **Silhouette Effect** (Figure 3.10).

The node-based concept allows new functionality can easily be added. Also, graphs for new video effects can be derived from an existing graph with little modification while a large part of already implemented nodes can be reused (cf. NF-03). The following examples exemplarily illustrate the adaptability of this graph-based processing concept.

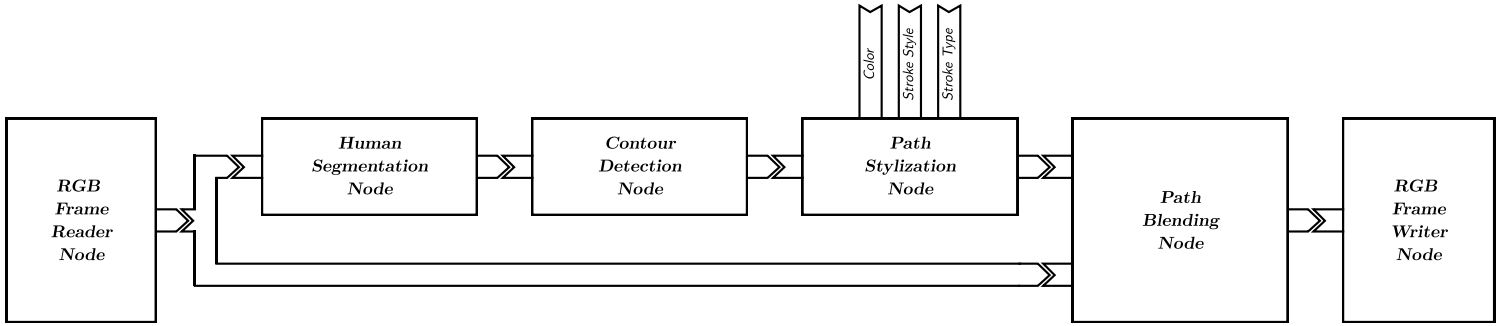


Figure 3.10.: *The depicted processing graph consists of a configuration of processing nodes for creating* ***Silhouette Effects****. Video frames from an RGB video are read by a RGB Frame Reader Node. The Human Segmentation Node masks people in the video frame, this mask information is used by an Contour Detection Node to generate silhouette outlines. The resulting bezier paths are styled by a Path Stylization Node. The stylized bezier path and the original RGB video frame are blended into a single video frame using the Path Blending Node, which finally gets written to a video file using a RGB Frame Writer Node.*

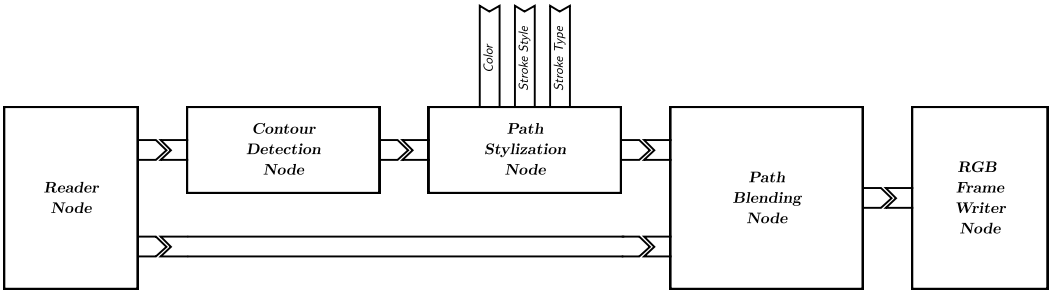


Figure 3.11.: *By reading precomputed human segmentation matte data together with RGB data from a multidimensional video file, the complexity of the processing graph in Figure 3.10 can be reduced and the processing performance increased.*

Precomputing Human Segmentation Mattes

Computing high-resolution human segmentation mattes are computational expensive [10] and, if an end device does not have sufficient hardware resources available, can result in non-interactive processing performance. One approach to solve this issue is to precompute the human segmentation matte frames before effect processing and store it as a dataset in a multidimensional video file (cf. section 2.2). For processing, RGB frame with associated human segmentation matte frames can then sequentially be read from the multidimensional video file by a *Reader Node* and the *Human Segmentation Node* is no longer needed (cf. Figure 3.11).

From Silhouette Effect to Halo Effect

The difference between Silhouette Effect and Halo Effect consists of the number and scale of graphical annotations, as well as stroke style and color (cf. Table 3.1). Visual variables for stroke style and color can be configured via the input ports at the *Path Stylization Node*. According to this, only the stylized bezier path must be duplicated in different scaling sizes for changing the processing pipeline of a Silhouette Effect to compute a Halo Effect. For this purpose, a corresponding node can be inserted into the known processing graph (cf. Figure 3.11) between *Path Stylization Node* and *Path Blending Node*, which duplicates the stylized bezier path using duplication and scaling factors provided via input ports (cf. Figure 3.12).

3.3. App and User Experience design

While the previous section presented a technical framework for creating silhouette-based motion visualization effects, in this section the focus is on the graphical user interface concept of a mobile app utilizing this framework. For this purpose, in the following, a view hierarchy is proposed including necessary app views for controlling such an app.

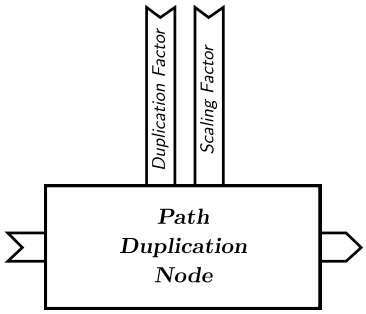


Figure 3.12.: *The Path Duplication Node scales and duplicates stylized bezier paths. The number of duplicates and a fixed value of scaling between duplicates can be set via the nodes' input ports.*

Further user interface specifications are derived from the functional and non-functional requirements, as formulated in subsection 1.2.1 and subsection 1.2.2. Finally, the central app views for creating and configuring video effects are conceptualized.

3.3.1. The App View Hierarchy

The structure of the mobile app is divided into several app views, which provide the user with the necessary functionality for capturing and managing multidimensional videos as well as enhancing videos with silhouette-based video effects. Figure 3.13 shows the hierarchical arrangement of the views, which provide the following functionalities to the user:

- Main View: The entry point of the app, which provides buttons to navigate to the subordinate views.
- Capture View: Provides a live preview of the camera stream while recording multidimensional video from multiple sensors.
- Import View: Allows to import of previously captured multidimensional video files.
- Edit View: Provides a timeline-based video preview interface and allows to place video effects on the video.
- Parameter View: This allows the user to customize effect appearance by adjusting effect parameters.
- Export View: This allows the user to export the resulting video and save or share it with others.
- Projects View: Using the Projects View, the user is able to save a current state of editing within a project to continue editing at a later point in time.

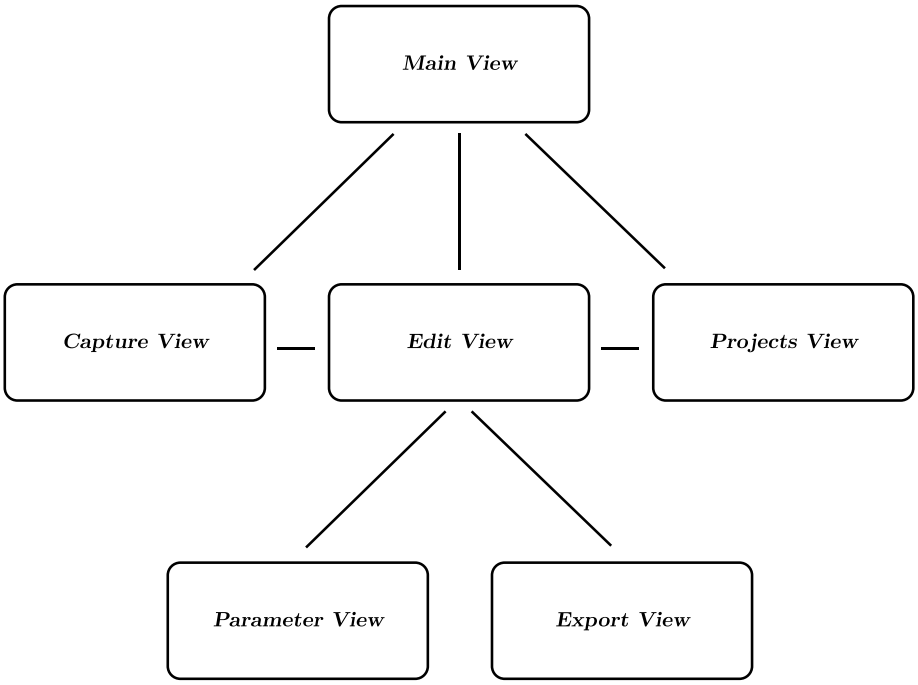


Figure 3.13.: *The structure of the mobile app is divided into several app views, which provide the user with the necessary functionality for video creation. The user is able to navigate between app views by buttons, indicated by the shown lines between views. The Main View is the entry point of the app. The user is able to capture multidimensional videos using the Capture View. Captured videos can be enhanced by silhouette-based video effects within the Edit View. Applied effects are fine-tuned using the Parameter View. After editing, the resulting video is created and saved or shared via the Export View. Additionally, the user is able to save the current state of editing as a project to continue editing at a later point in time using the Projects View.*

3.3.2. Intuitive App Interface for Video Effects

Specific requirements for the user experience of the mobile app result from the functional and non-functional requirements formulated in subsection 1.2.1 and subsection 1.2.2. In the following, user experience requirements and the resulting user interface concepts are presented.

From NF-01 follows, that the user interface of a mobile app must be *intuitive* to the user, and using the app must not require any professional knowledge. Regarding Baerentsen, "[...] an intuitive interface may be defined as an interface, which is immediately understandable to all users, without the need neither for special knowledge by the user nor for the initiation of special educational measures. Anybody can walk up to the system; see what kind of services it affords, and what should be done in order to operate it." [5] As Klingbeil et al. noted in "Challenges in User Experience Design of Image Filtering Apps" [24], using established control concepts is expected by users when operating stylization apps.

Therefore, well-known and proven interaction patterns for touch devices, such as button taps, tap-and-hold, and tap-and-drag gestures are used to provide users with intuitive control of the app. Also, interaction buttons in the app are described by self-explanatory icons or meaningful texts. Different app functions (e.g. effect types) are visually distinguished from each other by color coding.

Since the app targets the user group of user-generated content and social media users, expertise in artistic expression, video processing, and effects creation cannot be assumed and should not be a requirement for using the app. Isenberg discusses which degrees of interaction are required to make *Non-Photorealistic Animation and Rendering* (NPAR) algorithms, to which silhouette-based video effects can be counted, practically useful. With regard to the way users want to interact with NPAR, he states that "the type of interactivity needed for an NPAR tool highly depends on the intended audience." For non-artists, i.e.,"people not trained in the visual arts", he sees the desire for high level and rather less fine-grained control. He further assumes that "many people of this user group prefer tools in the form of filters that can easily be applied to visual media and that have some easy-to-control and easy-to-understand parameters." This corresponds with Baerentsen's definition of intuitive interfaces, which should be able to be controlled by anyone without any special knowledge or education of the interface. [20]

However, as explained in the previous subchapter, creating silhouette-based video effects requires the complex tasks of defining a variety of visual parameters and assembling individual processing steps into a processing graph. To still provide a wide audience of users with the ability to interactively and intuitively apply and customize video effects, by the use of "easy-to-control and easy-to-understand parameters", this technical complexity must be reduced.

Based on the findings of Isenberg and Klingbeil et al., we propose the following specification for an intuitive user interface concept for creating video effects:

1. **Effect Presets**: For each processing graph, the effect designer creates a so-called *Preset* by setting default values for all parameterization options (visual or technical

variables). The resulting *Presets* are displayed as single buttons to the user to allow for an efficient high-level effect creation.

2. **Visual Evaluation**: Users are enabled to quickly and easily place, move, and delete video effects for each effect type to visually explore and evaluate the processing result for a given video.

3. **Constrained High-level Parameterizations**: Only a curated subset of parameters, selected by the effect designer, is presented to the user. Therefore, several parameters on the processing graph can be combined into one control parameter for the user. For every control parameter, the effect designer specifies the value range, in which the parameter can be manipulated by the user.

4. **Uniform Parameter Interfaces**: Starting from the *Preset*, the user adjusts the appearance of effects via the provided parameters. For different types of parameters (e.g. numerical values, set of distinct values, colors) the user is offered interaction elements, which are similar in appearance and interaction so that the user must not know about the underlying technical parameters. Parameter adjustments can be explored and immediately assessed visually via a video effect preview.

By reducing the complexity of video effects to a few intuitively understandable parameter options during the app's development process, and by allowing the user to playfully evaluate the adjustment of parameters visually at any time, a low-threshold user interface for a broad user group in the field of user-generated content is created.

Through observations, we found that effect creation is often an iterative process. To support this, all configurations to the video are saved. To enables the user to recall previously created configurations, these can be saved as a *Project* and accessed via the Project View. Therefore, a consistent effect representation and quality are critical (NF-02), which is ensured by the implementation of the processing graph.

3.3.3. Timeline-Based Video Effect Creation

Applying video effects to a video is a visual task, as the main objective of the user is the visual appearance of the resulting video. By conducting qualitative user studies with the app and other video editing tools, we identified the following important steps in video stylization:

1. Reviewing video footage to find the right position for an effect.

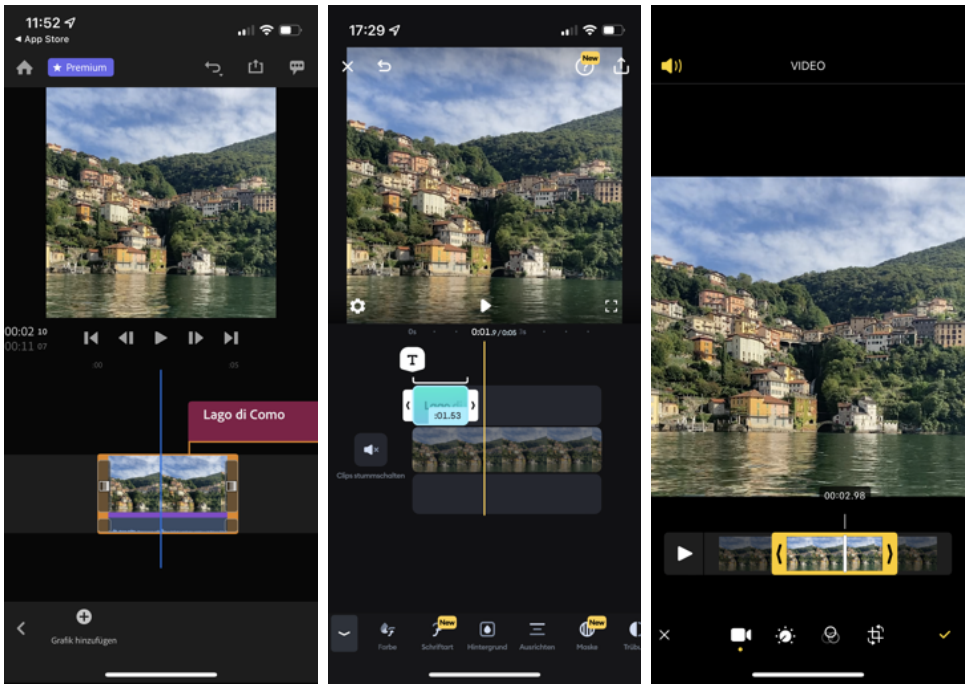
2. Placing a video effect on a part of the video.

3. Adjusting the exact playback position and duration of the video effect.

4. Adjusting the appearance of the effect by customizing effect parameters.

5. Repeating one or multiple of the above steps until the desired appearance is found.

Users of mobile touch devices are familiar with the concept of timeline sliders for videos, it enables them to see the current playback position in relation to the video length and to navigate through the video by moving forward and backward using a tap-and-dragging gesture. Video apps and tools, like Adobe Premiere Rush [1], Splice [41] or Apple Photos [37] also use timeline representations to allow the user to sequence and trim video clips as well as apply video filters and effects (cf. Figure 3.15).



(a) *Screenshot of the timeline-based video editing app Adobe Premiere Rush on an Apple iPhone.*

(b) *Screenshot of the timeline-based video editing solution Splice on an Apple iPhone.*

(c) *Screenshot of the video trimming functionality within the Apple Photos app on an Apple iPhone.*

Figure 3.14.: *Timeline sliders are a typical interaction element on touch-enabled devices for navigating and manipulating videos. The screenshots show well-known app examples for the Apple iOS operating system.*

We adopt this proven user interface concept to enable the creation, arrangement, and editing of silhouette-based video effects (cf. F-04). Effects are created by a tap-and-hold gesture on the desired effects *Preset* button. With the first tap on the button, a new effect is created at the current playback position, keyframe K_0 is set to the respective video timestamp. While holding the effect *Preset* button, the chosen effect is applied to the video shown within the preview (cf. F-04) and the position of the effect is marked on the timeline slider (*Timeline Effect Marker*, cf. Figure 3.15(a)). Lifting the finger from the screen ends the effect and sets keyframe K_3 to the respective video timestamp (cf. Figure 3.15(b)).

During effect creation, the Intro Stage effect animation is displayed with a specified length, as defined by the effect *Preset*. The Outro Stage is not displayed at effect creation, since K_2 is calculated from the defined length for the Outro Stage and the timestamp of K_3. Once an effect is created and all keyframes are specified, the user is able to preview the complete effect with its animations.

Effects can be created both when the video is playing and when it is paused. During the effect creation, the video with the applied effect can always be previewed and after the effect ends, the playback returns to the previous state.

An effect can be repositioned by the user by tap-and-dragging the timeline effect marker. The duration of an effect can be adjusted by tap-and-holding the marked button areas at the start and end of an effect to enlarge or shorten it (cf. Figure 3.15(c)). Thereby K_0 and K_3 get adjusted and K_1 and K_2 recalculated so that the duration of the Intro and Outro Stage stays unchanged. The effect designer is able to set time constraints for the minimum and maximum duration of an effect within the effect *Preset*, which is reflected in the user interface by a timeline marker not drawing larger or smaller than defined.

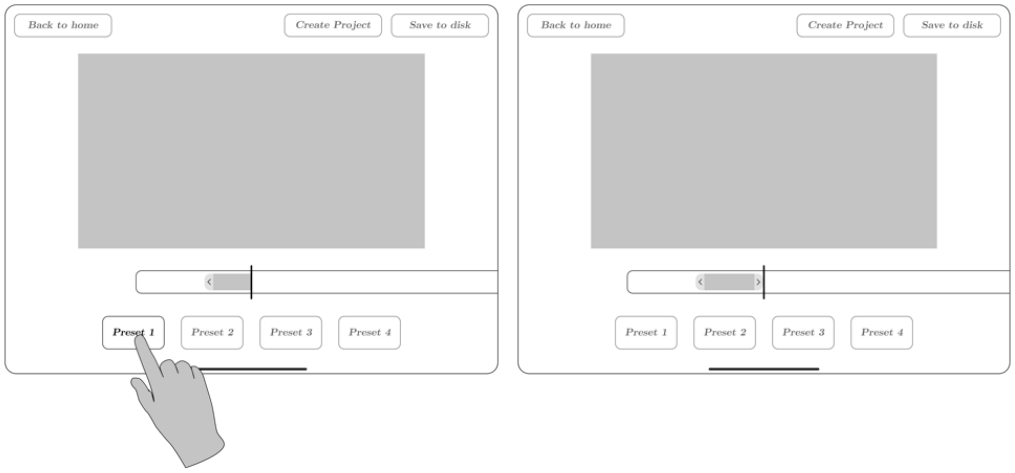
By tapping a timeline marker, an effect tooltip with buttons for parameter adjustments and effect deletion is shown (cf. Figure 3.15(d)). While the delete button immediately deletes the selected effect, the second button opens the *Parameter View* for customizing the effect representation, as explained in the following.

3.3.4. Customizing Effect Representations

Within the Parameter View (cf. Figure 3.16), the user is able to apply adjustments to the visual appearance of a created effect (F-04). The user can thus, within the limits set by the effect designer, individualize the effect.

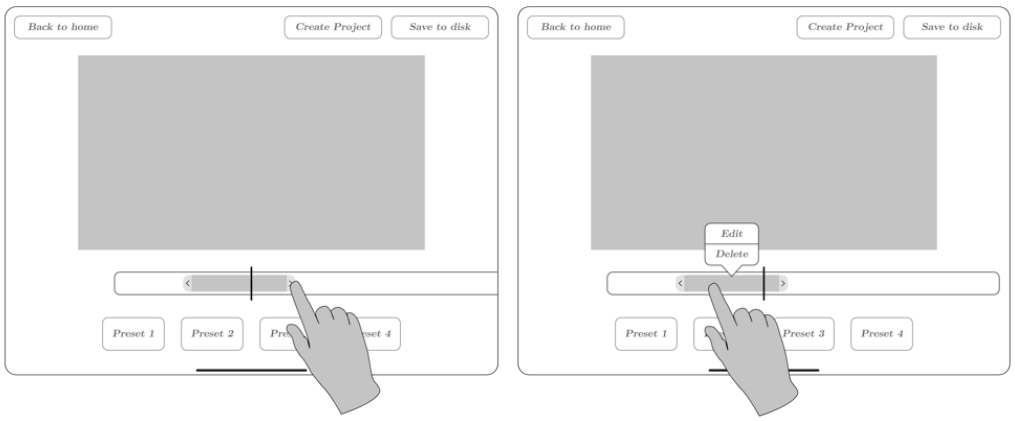
Therefore, the user must be enabled to manipulate the values of the processing graph from the user interface. To provide the user with intuitive and uniform parameter interfaces (cf. subsection 3.3.2, Uniform Parameter Interfaces), the effect designer is provided with the following four generic user interface elements:

- **Switch**: The user selects between two distinct values, e.g. for enabling or disabling the Intro Stage (cf. Figure 3.16(a)).



(a) *An effect is created by tap-and-holding on one of the effects Preset buttons. The position of the video effect is visualized using a Timeline Effect Marker.*

(b) *The end of an effect is defined by releasing the touch gesture.*



(c) *An effect is repositioned using a tap-and-drag gesture on the Timeline Effect Marker. Using the marked areas at the start and end of the Timeline Effect Marker, the effect can also be adjusted in its length.*

(d) *Customizing and deleting an effect is achieved by tapping on the relevant Timeline Effect Marker, which opens a dialog.*

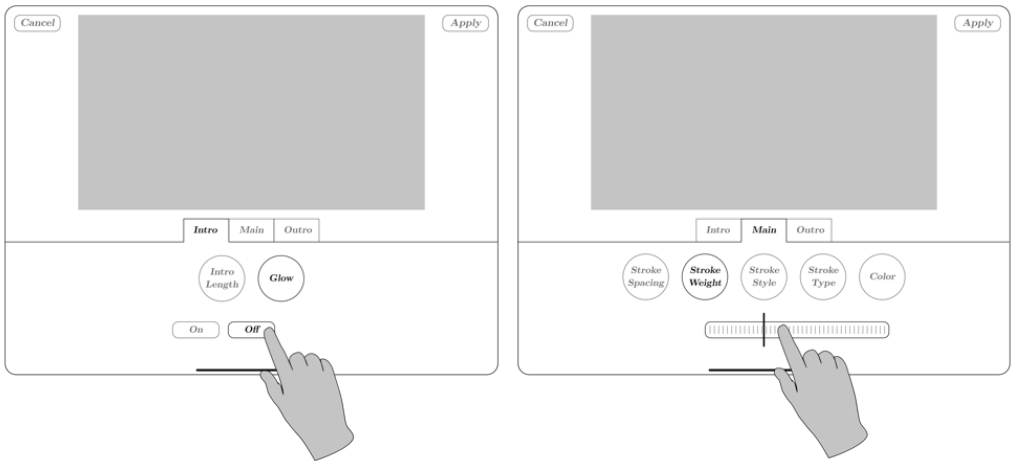
Figure 3.15.: *From the apps Edit View the user reviews the selected video footage and creates, adjusts, and deletes silhouette-based video effects. By tapping an effect Preset button, the effect can be previewed while the effect gets placed in the timeline. The user adjusts the position and length of an effect by well-known tap-and-drag gestures within the timeline slider. Tapping an effect marker within the timeline allows to customize the effect representation via the Parameter View as well as to delete an effect.*

- **Slider**: The user selects a numeric value within a given range, e.g. for adjusting the duration of the Intro Stage (cf. Figure 3.16(b)).
- **Stepper**: The user selects one value from a set of distinct values, e.g. the number of outlines for the Halo Effect (cf. Figure 3.16(c)).
- **Color Picker**: The user selects a color from a set of colors, e.g. the outline color (cf. Figure 3.16(d)).

Input ports of the processing graph are linked to parameter interfaces, giving the user direct access to customize input values of a processing node (e.g. Stroke Style, Stroke Weight, Color, Intro/Outro Stage duration). This can be achieved either via a direct connection between the parameter interface and input port or via so-called meta-parameters. The latter allows the user to change the values of several input ports by a single parameter interface. The mapping between parameter interface and input ports is thereby specified by the effect designer.

For each parameter interface, the effect designer specifies meaningful value ranges or value sets, within which the user is able to apply adjustments (cf. subsection 3.3.2, Constrained High-level Parameterizations). Further, the effect designer is able to categorize parameter interfaces into Intro, Main, and Outro stages. By this, the user is presented with different customization options for the three temporal sections of an effect.

Parameter interfaces implement the same reactive mechanism as processing nodes so that changes via the user interface are instantly reflected within the processing graph. Thereby, the video effect preview within the Parameter View gets immediately updated, providing the user with immediate visual feedback of the adjustments, which further contributes to the intuitive and explorative approach of the app. For the effect designer, the predefined elements provide a programming interface between the processing graph and interface elements, which abstracts from the technical complexity and enables to provide meaningful effect adjustment options to the user using uniform user interface elements.



(a) *Using the Switch interface, the user is able to select between two distinct values.*

(b) *A numeric value within a given value range can be selected using the Slider interface.*

(c) *The user is able to select from a set of distinct value choices using the Stepper interface.*

(d) *For specifying a color choice, the Color Picker interface is available.*

Figure 3.16.: *The app's Parameter View allows the user to customize effect appearance. Therefore, four generic parameter interfaces are provided: Switch (Figure 3.16(a)), Slider (Figure 3.16(b)), Stepper (Figure 3.16(c)) and Color Picker (Figure 3.16(d)).*

Chapter 4

Implementation

This chapter is dedicated to the implementation of the processing framework and the MotionViz app based on it. The general system architecture, the components contained and the external libraries and frameworks used are explained. Furthermore, the implementation of the multidimensional video processing framework and the functionality of the node-based processing, the implementation of effects, and the integration of effects into processing pipelines are explained. Finally, the developed mechanism to automatically display user interface elements for manipulating effect parameters is presented.

4.1. System Architecture

The mobile app is developed for the reference hardware of an iPad Pro (11-inch, 2nd generation with A12Z GPU and 6 GB shared memory, running iPad OS 15), as it has the necessary hardware for recording multidimensional video data. The mobile app is implemented in the Swift programming language (version 5) and uses numerous software libraries and frameworks provided by Apple. The project code is structured into five main components *User Interface*, *Pipeline*, *Managers*, *IO* and *Processing* (cf. Figure 4.1). The tasks of the individual components as well as the external frameworks and libraries used for it are explained in the following.

The *User Interface* component implements the graphical representation of the app and allows the user to interact with app functionalities, such as creating effects and adjusting effect parameters. The app views included in this component are implemented based on the specifications described in subsection 3.3.1 using the SwiftUI framework[1]. For the development of graphical user interfaces, the reactive programming paradigm is used, since it "is well-suited for developing event-driven and interactive applications" [6]. For the Swift programming language, reactive programming is enabled by the usage of the Apple Combine Framework[2]. Furthermore, the AVKit library[3] is used in this component to enable the usage and control of media playback.

The *Pipeline* component implements the multidimensional video processing framework and thus represents the core component of the app. It is divided into subcomponents for the implementations of effects and nodes, using the terminology introduced in sec-

[1] Apple framework for graphical user interfaces, cf. [42]
[2] Apple framework for asynchronous and event-driven processing, cf. [12]
[3] Apple library for creating user interfaces for media playback, cf. [4]

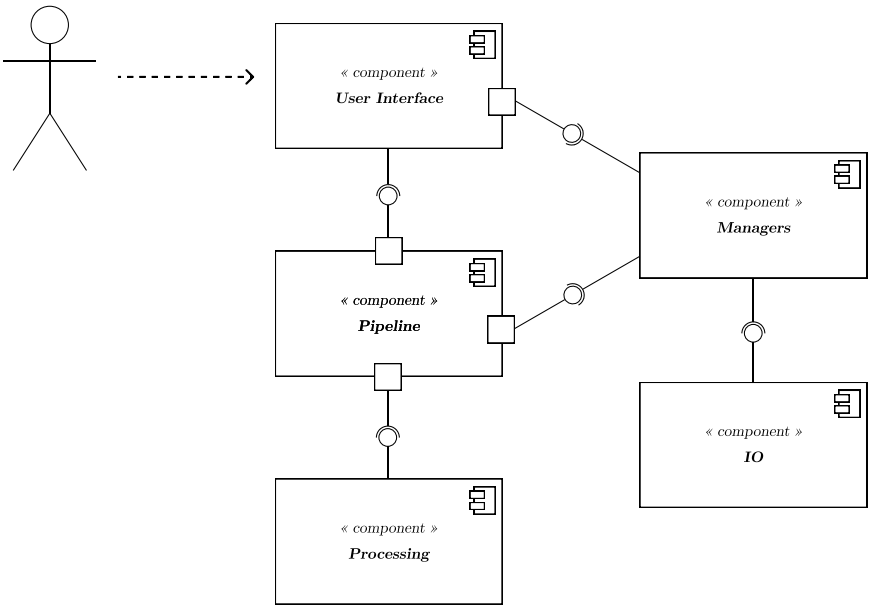


Figure 4.1.: *The component diagram shows the five main components of the app: The user interface component implements the graphical representation of the app and allows the user to interact with the app functionalities. The pipeline component implements the multidimensional video processing framework and, thus, represents the core component of the app. The managers component contains a set of state managers that act as interfaces between user interface, pipeline and IO component. The latter provides data access to sensors, local memory, and the media library of the device. Additionally, custom-developed processing routines are bundled in the Processing component.*

tion 3.2. The processing nodes use the Vision[4], Core Image[5] and AVFoundation[6] libraries for the implementation of computer vision, image manipulation and IO-related tasks. Data flow between processing nodes (cf. Input and Output Ports, subsection 3.2.2) was implemented also using the *Combine* framework, which enables both the reactivity of interfaces between nodes as well as the implementation of asynchronous processing routines.

The *Managers* component contains a set of state managers that act as interfaces between the *User Interface*, the *Pipeline* and the *IO* component. The latter provides data access to sensors, local memory, and the media library of the device in order to record multidimensional data and store data such as recorded multidimensional and processed videos. It also provides implementations to persistently save processing states as a *Project* (cf. Project, subsection 3.3.2) file using the Zip library[7], so that the user can continue video editing at a later time.

Most computer vision and image manipulation algorithms necessary for creating silhouette-based video effects are provided by Apple frameworks and libraries. Furthermore, the proprietary still and video processing framework Metal Rendering Core[8] and the open source library CoreImageExtensions[9], both provided by Digital Masterpieces GmbH, were used. Additionally, this project developed processing routines, which extend the *Processing* component and partly use Apple's MetalKit library[10] for GPU-accelerated processing.

4.2. Implementation of the Multidimensional Video Processing Framework

In the following implementation of the multidimensional video, the processing framework is described in more detail. In order to meet the specified functional and non-functional requirements (cf. subsection 1.2.1 and subsection 1.2.2), particular care was taken when implementing the *Pipeline* component to ensure that the processing framework can be easily adapted to new data types and can be extended with new functionalities, e.g. by adding processing nodes and effects.

By implementing the basic pipeline framework using *Generics* [16], the framework is not type-agnostic and can basically be used to process any type of temporal data. Thereby, only in the implementation of specific processing nodes the expected data types (e.g. `Rgb`, `Segmentation`, `UIBezierPath`) must be specified. This allows the processing implementation to be adapted to new data types without changing architecture and interfaces.

[4] Apple framework for computer vision algorithms, cf. [46]
[5] Apple framework for still and video image processing, cf. [13]
[6] Apple framework for working with audiovisual assets and sensor data, cf. [3]
[7] Open Source library to bundle and compress files, cf. [29]
[8] Proprietary still and video processing framework by Digital Masterpieces GmbH
[9] Open source extension library for Apple Core Image by Digital Masterpieces GmbH, cf. [18]
[10] Apple library for creating GPU-accelerated apps, cf. [31]

```swift
public final class Frame<T>: TimedData {

    var value: T
    var timestamp: CMTime

    init(value: T, timestamp: CMTime) {
        self.value = value
        self.timestamp = timestamp
    }

    func adapt<NewType>(for newValue: NewType) -> Frame<NewType> {
        return Frame<NewType>(
            value: newValue,
            timestamp: self.timestamp
        )
    }

}
```

Listing 4.1: *Simplified except of the* `Frame` *class from* `Frame.swift`*. It wraps processing data by assigning it to its* `value` *property and can be adapted to a new* `Frame` *instance of differently typed* `value` *property while passing its meta data, the* `timestamp` *property, via the* `adapt<NewType>(for newValue: NewType)`*.*

Frame

A central part of the pipeline implementation is the generic `Frame` class. It provides a wrapper for arbitrary data and includes necessary metadata for processing (e.g. associated position of a frame in the video, frame rotation information, timestamp of the current frame, ...). A `Frame` instance traverses the processing pipeline by getting passed from processing node to processing node. Due to the generic definition, a `Frame` instance can store RGB video frames, bezier paths, segmentation data, depth data and more within its `value` property (cf. Listing 4.1). If the input data type is different from the output data type of a processing node (e.g. a bezier path is generated from an RGB frame), a new `Frame` instance with a different type can be created using the `adapt<NewType>(for newValue: NewType)` method, passing its metadata to a new, differently typed instance.

Node

The interface between nodes (cf. Input and Output Ports, subsection 3.2.2) is implemented by means of a producer-consumer pattern using the Apple *Combine* framework. Thereby, for each newly `Frame` instance received from a node, the data of the `value` property is

processed and the processing results are in turn passed on to subsequent nodes.

In *Combine*, producers are implemented via the `Publisher` protocol. As the name suggests, publishers publish new data by forwarding it to one or multiple registered subscribers via an asynchronous event. The association between publishers and subscribers is established either explicitly via the `subscribe(_:)` method or the `sink(receiveCompletion:receiveValue:)` operator on a class implementing the `Publisher` protocol. For disassociation between publisher and subscriber, for example, when all available data has been sent, the publisher sends an event of type `Subscribers.Completion` to registered subscribers. In this case of a video processing pipeline, this mechanism is used to indicate that the last video frame has been processed.

Four generic *Input Node* classes, to which extent to a basic `Node` class, have been implemented to provide multiple numbers of reactive input ports: `SingleInputNode<Input>`, `DualInputNode<Input1, Input2>`, `TripleInputNode<Input1, Input2, Input3>` and `QuadInputNode<Input1, Input2, Input3, Input4>`. The generic types `Input` and `Input1` to `Input4` determine the expected input data type for processing and get specified in subclasses.

Listing 4.2 shows a simplified implementation of the `SingleInputNode<Input>` class: The constructor of the class receives a publisher of generic type `Upstream`, where the data type to be sent by the `Publisher` is restricted to the data type to be processed by this node.

The `setupPipeline(input:)` method is called within class instantiation to subscribe the node to new values from the given publisher. The `sink(receiveCompletion:receiveValue:)` method defines closures that handle both the reception of new data as well as the disassociation of the node from the publisher. The latter are passed to the `process(input:)` method, which is to be implemented by subclasses of the `SingleInputNode<Input>`.

By the use of Generics, `Input Nodes` are designed for processing any type of data. Thereby the framework can be flexibly adapted to other use cases. Since in this work the data is always wrapped in a `Frame` instance, only input nodes of the generic type `SingleInputNode<Frame<Input>>` are used. For convenience, this type specification and also the type definitions of `DualInputNode`, `TripleInputNode` and `QuadInputNode` can be shortened by the use of type aliases (cf. Listing 4.3), so that only the data type included in the `Frame` instance needs to be specified.

Based on the *Input Nodes* and type aliases, specific node implementations for processing data can be created. Listing 4.4 shows a simplified implementation of the `ContourNode`, which creates contour paths from RGB frames. Since the creation of subscriptions is already implemented within the `setupPipeline(input:)` method of the parent class, the implementation of the `ContourNode` is limited to the `process(input:)` method and the definition of the parameter `contour`, which holds the processing result. For each published `Frame`, the `process(input:)` method is called and computes an `UIBezierPath` from the RGB data provided in the `Frame` instance using the Vision Framework. The result of the `VNDetectContoursRequest`, which is received asynchronously, is adapted to a new `Frame` instance and assigned to the `contour` property of the `ContourNode`.

```
open class SingleInputNode<Input>: Node {

    init<Upstream: Publisher>(input: Upstream)
        where Upstream.Output == Input, Upstream.Failure == Never {
        super.init()
        setupPipeline(input: input)
    }

    func setupPipeline<Upstream: Publisher>(input: Upstream)
        where Upstream.Output == Input, Upstream.Failure == Never {
        input.sink { [weak self] completion in
            self.finish(with: completion)
        } receiveValue: { [weak self] val in
            self.process(input: val)
        }
        .store(in: &cancellables)
    }

    func process(input: Input) {
        fatalError("This method must be implemented by subclass!")
    }
}
```

Listing 4.2: *The* `SingleInputNode` *class is one of four generic base classes for processing node implementations. It receives a single value publisher compatible with the* `Input` *type. Within the constructor,* `setupPipeline(input:)` *is called, which subscribes to values published by the publisher. For each received value, the data is passed to the* `process(input:)` *method. Subclasses overwrite* `process(input:)` *to implement the actual data processing.*

```
typealias SingleFrameNode<Input> = SingleInputNode<Frame<Input>>
typealias DualFrameNode<Input1, Input2>
    = DualInputNode<Frame<Input1>, Frame<Input2>>
typealias TripleFrameNode<Input1, Input2, Input3>
    = TripleInputNode<Frame<Input1>, Frame<Input2>, Frame<Input3>>
typealias QuadFrameNode<Input1, Input2, Input3, Input4>
    = QuadInputNode<Frame<Input1>, Frame<Input2>, Frame<Input3>,
        Frame<Input4>>
```

Listing 4.3: *By using type aliases, the generic node classes,* `SingleInputNode`*,* `DualInputNode`*,* `TripleInputNode` *and* `QuadInputNode`*, are specified for processing data wrapped in a* `Frame` *instance. A processing node that inherits from these nodes therefore only needs to specify the generic type* `Input`*, which determines the data type of the data contained in a* `Frame` *instance.*

```swift
final class ContourNode: SingleFrameNode<Rgb> {

    @Output var contour: Frame<UIBezierPath>

    override func process(input image: Frame<Rgb>) {
        let request = VNDetectContoursRequest { request, error in
            self.contour = autoreleasepool {
                guard let path = request.results?.first as?
                    VNContoursObservation else { return nil  }
                return image.adapt(for: UIBezierPath(cgPath: path))
            }
        }
        let imageRequestHandler = VNImageRequestHandler(
            ciImage: image.value, options: [:]
        )
        try imageRequestHandler.perform([request])
    }
}
```

Listing 4.4: *Shown is a simplified except of the* `ContourNode` *class from* `ContourNode.swift`*. Due to the functionality inherited from the* `SingleFrameNode`*, the implementation mainly consists of the* `process(input:)` *method for extracting an* `UIBezierPath` *from an* `Rgb` *frame. The* `@Output` *property wrapper provides the created* `UIBezierPath` *data via a publisher to subsequent nodes.*

```swift
@propertyWrapper final class Output<T>: OutputProtocol {

    private(set) var subject = PassthroughSubject<T, Never>()

    var wrappedValue: T {
        set { self.subject.send(newValue) }
    }

    var projectedValue: AnyPublisher<T, Never> {
        self.subject.eraseToAnyPublisher()
    }

    func finish(with completion: Subscribers.Completion<Never>) {
        self.subject.send(completion: completion)
    }
}
```

Listing 4.5: *Simplified except of the* `Output` *property wrapper from* `Node.swift`. *It provides a convenient and clean way to create publishers for distributing data via the Combine Framework.*

In order that each newly calculated `UIBezierPath` can be received by subsequent nodes, the `contour` property is annotated with `@Output`. This assigns the `Output` property wrapper to the `contour` property. Property wrappers are a Swift syntax feature, which implements an abstraction layer for property storage management and also provides *Property Observers* [35, 34]. In this case, the implementation of the `Output` property wrapper provides a *Combine* publisher to subscribe to any value assigned to the annotated property. This is achieved by the property wrapper holding a *Combine* `PassThroughSubject` (cf. Listing 4.5), which provides a publisher via the wrappers `projectedValue` property. The value of a `projectedValue` is accessible at the class instance holding the annotated property by prepending `$` to the annotated property name. With respect to the `ContourNode`, this means that a publisher, which publishes data of type `Frame<UIBezierPath>`, is accessible via `$contour` at a `ContourNode` instance. Therefore, subsequent nodes, that subscribe to the contour publisher via the `$contour` property, receive value updates each time the `process(input:)` method is called.

By combining class inheritance, type aliases, and property wrappers, the implementation of communication between nodes is abstracted to the point where no knowledge about the internal workings of the framework is required for implementing new processing nodes. Thus, developers can fully focus their attention on the development of algorithms for data processing.

Effect

By linking multiple processing nodes, the effect designer is able to create effects. This is also possible without any further knowledge about the functionality of the framework.

For this purpose, the effect designer creates a new effect class that inherits from `RenderEffect`, as can be seen in the example of the `OutlineEffect` (cf. Listing 4.6). By overriding class variables, the effect is given a dedicated name, an associated color, and a minimum and maximum duration, which are relevant for providing the effect to the app user via the user interface. The processing nodes, in this case a `ContourNode`, `LineRenderNode` and a `BlendBackgroundNode`, are declared as instance properties.

Via the `buildPipeline(for:)` method, nodes get instantiated and linked together. The `input` parameter of the method, which provides a publisher, is used as a starting point and provides multidimensional video data, in this case, RGB and segmentation data. The publisher of segmentation frames is passed to the constructor of the `ContourNode`, whereby the `ContourNode` subscribes to data updates from the publisher. `UIBezierPath` frames, provided by the `ContourNode`, are in turn passed to the `LineRenderNode`, which rasterizes bezier paths into RGB frames. Its publisher is further chained together with the publisher of the original RGB input into the `BlendBackgroundNode`, which blends both image data into one image frame. The resulting publisher of the `BlendBackgroundNode` `$blendedFrame` is finally returned by the method. In this way, the data flow between nodes is statically defined, while the actual data flow while processing is handled by the *Combine* framework.

Similar to the implementation of the `SingleFrameNode`, the `RenderEffect` uses a combination of inheritance and type aliases to hide the internal framework implementations and therefore makes it easy for the effect designer to create new effects. An `Effect<Input, Output>` class provides the basic effect interfaces. It is extended by the `TimedEffect<Input: TimedData, Output: TimedData>` class, which allows an effect to be applied to a temporal section of a video, specified via its `timeRange` property.

For the creation of silhouette-based video effects, in this work RGB, depth, and segmentation data, are used as input data for the effects. Therefore, the type alias `OptionalFrameSet` defines a named touple of types `Rgb`, `Depth` and `Segmentation` (cf. Listing 4.7). As the iPad Pro sensors may not provide segmentation and depth data for each captured frame, these values are optional. The `OptionalFrameSet` wrapped into a `Frame` is named as `RenderInput` via a type alias. Since the desired result of the app is a stylized RGB video, the `RenderOutput` type alias refers to a `Frame` containing RGB data. Accordingly, in this work a `RenderEffect` is a silhouette-based effect, applied to a certain section of a video, which receives multidimensional video frames (`RenderInput`) as input and returns RGB frames (`RenderOutput`).

In summary, `RenderEffect` instances essentially consist of a set of node instances, which are linked using the `buildPipeline(for:)` method, and a `timeRange` property that specifies the section of a video the effect should be applied to.

```swift
final class OutlineEffect: RenderEffect {

    override class var name: String { "Contour" }
    override class var color: Color { .blue }
    override class var allowedDurations: (Double, Double) {
        (1.0, .infinity)
    }

    private var contourNode: ContourNode!
    private var lineNode: LineRenderNode!
    private var blendNode: BlendBackgroundNode!

    override func buildPipeline<Upstream: Publisher>
        (for input: Upstream) -> AnyPublisher<RenderOutput, Never>
        where Upstream.Output == RenderInput,
        Upstream.Failure == Never {

        self.contourNode = ContourNode(rgb: input.segmentation)
        self.lineNode = LineRenderNode(
            path: self.contourNode.$contour,
            frameExtent: input.frameExtent
        )
        self.blendNode = BlendBackgroundNode(
            foreground: self.lineNode.$lineFrame,
            background: input.rgb
        )

        return self.blendNode.$blendedFrame
    }
}
```

Listing 4.6: *Simplified except of the* `OutlineEffect` *class from* `OutlineEffect.swift`*. It combines multiple processing nodes for creating a silhouette-based video effect from multidimensional data input.*

```
typealias OptionalFrameSet = (
    rgb: Rgb, depth: Depth?, segmentation: Segmentation?
)
typealias RenderInput = Frame<OptionalFrameSet>
typealias RenderOutput = Frame<Rgb>
typealias RenderEffect = TimedEffect<RenderInput, RenderOutput>
```

Listing 4.7: *By the definition of type aliases, the allowed data types for the input and output publishers of an* `TimedEffect<Input: TimedData, Output: TimedData>` *class get specified.*

Pipeline

If a user creates effects for different sections of a video via the user interface, an `RenderEffect` instance is created for each effect and stored in one instance of the `Pipeline` class. While effects are applied to certain sections of a multidimensional video, the *Pipeline* instance traverses every input frame of the entire video and is responsible for selecting the correct effect by its respective `timeRange` property.

Listing 4.8 shows a simplified implementation of the pipeline class. The class property `effects` holds all created effect instances as an array. Whenever the `effects` array changes, e.g. effects are added or removed, the `buildPipeline()` of the `Pipeline` method is called via the `didSet` property observer. Within the `buildPipeline()` method, all effects are connected to the pipeline via a single *Combine* subscription.

For this, the *manualDownstream* mechanism of the base `Effect` class (cf. Listing 4.9) is used: Frames with multidimensional data are sent to effects using its `send(_:)` method. The data is received by the `subject` property, which is a `PassthroughSubject` object. `PassthroughSubject`s are a special kind of *Combine* publisher whose `send()` method can be used to publish data to already associated subscribers. This publisher is used in the `buildPipeline(for:)` method as the initial publisher for the chain of processing nodes. The publisher returned by the method is in turn stored by the computed property `manualDownstream`.

Back at the `Pipeline`, `manualDownstream` publishers of all effects are collected and, together with an `fallbackSubject`, merged into a single `subscription` property using the `Publishers.MergeMany()` method. This subscription receives every published frame from any of the merged publishers. The `fallbackSubject` is added to the list of publishers as a kind of "passthrough effect" for publishing unchanged RGB frames at temporal video sections, where no effect is to be applied.

Multidimensional data inside a `Frame` wrapper can be fed into the pipeline using the `processAsync(input:fallback:)` method. The frame is sent to the first effect whose *timeRange* contains the *timestamp* of the *Frame* instance. This leads to the processing

```swift
open class Pipeline<Input: TimedData, Output: TimedData>
    where Output: Finishable {

    final var effects = [TimedEffect<Input, Output>]() {
        didSet { buildPipeline() }
    }

    private final let fallbackSubject
        = PassthroughSubject<Output, Never>()
    private final var subscription: AnyCancellable?

    final func buildPipeline() {
        var publishers = effects.map { $0.manualDownstream }
        publishers.append(fallbackSubject.eraseToAnyPublisher())

        subscription = Publishers.MergeMany(publishers)
            .sink { result in
                result.finish()
            }
    }

    final func processAsync(input: Input, fallback: Output) {
        if let effect = effects.first(
            where: { $0.shouldApplyOn(input) }
        ) {
            effect.send(input)
        } else {
            fallbackSubject.send(fallback)
        }
    }
}
```

Listing 4.8: *Simplified except of the* `Pipeline` *class from* `Pipeline.swift`*. It receives incoming data via the* `processAsync(input:fallback:)` *method and routes it to the first effect whose* `timeRange` *property matches the* `timestamp` *of incoming data.*

```
open class Effect<Input, Output> {
    // ...

    private final let subject = PassthroughSubject<Input, Never>()
    func send(_ input: Input) { subject.send(input) }
    lazy private(set) var manualDownstream: AnyPublisher<Output, Never>
        = { self.buildPipeline(for: self.subject) }()

    // ...
}
```

Listing 4.9: *Excerpt from the* `Effect` *base class in* `Effect.swift` *showing the manualDownstream mechanism. By a combination of* `PassthroughSubject` *and the* `manualDownstream` *computed property, single data frames can be pushed to an* `Effect` *via its* `send(_:)` *method.*

of the frame using the selected effect, which sends the resulting RGB frame to the `subscription` property of the `Pipeline`. If no effect is to be applied for the given `timestamp` of the `Frame`, the provided `fallback` frame is send instead. The RGB frames arriving at the `subscription` are finally used for showing a video preview to the user or for writing the RGB frames to a video file.

4.3. Interactive Parameter Adjustment

A special feature of the app is the ability to adjust effect parameters during processing. This allows the user to visually evaluate effect adjustments immediately. The implementation of user-adjustable parameters is also designed to make it easy for the effect designer to provide parameters to the user.

The implementation of the user-adjustable parameters consists of two layers of abstraction:

1. Node-level processing parameters are made accessible through annotation with the `NodeParameter` property wrapper.

2. Effect-level processing parameters, annotated with a subclass of the `Viewable Parameter` class, namely `SwitchValue`, `SliderValue`, `PickerValue` or `ColorValue`, connect to the parameters on node-level and automatically create the corresponding user interface elements for user interaction.

Listing 4.10 shows the implementation of a node for applying a Gaussian blur to RGB frames. The `sigma` property controls the intensity of the applied blur. By annotating `sigma` with the `NodeParameter` property wrapper, the value of `sigma` can be changed via

```swift
final class BlurNode: SingleFrameNode<Rgb> {

    @Output var blurredFrame: Frame<Rgb>

    @NodeParameter var sigma: Double = 10

    init<RgbFrame: Publisher>(rgb: RgbFrame, sigma: Double)
        where RgbFrame.Output == Frame<Rgb>, RgbFrame.Failure == Never {
        self._sigma = NodeParameter(wrappedValue: sigma)
        super.init(input: rgb)
    }

    override func process(input rgb: Frame<CIImage>) {
        let blurred = rgb.value
            .applyingGaussianBlur(sigma: sigma)
            .cropped(to: rgb.extent)
        blurredFrame = rgb.adapt(for: blurred)
    }
}
```

Listing 4.10: *Simplified excerpt of the* `BlurNode` *from* `BlurNode.swift`*. It applies a gaussian blur via its* `process(input:)` *method. The resulting blurred frames can be subscribed via the annotated* `blurredFrame` *property. The blur intensity is controlled by the* `sigma` *property, which either provide the assigned static value of* `10` *or an externally controlled value, assigned via a binding at the* `NodeParameter` *property wrapper.*

```
1  @propertyWrapper open class NodeParameter<T> {
2
3      private final var defaultValue: T
4
5      open var wrappedValue: T {
6          get { self.projectedValue?.wrappedValue ?? self.defaultValue }
7          set {
8              self.projectedValue?.wrappedValue = newValue
9              self.defaultValue = newValue
10         }
11     }
12
13     open var projectedValue: Binding<T>?
14
15     public init(wrappedValue: T) {
16         self.defaultValue = wrappedValue
17     }
18 }
```

Listing 4.11: *Simplified excerpt of the* `NodeParameter` *property wrapper from* `Node.swift`*. It is used to annotate parameter properties at the node level and enable the change of parameter values via the user interface. If a binding is assigned at the* `projectedValue` *property, the data provided via the binding is used. Otherwise the* `defaultValue`*, which was assigned to the annotated property at initiation, is returned as a* `wrappedValue`*.*

the user interface. This is achieved by the property wrapper holding an optional `Binding` (cf. Listing 4.11), which is a SwiftUI object type that enables the creation of a reactive two-way connection to a data source stored "elsewhere" [7]. If a binding is assigned to the *projectedValue* (`$sigma`), the Gaussian blur will be applied using the `sigma` value provided via the binding. The value initially assigned to the node property (cf. Listing 4.10, line 5) is stored by the property wrapper as `defaultValue` (cf. Listing 4.11, line 3) and is used for processing if no binding is assigned.

To publish parameters in the user interface, the effect designer creates corresponding properties on the effect class, in which the nodes are used. These effect-level properties are annotated with one of the property wrappers inheriting from `ViewableParameter`.

Each property wrapper is compatible with certain data types and provides the user with suitable interaction elements in the user interface: `SwitchValue` is used for boolean values and displays buttons to toggle between two states (cf. Figure 3.16(a)). The `SliderValue` property wrapper is used for floating point numbers and is displayed to the user as a slider interface (cf. Figure 3.16(b)). Integers and enumerations can be selected by the `PickerValue` and are displayed as a list of buttons (cf. Figure 3.16(c)). Finally, `ColorValue` allows to adjust color properties by a displayed color slider (cf.

Figure 3.16(d)).

Listing 4.12 shows an example for the blur effect, at which a `SliderValue` is used to enable the user to adjust the `sigma` node parameter. In line 9 to 13, a property of type `Double` is initiated with a default value of `8`. It is annotated with a `SliderValue` property wrapper, to which additional information is assigned:

- The effect stage this effect parameter applies to (Main Stage, cf. subsection 3.1.2).
- A descriptive name to display to the user.
- In case of the `SliderValue`, value ranges in which the user is able to adjust the annotated parameter.

For connecting an effect parameter with a node parameter, each of `SwitchValue`, `SliderValue`, `PickerValue` and `ColorValue` provide a binding as `projectedValue`. In line 21, the binding of the `SliderValue` is assigned to the `NodeParameter`. Thus, any change in the value of the effect parameter directly affects the linked node parameter and will be used by the next call of the nodes `process(input:)` method.

By distinguishing between node and effect parameters, nodes can be implemented to provide any number of parameters from which the effect designer can choose when creating effects. Default values of a node parameter can be overwritten with a value statically assigned at effect level, which helps the effect designer tune an effect to the desired appearance. Furthermore, several node parameters can be linked to one effect parameter, directly or via a value mapping, so that the user controls several node parameters by one interaction element (cf. meta-parameters, subsection 3.3.4).

The presentation of interactive elements within the user interface is enabled by the implementation of the `ViewableParameter` class. It holds the current parameter value as a source of truth and provides the bindings to node parameters. It further implements the user interface elements as SwiftUI views and manages the propagation of values between node parameters and the user interface.

Therefore, `ViewableParameter` implements the `Viewable` protocol (cf. Figure 4.2). It enables the building of a SwiftUI view for each annotated effect parameter, which can be stored as a list of `AnyIterableViews` within an effect instance. For each created effect instance, the interaction element views are presented in the Parameter View (cf. subsection 3.3.4), grouped by the assigned effect stage.

```swift
final class BlurEffect: RenderEffect {

    override class var name: String { "Blur" }
    override class var color: Color { .yellow }
    override class var allowedDurations: (Double, Double) { (1.5, 3.5) }

    private var blurNode: BlurNode!

    @SliderValue(
        stage: .main,
        name: "Blur Radius",
        range: 0...20
    ) var sigma: Double = 8

    override func buildPipeline<Upstream: Publisher>
        (for input: Upstream) -> AnyPublisher<RenderOutput, Never>
        where Upstream.Output == RenderInput,
        Upstream.Failure == Never {

        self.blurNode = BlurNode(rgb: input.rgb)
        self.blurNode.$sigma = self.$sigma
        return self.blurNode.$blurredFrame.eraseToAnyPublisher()
    }
}
```

Listing 4.12: *Simplified exerpt of the* `BlurEffect` *from* `BlurEffect.swift`*. It defines* `name`*,* `color` *to be displayed in the user interface. Further, an* `allowedDurations` *defines the minimum and maximum duration of the effect. The effect consists of a single* `BlurNode`*, which blurs an RGB video frame. The blur intensity is defined by the property* `sigma`*. It is annotated with the* `Slider Value` *property wrapper, which automatically displays a slider interface within the Parameter View of the effect. The* `sigma` *parameter is identified with the set* `name` *property on the property wrapper and changes via the user interface are restricted to a value range between* `0` *and* `20`*. It further defines the effect stage to which the parameter is applied.*

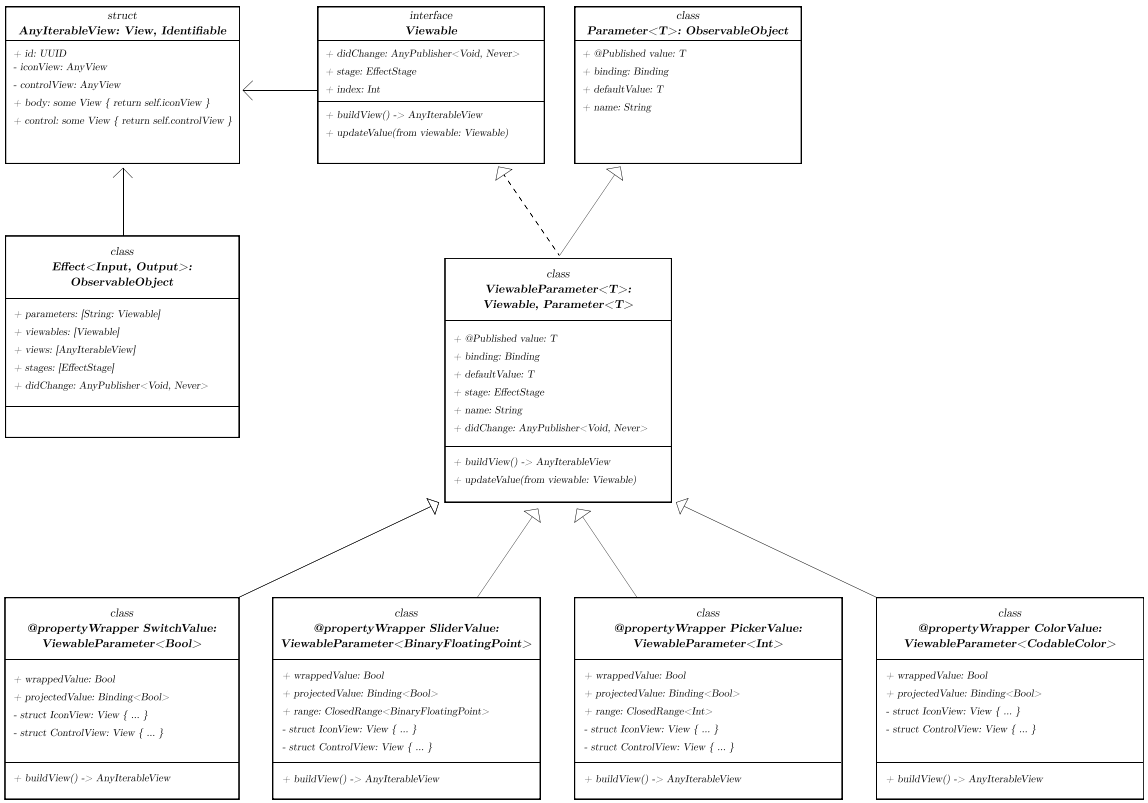


Figure 4.2.: *Overview of the classes used for providing interactive parameter adjustment. The property wrappers* `SwitchValue`, `SliderValue`, `PickerValue` *and* `ColorValue`*, which hold the current parameter value and provide a SwiftUI view for the user interface, inherit from the generic* `ViewableParameter` *class. The* `Viewable` *protocol ensures that SwiftUI views of type* `AnyIterableView` *can be created for* `ViewableParameter` *instances. The* `Parameter` *parent class implements the reactive data storage mechanism of the* `value` *property, which allows propagating value changes from the user interface up to the node parameters. While the* `ViewableParameter` *subclasses annotate parameter properties on effect-level, an additional* `views` *array holds the user interface views, provided by the* `buildView()` *method, to be displayed to the user.*

The `Parameter<T>` parent class of `ViewableParameter` is responsible for storing and forwarding parameter values. It implements the `ObservableObject` protocol to be used as data source for SwiftUI views.

As shown by the `SwitchValue` example in Listing 4.13, the `ViewableParameter` subclasses define the respective SwiftUI views. The `IconView` provides a preview view for selecting a particular parameter from a list of all effect parameters. The `ControlView` displays the respective interaction element, in this case, buttons to switch the boolean state. By the use of the `buildView()` method, `IconView` and `ControlView` are packaged into an `AnyIterableView` instance. The `AnyIterableViews` of all effect parameters are created using the `buildView()` method on effect instantiation and stored within a list in the `views` property of the effect (cf. Figure 4.2). In this way, if the parameter view is opened for an effect, the views of all parameters can be displayed dynamically.

```
@propertyWrapper final class SwitchValue: ViewableParameter<Bool> {

    var wrappedValue: Bool {
        get { value }
        set { value = newValue }
    }
    var projectedValue: Binding<Bool> { binding }

    struct IconView: View {
        @ObservedObject var parameter: SwitchValue
        var body: some View {
            ParameterSwitchIconView(
                value: $parameter.value,
                defaultValue: parameter.defaultValue,
                name: parameter.name,
                isActive: parameter.isActive
            )
        }
    }

    struct ControlView: View {
        @ObservedObject var parameter: SwitchValue
        var body: some View {
            return ParameterSwitchControlView(
                value: $parameter.value,
                defaultValue: parameter.defaultValue
            )
        }
    }

    override func buildView() -> AnyIterableView {
        AnyIterableView(
            isActive: self.isActive,
            iconView: AnyView(IconView(parameter: self)),
            controlView: AnyView(ControlView(parameter: self))
        )
    }

}
```

Listing 4.13: *Simplified excerpt of the* `SwitchValue` *property wrapper from* `Switch.swift`*. It inherits from the generic* `ViewableParameter` *class and overwrites the* `buildView()` *method, which constructs an* `AnyIterableView` *from the included* `structs IconView` *and* `ControlView`*, which define SwiftUI view layouts. The SwiftUI views contain the* `SwitchValue` *instance itself as* `@ObservedObject` *for binding its boolean value property to the user interface reactively. Via the* `projectedValue` *property of* `SwitchValue`*, a binding for the* `NodeParameter` *property wrapper is provided for binding node parameters to the* `value` *property of* `SwitchValue`*.*

Chapter 5

Discussion and Future Work

The following chapter evaluates and discusses the results of this work concerning the developed MotionViz app. Furthermore, the multidimensional video processing framework is discussed in terms of the quality of generated motion visualization effects and its processing performance using the reference hardware of an iPad Pro. In addition, the experience of developers in using and extending the framework is evaluated in form of an interview. Finally, an outlook on future work is given.

5.1. Introduction to the MotionViz App

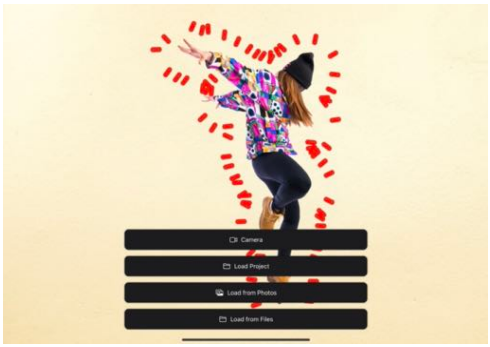
In the following, the developed MotionViz app and its functions are presented and discussed. For the evaluation, the focus is on the implementation of the concept for the MotionViz app and the processing framework concerning the formulated functional and non-functional requirements. For this, the following sections will illustrate and discuss the user journey through the app, starting with video recording (cf. subsection 5.1.1), followed by the creation of content-aware video effects (cf. subsection 5.1.2) and finally the video and project export (cf. subsection 5.1.3).

5.1.1. Capturing and Storing of Multidimensional Video Data

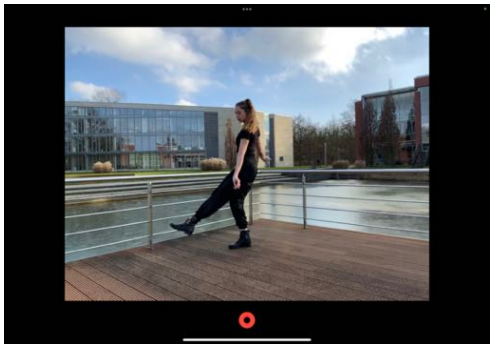
When opening the app, the user is presented with the start screen (cf. Figure 5.1(a)). From this, multidimensional videos can be recorded and already recorded videos can be loaded from the photo library or the file system. In addition, the user is able to open previously created projects, which will be explained in more detail in subsection 5.1.3.

The “Capture” button takes the user to the capture screen (cf. Figure 5.1(b)). It consists of a live preview of the iPad Pro RGB camera stream and a recording button. As required in F-01 (cf. subsection 1.2.1), by tapping the recording button a multidimensional video is captured with at least 30 frames per second. The sensor array at the back of an iPad Pro thereby enables to capture of RGB as well as depth and segmentation streams simultaneously with frame rates up to 240 frames per second, depending on hardware utilization. By default, multidimensional videos are captured at 50 frames per second.

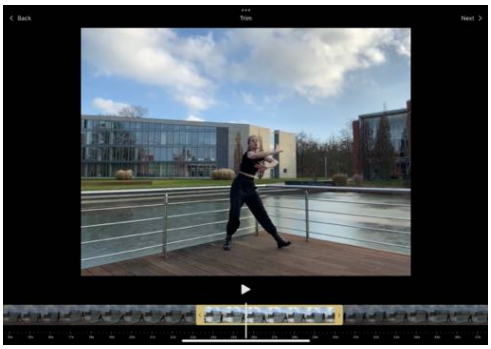
As soon as the user ends the recording by tapping the recording button once again, RGB, depth, and segmentation streams as well as an audio stream are persistently stored as a single media asset in the user’s media library (cf. F-02, subsection 1.2.1). This is achieved by storing the data streams as individual video tracks in an H.264 container.



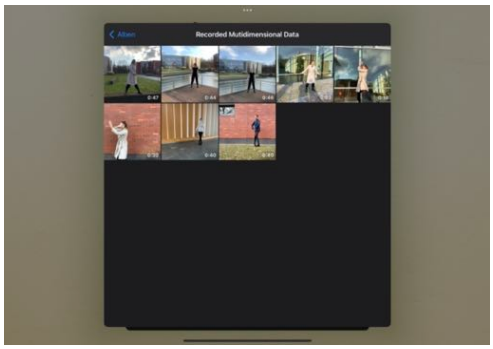
(a) *Screenshot of the start screen.*



(b) *Screenshot of the capture screen.*



(c) *Screenshot of the trim screen.*



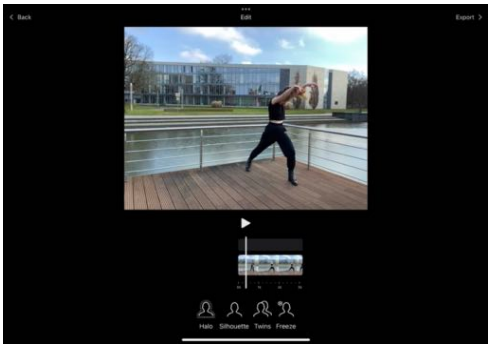
(d) *Screenshot of the loading screen.*

Figure 5.1.: *The figure shows screenshots of the developed MotionViz app, including the start screen providing navigation options to the functionalities of the app, the capture screen while recording a multidimensional video, the trim screen for trimming a captured video to the desired length as well as the loading screen for loading captured multidimensional video files.*

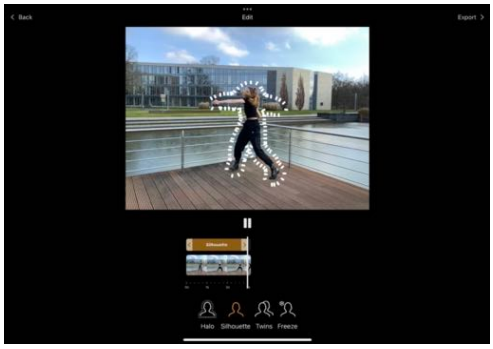
This way the resulting multidimensional video file is compatible with common media libraries and video players, which still are able to play RGB video and audio tracks, ignoring depth and segmentation data.

After stopping the recording, the user is taken to the trim screen (cf. Figure 5.1(c)). There, the RGB track can be viewed in a video preview. Optionally, via a selection in the timeline of the video player, the user is able to trim the video in length and thereby select a desired section of the video. By tapping the “Back” button current recording is discarded and the user returns to the capture screen. The “Next” button takes the user to the edit screen.

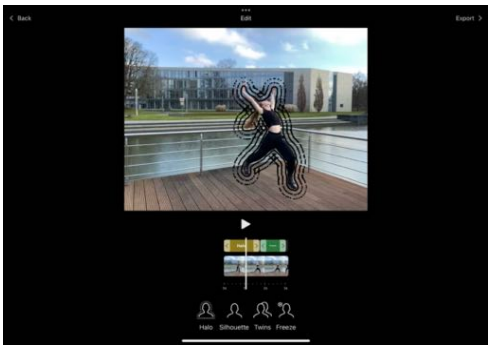
If an already saved multidimensional video is to be processed, the user has the option to load data from the media library or from the file system into the app via the buttons “Load from Photos” and “Load from Files” on the start screen. Via an opening dialog window, directories can be searched and the desired asset can be selected (cf. Figure 5.1(d)). After selecting a compatible multidimensional video file, the user is



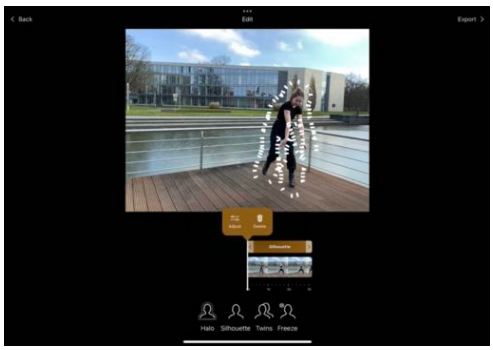
(a) *Screenshot of the edit screen.*



(b) *Effects are created by tap-and-holding the effect buttons.*



(c) *Multiple effects can be added to a timeline consecutively.*



(d) *The options for editing parameters and deleting an effect are accessible via a tooltip.*

Figure 5.2.: *The figure provides screenshots of the MotionViz apps edit screen. Figure 5.2(a) shows the edit screen with its four effect buttons and no applied effect. At Figure 5.2(b), a Silhouette Effect gets applied by tap-and-holding the Silhouette Effect button. Figure 5.2(c) shows, that multiple effects can be applied in different sections of the video. In Figure 5.2(d), by tapping an effect the tooltip is shown for adjusting effect parameters and for deleting the effect.*

returned to the trim screen to view the material and, if desired, to select a video section.

5.1.2. Creating Content-Aware Video Effects

The creation and editing of video effects is the main goal of the MotionViz app. In section 3.3, the design of the user interface has been conceptualized in order to fulfill the functional and non-functional requirements of the app. Referring to this, the implementation of the app concept is evaluated in the following.

After trimming the video, the user is taken to the edit screen shown in Figure 5.2(a). Here, the video preview can be used to find the appropriate position for the desired video effect. The creation of the video effects is intuitively possible via the effect button provided with an illustrative icon at the bottom of the screen. An effect is created by a tap-and-hold gesture on the effect button at the current video position. While the effect

is created, the result is shown within the video preview. How long the user holds the button determines the length of the video effect.

The exact position and length of an effect can be adjusted afterward by dragging the Timeline Effect Marker (cf. subsection 3.3.3). It is also possible to arrange multiple effects one after the other (cf. Figure 5.2(c)). The different colors per effect type help to quickly differentiate effect instances of different types on the timeline. However, applying multiple effects at the same time is not currently supported. A tap on the Timeline Effect Marker opens a tooltip, which offers the user an option to delete the effect as well as to refine the effect via the parameter screen (cf. Figure 5.2(d)).

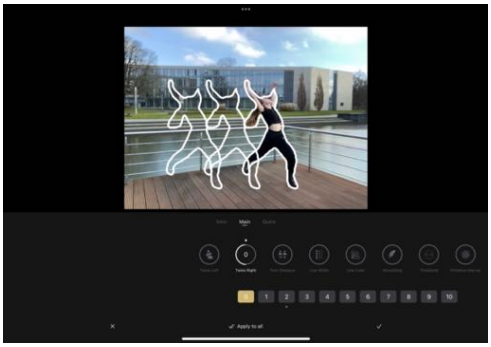
In the upper section of the parameter screen, an effect preview displays the applied effect in a continuous preview loop. The lower part of the screen consists of the effect parameters available to the user (cf. Figure 5.3). Effect parameters are divided into Intro, Main, and Outro Stages to configure intro and outro animations independently from the main part of the effect. All available parameters of a stage are listed by round symbols, the currently selected parameter is positioned horizontally centered and highlighted. Below this, the corresponding interaction element for the selected parameter is displayed. At the bottom of the screen three buttons are arranged: The left one resets the parameter changes made, the right one applies the parameters set for the selected effect. And the middle "Apply to all" button applies the parameter set to all created effects of the same type. After tapping on any of the buttons the user is returned to the edit screen.

Using the Twins Effect as an example, the screenshot Figure 5.3(a) shows the picker interface, which can be used to set the number of silhouette twins to the right of the person. The "Twin Distance" parameter, which specifies the horizontal distance between two silhouettes, can be adjusted using the slider interface (cf. Figure 5.3(b)). A color adjustment of the silhouette is possible by means of the color interface of the "Line Color" parameter, as shown in Figure 5.3(c). And by means of the switch interface of the "Chalky" parameter (cf. Figure 5.3(d)), a chalk-like structure of the silhouette line can be set or unset.

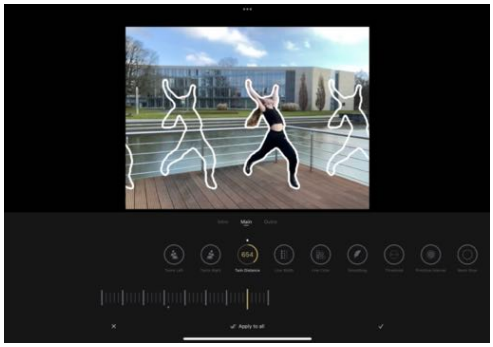
Through unstructured qualitative expert interviews with users both having technical and design backgrounds, the assumptions made in subsection 3.3.2 for an intuitive and user-friendly user interface have been confirmed. Even with no professional knowledge of video editing, testers had no difficulty using the app. Placing effects on the timeline and adjusting effects as desired was successfully executed without further instruction. The testers found the process of creating effects fun and did not encounter technical errors. (cf. NF-01 and NF-02, subsection 1.2.2)

5.1.3. Video and Project Export

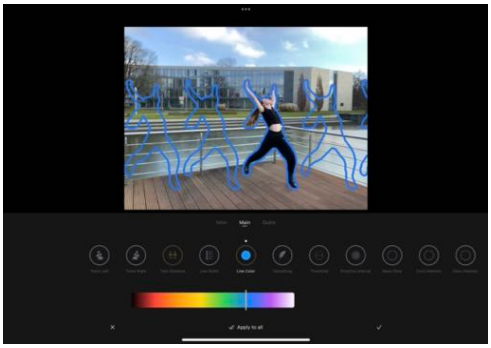
Using the "Export" button on the edit screen the user gets to the render screen, where all created effects get rendered into a single RGB video file. The screen provides the user with a preview of rendered frames as well as a progress bar for the video render progress (cf. Figure 5.4(a)). After the render process is finished, the user is able to view the resulting video and save it to the media library, and alternatively share it, e.g. on



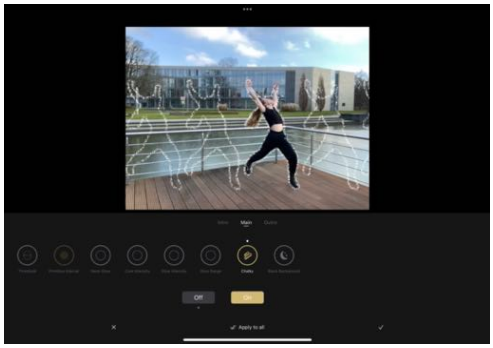
(a) *Screenshot of parameter screen with the picker interface.*



(b) *Screenshot of parameter screen with the slider interface.*

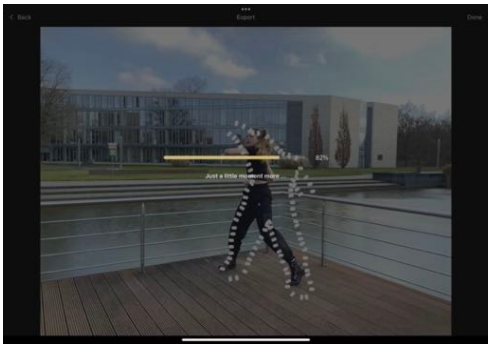


(c) *Screenshot of parameter screen with the color interface.*

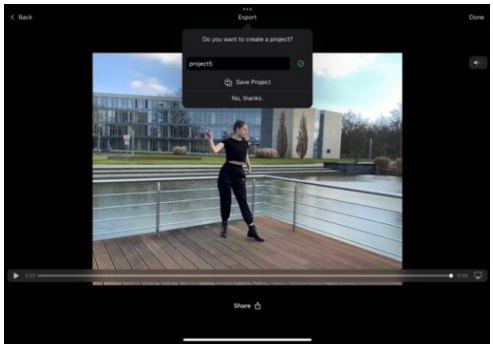


(d) *Screenshot of parameter screen with the toggle interface.*

Figure 5.3.: *The figure shows screenshots of the parameter screen of an applied Silhouette Effect. Parameters are listed for the main stage. In Figure 5.3(a), the Twins Right parameter is selected, showing a picker interface for parameter adjustment. Twin Distance can be adjusted by the use of a slider interface in Figure 5.3(b). The Line Color parameter was adjusted to blue in Figure 5.3(c) using a color interface. And in Figure 5.3(d) the Chalky mode is enabled by the use of a toggle interface.*



(a) *Screenshot of the export screen while video rendering.*



(b) *Screenshot of the export screen after video rendering.*

Figure 5.4.: *The figure shows screenshots of the export screen. In Figure 5.4(a), the progress of the current video export is shown together with the current rendered video frame. Figure 5.4(b) shows the rendered video together with the share button for saving and sharing the video. Also, by clicking the Done button the project save modal opened up to save the effects and parameter configurations.*

social media platforms.

On tapping the "Done" button, the user is offered to save the configured effects and parameter settings as a Project. This optional step serializes the multidimensional video data together with created effect instances and their user-adjusted parameter configurations. This allows the user to restore the editing status at any time via the "Load Project" button on the home screen (cf. F-05, subsection 1.2.1). As with loading multidimensional video files from the media library and from local storage, the user is presented with an overview of all stored projects (cf. Figure 5.5). By selecting one of the thumbnails, the user is taken directly to the edit screen for further editing.

Figure 5.5.: *This screenshot of the MotionViz app depicts the loading screen for loading saved projects into the edit screen.*

5.2. Evaluation of the Multidimensional Video Processing Framework

In the following, the multidimensional video processing framework is evaluated with respect to the automated creation of content-aware video effects. For this purpose, the effects implemented in the MotionViz app are compared with the video frame examples from the music video "Wait for You" by Tom Walker (cf. subsection 5.2.1). Furthermore, the performance of the framework is analyzed and the extensibility of new effects and parameters is evaluated (cf. subsection 5.2.2). Finally, the experiences of developers, which have already worked with and extended the framework, are shared (cf. subsection 5.2.3).

5.2.1. Effect Evaluation

With the MotionViz app, silhouette-based video effects are implemented as an example of content-aware video effects. Inspired by the music video "Wait for You" by Tom Walker, four effect types were defined and conceptualized in subsection 3.1.2: Halo, Silhouette, Twins, and Freeze Effect. In the following, the resulting video effects created by the effect implementations of the processing framework are compared with the effect examples from the music video.

Halo Effect

To recap, the Halo Effect is characterized by several lines of different scales around the person depicted in the video. The contour lines are based on the silhouette of the person and are displayed at different scales (cf. Table 3.1).

Comparing the video effects from the music video and the Halo Effect created by the app (cf. Figure 5.6), there is a clear similarity between the visualizations in their visual appearance. Both have multiple contour lines that outline and highlight the person in their dancing movements. A closer look reveals minor differences, which are due to implementation details as well as stylistic decisions: While the sample frame from the music video uses a jittery line style to give the contour lines a choppy look, the effect created by the MotionViz app uses a broken, partially dotted line. In addition, it can be seen that, especially on the arms, the contour lines in Figure 5.6(a) are less close to the person's body than in Figure 5.6(b). While these visual differences are discernible in a direct comparison of the displayed individual frames in Figure 5.6, they are less noticeable when comparing the associated video sequences.

The video effect in the music video is built up by the successive appearance of the individual contour lines and is ended by dissolving the contour lines into fragments of decreasing size. The Halo Effect generated by the MotionViz app replicates these animations, giving the two effects an identical aesthetic appearance and visual quality (cf. NF-04, subsection 1.2.2).

The implementation of the Halo Effect consists of three concatenated nodes: A *ContourNode* for retrieving a contour path from segmentation matte data. An *Animat-*

(a) *Video frame from the music video by Tom Walker.*

(b) *Video frame of a Halo Effect produced by the MotionViz app.*

Figure 5.6.: *The figure compares visualizations of a Halo Effect from the music video "Wait for You" by Tom Walker (Figure 5.6(a)) and the produced effect by the MotionViz app (Figure 5.6(b)).*

edHaloNode for multiplying, styling, and rendering a contour path and animating the visualization for intro and outro stages. And a *BlendBackgroundNode* for blending the stylized Halo paths onto an RGB frame.

In general, it can be stated, that since the vector path calculated in the *ContourNode* is derived directly from the segmentation matte, its accuracy is also decisive for the accuracy of the contour lines. Inaccuracies in the segmentation information, such as unrecognized limbs or incorrectly recognized frame regions, can thus lead to inaccuracies or artifacts in individual effect frames. Since, as described below, all four effects work with vector paths based on human segmentation data, this also applies to the Silhouette, Twins, and Freeze Effect.

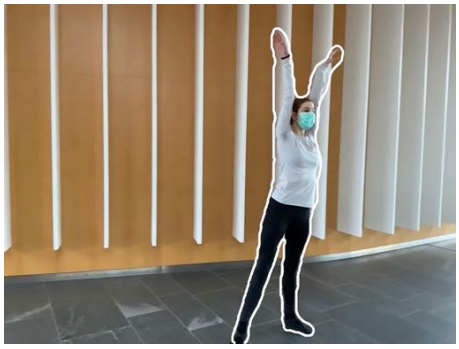
For rendering vector paths to rasterized lines, the MotionViz app implements an instanced rendering approach [33, Chapter 3], in which primitives, such as solid color shapes or image textures, are drawn at equidistant points on a contour path within a single render call. This rendering approach allows to efficiently create various stylizations of vector paths by adjusting the distance between primitives and using different textures for creating solid as well as patterned line styles. In the case of the Halo Effect, this creates both solid and partially dotted line segments. Further design possibilities of the instanced rendering approach become clear by the evaluation of the following effects.

Silhouette Effect

A single sketchy contour line that fits closely to the person characterizes the appearance of the Silhouette Effect. Comparing the video frame from the music video and the Silhouette Effect generated by the MotionViz app (cf. Figure 5.7), both motion visualizations show great similarity. Both have a sketchy stroke style and enclose the silhouette of the dancer depicted in the video. The white line color makes them stand out well from the video scene and highlights the dancer's movements.

Like the Halo Effect, the Silhouette Effect is based on segmentation data and is

(a) *Video frame from the music video by Tom Walker.*



(b) *Video frame of a Silhouette Effect produced by the MotionViz app.*

Figure 5.7.: *The figure compares visualizations of a Silhouette Effect from the music video "Wait for You" by Tom Walker (Figure 5.7(a)) and the produced effect by the MotionViz app (Figure 5.7(b)).*

composed of three chained nodes. Instead of the *AnimatedHaloNode* the *AnimatedSilhouetteNode* is used, which also uses the instanced rendering approach to render and stylize the contour path.

With regard to the animation of the generated Silhouette effect, the following can be noted: In the Intro Stage, the contour line is formed from dashes aligned at ninety degrees to the contour line, and the Outro Stage dissolves the line in reverse order of the intro stage.

Twins Effect

The Twins Effect draws not only one silhouette, but several identical silhouettes that are displayed horizontally shifted (cf. Figure 5.8). Comparing the example from the music video with the Twins Effect generated by the app, again a very similar visual appearance of the stylizations can be noticed.

The fact that in addition to the contour twins on the left and right, the silhouette around the person is also traced is a stylistic decision of the effect designer and a representation as in the original would also be technically possible.

Regarding the underlying processing graph, the effect was implemented on the basis of the Silhouette Effect. The additional *TwinsNode* duplicates the contour and sets the positions of the contours in the video frame.

A limitation of the segmentation matte-based contour algorithm is that overlapping body parts cannot be distinguished. In the original video frame, an arm located behind the body and a heel protruding behind the front foot is outlined. Upgrading the current contour algorithm to achieve a higher level of detail and to represent overlapping limbs is the subject of further research and development (cf. section 5.3).

(a) *Video frame from the music video by Tom Walker.*

(b) *Video frame of a Twins Effect produced by the MotionViz app.*

Figure 5.8.: *The figure compares visualizations of a Twins Effect from the music video "Wait for You" by Tom Walker (Figure 5.8(a)) and the produced effect by the MotionViz app (Figure 5.8(b)).*

(a) *Video frame from the music video by Tom Walker.*

(b) *Video frame of a Freeze Effect produced by the MotionViz app.*

Figure 5.9.: *The figure compares visualizations of a Freeze Effect from the music video "Wait for You" by Tom Walker (Figure 5.9(a)) and the produced effect by the MotionViz app (Figure 5.9(b)).*

Freeze Effect

The Freeze Effect is characterized by a single contour line that is unchanged over several frames, while the actual video plays on (cf. Figure 5.9). In the music video, the outline is drawn with a chalk-like textured dashed line. Instead, a dotted line style is used by the generated effect. Considering the visual appearance of the effects over several frames of a played video, the two effect versions appear quite similar. The dancer's silhouette, which remains stationary and, thus, contrasts the dancer's dance movements, clearly enhances the video visually in both cases.

The effect consists of an *AnimatedFreezeNode*, which generates, stylizes, and animates the contour line, as well as the *BlendBackgroundNode*, which blends the contour line onto the RGB frame. Since the Freeze Effect does not recalculate the contour line on the segmentation matte of each frame, the calculation is embedded inside the

AnimatedFreezeNode, where the segmentation matte of the first frame, on which the frozen path is based, is cached and reused for all successive frames of the effect.

Like the other effects, the Freeze Effect uses the instanced rendering to create the contour lines. In this case, a large point spacing was chosen to create a dotted line.

Concluding Effects Evaluation

To summarize, it can be stated that the video effects modeled after the music video by Tom Walker and implemented by the MotionViz app are visually and aesthetically close to the original and result in fluid and appealing motion visualizations. Movements of people are highlighted and the effect animations in Intro and Outro Stage create a smooth and appealing visual transition between video sections with and without effects. The artistic style of hand-drawn annotations is thereby successfully imitated.

Compared to manually creating a Silhouette Effect using a professional video editing program, the MotionViz app offers numerous advantages: First, the app's intuitive user interface enables that no prior or professional knowledge of video editing software is necessary. Even untrained users, as typically found in the field of user-generated content, can create impressive video effects and adapt them to their own preferences, which is enabled by simplified parameterization options in the user interface abstracting technical complexity. Further, when manually creating silhouette-based video effects, for each video frame each individual contour line must be drawn by hand. Because of the MotionViz app's content awareness, this laborious manual work can be automated, allowing even minute-long video effects to be created in seconds. This makes the app relevant not only for individuals but also for the professional context, as its use leads to relevant cost savings on the one hand. And on the other hand, it enables video designers to focus their attention on the art direction of effects instead of being forced into the repetitive process of drawing video annotations by hand.

5.2.2. Performance Evaluation

In the following, the performance of the multidimensional video processing framework is evaluated. For this purpose, the performance of effect rendering was measured based on a 30-second multidimensional video recorded with the MotionViz app. The multidimensional video consists of an RGB, segmentation, depth, and audio track and has a video track resolution of 1440 by 1080 pixels at a frame rate of 50 frames per second.

To evaluate the performance, a distinction is made between preview and export rendering. While export rendering creates video effects in the best possible quality by (re-)computing high-resolution segmentation data, for video preview the recorded segmentation data is used, which creates video effects faster and in lower quality. Furthermore, during video preview frames are rendered using a variable frame rate, which is automatically adjusted to the resource utilization of the device. During export, all video frames are rendered successively.

Measurements were made for each of the four effects Halo, Silhouette, Twins, and Freeze, with the effects applied to the total length of the video. For comparison, the

	Halo	Silhouette	Twins	Freeze	No Effect
Preview (Original Res.)	20.8 fps	25.8 fps	23.8 fps	32.6 fps	50.6 fps
Preview (Low Res.)	21.1 fps	26.3 fps	24.5 fps	35.9 fps	50.6 fps
Export (Original Res.)	14.3 fps	14.2 fps	13.6 fps	16.7 fps	133.2 fps
Export (Low Res.)	15.6 fps	14.4 fps	14.5 fps	18.2 fps	134.1 fps

Table 5.1.: *The table shows performance measurement results for the rendering of the video preview and video export for the four effect types Halo, Silhouette, Twins, and Freeze. For the video preview, the average adaptive frame rate during video playback was measured in frames per second (fps). The performance of the export rendering was determined based on the total duration of the rendering process and divided by the total number of video frames to get the fps metric. Measurements are based on a 30-second multidimensional video with RGB, segmentation, depth, and audio track with a resolution of 1440 by 1080 pixels per video track at 50.6 fps (Original Res.). For comparison, additional measurements were performed on the same video with a reduced resolution of 1280 by 960 pixels (Low Res.). Video effects are applied to the entire video length in each case and additional measurements without applied effects were taken for reference. All measurements were performed on an iPad Pro (11-inch, 2nd generation with A12Z GPU and 6 GB shared memory), running iPad OS 15.4.1.*

measurements were additionally performed for a reduced resolution of 1280 by 960 pixels for RGB, depth, and segmentation tracks. The measurement results are listed in Table 5.1.

Measurements show, that for the recorded video in original resolution all video effects in the preview are rendered at an average frame rate of over 20 frames per second. Thus, interactive performance can be observed during effect creation and editing. While the adaptive frame rate of the Freeze Effect is highest with an average of 32.6 frames per second, for the Halo Effect the frame rate drops to an average of 20.8 frames per second due to the computational complexity for the effect. This can be explained by the rendering of three individual contour lines per frame, while Freeze and Silhouette Effects only require one contour line per frame. The Twins Effect also consists of several contour lines, but since these are identical, they are generated only once and duplicated afterward, which in comparison results in better rendering performance.

When rendering effects for export, the complexity of the effects affects render performance less. This can be attributed to the recalculation of the high-resolution segmentation matte, which is generated for all effects equally and accounts for a large part of the processing time per frame. For all four effects, a frame rate of about 14 frames per second was measured, with the Freeze Effect being generated fastest at 16.7 fps and the Twins Effect slowest at 13.6 fps. When exporting, however, it is particularly noticeable that the video without effect was exported at 133.2 fps, which results in a render duration about 2.6 times faster than normal playback.

For video export, the measurements show that processing effects take about three to five times the effect duration. However, since video effects are typically used only

accentuated with a few seconds duration, as shown by the music video "Wait for You", and video sections without effects are rendered faster than real-time (cf. Table 5.1), there are no long waiting times for users.

To evaluate the impact of video resolutions on processing performance, the measurements for original and reduced resolutions of the multidimensional videos are compared below. The measurement values documented in Table 5.1 clearly show that, although the reduced multidimensional video only has two-thirds of the pixel resolution, the processing performance for both preview and export differ little from the values in original resolution. For the video preview, the smallest difference can be seen in the Halo Effect, where the reduced resolution only leads to a 1.4 percent faster calculation. The biggest performance jump for the preview is observed for the Freeze Effect, where the reduced resolution leads to a speedup of 10.1 percent. The previews without applied effect are displayed in the original frame rate in both resolutions. It is similar to the export, where the performance deviation ranges between 1.4 (Halo Effect) and 9.0 percent (Freeze Effect). Thus, it can be stated that the framework behaves very stable with respect to different resolutions in terms of processing performance. No significant performance degradation is to be expected for multidimensional video material captured in higher resolutions.

Besides rendering performance, the app's memory usage for preview rendering and export was measured (cf. Figure 5.10). It was found that the MotionViz app maintains a low memory load during effect preview, processing, and export. The highest measured memory usage is about 258 MB, which corresponds to 4.6 percent of the total memory available on the iPad Pro. Thus, it can be stated that the app not only offers good render performance but also generates low memory pressure. And therefore it can be expected that the app memory-wise will also work efficiently for high-resolution multidimensional videos as well as mobile devices with lower memory resources.

5.2.3. Developers' Feedback

As part of the *Algorithms for Processing Visual Media (APVM)* seminar at the Computer Graphics Chair of the Hasso Plattner Institute Potsdam, Bachelor students Rosmarie Debski and Ole Schmitt have been working on extending the MotionViz app with new, non-silhouette-based, video effects. Their experiences working with the multidimensional video processing framework and the MotionViz app, especially regarding the adaptability for new data types and use cases as well as the extension with new video effects (cf. NF-3, subsection 1.2.2), were queried by means of an interview. The full-length interview can be found in Appendix A.

Their task in the context of the seminar was "to implement a 'Glow Stick' effect inspired by a TikTok video", which looks like dancers have glow sticks attached to their bodies and filmed themselves in the dark. Secondly, a "Light Trails" effect was aimed for, which imitates "long exposure timelapse videos of dancers in the dark with lights attached to their wrists". For this purpose, the students extended the framework by processing nodes in order to extract human joint point data from RGB video frames and process them into line representations. The human joint points, extracted with the

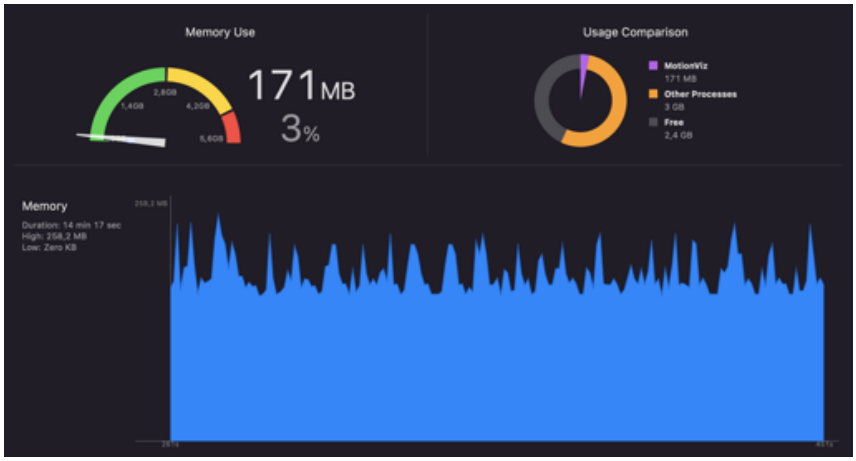


Figure 5.10.: *Screenshot of the Xcode Memory Report screen. To measure memory usage, several effects were created, previewed, their parameters adjusted, and resulting videos exported. The typical memory consumption of the MotionViz app is between about 140 and 200 MB, with memory peaks of up to 258.2 MB measured. The measurement is based on a recorded 30-second multidimensional video with RGB, segmentation, depth, and audio track (1440 by 1080 pixels, 50 frames per second) and was performed on an iPad Pro (11-inch, 2nd generation with A12Z GPU and 6 GB shared memory), running iPad OS 15.4.1.*

help of the Apple Vision framework, are thereby seen as an additional data dimension processed by the multidimensional video processing framework.

The two fourth-semester students describe themselves as not inexperienced software developers, but without experience in larger software projects. In addition, they developed for the Apple ecosystem and in the Swift programming language for the first time as part of their seminar. Nevertheless, after project setup, they were able to achieve a first naive implementation of a GlowStick effect within just eight hours. Figure 5.11 shows this preliminary version of the developed GlowStick Effect, as it was presented by the students at the interim presentation of the seminar.

They describe, that this was made possible, as they were able to reuse already implemented effects as a guide when creating new ones. In addition, they were able to reuse numerous existing processing nodes for the development of their effects, so only two new nodes had to be added for their GlowStick and LightTrails effects. In their own words they state: “Understanding the code of a given effect is easy since every effect is defined by a combination of nodes that are ‘plugged’ together. [...] This led to fast visual results.”

The mechanisms provided by the framework were found very helpful for integrating effect parameters in the app’s user interface: “We didn’t have to bother with user interface at all, since the provided standard sliders, toggles, etc. were sufficient for our needs. The usage is easy to understand and changing the user interface for an effect is done within seconds since only a few lines of code have to be changed.”

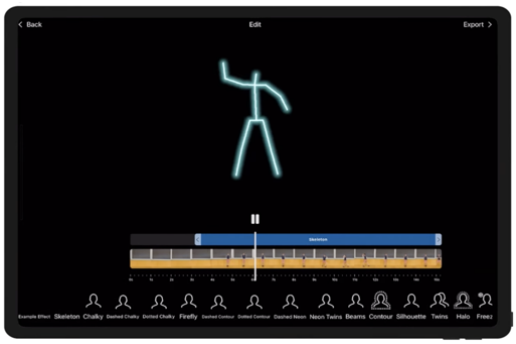


Figure 5.11.: *Shown is an excerpt from the interim presentation by Rosmarie Debski and Ole Schmitt in the seminar Algorithms for Processing Visual Media (APVM) at the Hasso Plattner Institute, in which the students extended the MotionViz app by a GlowStick Effect.*

In summary, they describe their development experience with the multidimensional video processing framework and the MotionViz app in the following terms: “It felt like we got an already safe and steady Lego house and a box full of Lego bricks to be creative.”

Rosmarie Debski and Ole Schmitt clearly demonstrate, that even developers new to the field of large software projects and inexperienced in the Apple app development ecosystem can quickly add compelling video effects to the MotionViz app by the use of the multidimensional video processing framework. Due to the predefined structure of the framework and the reusability of existing nodes, the extension of the framework for new data types as well as the development of new processing nodes and effects is easily achieved. The predefined user interfaces for effect parameterization also make it easy for developers to offer meaningful customization options for effects to the user.

5.3. Future Work

In the following, an outlook on further research and development directions regarding the improvement and extension of the multidimensional video processing framework and the MotionViz app is given. Since it has already been shown in subsection 5.2.3 that adding novel video effects to the app is efficiently possible, the following section will focus on improving the already implemented silhouette-based video effects.

Correcting Segmentation Artefacts

The quality of the silhouette-based effects (cf. subsection 3.1.1) highly depends on the quality of the underlying segmentation data. If the segmentation data contains incorrect information, such as areas that were incorrectly identified as people (false positives) or areas that depict people but were not identified as such (false negatives), it results in a direct effect on the derived contour paths and thus on the effect quality (cf. (cf. Figure 5.12, section 5.2.1). We refer to these errors as *Segmentation Artifacts*.

Figure 5.12.: *The shown video frame from a video created with the MotionViz app shows segmentation artifacts for the applied Silhouette Effect. These come from errors in the person segmentation data for this video frame. First, the dancer's arms were not segmented correctly. In addition, a bush in the background was incorrectly segmented. Due to the segmentation artifacts, the displayed contour lines do not correctly trace the dancer, and in addition, a silhouette is drawn around the bush in the background.*

Since the segmentation data is provided by Apple frameworks, it is not possible to directly influence its quality. However, it is to be expected that the segmentation quality of the frameworks will steadily improve, as has already been seen in recent years [10, 22, 27]. Nevertheless, segmentation artifacts cannot be completely excluded.

Therefore, one possible solution is to use consecutive frames for correcting such temporal discontinuities. By continuously evaluating segmentation information, e.g. with a sliding window approach (cf. [25]), short-term segmentation artifacts can be identified as outliers and corrected in the segmentation data by interpolating between previous and subsequent frames. Alternatively, such a correction process can be accomplished after the contour path is created by correcting vector points of the contour path by interpolation. Both approaches can be implemented as processing nodes and thus easily integrated into existing effects.

Displaying Overlapping Contours

Another current limitation of the segmentation data-based contour approach is that contour lines can only represent the silhouettes of people. Contour lines for arms and legs that are located in front of the body cannot be achieved in this way (cf. Figure 5.13, section 5.2.1).

One possible solution to this is to additionally evaluate depth information, which is already recorded by the MotionViz app. With sufficient depth resolution, it is thus possible to detect contour lines at depth discontinuities, which, in addition to the distinction between person and non-person, also allows finer segmentation gradations to be made, e.g. for arms and legs located within the silhouette of a person. Combining segmentation and depth information, allows more detail to be illustrated through contour lines in video

Figure 5.13.: *The displayed video frame from the music video "Wait for You" by Tom Walker shows a motion visualization that, in addition to the dancer's contour line, distinguishes other details by means of a contour line. The dancer's right forearm is traced, and further, another contour line distinguishes between the left arm and the left foot in the background.*

effects. By such a combination of information, further motion visualization effects are realizable, as they are exemplarily also represented in the music video in Figure 5.13.

Enabling Spatial Separation

During the development of the MotionViz app, particular attention was paid to the use of the app with only one person in the video. The segmentation algorithm of the Apple Vision framework basically supports the segmentation of multiple people in a video. Thereby, it is also currently possible to create video effects for multiple people, but the same effect is applied to all people within the video. However, it is currently not possible to assign different effects or effect parameterizations to individual persons in the video.

In order to offer users the possibility of assigning motion visualization effects to people independently of each other, a spatial separation for effects is necessary. And to be able to distinguish between persons whose silhouettes overlap, an approach similar to that for handling overlapping contours is appropriate.

In addition to extending the processing concept, also the concept of the user interface has also to be adapted so that users can spatially select between persons to create and adjust independent effects accordingly. One possible approach to this is that timeline effect markers (cf. Figure 3.15(a)) can be set in multiple independent effect tracks, which are arranged one below the other in the timeline. Each track can be assigned to one spatial dimension, i.e. one person in the video. This kind of usage concept with multitrack effect editing is known in the field of professional video processing. However, in order to maintain the intuitiveness of the app also for non-professional users, appropriate user studies should be conducted for such an adaptation of the usage concept.

Video Stylization and Concurrent Effects

The computer-assisted artistic stylization of images and videos is an essential part of current research in computer graphics. Using neural style transfer (NST), for example, drawing or painting styles from artworks can be transferred to captured images and recorded videos. [36, 9, 49]

It, therefore, stands to reason to enable users, in addition to motion visualization effects, to stylize the RGB video with such state-of-the-art video stylization techniques. Depending on the style of the video filters (e.g. comic, oil painting, line drawing), the motion visualization effects parameters could be adjusted to best fit into the aesthetics of the stylized video.

In order to be able to offer such stylization effects in the MotionViz app, an extension of the usage concept is necessary. The approach of multiple effect tracks described in the previous section is also a possible way of implementation here so that users can intuitively apply video stylization filters in the same way as motion visualization effects to sections or even the entire video.

Chapter 6

Conclusion

This thesis presents the design and implementation of a framework for the automatic creation of content-aware video effects. Based on the proposed framework, a mobile app for creating video effects in an automated and user-friendly manner has been designed and implemented in the context of this thesis. The proposed framework and app are capable to generate video effects that are stylistically inspired by motion visualizations from the music video "Wait for You" by Tom Walker.

By processing multidimensional video data, which extends RGB video data by additional data dimensions (e.g. depth or segmentation information), the app enables the automated creation of silhouette-based motion visualizations. This automated effect creation saves video animators the high effort of manual video annotation. An intuitive graphical user interface also opens up video annotation to a non-professional user group, benefiting content creators on user-generated content platforms. Thus, the MotionViz app supports motion designers in their professional work and also fulfills a demand for fast and intuitive video stylization apps.

Currently, there are no software frameworks available that allow for an effective and efficient extension of RGB video data by additional data such as depth or segmentation information. Further, most frameworks do not support the simultaneous processing of multiple data streams for creating content-aware video effects. To cope with these challenges, a software framework has been developed that processes multidimensional data on a frame-by-frame basis. To hide implementation complexity, processing functionality is encapsulated in processing nodes that are organized in a graph-based structure to form processing pipelines.

The framework is implemented in the Swift programming language for mobile devices. Data flow within the processing graph is realized as an asynchronous event pipeline using the Apple Combine framework. This allows efficient implementation of both traditional and asynchronous processing flows, as commonly used for machine learning and GPU-accelerated tasks. The processing algorithms are implemented using Apple frameworks for computer vision and computer graphics as well as custom-developed processing algorithms and allow processing with interactive performance. A major focus of the framework is on easy extensibility, to enable developers to include additional data types and video effects, which was achieved through a clear project structure and strict use of object orientation as well as modern syntax features of the Swift programming language.

The conceptional flexibility and extensibility of the framework were confirmed when implementing silhouette-based video effects by the MotionViz app. By encapsulating processing steps in nodes, processing functionality can be reused for multiple effects and new data types are introduced into the processing graph with ease. Even inexperienced developers are able to create novel, visually appealing video effects in a very short time. Bachelor students of the Hasso Plattner Institute, who extended the framework for a seminar, described it as follows: “It felt like we got an already safe and steady Lego house and a box full of Lego bricks to be creative.” (cf. Appendix A)

To also enable even inexperienced end users to creatively use the framework, the MotionViz app was developed. It extends the framework with an intuitive and well-integrated graphical user interface while hiding the technical complexity of multidimensional data processing. With it, users are able to record multidimensional video with RGB, segmentation, and depth data and stylize recorded videos with four provided silhouette-based video effects: Halo, Silhouette, Twins, and Freeze. The app is operated with intuitive and familiar touch gestures, with which video effects can be created using effect presets predefined by an effect designer. Created effects can be individualized in many ways by utilizing parameterization options via the user interface. User-created projects, which include both the input video, created effects, and adjusted effect parameters, can be saved and recalled at a later time. The export function allows projects to be rendered as RGB videos for sharing, for example on social media platforms.

The predefined framework structure makes it easy for effects developers to add video effects. A software interface, which enables the automatic display of graphical user interface elements for effect parameters, relieves the developer of the task of linking developed effects to the user interface and also ensures a uniform interface for the user. Further, the automated timeline-based creation of video effects enables intuitive and time-efficient video stylization for end users, even without professional knowledge in video editing.

A quantitative and qualitative evaluation of the framework and app has shown that the video effects provided are of high visual quality and can be created by users with interactive performance. Nevertheless, there are several future directions to improve the concept and the implementation: Inaccuracies in contour lines could be addressed by improving segmentation information. The effects can be visually enhanced by mapping a higher level of detail with contour lines. And spatial separation can give users new stylization options for videos that feature multiple people. Offering additional video filters provides the user with even more exciting video stylization options beyond motion visualization effects.

Already today, the framework and the app are used by students at the Hasso Plattner Institute in seminars, and teaching activities as well as for research projects. In addition, the scientific relevance of this work has been confirmed by the acceptance of the publication “MotionViz: Artistic Visualization of Human Motion on Mobile Devices” [30] at SIGGRAPH Appy Hour 2021, where the app was presented to a professional audience.

In general, the MotionViz app has shown that processing multidimensional video data opens up many exciting new possibilities for the design of video stylizations. It is

expected that new sensor technology and increasing computing power will make the field of multidimensional video processing even more relevant in the coming years. Therefore, the multidimensional video processing framework is able to provide a good foundation for further teaching and research projects in this area.

Bibliography

[1] *Adobe Premiere Rush:Edit Video*. https://apps.apple.com/us/app/adobe-premiere-rush-edit-video/id1188753863. [Online; accessed 2022-07-28]. Adobe Inc.

[2] *Apple unveils new iPad Pro with breakthrough LiDAR Scanner and brings trackpad support to iPadOS*. https://www.apple.com/newsroom/2020/03/apple-unveils-new-ipad-pro-with-lidar-scanner-and-trackpad-support-in-ipados/. [Online; accessed 2022-08-02]. Apple Inc.

[3] *AVFoundation: Work with audiovisual assets, control device cameras, process audio, and configure system audio interactions*. https://developer.apple.com/documentation/AVFoundation. [Online; accessed 2022-05-30]. Apple Inc.

[4] *AVKit: Create user interfaces for media playback, complete with transport controls, chapter navigation, picture-in-picture support, and display of subtitles and closed captions*. https://developer.apple.com/documentation/avkit. [Online; accessed 2022-05-30]. Apple Inc.

[5] Klaus B Bærentsen. "Intuitive user interfaces". In: *Scandinavian Journal of Information Systems* 12.1 (2000), p. 4.

[6] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013). ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[7] *Binding: A property wrapper type that can read and write a value owned by a source of truth*. https://developer.apple.com/documentation/swiftui/binding. [Online; accessed 2022-06-07]. Apple Inc.

[8] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics* -. Prentice-Hall, 1996. ISBN: 978-0-130-73487-7.

[9] Benito Buchheim, Max Reimann, Sebastian Pasewaldt, Jürgen Döllner, and Matthias Trapp. "StyleTune: Interactive Style Transfer Enhancement on Mobile Devices". In: *ACM SIGGRAPH 2021 Appy Hour*. SIGGRAPH '21. Association for Computing Machinery, 2021. ISBN: 9781450383585. DOI: 10.1145/3450415.3464400.

[10] Quan Chen, Tiezheng Ge, Yanyu Xu, Zhiqiang Zhang, Xinxin Yang, and Kun Gai. "Semantic Human Matting". In: *Proceedings of the 26th ACM International Conference on Multimedia*. MM '18. Association for Computing Machinery, 2018, pp. 618–626. ISBN: 9781450356657. DOI: 10.1145/3240508.3240610.

[11] *CLIP2COMIC: Transform your photos and videos into cartoon and sketches*. https://www.digitalmasterpieces.com/clip2comic/. [Online; accessed 2022-07-28]. Digital Masterpieces GmbH.

[12] *Combine: Customize handling of asynchronous events by combining event-processing operators*. https://developer.apple.com/documentation/combine. [Online; accessed 2022-05-30]. Apple Inc.

[13] *Core Image: Use built-in or custom filters to process still and video images*. https://developer.apple.com/documentation/coreimage. [Online; accessed 2022-05-30]. Apple Inc.

[14] Brian Curtin. "Semiotics and visual representation". In: *Semantic Scholar* (2009).

[15] James E Cutting. "Representing Motion in a Static Image: Constraints and Parallels in Art, Science, and Popular Culture". In: *Perception* 31.10 (Oct. 2002), pp. 1165–1193. DOI: 10.1068/p3318.

[16] *Generics*. https://docs.swift.org/swift-book/LanguageGuide/Generics.html. [Online; accessed 2022-05-30]. Apple Inc.

[17] A. Girgensohn, J. Boreczky, and L. Wilcox. "Keyframe-based user interfaces for digital video". In: *Computer* 34.9 (2001), pp. 61–67. DOI: 10.1109/2.947093.

[18] Digital Masterpieces GmbH. *CoreImageExtensions: Useful extensions for Apple's Core Image framework*. https://github.com/DigitalMasterpieces/CoreImageExtensions. [Online; accessed 2022-05-30].

[19] Simon Hawe, Ulrich Kirchmaier, and Klaus Diepold. "MutanT: A modular and generic tool for multi-sensor data processing". In: *2009 12th International Conference on Information Fusion*. 2009, pp. 1304–1309.

[20] Tobias Isenberg. "Interactive NPAR: What Type of Tools Should We Create?" In: *Proc. NPAR*. Eurographics Association, 2016, pp. 89–96. DOI: http://dx.doi.org/10.2312/exp.20161067.

[21] R. A. Jarvis. "A Perspective on Range Finding Techniques for Computer Vision". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-5.2 (1983), pp. 122–139. DOI: 10.1109/TPAMI.1983.4767365.

[22] Xiaoyan Jiang, Yongbin Gao, Zhijun Fang, Peng Wang, and Bo Huang. "An End-to-End Human Segmentation by Region Proposed Fully Convolutional Network". In: *IEEE Access* 7 (2019), pp. 16395–16405. DOI: 10.1109/ACCESS.2019.2892973.

[23] Zhanghan Ke, Jiayu Sun, Kaican Li, Qiong Yan, and Rynson W. H. Lau. *MODNet: Real-Time Trimap-Free Portrait Matting via Objective Decomposition*. 2020. DOI: 10.48550/ARXIV.2011.11961.

[24] Mandy Klingbeil, Sebastian Pasewaldt, Amir Semmo, and Jürgen Döllner. "Challenges in User Experience Design of Image Filtering Apps". In: *SIGGRAPH Asia 2017 Mobile Graphics & Interactive Applications*. SA '17. Association for Computing Machinery, 2017. ISBN: 9781450354103. DOI: 10.1145/3132787.3132803.

[25] V. Kober. "Robust and efficient algorithm of image enhancement". In: *IEEE Transactions on Consumer Electronics* 52.2 (2006), pp. 655–659. DOI: 10.1109/TCE.2006.1649693.

[26] H. T. Kung. "Synchronized and asynchronous parallel algorithms for multiprocessors". In: *Algorithms and Complexity: New Directions and Recent Results*. Academic Press Rapid Manuscript Reproduction. Academic Press, June 1976. ISBN: 9780126975406.

[27] Bo Liu, Haipeng Jing, Guangzhi Qu, and Hans W. Guesgen. "Cascaded Segmented Matting Network for Human Matting". In: *IEEE Access* 9 (2021), pp. 157182–157191. DOI: 10.1109/ACCESS.2021.3125356.

[28] Hanna Lustig. *TikTok star Charli D'Amelio debuted the 'distance dance' to drive donations during the COVID-19 pandemic*. https://www.insider.com/charli-damelio-distance-dance-on-tik-tok-to-drive-donations-2020-3. [Online; accessed 2022-05-14].

[29] Roy Marmelstein. *Zip: Zip and unzip files in Swift*. https://github.com/marmelroy/zip. [Online; accessed 2022-05-30].

[30] Maximilian Mayer, Philipp Trenz, Sebastian Pasewaldt, Mandy Klingbeil, Jürgen Döllner, Matthias Trapp, and Amir Semmo. "MotionViz: Artistic Visualization of Human Motion on Mobile Devices". In: *ACM SIGGRAPH 2021 Appy Hour*. SIGGRAPH '21. Association for Computing Machinery, 2021. ISBN: 9781450383585. DOI: 10.1145/3450415.3464398.

[31] *MetalKit: Build Metal apps quicker and easier using a common set of utility classes*. https://developer.apple.com/documentation/metalkit. [Online; accessed 2022-05-30]. Apple Inc.

[32] Sebastian Pasewaldt, Amir Semmo, Jürgen Döllner, and Frank Schlegel. "Becasso: Artistic Image Processing and Editing on Mobile Devices". In: *SIGGRAPH ASIA 2016 Mobile Graphics & Interactive Applications*. SA '16. Association for Computing Machinery, 2016. ISBN: 9781450345514. DOI: 10.1145/2999508.2999518.

[33] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005. ISBN: 0321335597.

[34] *Properties: Property Observers*. https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID262. [Online; accessed 2022-06-02]. Apple Inc.

[35] *Properties: Property Wrappers*. https://docs.swift.org/swift-book/LanguageGuide/Properties.html#ID617. [Online; accessed 2022-06-02]. Apple Inc.

[36] Max Reimann, Mandy Klingbeil, Sebastian Pasewaldt, Amir Semmo, Matthias Trapp, and Jürgen Döllner. "MaeSTrO: A Mobile App for Style Transfer Orchestration Using Neural Networks". In: *2018 International Conference on Cyberworlds (CW)*. 2018, pp. 9–16. DOI: 10.1109/CW.2018.00016.

[37] *Relive, edit, and share your picture-perfect moments.* https://www.apple.com/ios/photos/. [Online; accessed 2022-07-28]. Apple Inc.

[38] R. Rosales and S. Sclaroff. "Inferring body pose without tracking body parts". In: *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*. Vol. 2. 2000, 721–727 vol.2. DOI: 10.1109/CVPR.2000.854946.

[39] Amir Semmo, Max Reimann, Mandy Klingbeil, Sumit Shekhar, Matthias Trapp, and Jürgen Döllner. "ViVid: Depicting Dynamics in Stylized Live Photos". In: *ACM SIGGRAPH 2019 Appy Hour*. SIGGRAPH '19. Association for Computing Machinery, 2019. ISBN: 9781450363068. DOI: 10.1145/3305365.3329726.

[40] Giovanni Sicuranza. *Multidimensional processing of video signals*. Springer, 1992. ISBN: 978-1-4615-3616-1. DOI: 10.1007/978-1-4615-3616-1.

[41] *Splice - Video Editor & Maker*. https://apps.apple.com/us/app/splice-video-editor-maker/id409838725. [Online; accessed 2022-07-28]. Bending Spoons Apps.

[42] *SwiftUI: Declare the user interface and behavior for your app on every platform.* https://developer.apple.com/documentation/swiftui/. [Online; accessed 2022-05-30]. Apple Inc.

[43] H. Tankovska. *Most popular categories on TikTok worldwide 2020, by hashtag views.* https://www.statista.com/statistics/1130988/. [Online; accessed 2022-07-28]. 2021.

[44] S. C. L. Terra and R. A. Metoyer. "Performance Timing for Keyframe Animation". In: *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '04. Eurographics Association, 2004, pp. 253–258. ISBN: 3905673142. DOI: 10.1145/1028523.1028556.

[45] Dom Vigil. *Tom Walker Releases "Wait For You" Video*. https://preludepress.com/news/2020/06/19/tom-walker-releases-wait-for-you-video/. [Online; accessed 2022-05-14]. 2020.

[46] *Vision: Apply computer vision algorithms to perform a variety of tasks on input images and video.* https://developer.apple.com/documentation/vision. [Online; accessed 2022-05-30]. Apple Inc.

[47] *VNGeneratePersonSegmentationRequest*. https://developer.apple.com/documentation/vision/vngeneratepersonsegmentationrequest. Accessed: 2022-05-20.

[48] Tom Walker. *Tom Walker - Wait for You (Official Video)*. https://www.youtube.com/watch?v=lwGEhnl0NxQ. [Online; accessed 2022-05-14]. 2020.

[49] Tongtong Wei and Lianxiang Zhu. “Comic style transfer based on generative confrontation network”. In: *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)*. 2021, pp. 1011–1014. DOI: 10.1109/ICSP51882.2021.9408938.

[50] John W. Woods. *Multidimensional Signal, Image, and Video Processing and Coding, Second Edition*. 2nd. Academic Press, Inc., 2011. ISBN: 0123814200. DOI: 10.5555/2049940.

[51] Zhou and Chellappa. “Computation of optical flow using a neural network”. In: *IEEE 1988 International Conference on Neural Networks*. 1988, 71–78 vol.2. DOI: 10.1109/ICNN.1988.23914.

Appendix A

Developers' Interview

This interview was conducted in written form in July 2022 with bachelor students Rosmarie Debski and Ole Schmitt. At the Computer Graphics Chair of the Hasso Plattner Institute, they extended the MotionViz app and the underlying multidimensional video processing framework with novel video effects as part of the Algorithms for Processing Visual Media (APVM) seminar. Their developer experiences in dealing with the framework and app were the focus of this interview.

Q: What was your task? What kind of video effects were you aiming for?

A: We were tasked to implement a "Glow Stick" effect inspired by a TikTok video. In the reference video a person attaches glow sticks to their body, turns off the light and dances. Another effect we want to implement is a "light trails" effect. It is inspired by long exposure timelapse videos of dancers in the dark with lights attached to their wrists we found on YouTube. Both effects are implemented on the basis of joint point data extracted from RGB frames by Apples Vision framework.

Both effects had to be automated. The effects should be usable by users without any knowledge of editing videos and or creating video effects. Applying the effect should thus be easy, fast and fun with a good, visually appealing outcome.

Q: Which types of multidimensional data have you worked with?

A: We worked mainly with RGB data. From this data we extract human joint points. This is achieved by using Apples Vision framework. One could consider this skeleton or joint point data another dimension. But we didn't add this dimension into the framework, saving the data extracted, since we didn't deem it necessary to achieve our task. We just extract the data again when it is needed.

Q: What functionalities have you added to the framework?

A: So far, we added a "naive" implementation of the glow stick effect. To achieve this effect we had to implement the joint point extraction functionality as a node. We then use these joint points to create a Bezier path which represents the skeleton of the tracked human in another node which uses the skeleton detection nodes output as an input. This path is then visualized by using existing nodes.

We added the light trails effect as well. To implement it we need to access data that

is not just extracted from the current frame, but which was extracted a few frames before the current one. After familiarizing us with the framework we found a convenient way to include a buffer into the node that collects the needed data. We use the added skeleton detection node to extract the joint points. They are then buffered by the light trail node we added. The light trail node extracts the joint points which need to be buffered. With the buffered joint points and the joint points extracted from the current frame the light trail node creates a Bezier path which is then visualized using existing nodes.

We now have a naive implementation of the *LightTrails* node and effect.

Q: What nodes or effects have you implemented for this?

A: We added the following nodes and effects: A *SkeletonDetection* node, a *GlowStick* effect, a *LightTrails* node and a an associated *LightTrails* effect.

Also, we added an effect which combines both effects and visualizes a skeleton with light trails. This was achieved by recombining already existing nodes with the ones we added within a few minutes. In this effect we combined the skeleton detection, skeleton construction and *LightTrail* node with existing nodes. We, for example, used the *Twins* node to create a dancing squad of skeletons with fancy light trails.

Q: What existing functionality, nodes, or effects were you able to adopt?

A: We were able to combine a lot of nodes into our new effects. All of our new effects are based on newly added nodes, but they all use the *LineNode*, *ChalkyNode*, *GlowNode*, *BlendNode* for vizualization. The *TwinsNode* is used for one effect as well.

Q: How did you find the handling of the framework?

A: For context: We are both students at the Hasso-Plattner-Institute Potsdam in the fourth semester. We do have some programming experience through university, but haven't worked on bigger projects with legacy code before. Before programming new features in the MotionViz framework we didn't have any prior experience with Apple or the Swift programming language.

After setting up the project we needed about eight hours to implement our naive *GlowStick* effect, getting familiar with the framework and Swift included. After that, adding a new effect was much easier and the second effect, *LightTrails*, was completely implemented within four hours of coding.

Implementing a simple node is easy, if the type of data you need to create the desired outcome is already implemented and provided. For example human segmentation mask and RGB data. We weren't really able to copy code or adapt code from already existing nodes since we worked with human joint points.

Effect implementation was easy since we could adapt code from already existing effects to create a new effect. Understanding the code of a given effect is easy since every effect is defined by a combination of nodes which are "plugged" together. If the nodes are named descriptively, which they are, the flow of control is easy to understand and adapt. The ability to recombine nodes to create a new outcome was quite helpful, since

we only had to focus on calculating Bezier curves, but didn't have to worry about their visualization. This led to fast visual results. Adding user adjustable parameterization of the effects is really easy as well due to the abstraction of creating user interface elements provided by the framework.

The integration of our added functionality into the user interface was extremely easy. We didn't have to bother with user interface at all, since the provided standard sliders, toggles etc. were sufficient for our needs. The usage is easy to understand and changing the user interface for an effect is done within seconds since only a few lines of code have to be changed.

Q: Has it made your work easier and, if so, in what way?

A: We were able to develop in an environment, that gave us clear directions where our features should be located at and also gave clear boundaries which functionality is supposed to be implemented at what place. It felt like we got an already safe and steady Lego house and a box full of Lego bricks to be creative.

Author's note: *LEGO® is a trademark of the LEGO Group of companies which does not sponsor, authorize or endorse this thesis.*

Acknowledgement

My special thanks go to Sebastian Pasewaldt, who always provided challenging and motivating support during the SIGGRAPH publication as well as this thesis. Further thanks go to Maximilian Mayer, who was a major contributor to the SIGGRAPH publication. I would also like to thank the team of Digital Masterpieces GmbH, who supported this work by providing inspiration, knowledge as well as software libraries. I sincerely thank Rosmarie Debski and Ole Schmitt for sharing their experiences with the framework and the app. And last but not least, I would like to thank the dancers who made themselves available for test shots with the MotionViz app and thus also contributed significantly to the visual appearance of this work: Liv Bünger, Lilly, and Olena Vyshnevska.

Statutory Declaration

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Master Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

Potsdam, August 22, 2022
(Place, Date)


(Signature)